Exact and Approximate Digraph Bandwidth

Pallavi Jain
Ben-Gurion University of the Negev, Beer-Sheva, Israel
pallavi@post.bgu.ac.il

Lawqueen Kanesh
The Institute of Mathematical Sciences, HBNI, India
lawqueen@imsc.res.in

William Lochet
University of Bergen, Norway
William.Lochet@uib.no

Saket Saurabh
The Institute of Mathematical Sciences, HBNI, India
saket@imsc.res.in

Roohani Sharma
The Institute of Mathematical Sciences, HBNI, India
saket@imsc.res.in

Abstract

In this paper, we introduce a directed variant of the classical Bandwidth problem and study it from the view-point of moderately exponential time algorithms, both exactly and approximately. Motivated by the definitions of the directed variants of the classical Cutwidth and Pathwidth problems, we define Digraph Bandwidth as follows. Given a digraph D and an ordering σ of its vertices, the *digraph bandwidth* of σ with respect to D is equal to the maximum value of $\sigma(v)-\sigma(u)$ over all arcs (u,v) of D going forward along σ (that is, when $\sigma(u) < \sigma(v)$). The Digraph Bandwidth problem takes as input a digraph D and asks to output an ordering with the minimum digraph bandwidth. The undirected Bandwidth easily reduces to Digraph Bandwidth and thus, it immediately implies that Directed Bandwidth is NP-hard. While an $\mathcal{O}^\star(n!)$[1] time algorithm for the problem is trivial, the goal of this paper is to design algorithms for Digraph Bandwidth which have running times of the form $2^{\mathcal{O}(n)}$. In particular, we obtain the following results. Here, n and m denote the number of vertices and arcs of the input digraph D, respectively.

- Digraph Bandwidth can be solved in $\mathcal{O}^\star(3^n \cdot 2^m)$ time. This result implies a $2^{\mathcal{O}(n)}$ time algorithm on sparse graphs, such as graphs of bounded average degree.
- Let G be the underlying undirected graph of the input digraph. If the treewidth of G is at most t, then Digraph Bandwidth can be solved in time $\mathcal{O}^\star(2^{n+(t+2)\log n})$. This result implies a $2^{n+\mathcal{O}(\sqrt{n}\log n)}$ algorithm for directed planar graphs and, in general, for the class of digraphs whose underlying undirected graph excludes some fixed graph H as a minor.
- Digraph Bandwidth can be solved in $\min\{\mathcal{O}^*(4^n \cdot b^n), \mathcal{O}^*(4^n \cdot 2^{b\log b\log n})\}$ time, where b denotes the optimal digraph bandwidth of D. This allow us to deduce a $2^{\mathcal{O}(n)}$ algorithm in many cases, for example when $b \leq \frac{n}{\log^2 n}$.
- Finally, we give a *(Single) Exponential Time Approximation Scheme* for Digraph Bandwidth. In particular, we show that for any fixed real $\epsilon > 0$, we can find an ordering whose digraph bandwidth is at most $(1+\epsilon)$ times the optimal digraph bandwidth, in time $\mathcal{O}^*(4^n \cdot (\lceil 4/\epsilon \rceil)^n)$.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms; Mathematics of computing → Approximation algorithms

Keywords and phrases directed bandwidth, digraph bandwidth, approximation scheme, exact exponential algorithms

[1] The $\mathcal{O}^\star$ notation hides the polynomial factors in the instance size.

39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019).
Editors: Arkadev Chattopadhyay and Paul Gastin; Article No. 18; pp. 18:1–18:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.18

Funding *Saket Saurabh*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819416).

1 Introduction

The BANDWIDTH problem is a famous combinatorial problem, where given an undirected graph G on n vertices, the goal is to embed its vertices onto an integer line such that the maximum stretch of any edge of G is minimized. More formally, given a graph G on n vertices and an ordering $\sigma : V(G) \to [n]$, the *bandwidth of σ with respect to G* is $\max_{(u,v)\in E(G)}\{|\sigma(u) - \sigma(v)|\}$. In the BANDWIDTH problem, given a graph G, the goal is to find an ordering $\sigma : V(G) \to [n]$, which has *minimum bandwidth* with respect to G. The bandwidth problem has found applications in an array of fields including, but not limited to, the design of faster matrix operations computation on sparse matrices, VLSI circuit design, reducing the search space of constraint satisfaction problems and problems from molecular biology [22]. In many of the real world applications, a fundamental principle that the BANDWIDTH problem captures is that of delays that occur as a result of allocation of tasks on the time interval that have dependencies among them. An ordering in many scenarios represent the allocation of tasks/objects on a time-line/one-dimensional hardware, and the stretch of an edge captures the delay/effort/expense incurred to reach the other end of the edge.

One restriction on the kind of models captured by BANDWIDTH is that, the models cannot be tuned to allow for asymmetry or bias. More specifically, what happens when the connections available between the tasks/objects are unidirectional? What happens when there is a bias in terms of delay/expense based on the direction of communication on the time-line/one-dimensional hardware? The above inquisitivities lead to our first contribution to this article, which is the concept of DIGRAPH BANDWIDTH[2]. Given a directed graph D on n vertices and an ordering $\sigma : V(D) \to [n]$, the *digraph bandwidth* of σ with respect to D is the *maximum stretch of the forward arcs in the ordering*, that is, $\max_{\substack{(u,v)\in E(D)\\ \sigma(u)<\sigma(v)}}\{\sigma(v) - \sigma(u)\}$. The DIGRAPH BANDWIDTH problem takes as input a digraph D and outputs an ordering $\sigma : V(D) \to [n]$ with the least possible *digraph bandwidth* with respect to D.

Observe that, with the introduction of directions in the input graph, DIGRAPH BANDWIDTH allows us to capture one-way dependencies, that can help in modelling scenarios where the links available for modelling the communication are one-directional. Also, by allowing to care only about the stretch of the forward arcs in the ordering, one can model channels where communication in one direction is cheaper/easier than the other. The later scenarios can occur while modelling an uphill-downhill communication, where the cost of going up is a matter of real concern whereas, the cost of going down is almost negligible.

Note that DIGRAPH BANDWIDTH is indeed a generalization of the notion of undirected bandwidth, as for any graph G, if $\overleftrightarrow{G}$ denotes the digraph obtained from G by replacing each edge of G by one arc in both direction, then the bandwidth of G is equal to the directed bandwidth of $\overleftrightarrow{G}$. We would like to remark here that on the theoretical front, the way we lift the definition of bandwidth in undirected graphs to directed graphs, by considering the

[2] We choose the name DIGRAPH BANDWIDTH over the more conventional DIRECTED BANDWIDTH to avoid clash of names from literature (which will be discussed later).

stretches of *only the forward arcs*, is not something unique that we do for Bandwidth. The idea of only considering arcs going in *one direction* for "optimizing some function" is very common to the directed setting. The simplest such example is the notion of a directed cut. If D is a digraph and X, Y are two disjoint subsets of vertices of D, then the *directed cut* of X and Y, $\texttt{dcut}(X, Y)$, is defined as the set of arcs (u, v) in $E(D)$, where $u \in X$ and $v \in Y$. Another closely related notion is the notion of Directed Cutwidth introduced by Chudnosky et al. [5]. A digraph D on n vertices has cutwidth at most k if there exists an ordering of the vertices σ such that for every $i \in [n-1]$, $\texttt{dcut}(\{\sigma(1), \ldots, \sigma(i)\}, \{\sigma(i+1), \ldots, \sigma(n)\})$ is at most k. Note that our notion of directed bandwidth is a stronger notion than cutwidth, as for any ordering σ, the cutwidth associated to σ is at most the digraph bandwith of σ. There is also a similar notion of Directed Pathwidth [5]. Observe that similar to Directed Cutwidth and Directed Pathwidth, Digraph Bandwidth is 0 on directed acyclic graphs (dags).

