Fixed-Parameter Algorithms for Longest Heapable Subsequence and Maximum Binary Tree

Karthekeyan Chandrasekaran
University of Illinois, Urbana-Champaign, USA
karthe@illinois.edu

Elena Grigorescu
Purdue University, USA
elena-g@purdue.edu

Gabriel Istrate
West University of Timişoara, Romania, and the e-Austria Research Institute
gabrielistrate@acm.org

Shubhang Kulkarni[1]
University of Illinois, Urbana-Champaign, USA
smkulka2@illinois.edu

Young-San Lin
Purdue University, USA
lin532@purdue.edu

Minshen Zhu
Purdue University, USA
zhu628@purdue.edu

Abstract

A heapable sequence is a sequence of numbers that can be arranged in a *min-heap data structure*. Finding a longest heapable subsequence of a given sequence was proposed by Byers, Heeringa, Mitzenmacher, and Zervas (ANALCO 2011) as a generalization of the well-studied longest increasing subsequence problem and its complexity still remains open. An equivalent formulation of the longest heapable subsequence problem is that of finding a maximum-sized binary tree in a given permutation directed acyclic graph (permutation DAG). In this work, we study parameterized algorithms for both longest heapable subsequence and maximum-sized binary tree. We introduce *alphabet size* as a new parameter in the study of computational problems in permutation DAGs and show that this parameter with respect to a fixed topological ordering admits a complete characterization and a polynomial time algorithm. We believe that this parameter is likely to be useful in the context of optimization problems defined over permutation DAGs.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability

Keywords and phrases maximum binary tree, heapability, permutation directed acyclic graphs

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.23

Funding *Karthekeyan Chandrasekaran*: Supported by NSF CCF-1814613 and NSF CCF-1907937
Elena Grigorescu: Supported by NSF CCF-1910659 and NSF CCF-1910411.
Gabriel Istrate: Supported by a grant of the Romanian Ministry of Research and Innovation, CNCS - UEFISCDI project number PN-III-P4-ID-PCE-2016-0842, within PNCDI III.
Young-San Lin: Supported by NSF CCF-1910411.
Minshen Zhu: Supported by NSF CCF-1910659.

[1] Work done while at Purdue University, USA.

1 Introduction

The longest increasing subsequence is a fundamental computational problem that has led to numerous discoveries in algorithms as well as combinatorics. The motivation behind this work is a generalization of the longest increasing subsequence problem, known as the *longest heapable subsequence* problem, introduced by Byers, Heeringa, Mitzenmacher, and Zervas [5]. We begin by defining this problem. A rooted tree whose nodes are labeled with values has the *heap property* if the value of every node is at least that of its parent; a sequence of natural numbers is *heapable* if the elements can be sequentially placed one at a time to form a binary tree with the heap property. For example, the sequence $1, 5, 3, 2, 4$ is not heapable while the sequence $1, 3, 3, 2, 4$ is heapable. Throughout this work, we will be interested in sequences whose elements are natural numbers. In the longest heapable subsequence problem, the goal is to find a longest heapable subsequence of a given sequence. Although the longest *increasing* subsequence problem is solvable in polynomial-time, the complexity of the longest *heapable* subsequence problem is still open.

The problem of verifying if a given sequence is heapable, although non-trivial, is solvable efficiently using a greedy approach [5]. In order to address the longest heapable subsequence problem, Porfilio [12] observed a connection to a graph problem on directed acyclic graphs (DAGs). The permutation DAG associated with a sequence $\sigma = (\sigma(1), \sigma(2), \ldots, \sigma(n))$, denoted $\mathsf{PermDAG}(\sigma)$, is obtained by introducing a vertex t_i for every sequence element $i \in [n]$, and arcs (t_j, t_i) for every $i, j \in [n]$ such that $i < j$ and $\sigma(i) \leq \sigma(j)$. We recall that a directed graph G is a *permutation DAG* if there exists a sequence τ such that G is isomorphic to $\mathsf{PermDAG}(\tau)$. We need the notion of a binary tree in a given directed graph G: a subgraph T of G is an r-rooted binary tree if r is the unique vertex in T with no outgoing edges, every vertex in T has a unique directed path to r in T, and every vertex in T has in-degree at most 2 in T; the size of T is the number of vertices in T. Porfilio showed that a longest heapable subsequence of a given sequence σ is equivalent to a maximum-sized binary tree in $\mathsf{PermDAG}(\sigma)$. This result raises the question of whether one can efficiently find a maximum-sized binary tree in a given permutation DAG. The complexity of this problem also remains open.

In an earlier work [6], we showed that maximum-sized binary tree in arbitrary input directed graphs is fixed-parameter tractable when parameterized by the solution size: we gave a $2^k n^{O(1)}$ time algorithm, where k is the size of the largest binary tree and n is the number of vertices in the input graph. This also implies that the longest heapable subsequence problem is fixed-parameter tractable when parameterized by the solution size. In this work, we consider two alternative parameterizations for the maximum-sized binary tree/longest heapable subsequence problem.

Firstly, we show that the longest heapable subsequence problem is fixed-parameter tractable when parameterized by the number of distinct values in the input sequence. Next, we introduce *alphabet size* as a new parameter in the study of computational problems in permutation DAGs. Our algorithmic result for longest heapable subsequence problem implies that the maximum-sized binary tree problem in a given permutation DAG is fixed-parameter tractable when parameterized by the alphabet size. We currently do not know how to compute the alphabet size of a given permutation DAG. As a stepping stone towards computing alphabet size, we show that alphabet size *with respect to a fixed topological ordering* can be computed efficiently and it also admits a min-max relation and a polyhedral description. Our results suggest that alphabet size is an interesting parameterization for computational problems defined on permutation DAGs and merits a thorough study. Finally, we design a fixed-parameter algorithm for the maximum-sized binary tree problem in undirected graphs when parameterized by treewidth. We elaborate on our contributions now.

1.1 Results

Our first result shows that the longest heapable subsequence problem is fixed-parameter tractable when parameterized by the number of distinct values in the sequence.

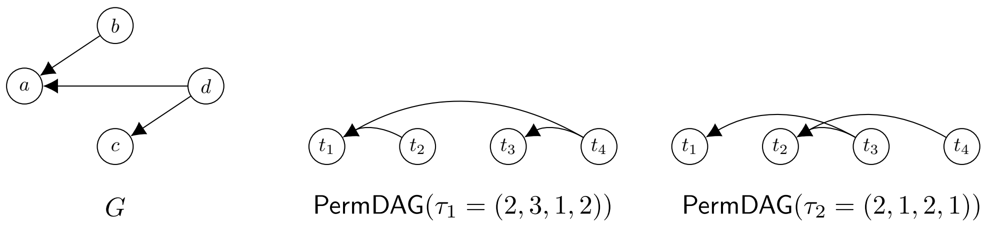
▶ **Theorem 1.** *There exists an algorithm that takes as input an n-length sequence τ with k distinct values and returns a longest heapable subsequence of τ in time $(k+1)! \cdot k \cdot O(n)$. Equivalently, our algorithm returns a maximum-sized binary tree in* $\mathsf{PermDAG}(\tau)$.

We emphasize that our algorithm also works in the streaming model of computation—i.e., when the input sequence arrives one by one and the algorithm has to find the longest heapable subsequence of the input that has arrived so far with sublinear memory (in particular, the algorithm does not have the capability to store the entire input sequence seen so far). The space complexity of our algorithm is $(k+1)! \cdot k \cdot O(\log n)$ and is logarithmic for constant k.

Theorem 1 can also be viewed as a fixed-parameter algorithm to find a maximum-sized binary tree in a given permutation DAG when parameterized by *alphabet size*. We define this parameter now. We note that for a fixed permutation DAG G, there could be several sequences τ such that $\mathsf{PermDAG}(\tau)$ is isomorphic to G (e.g., see Figure 1). In fact, there could be sequences τ_1 and τ_2 such that the difference between the number of distinct symbols in τ_1 and τ_2 may be arbitrarily large—e.g., consider an n-vertex tournament DAG G in its unique topological ordering (all arcs oriented in the backward direction) which is isomorphic to $\mathsf{PermDAG}(\tau_1)$ as well as $\mathsf{PermDAG}(\tau_2)$ where $\tau_1 = (1, 2, \dots, n)$ and $\tau_2 = (1, 1, \dots, 1)$. This motivates our parameterization for permutation DAGs: The *alphabet size* of a n-vertex permutation DAG G, denoted $\alpha(G)$, is defined as follows (see Figure 1 for an example):

$$\alpha(G) := \min\{k : \exists \text{ sequence } \tau \in [k]^n \text{ with } \mathsf{PermDAG}(\tau) \text{ being isomorphic to } G\}.$$

We recall that directed graphs $G = (V, A)$ and $G' = (V,', A')$ are isomorphic if there exists a bijection $\phi : V' \to V$ such that $(u', v') \in A'$ if and only if $(\phi(u'), \phi(v')) \in A$. Theorem 1 also implies that there exists an algorithm that takes as input, an n-vertex permutation DAG G and a sequence τ with $\alpha(G) = k$ distinct values such that $\mathsf{PermDAG}(\tau)$ is isomorphic to G and returns a maximum-sized binary tree in G in time $(k+1)! \cdot k \cdot O(n)$, i.e., a fixed-parameter algorithm for maximum-sized binary tree in permutation DAGs when parameterized by alphabet size.



