Parameterized Algorithms for String Matching to DAGs: Funnels and Beyond

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

─── Abstract ───

The problem of String Matching to Labeled Graphs (SMLG) asks to find all the paths in a labeled graph $G = (V, E)$ whose spellings match that of an input string $S \in \Sigma^m$. SMLG can be solved in quadratic $O(m|E|)$ time [Amir et al., JALG 2000], which was proven to be optimal by a recent lower bound conditioned on SETH [Equi et al., ICALP 2019]. The lower bound states that no strongly subquadratic time algorithm exists, even if restricted to directed acyclic graphs (DAGs).

In this work we present the first parameterized algorithms for SMLG on DAGs. Our parameters capture the topological structure of G. All our results are derived from a generalization of the Knuth-Morris-Pratt algorithm [Park and Kim, CPM 1995] optimized to work in time proportional to the number of prefix-incomparable matches.

To obtain the parameterization in the topological structure of G, we first study a special class of DAGs called funnels [Millani et al., JCO 2020] and generalize them to k-funnels and the class $\mathcal{ST}_k$. We present several novel characterizations and algorithmic contributions on both funnels and their generalizations.

1   Introduction

Given a labeled graph $G = (V, E)$[1] and a string S of length m over an alphabet Σ of size σ, the problem of *String Matching to Labeled Graph (SMLG)* asks to find all paths in G spelling S in their characters; such paths are known as *occurrences* or *matches* of S in G. This problem is a generalization of the classical *string matching (SM)* to a text T of length n, which can be encoded as an SMLG instance with a path labeled with T. Labeled graphs are present in many areas such as information retrieval [28, 73, 14], graph databases [11, 10, 75, 16] and bioinformatics [27], and SMLG is a primitive operation to locate information on them.

It is a textbook result [2, 29, 81] that the classical SM can be solved in linear $O(n + m)$ time. For example, the well-known *Knuth-Morris-Pratt* algorithm (KMP) [60] preprocesses S and then scans T while maintaining the longest matching prefix of S. However, for SMLG a recent result [12, 34] shows that there is no strongly subquadratic $O(m^{1-\epsilon}|E|), O(m|E|^{1-\epsilon})$ time algorithm unless the *strong exponential time hypothesis (SETH)* fails, and the most

---

[1] We consider the case where each vertex is labeled with a single character from Σ.

efficient current solutions [8, 72, 77, 52] match this bound, thus being optimal in this sense. Moreover, these algorithms solve the approximate version of SMLG (errors in S only) showing that both problems are equally hard under SETH, which is not the case for SM [13].

The history of (exact) SMLG. SMLG can be traced back to the publications of Manber and Wu [67] and Dubiner et al. [33] where the problem is defined for the first time, and solved in linear time on directed trees by using an extension of KMP. Later Akutsu [4] used a sampling on V and a suffix tree of S to solve the problem on (undirected) trees in linear time and Park and Kim [74] obtained an $O(N + m|E|)^2$ time algorithm for directed acyclic graphs (DAGs) by extending KMP on a topological ordering of G (we call this the *DAG algorithm*). Finally, Amir et al. [8] showed an algorithm with the same running time for general graphs with a simple and elegant idea that was later used to solve the approximate version [77, 52], and that has been recently generalized as the labeled product [78]. The lower bound of Equi et al. [34] shows that the problem remains quadratic (under SETH) even if the problem is restricted to deterministic DAGs with vertices of two kinds: indegree at most 1 and outdegree 2, and indegree 2 and outdegree at most 1 [34, Theorem 1], or if restricted to undirected graphs with degree at most 2 [34, Theorem 2]. Furthermore, they show how to solve the remaining cases (in/out-trees whose roots can be connected by a cycle) in linear time by an extension of KMP. Later they showed [35, 36] that the quadratic lower bound holds even when allowing polynomial indexing time.

An (important) special case. Gagie et al. [42] introduced *Wheeler graphs* as a generalization of prefix sortable techniques [38, 45, 37, 22, 83] applied to labeled graphs. On Wheeler graphs, SMLG can be solved in time $O(m \log |E|)$ [42] after indexing, however, it was shown that the languages recognized by Wheeler graphs (intuitively the set of strings they encode) are very restrictive [5, 6]. Later, Cotumaccio and Prezza [31] generalized Wheeler graphs to *p-sortable graphs*, capturing every labeled graph by using the parameter p: the minimum width of a colex relation over the vertices of the graph. On p-sortable graphs, SMLG can be solved in time $O(mp^2 \log (p\sigma))$ after indexing, however, the problems of deciding if a labeled graph is Wheeler or p-sortable are NP-hard [44]. In a recent work, Cotumaccio [30] defined *q-sortable graphs* as a relaxation of p-sortable ($q < p$), which can be indexed in $O(|E|^2 + |V|^{5/2})$ time but still solve SMLG in time $O(mq^2 \log (q\sigma))$.

1.1 Our results

We present parameterized algorithms for SMLG on DAGs. Our parameters capture the topological structure of G. These results are related to the line of research "FPT inside P" [43, 26, 40, 61, 1, 23, 25, 24, 66, 64] of finding parameterizations for polynomially-solvable problems.

All our results are derived from a new version of the DAG algorithm [74], which we present in Section 3. Our algorithm is optimized to only carry *prefix-incomparable* matches (Definition 3) and process them in time proportional to their size (Lemma 5 further optimized in Lemma 6 and Theorem 7). Prefix-incomparable sets suffice to capture all prefix matches

---

[2] N is the total length of the labels in G in a more general version of the problem where vertices are labeled with strings.

of S ending in a vertex v (Definition 4). By noting that the size of prefix-incomparable sets is upper-bounded by the structure of S (Lemma 27 in Appendix A), we obtain a parameterized algorithm (Theorem 28 in Appendix A) that beats the DAG algorithm on periodic strings.

To obtain the parameterization on the topological structure of the graph we first study and generalize a special class of DAGs called *funnels* in Section 4.

Funnels. Funnels are DAGs whose source-to-sink paths contain a *private* edge that is not used by any other source-to-sink-path. Although more complex that in/out-forests, their simplicity has allowed to efficiently solve problems that remain hard even when the input is restricted to DAGs, including: DAG partitioning [70], k-linkage [70], minimum flow decomposition [57, 56], a variation of network inhibition [62] and SMLG (this work). Millani et al. [70] showed that funnels can also be characterized by a partition into an in-forest plus an out-forest (the *vertex partition* characterization), or by the absence of certain forbidden paths (the *forbidden path* characterization), and propose how to find a minimal forbidden path in quadratic $O(|V|(|V| + |E|))$ time and a recognition algorithm running in $O(|V| + |E|)$ time. They used the latter to develop branching algorithms for the NP-hard problems of vertex and edge deletion distance to a funnel, obtaining a fix-parameter quadratic solution. Analogous to the minimum feedback set problem [54], the vertex (edge) deletion distance to a funnel problem asks to find the minimum number of vertices (edges) that need to be removed from a graph so that the resulting graph is a funnel.

We propose three (new) linear time recognition algorithms of funnels (Section 4.1), each based on a different characterization, improving the running time of the branching algorithm to parameterized linear time (see Appendix B). We generalize funnels to k-funnels by allowing private edges to be shared by at most k source-to-sink paths (Definitions 14 and 15). We show how to recognize them in linear time (Lemma 16) and find the minimum k for which a DAG is a k-funnel (Corollary 17 and Lemma 18). We then further generalize k-funnels to the class of DAGs $\mathcal{ST}_k$ (Definition 20 and Lemma 21), which (unlike k-funnels for $k > 1$, see Figure 2) can be characterized (and efficiently recognized, see Lemma 22) by a partition into a graph of the class $\mathcal{S}_k$ (generalization of out-forest, see Definition 19) and a graph of the class $\mathcal{T}_k$ (generalization of in-forest, see Definition 19).