We would like to remark that ours is not the first attempt in generalising the definition of bandwidth for digraphs. A notion of bandwidth for directed graphs appeared in 1978 in the paper by Garey et al. [16]. But the notion was defined only for dags. In their problem, which they call Directed Bandwidth (DAG-BW), given a dag D, one is interested in finding a *topological* ordering (a linear ordering of vertices such that for every directed arc (u, v) from vertex u to vertex v, u comes before v in the ordering) of minimum bandwidth. Note that this is very different from our notion of Digraph Bandwidth which is always 0 for dags.

Algorithmic Perspective

Bandwidth is one of the most well-known and extensively studied graph layout problems [17]. The Bandwidth problem is NP-hard [25] and remains NP-hard even on very restricted subclasses of trees, like caterpillars of hair length at most 3 [24]. Furthermore, the bandwidth of a graph is NP-hard to approximate within a constant factor for trees [3]. Polynomial-time algorithms for the exact computation of bandwidth are known for a few graph classes including caterpillars with hair length at most 2 [2], cographs [29], interval graphs [20] and bipartite permutation graphs [19]. A classical algorithm by Saxe [26] solves Bandwidth in time $2^{\mathcal{O}(k)}n^{k+1}$, which is polynomial when k is a constant. In the realm of parameterized complexity, Bandwidth is known to be W[t]-hard for all $t \geq 1$, when parameterized by the bandwidth k of the input graph [4]. However, on trees it admits a parameterized approximation algorithm [12] and an algorithm with running time $2^{\mathcal{O}(k \log k)}n^{\mathcal{O}(1)}$ on AT-free graphs [18]. Unger showed in [27] that the problem is APX-hard. The best known approximation algorithm for this problem is due to Krauthgamer et al. [21] and it provides an $\mathcal{O}(\log^3 n)$ factor approximation.

The Bandwidth problem is one of the test-bed problems in the area of moderately exponential time algorithms and has been studied intensively. Trying all possible permutations of the vertex set yields a simple $\mathcal{O}^*(n!)$ time algorithm while the known algorithms for the problem with running time $2^{\mathcal{O}(n)}$ are far from straightforward. The $\mathcal{O}^*(n!)$ barrier was broken by Feige and Kilian [13] who gave an algorithm with running time $\mathcal{O}^\star(10^n)$. This result was subsequently improved by Cygan and Pilipczuk [6] down to $\mathcal{O}^\star(5^n)$. After a series of improvements, the current fastest known algorithm, due to Cygan and Pilipczuk [9, 7] runs in time $\mathcal{O}^\star(4.383^n)$. We also refer the readers to [8] for the best known exact algorithm running in polynomial space. For graphs of treewidth t, one can design an algorithm with running time $2^n n^{\mathcal{O}(t)}$ [1, 7]. On the other hand, Feige and Talwar [14] showed that the bandwidth of a graph of treewidth at most t can be $(1+\varepsilon)$-approximated in time $2^{\mathcal{O}(\log n (t+\sqrt{\frac{n}{\varepsilon}}))}$. Vassilevska et al. [28] gave a hybrid algorithm which after a polynomial time test, either computes the bandwidth of a graph in time $4^{n+o(n)}$, or provides $\gamma(n)\log^2 n \log\log n$-approximation in polynomial time for any unbounded γ. Moreover, for any two positive integers $k \geq 2$, $r \geq 1$, Cygan and Pilipczuk presented a $(2kr-1)$-approximation algorithm that solves Bandwidth for an arbitrary input graph in $\mathcal{O}(k^{\frac{n}{(k-1)r}} n^{\mathcal{O}(1)})$ time and polynomial space [7]. Finally, Fürer et al. [15] gave a factor 2-approximate algorithm for Bandwidth running in time $\mathcal{O}(1.9797^n)$. DAG-BW, as defined by Garey et al. [16] for dags, was shown to admit a polynomial time algorithm for testing if a dag has bandwidth at most 2. Also, it was proved that the problem to determine if the directed bandwidth of a dag is at most k, for any $k > 2$, is NP-hard even in the case of oriented trees. This notion of directed bandwidth *reappeared* in [23], where it was studied for dense digraphs.

Our Results

The main objective of this paper is to *introduce a directed* variant of the Bandwidth problem for general digraphs and study it from the view point of moderately exponential time algorithms, both *exactly and approximately.* Throughout the remaining, n, m denote the number of vertices and arcs in the input digraph, respectively. For many linear layout problems on graphs on n vertices, beating even the trivial $\mathcal{O}^\star(n!)$ algorithm asymptotically remains a challenge. In this article we design $2^{\mathcal{O}(n)}$ time algorithms for Digraph Bandwidth. Below we mention the challenges that Digraph Bandwidth imposes when we try to apply the techniques used in the design of $2^{\mathcal{O}(n)}$ algorithm for Bandwidth, and how we bend our ways to overcome them to design the desired algorithms.

The $2^{\mathcal{O}(n)}$ time algorithms for Bandwidth that exist in literature (cited above), all follow a common principle of *bucket-then-order.* Suppose one is interested in checking whether the input graph has an ordering of bandwidth b. The *bucket-then-order* procedure is a 2-step procedure, where in the first step, instead of directly guessing the position of the vertex in the ordering, for a range of consecutive positions (called *buckets*) of size $\mathcal{O}(b)$, one guesses the set of vertices that will occupy these positions in the final ordering. This process of allocating a set of vertices to a range of consecutive positions is called *bucketing.* Since one can always assume that the graph is connected, once a bucket for the first vertex is guessed using n trials, its neighbours only have a choice of some c buckets for a small constant c depending on the constant in the order notation of the size of the bucket. This, makes the bucketing step run in time $2^{\mathcal{O}(n)}$. The outcome of the first step is a collection of bucketings which contains a bucketing that is "consistent" with the final ordering. In the second step, given such a consistent bucketing, one can find the final ordering using either a recursive divide and conquer technique or a dynamic programming procedure or a measure and conquer kind of an analysis.

In the case of Digraph Bandwidth, finding a bucketing that is consistent with the final ordering becomes a challenge as even the *information that a vertex is placed in some fixed bucket does not decrease the options of the number of buckets in which its neighbours can be placed.* This is because there could be some out-neighbours (resp. in-neighbours) of it that need to be placed before (resp. after) it thereby contributing to backward arcs, which eventually results in the need for allocating them to far off buckets. We cope up with this challenge of bucketing in two ways - both of which lead to interesting algorithms that run in $2^{\mathcal{O}(n)}$ time in different cases. As a first measure of coping up, we take the strategy of "kill what cause you trouble". Formally speaking, it is the set of backward arcs in the final ordering that have unbounded stretch and hence, make the bucketting process difficult. One way to get back to the easy bucketting case is to guess the set of arcs that will appear as backward arcs in the final ordering. Having guessed these arcs, one can remove them from the graph and preserve the information that the arcs which remain all go forward the final ordering. This problem becomes similar to the DAG-BW problem defined on dags by Garey et al. [16]. We show that one can do the bucketing tricks similar to the undirected case here to design a $2^{\mathcal{O}(n)}$ algorithm for this problem (Theorem 1.1). This together with the initial guessing of the backward arcs gives Theorem 1.2.

▶ **Theorem 1.1.** DAG-BW *on dags can be solved in* $\mathcal{O}^\star(3^n)$ *time.*

▶ **Theorem 1.2.** Digraph Bandwidth *can be solved in* $\mathcal{O}^\star(3^n \cdot 2^m)$ *time.*

Note that even though the 2^m in the running time of Theorem 1.2 looks expensive, it already generates an algorithm better that $\mathcal{O}^\star(n!)$ for any digraph that has at most $o(n \log n)$ arcs. In particular, this implies an exact algorithm with running time $2^{\mathcal{O}(n)}$ whenever $|E(D)| = \mathcal{O}(|V(D)|)$, for example for digraphs with bounded average degree.