■ **Figure 1** Let G be the input permuation DAG. The graph G is isomorphic to $\mathsf{PermDAG}(\tau_1)$ and $\mathsf{PermDAG}(\tau_2)$. The sequence $\tau_1 = (2, 1, 2, 1)$ uses only two distinct values, which turns out to be the minimum, so $\alpha(G) = 2$.

Next, we explore algorithmic aspects of our newly defined parameter, namely the alphabet size. A natural question is whether the alphabet size of a given permutation DAG can be computed in polynomial-time. Currently, we do not know the answer to this question. However, there is a natural related problem that seems like a stepping stone towards resolving the complexity of computing the alphabet size of permutation DAGs. We define this related problem now.

We recall that every DAG $G = (V, A)$ admits a topological ordering—a bijection $\gamma : V \to [n]$ corresponding to a permutation of its n vertices such that every arc $(v, u) \in A$ has $\gamma(u) < \gamma(v)$ (i.e., all edges are oriented in the backward direction with respect to the ordering defined by γ). For a fixed topological ordering $\gamma : V \to [n]$ of an n-vertex permutation DAG G, we define the *γ-alphabet size of G*, denoted $\alpha(G, \gamma)$, as follows (see Figure 2 for an example illustrating the definition):

$$\begin{aligned}\alpha(G,\gamma) := \min\Big\{k : \exists \text{ sequence } \tau \in [k]^n \text{ with } \mathsf{PermDAG}(\tau) = (\{t_1,\dots,t_n\}, A') \\ \text{being isomorphic to } G \text{ under the mapping } \phi : \{t_1,\dots,t_n\} \to V(G) \\ \text{given by } \phi(t_i) = \gamma^{-1}(i)\ \forall\ i \in [n]\Big\}.\end{aligned}$$

We note that the optimization problem $\alpha(G, \gamma)$ may be infeasible in which case, we use $\alpha(G, \gamma) := \infty$ as the convention. The following relationship between alphabet size and γ-alphabet size is immediate for a permutation DAG G:

$$\alpha(G) = \min\{\alpha(G, \gamma) : \gamma \text{ is a topological ordering of } G\}.$$

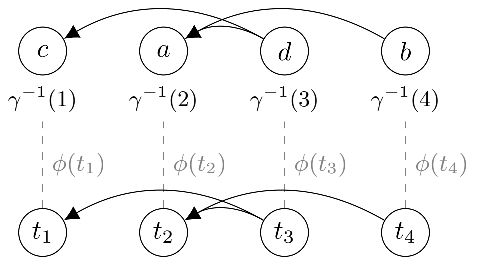


Figure 2 The graph at the top corresponds to G in topological order γ where $\gamma(c) = 1$, $\gamma(a) = 2$, $\gamma(d) = 3$, and $\gamma(b) = 4$. The graph at the bottom corresponds to $\mathsf{PermDAG}(\tau = (2, 1, 2, 1))$. Note that the two graphs are isomorphic under the mapping $\phi : \{t_1, t_2, t_3, t_4\} \to V(G)$ given by $\phi(t_1) = \gamma^{-1}(1) = c$, $\phi(t_2) = \gamma^{-1}(2) = a$, $\phi(t_3) = \gamma^{-1}(3) = d$, and $\phi(t_4) = \gamma^{-1}(4) = b$ (shown by dotted lines). We have that $\alpha(G, \gamma) = 2$ and is achieved by the sequence τ.

As a stepping stone towards understanding $\alpha(G)$, we show that $\alpha(G, \gamma)$ for a given topological ordering γ of G (i.e., the γ-alphabet size of G) can be computed in polynomial time.

▶ **Theorem 2.** *There exists a polynomial-time algorithm that takes a permutation DAG G and a topological ordering γ of G as input and detects if $\alpha(G, \gamma)$ is finite and if so, then returns a sequence that achieves $\alpha(G, \gamma)$.*

Our algorithm underlying Theorem 2 also reveals a min-max relation for γ-alphabet size that we describe now. Let $G = (V, A)$ be a permutation DAG with n vertices and let $\gamma : V \to [n]$ be a topological ordering of G such that $\alpha(G, \gamma)$ is finite. Let $\overrightarrow{E} := \{(u, v) : \gamma(u) < \gamma(v) \text{ and } (v, u) \notin A\}$ and $H(G, \gamma) := (V, A \cup \overrightarrow{E})$. We note that $H(G, \gamma)$ is a tournament.[2] Also, let $w : A \cup \overrightarrow{E} \to \{0,1\}$ be an arc weight function for $H(G,\gamma)$ defined as follows:

$$w(e) := \begin{cases} 0 & \text{if } e \in A, \\ 1 & \text{if } e \in \overrightarrow{E}. \end{cases}$$

Then, we have the following min-max relation for the minimization problem corresponding to $\alpha(G,\gamma)$.

▶ **Theorem 3.** *Let $G = (V, A)$ be a permutation DAG and γ be a topological ordering of V such that $\alpha(G,\gamma)$ is finite. Then,*

$$\alpha(G,\gamma) = 1 + \max\left\{ \sum_{e \in P} w(e) : P \textit{ is a path in } H(G,\gamma) \right\}.$$

In addition to the algorithm and the min-max relation, we give a polyhedral description (see Theorem 17 in Section 3.3) that also leads to an LP-based algorithm to compute $\alpha(G,\gamma)$. We believe that alphabet size, as a parameter, is likely to be useful in the context of permutation DAGs and consequently, merits a thorough study. We view Theorems 2 and 3 as stepping stones towards the problem of efficiently computing the alphabet size of a given permutation DAG and Theorem 1 to be an application of this parameter. Resolving the complexity of computing the alphabet size is an intriguing open problem.

Next, we address the maximum binary tree problem in undirected graphs with bounded treewidth. Here, we are given an undirected graph G and the goal is to find a subgraph that is a binary tree with maximum number of nodes. An undirected graph is said to be a *binary tree* if the graph is acyclic and every vertex has degree at most 3. We observe that the existence of a binary tree can be expressed as a monadic second order logic property and hence, extensions of Courcelle's theorem [7] can be used to obtain an algorithm for maximum-sized binary tree that runs in time $f(w)n$ for some function $f(w)$, where n is the number of vertices and w is the treewidth of the input graph. However, the run-time dependence $f(w)$ is at least doubly exponential on the treewidth w in this approach. We improve this dependence substantially.

▶ **Theorem 4.** *Given a tree decomposition of an n-vertex undirected graph G with treewidth w, there exists an algorithm to find a maximum-sized binary tree in G in time $w^{O(w)}n$.*

1.2 Related Work

Heapability of integer sequences was introduced in [5] and has been investigated further in [9, 12, 10, 2, 3, 4, 1]. Heapability of integer sequences can be decided by a simple greedy algorithm [5] (see also [10] for an alternate approach based on integer programming, and [1] for connections with Dilworth's theorem and an algorithm based on network flows). Besides introducing the longest heapable subsequence problem, [5] also showed that deciding if a sequence can be arranged in a *complete* binary heap is NP-complete.

Heapable sequences of integers can be regarded as "loosely increasing". The celebrated *Ulam-Hammesley problem* aims to understand the length of the longest increasing sequence of a random permutation. This has a long history with deep connections to many areas

[2] A tournament is a directed graph $H = (V, A)$ in which we have exactly one of the two arcs (v, u) and (u, v) for every pair of distinct vertices $u, v \in V$.

of science (e.g., see [13]). [5] studied the counterpart of this problem for heapability: they showed that the longest heapable subsequence of a random permutatoin of length n is of size $n - o(n)$ with high probability and it can also be found in an online fashion.

