Order manuscript No.
(will be inserted by the editor)

A Fast, Simple Algorithm for Computing the Bump Number of a Poset

Gara Pruesse · Derek Corneil · Lalla Mouatadid

November 1, 2018

Abstract We present a new algorithm for the *Bump-Number* problem: given a partial order of elements or *jobs*, find a linear ordering of the jobs that extends the partial order and minimizes the number of jobs scheduled immediately after one of its precedents in the partial order.

Polynomial time algorithms that solve this problem are known. The fastest achieve linear time in the size of the transitive reduction of the poset, but are highly complex in the programming sense; the proofs of correctness are long and involved; and to achieve the linear time bound, they rely on a somewhat elaborate implementation of Union-Find algorithms for the special case that the union and find operations are known in advance. We present a simple algorithm solving the problem efficiently ($O(m + n \log w)$, where w is the width of the poset), using simple data structures, and with short proofs of correctness; on the cocomparability graph, the algorithm runs in linear time. Our algorithms and proofs offer insight into the effectiveness of Lexicographic Depth-First Search as a preprocessing step for certain algorithms on cocomparability graphs.

Keywords p
oset, bump number, scheduling, cocomparability, hamiltonicity, hamilton path, minimum path cover

1 Introduction

In this paper we consider a partial order on a set of elements or vertices as modelling a set of ordering constraints on the elements. The problem we study here is the *bump number* problem, which can be posed as a scheduling problem. Consider the elements of the partially ordered set as jobs to be scheduled on a single processor. No job may be executed until its precedents have completed; and furthermore there is a unit cost associated with scheduling a job u immediately after a job v which constrains it, i.e., if $v < u$ in the partial order, there is a cost associated with scheduling u immediately after v. This models a situation, for example, in which the precedence constraints reflect that outputs from one job will be utilized as inputs to other jobs, and the cost function reflects that there is a unit-cost communication delay when successive jobs are related by the communication of outputs. The bump number problem seeks the find a total order consistent with the partial order that minimizes the costs of precedence-related jobs being scheduled successively.

Consider the Tricky Poset, the Hasse diagram of which is given in Figure **??**. The linear extension $abcdegfh$ has no bumps. However, any linear extension starting with $abdc$ cannot be completed without introducing a bump. Elements c and d are at the same depth and height, and each has the same number of upper covers, ancestors, lower covers, and descendents; and yet the selection of c or d after ab determines whether a minimum bump linear extension can be found.

Cocomparability graphs are graphs that correspond to a poset in the following manner: two vertices are adjacent in the graph precisely if they are incomparable in the partially ordered set. This class of graphs

G. Pruesse
Vancouver Island University, Department of Computing Science, Nanaimo, BC, Canada
E-mail: Gara.Pruesse@viu.ca, WWW: `http://www.csci.viu.ca/~gpruesse`

D. Corneil · L. Mouatadid
University of Toronto,
Department of Computer Science,
Toronto, ON, Canada

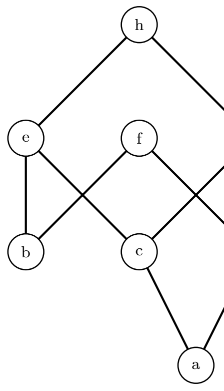

Fig. 1: The Tricky Poset. $abdcfegh$ is a 1-bump greedy linear extension, but there is a 0-bump linear extension.

contains the interval graphs, and are a well-studied subclass of the asteroidal-triple-free graphs. A minimum-bump ordering of the poset is equivalent to a minimum path cover of the associated cocomparability graph. Every cocomparability graph that is Hamilton traceable has a zero-bump ordering in its associated posets, and vice versa. Hence algorithms for minimum-bump orderings of posets can be used to solve minimum path cover and Hamiltonian path problems in cocomparability graphs.

We present here an elegant, efficient algorithm to find minimum-bump orderings. Our algorithm is fast in practice, and asymptotically achieves $\mathrm{O}(m + n \log w)$ running time where n is the number of elements, m is the number of covers relations (edges in the transitive reduction), and w is the width of the poset (the size of the maximum antichain); alternatively, the algorithm can be made to run in time $\mathrm{O}(n + m')$, where m' is the number of less-than relations (i.e., the number of edges in the transitive closure). Therefore the running time is asymptotically as fast as the fastest known algorithm for minimum path cover on cocomparability graphs [5], for which this algorithm also provides a solution, and is fast in practice but not linear time on the more compact representation, the transitive reduction. Furthermore, the algorithm is much simpler and is easier to implement than previously known algorithms, and has short proofs of correctness and complexity analysis. In contrast, the algorithms that achieve linear time on the transitive reduction are extremely complex to implement, and it is unkown by the authors whether any attempt has been made to do so.

