Sparse Dynamic Programming on DAGs with Small Width*

VELI MÄKINEN, ALEXANDRU TOMESCU, ANNA KUOSMANEN, and TOPI PAAVILAINEN, Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, FINLAND
TRAVIS GAGIE, EIT, Diego Portales University, CHILE
RAYAN CHIKHI, CNRS, CRIStAL, University of Lille, FRANCE

The *minimum path cover* problem asks us to find a minimum-cardinality set of paths that cover all the nodes of a *directed acyclic graph* (DAG). We study the case when the size k of a minimum path cover is small, that is, when the DAG has a small *width*. This case is motivated by applications in *pan-genomics*, where the genomic variation of a population is expressed as a DAG. We observe that classical alignment algorithms exploiting *sparse dynamic programming* can be extended to the sequence-against-DAG case by mimicking the algorithm for sequences on each path of a minimum path cover and handling an evaluation order anomaly with *reachability queries*.

Namely, we introduce a general framework for DAG-extensions of sparse dynamic programming. This framework produces algorithms that are slower than their counterparts on sequences only by a factor k. We illustrate this on two classical problems extended to DAGs: *longest increasing subsequence* and *longest common subsequence*. For the former, we obtain an algorithm with running time $O(k|E|\log|V|)$. This matches the optimal solution to the classical problem variant when the input sequence is modeled as a path. We obtain an analogous result for the longest common subsequence problem. We then apply this technique to the *co-linear chaining* problem, which is a generalization of both of the above two problems. The algorithm for this problem turns out to be more involved, needing further ingredients, such as an FM-index tailored for large alphabets, and a two-dimensional range search tree modified to support range maximum queries. We also study a general sequence-to-DAG alignment formulation that allows affine gap costs in the sequence.

The main ingredient of the proposed framework is a new algorithm for finding a minimum path cover of a DAG (V, E) in $O(k|E|\log|V|)$ time, improving all known time-bounds when k is small and the DAG is not too dense. In addition to boosting the sparse dynamic programming framework, an immediate consequence of this new minimum path cover algorithm is an improved space/time tradeoff for reachability queries in arbitrary directed graphs.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Packing and covering problems**; **Dynamic programming**; • **Applied computing → Computational genomics**;

Authors' addresses: Veli Mäkinen, veli.makinen@helsinki.fi; Alexandru Tomescu, alexandru.tomescu@helsinki.fi; Anna Kuosmanen, anna.kuosmanen@helsinki.fi; Topi Paavilainen, Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, P. O. Box 68, Gustaf Hällströmin katu 2b, Helsinki, 00014, FINLAND; Travis Gagie, EIT, Diego Portales University, Av. Ejército 441, Santiago, CHILE, travis.gagie@gmail.com; Rayan Chikhi, CNRS, CRIStAL, University of Lille, Bâtiment M3 extension, Avenue Carl Gauss, Villeneuve d'Ascq Cedex, 59655, FRANCE, rayan.chikhi@univ-lille.fr.

1549-6325/2019/2-ART29 $15.00
https://doi.org/10.1145/3301312

Additional Key Words and Phrases: pattern matching, longest common subsequence, co-linear chaining, pan-genomics

ACM Reference Format:
Veli Mäkinen, Alexandru Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. 2019. Sparse Dynamic Programming on DAGs with Small Width. *ACM Trans. Algor.* 15, 2, Article 29 (February 2019), 21 pages. https://doi.org/10.1145/3301312

1 INTRODUCTION

A *path cover* of a *directed acyclic graph* (DAG) $G = (V, E)$ is a set of paths such that every node of G belongs to some path (note that the paths are not required to be disjoint). A *minimum path cover* (MPC) is one having the minimum number of paths. The size of a MPC is also called the *width* of G. Many DAGs commonly used in genome research, such as graphs encoding human variations [10] and graphs modeling gene transcripts [21], can consist, in the former case, of millions of nodes and, in the latter case, of thousands of nodes (see [31] for a survey of such *pan-genomics* approaches). However, they generally have a small width on average; for example, splicing graphs for most genes in human chromosome 2 have width at most 10 [46, Fig. 7]. To the best of our knowledge, among the many MPC algorithms [8, 9, 18, 23, 36, 41], there are only three whose complexities depend on the width of the DAG. Say the width of G is k. The first algorithm runs in time $O(|V||E| + k|V|^2)$ and can be obtained by slightly modifying an algorithm for finding a minimum chain cover in partial orders from [15]. The other two algorithms are due to Chen and Chen: the first one works in time $O(|V|^2 + k\sqrt{k}|V|)$ [8], and the second one works in time $O(\max(\sqrt{|V|}|E|, k\sqrt{k}|V|))$ [9].

In this paper we present a MPC algorithm running in time $O(k|E| \log |V|)$. This is better than all previous algorithms when k is small. For example, this holds for $k = o(\sqrt{|V|}/\log |V|)$ and $|E| = O(|V|^{3/2})$ (and with smaller k, the graph can be denser). Our algorithm is based on a standard reduction of a minimum flow problem to a maximum flow problem [2].

We then proceed to show that some problems (like alignments) that admit efficient *sparse dynamic programming* solutions on sequences [13] can be extended to DAGs, so that their complexity increases only by the minimum path cover size k. Namely, our improvement applies to many cases where a data structure over previously computed solutions is maintained and queried for computing the next value. Our new MPC algorithm enables this, as its complexity is generally of the same form as that of solving the extended problems. Careful bookkeeping is necessary due to the evaluation order not matching the reachability order among the nodes of the path cover. Given a path cover, our technique first computes so-called *forward propagation links* indicating how the partial solutions in each path in the cover must be synchronized.

To best illustrate the versatility of the technique itself, we show how to compute a longest increasing subsequence (LIS) in a labeled DAG, in time $O(k|E| \log |V|)$. This matches the optimal solution to the classical problem on a single sequence when it is modeled as a path (where $k = 1$). We also illustrate our technique with the longest common subsequence (LCS) problem between a labeled DAG $G = (V, E)$ and a sequence S.

We then demonstrate the use of the framework on a more complex problem—co-linear chaining (CLC)—first introduced in [32]. It has been proposed as a model of the sequence alignment problem that scales to massive inputs, and has been a subject of recent interest (see e.g. [29, 38, 42, 47, 51–53]). In the CLC problem, the input is directly assumed to be a set of N pairs of intervals in the two sequences that match (either exactly or approximately). The CLC alignment solution asks for a subset of these plausible pairs that maximizes the coverage in one of the sequences, and whose elements appear in increasing order in both sequences. The fastest algorithm for this problem runs in the optimal $O(N \log N)$ time [1].

We define a generalization of the CLC problem between a sequence and a labeled DAG. As motivation, we mention the problem of aligning a long sequence, or even an entire chromosome, to a DAG storing all known variations of a population with respect to a reference genome (such as the above-mentioned [10]). Here, the N input pairs match intervals in the sequence with paths (also called *anchors*) in the DAG. This problem is not straightforward, as the topological order of the DAG might not follow the reachability order between the anchors. Existing tools for aligning DNA sequences to DAGs (BGREAT [27], vg [20]) rely on anchors, and our techniques have the potential to impact the design of such tools.

The algorithm we propose for the CLC extension is more involved, and we we will develop it in stages.

We conclude with a general alignment-specific formulation of the framework that admits affine gap costs, and has the LIS, LCS, and a limited variant of CLC as special cases.

Related work. Extending pattern matching to DAGs has been studied before [3, 30, 33, 37, 39]. On a graph with nodes labeled with characters, finding a path matching a pattern with a minimum number of edit operation can be done in $O(m|E|)$ time [33], where m is the pattern length and E is the set of edges. There is a matching conditional lower bound holding already for the linear case of the graph being a single path: The edit distance between two strings of length n cannot be computed in $O(n^{2-\epsilon})$ time, for any constant $\epsilon > 0$, unless the Strong Exponential Time Hypothesis (SETH) fails [4]. This quadratic boundary has been the motivation to study sparse dynamic programming solutions on sequences [13]. We extend the work on sparse dynamic programming to the case where one of the inputs is a DAG. To compare the obtained results to the non-sparse setting, let us consider the longest common subsequence problem. Being a special case of the problem studied in [33], finding the longest common subsequence of an input sequence of length n and a path in a DAG can be done in $O(n|E|)$ time. Our sparse dynamic algorithm for the same problem works in $O(k|E| \log |V| + (|V| + n) \log n + k|M| \log \log n)$ time, where M is the set of matching character pairs between the sequence and the graph. For small k and $|M|$ this can be significantly better than the non-sparse solution.

Notation. To simplify notation, for any DAG $G = (V, E)$ we will assume that V is always $\{1, \ldots, |V|\}$ and that $1, \ldots, |V|$ is a topological order on V (so that for every edge (u, v) we have $u < v$). We will also assume that $|E| \geq |V| - 1$. A *labeled DAG* is a tuple (V, E, ℓ, Σ) where (V, E) is a DAG, and $\ell : V \mapsto \Sigma$ assigns to the nodes labels from Σ, Σ being an ordered alphabet.

For a path $P = (v_1, \ldots, v_t)$ in G, let the *label* of P, denoted $\ell(P)$, be the concatenation of the labels of the nodes of P, namely $\ell(v_1) \cdots \ell(v_t)$. Moreover, the first node of P will be called its *startpoint*, and its last node will be called its *endpoint*. For a node $v \in V$, we denote by $N^-(v)$ the set of in-neighbors of v and by $N^+(v)$ the set of out-neighbors of v. If there is a (possibly empty) path from node u to node v we say that u reaches v. We denote by $R^-(v)$ the set of nodes that reach v.

We denote a set of consecutive integers with interval notation $[i..j]$, meaning $\{i, i + 1, \ldots, j\}$. For a pair of intervals $m = ([x..y], [c..d])$, we use $m.x$, $m.y$, $m.c$, and $m.d$ to denote the four respective endpoints. We also consider pairs of the form $m = (P, [c..d])$ where P is a path, and use $m.P$ to access P. For a set M we may fix an order, to access an element as $M[i]$.