As mentioned earlier, Porfilio [12] showed that the longest heapable subsequence is equivalent to solving the maximum-sized binary tree problem in permutation DAGs. In an earlier work [6], we showed that the maximum-sized binary tree problem is NP-hard in DAGs and showed further inapproximability results. We also gave a fixed-parameter algorithm for the maximum binary tree problem when parameterized by the solution size. Furthermore, we designed a polynomial-time algorithm to solve the maximum-sized binary tree problem in the special class of bipartite permutation graphs. It is also known that maximum-sized binary tree problem in DAGs induced by sets of intervals can be solved in polynomial time [1].

Organization. In Section 2, we present the fixed-parameter algorithm for longest heapable subsequence when parameterized by the alphabet size and prove Theorem 1. In Section 3, we address the problem of computing γ-alphabet size and present a min-max relation. We present a polyhedral description for γ-alphabet size that leads to an LP-based algorithm for $\alpha(G, \gamma)$ in Section 3.3. Due to page limits, we present our fixed-parameter algorithm for computing a maximum-sized binary tree in bounded treewidth graphs in the full version[3].

2 Longest heapable subsequence parameterized by alphabet size

In this section, we prove Theorem 1 by giving a $(k+1)! \cdot k \cdot O(n)$-time algorithm to compute the longest heapable subsequence of a given n-length sequence containing k distinct values. We begin with certain useful definitions. Given a rooted non-empty binary tree T, we define the extended binary tree $Ext(T)$ by introducing new leaf nodes in a way that makes every node in T have exactly 2 children. The nodes in T are also referred to as *internal nodes*, and the new leaf nodes are referred to as *external nodes* (see Figure 3). We will denote a directed binary tree where each node is labeled by some number in $[k] := \{1, 2, \dots, k\}$ such that the labels on every leaf to root path is non-increasing as a *heap over alphabet* $[k]$.

▶ **Definition 5** (Shape). *Given a heap H over alphabet $[k]$, we define its* shape *as a tuple $\mathbf{x} = (x_0, x_1, \dots, x_{k-1}, x_k)$, where x_i is the number of external nodes whose parents have label i in $Ext(H)$. We also follow the convention that the shape of an empty heap is $(1, 0, \dots, 0)$.*

Intuitively, an external node represents the location of a potential future insertion into the heap. Since an insertion is effectively replacing an external node with a new internal node (thus introducing two new external nodes), it is captured by simple manipulations of shapes. This naturally leads us to defining insertions with respect to shapes. Given a shape $\mathbf{x} = (x_0, \dots, x_k)$ and labels $a \le b$, the shape obtained by inserting b under a, denoted $\mathbf{x}(a \leftarrow b)$, is defined as

$$\mathbf{x}(a \leftarrow b) := \begin{cases} (x_0, \dots, x_{a-1}, x_a + 1, x_{a+1}, \dots, x_k) & \text{if } x_a > 0 \text{ and } a = b, \\ (x_0, \dots, x_{a-1}, x_a - 1, x_{a+1}, \dots, x_{b-1}, x_b + 2, x_{b+1}, \dots, x_k) & \text{if } x_a > 0 \text{ and } a < b, \\ \perp & \text{if } x_a = 0. \end{cases}$$

For example, consider the shape $\mathbf{x} = (0, 1, 0, 2, 0)$. The shape $\mathbf{x}(1 \leftarrow 2)$ is $(0, 0, 2, 2, 0)$, and the shape $\mathbf{x}(2 \leftarrow 3)$ is $\perp$. Given a heap H and a sequence $a = (a_1, \dots, a_n)$, consider a longest

[3] The full version is available at https://www.cs.purdue.edu/homes/lin532/file/FPT_algorithms_for_MBT_IPEC.pdf.

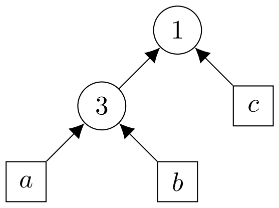


Figure 3 Given a binary tree T composed of two nodes 1 and 3, the extended binary tree $Ext(T)$ has two internal nodes 1 and 3, and three new external nodes a, b, and c. Suppose $k = 4$. The shape of the heap above is $\mathbf{x} = (x_0, x_1, x_2, x_3, x_4) = (0, 1, 0, 2, 0)$ because the parent of a and b has label 3 and the parent of c has label 1.

subsequence of a which can be sequentially inserted to H as leaf nodes while maintaining the heap property. We will call such a subsequence a *longest heapable subsequence starting from H*. We observe that the longest heapable subsequence of a given sequence starting from H depends only the shape of H and not the precise structure of H (i.e., the optimum does not change for two different heaps H_1 and H_2 sharing the same shape). Therefore, it is equivalent and also convenient to consider the longest heapable subsequence problem starting from an initial shape instead of an initial heap. This line of thought also suggests a natural dynamic programming approach where the subproblems are specified by shapes.

To analyze the running time, we need to upper bound the number of subproblems, which is the same as the number of distinct shapes. As a starting point, the number of distinct shapes can be upper bounded by $n^{O(k)}$. This is because in any n-node heap H there are exactly $n + 1$ external nodes in $Ext(H)$ (an elementary property of binary trees). Therefore, the number of shapes is bounded by the number of non-negative integral solutions to $x_0 + x_1 + \ldots + x_k = n + 1$, which is $n^{O(k)}$. Although this estimate seems like a very crude upper bound, bringing down the estimate into the fixed-parameter regime (i.e., $f(k)n^{O(1)}$) seems very difficult. We employ additional ideas to design a fixed-parameter algorithm.

Consider the longest heapable subsequence problem starting from initial shape $\mathbf{x} = (x_0, x_1, \ldots, x_k)$. Suppose that the initial shape also satisfies the condition that $x_j \geq k - j + 1$ for some $j \in [k]$. Our key observation is that all elements with labels at least j are heapable from $\mathbf{x}$: we can reserve an external node attached to j for each label $v \in \{j, j + 1, \ldots, k\}$, which can then be used to form a chain of elements with the same label v. Essentially, once we have reached the shape $\mathbf{x}$, there are "infinitely" many external nodes available for future elements with label at least j, and hence, we no longer need to keep track of the precise values of $x_j, x_{j+1}, \ldots, x_k$. This motivates the following notion of refined shapes.

▶ **Definition 6** (Refined shapes). *A tuple $(x_0, x_1, \ldots, x_k)$ is a* refined shape *(over alphabet size k) if for each $j \in \{0, 1, \ldots, k\}$ we have $x_j \in \{0, 1, \ldots, k - j\} \cup \{\infty\}$, and $x_j = \infty$ implies $x_\ell = \infty$ for all $\ell > j$. We will write $\mathcal{X}_k$ for the set of all refined shapes over alphabet size k.*

We are going to see later that the total number of refined shapes is bounded by $O((k+1)!)$. The operation refine$(\cdot)$ introduced below formalizes the intuition discussed earlier.

▶ **Definition 7.** *Let $\mathbf{x} = (x_0, \ldots, x_k)$ be such that $x_j \in \mathbb{N} \cup \{\infty\}$ for all j. Let*

$$\text{refine}(\mathbf{x}) := \begin{cases} \mathbf{x} & \text{if } x_j \leq k - j \text{ for all } j, \\ (x_0, \ldots, x_{j_0 - 1}, \infty, \infty, \ldots) & j_0 \text{ is the smallest } j \text{ such that } x_j \geq k - j + 1. \end{cases}$$

We remark that refine($\mathbf{x}$) $\in \mathcal{X}_k$ for any $\mathbf{x}$. Next we define insertions with respect to refined shapes. Given a refined shape $\mathbf{x} = (x_0, \dots, x_k)$ and labels $a \le b$, the shape obtained by inserting b under a, denoted $\mathbf{x}(a \leftarrow b)$, is defined as

$$\mathbf{x}(a \leftarrow b) \coloneqq \begin{cases} \text{refine}\,(x_0, \dots, x_{a-1}, x_a + 1, x_{a+1}, \dots, x_{k-1}) & \text{if } x_a > 0 \text{ and } a = b, \\ \text{refine}\,(x_0, \dots x_{a-1}, x_a - 1, x_{a+1}, \dots, x_{b-1}, x_b + 2, x_{b+1}, \dots, x_{k-1}) & \text{if } x_a > 0 \text{ and } a < b, \\ \perp & \text{if } x_a = 0. \end{cases}$$

where we followed the convention that $\infty > 0$ and $\infty + c = \infty$ for any constant c.