We obtain our parameterized results in Section 5 by noting that, analogous to the fact that in KMP we only need the longest prefix match, in the DAG algorithm we can bound the size of the prefix-incomparable sets by the number of paths from a source or the number of paths to a sink, $\mu_s(v)$ and $\mu_t(v)$, respectively (Lemma 23).

▶ **Theorem 1.** *Let $G = (V, E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k + \sigma m)$, where $k = \min(\max_{v \in V} \mu_s(v), \max_{v \in V} \mu_t(v))$.*

In particular, this implies linear time algorithms for out-forests and in-forests, and for every DAG in $\mathcal{S}_k$ or $\mathcal{T}_k$ for constant k. Finally, we solve the problem on DAGs in $\mathcal{ST}_k$ (thus also in k-funnels), by using the vertex partition characterization of $\mathcal{ST}_k$ (Lemma 22), solving the matches in each part separately with Theorem 1, and resolve the matches crossing from one part to the other with a precomputed data structure (Lemma 26).

▶ **Theorem 2.** *Let $G = (V, E)$ be a DAG, $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k^2 + m^2)$, where $k = \max_{v \in V}(\min(\mu_s(v), \mu_t(v)))$.*

2   Preliminaries

We work with a (directed) graph $G = (V, E)$, a function $\ell : V \to \Sigma$ labeling the vertices of G with characters from a finite alphabet Σ of size σ, and a sequence $S[1..m] \in \Sigma^m$.

Graphs.   A graph $H = (V_H, E_H)$ is said to be a *subgraph* of G if $V_H \subseteq V$ and $E_H \subseteq E$. If $V' \subseteq V$, then $G[V']$ is the subgraph *induced by* V', defined as $G[V'] = (V', \{(u, v) \in E : u, v \in V'\})$. We denote $G^r = (V, E^r)$ to be the *reverse* of G ($E^r = \{(v, u) \mid (u, v) \in E\}$). For a vertex $v \in V$ we denote by N_v^- (N_v^+) the set of *in(out)-neighbors* of v, and by $d_v^- = |N_v^-|$ ($d_v^+ = |N_v^+|$) its *in(out)degree*. A *source (sink)* is a vertex with zero in(out)degree. The *edge contraction* of $(u, v) \in E$ is the graph operation that removes (u, v) and merges u and v. A *path* P is a sequence $v_1, \ldots, v_{|P|}$ of different vertices of V such that $(v_i, v_{i+1}) \in E$ for every $i \in \{1, \ldots, |P| - 1\}$. We say that P is *proper* if $|P| \geq 2$, a *cycle* if $(v_{|P|}, v_1) \in E$, and *source-to-sink* if v_1 is a source and $v_{|P|}$ is a sink. We say that $u \in V$ *reaches* $v \in V$ if there is a path from u to v. If G does not have cycles it is called *directed acyclic graph* (DAG). A *topological ordering* of a DAG is a total order $v_1, \ldots, v_{|V|}$ of V such that for every $(v_i, v_j) \in E$, $i < j$. It is known [53, 84] how to compute a topological ordering in $O(|V| + |E|)$ time, and we assume one $(v_1, \ldots, v_{|V|})$ is already computed if G is a DAG[3]. An *out(in)-forest* is a DAG such that every vertex has in(out)degree at most one, if it has a unique source (sink) it is called an *out(in)-tree*. The *label* of a path $P = v_1, \ldots, v_{|P|}$ is the sequence of the labels of its vertices, $\ell(P) = \ell(v_1) \ldots \ell(v_{|P|})$.

Strings.   We say that S has a *match* in (G, ℓ) if there is a path whose label is equal to S, every such path is an *occurrence* of S in (G, ℓ). We denote $S[i..j]$ (also $S[i]$ if $i = j$, and the empty string if $j < i$) to be the substring of S between position i and j (both inclusive), we say that it is *proper* if $i > 1$ or $j < m$, a *prefix* if $i = 1$ and a *suffix* if $j = m$. We denote S^r to be the reverse of S ($S^r[i] = S[m - i + 1]$ for $i \in \{1, \ldots, m\}$). A substring of S is called a *border* if it is a proper prefix and a proper suffix at the same time. The *failure function* of S, $f_S : \{1, \ldots, m\} \to \{0, \ldots, m\}$ (just f if S is clear from the context), is such that $f_S(i)$ is the length of the longest border of $S[1..i]$. We also use f_S to denote the in-tree $(\{0, \ldots, m\}, \{(i, f_S(i)) \mid i \in \{1, \ldots, m\}\})$, also known as the *failure tree* [46] of S. By definition, the lengths of all borders of $S[1..i]$ in decreasing order are $f_S(i), f_S^2(i), \ldots, 0$. The *matching automaton* of S, $A_S : \{0, \ldots, m\} \times \Sigma \to \{0, \ldots, m\}$, is such that $A_S(i, a)$ is the length of the longest border of $S[1..i] \cdot a$. It is known how to compute f_S in time $O(m)$ [60] and A_S in time $O(\sigma m)$ [2, 29, 81], and we assume they are already computed[4].

3   The DAG algorithm on prefix-incomparable matches

A key idea in our linear time parameterized algorithm is that of prefix-incomparable sets of the string S. We will show that one prefix-incomparable set per vertex suffices to capture all the matching information. See Figure 1 for an example of these concepts.

▶ **Definition 3** (Prefix-incomparable). *Let $S \in \Sigma^m$ be a string. We say that $i, j \in \{0, \ldots, m\}, i < j$ are prefix-incomparable (for S) if $S[1..i]$ is not a border of $S[1..j]$. We say that $B \subseteq \{0, \ldots, m\}$ is prefix-incomparable (for S) if for every $i < j \in B$, i and j are prefix-incomparable (for S).*

---

[3] Our algorithms run in $\Omega(|V| + |E|)$ time.
[4] Our algorithms run in $\Omega(\sigma m)$ time.
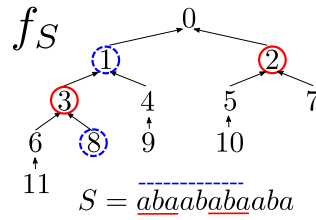
Figure 1 A string $S = abaababaaba$ and its failure tree f_S, with $w = |\{i \in \{0, \ldots, 11\} \mid \nexists j, f_S(j) = i\}| = |\{7, 8, 9, 10, 11\}| = 5$. On the string it is shown in segmented (blue) and solid (red) lines that prefix $S[1..3] = aba$ is a border of prefix $S[1..8] = abaababa$, which can also be seen in the tree since 3 is an ancestor (parent in this case) of 8. In the tree two sets are shown $B_1 = \{1, 8\}$ in segmented (blue) circles, which is prefix-comparable, and $B_2 = \{2, 3\}$ in solid (red) circles, which is prefix-incomparable. If $B_1 \cup B_2 \cup \{0\}$ is B_v for some $v \in V$, then $PI_v = \{2, 8\}$.

In our algorithm we will compute for each vertex v a prefix-incomparable set representing all the prefixes of S that match with a path ending in v. More precisely, if B_v is the set of all the prefixes of S that match with a path ending in v, then the algorithm will compute $PI_v \subseteq B_v$ such that PI_v is prefix-incomparable and for every $i \in B_v$ there is a $j \in PI_v$ such that i is ancestor of j. Note that such a set always exists and it is unique, it corresponds to the leaves of $f_S[B_v]$. To obtain a linear time parameterized algorithm we show how to compute PI_v from the sets PI_u, $u \in N_v^-$, in time parameterized by the size of these sets.