In Table 1 we summarize some of these notations, together with some other notions which will be introduced later.

2 THE MPC ALGORITHM

In this section we assume basic familiarity with network flow concepts; see [2] for further details. In the *minimum flow problem*, we are given a directed graph $G = (V, E)$ with a single source and a single sink, with a *demand* $d : E \to \mathbb{Z}$ for every edge. The task is to find a flow of minimum

Notation	Definition
(V, E, ℓ, Σ)	A *labeled DAG*, where (V, E) is a DAG, and $\ell : V \mapsto \Sigma$ assigns labels from an ordered alphabet Σ to the nodes.
$\ell(P)$	For a labeled DAG (V, E, ℓ, Σ) and a path $P = (v_1, \ldots, v_t)$ in it, the *label* of P is $\ell(P) = \ell(v_1) \cdots \ell(v_t)$.
$R^-(v)$	For a DAG G and a node v, $R^-(v)$ is the set of nodes that *reach* v, namely that have a (possibly empty) path to v.
$\texttt{last2reach}[v, i]$	For a path cover $P_1, \ldots, P_K$ of a DAG, $\texttt{last2reach}[v, i]$ is the last node on P_i different from node v, that reaches v, if this node exists.
$\texttt{forward}[u]$	In a DAG, the *forward propagation links* from a node u is the set $\texttt{forward}[u]$ of pairs (v, i), such that $(v, i) \in \texttt{forward}[u]$ holds for any node v and index i with $\texttt{last2reach}[v, i] = u$.
$\texttt{index}[v, i]$	For a path cover $P_1, \ldots, P_K$, $\texttt{index}[v, i]$ is the position of v in P_i, starting from 0, or -1 if v does not appear in P_i.
$\mathcal{T}.\texttt{update}(k, \texttt{val})$	Given a search tree $\mathcal{T}$, for the leaf w with $\texttt{key}(w) = k$, update $\texttt{value}(w) = \texttt{val}$.
$\mathcal{T}.\text{RMaxQ}(l, r)$	Given a search tree $\mathcal{T}$, return $\max_{w\,:\,l \le \texttt{key}(w) \le r} \texttt{value}(w)$ (*Range Maximum Query*).

Table 1. Key notations used in this paper.

value (the *value* is the sum of the flow on the edges exiting the source) that satisfies all demands (such a flow is called *feasible*). The standard reduction from the minimum path cover problem to a minimum flow one (see, e.g. [35]) creates a new DAG G^* by replacing each node v with two nodes v^-, v^+, adds the edge (v^-, v^+) and adds all in-neighbors of v as in-neighbors of v^-, and all out-neighbors of v as out-neighbors of v^+. Finally, the reduction adds a global source with an out-going edge to every node, and a global sink with an in-coming edge from every node. Edges of type (v^-, v^+) get demand 1, and all other edges get demand 0. The value of the minimum flow equals k, the width of G, and any decomposition of it into source-to-sink paths induces a minimum path cover in G.

Our MPC algorithm is based on the following simple reduction of a minimum flow problem to a maximum flow one (see e.g. [2]): (i) find a feasible flow $f : E \to \mathbb{Z}$; (ii) transform this into a minimum feasible flow, by finding a maximum flow f' in G in which every $e \in E$ now has capacity $f(e) - d(e)$. The final minimum flow solution is obtained as $f(e) - f'(e)$, for every $e \in E$.

We solve step (i) in time $O(k|E| \log |V|)$ by finding a path cover in G^* whose size is larger than k only by a factor $O(\log |V|)$. This is based on the classical greedy set cover algorithm, see e.g. [50, Chapter 2]: at each step, select a path covering most of the remaining uncovered nodes. The following lemma shows how to do this, by dynamic programming.

Lemma 2.1. *Let $G = (V, E)$ be a DAG, and let k be the width of G. In time $O(k|E| \log |V|)$, we can compute a path cover $P_1, \ldots, P_K$ of G, such that $K = O(k \log |V|)$.*

Proof. The algorithm works by choosing, at each step, a path that covers the most uncovered nodes. For every node $v \in V$, we store $\texttt{m}[v] = 1$, if v is not covered by any path, and $\texttt{m}[v] = 0$

otherwise. We also store $\mathsf{u}[v]$ as the largest number of uncovered nodes on a path starting at v. The values $\mathsf{u}[\cdot]$ are computed by dynamic programming, by traversing the nodes in inverse topological order and setting $\mathsf{u}[v] = \mathsf{m}[v] + \max_{w \in N^+(v)} \mathsf{u}[v]$. Initially we have $\mathsf{m}[v] = 1$ for all v. We then compute $\mathsf{u}[v]$ for all v, in time $O(|E|)$. By taking the node v with the maximum $\mathsf{u}[v]$, and tracing back along the optimal path starting at v, we obtain our first path in time $O(|E|)$. We then update $\mathsf{m}[v] = 0$ for all nodes on this path, and iterate this process until all nodes are covered. This takes overall time $O(K|E|)$, where K is the number of paths found.

This algorithm analysis is identical to that of the classical greedy approximation algorithm for the set cover problem [50, Chapter 2], because the universe to be covered is V and each possible path in G is a possible covering set, which implies that $K = O(k \log |V|)$. □

This path cover induces a flow of value $O(k \log |V|)$. Thus, for step (ii) we need to shrink this flow into a flow of value k. If we run the Ford-Fulkerson algorithm, this means there are $O(k \log |V|)$ successive augmenting paths, each of which can be found in time $O(|E|)$. This gives a time bound for step (ii) of $O(k|E| \log |V|)$. Combining Lemma 2.1 with this observation, we obtain our first result:

Theorem 2.2. *Given a DAG $G = (V, E)$ of width k, the MPC problem on G can be solved in time $O(k|E| \log |V|)$.*

Our approach for Theorem 2.2 is similar to the one from [15] for finding the minimum number k of chains to cover a partial order of size n. A *chain* is a set of pairwise comparable elements. The algorithm from [15] runs in time $O(kn^2)$, and it has the same features as ours: it first finds a set of $O(k \log n)$ chains in the same way as we do (longest chains covering most uncovered elements), and then in a second step reduces these to k. However, if we were to apply this algorithm to DAGs, it would run in time $O(|V||E| + k|V|^2)$, which is slower than our algorithm for small k. This is because it uses the classical reduction given by Fulkerson [18] to a bipartite graph, where each edge of the graph encodes a pair of elements in the relation. Since DAGs are not transitive in general, to use this reduction one needs first to compute the transitive closure of the DAG, in time $O(|V||E|)$. Finally, we mention that such an approximation-refinement approach has also been applied to other covering problems on graphs, such as a 2-hop cover [11].

3 THE DYNAMIC PROGRAMMING FRAMEWORK

In this section we give an overview of our approach and introduce a few technical notions. Our aim is to single out its key features, so that it is easier to check if other problems can also benefit from it. However, we do not prove general necessary or sufficient conditions, nor general correctness results. Correctness will be proved for each application of this framework.

Suppose we have a problem involving DAGs that is solvable, for example by dynamic programming, by traversing the nodes in topological order. Thus, assume also that a partial solution at each node v is obtainable from all (and only) nodes $R^-(v)$ of the DAG that can reach v, plus some other independent objects, such as another sequence. Furthermore, suppose that at each node v we need to query (and maintain) a data structure $\mathcal{T}$ that depends on $R^-(v)$, so that the query result $\mathcal{T}_{R^-(v)}.\mathsf{Query}(\cdot)$ of the data structure at node v is obtainable from the query results $\mathcal{T}_{R^-(v_i)}.\mathsf{Query}(\cdot)$ at previous nodes v_i, where $\{v\} \cup \bigcup_i R^-(v_i) = R^-(v)$ and the sets $R^-(v_i)$ are not necessarily disjoint. We express this with the following formula:

$$\mathcal{T}_{R^-(v)}.\mathsf{Query}(\cdot) = \bigoplus_i \mathcal{T}_{R^-(v_i)}.\mathsf{Query}(\cdot). \tag{1}$$

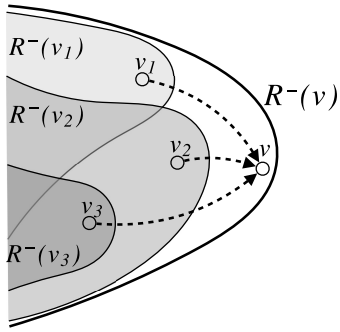

Fig. 1. A conceptual representation of the decomposition of the set $R^-(v)$ of nodes that reach v into three sets $R^-(v_1), R^-(v_2), R^-(v_3)$.

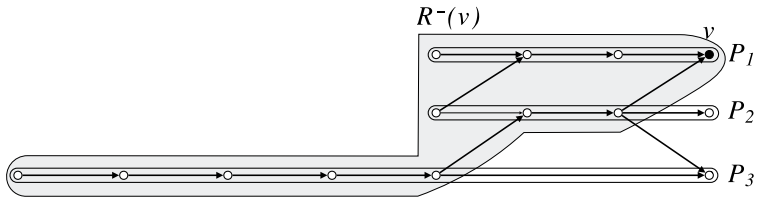

Fig. 2. A path cover P_1, P_2, P_3 of a DAG. We mark in gray the set $R^-(v)$ of nodes that reach v.

In the above, we let $\bigoplus$ be some operation on the query results, such as min or max. See Fig. 1 for a conceptual illustration.

In order to obtain such sets $R^-(v_i)$, our key idea is to decompose the graph into a path cover $P_1, \ldots, P_K$, and perform the computation only along these paths. We will employ K data structures $\mathcal{T}_1, \ldots, \mathcal{T}_K$, such that, after processing node v on path P_i, data structure $\mathcal{T}_i$ stores the correct result for $R^-(v)$. See Figures 2 and 3 for an example.