Now we are ready to state the dynamic programming algorithm. In the following, we fix $(a_1, a_2, \dots, a_n)$ as the input sequence. For $\mathbf{x} \in \mathcal{X}_k$ and $i \in [n]$ define $\mathsf{LHS}[i, \mathbf{x}]$ to be the length of the longest heapable subsequence in the prefix sequence $(a_1, a_2, \dots, a_i)$, with an additional constraint that the refined shape of the heap constructed from the subsequence should be $\mathbf{x}$. We write $\mathsf{LHS}[i, \mathbf{x}] = -\infty$ if there is no feasible solution (i.e. shape $\mathbf{x}$ is not reachable by any subsequence of $(a_1, \dots, a_i)$). With this definition, the longest heapable subsequence of the given sequence has length $\max_{\mathbf{x} \in \mathcal{X}_k} \mathsf{LHS}[n, \mathbf{x}]$. Our goal now is to compute $\mathsf{LHS}[n, \mathbf{x}]$ for each $\mathbf{x} \in \mathcal{X}_k$.

For a label $v \in \{1, 2, \dots, k\}$ and two refined shapes $\mathbf{x}$ and $\mathbf{x}'$, we say that $\mathbf{x}$ *is reachable from* $\mathbf{x}'$ *via an insertion of* v if there exists $b \le v$ such that $\mathbf{x}'(b \leftarrow v) = \mathbf{x}$. We denote by $\texttt{prev}(\mathbf{x}, v)$ the set of refined shapes from which $\mathbf{x}$ is reachable via an insertion of v. We show that LHS satisfies the following recurrence relation.

▶ **Lemma 8.** *For every $i \in [n]$ and $\mathbf{x} \in \mathcal{X}_k$, we have that*

$$\mathit{LHS}[i, \mathbf{x}] = \max \left\{ \mathit{LHS}[i-1, \mathbf{x}], \max_{\mathbf{x}' \in \mathit{prev}(\mathbf{x}, a_i)} \{\mathit{LHS}[i-1, \mathbf{x}']\} + 1 \right\}.$$

Proof. We will show that

$$\mathsf{LHS}[i, \mathbf{x}] \le \max \left\{ \mathsf{LHS}[i-1, \mathbf{x}], \max_{\mathbf{x}' \in \texttt{prev}(\mathbf{x}, a_i)} \{\mathsf{LHS}[i-1, \mathbf{x}']\} + 1 \right\}$$

as the other direction is trivial. Let us fix an optimal heapable subsequence s of $(a_1, \dots, a_i)$. If a_i does not belong to s, it must be the case that s is also an optimal heapable subsequence of $(a_1, \dots, a_{i-1})$. In this case $\mathsf{LHS}[i, \mathbf{x}] = \mathsf{LHS}[i-1, \mathbf{x}]$. If a_i belongs to s, we further fix an optimal heap H (with refined shape $\mathbf{x}$) and assume that a_i is inserted under an element with value b in H. Removing a_i from H results in a heap H' with a shape $\mathbf{x}'$ satisfying $\mathbf{x}'(b \leftarrow a_i) = \mathbf{x}$. In particular, $\mathbf{x}' \in \texttt{prev}(\mathbf{x}, a_i)$. In this case, $\mathsf{LHS}[i, \mathbf{x}] = \mathsf{LHS}[i-1, \mathbf{x}'] + 1 \le \max_{\mathbf{x}' \in \texttt{prev}(\mathbf{x}, a_i)} \{\mathsf{LHS}[i-1, \mathbf{x}']\} + 1$. ◀

Proof of Theorem 1. Given Lemma 8, it remains to show that the recurrence relation can be implemented in time $(k+1)! \cdot k \cdot O(n)$. We observe that the number of subproblems is bounded by $O(n|\mathcal{X}_k|)$. The set $\texttt{prev}(\mathbf{x}, a_i)$ can be enumerated in time $O(k)$ by inverting the operation $\mathbf{x}'(b \leftarrow a_i)$ for each $b \le a_i$. Therefore, it suffices to show that $|\mathcal{X}_k| = O((k+1)!)$.

In order to bound the size of $\mathcal{X}_k$, we observe that for every $\mathbf{x} = (x_0, x_1, \dots, x_k) \in \mathcal{X}_k$, we have that $x_0 = 0$ unless $\mathbf{x} = (1, 0, \dots, 0)$, and that $x_j \in \{0, 1, \dots, k-j\} \cup \{\infty\}$ for $j \ge 1$. Therefore $|\mathcal{X}_k| \le 1 + \prod_{j=1}^{k} (k - j + 2) = (k+1)! + 1$. ◀

Algorithm 1 gives an implementation of this dynamic programming algorithm. This implementation requires space complexity $O((k+1)! n \cdot \log n)$, which can be optimized to

$O((k+1)! \cdot \log n)$ using a standard rolling array technique: we observe that in the recurrence relation, $\mathsf{LHS}[i, \mathbf{x}]$ depends only on $\mathsf{LHS}[i-1, \mathbf{x}']$ but not on $\mathsf{LHS}[j, \mathbf{x}']$ for any $j < i-1$. Therefore the values $\mathsf{LHS}[i-2, \mathbf{x}]$ become obsolete and the space can be recycled to store new values. Essentially, we only need two arrays $\mathsf{LHS}_1[\mathbf{x}]$ and $\mathsf{LHS}_2[\mathbf{x}]$ and store new values alternately between them.

```
Input: A sequence $a = (a_1, \ldots, a_n)$ such that $\forall i \in [n]$, $a_i \in \{1, 2, \ldots, k\}$.
Output: The length of longest heapable subsequence in $a$.

LHS$(a_1, a_2, \ldots, a_n)$ :
 1: $\mathcal{X} \leftarrow \{(1, 0, \ldots, 0)\}$        ▷ $\mathcal{X}$ maintains a set of reachable refined shapes
 2: LHS ← integer array of size $n \times (k+1) \times k \times \ldots 2 \times 1$
 3: $\mathsf{LHS}[0, (1, 0, \ldots, 0)] \leftarrow 0$
 4: for $i \leftarrow 1$ to $n$ do                                  ▷ DP main body
 5:     for $\mathbf{x} \in \mathcal{X}$ do
 6:         $\mathsf{LHS}[i, \mathbf{x}] \leftarrow \mathsf{LHS}[i-1, \mathbf{x}]$                    ▷ Discard $a_i$
 7:     for $\mathbf{x} \in \mathcal{X}$ do
 8:         for $b \in \{b' : 0 \le b' \le a_i, x_{b'} > 0\}$ do
 9:             $\mathbf{x}' \leftarrow \mathbf{x}(b \leftarrow a_i)$        ▷ Insert $a_i$ under $b$ to reach refined shape $\mathbf{x}'$
10:             if $\mathbf{x}' \notin \mathcal{X}$ then             ▷ First time reaching shape $\mathbf{x}'$
11:                 $\mathcal{X} \leftarrow \mathcal{X} \cup \{\mathbf{x}'\}$
12:                 $\mathsf{LHS}[i, \mathbf{x}'] \leftarrow \mathsf{LHS}[i-1, \mathbf{x}] + 1$
13:             else if $\mathsf{LHS}[i, \mathbf{x}'] < \mathsf{LHS}[i-1, \mathbf{x}] + 1$ then
14:                 $\mathsf{LHS}[i, \mathbf{x}'] \leftarrow \mathsf{LHS}[i-1, \mathbf{x}] + 1$
    return max $\{\mathsf{LHS}[n, \mathbf{x}] : \mathbf{x} \in \mathcal{X}\}$
```

1 Longest Heapable Subsequence for Alphabet Size k

Remark. We note that our dynamic programming algorithm also works in the streaming model, where the elements of the input sequence have to be processed one by one without storing all of them in memory and the goal is to find the length of a longest heapable subsequence of the input that has arrived so far. For constant alphabet size k, the space complexity of our algorithm is $O(\log n)$.

3 γ-Alphabet Size of Permutation DAGs

In this section, we consider the problem of computing the γ-alphabet size of a permutation DAG G, where γ is a given topological ordering of G. We give an efficient algorithm in Section 3.1 and a min-max relation in Section 3.2. We also give a polyhedral description in Section 3.3. We begin with some useful background on permutation DAGs.