These results will be of interest to researchers into algorithms for cocomparability graphs and subclasses such as interval graphs, as these results provide a framework for understanding the efficacy of Lexicographic Depth-First Search as a preprocessing step for Hamiltonicity, minimum path cover, and independent set algorithms for these graph classes. Furthermore, a straightforward adaptation of the Greedlex algorithm, applied to cocomparability graphs, yields a linear time algorithm algorithm to solve the Hamilton path and minimum path cover problems on cocomparability graphs.

2 Previous work

The bump-number problem is related to two-processor scheduling, in the following manner. Recall that a minimum bump linear extension $v_1 v_2 \cdots v_n$ that minimizes the number of v_i where $v_i < v_{i+i}$ – i.e., minimizes the number of bumps. is one that has the fewest On the other hand, a two processor schedule for precedence-constrained jobs, where the precedence constraints are captured by a poset on the jobs, is a linear extension $v_1 v_2 \cdots v_n$ that can be divided into a sequence of pairs and singletons (e.g., $(v_1, v_2), (v_3)(v_4, v_5) \cdots (v_{n-1}, v_n)$) so that no bump appears within a pair.

The two processor scheduling problem arose out of scheduling theory, and was investigated by Fujii, Kasarni and Ninarniya, [1] who in 1969 published a $\mathrm{O}(n^4)$ algorithm for two processor schedule based on first finding the transitive closure of the poset and then finding a matching. Following on this work, researchers sought solutions that avoid the expensive step of transitively closing the partial order. Two different approaches yielded results.

Coffman and Graham [2] employ lexicographic labelling as a kind of secondary sort to extend the partial order already imposed; the resulting algorithm runs in $\mathrm{O}(n^2)$. Later, Sethi [4] published a refinement of the Coffman-Graham algorithm; it includes a compact and efficient algorithm for lexicographic labelling; the remainder of the scheduling algorithm – given the labelling, find an optimal schedule – runs in $\mathrm{O}(n\alpha(n) + m)$ time, as it uses Union-Find data structures.

Gabow's algorithm for two-processor scheduling [6] processes the poset in *levels*, each level corresponding to the elements at a given height. Elements in a given height form an antichain in the poset; Gabow's two processor scheduling algorithm pairs elements at the same height (level) when possible, and then employs strategies to deal with levels that are populated by an odd number of elements. The algorithm was proved to run in [running time here]. Later, Gabow and Tarjan devised an algorithm for Union-Find that runs in linear time when the operations are known in advance; utilizing this yields a linear time two processor scheduling algorithm [Gabow and Tarjan?].

Polynomial time algorithms were found for to find the bump number of a poset (Habib, Mohring and Steiner). Then Schaffer and Simons (1986) used the Gabow two-processor scheduling algorithm as a basis for a bump number algorithm, and thus with the Gabow and Tarjan speed-up were able to show that the bump number problem could be solved in linear time.

We know of no actual implementation of the linear time algorithm for bump number (or two processor scheduling). The main result of this paper is easily-stated theorems for the bump-number that lead to algorithms for both problems that are trivial to implement and fast in practice. The theorems also lend insight into the algorithms for Hamiltonicity and Minimum Path Cover problems for comparability graphs.

3 Definitions and Terminology

A multiset S_1 is said to be *lexicographically larger* than a multiset S_2 if it is a proper superset of S_2 or if $\max(S_1 \setminus S_2) > \max(S_2 \setminus S_1)$. We will write $S_1 >_{lexico} S_2$ to denote this relation. We use this notation because we are reserving "$<$" for the poset relation; but the lexicographic comparator operation for sets is a standard comparator relation for sets. For example, $\{7, 5, 4, 4, 2, 2, 1\} >_{lexico} \{7, 5, 4, 3, 2, 1, 1\}$, since the largest number in which they differ is 4, and that occurs in the former set.

A *poset* $P = (E, R)$ is a set of elements E and a reflexive, antisymmetric, transitive relation R. If $(u, v) \in R, u \neq v$, we say u is less than v in P and we denote this with $u <_P v$, or simply $u < v$ if the poset is understood; and we use $u \leq v$ similarly. If neither (u, v) nor (v, u) are in R, we say u is incomparable to v, and denote this with $u||v$. We use maxima(P) and minima(P) to denote the maximal and minimal elements of P, respectively. The *transitive reduction* of a poset P is the set $T \subseteq R$ such that $(u, v) \in T$ and $(v, w) \in T$ implies $(u, w) \notin T$; in other words, T is the smallest set whose transitive closure is R. We call the relations in the transitive reduction the *covers* relations of P; if $(u, v) \in R'$ then we say v *covers* u and denote it $u \prec_P v$ and $v \succ_P u$; we use $\prec$ and $\succ$ when the po set is understood. The *Hasse diagram* of a poset is the diagram of the transitive reduction graph, with the minima at the bottom and maxima at the top; if $u \prec v$ in P then we draw u below v and connect them with an edge.