Our second idea concerns the order in which the nodes on these K paths are processed. Because the answer at v depends on $R^-(v)$, we cannot process the nodes on the K paths (and update the corresponding $\mathcal{T}_i$'s) in an arbitrary order. As such, for every path i and every node v, we distinguish the *last node* on path i different from v that reaches v (if it exists). We will call this node `last2reach`$[v, i]$. See Figure 3 for an example. We note that this insight is the same as in [24], which symmetrically identified the *first* node on a chain i that can be reached from v (a *chain* is a subsequence of a path). The following observation is the first element for using the decomposition (1).

Observation 1. *Let $P_1, \ldots, P_K$ be a path cover of a DAG $G = (V, E)$, and let $v \in V$. For every $i \in [1..K]$, if* `last2reach`$[v, i]$ *exists, let $v_i =$* `last2reach`$[v, i]$ *and let $R_i = R^-(v_i)$. Otherwise, if* `last2reach`$[v, i]$ *does not exist, let $R_i = \emptyset$. Then it holds that $R^-(v) = \{v\} \cup \bigcup_{i=1}^{K} R_i$.*

Proof. It is clear that $\{v\} \cup \bigcup_{i=1}^{K} R_i \subseteq R^-(v)$. To show the reverse inclusion, consider a node $u \in R^-(v)$. Since $P_1, \ldots, P_K$ is a path cover, then u appears on some P_i. Since u reaches v, then $u = v$, or u appears on P_i before `last2reach`$[v, i]$, or $u =$ `last2reach`$[v, i]$. Therefore $u = v$, or u appears in some R_i, as desired. □

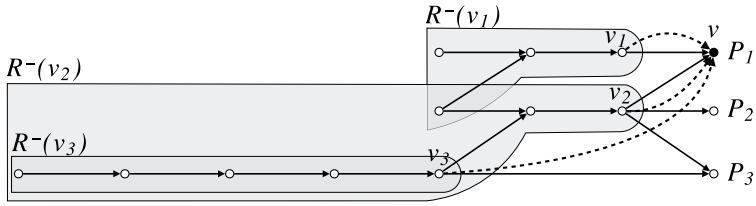

Fig. 3. The same DAG as in Fig. 2. For each path P_i, we marked the node $v_i = \texttt{last2reach}[v, i]$, namely the last node on path P_i, different from v, that reaches v. We show in gray the three sets $R^-(v_1), R^-(v_2), R^-(v_3)$ into which $R^-(v)$ is decomposed. We draw with dashed lines the forward links from each v_i to v. Note that the nodes v_i also have other outgoing forward links: see Example 4.3 for all forward links from v_3.

This allows us to identify, for every node u, a set of *forward propagation links* $\texttt{forward}[u]$, where $(v, i) \in \texttt{forward}[u]$ holds for any node v and index i with $\texttt{last2reach}[v, i] = u$ (see Fig. 3 and also Example 4.3 for some illustrations of this concept). Observe also that every node can have at most K incoming forward links, for each of the K paths of the path cover, and thus $\left|\bigcup_{u \in V} \texttt{forward}[u]\right| = O(K|V|)$.

These propagation links are the second element in the decomposition. Once we have computed the correct value at u, we update the corresponding data structures $\mathcal{T}_i$ for all paths i to which u belongs. We also propagate the query value of $\mathcal{T}_i$ in the decomposition (1) for all nodes v with $(v, i) \in \texttt{forward}[u]$. This means that when we come to process v, we have already correctly computed all terms in the decomposition (1) and it suffices to apply the operation $\bigoplus$ to these terms.

The next lemma shows how to compute the values $\texttt{last2reach}$ (and, as a consequence, all forward propagation links), also by dynamic programming.

Lemma 3.1. *Let $G = (V, E)$ be a DAG, and let $P_1, \ldots, P_K$ be a path cover of G. For every $v \in V$ and every $i \in [1..K]$, we can compute $\texttt{last2reach}[v, i]$ in overall time $O(K|E|)$.*

Proof. For each P_i and every node v on P_i, let $\texttt{index}[v, i]$ be the position of v in P_i, starting from 0. Our algorithm actually computes $\texttt{last2reach}[v, i]$ as the index of this node in P_i. Initially, we set $\texttt{last2reach}[v, i] = -1$ for all v and i. At the end of the algorithm, $\texttt{last2reach}[v, i] = -1$ will hold precisely for those nodes v that cannot be reached by any node of P_i (different than v itself).

We traverse the nodes in topological order. For each node v and every $i \in [1..K]$, we do as follows. If v is on P_i, we set $\texttt{last2reach}[v, i] = \texttt{index}[v, i]$. Otherwise, we set

$$\texttt{last2reach}[v, i] = \max_{u \in N^-(v)} \texttt{last2reach}[u, i].$$

Observe that after computing the values $\texttt{last2reach}[v, i]$ for each node v and every $i \in [1..K]$, we incorrectly have that for every path P_i on which v appears, we have $\texttt{last2reach}[v, i] = \texttt{index}[v]$. As such, we must set $\texttt{last2reach}[v, i]$ to the index of the previous node of v, namely $\texttt{last2reach}[v, i] = \texttt{index}[v] - 1$ (so that $\texttt{last2reach}[v, i]$ also gets set to -1 if v is the first node on path P_i). □

An immediate application of Theorem 2.2 and of the values $\texttt{last2reach}[v, i]$ is for solving reachability queries (see Sect. 4.1). Another simple application is an extension of the *longest increasing subsequence (LIS)* problem to labeled DAGs (see Sect. 4.2).

The LIS problem, the *longest common subsequence* (LCS) problem of Section 5, *co-linear chaining* (CLC) problem of Section 6, as well as *anchored global alignment under affine gaps costs* of Section 7 make use of the following standard data structure (see e.g. [28, p.20]).

LEMMA 3.2. *The following two operations can be supported with a balanced binary search tree $\mathcal{T}$ in time $O(\log n)$, where n is the number of leaves in the tree.*

- update$(k, \texttt{val})$*: For the leaf w with* $\texttt{key}(w) = k$*, update* $\texttt{value}(w) = \texttt{val}$.
- RMaxQ(l, r)*: Return* $\max_{w\,:\,l \le \texttt{key}(w) \le r} \texttt{value}(w)$ *(*Range Maximum Query*).*

Moreover, the balanced binary search tree can be constructed in $O(n)$ time, given the n pairs (key, value) *sorted by component* key.

In some of the solutions we can exploit a faster data structure summarized below.

LEMMA 3.3. *Consider the operations of Lemma 3.2 for keys in range $[0..n]$ so that the query range is semi-infinite $(0, r)$ and updates are only allowed to increase the value. Then there is a data structure $\mathcal{T}$ that can be constructed in $O(n \log\log n)$ time and that supports these restricted versions of operations in amortized $O(\log\log n)$ time.*

PROOF. Recall that a van Emde Boas tree [48, 49] supports maintaining a subset S of $[1..n]$ under insertions ($S = S \cup \{i\}$) and deletions ($S = S \setminus \{i\}$). It also supports operations predecessor(i) and successor(i), that give the largest element of S smaller than i, and smallest element larger than i, respectively. These operations take $O(\log\log n)$ time. There is a simple reduction to support semi-infinite range maximum queries [19]: We store the values in an auxiliary array $V[0..n]$ (with all values initialized to $-\infty$) and keep an invariant that S contains only the keys k such that $V[k]$ is a left-most answer to some semi-infinite range maximum query. This means that values $V[k]$, $k \in S$, form a strictly increasing series in $V[1..n]$. That is, query RMaxQ$(0, r)$ can be answered by $V[\text{predecessor}(r + 1)]$ (or by $-\infty$ if no such predecessor exists). To maintain this invariant, we initially set $S = \{0\}$ and update it on each operation update$(k, \texttt{val})$ as follows. If $\texttt{val} \le V[k]$ or $\texttt{val} \le V[\text{predecessor}(k)]$ we do nothing. Otherwise, we insert i to S (if it is not yet there), set $V[k] = \texttt{val}$, and delete all elements j of S succeeding k with $V[j] \le V[k]$. These deletions (each preceded by a successor-operation) can be amortized to the update-operations as each such operation causes at most one insertion. □

Throughout the paper we assume that the data structure of Lemmas 3.2 is initialized with (key, value) pairs with each $\texttt{value} = -\infty$. (The structure of Lemma 3.3 gets implicitly initialized accordingly.)

4 TWO SIMPLE APPLICATIONS

4.1 Reachability queries

Recall Theorem 2.2 and the values $\texttt{last2reach}[v, i]$. If we have all these $O(k|V|)$ values, then we can answer in constant time whether a node y is reachable from a node x (with $x \neq y$), as in [24]: we check

$$\texttt{index}[x, i] \le \texttt{index}[\texttt{last2reach}[y, i], i],$$

where $\texttt{index}[v, i]$ is defined as the position of v in P_i, starting from 0 (defined in the proof of Lemma 3.1), P_i is any path containing x, and we take by convention $\texttt{index}[-1, i] = -1$. Recall also that reachability queries in an arbitrary graph can be reduced to solving reachability queries in its DAG of strongly connected components, because nodes in the same component are pairwise reachable. See Table 2 for existing tradeoffs for solving reachability queries.[1]

[1]Note that [8] incorrectly attributes to [24] query time $O(\log k)$, and as a consequence [25, 45] incorrectly mention query time $O(\log k)$ for [8].

Construction time	Index size	Query time	Reference
$O(k\|E\|\log\|V\|)$	$O(k\|V\|)$	$O(1)$	this paper
$O(\|V\|^2 + k\sqrt{k}\|V\|)$	$O(k\|V\|)$	$O(1)$	[8]
$O(k\|E\|)$	$O(k\|V\|)$	$O(\log^2 k)$	[25]
$O(k(\|V\| + \|E\|))$	$O(k\|V\|)$	$O(k)$ or $O(\|V\| + \|E\|)$	[54]

Table 2. Previous comparable space/time tradeoffs for reachability queries. Compiled from [45, Table 1].