We recall that a directed graph G is a permutation DAG if there exists a sequence σ such that $\mathsf{PermDAG}(\sigma)$ is isomorphic to G. We note that permutation DAGs are *transitively closed*, i.e., for a permutation DAG $G = (V, A)$, if $(u, v), (v, w) \in A$, then $(u, w) \in A$. In order to recognize if a given DAG is a permutation DAG, we need the notion of *umbrella-free ordering* defined below (see Figure 4 for an example). This notion will also help us recognize if $\alpha(G, \gamma)$ is finite.

▶ **Definition 9** (Umbrella-free Order). *Let $G = (V, A)$ be an n-vertex DAG. An order $\gamma : V \to [n]$ of V is* umbrella-free *if for all $(v, u) \in A$ and for every vertex $w \in V$ with*

$\gamma(u) < \gamma(w) < \gamma(v)$, either $(w, u) \in A$ or $(v, w) \in A$ (or both).

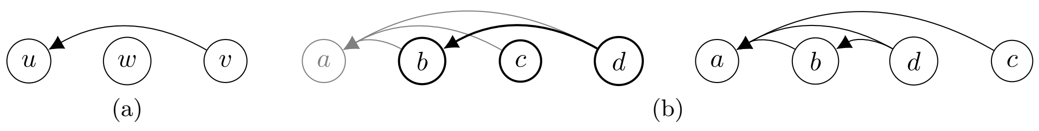


Figure 4 (a) Scenario when the triple (u, w, v) is an *umbrella*. (b) Two topological orderings of the same DAG. The order (a, b, c, d) is not umbrella-free due to the (highlighted) umbrella (b, c, d), while the the order (a, b, d, c) is umbrella-free.

The following lemma characterizes permutation DAGs in terms of the existence of an umbrella-free topological ordering.

▶ **Lemma 10** ([11, 8]). *Let $G = (V, A)$ be a transitively closed DAG. Then G is a permutation DAG if and only if there exists an umbrella-free topological ordering of G. Moreover, there exists a polynomial-time algorithm to verify if a given DAG G is a permutation DAG and if so, then construct an umbrella-free topological ordering of G.*

Lemma 10 implies that $\alpha(G, \gamma)$ is finite if and only if γ is an umbrella-free topological ordering of G.

3.1 Algorithm

In this section, we will prove Theorem 2—we will give an algorithm to compute the γ-alphabet size of a given permutation DAG G, i.e., $\alpha(G, \gamma)$, where γ is a topological ordering of G.

We note that umbrella-freeness of a given topological ordering can be verified in polynomial-time, so we may henceforth assume that the input γ is in fact an umbrella-free topological ordering of G. We will give an iterative algorithm to compute $\alpha(G, \gamma)$. We observe that computing $\alpha(G, \gamma)$ involves assigning a value to each vertex of G such that the sequence obtained by ordering the values of the vertices in the same order as γ gives the same permutation DAG as G. At each iteration, our algorithm will choose a vertex of G and assign a value to it. The next definition will allow us to formally define the choice of this vertex.

▶ **Definition 11** (Fully Suffix Connected Vertex). *Let $G = (V, A)$ be a permutation DAG and γ be a topological ordering of G. A vertex $u \in V$ is* fully suffix connected *if for all $v \in V$ such that $\gamma(v) > \gamma(u)$, we have $(v, u) \in A$. The γ-*least fully suffix connected (γ-LFSC) vertex *is the fully suffix connected vertex u with smallest $\gamma(u)$.*

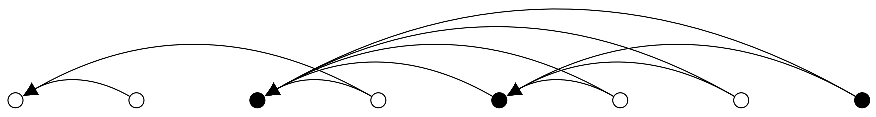


Figure 5 The DAG in the given topological order γ has 3 fully suffix connected vertices that are depicted as filled circles. The leftmost fully suffix connected vertex is the (unique) γ-LFSC vertex.

See Figure 5 for an example showing fully suffix connected vertices. We note that γ-LFSC is unique. The following lemma states a useful property of the γ-LFSC vertex.

▶ **Lemma 12.** *Let $G = (V, A)$ be a permutation DAG and γ be an umbrella-free topological ordering of G. Then, the γ-*LFSC *vertex has no outgoing arcs in G.*

Proof. Let $v \in V$ be the γ-LFSC and suppose for contradiction that v has an outgoing arc in G. Let u be the vertex with largest $\gamma(u)$ such that $(v, u) \in A$. We note that $\gamma(u) < \gamma(v)$ since γ is a topological ordering. We will show that such a vertex u is fully suffix connected and hence contradicts the γ-least fully suffix connected property of vertex v.

We first show that for every vertex $w \in V$ such that $\gamma(w) \geq \gamma(v)$, we have $(w, u) \in A$. For $w = v$, this follows since $(v, u) \in A$ by the choice of u. Let w be a vertex such that $\gamma(w) > \gamma(v)$. Since v is fully suffix connected, we have that $(w, v) \in A$. Also, since G is a permutation DAG, it is transitively closed. Hence, $(v, u) \in A$ implies that $(w, u) \in A$.

Next, we show that for every vertex $w \in V$ such that $\gamma(u) < \gamma(w) < \gamma(v)$, we have $(w, u) \in A$. Let w be a vertex such that $\gamma(u) < \gamma(w) < \gamma(v)$. By assumption, the ordering γ is umbrella-free. Thus, at least one of (w, u) or (v, w) must exist in A. However, $(v, w) \notin A$ as otherwise, w will contradict the choice of vertex u. Therefore, $(w, u) \in A$. ◀

We now discuss a high level overview of our iterative greedy algorithm for computing $\alpha(G, \gamma)$. During the first iteration, the algorithm greedily chooses the γ-LFSC vertex v_1 (say) in $G_1 := G$ to assign the smallest alphabet, namely $\sigma(v_1) = 1$. The vertex v_1 and its incident edges are deleted from G_1 to form G_2, and the remaining $n - 1$ vertices $V \backslash \{v\}$ are ordered in the same relative order as γ—denote this ordering as γ_2. In the second iteration, our algorithm greedily chooses the γ_2-LFSC vertex v_2 (say) in G_2 to assign the next smallest alphabet—the next smallest alphabet is chosen based on whether v_2 lies to the left or right of v_1: if v_2 lies to the left of v_1 with respect to γ, then we set $\sigma(v_2) = \sigma(v_1) + 1$, otherwise we set $\sigma(v_2) = \sigma(v_1)$. This iterative removal and assignment process continues for n iterations, i.e., until all vertices are removed from G. The final output sequence will just be the sequence of assigned values in the order of vertices in γ. Before presenting our complete algorithm (Algorithm 2), we introduce a definition to formalize the reordering of vertices after removing a vertex from G — this will allow us to obtain γ_{i+1} from γ_i.

▶ **Definition 13** (Projected order)**.** *Let $G = (V, A)$ be an n-vertex DAG, and γ be a topological ordering of V. Let $H = G - v$. Then the* projection of γ onto H, *denoted by* $\mathsf{Proj}_H[\gamma] : V \backslash \{v\} \to [n - 1]$, *is defined as*

$$\mathsf{Proj}_H[\gamma](u) = \begin{cases} \gamma(u) & \textit{if } \gamma(u) < \gamma(v), \\ \gamma(u) - 1 & \textit{if } \gamma(u) > \gamma(v). \end{cases}$$

Armed with the notions of fully suffix connected vertices and projected order, we state our algorithm below.

```
Input: Permutation DAG $G = (V, A)$ on $n$ vertices in umbrella-free topological order $\gamma : V \to [n]$
Output: Sequence $\sigma$ of length $n$

GreedyAssign($G, \gamma$):
 1: Initialize $\alpha \leftarrow 1$;   $G_1 \leftarrow G$;   $\gamma(v_0) \leftarrow -\infty$;   $\gamma_1 \leftarrow \gamma$
 2: for $i \leftarrow 1$ to $n$ do
 3:     $v_i \leftarrow \gamma_i$-LFSC in $G_i$
 4:     if $\gamma(v_i) < \gamma(v_{i-1})$ then $\alpha \leftarrow \alpha + 1$
 5:     $G_{i+1} \leftarrow G_i - v_i$
 6:     $\gamma_{i+1} \leftarrow \mathsf{Proj}_{G_{i+1}}[\gamma_i]$
 7:     $\sigma(v_i) \leftarrow \alpha$
 8: Return $\sigma \leftarrow (\sigma(\gamma^{-1}(1)) \dots \sigma(\gamma^{-1}(n)))$
```

2 `GreedyAssign` algorithm to compute $\alpha(G, \gamma)$

The algorithm can be implemented to run in polynomial-time since a γ-LFSC vertex in G can be computed in polynomial-time. We now prove the correctness of the algorithm. Let $G = (V, A)$ be an n-vertex permutation DAG, and γ be an umbrella-free topological ordering of G. Let $v_1, \dots, v_n$ be the sequence of vertices chosen in the execution of `GreedyAssign`(G, γ). Let α_i, G_i and γ_i denote the alphabet size α at the end of the i^{th} iteration, the remaining subgraph at the start of the i^{th} iteration, and γ projected onto G_i respectively. Finally, let σ be the sequence returned by `GreedyAssign`(G, γ). We have the following observations about the execution of the algorithm.