▶ **Definition 4** (B_v, PI_v). *Let $G = (V, E)$ be a DAG, $\ell : V \to \Sigma$ a labeling function, and $S \in \Sigma^m$ a string. For every $v \in V$ we define the sets:*
- $B_v = \{i \in \{0, \ldots, m\} \mid \exists P \text{ path of } G \text{ ending in } v \text{ and } \ell(P) = S[1..i]\}$[5]
- $PI_v \subseteq B_v$ *as the unique prefix-incomparable set such that for every $i \in B_v$ there is a $j \in PI_v$ such that $i = j$ or $S[1..i]$ is a border of $S[1..j]$*

▶ **Lemma 5.** *Let $G = (V, E)$ be a DAG, $v \in V$, $S \in \Sigma^m$ a string, f_S its failure tree and A_S its matching automaton. We can compute PI_v from PI_u for every $u \in N_v^-$ in time $O\left(w^2 \cdot d_v^-\right)$ or in time $O\left(\left(k_v := \sum_{u \in N_v^-} |PI_u|\right)^2\right)$, after $O(m)$ preprocessing time, where $w = |\{i \in \{0, \ldots, m\} \mid \nexists j, f_S(j) = i\}|$.*

Proof. We precompute constant time lowest common ancestor (LCA)[6] queries [3] of f_S in $O(m)$ time [41, 79, 20, 17, 18, 7, 39]. Note that with this structure we can check whether $i < j$ are prefix-incomparable in constant time $(LCA(i, j) < i)$.

If v is a source we have that either $B_v = PI_v = \{0\}$ if $\ell(v) \neq S[1]$ or $PI_v = \{1\}$ if $\ell(v) = S[1]$, otherwise we proceed as follows. To obtain the $O\left(k_v^2\right)$ time, we first append all the elements of every PI_u for $u \in N_v^-$ into a list $\mathcal{L}$ (of size k_v), then we replace every $i \in \mathcal{L}$ by $A_S(i, \ell(v))$, and finally we check (at most) every pair $i < j$ of elements of $\mathcal{L}$ and test (in constant time) if they are prefix-incomparable, if they are not we remove i from the list. After these $O(|\mathcal{L}|^2) = O(k_v^2)$ tests $\mathcal{L} = PI_v$.

To obtain the $O\left(w^2 \cdot d_v^-\right)$ time, we process the in-neighbors of v one by one and maintain a prefix-incomparable set representing the prefix matches incoming from the already processed in-neighbors. That is, we maintain a prefix-incomparable set PI', and when we process the next in-neighbor $u \in N_v^-$ we append all elements of PI' and $\{A_S(i, \ell(v)) \mid i \in PI_u\}$ into a

---

[5] Here we consider that the empty path always exists and its label is the empty string, thus $0 \in B_v$.
[6] $LCA(i, j)$ returns the lowest node in the tree f_S that is ancestor of both i and j.

list $\mathcal{L}'$ of size $O(w)$ (by Lemma 27), then we use the same quadratic procedure applied on $\mathcal{L}$ in time $O(w^2)$ to obtain the new PI'. After processing all in-neighbors in time $O(d_v^- \cdot w^2)$, we have $PI' = PI_v$. Next, we show the correctness of both procedures.

Let PI_v' be the result after applying one of the procedures explained before (that is the final state of PI' or $\mathcal{L}$), by construction PI_v' is prefix-incomparable. Now, consider $i \in B_v$ and a path P ending in v with $\ell(P) = S[1..i]$. If P is of length zero (only one vertex), then $i = 1$ and $\ell(P) = \ell(v) = S[1]$. Consider any $u \in N_v^-$, and any $j \in PI_u$ (there is at least one since $0 \in B_u$), this value is mapped to $A_S(j, \ell(v)) = A_S(j, S[1]) = j'$. By definition of the matching automaton, j' is the longest border of $S[1..j] \cdot S[1]$, thus $S[1]$ is a border of $S[1..j']$ or $j' = 1$, in both procedures j' can only be removed from PI_v' if a longer prefix contains $S[1..j']$ as a border and also $S[1]$. If P is a proper path and $i > 1$, consider the second to last vertex u of P. Note that $i - 1 \in B_u$, and thus there is $j \in PI_u$, such that $S[1..i-1]$ is a border of $S[1..j]$, this value is mapped to $j' = A_S(j, \ell(v)) = A_S(j, S[i])$, which is the length of the longest border of $S[1..j] \cdot S[i]$, but since $S[1..i]$ is also a border of $S[1..j] \cdot S[i]$, then $S[1..i]$ is also a border of $S[1..j']$ or $j' = i$. Again, in both procedures j' can only be removed from PI_v' if a longer prefix contains $S[1..j']$ as a border and also $S[1..i]$. Finally, we note that $PI_v' \subseteq B_v$ since every $i \in PI_v'$ corresponds to a match of $S[1..i]$ by construction.   ◄

We improve the dependency on w and k_v by replacing the quadratic comparison by sorting plus a linear time algorithm on the balanced parenthesis representation [51, 71] of f_S.

▶ **Lemma 6.** *We can obtain Lemma 5 in time $O(sort(w, m) \cdot d_v^-)$ or in time $O(sort(k_v, m))$, where $sort(n, p)$ is the time spent by an algorithm sorting n integers in the range $\{0, \ldots, p\}$.*

Proof. We compute the balanced parenthesis (BP) [51, 71] representation of the topology of f_S, that is, we traverse f_S from the root in preorder, appending an open parenthesis when we first arrive at a vertex, and a closing one when we leave its subtree. As a result we obtain a balanced parenthesis sequence of length $2(m + 1)$, where every vertex $i \in f_S$ is mapped to its open parenthesis position $open[i]$ and to its close parenthesis position $close[i]$, which can be computed and stored at preprocessing time. Note that in this representation, i is ancestor of j (and thus prefix-comparable) if and only if $open[i] \leq open[j] \leq close[i]$. As such, if we have a list of $O(k_v)$ (or $O(w)$) ($\mathcal{L}$ and $\mathcal{L}'$ from Lemma 5) values, we can compute the corresponding prefix-incomparable sets as follows.

First, we sort the list by increasing $open$ value, this can be done in $O(sort(k_v, m))$ (or $O(sort(w, m))$, since this sorting is equivalent to sort by increasing $open/2 \in \{0, \ldots, m\}$ value. Then, we process the list in the sorted order, if two consecutive values i and j in the order are prefix-comparable (that is, if $open[j] \leq close[i]$) then we remove i and continue to the next value j. At the end of this $O(k_v)$ (or $O(w)$) time processing we obtain the desired prefix-incomparable set.   ◄

If we use techniques for integer sorting [87, 90, 59] we can get $O(k_v \log \log m)$ (or $O(w \log \log m)$) time for sorting, however introducing m into the running time. We can solve this issue by using more advanced techniques [48, 9, 47] obtaining an $O(k_v \log \log k_v)$ (or $O(w \log \log w)$) time for sorting. However, we show that by using the suffix-tree [89, 68, 85] of S^r we can obtain a linear dependency on w and k_v.

▶ **Theorem 7.** *We can obtain the result of Lemma 5 in time $O(w \cdot d_v^-)$ or in time $O(k_v)$.*

Proof. We reuse the procedure of Lemma 6 but this time on the BP representation of the topology of the suffix-tree T_r of S^r, which has $O(m)$ vertices and can be built in $O(m)$ time [89, 68, 85]. Note that every suffix represented in T_r corresponds to a prefix of S (spelled

in the reverse direction). Moreover, $i \leq j$ are prefix-comparable if and only if the vertex representing i in T_r ($T_r[i]$) is a ancestor of $T_r[j]$, the same property as in f_S. Furthermore, if B is prefix-incomparable and $A(j, a) = j + 1$ for every $j \in B$, then the positions of the vertices in $A(B, a)$ in T_r follow the same order as the ones in B, since the suffix-tree is lexicographically sorted.

Now, we show how to obtain the prefix-incomparable set representing $A(PI_u, \ell(v))$ in $|PI_u|$ time assuming that PI_u is sorted by increasing (*open*) position in T_r.