Corollary 4.1. *Let $G = (V, E)$ be an arbitrary directed graph and let the width of its DAG of strongly connected components be k. In time $O(k|E|\log|V|)$ we can construct from G an index of size $O(k|V|)$, so that for any $x, y \in V$ we can answer in $O(1)$ time whether y is reachable from x.*

4.2 The LIS problem

The *longest increasing subsequence* (LIS) problem asks us to delete the minimum number of values from an input sequence $s_1 \cdots s_n$ such that the remaining values form a strictly increasing series of values. Here the input sequence is assumed to come from an ordered alphabet Σ. For example, on input sequence $1, 4, 2, 3, 7, 5, 6$, from the alphabet $\Sigma = \{1, 2, 3, 4, 5, 6, 7\}$, the unique optimal solution is $1, 2, 3, 5, 6$. Such a longest increasing subsequence can be found in $O(n \log n)$ time [17]. This is optimal in the comparison model of computation, but can be improved to $O(n \log\log L)$ [12] on an integer alphabet and the RAM model of computation, where L is the length of the solution.

Let us consider an $O(n \log n)$ solution that can be easily modified to $O(n \log\log n)$ time on an integer alphabet. We first map Σ to a subset of $\{1, 2, \ldots, n\}$ with an order-preserving mapping, in $O(n \log n)$ time (by e.g. sorting the sequence elements, and relabeling by the ranks in the order of distinct values). We then store, at every index i of the input sequence, the value $\texttt{LLIS}[i]$ defined as the length of the longest strictly increasing subsequence ending at i and using the i-th symbol. The values $\texttt{LLIS}[i]$ can be computed by dynamic programming, by storing all previous key-value pairs $(s_j, \texttt{LLIS}[j])$ in the data structure $\mathcal{T}$ of Lemma 3.3, and querying $\mathcal{T}.\mathsf{RMaxQ}(0, s_i - 1)$. Notice that by definition we have $\texttt{LLIS}[j'] \leq \texttt{LLIS}[j]$ for all $j' < j$ such that $s_{j'} = s_j$. Hence, updates on $\mathcal{T}$ never try to decrease the value and condition of Lemma 3.3 holds. In fact, this feature is common to all the subsequent algorithms and we will omit mentioning this in the sequel.

Consider the following extension of the LIS problem to a labeled DAG $G = (V, E, \ell, \Sigma)$ of width k. Among all paths P in G, and among all subsequences of the label $\ell(P)$ of P, we need to find a longest strictly increasing subsequence.

We now explain how to extend the previous dynamic programming algorithm for this problem. We analogously map Σ to a subset of $\{1, 2, \ldots, |V|\}$ with an order-preserving mapping in $O(|V|\log|V|)$ time, as above. Recall that we assume $V = \{1, \ldots, |V|\}$, where $1, \ldots, |V|$ is a topological order. Assume also that we have K paths to cover V and $\texttt{forward}[u]$ is computed for all $u \in V$.

For each node v, we aim to analogously compute $\texttt{LLIS}[v]$ as the length of a longest strictly increasing subsequence of the labels of all paths ending at v, with the property that $\ell(v)$ is the last element of this subsequence.

For each $i \in [1..K]$, we let $\mathcal{T}_i$ be the data structure of Lemma 3.3. We start with setting $\mathcal{T}_i.\texttt{update}(0, 0)$ to indicate that any position can start a new increasing subsequence. For each

$v \in V$, we also initialize $\texttt{LLIS}[v] = 0$. The algorithm proceeds in the fixed topological ordering. Assume now that we are at some position u, and have already updated all search trees associated with the covering paths going through u. For every $(v, i) \in \texttt{forward}[u]$, we update $\texttt{LLIS}[v] = \max(\texttt{LLIS}[v], \mathcal{T}_i.\text{RMaxQ}(0, \ell(v) - 1) + 1)$. Once the algorithm reaches v in the topological ordering, value $\texttt{LLIS}[v]$ has been updated from all u' such that $(v, i) \in \texttt{forward}[u']$. It remains to show how to update each $\mathcal{T}_i$ when reaching v, for all covering paths i on which v occurs. This is done as $\mathcal{T}_i.\text{update}(\ell(v), \texttt{LLIS}[v])$. See Example 4.3 and Figure 4 for examples.

The final answer to the problem is $\max_{v \in V} \texttt{LLIS}[v]$, with the actual LIS to be found with a standard traceback. The algorithm runs in $O(|V| \log |V| + K|V| \log \log |V|)$ time. With Theorem 2.2 plugged in, we have $K = k$ and the total running time becomes $O(k|E| \log |V| + |V| \log |V| + k|V| \log \log |V|) = O(k|E| \log |V|)$, under our assumption $|E| \geq |V| - 1$. The following theorem summarizes this result.

Theorem 4.2. *Let $G = (V, E, \ell, \Sigma)$ be a labeled DAG of width k, where Σ is an ordered alphabet. We can find a longest increasing subsequence in G in time $O(k|E| \log |V|)$.*

When the DAG is just a labeled path with $|E| = |V| - 1$ (modeling the standard LIS problem), then the algorithm from Lemma 2.1 returns one path ($K = 1$). The complexity is then $O(|V| \log |V|)$, matching the best possible bound for the standard LIS problem [17]. For integer alphabets $\Sigma = \{1, 2, .., n\}$ the standard LIS problem can be solved in $O(n \log \log L)$ time [12], for $L \leq n$ being the length of the solution. In this setting our algorithm comes close with its $O(n \log \log n)$ bound when restricted to a single path.

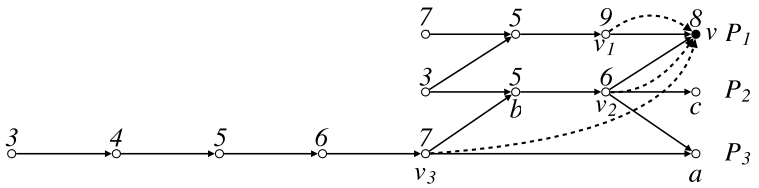

Fig. 4. Example of the LIS problem on a DAG where each node is labeled by an integer. One seeks a longest increasing sequence of a path label, among all paths of the DAG.

Example 4.3. We detail the steps needed for solving a LIS instance on the DAG from Figures 2 and 3, with node labels as shown in Fig. 4 above. As an example, observe that the forward links exiting v_3, namely the set $\texttt{forward}[v_3]$, is the set $\{(a, 3), (b, 3), (v_2, 3), (c, 3), (v, 3)\}$. We do not draw these links, but we draw instead the forward links incoming to v.

Assume now that the topological order first contains the nodes of P_3 until v_3, followed by the nodes of P_2 until v_2, and then by the nodes of P_1 until v_1.

- When we reach v_3 on P_3, we have computed $\texttt{LLIS}[v_3] = 5$ and have performed $\mathcal{T}_3.\text{update}(7, 5)$. At this point, $\mathcal{T}_3$ contains $(0, 0), (1, -\infty), (2, -\infty), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5), \ldots, (10, -\infty)$. We then perform updates along the forward links exiting v_3, including the forward link from v_3 to v. As such, we update $\texttt{LLIS}[v] = \max(\texttt{LLIS}[v], \mathcal{T}_3.\text{RMaxQ}(0, 7) + 1) = \max(0, 6) = 6$.
- When we reach v_2 on P_2, we have computed $\texttt{LLIS}[v_2] = 4$ and have performed $\mathcal{T}_2.\text{update}(6, 4)$. At this point, $\mathcal{T}_2$ contains $(0, 0), (1, -\infty), (2, -\infty), (3, 1), (4, -\infty), (5, 3), (6, 4), \ldots, (10, -\infty)$. (Keep in mind that there is also a forward link from v_3 to v_2, not drawn.) We then perform updates along the forward links exiting v_2, including the forward link from v_2 to v. As such, we update $\texttt{LLIS}[v] = \max(\texttt{LLIS}[v], \mathcal{T}_2.\text{RMaxQ}(0, 7) + 1) = \max(6, 4 + 1) = 6$.

- When we reach v_1 on P_1, we have computed $\texttt{LLIS}[v_1] = 3$ and have performed $\mathcal{T}_1.\texttt{update}(9, 3)$. At this point, $\mathcal{T}_1$ contains $(0, 0), (1, -\infty), \ldots, (5, 2), (6, -\infty), (7, 1), (8, -\infty), (9, 3), (10, -\infty)$. We then perform updates along the forward links exiting v_1, including the forward link from v_1 to v. As such, we update $\texttt{LLIS}[v] = \max(\texttt{LLIS}[v], \mathcal{T}_1.\text{RMaxQ}(0, 7) + 1) = \max(6, 3 + 1) = 6$.
- When we finally reach v, all forward links to v have been processed, and the value $\texttt{LLIS}[v] = 6$ is final and correct.

5 THE LCS PROBLEM

Consider a labeled DAG $G = (V, E, \ell, \Sigma)$ and a sequence $S \in \Sigma^*$, where Σ is an ordered alphabet. We say that the *longest common subsequence* (LCS) between G and S is a longest subsequence C of any path label in G such that C is also a subsequence of S.

We will modify the LIS algorithm of Sect. 4.2 minimally to find an LCS between a DAG G and a sequence S.

For any $c \in \Sigma$, let $S(c)$ denote set $\{j \mid S[j] = c\}$. For each node v and each $j \in S(\ell(v))$, we aim to store in $\texttt{LLCS}[v, j]$ the length of the longest common subsequence between $S[1..j]$ and any label of a path ending at v, among all subsequences having $\ell(v) = S[j]$ as the last symbol.