▷ Observation 1. The vertex v_i has no outgoing arcs in G_i for all $i \in [n]$.

▷ Observation 2. If $\gamma(v_{i+1}) < \gamma(v_i)$ then $\sigma(v_{i+1}) = \sigma(v_i) + 1$ and $(v_i, v_{i+1}) \notin A$, otherwise $\sigma(v_{i+1}) = \sigma(v_i)$ and $(v_{i+1}, v_i) \in A$. Thus, alphabet assignments by `GreedyAssign` are non-decreasing with increasing iterations i.e. $\sigma(v_i) \leq \sigma(v_j)$ for all $i, j \in [n]$ with $i < j$.

Observation 1 directly follows from Lemma 12. Observation 2 is due to the conditional increment of the alphabet size, α, in `GreedyAssign`. The next two lemmas show feasibility and optimality of `GreedyAssign` respectively. Theorem 2 then immediately follows from Lemmas 14 and 15.

▶ **Lemma 14** (Feasibility of `GreedyAssign`). *Let* $\mathsf{PermDAG}(\sigma) = (\{t_1 \dots t_n\}, A')$. *Then* $\mathsf{PermDAG}(\sigma)$ *is isomorphic to* G *under the mapping* $\phi : \{t_1 \dots t_n\} \to V$ *given by* $\phi(t_i) = \gamma^{-1}(i)$.

Proof. We will prove isomorphism of the two graphs under ϕ by showing that $(u, v) \in A$ if and only if $(\phi^{-1}(u), \phi^{-1}(v)) \in A'$.

For the forward direction, it suffices to show that $\sigma(u) \leq \sigma(v)$ whenever $(u, v) \in A$. We observe that if $(u, v) \in A$, then $\gamma(v) < \gamma(u)$. By Observation 1, `GreedyAssign` must assign $\sigma(v)$ before $\sigma(u)$. Observation 2 then implies that $\sigma(u) \leq \sigma(v)$.

Next we show the contrapositive of the converse direction. Assume that $(u, v) \notin A$. We first consider the case when $\gamma(v) > \gamma(u)$. Let $\phi^{-1}(u) = t_{\gamma(u)}$ and $\phi^{-1}(v) = t_{\gamma(v)}$. By definition of permutation DAGs, $\mathsf{PermDAG}(\sigma)$ does not have arc (t_i, t_j) when $i < j$. Thus $(t_{\gamma(u)}, t_{\gamma(v)}) \notin A$. Next, we consider the case when $\gamma(v) < \gamma(u)$. For this, it suffices to show that $\sigma(u) < \sigma(v)$. Since $(u, v) \notin A$, the vertex v will never become fully suffix connected before the removal of u. Thus `GreedyAssign` sets $\sigma(u)$ before $\sigma(v)$. Thus, by Observation 2, we have that $\sigma(u) \leq \sigma(v)$. Let $u = v_i$ and $v = v_j$, where $i, j \in [n]$ are the iteration numbers during which `GreedyAssign` assigns $\sigma(u)$ and $\sigma(v)$ respectively. Then, there exists

k such that $i \leq k < j$ and $\gamma(v_{k+1}) < \gamma(v_k)$ as otherwise, Observation 2 would imply that $\gamma(v_i) < \gamma(v_j)$, a contradiction. Thus, $\sigma(u) < \sigma(v)$. ◀

▶ **Lemma 15** (Optimality of `GreedyAssign`). *Let σ^* be a sequence achieving $\alpha(G, \gamma)$. Then, $\sigma(v_i) \leq \sigma^*(v_i)$ for all $i \in [n]$.*

Proof. We will show this by induction on i. For the base case of $i = 1$, `GreedyAssign` always sets $\sigma(v_1) = 1$, the smallest possible alphabet assignment. Thus $\sigma(v_1) \leq \sigma^*(v_1)$ holds. For the induction step, let $i \geq 2$. We have the following two cases based on whether `GreedyAssign` incremented the alphabet size while assigning v_i.

1. Suppose $\sigma(v_i) = \sigma(v_{i-1})$. By the description of the algorithm `GreedyAssign`, we have that v_{i-1} is fully suffix connected in G_{i-1} and $\gamma(v_i) > \gamma(v_{i-1})$. Thus, the arc (v_i, v_{i-1}) must exist in G_{i-1} and so also in G. It follows that

$$\sigma(v_i) = \sigma(v_{i-1}) \leq \sigma^*(v_{i-1}) \leq \sigma^*(v_i).$$

   Here, the first inequality is by the induction hypothesis, while the second inequality is due to the observation that $(v_i, v_{i-1}) \in A$.
2. Suppose $\sigma(v_i) \neq \sigma(v_{i-1})$. By the description of the algorithm `GreedyAssign`, we have that $\gamma(v_i) < \gamma(v_{i-1})$. Thus by Observation 1, the arc (v_{i-1}, v_i) does not exist in G_{i-1} and hence, does not exist in G. It follows that

$$\sigma(v_i) = \sigma(v_{i-1}) + 1 \leq \sigma^*(v_{i-1}) + 1 \leq \sigma^*(v_i).$$

   The equality relation is due to Observation 2. The first inequality is due to the induction hypothesis while the second inequality is due to our observation that $(v_{i-1}, v_i) \notin A$.

◀

Remark. Algorithm 2 can be implemented to run in $O(|V| + |A|)$ time. This can be done by using a *priority queue* data structure initialized as a *stack*. All fully suffix connected vertices should be added to the priority queue with priorities being position in γ. The choice of vertex to assign is the vertex with the *minimum* priority. The alphabet size should be incremented whenever a vertex removal results in new vertices becoming fully suffix connected.

3.2 Min-Max Relation

Min-max relations are significant in optimization literature as they are strong indicators for the existence of a polynomial-time algorithm. In the context of algorithm design, min-max relations bring the optimization problem into NP ∩ coNP, thus providing strong evidence for the existence of polynomial-time algorithms. In this section, we prove the min-max relation for $\alpha(G, \gamma)$, i.e., Theorem 3. A consequence of our min-max relation will be an alternative linear time algorithm for computing $\alpha(G, \gamma)$. We believe that the min-max relation could be a useful tool towards computing $\alpha(G)$. We restate and prove the min-max relation below (see Section 1.1 for the definition of the graph $H(G, \gamma)$ and weights w for this graph).

▶ **Theorem 3.** *Let $G = (V, A)$ be a permutation DAG and γ be a topological ordering of V such that $\alpha(G, \gamma)$ is finite. Then,*

$$\alpha(G, \gamma) = 1 + \max \left\{ \sum_{e \in P} w(e) : P \text{ is a path in } H(G, \gamma) \right\}.$$

Proof. We will show the equation by showing inequality in both directions. We begin by showing the lower bound on $\alpha(G,\gamma)$. Let P be any path in $H(G,\gamma)$, and σ be any sequence such that $\mathsf{PermDAG}(\sigma) = (\{t_1,\dots,t_n\}, A')$ is isomorphic to G under the mapping $\phi : \{t_1,\dots,t_n\} \to V$ given by $\phi(t_i) = \gamma^{-1}(i)$. For every arc $(\phi(t_i),\phi(t_j)) \in P$ such that $(\phi(t_i),\phi(t_j)) \in \overrightarrow{E}$, we have the following two observations. First, the arc $(\phi^{-1}(t_j),\phi^{-1}(t_i)) \notin A'$ as the arc $(\phi(t_j),\phi(t_i)) \notin A$. Second, $\sigma(\phi(t_i)) \geq \sigma(\phi(t_j)) + 1$ as $i < j$. It follows that

$$w(P) = \sum_{(u,v)\in P} w(u,v) = \sum_{(u,v)\in P\cap\overrightarrow{E}} w(u,v) \leq \sum_{(u,v)\in P\cap\overrightarrow{E}} \sigma(u) - \sigma(v) \leq \alpha(G,\gamma) - 1.$$

The first and second equations are by definition of $w(P)$ and the weight function w respectively. The first inequality is due to our observation that $\sigma(u) \geq \sigma(v) + 1$ whenever $(u,v) \in \overrightarrow{E}$. Let a and b be the first and last vertices on P. Then the final inequality follows from $\sigma(a) \geq 1$ and $\sigma(b) \leq \alpha(G,\gamma)$.