We will now briefly explain about our second way of dealing with the bucketing phase. As discussed earlier, getting a hold over the arcs which will go backward in the final ordering, eases out the remaining process. In this strategy, instead of guessing the arcs that goes backward by a brute force way (that takes 2^m), we exploit the fact that guessing a partition of the vertex set into two parts, left and right - which corresponds to the first $n/2$ vertices in the final ordering and the last $n/2$ vertices in the final ordering, also gives hold on *some* if not all backward arcs in the final ordering. We place this simple observation into the framework of a divide and conquer algorithm to get a bucketting that is not necessarily "consistent" with the final ordering, but is not too far away to yield a "close enough" approximation to the optimal ordering. This result is formalized in Theorem 1.3. Effectively, the result states that one can find an ordering whose digraph bandwidth is at most $(1+\epsilon)$ times the optimal in time $\mathcal{O}^\star(1/\epsilon)^n$. Note that, this result is in contrast with the result of Feige and Talwar [14] for undirected bandwidth where they gave an exponential time approximation scheme that run in time which had a dependence on the treewidth of the graph($2^{\mathcal{O}(\log n(t+\sqrt{\frac{n}{\epsilon}}))}$). As a side result of our strategy, we can also design an algorithm for solving Digraph Bandwidth optimally on general digraphs in time $\mathcal{O}^\star(2^{\mathcal{O}(n)} \cdot OPT^n)$ or $\mathcal{O}^\star(2^{\mathcal{O}(n)} \cdot 2^{OPT \log OPT \log n})$, where OPT is the optimal digraph bandwidth of the input digraph. This result is stated in Theorem 1.4. Note that, on one hand where $\mathcal{O}(OPT^n)$ is easy to get for the undirected case (because fixing the position of one vertex in the ordering leaves only $2 \cdot OPT$ choices for its neighbours), it is not trivial for the directed case. Also, observe that Theorem 1.4 gives a $2^{\mathcal{O}(n)}$ algorithm whenever $b \leq {}^{n}\!/\!_{\log^2 n}$.

▶ **Theorem 1.3** ((Single) Exponential Time Approximation Scheme)**.** *For any real number* $\epsilon > 0$*, for any digraph* D*, one can find an ordering of digraph bandwidth at most* $(1+\epsilon)$ *times the optimal, in time* $\mathcal{O}^*(4^n \cdot (\lceil 4/\epsilon \rceil)^n)$*.*

▶ **Theorem 1.4.** Digraph Bandwidth *can be solved in* $\min\{\mathcal{O}^*(4^n \cdot b^n), \mathcal{O}^*(4^n \cdot 2^{b \log b \log n})\}$ *time, where* b *is the optimal digraph bandwidth of the input digraph.*

Our last result is based on the connection of the Bandwidth problem with a subgraph isomorphism problem. Amini et al. [1] viewed the Bandwidth problem, on undirected graphs, as a subgraph isomorphism problem, and using an inclusion-exclusion formula with the techniques of counting homomorphisms on graphs of bounded treewidth, they showed that an optimal bandwidth ordering of a graph on n vertices of treewidth at most t can be computed in time $\mathcal{O}^\star(2^{t \log n + n})$ and space $\mathcal{O}^\star(2^{t \log n})$. Using this approach and by relating Digraph Bandwidth via directed homomorphisms to directed path-like-structures, we obtain the following result.

▶ **Theorem 1.5.** *Let D be a digraph on n vertices and D' be the underlying undirected graph. If the treewidth of D' is at most t, then* Digraph Bandwidth *can be solved in time $\mathcal{O}^\star(2^{n+(t+2)\log n})$.*

Observe that Theorem 1.5 provides $\mathcal{O}^\star(2^{n+\mathcal{O}(\sqrt{n}\log n)})$ algorithm for directed planar graphs and for digraph whose underlying undirected graph excludes some fixed graph H as a minor. This algorithm in fact, yields a $2^{\mathcal{O}(n)}$ time algorithm even when the treewidth of the underlying undirected graph of the given digraph is $\mathcal{O}(n/\log n)$. Notice that Theorem 1.2 gives $2^{\mathcal{O}(n)}$ time algorithm for digraphs of constant average degree, while Theorem 1.5 will not apply to these cases as these digraphs could contain expander graphs of constant degree whose treewidth of the underlying undirected graph could be n/c, for some fixed constant c. On the other hand Theorem 1.5 could give $2^{\mathcal{O}(n)}$ time algorithm for digraphs that have $\mathcal{O}(n^2/\log n)$ arcs but treewidth is $\mathcal{O}(n/\log n)$. Thus, Theorems 1.2 and 1.5 give $2^{\mathcal{O}(n)}$ time algorithm for different families of digraphs.

Road Map

In Section 2, we introduce some notation and definitions. Section 3 is devoted to the proof of Theorem 1.1 and Section 4 proves Theorem 1.2 with the help of Theorem 1.1. Section 5 leads to the proofs of Theorems 1.3 and 1.4. Section 6 proves Theorem 1.5. We finally conclude in Section 7.

2 Preliminaries

For positive integers i, j, $[i] = \{1, \cdots, i\}$ and $[i, j] = \{i, \cdots, j\}$. For any set X, by $X = (X_1, X_2)$ we denote an ordered partition of X, that is $X_1 \cup X_2 = X$, $X_1 \cap X_2 = \emptyset$ and, (X_1, X_2) and (X_2, X_1) are two different partitions of X. For any functions $f_1 : X_1 \to Y_1$ and $f_2 : X_2 \to Y_2$, we say that f_1 *is consistent with* f_2 if for each $x \in X_1 \cap X_2$, $f_1(x) = f_2(x)$. If f_1 and f_2 are consistent, then $f_1 \cup f_2 \colon X_1 \cup X_2 \to Y_1 \cup Y_2$ is defined as $(f_1 \cup f_2)(x) = f_i(x)$, if $x \in X_i$. For any set V of size n, we call a function $\sigma \colon V \to [n]$ as an *ordering* of V). Given an ordering σ of $V(D)$, an arc $(u, v) \in E(D)$ is called a *forward arc in* σ if $\sigma(u) < \sigma(v)$, otherwise it is called a *backward arc*. For a natural number $b \in \mathbb{N}$, we call σ as a *b-ordering of D* if for any forward arc $(u, v) \in E(D)$, $\sigma(v) - \sigma(u) \leq b$, that is, if it has digraph bandwidth at most b. Given a set V and an integer b, a b-bucketing of V is a function $\mathtt{B} : V \to [p, q]$, such that $p, q \in \mathbb{N}$ and for each $i \in [p, q-1]$, $|\mathtt{B}^{-1}(i)| = b$ and $|\mathtt{B}^{-1}(q)| \leq b$. Note that, if $|V|$ is a multiple of b, then $\mathtt{B}^{-1}(q) = b$ and $(q - p + 1) \cdot b = |V|$. If for each $i \in [p, q]$, $|\mathtt{B}^{-1}(i)| \leq b$, we call $\mathtt{B}$ a *partial b-bucketing* of V. Note that, for any b, every b-bucketing is a partial b-bucketing. For a (partial) b-bucketing $\mathtt{B} : V \to [p, q]$, we say that an element $v \in V$ is assigned the *i-th bucket of* $\mathtt{B}$ if $\mathtt{B}(v) = i$ and $\mathtt{B}(v)$ is called the *bucket of* v. Also, b is called the *size of the bucket* $\mathtt{B}(v)$. If $\mathtt{B}(u) = i$ and $\mathtt{B}(v) = j$ and $j > i$, then *the number of buckets between the buckets of u and v* is equal to $j - i - 1$. Also, the *number of elements of V in the buckets between i and j is* $(j - i - 1) \cdot b$. In explanations, we sometimes drop b and call $\mathtt{B}$ a *(partial) bucketing* to mean that it is a b-bucketing for some b that should be clear from the context. Given a set V, an integer b and an ordering σ of V, one can associate a b-bucketing with σ which assigns the first b elements in σ the 1-st bucket, the next b-elements the next and so on. This is formalized below. Given a set V, an integer b and an ordering σ of V, we say a b-bucketing $\mathtt{B}$ *respects* σ if $\mathtt{B} : V \to [\lceil |V|/b \rceil]$ is defined as follows. For any $x \in [|V|]$, if $x = ib + j$ for some $i \in \mathbb{N}$ such that $j < b$, then $\mathtt{B}(\sigma_x) = i + 1$ if $j > 0$, and $\mathtt{B}(\sigma_x) = i$ if $j = 0$.

The proofs marked with $\star$ have been omitted because of space constraints and will appear in the full version.

3 Exact Algorithm for Directed Bandwidth for dags

The goal of this section is to prove Theorem 1.1. The algorithm follows the ideas of Cygan and Pilipczuk [10]. We give the details here for the sake of completeness and to mention the little details where we deviate from the algorithm of [10]. Throughout this section, without loss of generality, we can assume that the input digraph D is weakly connected, as otherwise, one can solve the problem on each of the weakly connected components of D and concatenate the orderings obtained from each of them, in any order, to get the final ordering. Also, instead of working on the optimization version of the problem, we work on the decision problem, where together with the input digraph D, one is given an integer b, and the goal is to decide whether there exists a topological ordering of $V(D)$ of bandwidth at most b. It is easy to see that designing an algorithm for this decision version with the desired running time is enough to prove Theorem 1.1. In the following, we abuse notation a little and call (D, b) as an instance of DAG-BW.

Throughout the remaining section, we call a topological ordering of D of bandwidth b as a *b-topological ordering*. A b-bucketing of $V(D)$ is called a *b-topological bucketing* if for all $(u, v) \in E(D)$, either $\mathtt{B}(u) = \mathtt{B}(v)$ or $\mathtt{B}(v) = \mathtt{B}(u)+1$. Our algorithm, like the algorithm of [10], has two phases : Bucketing and Ordering. The Bucketing phase of the algorithm is described by Lemma 3.1.

▶ **Lemma 3.1.** ($\star$) Given an instance (D, b) of DAG-BW, one can find a collection $\mathcal{B}$, of $(b+1)$-topological bucketings of $V(D)$ of size at most $2^{n-1} \cdot \lceil n/b+1 \rceil$, in time $\mathcal{O}^\star(2^n)$, such that for every b-topological ordering σ of D, there exists a bucketing $\mathtt{B} \in \mathcal{B}$ such that $\mathtt{B}$ respects σ.

In the Ordering phase, given a $(b+1)$-topological bucketing $\mathtt{B}$, the algorithm finds a b-topological ordering σ of D, if it exists, such that $\mathtt{B}$ respects σ. From Lemma 3.1, the family $\mathcal{B}$ guarantees the existence of a $(b+1)$-topological bucketing $\mathtt{B}$ of the final desired ordering, if it exists.

To execute this step, we use the idea of finding a sequence of *lexicographically embeddibe sets* using dynamic programming as used in [10]. To define lexicographically embeddible set, the authors first defined the notion of *lexicographic ordering of slots*. We use the same definition in this paper.

▶ **Definition 3.2** (Lexicographic ordering of *slots*). *Given an integer b, let* $\mathtt{bucket}\colon [n] \to \lceil n/(b+1) \rceil$ *be a function such that* $\mathtt{bucket}(i) = \lceil i/(b+1) \rceil$ *and* $\mathtt{pos}\colon [n] \to [b+1]$ *be a function such that* $\mathtt{pos}(i) = ((i-1) \mod (b+1)) + 1$. *We define the lexicographic ordering of slots as the lexicographic ordering of* $(\mathtt{pos}(i), \mathtt{bucket}(i))$*, where* $i \in [n]$.

For the Bandwidth problem, the authors of [10] proceed as follows. Given a graph $G = (V, E)$, and a $(b+1)$-bucketing, $\mathtt{B}$ of $V(G)$, they prove that there exists an ordering σ of G such that $\mathtt{B}$ respects σ if and only if there exists a sequence of subsets of $V(G)$, $\emptyset \subset S_1 \subset \cdots \subset S_n$, $|S_i| = i$, for all $i \in [n]$, such that each S_i satisfies the following properties: (i) for each S_i, there is a mapping $\gamma_{S_i}: S_i \to [n]$ such that $\mathtt{bucket}(\gamma_{S_i}(v)) = \mathtt{B}(v)$, and the set $\{(\mathtt{pos}(\gamma_{S_i}(v)), \mathtt{bucket}(\gamma_{S_i}(v))) \mid v \in S_i\}$ is the set of first $|S_i|$ elements in the lexicographic ordering of slots, and (ii) if $u \in S_i$ and $v \notin S_i$, then $\mathtt{B}(v) \leq \mathtt{B}(u)$. They call such a set S_i as a lexicographically embeddible set. They then obtain $\gamma_{S_{i+1}}$ by extending γ_{S_i} as follows. If $v \in S_i \cap S_{i+1}$, then $\gamma_{S_{i+1}}(v) = \gamma_i(v)$, otherwise $(\mathtt{pos}(\gamma_{S_{i+1}}(v)), \mathtt{bucket}(\gamma_{S_{i+1}}(v)))$ is the $|S_{i+1}|^{\text{th}}$ element in the lexicographic embedding of slots. Recall that there is only one vertex v in $S_{i+1} \setminus S_i$. Furthermore, if v has a neighbor u in S_i, then $\mathtt{B}(v) \leq \mathtt{B}(u)$. If $\mathtt{B}(v) = \mathtt{B}(u)$, then since bucket size is at most $b+1$, $|\gamma(v) - \gamma(u)| \leq b$. If $\mathtt{B}(v) < \mathtt{B}(u)$, then by construction of $\gamma_{S_{i+1}}$, $\texttt{pos}(\gamma_{S_{i+1}}(v)) > \texttt{pos}(\gamma_{S_{i+1}}(u))$. Now, again since each bucket size is at most $b+1$, $|\gamma(v) - \gamma(u)| \leq b$. Therefore, γ_{S_i} can be extended to $\gamma_{S_{i+1}}$. Thus, γ_{S_n} will yield the final ordering. Hence, the goal reduces to finding a sequence $S_1 \subset \cdots \subset S_n$, $|S_i| = i$, for all $i \in [n]$, such that each S_i is a lexicographically embeddible set. We will call such a sequence as a *lexicographically embeddible sequence* (les, in short).

We proceed in a similar way for Directed Bandwidth. We first note that one cannot use the same definition of lexicographically embeddible set as defined above due to the following reason. Suppose that S_i is a lexicographical embeddible set. Consider a vertex $u \in S_i$. Suppose that there exists a vertex $v \notin S_i$ that is an in-neighbor of u, then v cannot belong to the bucket of u as it will not lead to the topological ordering using the method defined above. Also, if v is an out-neighbor of u, then since we are working with topological bucketing, v does not belong to the bucket that precedes the bucket of u. Hence, v belongs to the bucket of u. We also want γ as a topological ordering. Therefore, we redefine the notion of lexicographically embeddible set for Directed Bandwidth as follows.

► **Definition 3.3** (Lexicographically embeddible set for digraphs). *Given a $(b+1)$-topological bucketing $\texttt{B}$, of $V(D)$, we say that $S \subseteq V(D)$ is a lexicographically embedibble set if the following condition holds.*

(C1) *For each arc $(u, v) \in E(D)$ such that $u \in S$ and $v \notin S$, $\texttt{B}(u) = \texttt{B}(v)$.*

(C2) *For each arc $(u, v) \in E(D)$ such that $v \in S$ and $u \notin S$, $\texttt{B}(u) = \texttt{B}(v) - 1$.*

(C3) *There exists a b-topological ordering $\gamma\colon S \to [n]$ such that for all $v \in S$, $\texttt{bucket}(\gamma(v)) = \texttt{B}(v)$, and $(\texttt{pos}(\gamma(v)), \texttt{bucket}(\gamma(v)))$ belongs to the first $|S|$ elements in the lexicographic ordering of slots. We refer γ as a partial b-topological ordering that respects lexicographic ordering of slots.*

Now we prove that given a $(b+1)$-topological bucketing $\texttt{B}$ of $V(D)$, to find a b-topological ordering σ such that $\texttt{B}$ respects σ,

► **Lemma 3.4.** ($\star$) Given a $(b+1)$-topological bucketing $\texttt{B}$ of $V(D)$, the following are equivalent. **(i)** There exists a b-topological ordering σ of the digraph D such that the unique $(b+1)$-bucketing induced by σ, that is $\texttt{B}_\sigma$, is $\texttt{B}$. In other words, $\texttt{B}_\sigma(v) = \texttt{B}(v)$, for all $v \in V(D)$, **(ii)**There exists a les, $\emptyset \subset S_1 \subset \cdots \subset S_n = V$.

Due to Lemma 3.4, our goal is reduced to find a les. Cygan and Pilipczuk [10] find les using dynamic programming over subsets of the vertex set of given graph. We use a similar dynamic programming approach with appropriate modification because of the revised definition of a lexicographically embeddible set. In the dynamic programming table, for each $S \subseteq V(D)$, $c[S] = 1$, if and only if S is a lexicographically embeddible set. To compute the value of $c[S]$, we first find a vertex $v \in S$ such that $S \setminus \{v\}$ is a lexicographically embeddible set, that is, $c[S \setminus \{v\}] = 1$, and v satisfies the following properties : (i) for all the arcs $(v, u) \in E(D)$ such that $u \notin S$, $\texttt{B}(v) = \texttt{B}(u)$; (ii) for all the arcs $(u, v) \in E(D)$ such that $u \notin S$, $\texttt{B}(u) = \texttt{B}(v) - 1$; and (iii) $\texttt{B}(v) = ((|S| - 1) \mod \lceil n/(b+1) \rceil) + 1$. We compute the value of $c[S]$ for every subset $S \subseteq V(D)$. Note that if $c[V(D)] = 1$, then $V(D)$ is a lexicographically embeddible set. Also, we can compute les by backtracking in the dynamic programming table.

► **Lemma 3.5.** ($\star$) Given an instance (D, b) of DAG-BW, and a $(b+1)$-topological bucketing $\texttt{B}$ of $V(D)$ that respects some b-topological ordering of D (if it exists), one can compute a les in time $\mathcal{O}^\star(2^n)$.

Note that using Lemmas 3.1, 3.4 and 3.5, one can solve DAG-BW in $\mathcal{O}^\star(4^n)$ time. The desired running time of $\mathcal{O}^\star(3^n)$ in Theorem 1.1 can be proved by careful analysis of two steps in the algorithm as done in Theorem 12 of [10]. Since the proof of this is the same as that of Theorem 12 of [10], we defer the proof here.

4 Exact Algorithm for Digraph Bandwidth via Directed Bandwidth

We call an ordering of the vertex set of a digraph a b-*ordering* if its digraph bandwidth is at most b. In order to prove Theorem 1.2 observe the following. Let (D, b) be an instance of (the decision version of) DIGRAPH BANDWIDTH. If it is a YES instance, then let σ be a b-ordering of D. Let R be the set of backward arcs in σ. Note that σ is a topological ordering of $D - R$. If we guess the set of backward arcs R in a b-ordering of D (which takes time 2^m), then the goal is reduced to finding a b-topological ordering, σ, of $D - R$ such that if $(u, v) \in R$, then $\sigma(u) > \sigma(v)$. In fact, one can also observe that it is sufficient to find a b-topological ordering, ρ, of $D - R$ such that for all $(u, v) \in R$ either $\rho(u) > \rho(v)$ or $\rho(v) - \rho(u) \leq b$. We claim that we can find the required ordering of $D - R$ using the algorithm for DIRECTED BANDWIDTH for dags given in Section 3. Suppose that σ is a b-ordering of D. Let $\mathtt{B}_\sigma$ be a $(b+1)$-bucketing that respects σ. Let R be the set of backward arcs in σ. Since σ is a b-topological ordering of $D - R$, using Lemma 3.1, $\mathtt{B}_\sigma$ belongs to the collection of $(b + 1)$-bucketings $\mathcal{B}$. Now, using Lemmas 3.5 and 3.4, we obtain a b-topological ordering ρ of $D - R$ that respects $\mathtt{B}_\sigma$. Note that ρ is a b-ordering of D, as for each arc $(u, v) \notin R$, $\rho(v) - \rho(u) \leq b$ because ρ is a b-topological ordering of $D - R$. Also, if $(u, v) \in R$, then observe that if $\mathtt{B}_\sigma(u) \neq \mathtt{B}_\sigma(v)$, then since both σ and ρ respect $\mathtt{B}_\sigma$ and (u, v) is a backward arc in σ, thus, (u, v) is a backward arc in ρ too, that is, $\rho(u) > \rho(v)$. Otherwise, if $(u, v) \notin R$ and $\mathtt{B}_\sigma(u) = \mathtt{B}_\sigma(v)$, then since ρ respects $\mathtt{B}_\sigma$ and the size of the buckets of $\mathtt{B}_\sigma$ is $(b + 1)$, therefore, $|\rho(u) - \rho(v)| \leq b$. Thus, the algorithm of Theorem 1.2 runs the algorithm for DAG-BW for each $R \subseteq E(D)$, to obtain the desired running time.

5 (Single) Exponential Time Approximation Scheme for Digraph Bandwidth

The goal of this section is to prove Theorems 1.3 and 1.4. Let (D, b) be an instance of (the decision version of) DIGRAPH BANDWIDTH. The algorithm relies on an interesting property of a b-bucketing that respects a b-ordering of D. Let σ be a b-ordering of D and let $\mathtt{B}$ be a b-bucketing of $V(D)$ that respects σ. An interesting property of such a bucketing $\mathtt{B}$ is that *if* $(u, v) \in E(D)$, *then either* $\mathtt{B}(u) > \mathtt{B}(v)$ *or* $\mathtt{B}(v) \leq \mathtt{B}(u) + 1$. This is because the size of each bucket in $\mathtt{B}$ is b and σ is a b-ordering of D. Let us call this property of a b-bucketing *useful*. What we saw in the previous section is that if we somehow have a bucketing that respects σ, then one can design an algorithm to fetch σ from this bucketing. In this section, instead of seeking for a bucketing that respects σ we seek for a bucketing with the above mentioned useful property. Observe that, while the existence of such a bucketing with this useful property might not necessarily imply the existence of some b-ordering of D, but having such a bucketing with, for example buckets of size b, definitely yields a $2b$-ordering of D. This is because, given such a bucketing one can assign positions to vertices in the ordering by choosing any arbitrary ordering amongst the vertices that belong to the same bucket and concatenating these orderings in the order of the bucket numbers. By changing the bucket size in the described bucketing, one can yield an ordering of digraph bandwidth at most $(1 + \epsilon)$ times the optimal. This procedure, as we will see, also give an optimal digraph bandwidth ordering when we use the bucket sizes to be 1. Below we give the description of the algorithm used to find a bucketing with the useful property.

We begin by formulating the useful property of a bucketing described above. Since the size of buckets is uniform in a bucketing, instead of defining the property in terms of bucket numbers, we describe it in terms of the number of vertices that can appear between the two buckets corresponding to the end points of a forward arc in the ordering. Such a shift in definition helps us to get slightly better bounds in our running time. For any positive integers b, s and a digraph D, given $X \subseteq V(D)$ and a (partial) b-bucketing of X, say $\texttt{B} : X \to [p, q]$ for some $p, q \in \mathbb{N}$, we say that the *external stretch of* $\texttt{B}$ *is at most* s if for each arc $(u, v) \in E(D[X])$, either $\texttt{B}(u) \geq \texttt{B}(v)$, or $(\texttt{B}(v) - \texttt{B}(u) - 1) \cdot b \leq s$. Recall that $\texttt{B}(u) - \texttt{B}(v) - 1$ denote the number of buckets between the bucket of u and the bucket of v. Our major goal now is to prove Lemma 5.1.

▶ **Lemma 5.1.** *Given a digraph D and positive integers b, s, there is an algorithm, that runs in time* $\min\{\mathcal{O}^*(4^n \cdot (\lceil s+1/b \rceil)^n), \mathcal{O}^*(4^n \cdot (\lceil s+1/b \rceil)^{2(b+s)\log n})\}$*, and computes a b-bucketing of $V(D)$, $\texttt{B} : V(D) \to [\lceil |V(D)|/b \rceil]$, of external stretch at most s.*

We give a recursive algorithm for Lemma 5.1 (Algorithm 1). Since Algorithm 1 is recursive, the input of the algorithm will contain a few more things in addition to D, b, s to maintain the invariants at the recursive steps. We give the description of the input to Algorithm 1 in Definition 5.2.

▶ **Definition 5.2** (Legitimate input for Algorithm 1)**.** *The input* $(D, b, s, first, last, left\text{-}bor(V(D)), right\text{-}bor(V(D)), \texttt{B}_{in})$ *is called* legitimate *for Algorithm 1 if the following holds. Let $\delta = \lceil s+1/b \rceil$.*

(P1) *D is a digraph, b, s are positive integers and $|V(D)| = 2^\eta \cdot b \cdot \delta$, where $\eta \geq 0$ is a positive integer.*

(P2) *$first$ and $last$ are positive integers such that $last - first + 1 = 2^\eta$, where η is such that $|V| = 2^\eta \cdot b \cdot \delta$.*

(P3) *$left\text{-}bor(V(D)), right\text{-}bor(V(D)) \subseteq V(D)$,*

(P4) *$\texttt{B}_{in} : left\text{-}bor(V(D)) \cup right\text{-}bor(V(D)) \to [first, last]$ is a partial b-bucketing such that for each $v \in left\text{-}bor(V(D))$, $\texttt{B}_{in}(v) \in [first, first + \delta - 1]$, for each $v \in right\text{-}bor(V(D))$, $\texttt{B}_{in}(v) \in [last - \delta + 1, last]$ and the external stretch of $\texttt{B}_{in}$ is at most s.*

Observe that $\delta - 1$ represents the number of buckets that can appear between the buckets of u and v in any b-bucketing of external stretch at most s, where the bucket of u precedes the bucket of v and $(u, v) \in E(D)$. We would like to remark that the condition of 1 is not serious as we could have worked without it. We state it like the way we do for the sake of notational and argumentative convenience in the proofs. All it states is that the number of vertices is a power of 2 multiplied by b and δ. The $first$ and $last$ in 2 represents the bucket number of the first and last buckets in the bucketing to be outputted. The relation between $first$ and $last$ in 2 is there to ensure that there are enough buckets to hold the vertices of D. At any recursive call, the sets $left\text{-}bor(V(D))$ and $right\text{-}bor(V(D))$ represent the sets of vertices whose buckets have already been fixed in the previous recursive calls. The set $left\text{-}bor(V)$ represents the set of vertices in V that have an in-neighbour to the vertices that have been decided to be placed in the buckets before the bucket numbered $first$ in the earlier recursive calls. Similarly, $right\text{-}bor(V)$ represents the set of vertices in V that have an out-neighbour to the vertices that have been decided to be placed in the buckets after the bucket numbered $last$ in the earlier recursive calls. Thus, in order to give the final bucketing of external stretch at most s, it is necessary that $left\text{-}bor(V)$ are placed in the first δ buckets and $right\text{-}bor(V)$ are placed in the last δ buckets. This is captured in 4. The next definition describes the properties of the bucketing that would be outputted by Algorithm 1.

▶ **Definition 5.3** (Look-out bucketing for a legitimate instance)**.** *Given a legitimate instance $\mathcal{I} = (D, b, s, first, last, left\text{-}bor(V(D)), right\text{-}bor(V(D)), \texttt{B}_{in})$, we say a bucketing $\texttt{B}$ is a* look-out bucketing for $\mathcal{I}$, *if $\texttt{B}$ is a b-bucketing $\texttt{B}_{out} : V \to [first, last]$ of external stretch at most s that is consistent with $\texttt{B}_{in}$.*

Observe that, for the algorithm of Lemma 5.1, a call to Algorithm 1 on $(D, b, s, 1, {}^{|V|}/_{b}, \emptyset, \emptyset, \phi)$ is enough. To give the formal description of Algorithm 1, we will use the following definition.

▶ **Definition 5.4** (B validates a partition (X_1, X_2)). *For any integers p, q and $X' \subseteq X$, let* $\mathtt{B} : X' \to [p, q]$ *be a partial bucketing of X'. Let (X_1, X_2) be some partition of X. We say that* B validates (X_1, X_2) *if the following holds. Let $r = \lfloor {}^{(p+q)}/_{2} \rfloor$. For each $v \in X_1 \cap X'$,* $\mathtt{B}(v) \in [p, r]$ *and for each $v \in X_2 \cap X'$,* $\mathtt{B}(v) \in [r + 1, q]$.

Algorithm 1 Algorithm for computing b-bucketing of external stretch at most s.

Input: $\mathcal{I} = (D, b, s, first, last, left\text{-}bor(V), right\text{-}bor(V), B_{in})$ such that $\mathcal{I}$ is legitimate for Algorithm 1.

Output: A look-out bucketing for $\mathcal{I}$, if it exists.

```
1: Let $V = V(D)$ and $\delta = \lceil {}^{s+1}/_{b} \rceil$.
2: if $|V| = b \cdot \delta$ then
3:     return any $b$-bucketing $\mathtt{B} : V \to [first, last]$ that it consistent with $\mathtt{B}_{in}$
4: Let $mid = {}^{(first+last-1)}/_{2}$.
5: for each partition $(L, R)$ of $V$ such that $|L| = |R|$ and $B_{in}$ validates $(L, R)$ do
6:     Let $bor_L = \{v \in L \mid$ there exists $u \in R, (v, u) \in E(D)\}$.
7:     Let $bor_R = \{v \in R \mid$ there exists $u \in L, (u, v) \in E(D)\}$.
8:     Let $\mathcal{B}$ be the collection of partial $b$-bucketings, $\mathtt{B} : bor_L \cup bor_R \to [mid - \delta + 1, mid + \delta]$, such that for each $v \in bor_L$, $\mathtt{B}(v) \in [mid - \delta + 1, mid]$, for each $v \in bor_R$, $\mathtt{B}(v) \in [mid + 1, mid + \delta]$, external stretch of B is at most $s$ and B is consistent with $\mathtt{B}_{in}$.
9:     for each $\mathtt{B} \in \mathcal{B}$ do
10:        Define $\mathtt{B}_{in}^{new} : left\text{-}bor(V) \cup bor_L \cup bor_R \cup right\text{-}bor(V) \to [first, last]$, such that for each $v \in left\text{-}bor(V) \cup right\text{-}bor(V)$, $\mathtt{B}_{in}^{new}(v) = \mathtt{B}_{in}(v)$ and, for each $v \in bor_L \cup bor_R$, $\mathtt{B}_{in}^{new}(v) = \mathtt{B}(v)$.
11:        Let $\mathtt{B}_{in}^{newL} : left\text{-}bor(V) \cup bor_L \to [first, mid]$ be such that $\mathtt{B}_{in}^{newL} = \mathtt{B}_{in}^{new}{}_{|L}$.
12:        Let $\mathtt{B}_{in}^{newR} : bor_R \cup right\text{-}bor(V) \to [mid + 1, last]$ be such that $\mathtt{B}_{in}^{newR} = \mathtt{B}_{in}^{new}{}_{|R}$.
13:        Define $left\text{-}bor(L) = left\text{-}bor(V)$ and $right\text{-}bor(L) = bor_L$.
14:        Define $left\text{-}bor(R) = bor_R$ and $right\text{-}bor(R) = right\text{-}bor(V)$.
15:        Let $\mathcal{I}_L^{\mathtt{B}}$ be the instance $(D[L], b, s, first, mid, left\text{-}bor(L), right\text{-}bor(L), \mathtt{B}_{in}^{newL})$.
16:        Let $\mathcal{I}_R^{\mathtt{B}}$ be the instance $(D[R], b, s, mid, last, left\text{-}bor(R), right\text{-}bor(R), \mathtt{B}_{in}^{newR})$.
17:        if $\mathcal{I}_L^{\mathtt{B}}$ and $\mathcal{I}_R^{\mathtt{B}}$ are legitimate inputs for Algorithm 1 then
18:            if Algorithm $1(\mathcal{I}_L^{\mathtt{B}})$ != NO and Algorithm $1(\mathcal{I}_R^{\mathtt{B}})$ != NO then
19:                return Algorithm 1 $(\mathcal{I}_L^{\mathtt{B}}) \cup$ Algorithm 1 $(\mathcal{I}_R^{\mathtt{B}})$
20: return NO
```

For the formal description of Algorithm 1 refer to the pseudocode. We give the informal description of Algorithm 1 here. In a legitimate instance when the number of vertices is $b \cdot \delta$, the number of buckets is δ. Recall that $\delta = \lceil {}^{s+1}/_{b} \rceil$. Note that in this case, every b-bucketing of the vertex set has external stretch at most s. This is because the number of buckets between any two buckets is at most $\delta - 2$ and hence, the number of vertices that appear in the buckets between any two buckets is at most $(\delta - 2) \cdot b \leq s$.

When the number of vertices is larger, the algorithm first guesses which vertices will be assigned a bucket from the first half buckets of the final bucketing (this corresponds to the set L) and which will be assigned the last half (this corresponds to the set R). Since the final bucketing has to be consistent with $\mathtt{B}_{in}$, from the description of $\mathtt{B}_{in}$ in Definition 5.2, the vertices of $left\text{-}bor(V)$ should belong to the first half buckets and the vertices of $right\text{-}bor(V)$ should belong to the last half buckets. Thus, the algorithm only considers those partitions (guesses) which $\mathtt{B}_{in}$ validates (Line 5).

Fix a guessed partition (L, R) of V. The set bor_L represents the set of vertices in L that have an out-neighbour in R. Similarly, the set bor_R represents the set of vertices in R with an in-neighbour in L. Since in any b-bucketing of external stretch at most s, the number of buckets that can appear between the buckets of the end points of a forward arc is at most $\delta - 1$, the vertices of bor_L can only be placed in the δ buckets closest to the middle bucket and before it. Similarly, the vertices of bor_R can only be placed in the δ buckets closest to the middle bucket and after it. The algorithm goes over all possible partial b-bucketings of these vertices in the described buckets, that are consistent with $\mathtt{B}_{in}$, and themselves have external stretch at most δ (Line 8).

For a fixed partial b-bucketing enumerated above, the algorithm recursively finds a bucketing of the L vertices in the first half buckets and the bucketing of the R vertices in the last half buckets that is consistent with $\mathtt{B}_{in}$ and the partial b-bucketing of the bor_L and bor_R vertices guessed. This final bucketing is then obtained by combing the two bucketings from the two disjoint sub-problems (Lines 9 to 19).

▶ **Lemma 5.5.** ($\star$) Algorithm 1 on a legitimate input $(D, b, s, first, last, left\text{-}bor(V), right\text{-}bor(V), B_{in})$, runs in time $\min\{\mathcal{O}^*(4^n \cdot \lceil s{+}1/b \rceil^n), \mathcal{O}^*(4^n \cdot \lceil s{+}1/b \rceil^{2(b+s)\log n})\}$, and returns a look-out bucketing for $\mathcal{I}$, if it exists.

Theorem 1.3 (resp. Theorem 1.3) can be proved by setting bucket size to be $\lceil b\epsilon/2 \rceil$ (resp. 1) and external stretch $b-1$ as parameters in the algorithm of Lemma 5.1. The full proofs are deferred to the full version.

6 Exact Algorithm for Digraph Bandwidth via Directed Homomorphisms

The goal of this section is to prove Theorem 1.5. Towards this, we give a reduction from Digraph Bandwidth to a subgraph homomorphism problem for digraphs. Given two digraphs D and H, a *directed homomorphism from D to H* is a function $h : V(D) \to V(H)$, such that if $(u, v) \in E(D)$, then $(h(u), h(v)) \in E(H)$. A directed homomorphism that is injective is called an *injective directed homomorphism.* For any positive integers n, b such that $b \leq n$, we denote by P_n^b the directed graph on n vertices such that $V(P_n^b) = [n]$ and $E(P_n^b) = E_f \uplus E_b$, where $E_b = \{(i, j) : i > j, i, j \in [n]\}$ and $E_f = \{(i, i{+}j) : i \in [n{-}1], j \in [b]\}$. In the following lemma, we build the relation between Digraph Bandwidth of D and injective homomorphism from D to P_n^b.

▶ **Lemma 6.1.** *For any digraph D and an integer b, D has bandwidth at most b if and only if there is an injective homomorphism from D to P_n^b.*

Proof. In the forward direction, suppose that D has digraph bandwidth at most b. Let σ be a b-ordering of D. Let $f \colon V(D) \to V(P_n^b)$ be a function such that $f(u) = \sigma(u)$. We claim that f is an injective homomorphism. Since σ is an ordering of D, f is an injective function. We prove that it is also a homomorphism. Consider an arc $(u, v) \in E(D)$. If $\sigma(u) < \sigma(v)$, then since σ is a b-ordering, $\sigma(v) - \sigma(u) \leq b$. Therefore, $(f(u), f(v)) \in E_f$. If $\sigma(u) > \sigma(v)$, then $(f(u), f(v)) \in E_b$. Hence, $(f(u), f(v)) \in E(P_n^b)$. In the backward direction, suppose that there exists an injective homomorphism from D to P_n^b. Let $f : V(D) \to V(P_n^b)$ be a function. Then, we claim that $\sigma = (f^{-1}(1), \cdots, f^{-1}(n))$ is a b-ordering of D. Suppose not, then there exists an arc $(u, v) \in E(D)$ such that $\sigma(v) - \sigma(u) > b$. Let $u = f^{-1}(j)$ and $v = f^{-1}(k)$. Note that $\sigma(u) = j$ and $\sigma(v) = k$. Therefore, $j < k$. Since $k - j > b$, $(j, k) \notin E(P_n^b)$, a contradiction that f is an injective homomorphism. ◀

For any two digraphs D and H, let $inj(D, H)$ denote the number of injective homomorphisms from D to H and let $hom(D, H)$ denote the number of homomorphisms from D to H. Then the following lemma holds from Theorem 1 in [1].

▶ **Lemma 6.2** (Theorem 1, [1]). *Suppose that D and H be two digraphs such that $|V(D)| = |V(H)|$. Then,*

$$inj(D, H) = \sum_{W \subseteq V(D)} (-1)^{|W|} hom(D \setminus W, H).$$

Now, we state the following known result about the number of homomorphisms between two given digraphs D and H.

▶ **Lemma 6.3** (Theorem 3.1, 5.1 [11]). *Given digraphs D and H together with a tree decomposition of D of width $\mathtt{tw}$, $hom(D, H)$ can be computed in time $\mathcal{O}(nh^{\mathtt{tw}+1} \min\{\mathtt{tw}, h\})$, where $n = |V(D)|$ and $h = |V(H)|$.*

Using Lemmas 6.2 and 6.3, we get the following result.

▶ **Lemma 6.4.** *Given digraphs D and H together with a tree decomposition of D of width $\mathtt{tw}$, $inj(D, H)$ can be computed in time $\mathcal{O}(2^n nh^{\mathtt{tw}+1} \min\{\mathtt{tw}, h\})$, where $n = |V(D)|$ and $h = |V(H)|$.*

Proof of Theorem 1.5. The proof follows from Lemmas 6.1 and 6.4. ◀

7 Conclusion

In this paper we gave exponential time algorithm for the Digraph Bandwidth problem, that either solved the problem exactly or computed it approximately. In particular, our results imply that whenever $b \leq \frac{n}{\log^2 n}$ or, the treewidth of the underlying undirected digraph is $\mathcal{O}(\frac{n}{\log n})$ or, the number of arcs in the digraph are linear in the number of vertices, then there exists a $2^{\mathcal{O}(n)}$ time algorithm for solving Digraph Bandwidth. Some important questions that remain open about Digraph Bandwidth are the following.

- Does Digraph Bandwidth admit an algorithm with running time $2^{\mathcal{O}(n)}$ on general digraphs?
- Another interesting question is the complexity of the Digraph Bandwidth problem, when b is fixed. Recall that, in the undirected case, Bandwidth can be solved in time $\mathcal{O}(n^{b+1})$ [26]. When $b = 0$, the problem is equivalent to checking if the input is a dag, which can be done in linear time. For $b = 1$, we are able to design an $\mathcal{O}(n^2)$ time algorithm. For $b = 2$, the problem seems to be extremely complex, and in fact, we will be surprised if the problem turns out to be polynomial time solvable. Overall, finding the complexity of Digraph Bandwidth, for a fixed $b \geq 2$, is an interesting open problem.

References

1 O. Amini, F.V. Fomin, and S. Saurabh. Counting Subgraphs via Homomorphisms. *SIAM J. Discrete Math.*, 26(2):695–717, 2012.

2 S.F. Assmann, G.W. Peck, M.M. Sysło, and J. Zak. The bandwidth of caterpillars with hairs of length 1 and 2. *SIAM J. Alg. Discrete Meth.*, 2(4):387–393, 1981.

3 G. Blache, M. Karpiński, and J. Wirtgen. *On approximation intractability of the bandwidth problem.* Citeseer, 1997.

4 H.L. Bodlaender, M.R. Fellows, and M.T. Hallett. Beyond NP-completeness for problems of bounded width (extended abstract): hardness for the W hierarchy. In *Proc. of STOC 1994*, pages 449–458, 1994.

5 M. Chudnovsky, A. Fradkin, and P. Seymour. Tournament Immersion and Cutwidth. *J. Comb. Theory Ser. B*, 102(1):93–101, 2012.

6 M. Cygan and M. Pilipczuk. Faster Exact Bandwidth. In *Proc. of WG 2008*, pages 101–109, 2008.

7 M. Cygan and M. Pilipczuk. Exact and approximate bandwidth. *Theor. Comput. Sci.*, 411(40-42):3701–3713, 2010.

8 M. Cygan and M. Pilipczuk. Bandwidth and distortion revisited. *Discrete Appl. Math.*, 160(4-5):494–504, 2012.

9 M. Cygan and M. Pilipczuk. Even Faster Exact Bandwidth. *ACM Trans. Algorithms*, 8(1):8:1–8:14, 2012.

10 Marek Cygan and Marcin Pilipczuk. Faster exact bandwidth. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 101–109. Springer, 2008.

11 J. Díaz, M. Serna, and D.M. Thilikos. Counting H-colorings of partial k-trees. *Theor. Comput. Sci.*, 281(1-2):291–309, 2002.

12 M.S. Dregi and D. Lokshtanov. Parameterized Complexity of Bandwidth on Trees. In *Proc. of ICALP 2014*, pages 405–416, 2014.

13 U. Feige. Coping with the NP-hardness of the graph bandwidth problem. In *Proc. of SWAT 2000*, pages 10–19. Springer, Berlin, 2000.

14 U. Feige and K. Talwar. Approximating the Bandwidth of Caterpillars. In *Proc. of APPROX-RANDOM 2005*, volume 3624, pages 62–73, 2005.

15 M. Fürer, S. Gaspers, and S.P. Kasiviswanathan. An exponential time 2-approximation algorithm for bandwidth. *Theor. Comput. Sci.*, 511:23–31, 2013.

16 M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity Results for Bandwidth Minimization. *SIAM J. Appl. Math.*, 34(3):477–495, 1978.

17 M.R. Garey and D.S. Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

18 P.A. Golovach, P. Heggernes, D. Kratsch, D. Lokshtanov, D. Meister, and S. Saurabh. Bandwidth on AT-free graphs. *Theor. Comput. Sci.*, 412(50):7001–7008, 2011.

19 P. Heggernes, D. Kratsch, and D. Meister. Bandwidth of bipartite permutation graphs in polynomial time. *J. Discrete Algorithms*, 7(4):533–544, 2009.

20 D.J. Kleitman and R. Vohra. Computing the Bandwidth of Interval Graphs. *SIAM J. Discrete Math.*, 3(3):373–375, 1990.

21 R. Krauthgamer, J.R. Lee, M. Mendel, and A. Naor. Measured Descent: A New Embedding Method for Finite Metrics. In *Proc. of FOCS 2004*, pages 434–443, 2004.

22 Yung-Ling Lai and Kenneth Williams. A survey of solved problems and applications on bandwidth, edgesum, and profile of graphs. *Journal of graph theory*, 31(2):75–94, 1999.

23 Marek M. Karpinski, Jürgen Wirtgen, and A. Zelikovsky. An Approximation Algorithm for the Bandwidth Problem on Dense Graphs. Technical report, University of Bonn, 1997.

24 B. Monien. The bandwidth minimization problem for caterpillars with hair length 3 is NP-complete. *SIAM J. Alg. Discrete Meth.*, 7(4):505–512, 1986.

25 C.H. Papadimitriou. The NP-Completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

26 J.B. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM J. Alg. Discrete Meth.*, 1(4):363–369, 1980.

27 W. Unger. The complexity of the approximation of the bandwidth problem. In *Proc. of FOCS 1998*, pages 82–91, 1998.

28 V. Vassilevska, R. Williams, and S.L.M. Woo. Confronting hardness using a hybrid approach. In *Proc. of SODA*, pages 1–10, 2006.

29 J.H. Yan. The bandwidth problem in cographs. *Tamsui Oxford J. Math. Sci*, 13:31–36, 1997.