Assume we have a path cover of size K and $\texttt{forward}[u]$ computed for all $u \in V$. Assume also we have mapped Σ to $\{0, 1, 2, \ldots, |S| + 1\}$ in $O((|V| + |S|) \log |S|)$ time (e.g. by sorting the symbols of S, binary searching labels of V, and then relabeling by ranks, with the exception that, if a node label does not appear in S, it is replaced by $|S| + 1$).

Let $\mathcal{T}_i$ be a data structure of Lemma 3.3. We start by setting $\mathcal{T}_i.\texttt{update}(0, 0)$, for each $i \in [1..K]$, so that any (v, j) with $\ell(v) = S[j]$ can start a new common subsequence. We also initialize $\texttt{LLCS}[v, j] = 0$ for all (v, j). The algorithm proceeds in fixed topological ordering on G. At a node u, for every $(v, i) \in \texttt{forward}[u]$ we now update an array $\texttt{LLCS}[v, j]$ for all $j \in S(\ell(v))$ as follows: $\texttt{LLCS}[v, j] = \max(\texttt{LLCS}[v, j], \mathcal{T}_i.\text{RMaxQ}(0, j - 1) + 1)$. The update step of $\mathcal{T}_i$ when the algorithm reaches a node v, for each covering path i containing v, is done as $\mathcal{T}_i.\texttt{update}(j', \texttt{LLCS}[v, j'])$ for all j' such that $j' \in S(\ell(v))$.

The final answer to the problem is $\max_{v \in V, j \in S(\ell(v))} \texttt{LLCS}[v, j]$, with the actual LCS to be found with a standard traceback. The algorithm runs in $O((|V| + |S|) \log |S| + K|M| \log \log |S|)$ time, where $M = \{(v, j) \mid v \in V, j \in [1..|S|], \ell(v) = S[j]\}$, and assuming a cover of K paths is given. Notice that $|M|$ can be $\Omega(|V||S|)$. With Theorem 2.2 plugged in, the total running time becomes $O(k|E| \log |V| + (|V| + |S|) \log |S| + k|M| \log \log |S|)$.

Theorem 5.1. *Let $G = (V, E, \ell, \Sigma)$ be a labeled DAG of width k, and let $S \in \Sigma^*$, where Σ is an ordered alphabet. We can find a longest common subsequence between G and S in time $O(k|E| \log |V| + (|V| + |S|) \log |S| + k|M| \log \log |S|)$, where M is the set of characters in the sequences that match, namely $M = \{(v, j) \mid v \in V, j \in [1..|S|], \ell(v) = S[j]\}$.*

When G is a path, the bound improves to $O((|V| + |S|) \log |S| + |M| \log \log |S|)$, which nearly matches the fastest sparse dynamic programming algorithm for the LCS on two sequences [13]. On two sequences of length n and m, respectively, the algorithm of [13] runs in time $O(n \log(\min(m, |\Sigma|) + d \log \log(\min(d, nm/d)))$, where $d \leq |M|$ is the number of so-called dominant matches, which are a subset of the $|M|$ total pairs of characters in the two sequences that match. Notice that our algorithm has a difference in the log log-factor due to a different data structure, which does not work for this order of computation.

6 CO-LINEAR CHAINING

We start with a formal definition of the co-linear chaining problem (see Figure 5 for an illustration), following the notions introduced in [28, Section 15.4].

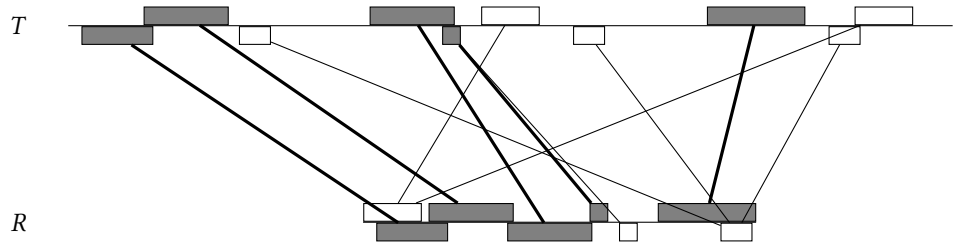

Fig. 5. In the co-linear chaining problem between two sequences T and R, we need to find a subset of pairs of intervals (i.e., anchors) so that (i) the selected intervals in each sequence appear in increasing order; and (ii) the selected intervals cover in R the maximum number of positions. The figure shows an input for the problem, and highlights in gray an optimal subset of anchors. Figure taken from [28].

Problem 1 (Co-linear chaining (CLC)). *Let T and R be two sequences over an alphabet Σ, and let M be a set of N pairs $([x..y],[c..d])$. Find an ordered subset $S = s_1 s_2 \cdots s_p$ of pairs from M such that*

- *$s_{j-1}.y < s_j.y$ and $s_{j-1}.d < s_j.d$, for all $1 \le j \le p$, and*
- *S maximizes the* ordered *coverage of R, defined as*

$$\texttt{coverage}(R,S) = |\{i \in [1..|R|] \mid i \in [s_j.c..s_j.d] \text{ for some } 1 \le j \le p\}|.$$

The definition of ordered coverage between two sequences is symmetric, as we can simply exchange the roles of T and R. But when solving the CLC problem between a DAG and a sequence, we must choose whether we want to maximize the ordered coverage on the sequence R or on the DAG G. We will consider the former variant.

First, we define the following *precedence relation*:

Definition 6.1. Given two paths P_1 and P_2 in a DAG G, we say that P_1 *precedes* P_2, and write $P_1 \prec P_2$, if one of the following conditions holds:

- P_1 and P_2 do not share nodes and there is a path in G from the endpoint of P_1 to the startpoint of P_2, or
- P_1 and P_2 have a suffix-prefix overlap and P_2 is not fully contained in P_1; that is, if $P_1 = (a_1,\ldots,a_i)$ and $P_2 = (b_1,\ldots,b_j)$ then there exists a $k \in \{\max(1, 2+i-j),\ldots,i\}$ such that $a_k = b_1$, $a_{k+1} = b_2$, …, $a_i = b_{1+i-k}$.

We then extend the formulation of Problem 1 to handle a sequence and a DAG (see Figure 6 for an illustration).

Problem 2 (CLC between a sequence and a DAG). *Let R be a sequence, let G be a labeled DAG, and let M be a set of N pairs of the form $(P,[c..d])$, where P is a path in G and $c \le d$ are non-negative integers (with the interpretation that the path label $\ell(P)$ matches the substring $R[c..d]$). Find an ordered subset $S = s_1 s_2 \cdots s_p$ of pairs from M such that*

- *for all $2 \le j \le p$, it holds that $s_{j-1}.P \prec s_j.P$ and $s_{j-1}.d < s_j.d$, and*
- *S maximizes the* ordered *coverage of R, analogously defined as* $\texttt{coverage}(R,S) = |\{i \in [1..|R|] \mid i \in [s_j.c..s_j.d] \text{ for some } 1 \le j \le p\}|$.

To illustrate the main technique of this paper, let us for now only seek solutions where paths in consecutive pairs in a solution do not overlap in the DAG. Suffix-prefix overlaps between paths turn out to be challenging; we will postpone this case until Sect. 6.3.

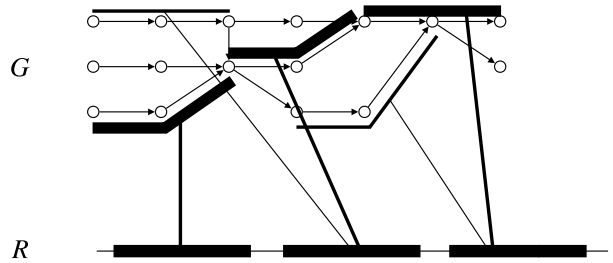

Fig. 6. An illustration of the CLC problem between a sequence and a DAG. We show three substrings of R and their matches in G, and highlight with thick lines an optimal subset of pairs.

Problem 3 (Overlap-limited CLC between a sequence and a DAG). *Under the same hypotheses as in Problem 2, find an ordered subset $S = s_1 s_2 \cdots s_p$ of pairs from M such that*

- *for all $2 \le j \le p$, it holds that there is a non-empty path from the last node of $s_{j-1}.P$ to the first node of $s_j.P$ and $s_{j-1}.d < s_j.d$, and*
- *S maximizes* $\mathsf{coverage}(R, S)$.

First, let us consider a trivial approach to solve Problem 3. Assume we have ordered in $O(|E| + N)$ time the N input pairs as $M[1], M[2], \ldots, M[N]$, so that the endpoints of $M[1].P, M[2].P, \ldots, M[N].P$ are in topological order, breaking ties arbitrarily. We denote by $C[j]$ the maximum ordered coverage of $R[1..M[j].d]$ using the pair $M[j]$ and any subset of pairs from $\{M[1], M[2], \ldots, M[j-1]\}$.

Theorem 6.2. *Overlap-limited co-linear chaining between a sequence and a labeled DAG $G = (V, E, \ell, \Sigma)$ (Problem 3) on N input pairs can be solved in $O((|V| + |E|)N)$ time.*

Proof. First, we reverse the edges of G. Then we mark the nodes that correspond to the path endpoints for every pair. After this preprocessing we can start computing the maximum ordered coverage for the pairs as follows: for every pair $M[j]$ in topological order of their path endpoints for $j \in \{1, \ldots, N\}$ we do a depth-first traversal starting at the startpoint of path $M[j].P$. Note that since the edges are reversed, the depth-first traversal checks only pairs whose paths are predecessors of $M[j].P$.

Whenever we encounter a node that corresponds to the path endpoint of a pair $M[j']$, we first examine whether it fulfills the criterion $M[j'].d < M[j].c$ (call this case (a)). The best ordered coverage using pair $M[j]$ after all such $M[j']$ is then

$$C^{\mathrm{a}}[j] = \max_{j' \,:\, M[j'].d < M[j].c} \{C[j'] + (M[j].d - M[j].c + 1)\}, \tag{2}$$

where $C[j]'$ is the best ordered coverage when using pairs $M[j']$ last.