A poset $P' = (E', R')$ is an *extension* of $P = (E, R)$ if $E' = E$ and $R' \supseteq R$. A linear extension is an extension that is also a total order. A poset $P' = (E', R')$ is an *induced subposet* of $P = (E, R)$ if $E' \subseteq E$ and $R' = R \cap E' \times E'$. We use $P \setminus S$ to denote the poset P induced on $E \setminus S$. A *filter* of a poset is an induced subposet that is closed under the covers relations (i.e., if x is in the filter, so is everything x covers). If P is a poset and L a linear extension of P, then note that $L \setminus S$ is a linear extension of $P \setminus S$. It is convenient sometimes to use string operations such as concatenation $(\cdot)$ in manipulating linear extensions.

A *chain* in a poset is a sequence $v_1 v_2 \cdots v_k$ of elements such that $v_i < v_{i+1}, \forall i, 1 \leq i < k$. The *depth* of an element v in a poset, denoted depth(v), is the length of the longest chain that has v as its minimal element.

In the context of linear extensions and job scheduling, it is helpful to think of the relations R as *precedence constraints*: if $a \prec b$ in P then a must appear before b in any linear extension of P, and the job a must be completed before job b is begun.

The *bump number* of a linear extension $L = x_1 x_2 \cdots x_n$ of P is the number of elements x_i that are a cover of their direct predecessor x_{i-1} in L. The bump number of P is the minimum bump number of any linear extension of P. In constructing linear extensions or job scheduling, the notion of *shelling* the poset can be useful – it consists in successively removing minimal elements from the poset. A shelling order for all the elements of the poset is necessarily a linear extension, and vice versa.

A considerable amount of recent work has focused on applying Lexicographic Depth-First-Search to cocomparability graphs to achieve fast algorithms to solve hamiltonicity problems and path cover problems on that class of graphs. To illuminate the link between that work and the present work, we provide here some definitions from the area of cocomparability graphs, and make the translation into the language of posets, in which the present work is framed. Readers primarily interested in the bump number problem on posets can jump to Section XXX without loss of continuity.

3.1 Relationships with Co-comparability Graphs

An undirected graph $G = (V, E)$ is a *comparability graph* if there is a poset $P = (V, <_P)$ such that $(u, v) \in E$ if and only if $u <_P v$ or $v <_P u$. A graph is a cocomparability graph if and only if its complement is a comparability graph. The mapping from the posets to their associated cocomparability graphs is many-to-one; the mapping establishes partition classes of posets, where a poset and its dual are in the same partition (possibly with other posets that have the same comparability graph, and therefore the same cocomparability graph). We will use the term “posets associated with G” to refer to the posets whose cocomparability graph is G.

Cocomparability graphs are a graph class that contains the interval graphs, and are contained in the asteroidal-triple-free graphs.

A total ordering π of the vertices of a cocomparability graph $G = (V, E)$ is a *cocomparability ordering* or *cocomp* ordering iff whenever $u <_\pi v <_\pi w$ and $(u, w) \in E(G)$ then either $(u, v) \in E$ or $(v, w) \in E$. An alternative characterization of cocomparability graphs is that a graph is cocomparability if and only if it has a cocomp ordering. All such orderings are linear extensions of posets associated with the cocomparability graph G, and indeed the associated posets partition the set of cocomp orderings of a cocomparability graph, by way of the relation “is a linear extension of”.

A *Lexicographic Depth-First Search* ordering, or Lex-DFS ordering, of a graph is an ordering of the vertices of the graph that corresponds to an order in which the vertices can be visited by depth-first search, labelling vertices in the order they are visited, starting at an arbitrary vertex, and proceeding by successively selecting the next vertex to visit by determining which one has the set of labels of visited neighbours that is lexicographically largest. For example, if v_1, v_2, v_3 is the order in which the first three vertices have been visited, v_3 being the most recently visited, and x has neighbours v_1 and v_3 whereas y has neighbours v_2 and v_3, then y will be selected next if the choice is between x and y. For cocomparability graphs, the Lex-DFS orderings that are also cocomp orderings have been of particular usefulness in algorithms for hamilton path, minimum path cover, and independent set. [**?**,**?**,**?**]

A cocomparability graph can be compactly represented by one of its associated posets. For sparse posets – the antichain on n elements, for example – the size of the Hasse diagram is $O(n)$ whereas the cocomparability graph is $O(n^2)$. The relationship is not reciprocal: posets with dense Hasse diagrams also have dense cocomparability graphs. Therefore one of the goals of pursuing cocomparability graph algorithms via their representation as posets is to ascertain whether there are efficient algorithms in the size of the *compact representation*. In essence, we ask: does the efficiency of, say, linear time algorithms for hamilton path in cocomparabilty graphs rely on the generous size of the representation scheme for the input? We seek algorithms on the poset representation that utilize the methods and insights of recent work on cocomparability graphs, and that have similarly efficient running times even when measured as a function of the more compact representation.