We first separate PI_u into the elements $i \in M$ with $S[i + 1] = \ell(v)$ and $i \in E$ with $S[i + 1] \neq \ell(v)$ (in the same relative order as in PI_u, which is supposed to be in increasing order). Since M is prefix-incomparable the positions of the vertices in $A(M, \ell(v))$ in T_r follow the same order as the ones in M. We then obtain the list E_u by applying $T_r[A(i, \ell(v)) - 1]$ for every $i \in E$ (if $A(i, \ell(v)) = 0$ we do not process i), and then for any pair of consecutive elements x before y in E_u such that $y \leq x$ we remove y from E_u, and repeat this until no further such inversion remains, thus obtaining an increasing list in E_u representing vertices in T_r. Next, since E_u is sorted we can obtain the list PI_E of prefix-incomparable elements representing E_u, and finally apply $A(PI_E, \ell(v))$ (which also follows an increasing order in T_r), merge it with $A(M, \ell(v))$, and compute the prefix-comparable elements of this merge.

The correctness of the previous procedure follows by the fact that if there is an inversion $y < x$ in E_u, then the prefix $A(j, \ell(v)) - 1$ represented by y in T_r is a border of the prefix $A(i, \ell(v)) - 1$ represented by x (and thus is safe to remove y). For this, first note that i appears before j in E, then $S[i..1] <_{lex} S[j..1]$, and since i is prefix-incomparable with j there is a $k \geq 1$ such that $S[i..i + k - 1] = S[j..j + k - 1]$ and $S[i + k] <_{lex} S[j + k]$. Then, since y appears before x in E_u, then $S[A(j, \ell(v)) - 1..1] <_{lex} S[A(i, \ell(v)) - 1..1]$, but since $A(i, \ell(v)) - 1$ is a border of i and $A(j, \ell(v)) - 1$ is a border of j, $S[A(j, \ell(v)) - 1..1]$ must be a prefix of $S[A(i, \ell(v)) - 1..1]$, and thus $S[1..A(j, \ell(v)) - 1]$ is a border of $S[1..A(j, \ell(v)) - 1]$.

The corollary is obtained by maintaining the PI_v sets sorted by position in T_r, and noting that the previous procedure runs in linear $O(|P_u|)$ time.   ◀

In Appendix A we show how to use Theorem 7 to derive a parameterized algorithm using parameter $w = |\{i \in \{0, \ldots, m\} \mid \nexists j, f_S(j) = i\}|$, improving on the classical DAG algorithm when S is a periodic string. Next, we will present our results on recognizing funnels and their generalization (Section 4), and how to use these classes of graphs and Theorem 7 to obtain parameterized algorithms using parameters related to the topology of the DAG (Section 5).

4   Funnels and beyond

Recall that funnels are DAGs whose source-to-sink paths have at least one *private* edge[7], that is, an edge used by only one source-to-sink path. More formally,

▶ **Definition 8** (Private edge). *Let $G = (V, E)$ be a DAG and $\mathcal{P}$ the set of source-to-sink paths of G. We say that $e \in E$ is* private *if $\mu(e) := |\{P \in \mathcal{P} \mid e \in P\}| = 1$. If $\mu(e) > 1$, we say that e is* shared.

▶ **Definition 9** (Funnel). *Let $G = (V, E)$ be a DAG and $\mathcal{P}$ the set of source-to-sink paths of G. We say that G is a funnel if for every $P \in \mathcal{P}$ there exists $e \in P$ such that e is private.*

Millani et al. [70] showed two other characterizations of funnels.

---

[7]   For the sake of simplicity, we assume that there are no isolated vertices, thus any source-to-sink path has at least one edge.

▶ **Theorem 10** ([70])**.** *Let $G = (V, E)$ be a DAG. The following are equivalent:*
1. *G is a funnel*
2. *There exists a partition $V = V_1 \dot\cup V_2$ such that $G[V_1]$ is an out-forest, $G[V_2]$ is an in-forest and there are no edges from V_2 to V_1*
3. *There is no path P such that its first vertex has more than one in-neighbor (a merging vertex) and its last vertex more than one out-neighbor (a forking vertex). Such a path is called forbidden*

They also gave an $O(|V| + |E|)$ time algorithm to recognize whether a DAG G is a funnel, and an $O(|V|(|V| + |E|))$ time algorithm to find a minimal forbidden path in a general graph, that is, a forbidden path that is not contained in another forbidden path.

4.1 Three (new) linear time recognition algorithms

We first show how to find a minimal forbidden path in time $O(|V| + |E|)$ in general graphs, improving on the quadratic algorithm of Millani et al. [70].

▶ **Lemma 11.** *Let $G = (V, E)$ be a graph. In $O(|V| + |E|)$ time, we can decide if G contains a forbidden path, and if one exists we report a minimal forbidden path.*

Proof. In the bioinformatics community minimal forbidden paths are a subset of *unitigs* and it is well known how to compute them in $O(|V| + |E|)$ time (see e.g. [55, 50, 58, 69]), here we include a simple algorithm for completeness. We first compute the indegree and outdegree of each vertex and check whether there exists a forbidden path of length zero or one, all in $O(|V| + |E|)$ time, in the process we also mark all vertices except the ones with unit indegree and outdegree. If no path is found we iterate over the vertices one last time. If the current vertex is not marked we extend it back and forth to the closest marked vertices and mark the vertices in these extensions, finally we check whether the first vertex is merging and the last forking. This last step takes $O(|V|)$ time in total. ◀

Lemma 11 provides our first linear time recognition algorithm and, as opposed to the algorithm of Millani et al. [70], it also reports a minimal forbidden path given a general graph. Moreover, in Appendix B, we show that Lemma 11 provides a linear time parameterized algorithm for the NP-hard (and inapproximable) problem of deletion distance of a general graph to a funnel [63, 70]. Millani et al. [70] solved this problem in (parameterized) quadratic time and in (parameterized) linear time only if the input graph is a DAG.

Next, we show another linear time recognition algorithm, which additionally finds the partition $V = V_1 \dot\cup V_2$ from Theorem 10. Finding such a partition will be essential for our solution to SMLG. From now we will assume that the input graph is a DAG since this condition can be checked in linear time [53, 84].

▶ **Lemma 12.** *Let $G = (V, E)$ be a DAG. We can decide in $O(|V| + |E|)$ time whether G is a funnel. Additionally, if G is a funnel, the algorithm reports a partition $V = V_1 \dot\cup V_2$ such that $G[V_1]$ is an out-forest, $G[V_2]$ is an in-forest and there are no edges from V_2 to V_1.*

Proof. We start a special BFS traversal from all the source vertices of G. The traversal only adds vertices to the BFS queue if they have not been previously visited (as a typical BFS traversal) and if its indegree is at most one. After the search we define the partition V_1 as the set of vertices visited during the traversal and $V_2 = V \setminus V_1$. Finally, we report the previous partition if there are no edges from V_2 to V_1, and if every vertex of V_2 has outdegree at most one. All these steps run in time $O(|V| + |E|)$.

Note that if the algorithm reports a partition, then this satisfies the required conditions to be a funnel ($G[V_1]$ is an out-forest since every vertex visited in the traversal has indegree at most one). Moreover, if G is a funnel, we prove that V_2 is an in-forest and that there are no edges from V_2 to V_1. For the first, suppose by contradiction that there is a vertex $v \in V_2$ with $d_v^+ > 1$, since every vertex is reached by some source in a DAG then there is a $u \in V_2$ with $d_u^- > 1$ (a vertex that was not added to the BFS queue) that reaches v, implying the existence of a forbidden path in G, a contradiction. Finally, there cannot be edges from V_2 to V_1 since the indegree (in G) of vertices of V_1 is at most one and its unique (if any) in-neighbor is also in V_1 by construction. ◀

Next, we present another characterization of funnels based on the structure of private/shared edges of the graph, which can be easily obtained by manipulating the original Definition 9.