If pair $M[j']$ does not fulfill the criterion for case (a), we then check whether $M[j].c \le M[j'].d \le M[j].d$ (call this case (b)). The best ordered coverage using pair $M[j]$ after all such $M[j']$ with $M[j'].c < M[j].c$ is then

$$C^{\mathrm{b}}[j] = \max_{j' \,:\, M[j].c \le M[j'].d \le M[j].d} \{C[j'] + (M[j].d - M[j'].d)\}. \tag{3}$$

Inclusions, i.e. $M[j].c \le M[j'].c$, can be left computed incorrectly in $C^{\mathrm{b}}[j]$, since there is a better or equally good solution computed in $C^{\mathrm{a}}[j]$ or $C^{\mathrm{b}}[j]$ that does not use them [1].

Finally, we take $C[j] = \max(C^{\mathrm{a}}[j], C^{\mathrm{b}}[j])$. Depth-first traversal takes $O(|V| + |E|)$ time and is executed N times, for $O((|V| + |E|)N)$ total time. □

However, we can do significantly better than $O((|V| + |E|)N)$ time. In the next sections we will describe how to apply the framework from Section 3 here.

6.1 Co-linear chaining on sequences revisited

We now describe the dynamic programming algorithm from [1] for the case of two sequences, as we will then reuse this same algorithm in our MPC approach.

First, sort input pairs in M by the coordinate y into the sequence $M[1], M[2], \ldots, M[N]$, so that $M[i].y \leq M[j].y$ holds for all $i < j$. This will ensure that we consider the overlapping ranges in sequence T in the correct order. Then, we fill a table $C[1..N]$ analogous to that of Theorem 6.2 so that $C[j]$ gives the maximum ordered coverage of $R[1..M[j].d]$ using the pair $M[j]$ and any subset of pairs from $\{M[1], M[2], \ldots, M[j-1]\}$. Hence, $\max_j C[j]$ gives the total maximum ordered coverage of R.

Consider Equations (2) and (3). Now we can use an *invariant technique* to convert these recurrence relations so that we can exploit the range maximum queries of Lemma 3.2:

$$\begin{aligned}
C^{\mathrm{a}}[j] &= (M[j].d - M[j].c + 1) + \max_{j' \,:\, M[j'].d < M[j].c} C[j'] \\
&= (M[j].d - M[j].c + 1) + \mathcal{T}.\mathsf{RMaxQ}(0, M[j].c - 1), \\
C^{\mathrm{b}}[j] &= M[j].d + \max_{j' \,:\, M[j].c \leq M[j'].d \leq M[j].d} \{C[j'] - M[j'].d\} \\
&= M[j].d + \mathcal{I}.\mathsf{RMaxQ}(M[j].c, M[j].d), \\
C[j] &= \max(C^{\mathrm{a}}[j], C^{\mathrm{b}}[j]).
\end{aligned}$$

For these to work correctly, we need to have properly updated the trees $\mathcal{T}$ and $\mathcal{I}$ for all $j' \in [1..j-1]$. That is, we need to call $\mathcal{T}.\mathsf{update}(M[j'].d, C[j'])$ and $\mathcal{I}.\mathsf{update}(M[j'].d, C[j'] - M[j'].d)$ after computing each $C[j']$. The running time is $O(N \log N)$.

Figure 5 illustrates the optimal chain on our schematic example. This chain can be extracted by modifying the algorithm to store traceback pointers.

Theorem 6.3 ([1, 42]). *Problem 1 on N input pairs can be solved in the optimal $O(N \log N)$ time.*

6.2 Co-linear chaining on DAGs using a minimum path cover

Let us now modify the above algorithm to work with DAGs, using the main technique of this paper.

Theorem 6.4. *Problem 3 on a labeled DAG $G = (V, E, \ell, \Sigma)$ of width k and a set of N input pairs can be solved in time $O(k|E| \log |V| + kN \log N)$ time.*

Proof. Assume we have a path cover of size K and $\mathtt{forward}[u]$ computed for all $u \in V$. For each path $i \in [1..K]$, we create two binary search trees $\mathcal{T}_i$ and $\mathcal{I}_i$. As a reminder, these trees correspond to coverages for pairs that do not, and do overlap, respectively, on the sequence. Moreover, recall that in Problem 3 we do not consider solutions where consecutive paths in the graph overlap.

As keys, we use $M[j].d$, for every pair $M[j]$, and additionally the key 0. Recall that the value of every key is initialized to $-\infty$.

After these preprocessing steps, we process the nodes in topological order, as detailed in Algorithm 1. If node v corresponds to the endpoint of some $M[j].P$, we update the trees $\mathcal{T}_i$ and $\mathcal{I}_i$ for all covering paths i containing node v. Then we follow all forward propagation links $(w, i) \in \mathtt{forward}[v]$ and update $C[j]$ for each path $M[j].P$ starting at w, taking into account all pairs whose path endpoints are in covering path i. Before the main loop visits w, we have processed all forward propagation links to w, and the computation of $C[j]$ has taken all previous pairs into account, as in the naive algorithm, but now indirectly through the K search trees. Exceptions are the pairs overlapping in the graph, which we omit in this problem statement. The

ALGORITHM 1: Co-linear chaining between a sequence and a DAG using a path cover.

Input: DAG $G = (V, E)$, a path cover $P_1, P_2, \ldots, P_K$ of G, and N pairs $M[1], M[2], \ldots, M[N]$ of the form $(P, [c..d])$.
Output: The index j giving $\max_j C[j]$.

```
Use Lemma 3.1 to find all forward propagation links;
for i ← 1 to K do
    Initialize search trees T_i and I_i with keys M[j].d, 1 ≤ j ≤ N, and with key 0, all keys associated with
      values −∞;
    T_i.update(0, 0);
    I_i.update(0, 0);
/* Save to start[i] (respectively, end[i]) the indexes of all pairs whose path starts
   (respectively, ends) at i.                                                        */
for j ← 1 to N do
    start[M[j].P.first].push(j);
    end[M[j].P.last].push(j);
    C[j] ← 0;
for v ∈ V in topological order do
    for j ∈ end[v] do
        /* Update the search trees for every path that covers v, stored in paths[v].   */
        for i ∈ paths[v] do
            T_i.update(M[j].d, C[j]);
            I_i.update(M[j].d, C[j] − M[j].d);
    for (w, i) ∈ forward[v] do
        for j ∈ start[w] do
            C^a[j] ← (M[j].d − M[j].c + 1) + T_i.RMaxQ(0, M[j].c − 1);
            C^b[j] ← M[j].d + I_i.RMaxQ(M[j].c, M[j].d);
            C[j] ← max(C[j], C^a[j], C^b[j]);
return argmax_j C[j];
```

forward propagation ensures that the search tree query results are indeed taking only reachable pairs into account. While $C[j]$ is already computed when visiting w, the startpoint of $M[j].P$, the added coverage with the pair is updated to the search trees only when visiting the endpoint.

There are NK forward propagation links, and both search trees are queried in $O(\log N)$ time. All the search trees containing a path endpoint of a pair are updated. Each endpoint can be contained in at most K paths, so this also gives the same bound $2NK$ on the number of updates. With Theorem 2.2 plugged in, we have $K = k$ and the total running time becomes $O(k|E| \log |V| + kN \log N)$. □

6.3 Co-linear chaining with path overlaps

We now consider how to extend the algorithms we developed for Problem 3 to work for the more general case of Problem 2, where overlaps between paths are allowed in a solution. The detection and merging of such path overlaps has been studied in [40], and we tailor a similar approach for our purposes.

We use an *FM-index* [16] tailored for large alphabets [22], and a two-dimensional range search tree [7] modified to support range maximum queries. The former is used for obtaining all ranges $[i'..i]$ in the coverage array C such that all input pairs $M[i'], \ldots, M[i]$ have a path $M[i''].P$, $i' \leq i'' \leq i$,

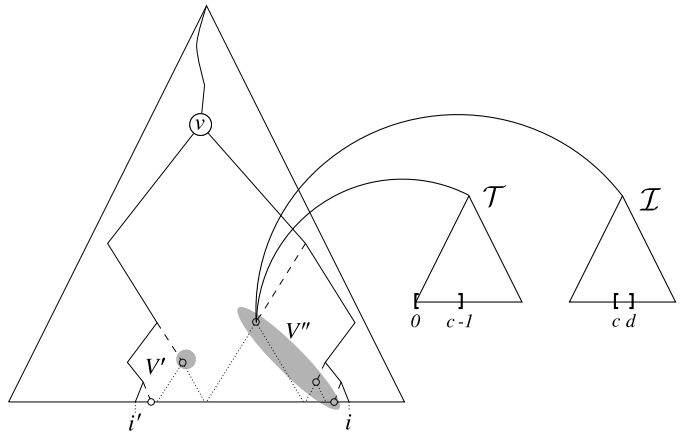

Fig. 7. Answering two-dimensional range maximum queries. A search interval $[i'..i]$ is split into $O(\log(i-i'+1))$ intervals / subtrees. In each subtree root we store a type $\mathcal{T}$ and a type $\mathcal{I}$ search tree of Lemma 3.2 containing values $C[i'']$ or $C[i''] - M[i''].d$, respectively, for indexes i'' in the corresponding subinterval. Figure modified from one in [28].

overlapping with the path $M[j].P$ of j-th input pair $M[j]$. Here, the endpoint of the j-th input pair is at node v visited in topological order. This implies that the paths of the input pairs $[i'..i]$ have already been visited, and thus, by induction, that $C[i'..i]$ values have been correctly computed (subject to the modification we are about to study). The sequence ranges $[M[i''].c..M[i''].d]$ for all $i'' \in [i'..i]$ may be arbitrarily located with respect to interval $[M[j].c..M[j].d]$, so we need to maintain an analogous mechanism with search trees of type $\mathcal{T}$ and $\mathcal{I}$ as in our co-linear chaining algorithm based on a path cover. This time we cannot, in advance, separate the input pairs to K paths with different search trees, but we have a dynamic setting, with interval $[i'..i]$ deciding which values should be taken into account. This is where a two-dimensional range search tree is used to support these queries in $O(\log^2 N)$ time: Figure 7 illustrates this.