It is worth noting here that the running time of our algorithm is linear in the size of the the cocomparability graph, and is therefore as efficient as the known minimum path cover algorithms for cocomparability graphs [Cocomp MinPathCover]. Cocomparability graphs admit a representation that is (usually) smaller than the graph itself: the transitive reduction of (any of) its underlying posets. Running time analysis for algorithms solving the Min-Bump-Ordering/Min-Path-Cover problem may be linear in the size of the transitive closure, as the most recent results on cocomparability graphs, but the transitive closure itself can be quadratic in the size of more compact representation as a Hasse diagram, the directed graph representation of the transitive reduction. The greedlex algorithm for bump number is is $\mathrm{O}(n \log(w) + m)$, where w is the width of the Hasse diagram, n the number of elements, m the number of edges in the transitive reduction. This is a tighter bound than linear in the size of the transitive closure, regardless of poset width.

4 Lex Labellings of Posets

Definition 1 For a poset $P = (E, R)$, a *lexicographic labelling lex*: $E(P) \to \mathbb{Z}$ is any labelling of the elements such that for any elements u and v in E, if the multiset of lex numbers of the upper covers of u is lexicographically greater then the multiset of lex numbers of the uppers covers of v then $lex(u) > lex(v)$; that is:

$$\{\mathrm{lex}(v') : v' \succ v\} >_{lexico} \{\mathrm{lex}(u') : u' \succ u\} \Rightarrow \mathrm{lex}(v) > \mathrm{lex}(u)$$

Note also that a lexicographic labelling of a poset induces a lexicographic labelling of any filter or ideal of the poset.

There are many lexicographic labellings of a poset. Two important types of lexicographic labellings are the following: the *parsimonious labelling*, which gives all maxima the label 1 and then ascribes the minimum possible label value to each element while adhering to the lexicographic rule; and the *sequential labellings*, which label the t maxima with the numbers $1, 2, \ldots, t$, and ascribes labels lexicographically to the remaining elements without label duplication.

The sequential labellings impose an arbitrary order on the maximal elements; the remainder of the labelling is then essentially fixed by that order, except that ties must be broken when distinct elements have the same upper-cover sets. Note that any sequential lex-labelling of a poset yields a total order of the elements – indeed, it is the reverse of a linear extension of the poset. Since we will be jumping between representations of the transitive closure and the transitive reduction, we need to confirm that lex labelling processes operate identically on the two representations. In other words, lex labelling using the *set of lex labels of upper covers* of each vertex v yeilds exactly the same labellings as using *the set of lex labels of all* $w > v$. The following lemma makes that clear.

In the lemma and its proof, we call the set of all $w > u$ the *up-set* of u. Call a labeling of the elements of a poset a *lex-plus labelling* if it arises from lexicographically labelling the vertices using up-sets rather than cover sets to determine the ordering of the labels. I.e., a function lex+: $E(P) \to \mathbb{Z}$ is a lex-plus labelling if: for any elements u and v in E, if the multiset of lex+ numbers of the up-set of u is lexicographically greater than the multiset of lex+ numbers of the up-set of v then lex+(u) >lex+(v).

Lemma 1 *A labeling is a lex-plus labelling if and only if it is a lexicographic labeling.*

Proof: This is easily proved by induction on the number of elements in the poset, once the following observation is made. Given two elements x and y whose ancestors are already labelled, and where both the lex and lex-plus labels are the same for those ancestors, then the largest lex-plus ancestor label at which they differ will be the lex-plus label of a cover of x that is incomparable to y, without loss of generality. This is also a largest lex label of a cover of x that is incomparable to y. If, on the other hand, they have the same lex-label set of upper covers, they will also have the same lex-plus-label set of ancestors. In other words, x can be lex-labeled before y if and only if it can be lex-plus-labelled before y, and the identical lex and lex-plus labelling can be extended.

We are now ready to prove the following lemma.

Lemma 2 *Let $\pi : [n] \to V$ be a Lex-DFS cocomp ordering of cocomparability graph G. Then there is a poset associated with G for which π^{-1} is a sequential lex-labelling.*

Proof: The claim, restated in its specifics, is that if $x_1 x_2 \cdots x_n$ is a Lex-DFS cocomp ordering of G, then the labelling $\text{lex}(x_i) = i$ is a lexicographic labelling of a poset associated with G – in particular, the poset where if $(x_i, x_j) \notin E(G)$ and $i < j$ then $x_i >_P x_j$, so that $x_n x_{n-1} \cdots x_1$ is a linear extension of P.

Observe that, given a cocomp ordering $x_1 x_2 \cdots x_n$ of any type (i.e., Lex-DFS or not), we can construct an associated poset for the graph by directing all the edges (x_i, x_j) in the complement of G whenever $i < j$ – i.e., direct the edge towards the rightmost of the two vertices in the cocomp ordering; the resulting directed graph is the transitive closure of a Hasse diagram of an associated poset (where all the edges are directed downwards).

Clearly every cocomp ordering is the reverse of a linear extension of the poset that thus arises. Furthermore, if the cocomp ordering is Lex-DFS, then the process used by Lex-DFS to select the i^{th} vertex x_i will choose a vertex/element with a lexicographically maximum set of labels among its neighbours in the cocomparability graph that are thus far labelled; this same vertex will have a lexicographically *minimum* set of *comparability* neighbours among the vertices so far labelled, i.e., comparable to it and above it in the poset. Since $x_n x_{n-1} \cdots x_1$ is a linear extension of P, all of x_i's upper covers will have been labelled before x_i is labelled, and no element y where $y < x$ will be labelled before x. Therefore the process of Lex-DFS is identical to the sequential lex-labelling of the poset defined above, except that the Lex-DFS is operating on neighbourhoods in the transitive closure, not the transitive reduction of the poset, as the lex-labelling does. However, By Lemma 1, the use of the transitive closure edges gives a labelling that can also be achieved through using only the transitive reduction.

It would be very interesting to find a linear time method for lexicographic labelling that work when the input is a directed acyclic graph (DAG) that is neither the transitive closure nor the transitive reduction of the DAG, but something in between, and that does not first convert to either the reduction or closure. However, the example of Figure REFERENCE gives a DAG (all edges are directed downwards) that contains some but not all transitive edges; Sethi's algorithm gives element 4 a smaller label than element 5, which contradicts the lex-labelling definition.

4.1 Theorems about Lexicographic Labellings

It is easily seen that if $\mathrm{depth}(u) < \mathrm{depth}(v)$ then $\mathrm{lex}(u) < \mathrm{lex}(v)$ for any poset and any lexicographic labelling lex; we state this in the following lemma, and provide a short proof.

Lemma 3 *Let P be a poset with a lexicographic labelling* lex*. For any two elements u and v in P, if* $\mathrm{depth}(u) < \mathrm{depth}(v)$ *then* $\mathrm{lex}(u) < \mathrm{lex}(v)$*.*

Proof: We show that any label at a given depth d is greater than any label at depth $d-1$, by induction on depth d. When $d = 0$, the claim is trivially true. For $d > 0$, let v be any element at depth d and u an element at depth $d-1$. Note that v has an upper cover in depth $d-1$, which by the inductive assumption has a larger lex-label than any element covering u and therefore the multiset of lex-labels of upper covers of v is lexicographically larger than multiset of lex labels of the upper covers of u, and hence $\mathrm{lex}(v) > \mathrm{lex}(u)$ by the definition of a lexicographic labelling. □

The following lemma captures the significance of lex-labellings for the algorithms that follow.

Lemma 4 (The Lex Lemma) *Let P be a poset with a lexicographic labelling* lex*, and suppose* $\mathrm{lex}(a) \geq \mathrm{lex}(b)$*, and there is an element c_1 such that $a||c_1$ and $b \prec c_1$. Then there exists an element c_2 such that $b||c_2$ and $a \prec c_2$, and where* $\mathrm{lex}(c_2) \geq \mathrm{lex}(c_1)$.

Proof: This follows from the definition of lexicographic labelling. □

5 The Bump Number Algorithm

The Greedlex Algorithm for finding a linear extension with the minimum bump number proceeds by first lexicographic labelling the poset, and then shelling minimal elements from the poset "greedily" (avoiding creating a bump involving the element currently being shelled, if possible) and then within the set of choices so allowed, selecting one that is lexicographically greatest.

```
Data: A non-empty poset $P = (E, <)$ and lex : $E \to \mathbb{Z}$ the lexicographic labelling of $E$
Result: A linear extension $\pi$ of $P$ with the minimum number of bumps, and bump is the bump
        number for $P$
$\pi$ is an initially empty shelling sequence
bump $\leftarrow 0$
while $P$ is not empty:
    if there are any minima of $P$ that are not upper covers of the last-shelled node:
        select such a minimum that has maximum lex label
        remove it from $P$ and add it to the end of $\pi$
    else:
        (the last-shelled node was a unique minimum of $P$)
        select a minimum with maximum lex label
        remove it from $P$ and add it to the end of $\pi$
        bump $\leftarrow$ bump $+ 1$
endwhile
```

Algorithm 1: The Greedlex Bump Number Algorithm

The engine of the proof of correctness of Greedlex for computing the bump number is the Lex-Yanking Lemma, stated below, and proved later as Lemma X. **Lex-Yanking Property** Let $P = (E, R)$ be a poset with

lexicographic labelling lex. P has the *lex-yanking property* if the following holds: If P has a linear extension that starts with b and has k bumps, and there is a minimum element a of P such that $\mathrm{lex}(a) \geq \mathrm{lex}(b)$, then P has a linear extension that starts with a and has k or fewer bumps.

The Lex-Yanking Lemma All posets have the lex-yanking property.

The implications of the Lex-Yanking Lemma are that, when constructing a minimum-bump linear extension by shelling the poset (that is, repeatedly removing minimal elements from P, the shelling order being a linear extension of P), it is safe to select an element that has the greatest lex label if it is not a cover of the last-shelled element (i.e., if it does not introduce a bump).

It remains, then, to determine the minimal element to shell when the ones with maximum lex-label are all upper covers of the most recently shelled element. The greedlex algorithm, outlined above, selects an element of maximum lex-label from among elements that do not cover the last-shelled element; if there are no such elements (i.e., the last-shelled element was a unique minimum of the remaining poset), then it selects the element with maximum lex-label from among the minima, and a bump is introduced.

We call a linear extension constructed in this way a *greedlex ordering* of the poset.

Definition 2 A *greedlex ordering* of a poset P is a shelling ordering of P that obeys the following constraints on shelling: a) elements that are covers of the last-shelled element are not shelled, if alternatives exist; b) among those that can be shelled in obedience to (a), an element with maximum lex-label is shelled.

We will provide the proof of the Lex-Yanking Lemma later in this paper, as Lemma 8. First, we will show that *if* a poset P and all of its subposets have the lex-yanking property, *then* any greedlex ordering of the poset is a minimum-bump linear extension. By proving this first as Lemma 7, we will be allowed, in our inductive proof of the Lex-Yanking Lemma, to assume that when the inductive hypothesis can be applied to a smaller poset, then greedlex orderings of the smaller poset have minimum bump number for that subposet. This simplifies the proof.

First, a few lemmas that identify how we can modify linear extensions to get other linear extensions, and what the consequences are for the bump number.

Lemma 5 (Yanking Lemma) *If P is a poset and w is a minimum element of P, and there is a linear extension L with k bumps, then $w \cdot L \setminus \{w\}$ is also a linear extension and has bump number between $k-1$ and $k+1$.*

Proof: The theorem states that a minimal element can always be yanked to the front of the linear extension, with an increase of at most one to the bump number. Let $L = x_1x_2\cdots x_{i-1}wx_{i+1}\cdots x_n$. Then $wx_1x_2\cdots x_{i-1}x_{i+1}\cdots x_n$ is also a linear extension with the same bumps as L except possibly with one (x_{i-1}, x_{i+1}) created and possibly with one (w, x_{i+1}) destroyed. □

Lemma 6 (Swapping Lemma) *Let P be a poset that has a and b among the minimal elements of P, and let L be a linear extension of P with k bumps. If a and b can be swapped without violating a precedence constraint then the resulting linear extension has between $k-1$ and $k+1$ bumps.*

Proof: If the swap does not violate precedence constraints, then both a and b are incomparable to all elements between them; the fact that they are minimal means they are incomparable to elements to the left of them. Hence swapping them can create or destroy only a bump involving the rightmost of a and b and its successor in the linear extension. □

We now prove that the lex-yanking property is sufficient to ensure that a linear extension produced by the greedlex algorithm has the minimum number of bumps.

Lemma 7 *If P is a poset such that the lex-yanking property holds for it and all its induced subposets, then any greedlex ordering of P has minimum bump number.*

Proof: Let P be a poset that has the lex-yanking property, and all its induced subposets have the lex-yanking property. Let $L = x_1x_2\cdots x_n$ be any greedlex ordering of P, and let k be the number of bumps in L.

Let us call an ordering $u_1u_2\cdots u_i$ *extensible* if it is the prefix of some minimum-bump linear extension of P. We are claiming that the prefixes $x_1\cdots x_i$ of L are all extensible. We proceed by induction on i. The empty prefix is clearly extensible.

Suppose that $x_1\cdots x_i$ is extensible. If x_{i+1} does not exist, we are done, so hereafter in the proof we assume $i < n$ and x_{i+1} exists.

Case 1. x_{i+1} has the maximum lex-label of all remaining elements. There are two sub-cases. First, if $x_i||x_{i+1}$ (i.e., no bump is introduced directly after x_i): given that $x_1\cdots x_i$ is extensible, then $x_1\cdots x_{i+1}$ is extensible,

by the lex-yanking property. Second, we consider the case where $x_i \prec x_{i+1}$. The fact that greedlex selected x_{i+1}, an upper cover of the last-shelled element x_i, implies that *all* remaining elements are comparable to x_i, so any extension of $x_1 \cdots x_i$, including the least-bump linear extension, will have a bump after x_i, and by the lex-yanking property of the induced subposet, x_{i+1} can be selected to extend the prefix.
Case 2: x_{i+1} does not have the maximum lex-label of all remaining elements. Then those unshelled elements with higher lex-label than x_{i+1} are comparable to x_i whereas x_{i+1} is not – this is a consequence of the Greedlex Algorithm's selection criteria.

Since $x_1 \cdots x_i$ is extensible, there is a linear extension $x_1 \cdots x_i y_{i+1} \cdots y_n$ that realizes the minimum bump number of P.

Case 2(a): If $x_i || y_{i+1}$, then $\text{lex}(y_{i+1}) \leq \text{lex}(x_{i+1})$, and therefore by the lex-yanking property, there is a linear extension of $P \setminus \{x_1, \ldots, x_i\}$ that starts with x_{i+1} and has no more bumps than $y_{i+1} \cdots y_n$; attaching this suffix to the prefix $x_1 \cdots x_i$ yields a linear extension of P with the same or fewer bumps as $x_1 \cdots x_i y_{i+1} \cdots y_n$, which has the minimum number. Hence $x_1 \cdots x_{i+1}$ is extensible.
Case 2(b): If $x_i \prec y_{i+1}$, then there is a bump x_i, y_{i+1}. Note that x_{i+1} is a minimum of the poset $P \setminus \{x_1 \cdots x_i\}$; therefore, by Lemma 6, we can yank x_{i+1} to the front of $y_{i+1} \cdots y_n$ and obtain a linear extension of $P \setminus \{x_1 \ldots x_i\}$ that starts with x_{i+1} and has no more than one bump more than does $y_{i+1} \cdots y_n$. When re-attached to the prefix $x_1 \cdots x_i$, this results in the sure destruction of one bump and, by the Yanking Lemma, the possible creation of one bump. Hence the resulting linear extension has no more bumps than $x_1 x_2 \cdots x_i y_{i+1} \cdots y_n$, which has the minimum number of bumps for P. Therefore $x_1 \cdots x_i x_{i+1}$ is extensible.

Hence by induction, the greedlex ordering $x_1 x_2 \cdots x_n$ is a minimum-bump linear extension of P. □

We are now ready to prove the Lex-Yanking Lemma for Bump Number.

Lemma 8 (The Lex-Yanking Lemma) *All posets have the lex-yanking property.*

Proof: The proof is by induction on $|E|$. The property is clearly held by posets on zero or one element. Let P be a poset on $n > 1$ elements. Assume the lex-yanking property is held by all posets on fewer elements; therefore, as a consequence of Lemma 7, we may assume that greedlex orderings are minimum-bump for smaller posets.

If P is a poset with only one minimal element, then the selection of the first element of any linear extension is fixed, and the Lemma is trivially true.

So let us turn to the remaining case: let a and b be any minimal elements of P with $\text{lex}(a) \geq \text{lex}(b)$, and let L be a linear extension of P that starts with b and has k bumps; we will use $L = by_2y_3 \cdots y_{t-1}ay_{t+1} \cdots y_n$ to denote the various constituents of L. Let L' be the linear order L induces on the set $E \setminus \{b\}$; hence $L = bL'$.

The poset $P \setminus \{b\}$ is smaller than P, so by the inductive hypothesis any greedlex ordering of $P \setminus \{b\}$ has the minimum number of bumps for that poset. Let $L'_g = x_2x_3 \cdots x_{i-1}ax_{i+1} \cdots x_n$ be a greedlex ordering of $P \setminus \{b\}$. The fact that greedlex did not pick a until after $x_2, \ldots x_{i-1}$ means that $\text{lex}(x_j) \geq \text{lex}(a) \geq \text{lex}(b)$, $\forall j$, $2 \leq j \leq i-1$. It follows that $b \not\prec x_j, \forall j$, $2 \leq j \leq i-1$, so a and b can be swapped in bL'_g without violating precedence. If no new bump is introduced, then swapping a and b in bL'_g gives a linear extension of the same (or smaller) bump number as L, as required by the Lemma.

If bumps are introduced by the swap, then by the Swapping Lemma, it is at most one bump, and that only if $b \prec x_{i+1}$ and $a \not\prec x_{i+1}$.

If that is so, and yet $\text{lex}(a) \geq \text{lex}(b)$, then there is some c' where $a \prec c', b||c'$ and $\text{lex}(c') \geq \text{lex}(x_{i+1})$, by the Lex Lemma. Note that it is possible that c' is not a minimum in $P \setminus \{a, b, x_1, \cdots, x_{i-1}\}$. Let c be a descendent of c' that is a minimum in that poset, and observe that $c||b$ and $\text{lex}(c) \geq \text{lex}(c') \geq \text{lex}(x_{i+1})$. Then by the inductive hypothesis, there is a linear extension of $P \setminus \{a, b, x_1, \ldots, x_{i-1}\}$, call it cL'', that starts with c and has no more bumps than $x_{i+1} \cdots x_n$. Hence the linear extension $ax_1 \cdots x_{i-1}bL''$ has no more bumps than L. □