▶ **Definition 13** (Funnel). *Let $G = (V, E)$ be a DAG. We say that G is a funnel if there is no source-to-sink path using only shared edges.*

As such, another approach to decide whether a DAG G is a funnel is to compute $\mu(e)$ for every $e \in E$ and then perform a traversal that only uses shared edges. Computing the number of source-to-sink paths containing e, that is $\mu(e)$, can be done by multiplying the number of source-to-e paths, $\mu_s(e)$, by the number of e-to-sink paths, $\mu_t(e)$, each of which can be computed in $O(|V| + |E|)$ time for all edges. The solution consists of a dynamic program on a topological order (and reverse topological order) of G with the following recurrences.

$$
\begin{aligned}
\mu_s(u) &= \mathbb{1}_{d_u^- = 0} + \sum_{u' \in N_u^-} \mu_s(u') \\
\mu_t(v) &= \mathbb{1}_{d_v^+ = 0} + \sum_{v' \in N_v^+} \mu_t(v') \\
\mu((u, v)) &= \mu_s(u) \cdot \mu_t(v)
\end{aligned}
\tag{1}
$$

Where $\mathbb{1}_A$ is the characteristic function evaluating to 1 if A is true and to 0 otherwise. It is simple to observe that the previous dynamic programs can be computed in time $O(|V| + |E|)$ each and that for every $e = (u, v) \in E, \mu_s(e) = \mu_s(u)$ and $\mu_t(e) = \mu_t(v)$. By simplicity, in the following we will use $\mu_s(e)$ and $\mu_s(u)$ (also $\mu_t(e)$ and $\mu_t(v)$) interchangeably. The previous algorithm assumes constant time arithmetic operations on numbers up to $\max_{e \in E} \mu(e)$, which can be $O(2^{|V|})$. To avoid this issue, we note that it is not necessary to compute $\mu(e)$, but only to verify that $\mu(e) > 1$. As such, we can recognize shared edges as soon as we identify that $\mu(e) > 1$, that is whenever $\mu_s(e)$ or $\mu_t(e)$ is greater than one in their respective computation. A formal description of this algorithm can be found in Lemma 16.

4.2 Generalizations of funnels

To generalize funnels we will allow source-to-sink paths to use only shared edges, but require to have at least one edge shared by at most k different source-to-sink paths.

▶ **Definition 14** (k-private edge). *Let $G = (V, E)$ be a DAG. We say that $e \in E$ is k-private if $\mu(e) \leq k$. If $\mu(e) > k$ we say that e is k-shared.*

▶ **Definition 15** (k-funnel). *Let $G = (V, E)$ be a DAG. We say that G is a k-funnel if there is no source-to-sink path using only k-shared edges.*

The next algorithm is a generalization of the last algorithm in Section 4.1 to decide if a DAG is a k-funnel. It assumes constant time arithmetic operations on numbers up to k.

▶ **Lemma 16.** *We can decide if a DAG $G = (V, E)$ is a k-funnel in $O(|V| + |E|)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(k)$.*

Proof. We process the vertices in a topological ordering and use Equation (1) to compute $\mu_s(e)$ in one pass, $\mu_t(e)$ in another pass and $\mu(e)$ in a final pass. To avoid arithmetic operations with numbers greater than k, we mark the edges having μ_s and μ_t greater than k as k-shared during the computations of μ_s, μ_t. Note that if $\mu_s(e) > k$ or $\mu_t(e) > k$ then $\mu(e) > k$. As such, before computing $\mu_s(e)$ ($\mu_t(e)$) we check if some of the edges from (to) the in(out)-neighbors is marked as k-shared. If that is the case we do not compute $\mu_s(e)$ ($\mu_t(e)$) and instead mark e as k-shared, otherwise we compute the respective sum of Equation (1), and if at some point the cumulative sum exceeds k we stop the computation and mark e as k-shared. Finally, we find all k-shared edges as the marked plus the unmarked with $\mu(e) = \mu_s(e) \cdot \mu_t(e) > k$, perform a traversal only using k-shared edges, and report that G is not a k-funnel if there is a source-to-sink path using only k-shared edges in time $O(|V| + |E|)$. ◀

We can use the previous result and exponential search [19, 15] to find the minimum k such that a DAG is a k-funnel.

▶ **Corollary 17.** *Let $G = (V, E)$ be a DAG. We can find the minimum k such that G is a k-funnel in $O((|V| + |E|) \log k)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(k)$.*

Assuming constant time arithmetic operations on numbers up to $\max_{e \in E} \mu(e)$ the problem is solvable in linear time by noting that the answer is equal to the weight of a widest path.

▶ **Lemma 18.** *Let $G = (V, E)$ be a DAG. We can find the minimum k such that G is a k-funnel in $O(|V| + |E|)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(\max_{e \in E} \mu(e))$.*

Proof. We compute $\mu(e)$ for every $e \in E$ by using the dynamic programming algorithm specified by Equation (1) on a topological ordering of G. Since constant time arithmetic operations are assumed for numbers up to $\max_{e \in E} \mu(e)$, the previous computation takes linear time. Then, we compute the weight of a source-to-sink path P maximizing $\min_{e \in P} \mu(e)$, and report this value. This problem is known as the widest path problem [76, 82, 65, 86, 80] and it can be solved in linear time on DAGs [88, 49] by a dynamic program on a topological order of the graph. By completeness, we show a dynamic programming recurrence to compute $W[e]$, the weight of a source-to-e path P maximizing $\min_{e' \in P} \mu(e')$.

$$W[e = (u, v)] = \mu(e) \cdot \mathbb{1}_{d_u^- = 0} + \sum_{u' \in N_u^-} \min(W[(u', u)], \mu((u', u)))$$

Finally, note that if we denote w to the weight of a widest path, then there is a source-to-sink path using only $w - 1$-shared edges, G is not $w - 1$-funnel. Moreover, there cannot be a source-to-sink path using only w-shared edges, since such a path would contradict w being the weight of a widest path. As such, w is the minimum k such that G is k-funnel. ◀

We now define three classes of DAGs closely related to k-funnels.

▶ **Definition 19.** *We say that a DAG $G = (V, E)$ belongs to the class $\mathcal{S}_k$ ($\mathcal{T}_k$) if for every $v \in V$, $\mu_s(v)$ ($\mu_t(v)$) $\leq k$.*
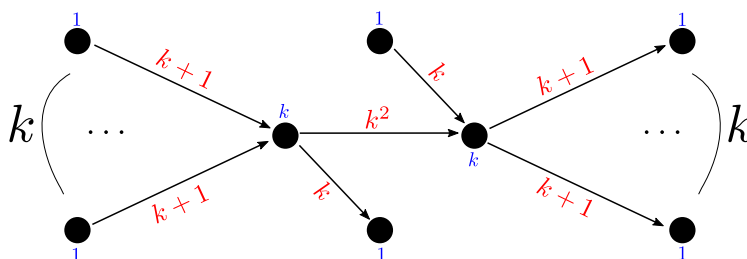
Figure 2 A DAG in $\mathcal{ST}_k$ that is not a k-funnel, for every $k > 1$. The central edge is a forbidden path whose first vertex has indegree k and outdegree 2, and last vertex has indegree 2 and outdegree k, the rest of the edges have either a source tail or a sink head. The blue label next to each vertex v corresponds to $\min(\mu_s(v), \mu_t(v))$, since the maximum of these labels is k the graph belongs to $\mathcal{ST}_k$. The red label next to each edge e corresponds to $\mu(e)$, since there is a source-to-sink path with no k-private edge the graph is not a k-funnel.

▶ **Definition 20.** *We say that a DAG $G = (V, E)$ belongs to the class $\mathcal{ST}_k$ if for every $v \in V$, $\mu_s(v) \leq k$ or $\mu_t(v) \leq k$.*

▶ **Lemma 21.** $\mathcal{S}_k, \mathcal{T}_k \subseteq k\text{-funnels} \subseteq \mathcal{ST}_k$.

