PATH-PRECEDENCE ORDERS IN TREES

By

Joseph Garrett Linn

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2013

Nashville, Tennessee

Approved:

Jeremy P. Spinrad

Ákos Lédeczi

Lawrence W. Dowdy

Paul H. Edelman

Ross M. McConnell

To my parents and sister for inspiration.

To my wife and son for motivation.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

BACKGROUND

1.1 Definitions

This dissertation assumes basic understanding of algorithmic analysis and complexity. Readers who are unfamiliar with these subjects may refer to Cormen *et al.* [8] for background on algorithms and analysis, and Garey and Johnson [18] for complexity theory. In this section, terms and concepts that may not be prevalent in all computer science disciplines are defined. More detailed information on notation and nomenclature on graphs and graph algorithms is found in Golumbic [20]. For partial orders, see Trotter [39].

1.1.1 Basic Definitions

A *graph* G is a pair (V, E) where V is a set of vertices, and E is a set of edges. Each edge corresponds to a pair $e = (x, y)$ where $x, y \in V$, and e is *incident* on x and y. If $(x, y) \in E$, x and y are *adjacent*, and also x and y are *neighbors*. The *degree* of a vertex x is the number of edges which are incident on x. The *neighborhood* of x is the set of neighbors of x. The *complement* $\bar{G} = (V, \bar{E})$ of G is a graph on the same vertex set with edges present if and only if they are not in E, *i.e.* $\bar{E} = \{(x, y) | x, y \in V, (x, y) \notin E\}$.

If the edges are ordered pairs, then the graph is *directed*. A vertex y is an *in-neighbor* of x if $(y, x) \in E$ and an *out-neighbor* if $(x, y) \in E$. *In-degree* and *out-degree* are the numbers of the in-neighbors and out-neighbors, respectively; and the *in-neighborhood* and the *out-neighborhood* of a vertex are the sets of the in-neighbors and out-neighbors respectively. A vertex which has only out-neighbors is called a *source*. A vertex with only in-neighbors is called a *sink*.

A *partial order*, also called a *poset*, is a pair $P = (X, \prec)$ where X is a set and $\prec$

is a irreflexive, antisymmetric, and transitive binary relation on X. X is sometimes called the *ground set* of P. If for $a, b \in X$, $a \prec b$ or $b \prec a$ then a and b are *comparable*, denoted $a \perp b$. If not, then a and b are *incomparable*, denoted $a \parallel b$. If $a \prec b$, then a *precedes* b and b *succeeds* a. An element b is a *successor* of a if $a \prec b$ and an *predecessor* of a if $b \prec a$. If the relation is also complete, i.e. every set of elements is comparable, the partial order is called a *total order*. If the incomparability relation is transitive, the partial order is called a *weak order*. A weak order is essentially a total order on a set of equivalence classes rather than single elements. The *dual* of a partial order $P = (X, \prec)$, denoted $P^\partial = (X, \prec^\partial)$ is the partial order on the same ground set with the inverse of $\prec$, *i.e.* $x \prec^\partial y$ if and only if $y \prec x$.

Partial orders may be seen as a restricted type of directed graphs (specifically those that are transitive and have no directed cycles) and as such the nomenclature from directed graphs is often used. For example, rather than referring to the size of a successor set, one may refer to the out-degree of an element. Undirected graphs which may have the edges directed so that the directed graph is transitive are called *comparability graphs*. In other words, transitive orientations of comparability graphs are partial orders.

Partial orders are drawn differently than other graphs because including the edges implied by transitivity can obscure the structure of the partial order. An element b *covers* an element a, if $a \prec b$ and there is no element c such that $a \prec c$ and $c \prec b$. The *cover graph* $G = (V, E)$ of a poset $P = (X, \prec)$ is then the graph where $V = X$, and $(x, y) \in E$ if and only if x covers y or y covers x. The cover graph includes those edges that are *not* implied by transitivity. The drawing of a partial order is called a *diagram*. In diagrams, only edges in the cover graph are drawn, and if $x \prec y$ then x will be drawn lower on the page than y will be.

When analyzing the running time of graph algorithms, it is standard practice to use n to denote the number of vertices and m to denote the number of edges. This

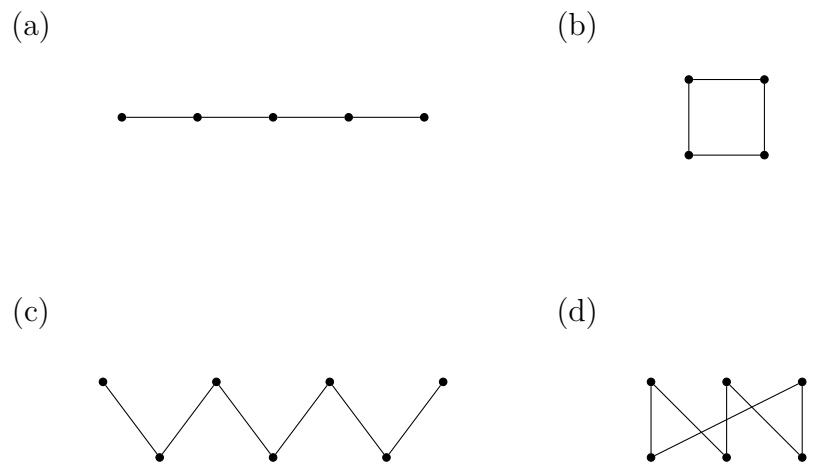


Figure 1.1: Examples of structures: (a) a path of length 4, and (b) a cycle of length 4. The diagrams of (c) a fence of length 6, and (d) a 6-crown.

notation is used for partial orders, using n and m for the size of the ground set and the size of the relation, respectively.

1.1.2 Definitions of Special Structures

In a graph G, a *path* of length n is a set of $n+1$ vertices of G, $v_0, v_1, ..., v_n$ such that for $0 \leq i < n$, (v_i, v_{i+1}) is an edge in G. The vertices v_0 and v_n are the *endpoints* of the path. If the edges are directed, then the path is a *directed path*. In a directed path, v_0 is called the *startpoint* and v_n the *finishpoint* of the path. If $v_0 = v_n$, then the structure is instead a *cycle*, or a *directed cycle* if the graph is directed.

In a partial order, a directed path is called a *chain*. A *fence* is a set of of n elements, $x_0, x_1, ..., x_{n-1}$ such that for $0 < i < n-1$, x_i is only comparable to x_{i-1} and x_{i+1} and either $x_{i-1} \prec x_i$ and $x_{i+1} \prec x_i$ or $x_i \prec x_{i-1}$ and $x_i \prec x_{i+1}$. In a fence, each successive pair is comparable, but the "direction" of the comparability alternates. A $2n$*-crown* is a set of $2n, n > 2$ elements $x_0, x_1, ..., x_{2n-1}$ where, for $i \leq n$, $x_i \prec x_{2i}, x_{(2i+1)\%n}$. A crown is essentially a fence that cycles around to its beginning.

A graph is *connected* if for every pair of vertices $x, y \in V$, there is a path from x

to y. A partial order is connected if the cover graph of the partial order is connected.[1] For a partial order, this definition is equivalent to requiring that any two elements are endpoints of a fence.

A *subgraph* of a graph $G = (V, E)$, is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq \{(x, y) \in E \mid x, y \in V'\}$. An *induced subgraph* of $G = (V, E)$ is a graph $G'' = (V'', E'')$ such that $V'' \subseteq V$ and $E'' = \{(x, y) \in E \mid x, y \in V''\}$. A subgraph is a subset of vertices and a subset of the edges whose endpoints are both in the subset of vertices. An induced subgraph is a subset of the vertices and *all* the edges whose endpoints are both in the subset of vertices.

A *tree* T is a graph for which, between every pair of vertices in T, there is a unique path connecting that pair. Trees may equivalently be defined as connected acyclic graphs. A *directed tree* is a graph whose underlying undirected graph is a tree. A *root* in a directed tree T is defined as a vertex v such that there exists a path from v to every other vertex in T. A *rooted tree* is a directed tree with a root. To emphasize the difference between rooted and other directed trees, directed trees which do not necessarily have a root are called *general trees* in this dissertation.

A *clique* is an induced subgraph where there is an edge between every pair of elements. An *independent set* is an induced subgraph with no edges. In partial orders, a clique corresponds to a chain, and independent sets are called *anti-chains*.

A *linear extension* of a partial order $P = (X, \prec)$ is a total order $T = (X, \prec_T)$ on X which respects $\prec$, i.e. if $x \prec y$ then $x \prec_T y$. Linear extensions are lists of elements where if two elements are comparable in the partial order, then they have the same relative order in the list. A *realizer* for a partial order $P = (X, \prec)$ is a set of linear extensions of P whose intersection is exactly $\prec$. The *dimension* of a partial order (sometimes called linear dimension) is the number of linear extensions in the minimum realizer. When partial orders have low dimension, it is convenient to refer

[1] As in [39], this is equivalent to every two elements having a series of pairwise comparable elements such that x begins the series and y ends the series.

to linear extensions as lists. For example, with a two dimensional partial order, there are first and second lists which can be manipulated and processed independently.

1.1.3 Certifying Algorithms

Certifying algorithms arise because implementations of algorithms can have errors. In addition to the solution to a problem, certifying algorithms return proof of the solution which can be verified independently. This idea was introduced by Mehlhorn [31]. A more recent and comprehensive treatment of the subject is given by McConnell *et al.* [27].

A problem that admits a certifying algorithm is the recognition of bipartite graphs, "Is a graph G bipartite?". Bipartite graphs are those graphs which may be partitioned into two sets such that no edge has both endpoints in the same set. It is well known that a graph is bipartite if and only if it does not contain a cycle of odd length. A recognition algorithm can therefore certify a "yes" answer by returning the partition of vertices, and the caller can confirm that all the edges have one endpoint in each set. A "no" answer can be certified by returning the set of vertices which are members of the odd cycle. For bipartite-graph recognition, the certificates can be constructed without increasing the time complexity of the recognition algorithm [27].

1.1.4 Partition Refinement

In several situations arising in this dissertation, there is a partition of vertices into sets, and one would like to efficiently refine those existing partitions given a vertex x into vertices that are neighbors of x and non-neighbors of x. Partition refinement was discussed as a general algorithmic technique by Paige and Tarjan [35]. The approach taken here was described by McConnell and Spinrad [29].