Theorem 1 *Any linear extension constructed by the Greedlex Algorithm has minimum bump number.*

Proof: By definition, the Greedlex Algorithm produces a greedlex ordering of the input poset. By Lemma XXX, any greedlex ordering is minimum-bump if the poset (and all its subposets) has the lex-yanking property. By the Lex-Yanking Lemma, every poset has the lex-yanking property. □

6 Implementation and Analysis

The Greedlex Algorithm for Bump Number is presented below in greater detail, to permit running-time analysis.

```
Data: A non-empty poset $P$ given as a set $E$ and for each $u$ in $E$, methods to iterate through the upper covers of
      $u$, the lower covers of $u$ and the ability to test if $u$ covers $v$; and lex$[E(P)]$ a lexicographic labelling of $E$
Result: A linear extension $\pi$ of $P$ with the minimum number of bumps
$i \leftarrow 0$
$bumps \leftarrow 0$
for $u \in min(P)$:
    Insert $u$ into the priority queue using $lex(u)$ as the key
repeat
    if priority queue is empty:
        $bumps \leftarrow bumps + 1$
        for $v \in (uppercovers(\pi[i]) \cap minima(P))$:
            Insert $v$ into the priority queue
    else:
        $i \leftarrow i + 1$
        $\pi[i] \leftarrow$ maximum element in priority queue
        remove maximum element from priority queue
        remove $\pi[i]$ from $P$
        for $v \in (uppercovers(\pi[i-1]) \cap minima(P))$:
            insert $v$ into the priority queue with lex$(v)$ as key
until $i = n$;
```

Algorithm 2: The Greedlex Bump Number Algorithm in greater detail

We proceed to outline our implementation, as the basis for our complexity analysis. We take the elements in the poset to be $\{1, 2, \ldots, n\}$, and the number of covers relations to be m, and the up-degree and down-degree of v to be the number of upper covers of v and number of lower covers of v respectively. We can implement data structures for the upper cover lists and lower cover lists as linked lists for each element, of size linear in the number of upper covers and lower covers respectively for each element; we assume that the poset data is given in this form (if only the upper covers lists is given, we can construct the lower cover lists in linear time). We pre-label the elements using the linear time lexicographic labelling algorithm of Sethi.

Each element is inserted and removed from the priority queue exactly once. The priority queue always contains a subset of some antichain in the poset (the priority queue elements are all minimal in the current poset P). Therefore the Remove-Max operations on the priority queue, as well as the insert operations, each run in time $O(\log w)$, where w is the width of the input poset.

The test for inserting elements into the priority queue (i.e., the last line of the "else" loop) requires the ability to test whether an element is minimal in the current poset. We accomplish this as follows: maintain an array numLowerCovers$[v]$, where v runs from 1 to n; numLowerCovers$[v]$ is initially set to be the number of lower covers of v in the input poset P. When we shell an element u from the poset ("remove $\pi[i]$ from P"), we traverse its list of upper covers; for each element v that is an upper cover of u, we decrement numLowerCovers$[v]$, and if that number is now zero, we insert it into the priority queue. In that way, the two for-loops nested within the if-clause and the else-clause can be implemented to take a total time of $O(m + n \log w)$ over the entire run of the program, including the priority queue inserts. Hence the total running time of the algorithm is $O(m + n \log w)$.

The fastest known algorithm for bump number is $O(m + n)$ [7]; that algorithm has not, as far as is known by the authors, been implemented, and would be extremely complex to implement. The Greedlex Algorithm is simple to implement; in fact it has been given as an exercise in undergraduate algorithms course. The implementation is very fast even on large posets of 25,000 elements. [give timing details.]

References

1. Fujii, M., Kasami , T., & Ninamiya, K.: Optimal sequencing of two equivalent processors. SIAM J. Appl. Math., 17:4, pp. 784-789 (1969)
2. Coffman, E., Graham, R.: Optimal scheduling for two processor systems. Acta Informatica, **1**, 200-213 (1972)
3. Habib, M., Möring, R.H., & Steiner, G.: Computing the bump number is easy. Order, **5**, 107–129 (1988)
4. Sethi, R.: Scheduling graphs on two processors. SIAM Journal on Computing, **5**, 73–82 (1976)
5. Corneil, D., Dalton, B. and Habib, M.: LDFS based certifying algorithm for the Minimum Path Cover problem on cocomparability graphs. SIAM Journal on Computing, **42**(3), 792-807 (2013).

6. Gabow, H.N.: An almost-linear algorithm for two-processor scheduling. Journal of the ACM. **29** 766–80 (1982)
7. Schäffer A.A., Simons B.B.: Computing the bump number with techniques from two-processor scheduling. Order bf 5 131–41 (1988)