Proof. We first prove that $\mathcal{S}_k, \mathcal{T}_k \subseteq k$-funnels. Consider $G \in \mathcal{S}_k$ ($G \in \mathcal{T}_k$), and take any source-to-sink path P of G. Let (u, v) be the last (first) edge of P, then by Equation (1) $\mu((u, v)) = \mu_s(u) \cdot \mu_t(v)$, but since $\mu_t(v) = 1$ (v is a sink) and $\mu_s(u) \leq k$ ($G \in \mathcal{S}_k$) (analogously, $\mu_s(u) = 1$ and $\mu_t(v) \leq k$), then $\mu((u, v)) \leq k$, and thus (u, v) is a k-private edge. To prove that k-funnels $\subseteq \mathcal{ST}_k$, suppose that G is a k-funnel, and by contradiction that there exists $v \in V$ with $\mu_s(v), \mu_t(v) > k$. Consider any source-to-sink path P using v. Now, let (u, w) be any edge in P before (after) v, then $\mu_t(w) \geq \mu_t(v) > k$ ($\mu_s(u) \geq \mu_s(v) > k$), and thus $\mu((u, w)) = \mu_s(u) \cdot \mu_t(w) > k$. As such, P does not have a k-private edge, a contradiction. ◀

For $k = 1$, $\mathcal{S}_1$ describes out-forests and $\mathcal{T}_1$ in-forests, thus being more restrictive than funnels. Moreover, we note that the in(out)-star of $k + 2$ vertices, that is $k + 1$ vertices pointing to a sink (pointed from a source), $\notin \mathcal{S}_k$ ($\mathcal{T}_k$), but this graph is a funnel. On the other hand, from the vertex partition characterization of funnels (Theorem 10 [70]) we have that $\mathcal{ST}_1 = (1\text{-})$funnels. However, for $k > 1$, the containment k-funnels $\subseteq \mathcal{ST}_k$ is strict (Figure 2).

By noting that the minimum k such that a DAG is in $\mathcal{S}_k$, $\mathcal{T}_k$ and $\mathcal{ST}_k$ is $\max_{v \in V} \mu_s(v)$, $\max_{v \in V} \mu_t(v)$ and $\max_{v \in V} \min(\mu_s(v), \mu_t(v))$, respectively, we obtain the same results as in Lemmas 16 and 18 and Corollary 17 (with analogous assumptions on the cost of arithmetic operations) for recognition of $\mathcal{S}_k$, $\mathcal{T}_k$ and $\mathcal{ST}_k$.

Next, we prove that although the vertex partition characterization of funnels does not generalize to k-funnels, it does for the class $\mathcal{ST}_k$ and it can be found efficiently.

▶ **Lemma 22.** *Let $G = (V, E) \in \mathcal{ST}_k$ and k given as inputs. We can find, in $O(|V| + |E|)$ time, a partition $V = V_1 \dot{\cup} V_2$ such that $G[V_1] \in \mathcal{S}_k$, $G[V_2] \in \mathcal{T}_k$ and there are no edges from V_2 to V_1. Moreover, if such a partition of a DAG G exists, then $G \in \mathcal{ST}_k$.*

Proof. We set $V_1 = \{v \in V \mid \mu_s(v) \leq k\}$ and $V_2 = V \setminus V_1$. Note that finding V_1 takes linear time, since we can apply the algorithm described in Lemma 16 to compute the μ_s values (or decide that they are more than k)[8]. By construction we know that every $v \in V_1$ has

---

[8] Recall that this procedure assumes constant time arithmetic operations of numbers up to $\Theta(k)$.

$\mu_s(v) \leq k$, and since $G \in \mathcal{ST}_k$ also every $v \in V_2$ has $\mu_t(v) \leq k$, thus $G[V_1] \in \mathcal{S}_k$, $G[V_2] \in \mathcal{T}_k$. Suppose by contradiction that there exists $e = (u,v) \in E \cap (V_2 \times V_1)$. As such, $\mu_s(u) > k$, but since $\mu_s(u) \leq \mu_s(v)$, then $\mu_s(v) > k$, a contradiction. Finally, if such a partition exists then $\mu_s(v) \leq k$ for every $v \in V_1$ and $\mu_t(v) \leq k$ for every $v \in V_2$, and thus $G \in \mathcal{ST}_k$.   ◄

5   Parameterized algorithms: The DAG

The main idea to get the parameterized algorithms in this section is to bound the size of the PI_v sets by a topological graph parameter and use Lemma 5 and Theorem 7 to obtain a parameterized solution. As in the KMP algorithm [60] only one prefix-incomparable value suffices (the longest prefix match until that point), we show that $\mu_s(v)$ prefix-incomparable values suffice to capture the prefix matches up to v.

▶ **Lemma 23.** *Let $G = (V,E)$ be a DAG, $v \in V$, $\mathcal{P}_{sv}$ the set of source-to-v paths, $\ell : V \to \Sigma$ a labeling function, $S \in \Sigma^m$ a string, and PI_v as in Definition 4. Then, $|PI_v| \leq \mu_s(v)$.*

Proof. Since any path ending in v is the suffix of a source-to-v path we can write B_v as:

$$B_v = \bigcup_{P_{sv} \in \mathcal{P}_{sv}} B_{P_{sv}} := \{i \in \{0,\ldots,m\} \mid \exists P \text{ suffix of } P_{sv}, \ell(P) = S[1..i]\}$$

However, for every pair of values $i < j \in B_{P_{sv}}$, $S[1..i]$ is a border of $S[1..j]$ (it is a suffix since they are both suffixes of $\ell(P_{sv})$). As such, at most one value of $B_{P_{sv}}$ appears in PI_v, and then $|PI_v| \leq |\mathcal{P}_{sv}| = \mu_s(v)$.   ◄

This result directly implies a parameterized string matching algorithm to DAGs in $\mathcal{S}_k$.

▶ **Lemma 24.** *Let $G = (V,E) \in \mathcal{S}_k$, $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G,ℓ) in time $O(|V|k + |E| + \sigma m)$.*

Proof. We compute the matching automaton A_S in $O(\sigma m)$ time. Then, we process the vertices in topological order, and for each vertex v we compute PI_v, the unique prefix-incomparable set representing B_v (all prefix matches of S with paths ending in v). We proceed according to Lemma 5 and Theorem 7 in $O(m)$ preprocessing time plus $O(k_v)$ time per vertex. There is a match of S in (G,ℓ) if and only if any PI_v contains m. The claimed running time follows since $k_v = \sum_{u \in N_v^-} |PI_u| \leq \sum_{u \in N_v^-} \mu_s(u) \leq \mu_s(v) \leq k$, by Lemma 23, Equation (1) and since $G \in \mathcal{S}_k$.   ◄

A simple but interesting property about string matching to graphs is that we obtain the same problem by reversing the input (both the graph and the string), that is, S has a match in (G,ℓ) if and only if S^r has a match in G^r, ℓ. This fact, plus noting that $G \in \mathcal{S}_k$ if and only if $G^r \in \mathcal{T}_k$ gives the following corollary of Lemma 24.

▶ **Corollary 25.** *Let $G = (V,E) \in \mathcal{T}_k$, $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G,ℓ) in time $O(|V|k + |E| + \sigma m)$.*

With these two results and the fact that we can compute the minimum k such that a DAG is in $\mathcal{S}_k, \mathcal{T}_k$ in time $O((|V| + |E|) \log k)$ (see Corollary 17) we obtain our first algorithm parameterized by the topology of the DAG.

▶ **Theorem 1.** *Let $G = (V,E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G,ℓ) in time $O((|V| + |E|)k + \sigma m)$, where $k = \min(\max_{v \in V} \mu_s(v), \max_{v \in V} \mu_t(v))$.*

Our final result is a parameterized algorithm for DAGs in $\mathcal{ST}_k$ (in particular for k-funnels). We note that the algorithm of Corollary 25 computes PI_v for S^r for every vertex in G^r. Recall that PI_v represents all the prefix matches of S^r with paths ending in v in G^r. In other words, it represents all suffix matches of S with paths starting in v in G. For clarity, let us call this set SI_v. The main idea of the algorithm for $\mathcal{ST}_k$ is to use Lemma 22 to find a partitioning $V = V_1 \dot\cup V_2$ into $\mathcal{S}_k$ and $\mathcal{T}_k$, use Lemma 24 and Corollary 25 to search for matches within each part and also to compute PI_v for every $v \in V_1$ and SI_v for every $v \in V_2$, and finally, to find matches using the edges from V_1 to V_2. The last ingredient of our algorithm consists of preprocessing the answers to the last type of matches.