In what follows we show that $O(L)$ queries are sufficient to take all overlaps into account throughout the algorithm execution (holding for both the trivial algorithm and for the one based on a path cover), where $L = \sum_i |M[i].P|$—the sum of the path lengths—is at most the total input length. The construction will actually induce an order for the input pairs such that $O(L)$ queries are sufficient: Since the other parts of the algorithms do not use the order of input pairs directly, we can safely reorganize the input accordingly.

With this introduction, we are ready to consider how all the intervals $[i'..i]$ related to j-th pair are obtained. We build in $O(L \log\log|V|)$ time the FM-index version proposed in [22] of sequences $T = (\prod_i \#(M[i].P)^{-1})\#$, where # is a symbol not in alphabet $\{1, 2, \ldots, |V|\}$ and considered smaller than other symbols, e.g. # = 0, and X^{-1} denotes the reverse $X[|X|]X[|X-1]] \cdots X[1]$ of X.

For our purposes it is sufficient to know that the FM-index of T, when given an interval $I(X)$ corresponding to lexicographically-ordered suffixes that start with X, can determine the interval $I(cX)$ in $O(\log\log|V|)$ time [22]. This operation is called *backward step*.

We use the index to search $M[j].P$ in the forward direction by searching its reverse with backward steps. Consider we have found interval $I((M[j].P[1..k])^{-1})$, for some k, such that backward step $I(\#(M[j].P[1..k])^{-1})$ results in a non-empty interval $[i'..i]$. This interval $[i'..i]$ corresponds to all suffixes of T that have $\#(M[j].P[1..k])^{-1}$ as a prefix. That is, $[i'..i]$ corresponds to input pairs whose path suffix has a length k overlap with the path prefix of j-th input pair. For this interval to match

with coverage array C, we just need to rearrange the input pairs according to their order in the first N rows of the array storing the lexicographic order of suffixes of T.

Since each backward step on the index may induce a range search on exactly one interval $[i'..i]$, the running time is dominated by the range queries.[2] On the other hand, this also gives the bound L on the number of range queries, as claimed earlier.

Alternatively, one can omit the expensive range queries and process each overlapping pair separately, to compute in constant time its contribution to $C[j]$. This gives another bound $O(L \log\log |V| + \texttt{\#overlaps})$, where `#overlaps` is the number of overlaps between the input paths. This can be improved to $O(L + \texttt{\#overlaps})$ by using a generalized suffix tree to compute the overlaps in advance [40, proof of Theorem 2]. We obtain the following result:

Theorem 6.5. *Let $G = (V, E, \ell, \Sigma)$ be a labeled DAG and let M be a set of N pairs of the form $(P, [c..d])$. The algorithms from Theorems 6.2 and 6.4 can be modified to solve Problem 2 with additional time $O(L \log^2 |V|)$ or $O(L + \texttt{\#overlaps})$, where L is at most the input length and `#overlaps` is the number of overlaps between the input paths.*

7 GENERAL ALIGNMENT-SPECIFIC FRAMEWORK

We will now express the framework of Section 3 in the light of alignments, removing special features of CLC, but adding different gap costs to the model, as in [13, 14].

We assume the reader is familiar with the concept of alignments (see e.g. [28, Section 6]). Consider an input sequence $R[1..m]$, a labeled input DAG $G = (V = [1..n], E, \ell, \Sigma)$, and a sparse set $A \subseteq [1..m] \times [1..n]$ of alignment anchors. We wish to compute the global alignment score $S(m+1, n+1)$ of the best alignment of R with a source-to-sink path of G among alignments whose substitutions are a subset of A. We call such alignment *anchored*. With proper initialization one can compute the score using recurrence

$$S(j, v) = s(j, v) + \max_{j' < j, u \in R^-(v), (j', u) \in A} \{S(j', u) + w(j', j, u, v)\}, \tag{4}$$

where $s(j, v)$ is the substitution score of aligning $R[j]$ with $\ell(v)$, w is the gap cost function associated to $P[j' + 1..j - 1]$ and to the shortest path from u to v.

Our general framework applies at least when the gap cost is defined as a linear function of the gap length in R, that is, under the *affine* gap cost model. For α and β constants, consider the following cost function:

$$w(j', j, u, v) = \begin{cases} 0, & \text{for } j' = j - 1, \text{ and} \\ -\alpha - \beta(j - j' - 1), & \text{for } j' < j - 1. \end{cases}$$

Then one can write Eq. (4) as

$$S(j, v) = \max \begin{cases} s(j, v) + \max\limits_{u \in R^-(v), (j-1, u) \in A} \{S(j - 1, u)\}, \\ s(j, v) - \alpha - \beta j + \max\limits_{j' < j, u \in R^-(v), (j', u) \in A} \{S(j', u) + \beta j'\}, \end{cases} \tag{5}$$

where the first part handles the case of two consecutive substitutions in the sequence and the second part the case of a gap between two substitution in the sequence. Here we used the same invariant trick as in co-linear chaining.

The rest of the details are analogous to the LIS, LCS, and CLC algorithms, except that the first part of Eq. (5) needs a different approach. Assume we have a path cover of G of size K and `forward`$[u]$

[2] A simple wavelet tree based FM-index would provide the same bound, but in case the range search part is later improved, we used the best bound for the subroutine.

computed for all $u \in V$. Let $\mathcal{T}_i$ be a data structure of Lemma 3.3 initialized with $\mathcal{T}_i$.update$(0,0)$, for each $i \in [1..K]$. These will be used for handling the second part of Eq. (5). Let $M_i[1..m]$ be an array initialized with values $-\infty$, for each $i \in [1..K]$. These will be used for handling the first part of Eq. (5). The algorithm proceeds in a fixed topological ordering on G. At a node u, for every $(v,i) \in$ forward$[u]$ we now update $S(j,v)$ for all j such that $(j,v) \in A$ as follows:

$$S(j,v) = \max \begin{cases} S(j,v), \\ s(j,v) + M_i[j-1], \\ s(j,v) - \alpha - \beta j + \mathcal{T}_i.\mathsf{RMaxQ}(0, j-1). \end{cases}$$

The update step of $\mathcal{T}_i$ when the algorithm reaches a node v, for each covering path i containing v, is done as $\mathcal{T}_i$.update$(j', S(j',v) + \beta j')$ for all j' such that $(j',v) \in A$. We also update $M_i[j'] = \max(M_i[j'], S(j',v))$ for all j' such that $(j',v) \in A$. Initialization is handled by the $(0,0)$ key-value at each $\mathcal{T}_i$ so that an alignment can start with a gap, and with values $-\infty$ in each M_i so that the first anchored substitution is handled properly.

Combined with the path cover algorithm, this global alignment generalization to the sequence-to-DAG case can thus be solved in the same running time as the LCS problem, with the match set M replaced now by the given set of anchors A.

Theorem 7.1. *Let $G = (V, E, \ell, \Sigma)$ be a labeled DAG of width k, let $R \in \Sigma^*$, and A be a set of alignment anchors. We can compute the maximum anchored global alignment score of the sequence R and a source-to-sink path in G in time $O(k|E| \log |V| + (|V| + |R|) \log |R| + k|A| \log\log |R|)$, under affine gap costs in R.*

8 DISCUSSION

For applying our solutions to Problem 2 in practice, one first needs to find the alignment anchors. As explained in the problem formulation, alignment anchors are such pairs $(P, [c..d])$ where P is a path in G and $\ell(P)$ matches $R[c..d]$. With sequence inputs, such pairs are usually taken to be *maximal exact matches* (MEMs) between the two sequences and can be retrieved in small space in linear time [5, 6]. It is largely an open problem how to retrieve MEMs between a sequence and a DAG efficiently: The case of length-limited MEMs is studied in [43], based on an extension of [44] with features such as suffix tree functionality. On the practical side, anchor finding has already been incorporated into tools for conducting alignment of a sequence to a DAG [20, 27, 34]. We also implemented the co-linear chaining algorithm[3], and reported some preliminary experimental results in the conference version of this article [26].

On the theoretical side, it remains open whether the MPC algorithm could benefit from a better initial approximation and/or one that is faster to compute. More generally, it remains open whether the overall bound $O(k|E| \log |V|)$ for the MPC problem can be improved. Also, many of the non-sparse solutions to pattern matching on graphs work also on general graphs (see e.g. [33]), while our solutions are restricted to DAGs. NP-hardness of path cover on general graphs is the main bottleneck, but even if a path cover is given, it is not clear how to extend the sparse dynamic programming framework to handle cycles.

In addition to the case of cycles, our dynamic programming framework could be extended to various gap cost models [13, 14]. We demonstrated that the framework applies to the case of affine (linear) gap costs in the input sequence, but it remains open how to handle more complex costs functions, and gap costs in the graph.

Finally, we have studied a sparse dynamic programming framework for sequence-to-DAG alignment problems, but it would also be natural to consider DAG-to-DAG alignment. Finding a path

[3]https://github.com/Anna-Kuosmanen/Seq2DagChainer

P^A from DAG $A = (V^A, E^A)$ and path P^B from DAG $B = (V^B, E^B)$ that minimizes the edit distance between $\ell(P^A)$ and $\ell(P^B)$ can easily be done in $O(|E^A||E^B|)$ time [28, Section 6.6.5]. Extending the sparse dynamic programming framework to this problem area is an interesting direction to consider.

ACKNOWLEDGMENTS

We thank Djamal Belazzougui for pointers to backward step on large alphabets, Gonzalo Navarro for pointing out the connection to pattern matching on hypertexts, and the anonymous reviewers for suggestions that improved the presentation and helped to put the contributions into a larger context.

This work was funded in part by the Academy of Finland (grant 274977 to AIT and grants 284598 and 309048 to AK and to VM), by Futurice Oy (to TP), and by Fondecyt grant 1171058 (to TG).

REFERENCES

[1] Mohamed Ibrahim Abouelhoda. 2007. A Chaining Algorithm for Mapping cDNA Sequences to Multiple Genomic Sequences. In *14th International Symposium on String Processing and Information Retrieval (LNCS)*, Vol. 4726. Springer, 1–13.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[3] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. 2000. Pattern Matching in Hypertext. *J. Algorithms* 35, 1 (2000), 82–99.

[4] Arturs Backurs and Piotr Indyk. 2015. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC '15)*. ACM, 51–58.

[5] Djamal Belazzougui. 2014. Linear time construction of compressed text indices in compact space. In *Proc. Symposium on Theory of Computing STOC 2014*. ACM, 148–193. http://doi.acm.org/10.1145/2591796.2591885.

[6] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. 2013. Versatile Succinct Representations of the Bidirectional Burrows-Wheeler Transform. In *Proc. 21st Annual European Symposium on Algorithms (ESA 2013) (LNCS)*, Vol. 8125. Springer, 133–144. http://dx.doi.org/10.1007/978-3-642-40450-4_12.

[7] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[8] Y. Chen and Y. Chen. 2008. An Efficient Algorithm for Answering Graph Reachability Queries. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 893–902.

[9] Y. Chen and Y. Chen. 2014. On the Graph Decomposition. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. IEEE, 777–784.

[10] Deanna M Church, Valerie A Schneider, Karyn Meltz Steinberg, Michael C Schatz, Aaron R Quinlan, Chen-Shan Chin, Paul A Kitts, Bronwen Aken, Gabor T Marth, Michael M Hoffman, et al. 2015. Extending reference assembly models. *Genome biology* 16, 1 (2015), 13.

[11] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355. http://dx.doi.org/10.1137/S0097539702403098.

[12] Maxime Crochemore and Ely Porat. 2010. Fast computation of a longest increasing subsequence and application. *Information and Computation* 208, 9 (2010), 1054 – 1059.

[13] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. 1992. Sparse Dynamic Programming I: Linear Cost Functions. *J. ACM* 39, 3 (July 1992), 519–545. http://doi.acm.org/10.1145/146637.146650.

[14] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. 1992. Sparse Dynamic Programming II: Convex and Concave Cost Functions. *J. ACM* 39, 3 (1992), 546–567.

[15] Stefan Felsner, Vijay Raghavan, and Jeremy Spinrad. 2003. Recognition Algorithms for Orders of Small Width and Graphs of Small Dilworth Number. *Order* 20, 4 (Nov 2003), 351–364. https://doi.org/10.1023/B:ORDE.0000034609.99940.fb.

[16] Paolo Ferragina and Giovanni Manzini. 2005. Indexing Compressed Text. *J. ACM* 52, 4 (July 2005), 552–581. http://doi.acm.org/10.1145/1082036.1082039.

[17] Michael L Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29–35.

[18] D. R. Fulkerson. 1956. Note on Dilworth's decomposition theorem for partially ordered sets. *Proc. Amer. Math. Soc.* 7, 4 (1956), 701–702.

[19] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC '84)*. ACM, New York, NY, USA, 135–143. http://doi.acm.org/10.1145/800057.808675.

[20] Garrison Erik, Sirén Jouni, Novak Adam M, Hickey Glenn, Eizenga Jordan M, Dawson Eric T, Jones William, Garg Shilpa, Markello Charles, Lin Michael F, Paten Benedict, and Durbin Richard. 2018. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology* 36 (aug 2018), 875.

[21] Steffen Heber, Max Alekseyev, Sing-Hoi Sze, Haixu Tang, and Pavel A. Pevzner. 2002. Splicing graphs and EST assembly problem. *Bioinformatics* 18 Suppl 1 (2002), S181–S188.

[22] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. 2009. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM J. Comput.* 38, 6 (2009), 2162–2178. http://dx.doi.org/10.1137/070685373.

[23] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231.

[24] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Trans. Database Syst.* 15, 4 (Dec. 1990), 558–598. http://doi.acm.org/10.1145/99935.99944.

[25] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs. *ACM Trans. Database Syst.* 36, 1 (March 2011), 7:1–7:44. http://doi.acm.org/10.1145/1929934.1929941.

[26] Anna Kuosmanen, Topi Paavilainen, Travis Gagie, Rayan Chikhi, Alexandru I. Tomescu, and Veli Mäkinen. 2018. Using Minimum Path Cover to Boost Dynamic Programming on DAGs: Co-linear Chaining Extended. In *Proc. 22nd Annual International Conference on Research in Computational Molecular Biology (RECOMB 2018) (Lecture Notes in Computer Science)*, Vol. 10812. Springer, 105–121.

[27] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. 2016. Read mapping on de Bruijn graphs. *BMC bioinformatics* 17, 1 (2016), 237.

[28] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-Scale Algorithm Design.* Cambridge University Press.

[29] Veli Mäkinen, Leena Salmela, and Johannes Ylinen. 2012. Normalized N50 assembly metric using gap-restricted co-linear chaining. *BMC Bioinformatics* 13 (2012), 255. https://doi.org/10.1186/1471-2105-13-255.

[30] U. Manber and S. Wu. 1992. Approximate string matching with arbitrary costs for text and hypertext. In *IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland.* 22–33.

[31] Tobias Marschall et al. 2018. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics* 19, 1 (2018), 118–135.

[32] Gene Myers and Webb Miller. 1995. Chaining Multiple-Alignment Fragments in Sub-Quadratic Time. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California.* SIAM, 38–47. http://dl.acm.org/citation.cfm?id=313651.313661.

[33] Gonzalo Navarro. 2000. Improved approximate pattern matching on hypertext. *Theor. Comput. Sci.* 237, 1-2 (2000), 455–463.

[34] Adam M Novak, Erik Garrison, and Benedict Paten. 2016. A graph extension of the positional Burrows-Wheeler transform and its applications. In *International Workshop on Algorithms in Bioinformatics (LNCS)*, Vol. 9838. Springer, 246–256.

[35] S. C. Ntafos and S. Louis Hakimi. 1979. On path cover problems in digraphs and applications to program testing. *IEEE Trans. Software Engrg.* 5, 5 (1979), 520–529.

[36] James B. Orlin. 2013. Max flows in $O(nm)$ time, or better. In *Proceedings of the 45th Annual ACM Symposium on the Theory of Computing (STOC '13)*. ACM, New York, NY, USA, 765–774.

[37] Kunsoo Park and Dong Kyue Kim. 1995. String Matching in Hypertext. In *CPM (LNCS)*, Vol. 937. Springer, 318–329.

[38] Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. 2017. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods* 14, 4 (04 2017), 417–419. http://dx.doi.org/10.1038/nmeth.4197.

[39] Mikko Rautiainen and Tobias Marschall. 2017. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv* (2017), 216–127.

[40] Romeo Rizzi, Alexandru I. Tomescu, and Veli Mäkinen. 2014. On the complexity of Minimum Path Cover with Subpath Constraints for multi-assembly. *BMC Bioinformatics* 15, S-9 (2014), S5.

[41] Claus-Peter Schnorr. 1978. An Algorithm for Transitive Closure with Linear Expected Time. *SIAM J. Comput.* 7, 2 (1978), 127–133.

[42] Tetsuo Shibuya and Igor Kurochkin. 2003. Match Chaining Algorithms for cDNA Mapping. In *Proc. Algorithms in Bioinformatics (WABI 2003) (LNCS)*, Vol. 2812. Springer, 462–475.

[43] Jouni Sirén. 2017. Indexing Variation Graphs. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 13–27.

[44] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. 2014. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 11, 2 (2014), 375–388.

[45] J. Su, Q. Zhu, H. Wei, and J. X. Yu. 2017. Reachability Querying: Can It Be Even Faster? *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (March 2017), 683–697.

[46] Alexandru I. Tomescu, Travis Gagie, Alexandru Popa, Romeo Rizzi, Anna Kuosmanen, and Veli Mäkinen. 2015. Explaining a Weighted DAG with Few Paths for Solving Genome-Guided Multi-Assembly. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12, 6 (2015), 1345–1354.

[47] Raluca Uricaru, Célia Michotey, Hélène Chiapello, and Eric Rivals. 2015. YOC, A new strategy for pairwise alignment of collinear genomes. *BMC Bioinformatics* 16, 1 (Apr 2015), 111. http://dx.doi.org/10.1186/s12859-015-0530-3.

[48] Peter van Emde Boas. 1977. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Inf. Process. Lett.* 6, 3 (1977), 80–82.

[49] Peter van Emde Boas, R. Kaas, and E. Zijlstra. 1977. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory* 10 (1977), 99–127.

[50] Vijay V. Vazirani. 2001. *Approximation Algorithms*. Springer-Verlag.

[51] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. 2015. A Long Fragment Aligner called ALFALFA. *BMC Bioinformatics* 16, 1 (May 2015), 159. http://dx.doi.org/10.1186/s12859-015-0533-0.

[52] Michaël Vyverman, Dieter De Smedt, Yao-Cheng Lin, Lieven Sterck, Bernard De Baets, Veerle Fack, and Peter Dawyndt. 2014. Fast and Accurate cDNA Mapping and Splice Site Identification. In *Proceedings of the International Conference on Bioinformatics Models, Methods and Algorithms (BIOSTEC 2014)*. SCITEPRESS, 233–238. http://hdl.handle.net/1854/LU-6851320.

[53] Sebastian Wandelt and Ulf Leser. 2014. RRCA: Ultra-Fast Multiple In-species Genome Alignments. In *Algorithms for Computational Biology - First International Conference, AlCoB 2014, Tarragona, Spain, July 1-3, 2014, Proceedings (LNCS)*, Vol. 8542. Springer, 247–261.

[54] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 276–284. http://dx.doi.org/10.14778/1920841.1920879.