Next, we show the upper bound on $\alpha(G,\gamma)$. We recall that $v_1,\dots,v_n$ is the order in which `GreedyAssign` processes vertices of G. Consider $P = (v_n, v_{n-1},\dots,v_1)$. To prove the upper bound, it suffices to show that (1) P is a path in $H(G,\gamma)$; and (2) $w(P) \geq \alpha(G,\gamma) - 1$. To prove (1), we show that $(v_i, v_{i-1}) \in A \cup \overrightarrow{E}$ for each $i \geq 2$. Consider the case when $\gamma(v_i) > \gamma(v_{i-1})$. Since v_{i-1} was γ_{i-1}-LFSC in G_i, the arc $(v_i, v_{i-1}) \in A$. Next, consider the case when $\gamma(v_i) < \gamma(v_{i-1})$. By Observation 1, we have that the arc $(v_{i-1}, v_i) \notin A$. Thus, the arc $(v_i, v_{i-1}) \in \overrightarrow{E}$ by definition of $\overrightarrow{E}$. We now prove (2). By Observation 2, and definitions of w and $\overrightarrow{E}$, we have $w(v_i, v_{i-1}) = \sigma(v_i) - \sigma(v_{i-1})$ It follows that

$$w(P) = \sum_{i=2}^{n} w(v_i, v_{i-1}) = \sum_{i=2}^{n} \sigma(v_i) - \sigma(v_{i-1}) = \alpha(G,\gamma) - 1.$$

The second equality is due to our previous observation. The final equality is due to the `GreedyAssign` assignments $\sigma(v_n) = \alpha(G,\gamma)$ and $\sigma(v_1) = 1$. ◀

We remark that although the RHS problem in the min-max relation given in Theorem 3 is the longest path problem in a directed graph, it can be solved in the graph $H(G,\gamma)$ owing to the following lemma. Lemma 16 allows the optimization problem in the RHS of Theorem 3 to be solved in $O(|V|+|A|)$ time by the classical dynamic programming algorithm for maximum weight path in a DAG. This leads to an alternative algorithm for computing $\alpha(G,\gamma)$.

▶ **Lemma 16.** *$H(G,\gamma)$ is a DAG.*

Proof. Suppose for contradiction that $H(G,\gamma)$ contains a cycle. Let $C = (u_1, u_2, \dots u_k, u_1)$ be a cycle with the smallest number of vertices. If $(u_1,u_3) \in A \cup \overrightarrow{E}$, then $C' = (u_1, u_3, \dots, u_k, u_1)$ is a cycle, contradicting our choice of C. Since $H(G,\gamma)$ is a tournament, the arc $(u_3,u_1) \in A \cup \overrightarrow{E}$, and $C' = (u_1,u_2,u_3,u_1)$ is also a cycle i.e. $k = 3$. We recall that the subgraph (V, A) is transitively closed. Thus, at most one edge of C can belong to A. To get the required contradiction, it suffices to show that the subgraph $(V, \overrightarrow{E})$ is transitively closed. Suppose for contradiction that $\overrightarrow{E}$ is not transitively closed. Then, there exist arcs $(u,v),(v,w) \in \overrightarrow{E}$ such that the arc $(u,w) \notin \overrightarrow{E}$. By definition of $\overrightarrow{E}$, we have that $\gamma(u) < \gamma(v) < \gamma(w)$. It follows that the arc $(w,u) \in A$, and the triple (u,v,w) is an umbrella in G ordered by γ. This contradicts that γ is umbrella-free. ◀

3.3 Polyhedral Description for γ-alphabet size

In this section, we give a polyhedral description for the convex-hull of sequences that are feasible for $\alpha(G,\gamma)$. As a consequence, it leads to an LP-based algorithm to compute $\alpha(G,\gamma)$. We emphasize that our polyhedral result is stronger than giving an LP-based algorithm to compute $\alpha(G,\gamma)$: it implies that one can efficiently compute an *integer-valued* sequence $\sigma = (\sigma(1),\ldots,\sigma(n))$ with minimum weight $\sum_{i=1}^{n} w_i\sigma(i)$ for any given non-negative weights $w_1,\ldots,w_n$ such that $\mathsf{PermDAG}(\sigma)$ is isomorphic to G under the mapping $\phi : \{t_1,\ldots,t_n\} \to V$ given by $\phi(t_i) = \gamma^{-1}(i)$ for every $i \in [n]$. The following is the main result of this section.

▶ **Theorem 17.** *Let $G = (V, A)$ be an n-vertex permutation DAG and γ be an umbrella-free topological ordering of its vertices. Let $Q(G,\gamma)$ be the convex-hull of indicator vectors of $\mathbf{x} \in \mathbb{N}^n$ whose sequence $\sigma := (x_1,\ldots,x_n)$ is such that $\mathsf{PermDAG}(\sigma) = (\{t_1,\ldots,t_n\}, A')$ is isomorphic to G under the mapping $\phi : \{t_1,\ldots,t_n\} \to V$ given by $\phi(t_i) = \gamma^{-1}(i)$ for all $i \in [n]$. Then,*

$$Q(G,\gamma) = \left\{ x \in \mathbb{R}^n \;\middle|\; \begin{array}{ll} x_{\gamma(u)} \le x_{\gamma(v)} & \forall (v,u) \in A, \\ x_{\gamma(v)} \le x_{\gamma(u)} - 1 & \forall (v,u) \notin A \textit{ with } \gamma(u) < \gamma(v), \textit{ and} \\ x_i \ge 1 & \forall\, i \in [n] \end{array} \right\}.$$

For notational convenience, let $P(G,\gamma)$ denote the polyhedron defined in the RHS of Theorem 17. Before proving Theorem 17, we describe how $\alpha(G,\gamma)$ can be obtained by optimizing over $P(G',\gamma')$ for a graph G' and an ordering γ' obtained from G and γ. Let $G' = (V', A')$ be obtained from G by adding a vertex t with edges (t,u) for all $u \in V$ and $\gamma' : V' \to [n+1]$ be defined as $\gamma'(u) = \gamma(u)$ if $u \in V$ and $\gamma'(t) = n+1$. We note that if G is a permutation DAG and γ is an umbrella-free topological ordering of G, then G' is also a permutation DAG and γ' is an umbrella-free topological ordering of G'. Moreover, we also have that

$$\alpha(G,\gamma) = \min\left\{ x_{\gamma'(n+1)} : x \in Q(G',\gamma') \right\}.$$

Thus, by Theorem 17, the γ-alphabet size of G, i.e., $\alpha(G,\gamma)$, can be computed by optimizing along the objective direction $(0,\ldots,0,1) \in \mathbb{R}^{n+1}$ over the polyhedron $P(G',\gamma')$.

We now prove Theorem 17.

Proof of Theorem 17. We recall that a point $\mathbf{x}$ is an *extreme point* of a polyhedron if $\mathbf{x}$ cannot be expressed as a convex combination of any two distinct points in the polyhedron. Any extreme point x of $Q(G,\gamma)$ satisfies the constraints defining $P(G,\gamma)$. Thus, $Q(G,\gamma) \subseteq P(G,\gamma)$. In order to show equality, it suffices to show that all extreme points of $P(G,\gamma)$ are integral. Lemma 18 shows that all extreme points of $P(G,\gamma)$ are integral, thus completing the proof of Theorem 17. ◀

▶ **Lemma 18.** *Let $G = (V, A)$ be an n-vertex DAG and γ be a topological ordering of its vertices. If $\mathbf{x}$ is an extreme point of $P(G,\gamma)$, then $\mathbf{x} \in \mathbb{Z}^n$.*

Proof. Suppose for contradiction that $\mathbf{x}$ is non-integral. We will show the existence of two points in $P(G,\gamma)$ such that $\mathbf{x}$ is a convex combination of these points. Let $S := \{i : x_i \notin \mathbb{Z}\}$. We note that the set S is non-empty due to our choice of $\mathbf{x}$. Let $\epsilon \in \mathbb{R}$ be as follows

$$\epsilon := \min_{i \in S} \left\{ \min(x_i - \lfloor x_i \rfloor, \lceil x_i \rceil - x_i) \right\}.$$

Since S is non-empty, we have $\epsilon > \frac{\epsilon}{2} > 0$. Let $\mathbf{y} \in \mathbb{R}^n$ be defined as follows:

$$y_i := \begin{cases} \epsilon/2 & \text{if } i \in S, \\ 0 & \text{otherwise.} \end{cases}$$

We note that $\mathbf{x} = \frac{1}{2}(\mathbf{x}+\mathbf{y}) + \frac{1}{2}(\mathbf{x}-\mathbf{y})$. It suffices to show that $\mathbf{x}+\mathbf{y}, \mathbf{x}-\mathbf{y} \in P$. We will show that the point $\mathbf{x}+\mathbf{y} \in P$ and remark that the proof of $\mathbf{x}-\mathbf{y} \in P$ is along very similar lines. We observe that $\mathbf{y} \geq 0$.