▶ **Lemma 26.** *Let $G = (V, E)$ a DAG, $(u, v) \in E$, $\ell : V \to \Sigma$ a labeling function, $S \in \Sigma^m$ a string and PI_u and SI_v as in Definition 4. We can decide if there is a match of S in (G, ℓ) using (u, v) in $O(|PI_u| \cdot |SI_v|)$ time, after $O(m^2)$ preprocessing time.*

Proof. We precompute a boolean table PS of $m \times m$ entries, such that $PS[i, j]$ is `true` if there is a length i' of a (non-empty) border of $S[1..i]$ (or $i' = i$) and a length j' of a (non-empty) border of $S[m - j + 1..m]$ (or $j' = j$) such that $i' + j' = m$, and `false` otherwise. This table can be computed by dynamic programming in $O(m^2)$ time as follows.

$$PS[i,j] = \begin{cases} \texttt{false} & \text{if } i + j < m \vee i = 0 \vee j = 0 \\ i + j = m \vee PS[i, f_{S^r}(j)] \vee PS[f_S(i), j] & \text{otherwise} \end{cases}$$

We then use this table to test every $PS[i, j]$ with $i \in PI_u, j \in SI_v$ and report a match if any of these table entries is `true`, in total $O(|PI_u| \cdot |SI_v|)$ time.

Since every match of S using (u, v) must match a prefix $S[1..i]$ with a path ending in u and a suffix $S[i + 1..m]$ with a path starting in v, the previous procedure finds it (if any). ◄

▶ **Theorem 2.** *Let $G = (V, E)$ be a DAG, $\ell : V \to \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k^2 + m^2)$, where $k = \max_{v \in V}(\min(\mu_s(v), \mu_t(v)))$.*

Proof. We first compute the minimum k such that the input DAG is in $\mathcal{ST}_k$ in time $O((|V| + |E|) \log k)$ (see Corollary 17). Then, we obtain the partition of G into $G[V_1] \in \mathcal{S}_k, G[V_2] \in \mathcal{T}_k$ and no edges from V_2 to V_1. We then search matches within $G[V_1]$ and $G[V_2]$ in time $O(|V|k + |E| + \sigma m)$ (Lemma 24 and Corollary 25) and we also keep PI_u for every $u \in V_1$ and SI_v for every $v \in V_2$. Finally, we process the matches using the edges (u, v) with $u \in V_1, v \in V_2$ in total $O(|E|k^2 + m^2)$ time (Lemma 26) since $O(|PI_u| \cdot |SI_v|) = O(k^2)$. ◄

6 Conclusions

In this paper we introduced the first parameterized algorithms for matching a string to a labeled DAG, a problem known (under SETH) to be quadratic even for a very special type of DAGs. Our parameters depend on the structure of the input DAG.

We derived our results from a generalization of KMP to DAGs using prefix-incomparable matches, which allowed us to bound the running time to parameterized linear. Further improvements on the running time of our algorithms remain open: is it possible to get rid of the automaton? or to combine prefix-incomparable and suffix-incomparable matches in better than quadratic (either in the size of the sets or the string)? (e.g. with a different tradeoff between query and construction time of the data structure answering these queries) and is there a (conditional) lower bound to combine these incomparable sets? (see e.g. [21]). Another

interesting question with practical importance is whether our parameterized approach can be extended to string labeled graphs with (unparameterized) linear time in the total length of the strings or extended to counting and reporting algorithms in linear time in the number of occurrences.

We also presented novel algorithmic results on funnels as well as generalizations of them. These include linear time recognition algorithms for their different characterizations, which we showed useful for the string matching problem but hope that can also help in other graph problems. We also showed how to find the minimum k for which a DAG is a k-funnel or $\in \mathcal{ST}_k$ (assuming constant time arithmetic operations on numbers up to $O(k)$) using an exponential search, but it remains open whether there exists a linear time solution.

References

1 Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 377–391. SIAM, 2016.

2 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

3 Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–132, 1976.

4 Tatsuya Akutsu. A linear time pattern matching algorithm between a string and a tree. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM 1993)*, pages 1–10. Springer, 1993.

5 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, pages 911–930. SIAM, 2020.

6 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021.

7 Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

8 Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.

9 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

10 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):1–40, 2017.

11 Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

12 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2016)*, pages 457–466. IEEE, 2016.

13 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. `doi:10.1137/15M1053128`.

14 Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

15 Ricardo Baeza-Yates and Alejandro Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pages 45–61. Springer, 2010.

16 Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2013)*, pages 175–188, 2013.

17 Michael A Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN 2000)*, pages 88–94. Springer, 2000.

18 Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

19 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(SLAC-PUB-1679), 1976.

20 Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

21 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132, pages 1–15. Schloss Dagstuhl-Leibniz Center for Informatics, 2019.

22 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics (WABI 2012)*, pages 225–235. Springer, 2012.

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

24 Manuel Caceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Minimum path cover in parameterized linear time. *arXiv preprint arXiv:2211.09659*, 2022.

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

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

27 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.

28 Jeff Conklin. Hypertext: An introduction and survey. *computer*, 20(09):17–41, 1987.

29 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

30 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *Proceedings of the 32nd Data Compression Conference (DCC 2022)*, pages 272–281. IEEE, 2022.

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

32 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

33 Moshe Dubiner, Zvi Galil, and Edith Magen. Faster tree pattern matching. *Journal of the ACM*, 41(2):205–213, 1994.

34 Massimo Equi, Roberto Grossi, Veli Mäkinen, Alexandru Tomescu, et al. On the complexity of string matching for graphs. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

35   Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In *Proceedings of the 47th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2021)*, pages 608–622. Springer, 2021.

36   Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 2023.

37   Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pages 184–193. IEEE, 2005.

38   Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE, 2000.

39   Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 36–48. Springer, 2006.

40   Fedor V Fomin, Daniel Lokshtanov, Saket Saurabh, Michał Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3):1–45, 2018.

41   Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

42   Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.

43   Archontia C Giannopoulou, George B Mertzios, and Rolf Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. *Theoretical Computer Science*, 689:67–95, 2017.

44   Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing Wheeler graphs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144, pages 51:1–51:16, 2019.

45   Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406, 2000.

46   Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1994)*, pages 697–704, 1994.

47   Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 602–608, 2002.

48   Yijie Han and Xiaojun Shen. Conservative algorithms for parallel and sequential integer sorting. In *Proceedgins of the 1st International Computing and Combinatorics Conference (COCOON 1995)*, pages 324–333. Springer, 1995.

49   Tzvika Hartman, Avinatan Hassidim, Haim Kaplan, Danny Raz, and Michal Segalov. How to split a flow? In *2012 Proceedings IEEE INFOCOM*, pages 828–836. IEEE, 2012.

50   Benjamin Grant Jackson. *Parallel methods for short read assembly*. PhD thesis, Iowa State University, 2009.

51   Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*, pages 549–554. IEEE Computer Society, 1989.

52   Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020.

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

54 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

55 John D Kececioglu and Eugene W Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1):7–51, 1995.

56 Shahbaz Khan, Milla Kortelainen, Manuel Cáceres, Lucia Williams, and Alexandru I Tomescu. Improving RNA assembly via safety and completeness in flow decompositions. *Journal of Computational Biology*, 2022.

57 Shahbaz Khan, Milla Kortelainen, Manuel Cáceres, Lucia Williams, and Alexandru I Tomescu. Safety and completeness in flow decompositions for RNA assembly. In *Proceedings of the 26th International Conference on Research in Computational Molecular Biology (RECOMB 2022)*, pages 177–192. Springer, 2022.