The key to performing this operation quickly is the data structure. First the graph is preprocessed, making a doubly-linked list node corresponding to each vertex. This vertex list node includes a pointer to its current set, as well as pointers to the list-

nodes of its neighbors. The vertices are all added to the list such that partitions are consecutive. Sets are also represented with by doubly-linked list nodes, with additional pointers to the first and last members of the set in the vertex list. The order of the set list begins is the same as the order as the sets in the vertex list. This preprocessing takes $O(m+n)$ time.

Using this structure, it is straight forward to partition a vertex v into a new set T, from an old set S. Using a pointer to v, remove v from the vertex list, adjusting the first and last pointers of S if necessary. Then, create a new set T, and insert that into the set list beside S[2]. Finally, make v the first and last vertex of T, and T the current set of v. These operations are easily accomplished in constant time, given the preprocessing.

When partitioning more than one vertex from a set, it is important to know whether the creation of a new set is required, or whether v should be added to the existing set. This problem is solved by maintaining a clock incremented after a partition step, and a creation time-stamp is associated with each set list node. If the time-stamp is equal to the current clock, then this set was created at the current partition step, so the v is added to the existing set. Otherwise, a new set is created. In this way, an existing partition can be refined by the neighborhood of a vertex in time proportional to the size of the neighborhood.

If new sets are created adjacent to the old ones in the list, then the relative order of vertices in partitions is maintained. If at some point the algorithm has partitioned the vertices into three sets and the order of the set list is RST, then as long as new partitions are created directly before or after the current set, all vertices currently in S will always be after all vertices in R, and before all vertices in T. In this way, partitions are refined, but never reordered.

[2]It is a matter of choice where to put the new set as long as it can be accessed in constant time from the source set.

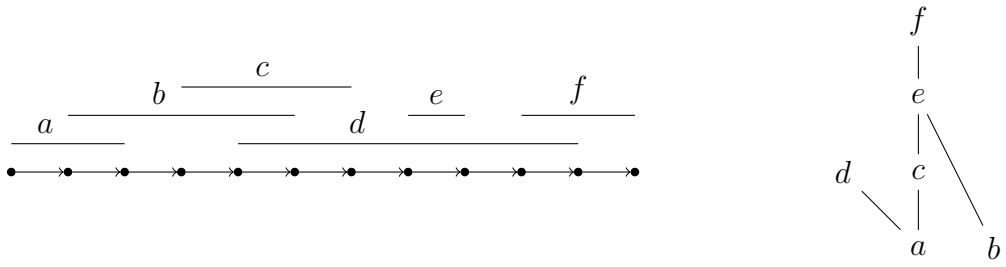


Figure 1.2: An interval model and the diagram of the corresponding interval order.

1.2 Interval Orders

The first mention of interval orders in the literature appears to be a paper by Norbert Wiener in 1914 [45]. Interval orders have been well studied, and the subject of a book by Fishburn [14], as well as a great deal of other work. Interval orders are formally defined as follows:

Definition 1.1. A partial order $P = (X, \prec)$ is an *interval order* if and only if there is a totally ordered set, $(Y, <_0)$, and a mapping, F, from X to closed intervals in Y, such that for all $x, y \in X$, $x \prec y$ if and only if $F(x) <_0 F(y)$.

Informally, interval orders are the orders of strict interval precedence on a totally ordered host (often taken to be the real line). An interval x precedes an interval y if and only if the finishpoint of x is less than the startpoint of y. The intervals are closed, meaning the endpoints are part of the interval. This implies that if two intervals share only an endpoint, they are intersecting and incomparable. The mapping of elements to intervals is called the interval model. The total order is referred to as the host of the interval model, and to the intervals as the embedded objects. Figure 1.2 shows an interval model and the corresponding interval order. As in the figure, $F(a)$ is referred to as the interval of a or simply a if the context is not ambiguous.

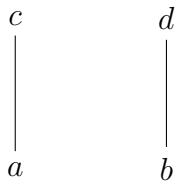


Figure 1.3: The forbidden induced suborder for interval orders

1.2.1 Characterization

A *characterization* of a graph class is a property that is true if and only if the graph is in the class.[3] A forbidden induced-subgraph characterization of a graph class C is a set of graphs S such that a graph G is not a member of C if and only if G contains, as an induced subgraph, a member of S. When the graph is a partial order, the terms forbidden induced subgraph and forbidden induced suborder are used interchangeably. Interval orders have a very simple characterization by forbidden induced suborder, shown in Figure 1.3 [14]. The forbidden suborder shall be denoted in-line as $\mid\ \mid$.

Theorem 1.1 ([14])**.** *A partial order is an interval order if and only if it does not contain contain* $\mid\ \mid$ *as an induced suborder.*

Proof. $\mid\ \mid$ is not an interval order because one of the minimal intervals, without loss of generality a, must end on an element in the total order that is less than or equal to the other, b. Since a ends first a's out-neighborhood contains b's out-neighborhood. In $\mid\ \mid$, the out-neighborhoods of the minimal intervals are incomparable; so it is not an interval order.

To show the other direction, if a partial order does not contain $\mid\ \mid$, the interval representation can be construct of the partial order. First, order the elements by out-neighborhood set containment. For two comparable elements this can be done because the relation is transitive. For two incomparable elements, if two out-neighborhoods are incomparable by set containment then there is a $\mid\ \mid$. This ordering on out-

[3]Since graph classes are defined with such a property, the term characterization is reserved for properties other than those included in the definition of a graph class.

neighborhoods is a weak order, and the equivalence classes are exactly those elements with identical out-neighborhoods. For each equivalence class create a corresponding element in a totally ordered host, and assign that element as the finishpoint for each member of the equivalence class. Assign as the startpoint for an element x the host element directly succeeding the host element corresponding to the first equivalence class that has x as an out-neighbor. This assignment preserves the partial order relation with the precedence of intervals. □

A characterizing representation of a graph class is a representation that is possible if and only if the graph is in the class [37]. Interval orders admit such a representation. Interval orders are exactly those partial orders where each element can be represented by two numbers (the endpoints of the intervals). The element x precedes y in the interval order if and only if the larger number (finishpoint) associated with x is less than the smaller number (startpoint) associated with y. To avoid problems of arbitrary precision, the numbers can be ordered and stored integers in the range 1 to n, in order of the endpoints.[4] Since those numbers take $O(\log n)$ space to store, interval orders can be stored in such a manner in $O(n \log n)$ bits, which is asymptotically better than the the $O(n^2)$ required for general binary relations.

1.2.2 Recognition

Algorithm 1.1 for interval order recognition follows the algorithm given by Belfer and Golumbic [1]. It uses the observation from the forbidden induced suborder proof that the out-neighborhoods can be ordered by set containment. By first ordering the elements of the partial order increasingly by the size of the out-neighborhood, an interval model can be built incrementally, starting with maximal elements. This ordering and transitivity guarantee that when an element is added, all of its out-neighbors have their finishpoints placed in the model already. An element is *closed*

[4] Although it may initially seem there may be $2n$ endpoints, the preceding proof implies that only one endpoint is needed for each unique out-neighborhood.

Algorithm 1.1: Interval Order Recognition

input : A graph $P = \{X, \prec\}$
output: An interval model of the partial order, or a | |

Let $n \longleftarrow |X|$
Let open $\longleftarrow \emptyset$
Let closed $\longleftarrow \emptyset$

Order $x_1, x_2, \ldots, x_n \in X$ by increasing out-degree
for $i \longleftarrow 1$ to $|X|$
  if there exists $y \in$closed such that $x_i \not\prec y$
    find z such that $x_i \prec z$ and $x_{i-1} \not\prec z$
    return $\{x_i, z, x_{i-1}, y\}$, a | |
  x_i.finishpoint $\longleftarrow n$
  for each z such that $x_i \prec z$ and $z \in$ *open*
    z.startpoint $\longleftarrow n + 1$
    closed $\longleftarrow$ closed $\bigcup\{z\}$
  open $\longleftarrow$ open $\bigcup\{x_i\}$
  $n \longleftarrow n - 1$
return the startpoint and finishpoint assignments for X

when both its startpoint and finishpoint have been assigned in the interval model. An element is *open* when only its finishpoint has been assigned.

As the model is built, the last element added precedes all of the closed elements in the interval model. When the i^{th} element x_i is considered for the interval model, there are two possible outcomes. If x_i's out-neighborhood contains the set of closed elements, any open elements are closed at the current minimal element of the model host, and then the host is extended with a new minimal element, which becomes the finishpoint of x_i. If there exists a closed element y which x_i does not precede, then, by the ordering of increasing out-degree, there must be an open element z which x_i does precede. Since x_{i-1} precedes y but not z, and $x_i \parallel x_{i-1}$ and $y \parallel z$ by transitivity constraints, there exists a | | .

This algorithm runs in $O(m + n)$ time. Ordering the elements takes $O(m + n)$ using a bucket sort. Partitioning the open and closed sets takes time proportional to the size of an element's out-neighborhood, giving $O(m)$ time overall. To verify the

certificate of a "no" requires scanning two adjacency lists, $O(n)$, while verifying a "yes" takes $O(m+n)$ time to check the model faithfully represents the partial order.

This dissertation also examines two types of problems which, for the graph classes discussed, are related to recognition. A construction problem on a graph class is the problem of building a representation of the class. For example, given a partial order, a construction problem on interval orders is building the interval model. As the certifying recognition algorithm also constructs the interval model, the construction problem on interval orders takes $O(m+n)$ time. The counting problem on a graph class is counting how many different graphs there are of size n in that class. If a graph can be represented in $O(n \log n)$ bits, then there are $2^{O(n \log n)}$ members of the class on n vertices.

1.2.3 Optimization Problems

The optimization problems considered in this work are finding the size of the maximum clique, and finding the size of the maximum independent set. Both of these problems are NP-hard on general graphs [18]. For comparability graphs and partial orders, there are polynomial algorithms for these problems. Finding a maximum clique in a partial order is equivalent to finding the longest chain. This can be done by modifying a topological sort algorithm to keep track of the longest chain ending at each vertex and can be done in $O(m+n)$ time [20]. Finding the largest independent set reduces to maximum flow, which can be done in $O(m\sqrt{n})$ time [20, 38].

When given the interval model as input, there are faster algorithms for these problems. It is assumed for these algorithms that the models are structured in the following way. First, the host is structured as a list, with the list node corresponding to each endpoint containing a references to the intervals which end there. Each interval stores references to its start and finishpoints in the host list. A model of this type can be constructed in linear time if only endpoints in the range from 1 to n are

given. However if the values of the endpoints are unbounded, and no linear order is given, then the endpoints must be sorted, taking $O(n \log n)$ time, which dominates the time complexity.

Work on these problems has been done mostly on the complement class, interval graphs. Modified versions of the algorithms for independent set on interval graphs [22] and minimum coloring on interval graphs [23] are presented here. Finding the number of colors used in a minimum coloring is equivalent finding the size of a maximum clique for interval graphs[5].

For finding the size of the maximum clique, Gupta *et al.* [22] make the following observation. The interval with the earliest finishpoint, a, can always be in a clique of maximum size. If a is not in a maximum clique, there is some intersecting element b that is. The element b can then be replaced with a. Since a finished before b, a precedes all of the elements that b precedes, and therefore can be substituted into the clique. Therefore, to find the maximum clique, first find the interval that ends first, add it to the clique, throw away all incomparable vertices, then repeat.

Here, the algorithm presented is modified in order to generalize more easily to the classes of graphs to be studied later. Each endpoint in the totally ordered host records the size of the largest clique that precedes that position. The minimum endpoint is assigned a zero value, and endpoints are then considered in order. For every finishpoint at a endpoint position, the size of the largest clique ending with that interval is computed by looking at the value directly preceding its startpoint. The value assigned to the endpoint is either the maximum value from all of the finishpoints, or the value at the previous endpoint, whichever is larger. This algorithm is presented in Algorithm 1.4. It computes the maximum clique in $O(n)$ time given the interval model: constant time for each interval and constant time for each endpoint in the host total order.

[5]Interval graphs are a subset of perfect graphs, in which the size of a maximum clique and a minimum coloring are the same, see [20].

Algorithm 1.2: Interval Order Maximum Clique

input : pos the the current endpoint position in the model. The algorithm assumes that that each earlier position stores the size of the maximum clique before that position in the model, in a data member called max. The next data member has a pointer to the next endpoint position in increasing order. prev_max which is the size of the maximum clique ending at or before the previous endpoint position.

output: The size of the largest clique in the interval order.

```
pos.max ⟵ prev_max
current_max ⟵ pos.max

for each finishpoint of x at pos
    m ⟵ (startpoint of x).max + 1
    if m > current_max
        current_max ⟵ m

if pos.next = ∅
    return current_max
else
    return Maximum Clique(pos.next, current_max)
```

This independent set algorithm uses the fact that interval intersection has the Helly property: if each pair-wise intersections of a subset of intervals is non-empty, then the intersection of all of the intervals in the subset is non-empty. In an interval model, if such a subset intersects anywhere, then the intersection contains an endpoint position. The algorithm for finding the maximum independent set is simply counting at each endpoint position the number of intervals intersecting at that position. This is accomplished by keeping a running total of current intervals. At each endpoint, the number of startpoints is added to the total to give the number of intersecting elements, then the number of finishpoints is subtracted. This algorithm is shown in Algorithm 1.3. Since each element has only two endpoints, this algorithm works in $O(n)$ time.

Algorithm 1.3: Interval Order Maximum Independent Set

input : pos the current position of endpoints in the model. open the number of intervals spanning the interval from the previous to the current position, max the size of the largest independent set in the model before the current position

open $\longleftarrow$ open $+|\{x|x$ starts at pos$\}|$
if open $>$ max
  max $\longleftarrow$ open
open $\longleftarrow$ open $-|\{x|x$ finishes at pos$\}|$
if pos.next $= \varnothing$
  return max
else
  return Maximum Independent Set(pos.next, open,max)

1.3 Generalizations of Interval Orders

There has been work on a number of classes which generalize interval orders. This section reviews different avenues of generalization, as well as pertinent results. Many of these classes are derived by generalizing the totally-ordered host of interval orders, and then using embedded objects in that host that have a natural ordering. These embedded objects are usually some sort of connected subset of the host, thereby generalizing the closed intervals.

Faigle *et al.* [12] define a class called generalized interval orders, as those partial orders where the out-neighborhoods of the elements are either disjoint or ordered by set containment. Garbe [17] gives a linear-time recognition algorithm for this class, as well as algorithms for some well-known graph parameters. Garbe also showed precedence-constrained scheduling of generalized interval orders is NP-complete.

Interval dimension generalizes the concept of (linear) dimension. For a partial order P, interval dimension is the minimum number of interval orders whose intersection is P. Asking whether the interval dimension of a partial order is k is NP-complete for $k \geq 3$ [47]. Much of the work on orders of interval dimension 2 has been done under the name trapezoid orders. The trapezoid name originates from imagining

the two totally-ordered hosts as parallel, and drawing a line between the startpoints and finishpoints of the intervals that represent the same element. The longest chain and anti-chain can be found in $O(n \log n)$ time [13]. There has also been some work done on unit models, in which each object is the same size, and proper models, in which no object contains another [6]. More work on this model has been done on the intersection graph class, trapezoid graphs(see [3, 9, 26]).

Tsoukiàs and Vincke defined a generalization of interval orders called PQI-interval orders based on a further refinement of incomparability [41]. For standard interval orders, if two intervals intersect they are incomparable. For PQI-interval orders, P represents the notion of precedence used in interval orders, Q is the relation "more to the right," where two intervals may intersect but one does not contain the other, and I which represents interval containment. Tsoukiàs and Vincke give a polynomial time recognition for this class, but this class has not been extensively studied in the literature.

Bogart [4] reviews generalizations of interval orders derived from generalizing the host structure and the intervals themselves. Classes using a weak order as the host [5], as well as some classes of lattices [32], have yielded nice characterizations. The class of tolerance orders is obtained by associating a tolerance with each interval in a standard interval model [21]. One interval x precedes another y if x begins before y and they intersect by less than the smaller of the two tolerances. There are variations on this model based assigning different tolerances to the different ends of the interval (called bitolerance) and bounding the tolerance to a certain length (see [4]).

Kratsch and Rampon [25] introduced a notion of visibility in a tree. One embedded object x "sees" another object y if and only if: (a) x and y do not intersect, and (b) there is a path from an element of x to an element of y in the tree. They define as tree-visibility orders the visibility orders of subtrees of an in-rooted tree, a tree where there is a single vertex which succeeds all other vertices. They also give a

forbidden induced-suborder characterization and an $O(mn)$ recognition algorithm for tree-visibility orders.

Müller and Rampon continue the work on visibility by defining further restrictions [34]. Of particular interest here is their notion of partial visibility, which further requires for $y \prec x$ that the path be from a minimal element of x to a maximal element of y. They show that, in this formulation, the visibility orders of paths in a rooted tree are (order) dual to the generalized interval orders of Faigle *et al.* [12][6]. They also show that the classes of visibility and partial visibility of trees in a rooted tree are equivalent.

1.4 Two-Dimensional Partial Orders

Dimension was introduced as a partial order parameter by Dushnik and Miller [11]. Two-dimensional partial orders have many nice properties that and have been well studied.

Two-dimensional partial orders do not have a nice forbidden induce-suborder characterization, but Dushnik and Miller give several characterizing properties. A *non-separating linear extension* is a linear extension in which, for every element a, the out-neighbors of a are consecutive, and directly follow a.

Theorem 1.2 ([11])**.** *The following four properties of a partial order P are equivalent.*

1. *P is co-comparability.*
2. *There exists a linear extension of P which is non-separating.*
3. *The dimension of $P \leq 2$*
4. *P is representable by the containment in an interval model.*

The difficulty of recognizing partial orders of dimension k steeply increases in difficulty as k increases. Recognition of total orders, *i.e.* one-dimensional partial

[6]The source of the order duality is the in-rooted tree host rather than the definition of a rooted tree used here.

Algorithm 1.4: Longest Increasing Subsequence

input : pos current position in the sequence, seq. Best an array, which at index i stores the value of the last element in an increasing subsequence of length i, such that the value is minimum among all last elements in increasing subsequences of length i ending before the current position.

output: the longest increasing subsequence in seq.

```
if pos == seq.end
  return Best.length()
i ⟵ BinarySearch(Best, pos.value)
Best[i + 1] ⟵ pos.value
return LongestIncreasingSubsequence(pos+1, Best)
```

orders, can easily be done in $O(m+n)$ time, by modifying a algorithm for topological sort to fail if there is ever a choice of which element to choose next. Recognition of two-dimensional partial orders was proven to be $O(m + n)$ by McConnell and Spinrad [28]. This time bound is achieved with a linear-time modular decomposition algorithm, which is extended to produce the two-dimensional realizer, if it exists. Recognition of k-dimensional partial orders for $k \geq 3$ was shown by Yannakakis [47] to be NP-complete. In the same paper, Yannakakis showed that recognition of k-dimensional, bipartite partial orders is NP-complete for $k \geq 4$. The time complexity of recognizing three-dimensional bipartite partial orders remains open.

When the input to the algorithm is the two-dimensional realizer, there are algorithms for clique and independent set which are faster than the general algorithms for comparability graphs. The algorithms for clique and independent set are essentially the same, because the maximum independent set of a graph is the maximum clique in the complement. For two-dimensional partial orders, the complement is also two-dimensional. The complement can be computed given the realizer by reversing the order of one of the lists, *i.e.* one of the two total orders in the realizer.

The problem of finding the size of the maximum clique in a two-dimensional partial

order reduces to finding the longest increasing subsequence. Each element is labeled with a value equal to its position in the first list. Then, these values are processed in order of the second list. There is a an increasing subsequence of length k if and only if there is a clique of size k. Suppose there is have an increasing subsequence of size k. Since the labels of elements are the positions in the first list, and the order in the second list is maintained, then the subsequence represents a set of elements each following the next in both lists. This is a clique in the partial order. Similarly, in a clique of size k, there are k elements whose order relative to each other is the same in both lists. This is an increasing subsequence: values increasing from the first list, and positions increasing in the second.

The algorithm for finding the longest increasing subsequence works as follows (since all position indexes are unique, the algorithm is simplified). An array at index i, stores the last element in "the best" increasing subsequence of length i. Here, "the best" means the increasing subsequence of length i that has the lowest label of the final element when compared to all other subsequences of length i. Keeping this specific subsequence ensures that if an element extends any subsequence of length i, it extends the one stored. For each element a, in order of the second list, search the array to find the longest subsequence in the array which is extended by a.

To achieve an $O(n \log n)$ time bound, the elements are entered into the array using binary search. This works because the values in the array are in sorted order. Clearly, the empty array is sorted. Then, when element e is added, e is added after the largest value which is smaller than e. e would not be added before something smaller, because e would extend that subsequence. The complete algorithm, for unique values, is given in Algorithm 1.4. It should be noted that, using an advanced data structure called a Van Emde Boas Tree, it is possible to achieve a time bound of $O(n \log \log n)$ [43]. Describing a Van Emde Boas tree is beyond the scope of this work, and therefore only the $O(n \log n)$ algorithm is presented.

For computing the independent set, a reverse order of the second list is used, essentially computing size of the maximum clique of the complement.

1.5 Path-Precedence Orders

This section defines the class of path-precedence orders, a generalization of interval orders motivated by a well-known hierarchy of intersection graphs. Like the tree-visibility model of Kratsch and Rampon [25], path-precedence generalize interval orders in the same sense that interval graphs have been generalized to larger classes of intersection graphs. Precedence seems a more natural interpretation of these orders, as visibility is not inherently transitive.

1.5.1 Intersection Graph Hierarchy

A graph is an intersection graph if the graph can be modeled by the intersection of objects in some host. For example, the intersection of circles in the plane or arcs of a circle give rise to classes of intersection graphs. Work on intersection graphs has led to a well-known hierarchy of well-studied, tractable intersection graph classes.

$$\text{Interval Graphs} \subset \text{Path Graphs} \subset \text{Chordal Graphs}$$

Interval graphs are the intersection graphs of intervals on the real line (also the intersection of subpaths in a path). Path graphs are the intersection graphs of paths in a tree. Chordal graphs are the intersection graphs of subtrees in a tree [19].[7]

Chordal graphs have a linear number of maximal cliques, which may be enumerated in linear time. Efficient algorithms for these classes often arise from arrangements of maximal cliques as the vertices of the host models. For interval graphs, there is a linear arrangement of maximal cliques such that the maximal cliques containing the same vertex are consecutive [15] in the model. For path graphs and chordal graphs,

[7]Chordal graphs are defined as those graphs having no chordless cycles on four or more vertices. This is a characterization of the class.

there is a tree arrangement of maximal cliques such that the maximal cliques containing a vertex form a path or tree, respectively [19, 33]. These representations can be constructed in $O(m+n)$ time [7, 20, 37]. Using these models can lead to efficient algorithms for problems that are NP-hard in general. Spinrad gives a summary of results on these three classes [37].

1.5.2 Path Precedence in Directed Trees

While this hierarchy of models has been well studied in intersection graphs, partial orders based on these models have received relatively little attention. Given the current understanding of interval orders, it seems natural to explore the orders defined by these types of models. Here, different notions of precedence for paths in directed trees are defined.

In an undirected tree, there does not seem to be a natural order of precedence. Monma and Wei give two types of directed trees in their work on intersection models [33]. In computer science, trees are often rooted, and there is a natural direction away from that root. There is another natural host class, which arises by taking an undirected tree, and directing the edges. By the nature of trees, the resulting directed graphs are acyclic. As such, reachability defines an order on the tree vertices, since reachability is anti-symmetric and transitive in directed acyclic graphs. In models with directed-tree hosts, the direction of the edges restricts the embedded objects in the tree. For paths, there must be a directed path in the tree from one endpoint (the startpoint) of the embedded path, to the other (the finishpoint). This notion of precedence of the tree vertices can define natural notions of precedence for the embedded objects.

Strict Precedence

To generalize interval orders, the notion of precedence from interval orders is adapted to trees. Interval orders require that $F(x) <_0 F(y)$, i.e. the finishpoint of the pre-

ceding interval must be strictly less than the startpoint of the succeeding interval. This can be adapted to trees by requiring that there be a path in the tree from the finishpoint of the preceding path x to the startpoint of the succeeding path y. Since x and y are themselves paths in the tree, there is a single path in the tree that contains x, y and the non-trivial path from the finishpoint of x to the startpoint of y. This notion is called strict precedence.

Definition 1.2. A partial order $P = (X, \prec)$ is a *strict path-precedence order* if and only if there exists a directed tree, T, and a mapping, M, from X to directed paths in T, such that for all $x, y \in X$, $x \prec y$ if and only if there is a directed path, D, in T that contains $M(x)$ and $M(y)$ and $M(x)$ precedes $M(y)$ along D.

The notion of strict precedence for paths is the same as that of visibility of paths, as y is visible from x and they do not intersect. As such, the proof of Müller and Rampon [34] implies that rooted, strict path-precedence orders are equivalent to generalized interval orders. The class yielded when the host model is a general tree has not been studied.

Weak Precedence

There is another natural notion of precedence both for intervals and for paths. Rather than require that one path completely precede another, it is required only for $x \prec y$ that the startpoint of x precede the startpoint of y and the finishpoint of x precede the finishpoint of y. This notion is called *weak precedence*.

With intervals, this notion of precedence gives a class equal to the two-dimensional partial orders.[8] An interval model can be created from a two-dimensional partial order by laying out the two realizers one after the other, and creating an interval that spans from the occurrence of an element in the first list to the occurrence in the second list.

[8]It is well known that containment of intervals gives a two-dimensional partial order. The proof is similar, just reverse the order of the second list.

Since one element precedes another if and only if it precedes that element in both lists, weak precedence represents that relation. Given an interval model, the first list is created by ordering the startpoints, and the second by ordering the the finishpoints. Again, the relation is maintained.

Generalizations of this notion of precedence to paths in trees have not been studied.

1.6 Contributions of the Dissertation

This dissertation studies the precedence orders of paths in trees. Linear-time algorithms that construct the realizer for partial orders with a tree diagram are presented in Chapter 2 for both rooted and general trees. A forbidden induced-suborder characterization and new certifying recognition algorithm are given in Chapter 3 for partial orders of strict precedence in a rooted tree. The orders of weak precedence in a rooted tree are shown in Chapter 4 to have a nice intersection characterization, and such partial orders that are also bipartite can be recognized in $O(m + n)$ time. The independent set and clique problems for both of these classes are studied in Chapter 5, and algorithms for these problems are presented which, given the tree-model as input, are asymptotically more efficient then the well-known algorithms for these problems on comparability graphs. Finally, a forbidden induced-suborder characterization of general, strict path-precedence orders is shown in Chapter 6, leading to an $O(n^2)$ certifying recognition algorithm. The final chapter discusses future directions for research, and open problems.

CHAPTER 2

COMPUTING THE REALIZERS OF TREES

This chapter studies the realizers of the tree orders which serve as hosts for classes of path-precedence orders. Wolk [46] showed that rooted trees are two-dimensional and and Trotter and Moore [40] showed that general trees are three-dimensional. Algorithms are presented which compute the realizer from either type of tree in $O(n)$ time. The approach for developing each algorithm is the same. The proofs of the theorems above imply the existence of a certain kind of embedding for each of the two tree types. By searching the tree in a particular way, the search ordering itself implies the embedding. Then, the properties of the embedding are used to reverse incomparable pairs and construct a minimum realizer. In later chapters, these results will yield efficient representations and class containment relations for our classes of precedence orders.

2.1 Rooted Trees

Wolk first proved that rooted trees are two-dimensional [46]. There are many ways to prove this theorem. Wolk uses the characterization of two-dimensional partial orders as co-comparability. Trotter [39] uses the fact that planar lattices are two-dimensional. Here, an elementary proof is presented that follows an observation by Service [36], that the tree may be embedded in the plane.

Theorem 2.1 ([46])**.** *If P is a partial order such that the diagram of P is a rooted tree,* $\dim(P) \leq 2$.

Proof. The rooted tree can be embedded in the Cartesian plane, such that the ordering of vertices along the x and the y axes is the two-dimensional realizer. Let the corresponding linear extensions be X and Y. The proof proceeds by induction

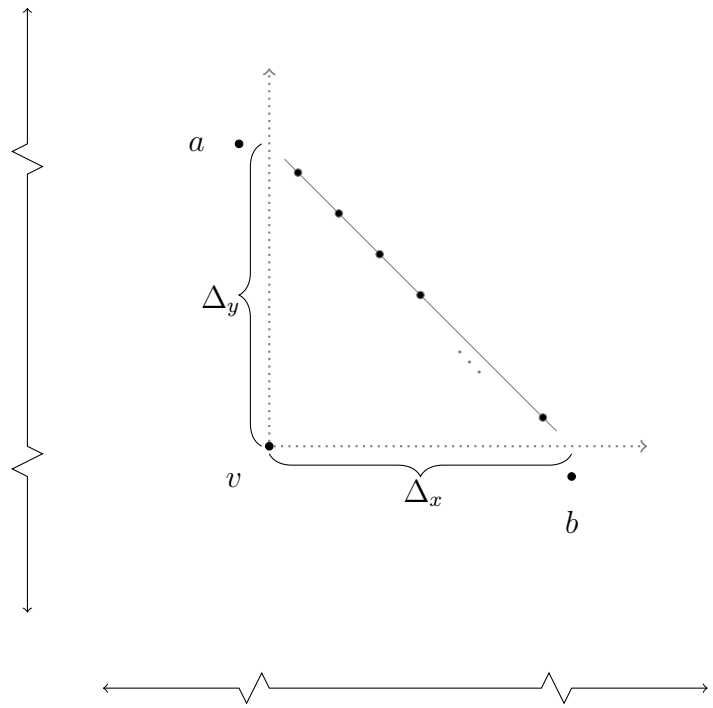


Figure 2.1: The placement of the children of an element v. a is the vertex that immediately succeeds v in Y before the children are added, and b is the vertex that succeeds v in X.

on the height of the tree, adding children to maximal elements. A rooted tree that consists of only the root is of height 1, and clearly that is two-dimensional.

Now, assume that all trees of height h are two-dimensional. All vertices in the tree which are of height h or less can be embedded in the Cartesian plane. For any tree of height $h+1$, there must be at least one vertex of height h with out-neighbors. Let v be a vertex of height h in the tree and positioned in the plane at point (v_x, v_y), with out-neighbors. There must be no vertex in the plane both above and to the right of v, as that element would be an out-neighbor of v. There must exist δ_x, δ_y such that the $v_y + \delta_y$ is less than the difference between v_y and v's immediate successor in Y, and $v_x + \delta_x$ is less than the difference between v_x and v's immediate successor in X. All of the children of v may now evenly be spaced along the line segment from $(v_x + \epsilon, v_y + \delta_y)$ to $(v_x + \delta_x, v_y + \epsilon)$. Figure 2.1 shows the placement of the children of v. The same procedure is done for every vertex in the tree of height h that also has

children.

Consider now the position of these children in X and Y. Clearly, they are after v and everything that preceded v in the tree. By construction, all of the children directly succeed v in both X and Y, and in opposite order, so they are incomparable. The children succeed in X those incomparable vertices that they precede in Y and *vice versa*. The tree is still two-dimensional at height $h + 1$. □

2.1.1 Computing the Two-Dimensional Realizer

The construction process of this tree maintains two invariants. The first is that all of the descendants of a vertex are consecutive, and immediately following that vertex in both X and Y. The second is that the order of the children is reversed in X and Y. The first of these is a well-known property of depth-first search; the second suggests a second depth-first search to complement the first.

Theorem 2.2. *Two depth-first search orderings, where the second depth-first search considers children in the reverse order of the first, create a two-dimensional realizer of a rooted tree.*

Proof. First, depth-first search on a rooted tree produces a valid topological sort ordering, so the orderings themselves are valid linear extensions. It remains to be shown that if two vertices x and y are incomparable their order is reversed in the two extensions. Consider the greatest common ancestor a of x and y. Since the tree is rooted such an ancestor must exist. By assumption, each vertex must be greater than or equal to a different child of a, x' and y' respectively. By the properties of depth-first search, a child of a and all of its descendants will come before the next child of a and all of that child's descendants. Thus, when the children x' and y' are reversed, all of their descendants are reversed as well, including x and y. Two incomparable elements are therefore in reverse order in the two linear extensions. □

Corollary 2.1. *Computing the two-dimensional realizer of a rooted tree is $O(n)$.*

Proof. This follows directly from Theorem 2.2 and from the fact that depth-first search on a tree is $O(n)$ as a tree has $n-1$ edges. □

2.1.2 Reconstructing the Tree

First, the problem of reconstructing the tree given the two-dimensional realizer is considered. The original paper on dimension by Dushnik and Miller [11] gives the following property of two-dimensional partial orders. They define a *non-separating linear extension* as a linear extension $L = (X, \prec_L)$ of $P(X, \prec)$ where $a, b, c \in X$, where $a \prec c$ and $a \prec_L b \prec_L c$ implies either $a \prec b$ or $b \prec c$. They prove the following result:

Theorem 2.3 ([11])**.** *The* $\dim(P) \leq 2$ *if and only if there exists a linear extension of P which is non-separating.*

In the previous section, depth-first search was used to find a linear extension, and this order is non-separating, as all descendants of any vertex are consecutive. For rooted trees, there is a stronger result:

Theorem 2.4. *If P is a partial order such that the diagram of P is a rooted tree, then both linear extensions in any minimum realizer are non-separating.*

Proof. Assume, for the sake of contradiction, that there exists a two-dimensional realizer for $P = (X, \prec)$, with a linear extension $L = (X, \prec_L)$ such that $a, b, c \in X$, where $a \prec_L b \prec_L c$, $a \prec c$, and $a, c \parallel b$. In a second linear extension for a two-dimensional realizer, a must still precede c, and b must be after c and before a, which is impossible. □

The proof here is simplified by the fact that in a rooted tree, there cannot be separation by having an incomparable a and b both precede c, as this is forbidden

Algorithm 2.1: RootedReconstruct

input : The vertices of a rooted tree, where each vertex stores its positions in linear extensions L_1, L_2. Each vertex begins with an empty list of children. The vertex that is first in both realizers must be the root.

output: The root of the rooted tree

```
branchpoints ⟵ empty stack
root ⟵ v such that L1(v) = L2(v) = 1
prev ⟵ root

for each vertex current in order of L1 starting with 2
    if current > prev
        Add(prev.children, current)
        if L2(current)> L2(prev)+1
            Push(branchpoints, prev)
    else
        while L2(Top(branchpoints) < L2(current)
            Pop(branchpoints)
        Add(Top(branchpoints).children, current))
    prev ⟵ current

return the root
```

in rooted trees. Theorem 2.4 shows that all minimum realizers for rooted trees must look like the realizers constructed with depth-first search. This structure can be used to rebuild the tree using a realizer.

The algorithm for rebuilding the tree works as follows. Let the first linear extension of the realizer be L_1 and the second be L_2. First, sort the vertices by their positions in L_1. The vertices will be processed in that order. Consider progressing along a path in the tree with no branches. Nothing is known about the absolute difference between the positions in L_1 and L_2, but while progressing forward, the position in each increases by one. If then, the position in L_1 increases by one, and the position in L_2 increases by a larger amount (but still increases), there must be a branch point. L_2 must have explored some other path but L_1 still increases by one. The algorithm marks that the branch point exists by pushing the position on the stack, and continues. If incrementing in L_1 by one causes a decrease in L_2, then a leaf has been reached,

Figure 2.2: Two graphs whose complements are minimally not a comparability graph [16]. These are the only such graphs with transitive orientations that have tree diagrams [39]

and so the algorithm must retreat to a place where L_2 had previously diverged. This branch vertex is at the position of the largest stored branch point which is less than the current vertex in L_1. Then a new branch is created in the tree, and processing continues. This algorithm RootedReconstruct, and is shown in Algorithm 2.1. For convenience, it is assumed that the input is given as a set of vertices where each vertex stores its position in each list. It is not difficult to transform list input into this format in $O(n)$ time, using a look-up table.

2.2 General Trees

While rooted trees are two-dimensional, there are examples of general trees that are not. A well-known characterization of two-dimensional partial orders due to Dushnik and Miller [11] is that the underlying comparability graph is also the complement of a comparability graph (co-comparability). Gallai gave a forbidden induced-subgraph characterization of comparability graphs [16] in which only two graphs had complements with transitive orientations that had a tree diagram. These graphs are shown in Figure 2.2. As Trotter [39, 40] points out, transitive orientations of these graphs characterize trees that require three dimensions. In this section, a proof by Trotter and Moore is presented that the dimension of a general tree is at most three; and a construction from their proof is used to aid in the computation of a 3-dimensional realizer.

2.2.1 Planar Partial Orders

The notion of planarity often arises in graph theory. A graph is called *planar* if there is a way to draw the graph in the plane (i.e. on one side of a flat surface) such that none of the edges cross. This definition is far too limiting for partial orders. Transitivity requires many edges, and so the standard notion of planarity is incredibly restrictive for partial orders.[1] When drawing partial orders, the diagram is used precisely to avoid those extra edges. Therefore, a partial order is called planar if it admits a planar diagram.

Partial orders have a 0-element, called simply 0, if there is one element which precedes all other elements. For example, the root in a rooted tree is a 0.

Theorem 2.5 ([40])**.** *The dimension of a planar partial order with a* 0*-element is at most 3.*

The formal proof of this result will not be presented here; however the construction of a three-dimensional realizer for such a partial order given by Trotter and Moore is useful. Consider a planar diagram of a planar partial order $P = (X, \prec)$ with a 0-element. For all points in the plane, let $\leq_x$ denote the standard left-to-right ordering of the horizontal axis, and $\leq_y$ denote the standard bottom to top ordering of the vertical axis. For each element $x \in X$, there exists a left-most path from 0 to x and a (possibly identical) right-most path from 0 to x. Since these paths meet at their highest and lowest points, these paths outline a section of the plane. This section is called $S(x)$. By definition, it is a property of S that if $x \prec y$, $S(x) \subset S(y)$.

Two additional relations are defined on X using S. The first, $\prec_L$, can be thought of as increasing while moving to the right, or read as"to the left of." Formally, $x \prec_L y$ if and only if there exists a horizontal line, l, such that $l \cap S(x), l \cap S(y) \neq \emptyset$, and for all $p \in l \cap S(x), q \in l \cap S(y), p \leq_x q$. The second relation is $\prec_U$, and can be thought

[1] Those familiar with planar graph theory can note that a chain of 5 elements has all the edges of K_5, the complete graph on 5 vertices, so before even a taking subgraphs or contracting edges, the height of the posets that are planar in the traditional graph-theoretic sense is limited to 4.

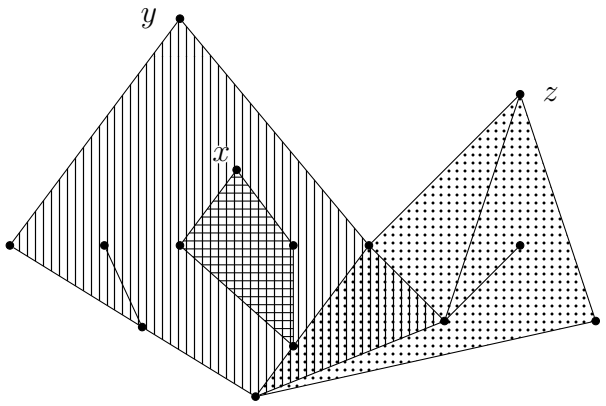


Figure 2.3: Examples of $S(x), S(y), S(z)$. In this example, $S(x) \subset S(y)$ so $x \prec_U y$. While $S(y)$ and $S(z)$ intersect there is a horizontal line across which they are disjoint (for example the horizontal line through z), and so $y \prec_L z$.

of as "underneath." For two elements $x, y \in X$, $x \prec_U y$ if and only if $x \parallel_P y$ and $S(x) \subset S(y)$. For a more detailed description of the properties of $\prec_L$ and $\prec_U$, see [40]. Specifically, $\prec_L$ and $\prec_U$ have the properties necessary for the following result:

Theorem 2.6 ([40])**.** *Let $P = (X, \prec)$ be a planar partial order with a 0-element. The transitive closures of the following relations are a realizer for P:*

1. $\prec \cup \prec_L \cup \prec_U$
2. $\prec \cup \prec_L^{\partial}$
3. $\prec \cup \prec_U^{\partial}$

It is not immediate what relationship a general tree has with a planar poset with a 0-element. Specifically, while it is not difficult to see that a tree is planar, it is not clear that a 0-element can be added to such a tree. Trotter and Moore additionally showed a construction of the planar diagram of a tree with an added 0-element [40]. They showed that it is possible to draw a tree diagram in the plane incrementally such that there is a straight line from the 0-element to every other element. The construction incrementally adds leaf elements (both maximal and minimal elements) to a tree such that the new elements have an edge from 0, and still no edges cross.

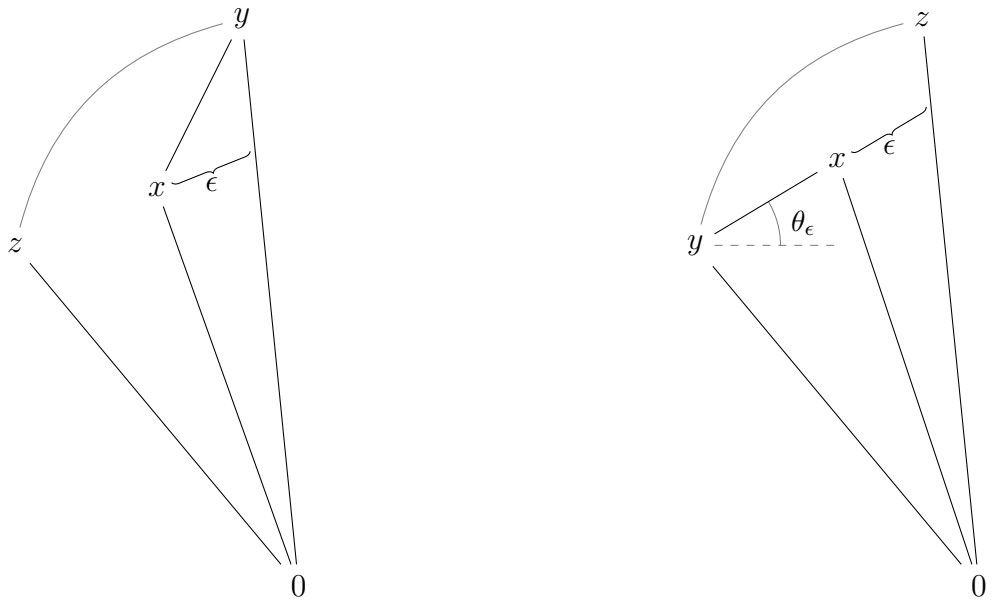


Figure 2.4: How to incrementally add a maximal or minimal element to an existing tree and still have the results be (poset) planar [40]. First x is added as an in-neighbor, then as an out-neighbor. In each case, suitably small ϵ and θ_ϵ always exist.

.

Figure 2.4 shows how to add a maximal or minimal leaf element. When adding an in-neighbor x to an element y, there always exists an offset ϵ from the edge from the 0-element to y such that x can be placed. When adding an out-neighbor x to y, then a suitably small angle θ_ϵ and an offset from the edge to the right of y always exist.

When referring to the left-to-right position of an vertex on the plane, this specifically refers to the relative ordering of the edges coming out of the 0-element.

Proposition 2.1. *There exists a planar embedding of a general tree with an edge from* 0 *to every element such that the edge from* 0 *to an element x is to the right of the edges from all in-neighbors, and to the left of all out-neighbors.*

Proof. This is clearly true by the construction of Trotter and Moore [40]. □

As in the two-dimensional example, the planarity of this construction does not depend on a particular ordering of edges around a vertex, as every such ordering works. All that is required is to fix an ordering, such that two more linear extensions

reverse all incomparable pairs. The following presents a method to find such an ordering, specifically $\prec \cup \prec_L \cup \prec_U$.

2.2.2 Properties of the Planar Tree Diagram

Now consider the properties of the planar drawing of the tree with a 0-element, specifically, the shape of S. By the definition of S if $x \prec y$, $S(x) \subset S(y)$. If x is minimal, then $S(x)$ consists entirely of the edge from 0 to x. For other vertices, S has the following properties.

Proposition 2.2. *For each vertex x, the right-most path from 0 to x is the edge from 0 to x.*

Proof. This is true by Proposition 2.1. □

The left-most extent of $S(x)$ must go through the left-most ancestor of x by definition.

Proposition 2.3. *For each vertex x, the left-most ancestor of x is a minimal vertex.*

Proof. The left-most ancestor of x must be a minimal vertex because, by Proposition 2.1, all in-neighbors would be to the left. □

Proposition 2.4. *For a minimal vertex x, there are no elements underneath x, and all elements to the left of x are descendants of minimal elements to the left of x which are not descendants of x.*

Proof. Since x is minimal, $S(x)$ is only the edge from 0 to x and it cannot contain any vertices, therefore no vertex can be underneath x. To be to the left of x, an element must have its edge from 0 to the left of x's edge from 0. Clearly by Proposition 2.1, the only way to be to the left of x is to connect to x through a descendant, from the left. Those vertices must themselves be descendants of other minimal vertices, also to the left of x. □

This shows that minimal elements make a good base case for building the $\prec_L$ and $\prec_U$ relations. No element is under a minimal element, and if the tree is processed in a left to right order, all previously seen elements are to the left of a minimal element. The notation $\mathrm{DC}_x(y)$ refers to the set of connected components disconnected from x in the partial order (the transitive closure of the tree) when y is removed. Thus, no component in $\mathrm{DC}_x(y)$ contains an ancestor or descendant of x as there are direct edges to those vertices in the transitive closure.

Lemma 2.1. *The vertices to the left of x are those vertices to the left of its left-most ancestor. The vertices underneath x are the vertices in $\mathrm{DC}_x(a)$ for all a that are ancestors of x, except the left-most parent of x, p_L. For p_L only those components of $\mathrm{DC}_x(p_L)$ which contain siblings of x that are after x in clockwise order are underneath x.*

Proof. The statement of elements to the left of x comes from the definition and is stated here only for completeness.

Where a is an ancestor of x, but not the leftmost parent, any member of a component in $\mathrm{DC}_x(a)$ that is comparable with a must be an out-neighbor of a, otherwise it would be an ancestor itself. So the members of $\mathrm{DC}_x(a)$ are to the right of the left-most extent of $S(x)$. They similarly must be to the left of the right-most extent of $S(x)$, otherwise the edge from 0 to x would not be straight. They are incomparable, and contained in $S(x)$ so they are underneath x.

Consider a child c of p whose edge from p is clockwise from the edge from p_L to x. In order for each to have an edge from 0, x must be more right than c, so the right extent of $S(x)$ is to the right of the right extent of $S(c)$. If p_L is the left-most parent of x, then it must also be the left-most parent of c, otherwise there would be an edge crossing. This implies that the left extent of $S(x)$ and $S(c)$ are the same. Since (1) x and c are incomparable, (2) the left extents of $S(x)$ and $S(c)$ are the same, (3) x is

above c, and (4) the right extent of $S(x)$ is more right than that of $S(c)$, c is under x. □

2.2.3 Topological Depth-First Search

Depth-first search was used to compute the two-dimensional realizer for rooted trees. A new variant of depth-first search is used here to aid in computing the three-dimensional realizer for general trees. This algorithm is called topological depth-first search (TDFS) because the order in which vertices are discovered would be a depth-first ordering on the underlying undirected graph, and the numbering scheme yields a topological ordering.

In the rooted-tree case, depth-first search gives a topological ordering because each vertex only has one in-neighbor. In the general-tree case, when a vertex is reached, a vertex may have many in-neighbors. All of those vertices must be visited before the current vertex can be added to the topological sort. Here is the first modification to a typical depth-first search: edges may be traversed in the reverse direction. It is required that all in-edges (going backward along the edge) are explored before any out-edges. The vertex is added to the ordering only after all of the in-edges have been explored, but before any out-edges are explored. The algorithm for TDFS is described recursively in Algorithm 2.2: TDFS-visit.

The ordering given by TDFS on a general tree defines an embedding of the partial order in the plane. For a vertex x, it fixes an order of in- and out-edges around x, in counter-clockwise order staring just below the left horizontal line. In-edges are considered in left-to-right order. When the right-most in-edge is fixed, the edge to the 0-element is added. Then the search continues with the out-edges. There is one complication for edges that are discovered by a child (moving backward along an edge). This edge is fixed as the last edge in counter-clockwise order.

Algorithm 2.2: TDFS-Visit

input : a vertex v, with a list of in-neighbors, N^-, and of out-neighbors N^+, a global clock is assumed to be initialized to 0 before the first call.
result : the vertices in the graph are labeled in tdfs order.

v.visited $\longleftarrow$ true

for each vertex u in N^-
  if u.visited = false
    visit(u)

v.number $\longleftarrow$ clock
clock $\longleftarrow$ clock+1

for each vertex u in N^+
  if u.visited = false
    visit(u)

return

Proposition 2.5. *In the embedding defined by a TDFS order, the clockwise order of edges around* 0 *is the same as the TDFS order.*

Proof. Now, consider the ordering of edges around 0. Let x be the first element that appears in clockwise order around 0 in a position different from its TDFS order. Any element that connects to x through an ancestor other than the left-most parent of x, will be counter-clockwise of x in the order. Otherwise, it would cross the edge from 0 to x. Any vertex that connects to x through a sibling of x that is clockwise of x around its parent must be similarly be counter-clockwise from x around 0. Any vertex that connects to x through a descendant of x or through a sibling of x that is counter-clockwise from x around the leftmost parent must be clockwise from x around 0, otherwise that vertex's edge to 0 would be crossed. However, these are precisely the conditions of a TDFS ordering. □

Proposition 2.6. *For two vertices x and y, if x precedes y in TDFS order, then all vertices in components of $\mathrm{DC}_x(y)$ also precede x in TDFS order:*

$$y \prec_{tdfs} x \Longrightarrow \{z \mid C \in \mathrm{DC}_x(y),\ z \in C\} \prec_{tdfs} x$$

Proof. Let a be the minimum ancestor of x such that $\mathrm{DC}_a(y) = \mathrm{DC}_x(y)$. From this construction, a must be a child of y. Every vertex contained in a component of $\mathrm{DC}_a(y)$ must be to the right of y because everything to the left connects to y through an ancestor, which would also be an ancestor of a. All such vertices must also be to the left of a, otherwise they would cross the edge from 0 to a. Since a is an ancestor of x, for these vertices $\{z \mid C \in \mathrm{DC}_x(y),\ z \in C\}$, the edge from 0 to z must be to the left of the edge from 0 to x, which by Proposition 2.5 means they precede x in TDFS order. □

Lemma 2.2. *A TDFS ordering of a general tree contains $\prec \cup \prec_L \cup \prec_U$ for an embedding in which (1) in-neighbors are visited in counter-clockwise order, (2) the edge to a tree vertex from the 0-element comes after all other in-neighbors in counterclockwise order, and finally (3) out-neighbors are visited in counterclockwise order, with the exception that if a vertex is visited through a child, that child is the last out-neighbor in counterclockwise order.*

Proof. Since TDFS is a topological sort ordering, it clearly contains $\prec$.

By Lemma 2.1, $\prec_L$ is a property of the left-most minimal ancestor a of a vertex, and by definition of S, $S(a)$ is just the edge from 0 to a. By Proposition 2.2, the right-most extent of a vertex is its edge from 0, so every vertex that is to the left of a has already been added to the ordering.

By Lemma 2.1, a vertex in $\mathrm{DC}_x(a)$ for an ancestor a that is not the left-most parent is underneath x. By Proposition 2.6, these precede x in TDFS order. By Lemma 2.1, the other set of vertices underneath x are the components of $\mathrm{DC}_x(p_L)$

which contain siblings of x that are after x in clockwise order. Since TDFS considers out-neighbors of all vertices in counter-clockwise order, these too will precede x in TDFS. □

2.2.4 Constructing the Realizer

The previous section showed that, using the properties of the planar tree diagram, topological depth-first search yields the first linear extension from Theorem 2.6: $\prec \cup \prec_L \cup \prec_U$. This section presents algorithms that rearrange the TDFS ordering of a general tree to create the other two linear extensions for the realizer. The result is a complete algorithm to compute the realizer for a general tree in $O(n)$ time.

Computing $\prec \cup \prec_L^{\partial}$

The approach for the algorithm is to start with an array of list nodes, where the list order and the array order are the same as the TDFS order. The array is processed in order, rearranging the list pointers in such a way as to maintain $\prec$ but to take the dual of $\prec_L$. Since the list nodes are stored in an array, there is constant time access to any node. This method requires supplemental information from TDFS. For each in-edge e to a vertex v, the range of values assigned during the exploration of that subtree must be stored. By the TDFS ordering, the first vertex in the range will be the left-most ancestor of v in that subtree, and the last will be the in-neighbor. This set of vertices will invariably remain contiguous in the ordering, but within the set, the ordering may change.

The algorithm, called Re-Right, proceeds by reversing in the list the order of the ranges assigned to each child. When a vertex v with multiple in-neighbors is reached, the range information stored during TDFS is used to find the range associated with each child. The vertices in all of the child ranges are a contiguous set because these vertices are all of the ancestors of v and all those vertices is $\mathrm{DC}_v(a)$ for each of those

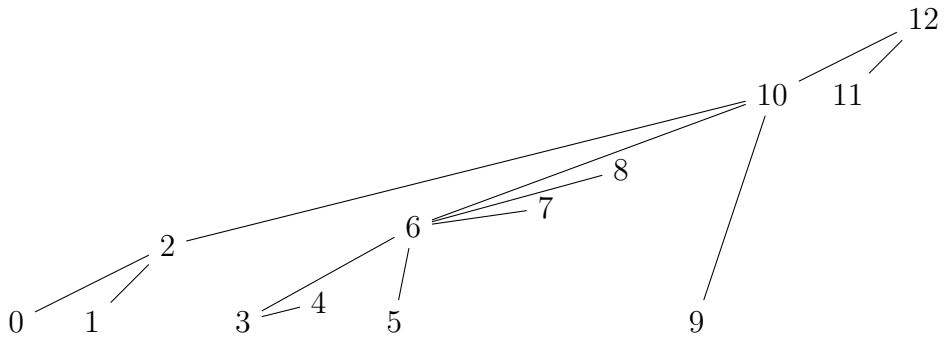


Figure 2.5: A tree labeled in TDFS order to illustrate the algorithm for reversing $\prec_L$.

ancestors a. In TDFS, these are all visited in a block, right before v. The ranges for each of the in-neighbors are likewise contiguous. For each such block, the link in the linked list into and out of each block is found, and then the list is reconnected to those blocks in reverse order. Figure 2.5 shows an example tree labeled in TDFS order. For clarity, the tree is not drawn exactly as the planar embedding described earlier. Figure 2.6 shows the progression of the algorithm on the tree.

The ranges, each of which is in its own box in Figure 2.6, represent the contiguous groups that are being re-ordered, one range for each in-neighbor. Also, the connections within a range are not changed, only the link entering the box and exiting the box. This alone gives a running time that is $O(n^2)$, for a vertex, scanning through the ranges of its in-neighbors for the in and out pointers. Proposition 2.7 shows that the vertices can be easily accessed with the in and out pointers.

Proposition 2.7. *Before the contiguous ranges of a vertex v's in-neighbors are re-ordered, the pointer into each range comes from the vertex directly before the range in TDFS order. Moreover, this is also the edge out of the previous range.*

Proof. Consider the last vertex x in any range. It follows from the TDFS ordering that x is either the in-neighbor of v in the range, or a maximal element that succeeds the in-neighbor of v (the vertex 6 is such a vertex in the example).[2] It follows that

[2]Perhaps a convenient way to think of this is both cases are maximal in the components of vertices

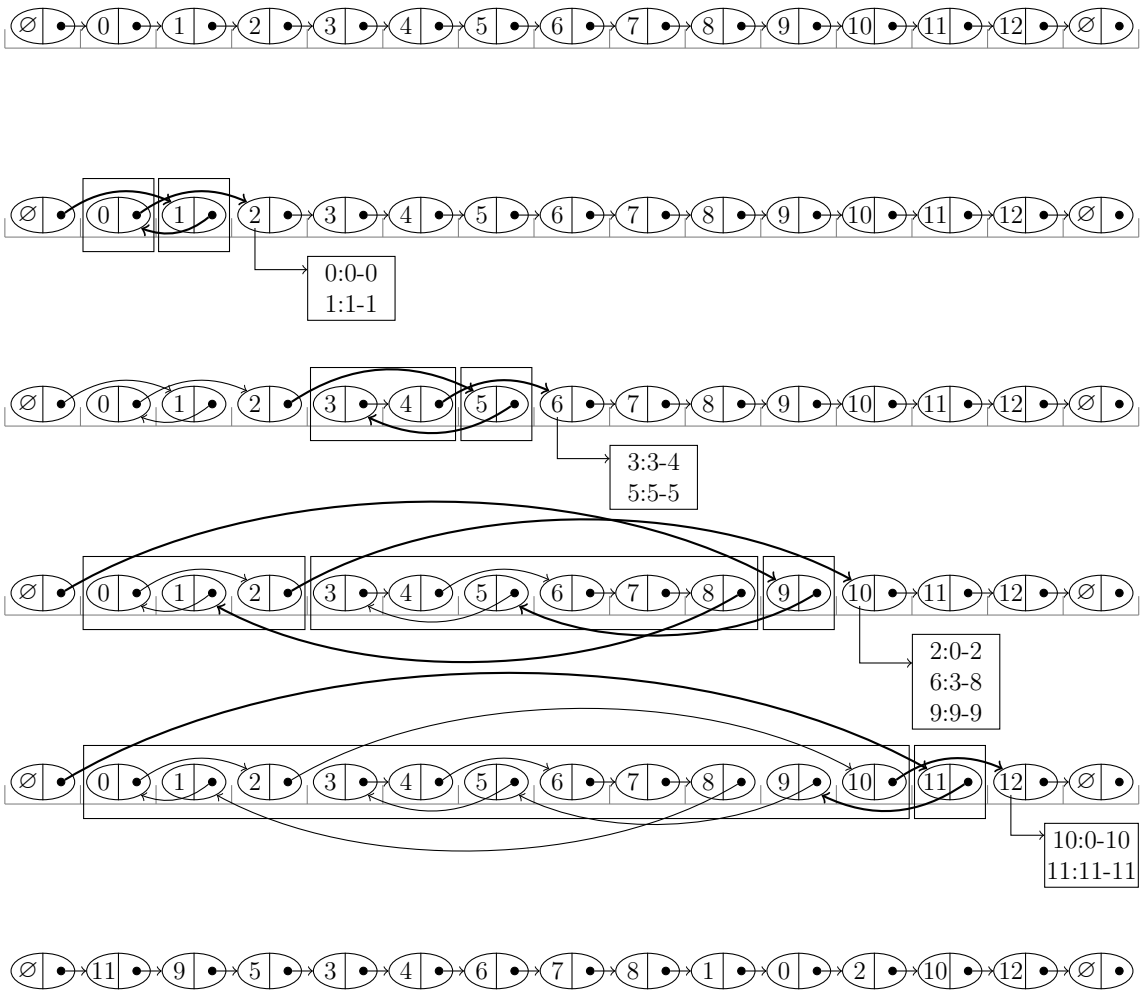


Figure 2.6: An example of Re-Right run on the tree in Figure 2.5

x has not been in a range previously, and so its pointer points to something after it in the TDFS order. Since it is also the last vertex in the range, it points into the range that immediately succeeds it, or if it is the range of the right-most child of v, it points to v.

The vertex (if any) that points into the first range also must be maximal in those vertices numbered before v, and therefore the same argument applies. □

Lemma 2.3. *Re-Right computes a total order containing* $\prec \cup \prec_L^{\partial}$ *in* $O(n)$ *time.*

Proof. The order computed by Re-Right only rearranges links between vertices that

with TDFS numbers $\leq x$.

are disconnected in the tree if the children of a vertex are removed. These vertices are not comparable in $\prec$, and therefore Re-Right maintains $\prec$ from the TDFS ordering.

The order computed by Re-Right reverses all incomparable pairs of $\prec_L$. Assume the contrary: there exist two vertices x, y such that $x \prec_L y$ are not reversed by Re-Right. Consider the fence in P that connects x and y. There can be no maximal vertex in the fence that is not x or y because at such a vertex, the right-to-left ordering would have been reversed. If x and y were comparable, then $x \prec_L y$ would be impossible, so the fence between them must have only one minimal element. This is a contradiction because, for descendants of a common ancestor, one is under the other, so they are not comparable by $\prec_L$.

The extra book-keeping required of the initial TDFS run takes a constant amount of time per edge. For each vertex, Re-Right changes two pointers per in-edge. By Proposition 2.7, these can be accessed in constant time. Therefore, for each vertex, the processing time is proportional to the in-degree. Since in a tree there are $n - 1$ edges, this gives a time complexity of $O(n)$. □

Computing $\prec \cup \prec_U^{\partial}$

The approach for this algorithm is to use the TDFS ordering of the vertices to perform a second search on the tree. This search will again output a topological ordering, but will explore vertices in an order that will reverse those pairs in $\prec_U$. The algorithm works as follows: at a vertex x, the tree is searched backward to find all the ancestors of x. When retreating, instead of exploring the descendants of those ancestors incomparable to x, as in TDFS, those vertices are pushed onto a stack to be explored later. When a maximal element is reached, the top vertex on the stack is explored. When the last child in TDFS order is reached, if x was discovered through that child, the routine returns. If x was discovered through a parent in the tree, then the routine must be called explicitly for the child. The algorithm, called UndoUnder, is given in

Algorithm 2.3: UndoUnderVisit(v)

input : a vertex v, with a list of in-neighbors, N^-, and of out-neighbors N^+ ordered by tdfs. Also a global clock variable, initialized at 0, that will be used to order the vertices, and a global stack.

result : the vertices in the graph are labeled in order of $\prec \cup \prec_U^{\partial}$.

```
v.visited ⟵ true
for each vertex u in N⁻
    if u.visited = false
        UndoUnderVisit(u)
v.number ⟵ clock
clock ⟵ clock+1
if N⁺ = ∅
    UndoUnderVisit(stack.pop())
else
    d⁺ ⟵ |N⁺|
    for each vertex u in N⁺[1] to N⁺[d⁺ − 1]
        stack.push(u)
    if N⁺[d⁺].visited = false
        UndoUnderVisit(N⁺[d⁺])
return
```

recursive form in Algorithm 2.3, UndoUnderVisit. Figure 2.7 shows an example tree, and Figure 2.8 shows the state of the stacks during the running of the algorithm on that tree.

Lemma 2.4. *The algorithm UndoUnder reverses all pairs in $\prec_U$ and runs in $O(n)$ time.*

Proof. Consider a vertex y which is under a vertex x, i.e. $y \prec_U x$. By Lemma 2.1, y falls into one of two cases. In the first case, y is in $\mathrm{DC}_x(a)$ for some a which is not the left-most parent of x, p_L, and in the second case y is in $\mathrm{DC}_x(p_L)$ with some sibling of x that is after x in clockwise order.

Since UndoUnder gives a topological ordering, clearly all of the ancestors of x come before x. For a given ancestor a, clearly it is not maximal, so the children that are not ancestors of x will be pushed onto the stack. These vertices, and the

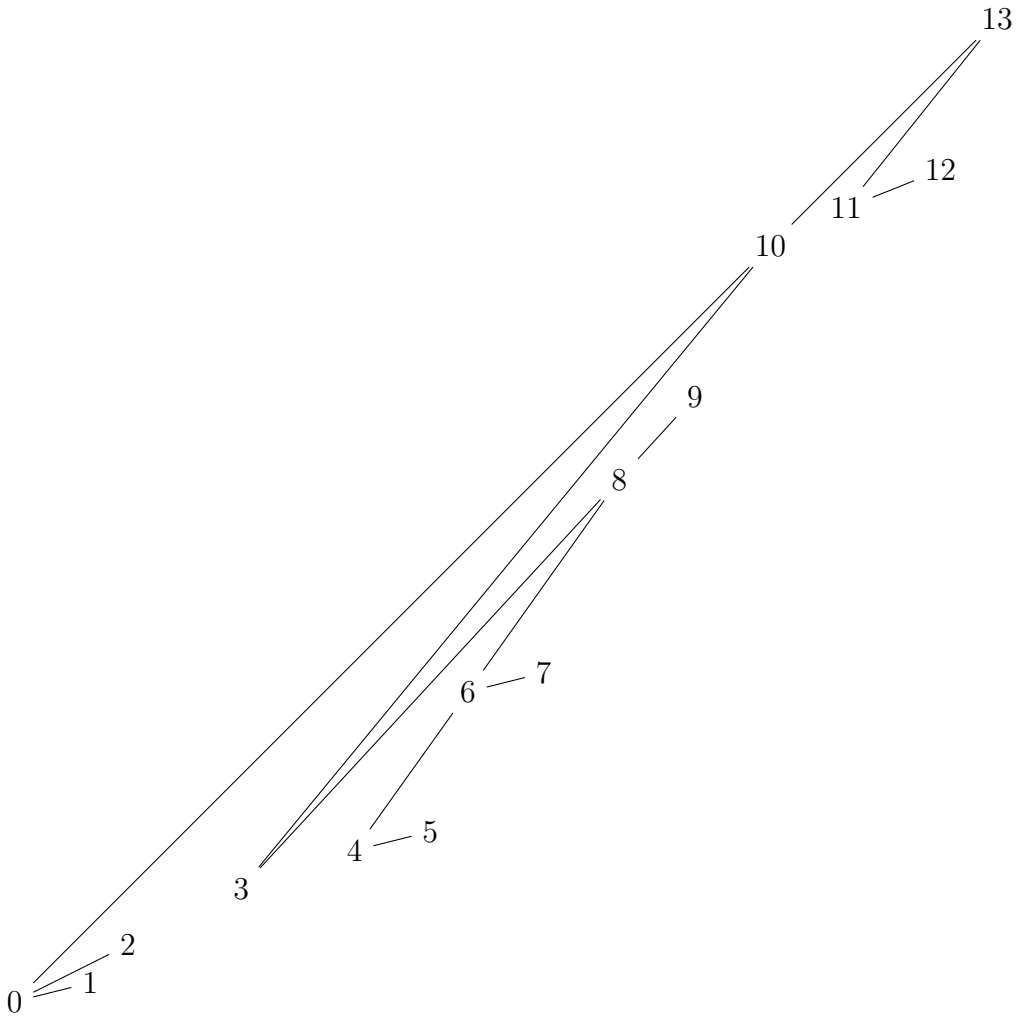


Figure 2.7: A tree labeled in TDFS order to illustrate the algorithm for dualizing $\prec_U$

.

components explored through them, will therefore not be visited until after a maximal element equal to or descending from x. If y is in $\mathrm{DC}_x(a)$, it will come after x.

The children of any parent are placed on the stack in TDFS order, which means they will be explored (as they are popped) in reverse TDFS order. The embedding defines this as clockwise, which means x will precede in the order given by UndoUnder those vertices in $\mathrm{DC}_x(p_L)$ that are clockwise from it around p_L. Therefore if y is in $\mathrm{DC}_x(p_L)$ and $y \prec_U x$, then y will come after x in the UndoUnder ordering.

The search technique used follows every edge once. The operations on the stack

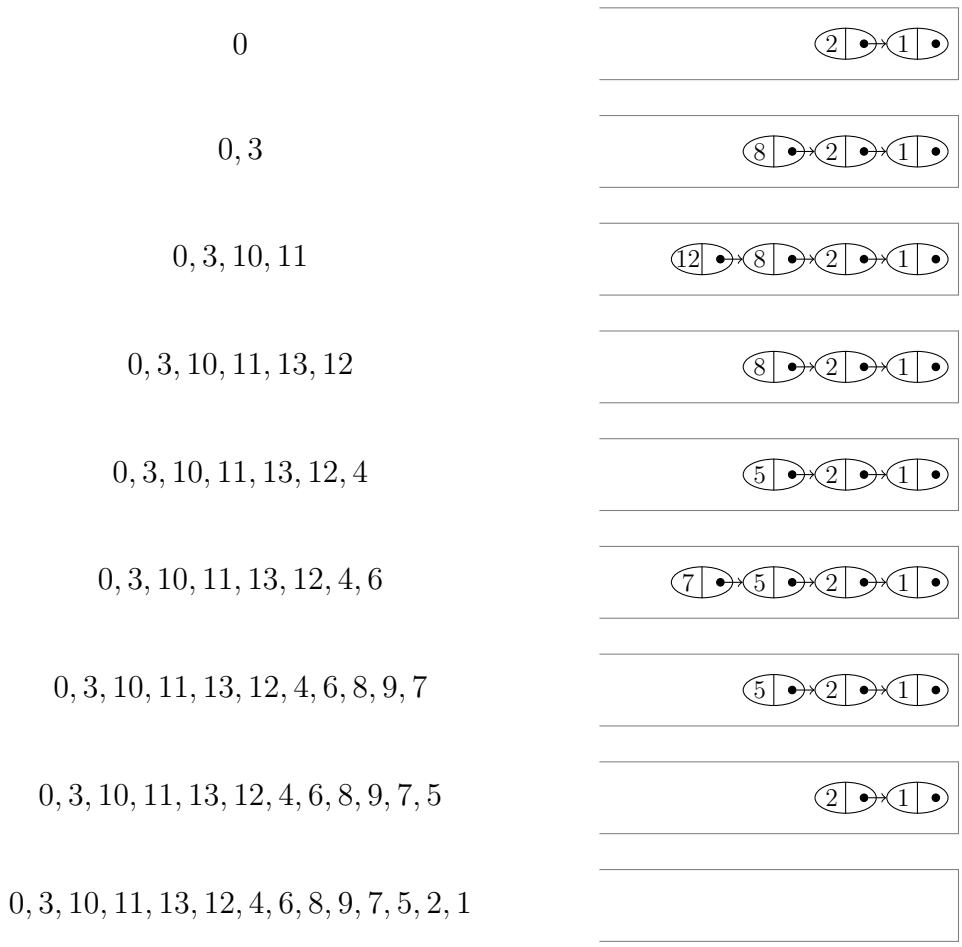


Figure 2.8: An example of UndoUnder run on the tree in Figure 2.7. On the left is a list of numbered vertices in the order they were numbered. The right shows the current state of the stack.

data structures are all constant time and simply change the order in which edges are followed. Since a tree has $n-1$ edges, this gives a time complexity of $O(n)$. □

Lemma 2.4 completes the results needed to find the realizer of a general tree.

Theorem 2.7. *The realizer for the transitive closure of a general (unrooted) directed tree can be computed in $O(n)$ time.*

Proof. To compute the realizer, one runs TDFS, and uses the output to then run ReRight and UndoUnder. Since the input is a tree, TDFS runs in $O(n)$ time. It follows from Lemmas 2.3, 2.4 that the entire procedure is $O(n)$. □

CHAPTER 3

ROOTED, STRICT PATH-PRECEDENCE ORDERS

This chapter presents results on rooted, strict path-precedence orders. This class was first studied by Faigle *et al.* [12] under the name generalized interval orders. