On Instance-Optimal Algorithms for a Generalization of Nuts and Bolts and Generalized Sorting

Mayank Goswami ✉ 📵
Queens College CUNY, Flushing, New York, USA

Riko Jacob ✉ 📵
IT University of Copenhagen, København S, Denmark

───── **Abstract** ─────

We generalize the classical nuts and bolts problem to a setting where the input is a collection of n nuts and m bolts, and there is no promise of any matching pairs. It is not allowed to compare a nut directly with a nut or a bolt directly with a bolt, and the goal is to perform the fewest nut-bolt comparisons to discover the partial order between the nuts and bolts. We term this problem *bipartite sorting*.

We show that instances of bipartite sorting of the same size exhibit a wide range of complexity, and propose to perform a fine-grained analysis for this problem. We rule out straightforward notions of instance-optimality as being too stringent, and adopt a *neighborhood-based* definition. Our definition may be of independent interest as a unifying lens for instance-optimal algorithms for other static problems existing in literature. This includes problems like sorting (Estivill-Castro and Woods, ACM Comput. Surv. 1992), convex hull (Afshani, Barbay and Chan, JACM 2017), adaptive joins (Demaine, López-Ortiz and Munro, SODA 2000), and the recent concept of universal optimality for graphs (Haeupler, Hladík, Rozhoň, Tarjan and Tětek, 2023).

As our main result on bipartite sorting, we give a randomized algorithm that is within a factor of $O(\log^3(n + m))$ of being instance-optimal w.h.p., with respect to the neighborhood-based definition.

As our second contribution, we generalize bipartite sorting to DAG sorting, when the underlying DAG is not necessarily bipartite. As an unexpected consequence of a simple algorithm for DAG sorting, we rule out a potential lower bound on the widely-studied problem of *sorting with priced information*, posed by (Charikar, Fagin, Guruswami, Kleinberg, Raghavan and Sahai, STOC 2000). In this problem, comparing keys i and j has a known cost $c_{ij} \in \mathbb{R}^+ \cup \{\infty\}$, and the goal is to sort the keys in an instance-optimal way, by keeping the total cost of an algorithm as close as possible to $\sum_{i=1}^{n-1} c_{x(i)x(i+1)}$. Here $x(1) < \cdots < x(n)$ is the sorted order. While several special cases of cost functions have received a lot of attention in the community, no progress on the general version with arbitrary costs has been reported so far. One reason for this lack of progress seems to be a widely-cited $\Omega(n)$ lower bound on the competitive ratio for *finding the maximum*. This $\Omega(n)$ lower bound by (Gupta and Kumar, FOCS 2000) uses costs in $\{0, 1, n, \infty\}$, and although not extended to sorting, this barrier seems to have stalled any progress on the general cost case.

We rule out such a potential lower bound by showing the existence of an algorithm with a $\widetilde{O}(n^{3/4})$ competitive ratio for the $\{0, 1, n, \infty\}$ cost version. This generalizes the setting of *generalized sorting* proposed by (Huang, Kannan and Khanna, FOCS 2011), where the costs are either 1 or infinity, and the cost of the cheapest proof is always $n - 1$.

1 Introduction

The classic nuts-and-bolts problem, originally mentioned as an exercise in [27], asks: a disorganized carpenter has n nuts and n bolts, and there is (perfect) matching bolt for every nut. The only allowed comparison is between a nut and a bolt, and the result of such a comparison is either $<, =$ or $>$. The goal is to find the matching without comparing two nuts or two bolts to each other. A simple Quicksort type algorithm can be shown to solve this problem in optimal $O(n \log n)$ comparisons with high probability: Pick a random nut, compare to all bolts, find the matching bolt, and compare that bolt to all nuts. The problem is now partitioned into two subproblems with the match at the boundary; recurse. In a later work [3], Alon, Blum, Fiat, Kannan, Naor and Ostrovsky developed a deterministic Quicksort-type algorithm that uses expanders and performs $O(n \text{ polylog } n)$ comparisons. This was later improved to an optimal $O(n \log n)$ comparisons by Komlós, Ma, Szémeredi [24], by performing substantial modifications on AKS sorting.

Our starting point is a remark by Komlós, Ma and Szémeredi: "In particular, the fact that we can sort the nuts and bolts at all relies on the fact that there is a match between them." Indeed, if there is no matching (all comparisons come out $<$ or $>$), one realizes that the above randomized Quicksort based algorithm fails, as there is no partitioning into subproblems. The only case where sorting without a matching is possible is when nuts and bolts alternate in the final sorted order. Call this the perfectly interleaved case. Komlós, Ma and Szémeredi observed (in a private communication with Aumann) that their AKS sorting-based algorithm sorts the nuts and bolts using $O(n \log n)$ comparisons in this setting[1]. However, a general instance may not be perfectly interleaved, and this setting was left open.

Generalized Nuts and Bolts. We focus on the problem alluded to by Komlós, Ma, and Szémeredi: what if the carpenter is completely disorganized, and has an unequal collection of nuts and bolts, without any matching pairs[2]? That is, assume the carpenter has a set R of n nuts and a set B of m bolts, and is only allowed to compare a nut to a bolt, and the result is either $<, or >$. Unless $m = n$ and we are in the perfectly interleaved case, sorting $R \cup B$ is not possible: there could be two (or more) nuts (resp. bolts) that compare the same way to all the bolts (resp. nuts). A natural goal for the carpenter now is to "sort as much as you can", i.e., partition the set of nuts R into subsets $R_1, R_2, \ldots$ such that for any $r, r' \in R_i$ and any $b \in B$, r and r' have the same order with b (either both are smaller, or both are larger), and vice-versa (see figure 1). We term this generalization of nuts and bolts as *bipartite sorting*: given the complete bipartite graph $G = (R \cup B, E)$, the goal is to discover the orientation[3] on all the edges in E by querying as few of them as possible.

The need for fine-grained analysis. If all nuts are smaller than all bolts, it is clear that nm comparisons are needed by any algorithm, and obviously this number of comparisons suffices for any instance. Recall that the perfectly interleaved case can be solved (in fact

---

[1] For a simple randomized algorithm that does the same, see Appendix A.1 of full version [15].
[2] It can be observed that having some matchings in the input only makes it easier to solve.
[3] An edge $e = (u, v)$ in G is oriented as $\vec{uv}$ if $u < v$ in the underlying DAG.

Figure 1 An example output to an instance of the bipartite sorting problem. Continuous runs of incomparable nuts and bolts are called "stripes".

completely sorted) much faster with only $O(n \log n)$ comparisons. Assuming for simplicity that $m = \Theta(n)$, the inherent complexities of instances range in $[\tilde{O}(n), \Omega(n^2)]$. Is there a way to define instance-optimality for bipartite sorting that captures its variety of underlying instances, and is there a good instance-optimal algorithm?

It is worthwhile to imagine the DAGs for the above two instances: if all nuts are smaller than all bolts, the DAG $\vec{G} = (V = R \cup B, \vec{E})$ will have all edges oriented from R to B, whereas the DAG in the perfectly interleaved case will be the transitive closure of the oriented edges in the *directed Hamiltonian path corresponding to the sorted order of the nuts and the bolts*. A *transitive reduction* of a DAG is the fewest number of oriented edges that by transitivity imply all other orientations. For the first instance, the transitive reduction has size nm, whereas for the second case, it has size $n - 1$, and therefore if we ignore $\log^{O(1)} n$ factors, for both instances, the sizes of their transitive reductions matches their complexity. This immediately suggests a parametrization similar to an output-sensitive setting: the number of comparisons of a good algorithm should be close to (say, within log factors of) the size of the transitive reduction of the underlying DAG.

However, the following instance dashes all hopes of an algorithm that performs only roughly as many queries as there are edges in the transitive reduction: $n - 1$ nuts are all smaller than a special bolt b, which is smaller than a special nut r, which is smaller than all other $n - 1$ bolts. We call this the *"one-inversion"* instance. Even though the transitive reduction here has size $2n - 1$, any algorithm must perform n^2 comparisons to find the hidden special pair (r, b). Thus the gap between the transitive reduction and the inherent complexity of this instance is $\Omega(n)$, which is as large as the gap between the complexity of any two instances. Observe that the DAG for this instance is in some sense just "one-flip-away" from the all-nuts-smaller-than-all-bolts DAG, a phenomenon that will be important later. Given that the transitive reduction fails to capture the instances, we ask:

Is there another meaningful way to define instance-optimality for bipartite sorting that captures its variety of underlying instances, and is there a good instance-optimal algorithm?

2   Our Results and Technical Overview

In this paper, we answer the above question in the affirmative, and in the process of doing so, make unexpected progress on a widely-studied problem called generalized sorting (Huang, Kannan and Khanna [20]), which in turn is a special case of the sorting with priced information problem introduced by Charikar et al. [8]. We explain this connection first before stating our results.

DAG Sorting.   One can generalize bipartite sorting to DAG sorting where the set of allowed comparisons is represented by an arbitrary (not necessarily bipartite) graph $G = (V, E)$. The goal is still to discover all orientations in the underlying DAG $\vec{G} = (V, \vec{E})$ or equivalently, its transitive reduction, by querying as few edges of G as possible, where a query of an edge $(u, v) \in E$ returns $<$ or $>$. Edges not present in E cannot be queried.
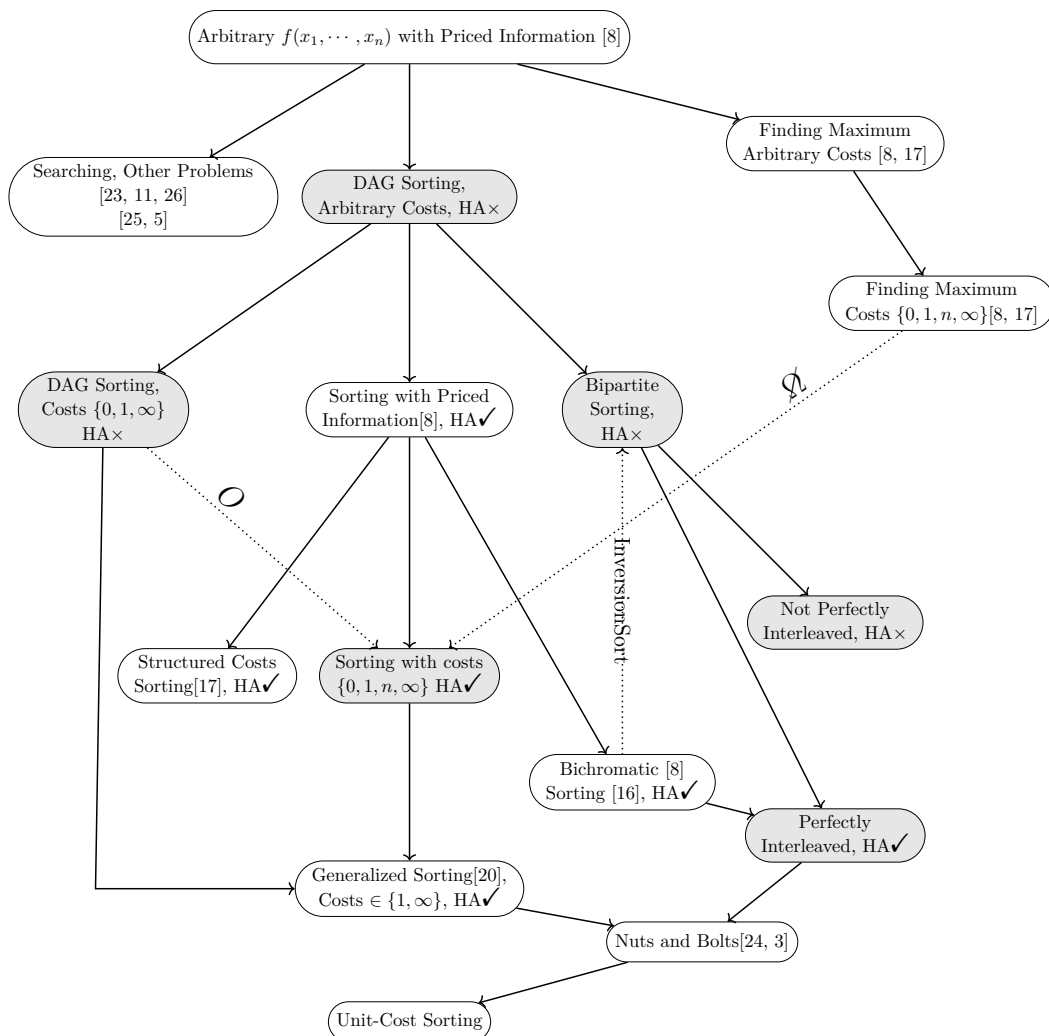
Figure 2 The landscape of sorting with priced information. Solid arrows go from a problem to its special case. HA✔indicates that the Hamiltonian path is assumed to exist and HA× indicates that a Hamiltonian path may not exist. Problems shaded in gray are introduced or studied in this paper for the first time. Dotted arrows highlight our results, arrows with O show algorithms carrying over from one problem to another, and $\not{O}$ show lower bounds not carrying over.

It turns out that under the promise that $\vec{G} = (V, \vec{E})$ has a directed Hamiltonian path, DAG sorting is exactly equivalent to generalized sorting. Thus both bipartite sorting[4] and generalized sorting can be viewed as special cases of DAG sorting. In fact, one can go a step further and assign non-negative costs to the edges in E in DAG sorting, and now ask for algorithms that find the transitive reduction with the cheapest cost. If the underlying DAG has a Hamiltonian path, this is exactly the problem of sorting with priced information.

---

[4]   And by transitivity, sorting nuts and bolts when perfectly interleaved, matching nuts and bolts, and classical sorting.

2.1 Bipartite Sorting

As our main result, we define a meaningful notion of *instance optimality* for bipartite sorting, and give an algorithm that is instance optimal up to a factor of $O(\log^3(n))$. We found this (definition, algorithm) result surprising because it shows that in some sense, the "one-flip" phenomenon (recall the "one-inversion" instance) mentioned in the introduction is the *only* obstruction to achieving (almost) instance optimality. We believe our definition may be of independent interest as it unifies several previously-studied notions of instance optimality.

2.1.1 Defining Instance-Optimality

The transitive reduction of $\vec{G}$, denoted as $\vec{T}$, can be thought of as the "cheapest proof" of the underlying DAG being $\vec{G}$. Such a comparison to the cheapest proof has historically been very useful in defining instance optimality. Indeed, for the problems of generalized sorting and sorting with priced information, when the transitive reduction equals the directed Hamiltonian path, the cost of this directed Hamiltonian path is the measure of instance-optimality (and the factor by which an algorithm exceeds it is called *competitive ratio* in the original work by Charikar et al.[8]). The cheapest proof appears again in the work by Demaine, López-Ortiz and Munro [10], who study the problems of comparison-based set unions, intersections and differences. However in our setting, as the one-inversion instance shows, the size of the transitive reduction $|\vec{T}|$ is *too strong a benchmark* to compare the performance of an algorithm to.

An important work on instance optimality deviating from a comparison to cheapest proof is by [Afshani, Barbay and Chan,[1]] which studies instance optimal algorithms for the convex hull problem. This was also adopted by [Cardinal, Dallant and Iacono [6]], who studied bichromatic rectangular visibility. When x is an input sequence of points, [1] and [6] define $OPT(x)$ to be the runtime of an algorithm that is *order-oblivious*, i.e., OPT's code can depend on the set of elements in x, but not on their order. The fact that *some* restriction on OPT is needed (at least for static problems) follows because comparing the runtime of an algorithm A on an input x to an algorithm (OPT) that is "tailor-made" for an input x is too strong: if OPT's code can depend on x, $OPT(x)$ can be very small compared to $A(x)$. In the convex hull problem $OPT(x)$ would just be $O(n)$ as OPT only needs to read the input[5], and for DAG/bipartite sorting if OPT knows the underlying DAG $\vec{G}$ then (since we only count query complexity) OPT equals zero!

We posit a *"neighborhood-based"* approach to defining notions of instance optimality. In a nutshell, we observe that most definitions for instance optimality boil down to choosing the "right" neighborhood. The smaller the neighborhood is, the stronger the notion of instance-optimality, and the harder it is to attain. If the neighborhood is the set of all instances, we are back to worst-case analysis. Thus the art is in choosing the smallest meaningful neighborhood that still allows for instance optimal algorithms. This general step-by-step process of increasing the neighborhood until one sees hope for instance-optimality reveals quite a bit about the fine-grained nature of a problem. Perhaps this way of looking at instance-optimality was already known, but since we have not seen it in print, we show (see Appendix C of the full version of this paper [15]) how most of the existing works on instance-

---

[5] [Afshani, Barbay and Chan [1]] state: "For example, consider the 2D convex hull problem, which has $\Theta(n \log n)$ worst-case complexity in the algebraic computation tree model: for every input sequence of n points, one can easily design an algorithm A (with its code depending on the input sequence) that runs in $O(n)$ time on that particular sequence, thus ruling out the existence of an instance-optimal algorithm."

optimality, ranging from the classical works on adaptive sorting and presortedness [13], to later works by Demaine et al. [10] and Afshani et al.[1], and including the very recent work on universal optimality [18], all agree with this paradigm.

We define a neighborhood of the underlying DAG $\vec{G}$, denoted as $\mathcal{N}(\vec{G})$. See Definition 8 for a precise definition but intuitively, this is the set of all DAGs that are either isomorphic, or "one-flip" away from $\vec{G}$. Define the runtime $T_A(\vec{G})$ of a randomized algorithm A on an instance $\vec{G}$ as its expected comparison-cost on instance $\vec{G}$. Let $\mathcal{C}(\vec{G})$ be the set of randomized (Las Vegas) algorithms that are correct for all instances in $\mathcal{N}(\vec{G})$. Now define

$$\text{OPT}(\vec{G}) = \inf_{A \in \mathcal{C}(\vec{G})} \max_{\vec{G}' \in \mathcal{N}(\vec{G})} T_A(\vec{G}').$$

For $\alpha \geq 1$, we say an algorithm A is α-instance optimal if for every instance $\vec{G}$, $T_A(\vec{G}) \leq \alpha \, \text{OPT}(\vec{G})$. Does there exist an algorithm achieving $\alpha = O(1)$, or $\alpha = \log^{O(1)} n$?

2.1.2   InversionSort: An Almost Instance Optimal Algorithm

The algorithm we investigate for bipartite sorting is a variant of an algorithm InversionSort that was recently presented by us (Goswami and Jacob [16]) for the version when there is a sorted order on the nuts and bolts, and nuts can be compared to nuts (at a cost $\alpha > 1$) and bolts can be compared to bolts (at a cost $\beta > 1$). In bipartite sorting, two nuts (or two bolts) cannot be compared to each other, i.e., $\alpha = \beta = \infty$, but due to the similarity to the previous algorithm we call the algorithm for bipartite sorting InversionSort too. Note that in [16] the previous algorithm was compared to the cost of the Hamiltonian, whereas now not only may the Hamiltonian cease to exist, its natural counterpart (the transitive reduction) is a weak lower bound.

As a first try, let us see what goes wrong when we apply the simple randomized QuickSort algorithm for nuts and bolts to bipartite sorting. We pick a random nut r and use it to pivot the bolts obtaining two sets $B_{<r}$ and $B_{>r}$. Since no match was obtained, in the alternating step we can select a random bolt b, say from $B_{>r}$, and pivot the nuts, obtaining $R_{<b}$ (containing r) and $R_{>b}$. Instead of the two perfectly-partitioned subproblems obtained in nuts and bolts, we unfortunately now have three subproblems: $(R_{<b}, B_{<r})$, $(R_{<b}, B_{>r})$, and $(R_{>b}, B_{>r})$.

If the nuts and bolts were perfectly interleaved, we show (see Appendix A.1 of the full version of this paper [15]) that a BFS-style Quicksort algorithm that we call BackboneSort sorts using $O(n \log n)$ comparisons. BackboneSort works in a sequence of alternating phases, a nut-phase and a bolt-phase. In a nut phase it tries to refine the nut subproblems and makes progress over the three subproblems above in a round-robin fashion: for example, to make progress in $(R_{<b}, B_{<r})$, it would select a random bolt in $B_{<r}$ and pivot $R_{<b}$, and vice-versa in a bolt-phase.

Unfortunately, in a general instance of bipartite sorting the nuts and bolts are not perfectly interleaved everywhere, and it turns out that BackboneSort can perform badly (see Theorem 29 of full version [15]). The reason for this behavior arises from selecting a "random" pivot in the subproblem: for the perfectly interleaved instance, its sub-instances are also perfectly interleaved (in particular all three subproblems $(R_{<b}, B_{<r})$, $(R_{<b}, B_{>r})$, and $(R_{>b}, B_{>r})$ can be shown to be roughly a constant fraction of the original problem with good probability) , and then such a random pivot is guaranteed to be good, just like in randomized Quicksort. However, if the underlying instance is lop-sided (think of many nuts in $R_{<b}$ sandwiched between two bolts in $B_{<r}$), a random bolt is not a good pivot. To add to this complication, an instance may be lop-sided in one region and perfectly interleaved in another, and an instance-optimal algorithm should ideally be able to detect such a situation.

InversionSort proceeds similarly to BackboneSort, but instead of selecting random pivots, it performs a random comparison: when trying to make progress in $(R_{<b}, B_{<r})$, it will look for "inversions", defined as a pair $(r, b) \in R_{<b} \times B_{<r}$ such that $b < r$. When it finds such a pair, it uses them as pivots. The intuition is that regions having "easy" subinstances (like perfectly interleaved ones) will resolve fast, whereas those that are lop-sided, like all nuts less than all bolts, will take longer, but this is a necessary task in this sub-region.

The following theorem quantifies the performance of InversionSort. Here, $N = n + m$ and Definition 8 is our precise definition of instance optimality, sketched in the previous section.

▶ **Theorem 1** (Instance Optimality of InversionSort). *There exists a constant $c > 0$, such that for every instance $\mathcal{I}$, the cost of InversionSort on $\mathcal{I}$ is, with probability at least $1 - 1/N$, at most $c(\log N)^3 \mathrm{OPT}(\mathcal{I})$, where $\mathrm{OPT}(\mathcal{I})$ is as in Definition 8.*

Thus InversionSort is $O(\log^3 N)$-instance optimal, with respect to a natural notion of instance-optimality that accounts for the one-flip neighborhood of the underlying DAG. We conjecture that InversionSort is actually $O(1)$-instance optimal with this notion[6].

2.2 Unexpected Result: Generalized Sorting and Sorting with Priced Information

While Theorem 1 gives an algorithm for bipartite sorting that is instance-optimal, what can we say for DAG sorting? Unfortunately proving a polylog-instance optimality guarantee seems out of reach for now. The reason is that even the "sortable" case of DAG sorting, where the DAG has a directed Hamiltonian path, has a current best bound of $\tilde{O}(n^{1.5})$ comparisons, or in other words, is a factor $\tilde{O}(\sqrt{n})$ away from the Hamiltonian cost (which is $n - 1$). This results from an interesting randomized algorithm by Huang, Kannan and Khanna [20] that uses the work by Kahn and Linial [22] on balancing extensions via the Brunn-Minkowski theorem.

For a DAG that is not sortable, we extend the algorithm of Huang, Kannan and Khanna to give an algorithm that performs $\tilde{O}(\min(wn^{1.5}, n^2))$ comparisons and outputs the transitive reduction of $\vec{G}$ (see Theorem 20). Here w denotes the width of the DAG, which is the size of its largest antichain (a set of incomparable elements). We now point out an unexpected consequence of this result.

The problem of *sorting with priced information* introduced by Charikar, Fagin, Guruswami, Kleinberg, Raghavan and Sahai, [8], is a generalization of the classical, unit-cost, comparison-based sorting, defined as follows. The input is a weighted undirected graph G on n vertices, with the cost $c_{ij} \in \mathbb{R}_{\geq 0}$ on the edge e_{ij} indicating the cost to compare keys (represented by vertices) v_i and v_j. As before, edges not in G have cost ∞ and cannot be queried, and a query on an edge e_{ij} reveals if $v_i < v_j$ (indicated as $\vec{e}_{ij}$) or $v_i > v_j$ (indicated as $\vec{e}_{ji}$).

Since the hidden Hamiltonian path $\mathcal{H}$ is the cheapest proof, its cost which equals $\sum_{i=1}^{n-1} c_{x(i)x(i+1)}$ is a lower bound. Here $x(1) < \cdots < x(n)$ is the sorted order. [8] propose finding algorithms that come as close to the cost of $\mathcal{H}$ as possible. The *competitive ratio* is defined as the ratio of the cost of the algorithm to the cost of $\mathcal{H}$, and the goal is to find an algorithm with small competitive ratio.

Several special cases of cost functions have been studied, but for arbitrary costs, almost nothing is known about the above problem. Note that the $\tilde{O}(n^{1.5})$ result by Huang, Kannan and Khanna [20] works when all costs are either 1 or ∞. What about the version with

---

[6] As evidence, we prove in Theorem 28 in Appendix A.2 of full version [15] that InversionSort solves the perfectly interleaved instance in an optimal $O(n \log n)$ comparisons.

arbitrary costs? Many works state that the general-cost version is "arbitrarily bad" (Huang, Kannan and Khanna [20]), "bleak" or "hopeless" (Gupta and Kumar [17]). The only evidence for this is an $\Omega(n)$ lower bound on the competitive ratio of any algorithm that *finds the maximum.* There is an example where the costs are either 0, 1, n or ∞, and one can show that any algorithm that finds the maximum element m must have cost $\Omega(n)$ times that of the cheapest proof (of m being the maximum) (Charikar et al. [8], Hartline et al. [19], Gupta and Kumar [17]). While it certainly provided intuition, the $\Omega(n)$ lower bound for maximum (with costs in $\{0, 1, n, \infty\}$) was never extended to sorting.

This makes the $\{0, 1, n, \infty\}$ version interesting due to three reasons:

- do instances in this cost regime contain an $\Omega(n)$ lower bound for sorting too?
- it is the natural step-up from generalized sorting with costs in $\{1, \infty\}$, and
- this is the first instance with forbidden comparisons that requires an *instance-specific analysis* for the competitive ratio. For generalized sorting and for stochastic sorting[7], the cost of the Hamiltonian is always $n - 1$, so one only has to bound the cost of an algorithm, without worrying about the underlying instance. However, when costs are in $\{0, 1, n, \infty\}$ the cost of the Hamiltonian can range from 0 to $n(n - 1)$, and so an algorithm must adapt to the underlying instance.

Our second theorem addresses this cost version of sorting with priced information, showing that it cannot be the $\{0, 1, n, \infty\}$ version that makes sorting hopeless!

▶ **Theorem 2.** *Consider the problem of sorting when every comparison has a cost in $\{0, 1, F, \infty\}$, for any $F \geq n^{3/4}$. There exists a polynomial time randomized algorithm whose competitive ratio is $\widetilde{O}(n^{3/4})$, with high probability.*

The main ingredient in the proof of this theorem is the aforementioned $\tilde{O}(\min(wn^{1.5}, n^2))$ comparisons algorithm for DAG sorting. Even though DAG sorting does not promise a Hamiltonian, it turns out to be useful because it can be used as a subroutine in a "greedy" algorithm for the $\{0, 1, n, \infty\}$ cost version: obtain with cheapest cost the partial DAG formed by all costs 0 and 1 comparisons.

Organization.   We state our problems precisely in Section 3. This is followed by our result on bipartite sorting (Theorem 1) in Section 4, and our result on sorting with priced information (Theorem 2) in Section 5.

3   Problem Definitions

We formally define the problem of bipartite sorting first, and then the problems of DAG sorting and sorting with priced information. We invite the reader to see Figure 2 for the landscape of these problems, their relations to each other, and how our results fit in this landscape.

▶ **Definition 3** (Bipartite Sorting). *Input: A complete bipartite undirected graph G of unit costs on $V = R \cup B$. Only edges in G can be queried (at unit cost), and querying an undirected edge (u, v) has one of two outcomes, $u < v$ (implying $\vec{uv} \in \vec{G}$) or $u > v$ (implying $\vec{vu} \in \vec{G}$). $N := |V|$.*

---

[7] Also initiated by Huang, Kannan and Khanna [20], this is the version where the input graph G is random

The instance of bipartite sorting is defined by a partition of R (reds) and B (blues) into stripes $(S_1, \cdots, S_k)$ (Figure 1), i.e., by the relative order[8] between the reds and the blues. Note that S_is are unordered sets. The DAG $\vec{G}$ has, for all $1 \le i \le k-1$, an edge from every element in S_i to every element of the other color greater than it, i.e., to every element in $\cup_{\ell \ge 0} S_{i+(2\ell+1)}$.
Output: The sequence of stripes $(S_1, \cdots, S_k)$ (see figure 1). Equivalently, the transitive reduction of $\vec{G}$.

DAG Sorting. Let $\mathcal{P}$ denote a partial order on a set of n elements, and let $\vec{P}$ denote the transitively closed DAG on $V = \{v_1, \cdots, v_n\}$ indicating all order relations in $\mathcal{P}$. That is, the vertex v_i is identified with element i, the edge $\overrightarrow{e_{ij}}$ between vertices i and j exists if $v_i < v_j$, the edge $\overleftarrow{e_{ij}}$ exists if $v_i > v_j$, and no edge between v_i and v_j exists if elements i and j are incomparable. For convenience, let P denote the set of (undirected) edges in $\vec{P}$ without their directions. Let w denote the width of $\vec{P}$.

▶ **Definition 4** (Implied and essential edge, transitive reduction [2]). *Given a DAG $\vec{G} = (V, \vec{E})$, an edge $(u, v) \in \vec{E}$ is implied, if there is a directed path in $\vec{G}$ from u to v. Otherwise, (u, v) is essential. The set of essential edges is called the transitive reduction of $\vec{G}$.*

Note that for every implied edge $(u, v) \in \vec{E}$, there is a directed u to v path of essential edges in $\vec{G}$. It turns out that the transitive reduction of a DAG is unique [2]. Let $\vec{\mathcal{T}}$ denote the transitive reduction of $\vec{P}$. Let $\mathcal{T}$ denote the undirected version of $\vec{\mathcal{T}}$.

▶ **Definition 5** (DAG Sorting, arbitrary costs). *Input: An undirected graph $G = (V, E \subset P)$ with costs $c_{ij} \in \mathbb{R}_{\ge 0}$ on edges. An oracle that answers, given an undirected edge e_{ij} of G, its orientation in $\vec{P}$.*

Promise: $\mathcal{T} \subset E$. That is, the queryable edges contain the edges of the transitive reduction.
Output: $\vec{\mathcal{T}}$.

Cost of an algorithm $\mathcal{A}$: The total cost of the edges queried by $\mathcal{A}$ on the instance $\vec{P}$. This will be denoted by $cost(\mathcal{A}, \vec{G})$, where $\vec{G}$ is the directed version of G (also referred to as the instance from now on), and contains all the information about $\vec{P}$.

When $c_{ij} = 1$ for all i, j, we call the problem simply DAG Sorting.

We will only care about the query cost of the algorithm, and while there may be a compact representation of $\vec{\mathcal{T}}$, we ask the algorithm to output $\vec{\mathcal{T}}$ in its entirety for simplicity.

DAG sorting generalizes sorting with priced information, which we describe next. This problem was introduced by [8] in the broader context of querying with priced information, where one wants to compute a function f of n inputs, and querying an input has a certain cost associated to it. The competitive ratio is defined as the (worst case) ratio of the cost of the query strategy to the *cost of the cheapest proof of f*. This work initiated a multitude of papers on priced information, studying problems like learning with attribute costs [23], stochastic boolean function evaluation [11], searching on trees [26, 25], and priced information in external memory [5].

▶ **Definition 6** (Sorting with Priced Information [8]). *Sorting with priced information is a special case of DAG sorting, when the partial order $\mathcal{P}$ is a total order. In this case, $\vec{P}$ is a tournament and therefore P is a complete graph. $\vec{\mathcal{T}}$ is simply a directed Hamiltonian path*

---

[8] S_1 is the set of all sources in the DAG $\vec{G}$; since red and blue elements can be compared with unit-cost, S_1 must necessarily be of one color. S_i can be iteratively defined as the set of sources in $\vec{G}$ after all stripes 1 to $i-1$ have been deleted.

$\mathcal{H}$ *in $\vec{P}$. The input G is any graph on n vertices containing the edges of $\mathcal{H}$ without their directions, and the output is $\mathcal{H}$. The total cost of the edges queried by an algorithm $\mathcal{A}$ on the instance $\vec{G}$ will be denoted by $cost(\mathcal{A}, \vec{G})$.*

Competitive ratio for sorting with priced information. The **competitive ratio** of A (as defined in [8]) is $\rho(\mathcal{A}, n) = \max_{\vec{G}} cost(A, \vec{G})/cost(\mathcal{H})$, where the maximum is taken over all instances $\vec{G}$ of n vertices with a total order. The goal is to find sorting algorithms with small competitive ratio. Here $cost(\mathcal{H})$ is considered as a proxy for the complexity of the instance $\vec{G}$, as it is the cheapest proof. It certainly is a valid lower bound, for the edges on the Hamiltonian *must* be queried by any algorithm.

For example, when G is the complete unit-cost graph, MergeSort or QuickSort achieve a competitive ratio of $\Theta(\log n)$ (the latter w.h.p. if randomized). Similarly, when G is unit-cost but not complete, the $\tilde{O}(n^{1.5})$ cost algorithm algorithm by Huang, Kannan and Khanna [20] has a competitive ratio of $\tilde{O}(\sqrt{n})$. As mentioned, very little is known about the case when G has arbitrary costs.

We end with the remark that DAG sorting is closely related to a line of work initiated by Faigle and Turán,[14] called sorting a partial ordered set, or identifying a poset. This was followed up by several works such as [9] and [12]. For a survey on this line of work that also includes generalized sorting, we refer the reader to [7].

4 Results on Bipartite Sorting

This section is divided into three subsections. In Section 4.1 we formally state our definition of instance-optimality. In Section 4.2 we derive some lower bounds on OPT stemming from the definition of instance-optimality. Finally, in Section 4.3 we define InversionSort, and prove that it comes close to achieving instance-optimality (Theorem 1) by charging the comparisons performed by InversionSort to the derived lower bounds.

4.1 Defining Instance-Optimality

As mentioned in the introduction, the following instance of bipartite sorting shows that comparing the cost of an algorithm to the transitive reduction is hopeless.

▶ **Definition 7** (One-inversion Instance). *Let $G = (R, B, E)$ be the undirected complete bipartite graph on $|R| = |B| = n/2$. Pick an arbitrary $r \in R$, $b \in B$, let $R_{-r} = R \setminus \{r\}$ and $B_{-b} = B \setminus \{b\}$. Define a DAG $\vec{G}$ via its transitive reduction as follows.*

$$TR(\vec{G}) = \{\vec{xb} : x \in R_{-r}\} \cup \{\vec{br}\} \cup \{\vec{ry} : y \in B_{-b}\}.$$

The transitive reduction has size $O(n)$, but any algorithm must spend $\Omega(n^2)$ comparisons to identify r and b. Thus the "cheapest proof" is too strong a benchmark. We now present our neighborhood-based approach for bipartite sorting. This approach is general, and in the Appendix C of the full version [15], we show how this neighborhood-based approach fits several works on instance-optimal algorithms for static problems, namely the works on adaptive sorting [13], on set intersection, union and difference [10], and the recent work on universal optimality [18].

We start with small neighborhoods and gradually increase them until there is no immediate obstruction to instance-optimality. Define $\mathcal{N}_{\mathcal{A}}(\vec{G})$ as the set of DAGs (Automorphic) isomorphic to $\vec{G}$ if all edges in $\vec{G}$ are unit-cost, and cost-isomorphic[9] otherwise. Next, define

---

[9] Meaning there exists an isomorphism btween the DAGs that preserves the costs.

the runtime of an algorithm on an instance $\vec{G}$ as its maximum comparison-cost on any instance in $\mathcal{N}_{\mathcal{A}}(\vec{G})$. Define $OPT_{\mathcal{A}}(\vec{G})$ as the smallest comparison-cost of any algorithm.

For unit-cost sorting, $\vec{G}$ is a complete DAG, and $|\mathcal{N}_{\mathcal{A}}(\vec{G})| = n!$. It is easily seen that now $OPT(\vec{G}) = \Omega(n \log n)$, and the unnecessary $\log n$ gap arising from comparing to the cheapest proof vanishes. This is also consistent with the fact that any $O(n \log n)$ algorithm that ignores the sequence of keys and only treats them as a set is $O(1)$ instance optimal in the order-oblivious setting.[10]

Moving on to the case when $\vec{G}$ is not complete, we see that the above definition is not sufficient by the following observation. Consider the case when $\vec{G}$ is a complete bipartite graph, with all edges going in the same direction, i.e., from one partition R to the other B. Now $|\mathcal{N}_{\mathcal{A}}(\vec{G})| = 1$, and any algorithm that knows that it is operating on $\vec{G}$ has zero comparison cost[11]. However, any instance-unaware algorithm needs $\Omega(|R||B|)$ comparisons to verify that the instance is indeed $\vec{G}$. This suggests that we need a larger neighborhood than just $\mathcal{N}_{\mathcal{A}}(\vec{G})$.

Let $\vec{E}$ denote the set of edges in the transitive reduction of $\vec{G}$, also called essential edges (Definition 4). Define $\mathcal{N}_E(\vec{G})$ as the set of DAGs that differ from $\vec{G}$ in exactly one essential edge being flipped, and any other changes it may imply. Again, define the runtime of an algorithm on an instance $\vec{G}$ as its maximum comparison-cost on any instance in $\mathcal{N}_E(\vec{G})$, and define $OPT_{\mathcal{A}}(\vec{G})$ as the smallest comparison-cost of any algorithm. It is straightforward now to observe that if $\vec{G}$ is sortable, $OPT(S) \geq \text{cost}(\mathcal{H})$ and if $\vec{G}$ is not sortable, $OPT(S) \geq \text{cost}(TR(\vec{G}))$. *Thus, we recover both definitions of competitive ratio by considering this one-flip neighborhood.*

The algorithms we consider here are randomized, and we hence want a definition of instance optimality that allows for randomization. The notion of "instance optimal in the random-order setting" of [1], based on Yao's principle, compares implicitly to the optimal expected running time of a correct randomized algorithm (Las Vegas style). Our following definition does this directly:

▶ **Definition 8** (α-Instance Optimality)**.** *Let $\mathcal{N}(\vec{G}) = \mathcal{N}_E(\vec{G}) \cup \mathcal{N}_{\mathcal{A}}(\vec{G})$. Define the runtime $T_A(\vec{G})$ of a randomized algorithm A on an instance $\vec{G}$ as its expected comparison-cost on instance $\vec{G}$. Let $\mathcal{C}(\vec{G})$ be the set of randomized (Las Vegas) algorithms that are correct for all instances in $\mathcal{N}(\vec{G})$. Define $\text{OPT}(\vec{G}) = \inf_{A \in \mathcal{C}(\vec{G})} \max_{\vec{G}' \in \mathcal{N}(\vec{G})} T_A(\vec{G}')$. For some $\alpha \geq 1$, an algorithm A is called α-instance optimal if for every instance $\vec{G}$, $T_A(\vec{G}) \leq \alpha \, \text{OPT}(\vec{G})$.*

4.2 Lower Bounds on OPT

We first refine our notion of instance-optimality to make it more amenable to deriving lower bounds.

▶ **Definition 9** (Instance Optimality Distribution)**.** *Let $\mathcal{C}'(\vec{G})$ be the set of deterministic algorithms that are correct for all instances in $\mathcal{N}(\vec{G})$. Let $\mathcal{D}(\vec{G})$ be the uniform distribution over $\mathcal{N}_{\mathcal{A}}(\vec{G})$. Define $\text{OPT}(\vec{G}) = \inf_{A \in \mathcal{C}'(\vec{G})} \mathbb{E}_{\vec{G}' \sim \mathcal{D}(\vec{G})} T_A(\vec{G}')$.*

An application of Yao's principle [28] shows that for any $\vec{G}$: $\text{OPT}_D(\vec{G}) \leq \text{OPT}(\vec{G})$. Note that the optimal algorithm is allowed to depend on $\mathcal{I}$. We remark that while the following lower bounds would be easy to prove for algorithms unaware of $\mathcal{I}$ using adversary arguments, we prove this for OPT (Definition 9), which requires some extra care.

---

[10] All algorithms that exploit certain presortedness in the input necessarily exploit the input sequence of keys. This corresponds to a smaller neighborhood than the $n!$ size automorphism neighborhood.

[11] Recall that we do not charge to write down the transitive reduction which has size $O(|R||B|)$, only the query cost.

▶ **Lemma 10** (Transitive Reduction or Verification lower bound). *Let $\mathcal{I}$ be an instance of bipartite sorting, let $K \subset \vec{E}_{\mathcal{I}}$ be its transitive reduction, and define $C_V = |K|$. Then, any algorithm that is correct for all inputs from $\mathcal{N}(\mathcal{I})$ must perform at least C_V comparisons.*

Proof. Assume there exists an algorithm $\mathcal{A}$ that is correct for all instances in $\mathcal{N}(\mathcal{I})$ simultaneously that performs at most $C_V - 1$ comparisons on input I. This means that there must exist an edge e on the transitive reduction that is not verified by $\mathcal{A}$. As $\mathcal{A}$ is deterministic, it would report $\mathcal{I}$ as output even when the input had e flipped because all other edges have the same direction as in $\mathcal{I}$, as we will argue now: If this (non-flipped) edge is between two stripes of size one, the two endpoints merge into two other stripes, but no edges changes direction. If this edge is between two stripes and both of them have size at least 2, then we create one additional inversion without changing other directions. If one stripe has size one and the other has size at least two, one element is moved from the size-at-least-two stripe to a neighboring stripe. Again, no bichromatic edges (implied or not) change their direction. ◀

While the above lower bound is natural as it is the cheapest proof, Definition 9 now allows for the following lower bound that captures the complexity of instances where transitive reduction is too weak a measure (recall the instance in Definition 7).

▶ **Lemma 11** (Inversion finding lower bound). *Let $\mathcal{I}$ be an instance of bipartite sorting with $n \geq 2$ red and $m \geq 2$ blue elements, where not all comparisons come out the same, and define*

$$C_I = \frac{nm}{\min(|\{(r,b) \in R \times B \mid r < b\}|, |\{(r,b) \in R \times B \mid r > b\}|)}.$$

Under the distribution of Definition 9, any deterministic algorithm $\mathcal{A}$ that does at most $C_I/2$ comparisons must fail with probability (at least) $1/8$.

Proof. Let $\mathcal{D}$ be the uniform distribution over $\mathcal{N}_A(\mathcal{I})$. Remember that $\mathcal{N}_A(\mathcal{I})$ contains all instances where the stripes are internally arbitrarily permuted. Observe $|\mathcal{N}_A(\mathcal{I})| \geq 4$ by the bounds on n and m. W.l.o.g., assume $|\{(r,b) \in R \times B \mid r < b\}| < |\{(r,b) \in R \times B \mid b < r\}|$, i.e., the usual outcome of a comparison is $b < r$ and the inversion is $r < b$. Let $p = 1/C_I = |\{(r,b) \in R \times B \mid r < b\}|/nm \leq 1/2$ be the probability that a randomly chosen pair of elements is an inversion. By Yaos principle, let $\mathcal{A}$ be a deterministic algorithm and think of it as a decision tree $\mathcal{T}$ where nodes are red-blue comparisons and non-inversion go to the left, inversions go to the right. Each leaf of the tree is marked with an output (that declares which instance was represented by the input), or a failure output.

Claim. Let v_k be the node v that is reached by k comparisons returning "non-inversion" (i.e. the leftmost node of $\mathcal{T}$ at depth k). When input is drawn from $\mathcal{D}$, the node v_k is reached with probability at least $1 - kp$.

Proof of Claim. An input reaches v_k if k (potentially dependent) random experiments all came out as "non-inversion", each having a probability $1 - p$. The claim follows from a union bound over the fail events. ◁

From the claim it follows that if v_k is a leaf for $k \leq C_I/2$, the algorithm must fail with probability at least $1/4$: Then $kp \leq C_I/2 \cdot 1/C_I = 1/2$ and $1 - kp \geq 1/2$, so half of the inputs reach v_k. Because there are at least four inputs, at least two reach v_k, but it can only be labeled with one, the other(s), which stand for at least $1/4$ of the inputs in $\mathcal{N}_A(I)$, make the algorithm fail. ◀

Finally, we will need to combine various lower bounds from different subproblems. Let $\mathcal{I}$ be an instance and $(S_1, \cdots, S_k)$ its stripes (see Figure 1). Consider pairs of indices $(a_1, b_1), \cdots, (a_\ell, b_\ell)$, where for all $1 \le j \le \ell$, a_j and b_j both belong to $\{1, \cdots, k\}$, and $a_j < b_j < a_{j+1} < b_{j+1}$ for all $j < \ell$. For $1 \le j \le \ell$ define the subinstance $\mathcal{I}_j$ by the subgraph of $\vec{G}_\mathcal{I}$ on the vertices $V_j = \bigcup_{i=a_j}^{b_j} S_i$.

▶ **Lemma 12** (Decomposition into Lower Bounds for Subproblems). *For $1 \le j \le \ell$, let $\mathcal{I}_j$ be a subinstances of $\mathcal{I}$ as above. Then* $\mathrm{OPT}_D(\mathcal{I}) \ge \sum_{j=1}^\ell \mathrm{OPT}_D(\mathcal{I}_j)$.

Proof. As we are working with Definition 9, we first have to check that an algorithm that is correct on $\mathcal{N}(\mathcal{I})$ is actually correct on each $\mathcal{N}(\mathcal{I}_j)$. To this end observe that every edge flip in $\mathcal{I}_j$ is also an edge flip in $\mathcal{I}$, and that any permutation of the labels/names in $\mathcal{I}_j$ is also a permutation in $\mathcal{I}$.

Run the algorithm A on an instance $\mathcal{I}'$ drawn from $\mathcal{D}(\mathcal{I})$, and let X be the random variable describing the number of comparisons of A. Define the random variables X_j to be the number of comparisons between vertices in subproblem $\mathcal{I}'_j$. Then $\sum X_j \le X$.

Let's conceptually draw $\mathcal{I}'$ from $\mathcal{D}(\mathcal{I})$ by first finding a position in the input list (name) for all vertices not in $\mathcal{I}_j$ and finally draw names for the vertices in $\mathcal{I}_j$. Now we can think of A as deterministic algorithm for $\mathcal{I}_j$ by considering all comparisons not in $\mathcal{I}_j$ as fixed, and we get $E[X_j] \ge \mathrm{OPT}_D(\mathcal{I}_j)$. The statement of the lemma follows by linearity of expectation. ◀

4.3 InversionSort and its $O(\log^3 n)$ instance-optimal guarantee

A generic state of InversionSort will be defined using a *backbone*, which is a sequence of elements of alternating colors, called *representatives* or *pivots*. Each representative will be assigned a *bucket*, which is a set of elements of the same color that lie between the two neighboring representatives of the other color on the backbone.

InversionSort makes progress from one state to the next by performing three steps: a) finding an *inversion* (which is defined soon) between neighboring representatives on the backbone, b) inserting this inversion on the backbone, and c) pivoting with these two elements, thereby refining the buckets.

4.3.1 Description of InversionSort

▶ **Definition 13** (Backbone, Representatives, and Buckets). *The backbone consists of a totally ordered, alternating list of* representatives $(u_0, u_1, u_2, \ldots, u_{2k}, u_{2k+1}) = (r_0, b_1, r_2, \ldots, r_{2k}, b_{2k+1})$, *where $r_{2i} \in R$ and $b_{2j+1} \in B$ with $r_i < b_{i+1}$ and $b_i < r_{i+1}$. Here, r_0 is an artificial red element that is smaller than all elements, and the last element b_{2k+1} is an artificial blue element that is larger than all elements. The representatives define the* buckets $(X_0, X_1, X_2, \ldots, X_{2k}, X_{2k+1}) = (R_0, B_1, R_2, \ldots, R_{2k}, B_{2k+1})$ *by $R_i = \{x \in R \mid b_{i-1} < x < b_{i+1}\} \setminus \{r_i\}$ and $B_i = \{x \in B \mid r_{i-1} < x < r_{i+1}\} \setminus \{b_i\}$. Here, as a convention, the representative is not included in the bucket. Again, $R_0 = \{x \in R \mid x < b_1\}$ and $B_{2k+1} = \{x \in B \mid r_{2k} < x\}$ are special cases.*

▶ **Definition 14** (active subproblems and buckets). *As long InversionSort did not create a certificate that there are no further inversions between x_i and x_{i+1}, the subproblem defined by the buckets X_i and X_{i+1} is called active, and so are X_i and X_{i+1}.*

Our previous work [16] now defines an *inversion*, which gives the algorithm its name. Consider adjacent representatives u_i and u_{i+1}, their corresponding adjacent buckets X_i and X_{i+1}, and a bichromatic pair (x, y) of elements $x \in X_i$ and $y \in X_{i+1}$. Observe that x and

🟨 **Algorithm 1** Algorithm InversionSort.

Require: elements R red, B blue
   create trivial backbone $\mathcal{B}$ from R and B, see Definition 13
   $\eta \leftarrow 0$
   while there is an active subproblem (see Definition 14) in $\mathcal{B}$ **do**
      $\eta \leftarrow \eta + 1$
      for each active (see Definition 14) bucket s **do**
         Sample one element x_s
      for each active subproblem between buckets s (left), and q (right) **do**
         Test for inversion between x_s and x_q
      for each active subproblem X_i, X_{i+1} where η - mark (age) $> |X_i||X_{i+1}|$ **do**
         do all comparisons between X_i and X_{i+1}
         update the backbone and certificates accordingly
         the subproblem is finished, i.e. no longer active
      for each found inversion **do**
         update the backbone, including splitting buckets and resampling pivots
         mark new subproblems with round η as age

---

y are not ordered by transitivity of the backbone. Because x and y are of different color, they can be compared. If $y < x$, the pair is called an *inversion*. This allows one to extend the backbone: we get $u_i < y < x < u_{i+1}$, which is a chain of actual comparisons between elements of alternating color.

The only way to find an inversion in a bipartite setting (this is where the bichromatic setting is different) is to uniformly at random, from all pairs in X_i and X_{i+1}, pick x and y. If the fraction of inversions is p, then the probability of finding an inversion is p and the expected number of trials to find one is $1/p$.

InversionSort starts by (trivially) having the backbone consist only of the artificial smallest red element r_0 and largest blue element b_1, and $R_0 = R$ and $B_1 = B$. For a given backbone $(u_0, u_1, u_2, \ldots, u_{2k}, u_{2k+1}) = (r_0, b_1, r_2, \ldots, r_{2k}, b_{2k+1})$, InversionSort first, for each pair X_i, X_{i+1} of adjacent buckets that have not yet found an inversion or a proof that there is no further inversion (i.e., reached the adjacent-stripe verification bound), in a round-robin manner, does one round of inversion-searching by randomly comparing pairs of elements in adjacent buckets. If this leads to an inversion, the inverted pair is saved and the algorithm moves to the next pair of adjacent buckets. At the end of the round, all identified inversions are considered and used to extend the backbone. Then InversionSort splits existing buckets by pivoting with new elements on the backbone. Because there is at most one pair of inversions between each two neighboring representatives on the backbone, each element is compared to at most two new representatives in each round.

This reestablishes the backbone and creates some new pairs of neighboring buckets, for which InversionSort initializes the inversion finding procedures. The algorithm stops once all neighboring pairs of buckets are shown to not have an inversion, i.e., the comparisons in the verification bound between neighboring stripes have been performed.

Analysis. [16] visualizes a run of InversionSort as a ternary (refinement) tree, where nodes correspond to subproblems. For an internal node v, there is a corresponding subinterval on the backbone defined by two consecutive pivots, say a blue pivot followed by a red pivot, $b_v < r_v$. If InversionSort finds an inversion $y < x$ (x is blue and y is red) between b_v and r_v, then v has three children with the respective pivots (b_v, y), (y, x), (x, r_v).

The random nature of the inversion searching of InversionSort, is made precise in Lemma 11 in [16]. However, a stronger version of this lemma applies for bipartite setting with the same proof.

▶ **Lemma 15** (Randomness in Inversion Finding, Stronger Version of [16, Lemma 11, p11]). *At any stage of the InversionSort, consider a successful inversion finding procedure, which finds an inversion $y < x$ between representatives $u_i < y < x < u_{i+1}$. Say, w.l.o.g., that u_i is red and u_{i+1} is blue, and hence x is red and y is blue.*
1. *for any $y \in X_{i+1}$, conditioned on y being in the inversion, x is uniformly distributed among all the red elements in $R_y = \{x \in X_i \mid y < x < u_{i+1}\}$, an.*
2. *for any $x \in X_i$, conditioned on x being in the inversion, y is uniformly distributed in $B_x = \{y \in X_{i+1} \mid u_i < y < x\}$.*

This gives a bound on the height of the tree.

▶ **Lemma 16** (Height of the refinement tree [16, Theorem 5, p11]). *Let $\mathcal{T}$ be the refinement tree of running InversionSort on an instance $\mathcal{I}$ with $N = n + m$ elements. With high probability in N, the height of $\mathcal{T}$ is $O(\log N)$.*

Handling Overlaps. Because of the overlapping nature of the problem, InversionSort cannot easily focus on elements between neighboring representatives. For example, for the child indicated by pivots (y, x), instead of only getting the reds and blues that actually lie in this range as input, InversionSort instead has to also work with the red elements contained in (b_v, y) and the blue elements inside (x, r_v). This "spill-in" from the neighboring subintervals on the backbone needs to be analyzed.

As is argued in [16], the cost of bichromatic inversion search procedure of InversionSort is justified by the subinstance (part of the Hamiltonian) between the neighboring elements on the backbone. In Section 4.3.2, we will analyse this in the bipartite setting using the notion of instance optimality. However, if the spill-in for this subproblem is too large, inversion search is too costly. Hence, [16] identify subproblems that do not have too much spill-in from their neighbors, and call these subproblems *unaffected*. Inversion search in unaffected subproblems can be charged to their subinstance. More precisely, [16] show that at any time, with high probability, at least roughly a $1/(\log N)^2$ fraction of all current problems are unaffected. Accounting over the whole tree, including the pivoting, introduces another $\log N$ factor corresponding to the depth of the tree.

4.3.2 Putting everything together: Proof of Theorem 1

We now complete the proof of our main result for bipartite sorting, restated here for convenience.

▶ **Theorem 1** (Instance Optimality of InversionSort). *There exists a constant $c > 0$, such that for every instance $\mathcal{I}$, the cost of InversionSort on $\mathcal{I}$ is, with probability at least $1 - 1/N$, at most $c(\log N)^3 \operatorname{OPT}(\mathcal{I})$, where $\operatorname{OPT}(\mathcal{I})$ is as in Definition 8.*

We will show how to charge the comparisons performed by InversionSort to the lower bounds presented in Section 4.2. First, Lemma 15 and Lemma 16 imply that the refinement tree height $h = O(\log n)$ and hence the pivoting cost is $O(n \log n)$. Second, the considerations about unaffected subproblems remains valid, there are at most $O(\log^2 n)$ affected subproblems per unaffected subproblem. The inversion searching cost in each unaffected subproblem is justified by Lemma 11 if an inversion is found, otherwise by Lemma 10. Adding the extra $O(\log \mathcal{N})$ factor (height of the tree), the next lemma completes the proof of Theorem 1.

▶ **Lemma 17.** *Let $\mathcal{T}$ be the refinement tree of height h for a run of InversionSort on instance $\mathcal{I}$, and let $V_{\mathcal{T}}$ be the set of nodes of $\mathcal{T}$. Then*

$$\sum_{v \in V_{\mathcal{T}}} \mathrm{OPT}\,(\mathcal{I}_v) \le 2h\,\mathrm{OPT}(\mathcal{I})$$

Proof. It suffices to show that for each level $\mathcal{L}$ of $\mathcal{T}$ the inequality $\sum_{v \in \mathcal{L}} \mathrm{OPT}\,(\mathcal{I}_v) \le 2\,\mathrm{OPT}(I)$ holds. Note that the subinstances $\mathcal{I}_v$ of the same polarity in any $\mathcal{L}$ do not share vertices. Hence the decomposition Lemma 12 is applicable and the lemma follows because there are two polarities. ◀

5 Result on Sorting with Priced Information: Lower bound does not extend

Recall that there exists an instance demonstrating the lower bound of $\Omega(n)$ on the competitive ratio of any algorithm that finds the **maximum** of a set of n elements. Announced in the original paper by Charikar et al.[8], this was spelled out one year later by Gupta and Kumar [17]]. The instance is simple to describe (see Appendix B of the full version [15]), and all comparisons in it have costs in $\{0, 1, n, \infty\}$. Although this example was never formally stated for sorting, its discovery seems to have dampened efforts to study (either better algorithms, or lower bounds for) the general version of sorting with priced information in the past 20 years.

In this section, we prove Theorem 2, that shows that the $\Omega(n)$ lower bound for maximum with costs in $\{0, 1, n, \infty\}$ cannot extend to sorting.

▶ **Theorem 2.** *Consider the problem of sorting when every comparison has a cost in $\{0, 1, F, \infty\}$, for any $F \ge n^{3/4}$. There exists a polynomial time randomized algorithm whose competitive ratio is $\widetilde{O}(n^{3/4})$, with high probability.*

While counterintuitive at a first glance (after all, the cost to sort is at least the cost to find the maximum), the simple explanation is that the cheapest proof for sorting is also more expensive than that of the maximum. This opens up the problem of arbitrary-cost sorting once again - is there a $\Omega(n)$ lower bound for sorting with arbitrary costs, or can our $\tilde{O}(n^{3/4})$ algorithm be extended to an $o(n)$ competitive algorithm for arbitrary costs?

Theorem 2 is achieved by *first developing an algorithm for DAG sorting.* Why consider the case of an unsortable DAG, when the DAG we have is sortable? Here is the reasoning. If we consider greedy algorithms for sorting with priced information, it is natural to try to discover as much of $\vec{G}$ as possible with low-cost edges[12]. However, note that the sub-DAG $\vec{G}_{\le w}$ consisting of edges with cost at most w in $\vec{G}$ may not be sortable, which is exactly the problem of DAG sorting.

We set up some notation first. Given an undirected complete graph G with costs in $\{0, 1, F, \infty\}$, let $\vec{G}$ denote the underlying DAG that contains a directed Hamiltonian path. Define $\vec{G}_0$ as the DAG obtained by revealing all cost 0 edges, observe that it may not have

---

[12] We remark that some time after we first uploaded a version of Theorem 2 to Arxiv, a preprint by [Jiang, Wang, Zhang and Zhang, Arxiv [21]] was uploaded, where the authors also use an algorithm for DAG sorting (they call it GPSC) parameterized by w and extend our $\widetilde{O}(n^{3/4})$ algorithm to obtain a $\widetilde{O}(n^{1-1/2W})$ competitive algorithm for sorting with priced comparisons with at most W distinct costs. For our setting when $W = 4$ they re-derive our result separately as their main theorem would give a $n^{7/8}$-competitive ratio.

a Hamiltonian path, and let w_0 be the width of $\vec{G}_0$. Similarly, denote by $\vec{G}_{01}$ the DAG obtained by revealing all cost 0 and 1 edges; $\vec{G}_{01}$ may not have a Hamiltonian path either, and let w_{01} be the width of $\vec{G}_{01}$. Finally, let k_1 and k_F be the number of cost 1 and cost F edges on the Hamiltonian path in $\vec{G}$.

5.1 Algorithm details

The following is our proposed algorithm for the $0, 1, F, \infty$ cost version of sorting with priced information. Below, we will abbreviate the algorithm by Huang, Kannan and Khanna [20] for the $1, \infty$ setting, by HKK.

█ **Algorithm 2** Algorithm for $0, 1, F, \infty$ cost.

Require: undirected graph $G = (V, E, c)$ with costs $c(e) \in \{0, 1, F, \infty\}$
Ensure: the total order (directed Hamiltonian)
    Probe all cost zero edges
    Run the following 4 algorithms in parallel, performing one comparison from each. If any of them discover the Hamiltonian, report the edges in the Hamiltonian path, and abort the other algorithms
        ○ Set $F = \infty$. Run the HKK algorithm on the cost 1 edges, starting from $\vec{G}_0$
        ○ Set $F = 1$. Run the HKK algorithm on the cost 1 edges, starting from $\vec{G}_0$
        ○ Run Algorithm 3 on the cost 1 edges, starting from $\vec{G}_0$
        ○ Find the 0-1 DAG using Theorem 20, use Algorithm 3 with cost F edges on it.

The running time of the final algorithm will then be a minimum of four running times. We briefly explain the first two algorithms, and then explain in detail the last two. Recall that HKK algorithm runs when costs are 1 or ∞. On an input with costs in $\{0, 1, F, \infty\}$, the first algorithm pretends that cost F edges are forbidden too, i.e., $F = \infty$, and probes whatever edges HKK would have probed from the cost 1 edges. Clearly, this will not find the Hamiltonian if it contains cost F edges, as they aren't queried. However, in the case that the Hamiltonian does not contain cost F edges, it will sort the input, and stops. The second algorithm does the opposite: it does not differentiate between cost F and cost 1 edges, and probes them if they would have been probed by the HKK algorithm. If run for long enough, this will find the Hamiltonian, and is stopped once it does so.

5.1.1 Algorithm 3 : Hamiltonian By Predecessors

The third algorithm in Algorithm 2 is Algorithm 3, which is also used as the second half of the fourth algorithm in Algorithm 2. This algorithm finds a Hamiltonian path in a partially revealed DAG. It utilizes Lemma 18, that generalizes binary search to searching for predecessors of a vertex in a DAG of width w. For two DAGs D' and D on the same set of vertices, we will write $D' \subset D$ if all the directed edges in D' are also contained in D.

▶ **Lemma 18** (Hamiltonian by predecessor search). *Let $D' \subset D$ be two DAGs on the vertex set V and assume that D contains a Hamiltonian path. Assume that the Hamiltonian path in D contains a set $\vec{S}$ of k edges that are not in D', and let S be the undirected version of $\vec{S}$. Let E be a set of edges that can be queried and assume $S \subset E$. Let w be the width of D'. Then, $k + 1 \geq w$ and the Hamiltonian in D can be found with $O(wk \log n)$ queries on edges in E.*

Below is the pseudocode for Algorithm 3. It uses in turn a predecessor searching subroutine that is captured by the following simple lemma. Lemma 18.

▶ **Lemma 19** (Predecessor search in DAG). *Given a DAG $D' = (V, E')$ of width w, and a vertex $v \in V$, $|V| = n$, let D be the DAG obtained by extending D' by probing all edges involving v. Define $P_v = \{u \mid (u, v)$ is in the transitive reduction of $D\}$. There exists an algorithm that computes P_v with $O(w \log n)$ queries, and runs in $O(n^2)$ time.*

Proof. Observe that any chain in D' can contain at most one element of P_v, and hence $|P_v| \leq w$. Indeed, we can run one binary search on each of the w chains in D', leading to at most w candidate predecessors. The number of queries is easily seen to be $O(w \log n)$ after computing an optimal partitioning into chains in polynomial time.   ◀

Proof of Lemma 18.   By Dilworth's Theorem, D' can be partitioned into w chains. To show $k + 1 \geq w$, assume otherwise, $w > k + 1$, and let A be $k + 2$ non-comparable vertices in D'. By the pigeonhole principle, there must be two vertices of A in the same of the $k + 1$ stretches of cost 0 edges on the Hamiltonian, a contradiction to them being incomparable.

To prove that Algorithm 3 performs at most $O(kw \log n)$ queries, observe that adding edges to D' does not increase its width. In the while loop, as long as D' is not the Hamiltonian path, let S be the first layer of a BFS traversal of the transitive reduction of D' with $|S| \geq 2$, and observe that S is an antichain and hence $|S| \leq w$. All vertices of S but one have their incoming edge on the directed Hamiltonian not yet revealed / queried: Assume there are two vertices $b \neq d \in S$ and their predecessors $a \prec b$ and $c \prec d$ are both already in D. Then, b and d are not sources in D, and hence S must be the set of successors of a vertex v. Additionally, there is only a single source s in D, and the set $\{x \mid x < v\}$ forms a chain in the transitive reduction of D starting in s. This means $w \leq a < b$ and $w \leq c < d$ contradicting them being different.

By the above arguments, D contains the Hamiltonian with only $k - |S| + 1$ unrevealed edges missing. We used $O(|S|w \log n)$ queries to reduce the number of unrevealed edges by $|S| - 1$ for $|S| \geq 2$, hence each search creating a missing edge of the Hamiltonian, and this search must justify at most one additional such search. Hence, the total number of queries to arrive at the Hamiltonian is $O(wk \log n)$.

Binary searching for a vertex v into one of the w chains takes $O(\log n)$ probes, and in $O(w \log n)$ probes one is sure to have at least discovered one edge from the Hamiltonian, namely the incoming edge to v. This can then be repeated k times, revealing the Hamiltonian.

■ **Algorithm 3** Hamiltonian By Predecessors.

Require: undirected $G = (V, E)$ defining which comparisons are allowed
Require: DAG D of already probed edges (initially the cost 0 edges)
Ensure: The updated D contains a directed Hamiltonian
　create (and maintain) a chain decomposition C of the transitive reduction of D
　while D has width $w > 1$, i.e. is not the intended result **do**
　　if The transitive reduction of D has several sources **then**
　　　Let S be the set of these sources
　　else
　　　Let v be the lowest vertex with more than 1 successors
　　　Let S be the set of successors of v

　　for each u in S **do**
　　　Find all predecessors of u in D (Lemma 19), adding answers to D
　　　　　　　　　　　　　　　　　　\\ there are at most w such predecessors

5.1.2 The fourth algorithm in Algorithm 2

We will develop another algorithm, that proceeds in two steps: a) compute only the 0-1 DAG, $\vec{G}_{01}$, and b) find the cost F edges (k_F-many of them) on the Hamiltonian path. Step b) is performed using Algorithm 3. If $k_F = 0$, step a) recovers the complete Hamiltonian path. Before we state the DAG sorting algorithm for step a), we note that it only needs to output the transitive reduction of $\vec{G}_{01}$.

▶ **Theorem 20.** *There is a poly-time randomized algorithm that w.h.p. solves DAG sorting for an instance $\vec{G}$ with edge costs in $\{0, 1, \infty\}$, using $O(\min(wn^{3/2} \log n, n^2))$ comparisons, where w is the width of $\vec{G}$.*

We defer the proof of Theorem 20 for now and analyze our algorithm assuming it.

5.2 Analysis of Algorithm 2

▶ **Lemma 21.** *Algorithm 2 incurs the following costs*

$$\begin{cases} O(\min(n^{1.5} \log n \,, w_0 k_1 \log n)) & \text{if } k_F = 0 \\ O(\min(Fn^{1.5} \log n \,, w_{01} n^{1.5} \log n + Fw_{01} k_F \log n)) & \text{if } k_F > 0 \end{cases}$$

Proof. If $k_F = 0$, the first algorithm that ignores cost F edges by setting $F = \infty$ never probes a cost F edge, and finishes in $O(n^{1.5} \log n)$ comparisons (this is the cost of the algorithm by Huang, Kannan and Khanna [20]). Since the DAG formed by cost 0 edges has width w_0 and $k_F = 0$, $w_0 \leq k_1 + 1$. The third step running Algorithm 3 finishes after at most $O(w_0 k_1 \log n)$ comparisons, by Lemma 18.

If $k_F > 0$, the first term comes from running HKK after setting $F = 1$: the true cost of probing an edge is at most a factor F larger. Finally, step 4 of Algorithm 2 runs the algorithm in Theorem 20 first, incurring at most $w_{01} n^{1.5} \log n$ many comparisons. With the 0-1 DAG obtained using Theorem 20, Algorithm 3 now inserts at most k_F many edges in the Hamiltonian, probing at most $w_{01} \log n$ many edges for each. Every probe costs F, for a total of $w_{01} n^{1.5} \log n + Fw_{01} k_F \log n$. ◀

5.2.1 Proof of $\tilde{O}(n^{3/4})$ competitive ratio of Algorithm 2

We claim that the competitive ratio is always bounded by $O(n^{3/4} \log n)$. Observe that the cost of the Hamiltonian is $k_1 + Fk_F$. If $k_1 = k_F = 0$, the Hamiltonian has a cost of 0 and our algorithm finds it at cost 0. From now on, we assume not both of k_1 and k_F are 0.

Consider the case $k_F = 0$ first. Note that this implies that the width w_{01} of $\vec{G}_{01}$ is 1. First consider the subcase when $w_0 \leq n^{3/4}$. In this case, the competitive ratio is bounded above by $O(w_0 k_1 \log n)/k_1 = O(w_0 \log n) \leq O(n^{3/4} \log n)$. In the subcase when $w_0 > n^{3/4}$, observe that this implies that $k_1 \geq n^{3/4}$ which implies that the competitive ratio is bounded above by $O(n^{1.5} \log n)/k_1 \leq O(n^{3/4} \log n)$.

Next, consider the case $k_F \geq 1$, and the cost of the Hamiltonian is at least Fk_F. Since $w_{01} \leq k_F + 1$, the cost of the algorithm is at most $O(w_{01} n^{1.5} \log n + Fw_{01} k_F \log n) < O(k_F n^{1.5} \log n + Fw_{01} k_F \log n)$, and dividing by FK_F (the lower bound on the cost of the Hamiltonian), we get a competitive ratio of at most $O((n^{1.5}/F + w_{01}) \log n)$. Since $F \geq n^{3/4}$, this ratio is $O(n^{3/4} \log n)$ as long as $w_{01} \leq n^{3/4}$. Else if $w_{01} > n^{3/4}$, we observe that $k_F \geq n^{3/4}$, and then the $Fn^{1.5} \log n$ query cost gives us a competitive ratio of at most $Fn^{1.5} \log n/Fk_F \leq n^{3/4} \log n$. Thus Theorem 2 is proved. ◀

It remains to prove Theorem 20, which is the topic of the next subsection.

5.3 Proof of Theorem 20

First, observe that if the width of $\vec{G}$ is at least $\sqrt{n}/4$, the statement of Theorem 20 is easy to achieve by probing all edges. Hence, let us assume the width is at most $\sqrt{n}/4$. We will show that there is an algorithm that computes $\vec{G}_{01}$ with cost at most $O(w_{01}n^{1.5}\log n)$. This algorithm will only probe cost 0 and 1 edges, and will be a generalization of the algorithm in [20]. Note that while the algorithm in [20] works on a cost $\{1, \infty\}$ setting under the promise of a Hamiltonian path in the true graph, our algorithm finds the transitive reduction of the DAG $\vec{G}_{01}$ of width w_{01}. We first address the challenges posed in extending the work by Huang Kannan and Khanna [20].

Challenges in extending the results of Huang, Kannan and Khanna [20]. At a high level, the algorithm in [20] alternates between three ways of making progress:

1. Finding and probing balanced edges, defined as those that reduce the number of possible linear extensions of the current DAG by a $1 - (1/(e\sqrt{n})))$ factor. Finding such edges requires approximating the average rank of vertices under all possible linear extensions at all stages of the algorithm.

2. After estimating the indegree of vertices upto an additive error of $\widetilde{O}(\sqrt{n})$ by an $\widetilde{O}(n^{1.5})$ sampling procedure, the algorithm probes free edges, defined as the set of edges (u, v) where the average rank of u is smaller than the average rank of v, and v has most $\widetilde{O}(\sqrt{n})$ unprobed incoming edges. Free edges that are balanced again reduce the number of linear extensions by a constant factor. Otherwise, they can contribute at most $\widetilde{O}(n^{1.5})$ to the total cost.

3. Binary Search - When there are no free edges, there must exist a set of $\sqrt{n}$ vertices with known total order (Lemma 3.5 in [20]). The other vertices can perform binary search into these $\sqrt{n}$ vertices at a cost of $O(n\log n)$, and doing so removes these $\sqrt{n}$ vertices from the picture. The total cost of binary search is therefore $\widetilde{O}(n^{1.5})$.

The third step of the algorithm is the step that guarantees a reduction in the problem size. However, the third step of this algorithm no longer works for DAG sorting: the existence of a set of $\sqrt{n}$ vertices with known total order crucially relies upon the existence of the Hamiltonian path.

Proof of Theorem 20. All of the definitions, algorithms, and accounting to estimate the in-degree of a vertex in [20] remain valid and unchanged. Observe that any topological sorting of the underlying directed graph, together with the undirected graph, reveal the directed graph. Define the **average rank of a vertex** as the average rank over all linear extensions of the true underlying directed graph. The following result implies that the average rank r on a path in (the transitive reduction of) a DAG is increasing by at least one per edge.

▶ **Lemma 22.** *Let $D = (V, E)$ be a DAG and $r \colon V \to \mathbb{Q}^{\geq 0}$ be the average rank. Then for $(u, v) \in E$ we have $r(u) + 1 \leq r(v)$.*

Proof. Let Π be the set of all linear extensions that are compatible with D, such that $r(x)|\Pi| = \sum_{\pi \in \Pi} \pi(x)$. Then $|\Pi|(r(v) - r(u)) = \sum_{\pi \in \Pi} \pi(v) - \pi(u) \geq |\Pi| \cdot 1$. ◀

▶ **Definition 23** (Convex vertex subset). *In a DAG $G = (V, E)$, a subset of vertices $S \subseteq V$ is* convex *if for every pair of vertices $u, v \in S$, every vertex w on any directed path from u to v in G is also in S.*[13]

Hence, considering a subset of the vertices by an upper and a lower bound on the average rank, leads to a convex subset. Next, a vertex is **live** if there is an unprobed edge incident to it, otherwise it is **exhausted**. The **assumed graph** is the same directed graph as in [20]. An **active vertex** is one that has at most $4\sqrt{n} \log n$ unprobed in-edges in the assumed graph. A **free edge** is an unprobed edge (u, v) where the endpoint v is active. The proof of the next lemma is identical to that in [20].

▶ **Lemma 24** (Generalization of Lemma 3.5 in [20]). *The $\sqrt{n}$ live vertices with smallest average rank are all active.*

▶ **Lemma 25** (Generalization of Lemma 3.6 in [20]). *If there are no free edges, and the width of the underlying $\vec{G}$ is at most $\sqrt{n}/4$, then there exists a set S of at least $(3/4)\sqrt{n}$ live vertices with known partial order who form a DAG of width at most $\sqrt{n}/4$.*

Proof. Consider the set S of at most $\sqrt{n}$ live vertices with smallest average rank. More precisely, we chose the largest upper bound on the average rank such that $|S| \leq \sqrt{n}$. By Lemma 22, at most $\sqrt{n}/4$ vertices can have the same average rank, such that $|S| \geq 3/4\sqrt{n}$. By Lemma 24, all vertices of S are active. Let $u, v \in S$ be a pair of vertices that have a directed path P from u to v in $\vec{G}$. Then, all of this path P is in S, and all live vertices of P are in S. Hence, because there are no free edges, and all vertices of P not in S are exhausted, all edges of P must be probed. Hence, S is convex. The statement on the width follows from a chain decomposition of $\vec{G}$ remaining a chain decomposition for a convex subset of vertices. ◀

Note that Lemma 25 does not imply that the algorithm can, or should, identify precisely this set S defined in the proof. Hence, the algorithm is going to approximate the smallest width subset of at least $3/4\sqrt{n}$ vertices among the live vertices. More precisely, starting from an empty S, it is going to iteratively find the longest (outside S) chain among the live vertices (also using edges that are implied by transitivity via edges in S). It stops once S contains at least $3/4\sqrt{n}$ vertices, and uses it as the DAG of small width in the setting of Lemma 19, and determine for every remaining live vertex its predecessors in S, with $O(w \log n)$ queries each, compared to the $O(\log n)$ queries if there is a Hamiltonian. Hence, the total number of queries increases from $O(n^{3/2} \log n)$ to $O(w n^{3/2} \log n)$, as claimed in Theorem 20, completing the proof.

References

1   Peyman Afshani, Jérémy Barbay, and Timothy M Chan. Instance-optimal geometric algorithms. *Journal of the ACM (JACM)*, 64(1):1–38, 2017.

2   Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

---

[13] Our definition of convexity differs from the definition in the metric graph theory (defined on undirected graphs), where convex subgraph contains the vertices of only the shortest paths between every pair of vertices [4].

3   Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor, and Rafail Ostrovsky. Matching nuts and bolts. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 690–696, 1994.

4   Hans-Jürgen Bandelt and Victor Chepoi. Metric graph theory and geometry: a survey. In Surveys on Discrete and Computational Geometry: Twenty Years Later, editors, *Jacob E. Goodman and János Pach and Richard Pollack*, volume 453 of *Contemporary Mathematics*, pages 49–86. AMS, 2008.

5   Michael A Bender, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Kostas Tsichlas. Batched predecessor and sorting with size-priced information in external memory. In *Latin American Symposium on Theoretical Informatics*, pages 155–167. Springer, 2021.

6   Jean Cardinal, Justin Dallant, and John Iacono. An instance-optimal algorithm for bichromatic rectangular visibility. In *29th Annual European Symposium on Algorithms (ESA 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

7   Jean Cardinal and Samuel Fiorini. On generalized comparison-based sorting problems. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 164–175. Springer, 2013.

8   Moses Charikar, Ronald Fagin, Venkatesan Guruswami, Jon Kleinberg, Prabhakar Raghavan, and Amit Sahai. Query strategies for priced information. *Journal of Computer and System Sciences*, 64(4):785–819, 2002.

9   Constantinos Daskalakis, Richard M Karp, Elchanan Mossel, Samantha J Riesenfeld, and Elad Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3):597–622, 2011.

10   Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752, 2000.

11   Amol Deshpande, Lisa Hellerstein, and Devorah Kletenik. Approximation algorithms for stochastic boolean function evaluation and stochastic submodular set cover. In *Proceedings of the twenty-fifth annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*, pages 1453–1466. SIAM, 2014.

12   Devdatt P Dubhashi, Kurt Mehlhorn, Desh Ranjan, and Christian Thiel. Searching, sorting and randomised algorithms for central elements and ideal counting in posets. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 436–443. Springer, 1993.

13   Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992. `doi:10.1145/146370.146381`.

14   Ulrich Faigle and Gy Turán. Sorting and recognition problems for ordered sets. *SIAM Journal on Computing*, 17(1):100–113, 1988.

15   Mayank Goswami and Riko Jacob. On instance-optimal algorithms for a generalization of nuts and bolts and generalized sorting, 2023. `arXiv:2211.04601`.

16   Mayank Goswami and Riko Jacob. An algorithm for bichromatic sorting with polylog competitive ratio. In *15th Innovations in Theoretical Computer Science Conference (ITCS 2024)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2024.

17   Anupam Gupta and Amit Kumar. Sorting and selection with structured costs. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 416–425. IEEE, 2001.

18   Bernhard Haeupler, Richard Hladík, Václav Rozhoň, Robert Tarjan, and Jakub Tětek. Universal optimality of dijkstra via beyond-worst-case heaps, 2023. `arXiv:2311.11793`.

19   J Hartline, E Hong, A Mohr, E Rocke, and K Yasuhara. Personal communication. In , 2000.

20   Zhiyi Huang, Sampath Kannan, and Sanjeev Khanna. Algorithms for the generalized sorting problem. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS'11)*, pages 738–747, 2011. `doi:10.1109/FOCS.2011.54`.

21   Shaofeng H. C. Jiang, Wenqian Wang, Yubo Zhang, and Yuhao Zhang. Algorithms for the generalized poset sorting problem, 2023. `arXiv:2304.01623`.

22   Jeff Kahn and Nathan Linial. Balancing extensions via brunn-minkowski. *Combinatorica*, 11(4):363–368, 1991.

23   Haim Kaplan, Eyal Kushilevitz, and Yishay Mansour. Learning with attribute costs. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC'05)*, pages 356–365, 2005.

24   János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in $O(n \log n)$ time. *SIAM Journal on Discrete Mathematics*, 11(3):347–372, 1998.

25   Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *In Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'08)*, volume 8, pages 1096–1105, 2008.

26   Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 379–388. IEEE, 2006.

27   Gregory JE Rawlins. *Compared to what? An introduction to the analysis of algorithms*. Computer Science Press, Inc., 1992.

28   Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 222–227, 1977. `doi:10.1109/SFCS.1977.24`.