58 Carl Kingsford, Michael C Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):1–11, 2010.

59 David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983.

60 Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

61 Tomohiro Koana, Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. *Journal of Experimental Algorithmics*, 26:1–30, 2021.

62 Jonas Lehmann. The computational complexity of worst case flows in unreliable flow networks. B.S. thesis, Institute for Theoretical Computer Science, University of Lübeck, 2017.

63 Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *Proceedings of the 20th International Colloquium on Automata, Languages, and Programming (ICALP 1993)*, pages 40–51. Springer, 1993.

64 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022.

65 Thomas Magnanti, R Ahuja, and J Orlin. Network flows: theory, algorithms, and applications. *PrenticeHall, Upper Saddle River, NJ*, 1993.

66 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Transactions on Algorithms*, 15(2):1–21, 2019.

67 Udi Manber and Sun Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Advances In Structural And Syntactic Pattern Recognition*, pages 22–33. World Scientific, 1992.

68 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

69 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *Proceedings of the 7th International Workshop on Algorithms in Bioinformatics (WABI 2007)*, pages 289–301. Springer, 2007.

70 Marcelo Garlet Millani, Hendrik Molter, Rolf Niedermeier, and Manuel Sorge. Efficient algorithms for measuring the funnel-likeness of DAGs. *Journal of Combinatorial Optimization*, 39(1):216–245, 2020.

71 J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

72 Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

73 Jakob Nielsen. *Hypertext and hypermedia*. Academic Press Professional, Inc., 1990.

74 Kunsoo Park and Dong Kyue Kim. String matching in hypertext. In *Proceeding of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995)*, pages 318–329. Springer, 1995.

75   Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

76   Maurice Pollack. The maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.

77   Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv*, page 216127, 2017.

78   Nicola Rizzo, Alexandru I Tomescu, and Alberto Policriti. Solving string problems on graphs using the labeled direct product. *Algorithmica*, pages 1–26, 2022.

79   Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

80   Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social choice and Welfare*, 36(2):267–303, 2011.

81   Robert Sedgewick and Kevin Wayne. *Algorithms (4th edn)*. Addison-Wesley, 2011.

82   Nachum Shacham. Multicast routing of hierarchical data. In *[Conference Record] SUPER-COMM/ICC'92 Discovering a New World of Communications*, pages 1217–1221. IEEE, 1992.

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

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

85   Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

86   Ehsan Ullah, Kyongbum Lee, and Soha Hassoun. An algorithm for identifying dominant-edge metabolic pathways. In *2009 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Digest of Technical Papers*, pages 144–150. IEEE, 2009.

87   Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS 1975)*, pages 75–84. IEEE, 1975.

88   Benedicte Vatinlen, Fabrice Chauvet, Philippe Chrétienne, and Philippe Mahey. Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths. *European Journal of Operational Research*, 185(3):1390–1401, 2008.

89   Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973.

90   Dan E Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

A   A parameterized algorithm: The String

A simple property about prefix-incomparable sets is that their sizes are bounded by the number of prefixes that are not a border of other prefixes of the string, equivalently, the number of leaves in the failure function of the string.

▶ **Lemma 27.** *Let $S \in \Sigma^m$ be a string, f_S its failure function/tree, and $B \subseteq \{0, \ldots, m\}$ prefix-incomparable for S. Then, $|B| \leq w$ such that w is the number of leaves of f_S, equivalently $w := |\{i \in \{0, \ldots, m\} \mid \nexists j, f_S(j) = i\}|$.*

Proof. First note that $i < j$ are prefix-incomparable if and only if i is not ancestor of j in f_S. Suppose by contradiction that $|B| > w$, and consider the w leaf-to-root paths of f_S. Note that these w leaf-to-root paths cover all the vertices of f_S. By pigeonhole principle, there must be $i < j \in B$ in the same leaf-to-root path, that is i is ancestor of j, a contradiction. ◀

▶ **Theorem 28.** *Let $G = (V, E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \to \Sigma$ a labeling function, $S \in \Sigma^m$ a string and f_S its failure function. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)w + \sigma m)$, where $w = |\{i \in \{0, \ldots, m\} \mid \nexists j, f_S(j) = i\}|$.*

Proof. We compute the matching automaton A_S in $O(\sigma m)$ time. Then, we process the vertices in topological order, and for each vertex v we compute PI_v, the unique prefix-incomparable set representing B_v (all prefix matches of S with paths ending in v). We proceed according to Lemma 5 and Theorem 7 in $O(m)$ preprocessing time plus $O(w \cdot d_v^-)$ time per vertex, adding up to $O(w(|V| + |E|))$ time in total. There is a match of S in (G, ℓ) if and only if any PI_v contains m. ◀

We note that $w \leq m$, thus our algorithm is asymptotically as fast as the DAG algorithm, which runs in time $\Omega((|V| + |E|)m)$. However, we note that for w to be $o(m)$, a (very) long prefix of S must be a (highly) periodic string. To see this, consider the longest prefix $S[1..i]$ of S, such that there exists $j > i$ with $i = f_S(j)$. By definition, $S[1..i]$ is a border of $S[1..j]$, thus $S[k] = S[k + j - i]$ for $k \in \{1, \ldots, i\}$, that is $S[1..j]$ is a periodic string with period $j - i$. Finally, note that if $w = o(m)$, then $m - i \in o(m)$, and thus the period $j - i \in o(m)$.

B   A linear time parameterized algorithm for the distance problems

Millani et al. [70] gave an $O(|V|(|V| + |E|))$ time algorithm to find a minimal forbidden path in a general graph. They used this algorithm to design branching algorithms (see e.g. [32]) for the problems of finding maximum sized sets $V' \subseteq V, d_v := |V'|$ and $E' \subseteq E, d_e := |E'|$, such that $G[V']$ and (V, E') are funnels, known as vertex and edge distance to a funnel. It is know that (unless P = NP) there is an $\epsilon > 0$ such that there is no polynomial time $|V|^\epsilon$ approximation [63] for the vertex version nor $(1 + \epsilon)$ approximation [70] for the edge version. The authors [70] noted that if we consider a minimal forbidden path P of G of length $|P| > 1$, then the edges of P can be contracted until $|P| = 1$ without affecting the size of the solution. Moreover, they noted that if we consider such a P, two in-neighbors of the first vertex and two out-neighbors of the last[9], then V' must contain at least one of those 6 vertices and E' one of those 5 edges, deriving $O(6^{d_v}|V|(|V| + |E|))$ and $O(5^{d_e}|V|(|V| + |E|))$ time branching algorithms for each problem[10] [70, Corollary 1]. The authors also developed a more involved branching algorithm, only for the edge distance problem on DAG inputs, running in time $O(3^{d_e}(|V| + |E|))$ [70, Theorem 4].

By noting that minimal forbidden paths can be further contracted to length zero (one vertex) in the vertex distance problem, and that a minimal forbidden path can be found in time $O(|V| + |E|)$ (Lemma 11) we obtain the following result.

▶ **Theorem 29.** *Let $G = (V, E)$ be a graph. We can compute the vertex (edge) deletion distance to a funnel in time $O(5^d(|V| + |E|))$, where d is the deletion distance.*

Proof. We follow the branching approach as in [70, Corollary 1], but in the case of vertex distance we further contract the forbidden paths to length 0, the correctness of this step follows by noting that any solution containing two different vertices in a forbidden path is not minimum, since we still get a funnel by removing one of them (from the solution). As such, the number of recursive calls is ≤ 5 for both problems. Moreover, by Lemma 11, we can find a minimal forbidden path in time $O(|V| + |E|)$. ◀

---

[9] This structure is known as a *butterfly*.

[10] After removing forbidden paths all cycles are vertex-disjoint thus the rest of the problem can be solved by removing one vertex (edge) per cycle in one $O(|V| + |E|)$ time traversal.