Constraint (3) is always satisfied as $x_i + y_i \geq x_i \geq 1$. We first focus on constraint (1). Consider any arc $(v, u) \in A$. Since $\mathbf{y} \geq 0$, the constraint is easily seen to be satisfied in the cases where (1) $x_{\gamma(u)}, x_{\gamma(v)} \in \mathbb{Z}$; (2) $x_{\gamma(u)}, x_{\gamma(v)} \notin \mathbb{Z}$; and (3) $x_{\gamma(u)} \in \mathbb{Z}$ but $x_{\gamma(v)} \notin \mathbb{Z}$. Consider the case when $x_{\gamma(u)} \notin \mathbb{Z}$ but $x_{\gamma(v)} \in \mathbb{Z}$. Then, we have that

$$x_{\gamma(u)} + y_{\gamma(u)} < x_{\gamma(u)} + \epsilon \leq \left\lceil x_{\gamma(u)} \right\rceil \leq x_{\gamma(v)} = x_{\gamma(v)} + y_{\gamma(v)}.$$

The first inequality is by $y_i \leq \epsilon/2$ for all $i \in [n]$. The second inequality is by definition of ϵ. The third inequality is due to $\mathbf{x} \in P$ and our case assumption that $x_{\gamma(v)} \in \mathbb{Z}$. The equality relation is by definition of $\mathbf{y}$.

Next, we consider constraint (2). Let $\gamma(u) < \gamma(v)$ but $(v, u) \notin A$. Similar to the above analysis, the constraint is easily seen to be satisfied in the cases where (1) $x_{\gamma(u)}, x_{\gamma(v)} \in \mathbb{Z}$; (2) $x_{\gamma(u)}, x_{\gamma(v)} \notin \mathbb{Z}$; and (3) $x_{\gamma(u)} \notin \mathbb{Z}$ but $x_{\gamma(v)} \in \mathbb{Z}$. Consider the case when $x_{\gamma(u)} \in \mathbb{Z}$ but $x_{\gamma(v)} \notin \mathbb{Z}$. Then, we have that

$$x_{\gamma(v)} + y_{\gamma(v)} < x_{\gamma(v)} + \epsilon \leq \left\lceil x_{\gamma(u)} \right\rceil \leq x_{\gamma(u)} = x_{\gamma(u)} + y_{\gamma(u)}.$$

The first inequality is due to $y_i \leq \epsilon/2$ for all $i \in [n]$. The second inequality is by definition of ϵ. The third inequality is due to $\mathbf{x} \in P$ and our case assumption that $x_{\gamma(u)} \in \mathbb{Z}$. The equality relation is by definition of $\mathbf{y}$. ◀

Based on Lemma 18, it is natural to wonder if the integral extreme points of $P(G, \gamma)$ have any combinatorial interpretation when G is an arbitrary DAG and γ is an arbitrary topological ordering of G. The following lemma shows that integrality of $P(G, \gamma)$ is useful only when G is a *permutation* DAG and γ is an *umbrella-free* topological ordering of G.

▶ **Lemma 19.** *Let G be a DAG and γ be a topological ordering of G. Then, $P(G, \gamma)$ is non-empty if and only if G is a permutation DAG and γ is umbrella-free.*

Proof. The reverse direction follows from the correctness of `GreedyAssign` (Lemma 14). We focus on proving the forward direction. Let $\mathbf{x} \in P(G, \gamma)$ be a feasible point. It suffices to show that G is transitively closed and γ is umbrella-free.

First, assume for contradiction that G is not transitively closed. Then, there exist arcs $(u, v), (v, w) \in A$ such that the arc $(u, w) \notin A$. Since $\mathbf{x}$ is feasible, we have the following: (1) $x_{\gamma(v)} \leq x_{\gamma(u)}$; (2) $x_{\gamma(w)} \leq x_{\gamma(v)}$; and (3) $x_{\gamma(u)} \leq x_{\gamma(w)} - 1$. However, these inequalities do not admit any feasible solution, a contradiction.

Next, assume for contradiction that γ is not umbrella-free. Then, there exists a triple (u, v, w) such that $\gamma(u) < \gamma(v) < \gamma(w)$, and the arc $(w, u) \in A$, but the arcs $(v, u), (w, v) \notin A$. Since the point $\mathbf{x}$ is feasible, we have the following: $x_{\gamma(w)} \leq x_{\gamma(v)} - 1$; (2) $x_{\gamma(v)} \leq x_{\gamma(u)} - 1$; and (3) $x_{\gamma(u)} \leq x_{\gamma(w)}$. However, these inequalities do not admit any feasible solution, a contradiction. ◀

References

1 János Balogh, Cosmin Bonchiş, Diana Diniş, Gabriel Istrate, and Ioan Todinca. The heapability of finite partial orders. *Discrete Mathematics and Theoretical Computer Science*, 22(1), 2020.

2 Anne-Laure Basdevant, Lucas Gerin, Jean-Baptiste Gouéré, and Arvind Singh. From Hammersley's lines to Hammersley's trees. *Probability Theory and Related Fields*, pages 1–51, 2016.

3 Anne-Laure Basdevant and Arvind Singh. Almost-sure asymptotic for the number of heaps inside a random sequence. *Electronic Communications in Probability*, 23(17), 2018.

4 Cosmin Bonchiş, Gabriel Istrate, and Vlad Rochian. The language (and series) of Hammersley-type processes. In *Proceedings of the Eighth Conference on Machines Computation and Universality (MCU'18)*, volume 10881 of *Lecture Notes in Computer Science*, 2018.

5 John Byers, Brent Heeringa, Michael Mitzenmacher, and Georgios Zervas. Heapable sequences and subseqeuences. In *Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics*, ANALCO '11, pages 33–44, 2011.

6 Karthekeyan Chandrasekaran, Elena Grigorescu, Gabriel Istrate, Shubhang Kulkarni, Young-San Lin, and Minshen Zhu. The maximum binary tree problem. *In Proceedings of the 32nd European Symposium on Algorithms (ESA'20), to appear*, 2020. arXiv preprint: 1909.07915. URL: https://arxiv.org/pdf/1909.07915.pdf.

7 Rodney G Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer Verlag, 2012.

8 Martin Charles Golumbic. Chapter 7 - permutation graphs. In Martin Charles Golumbic, editor, *Algorithmic Graph Theory and Perfect Graphs*, pages 157 – 170. Academic Press, 1980.

9 Gabriel Istrate and Cosmin Bonchiş. Partition into heapable sequences, heap tableaux and a multiset extension of Hammersley's process. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15), Ischia, Italy*, volume 9133 of *Lecture Notes in Computer Science*, pages 261–271. Springer, 2015.

10 Gabriel Istrate and Cosmin Bonchiş. Heapability, interactive particle systems, partial orders: Results and open problems. In *Proceedings of the 18th International Conference on Descriptional Complexity of Formal Systems (DCFS'2016), Bucharest, Romania*, volume 9777 of *Lecture Notes in Computer Science*, pages 18–28. Springer, 2016.

11 A. Pnueli, A. Lempel, and S. Even. Transitive orientation of graphs and identification of permutation graphs. *Canadian Journal of Mathematics*, 23(1):160–175, 1971.

12 Jaclyn Porfilio. A combinatorial characterization of heapability. Master's thesis, Williams College, 2015.

13 Dan Romik. *The surprising mathematics of longest increasing subsequences*. Cambridge University Press, 2015.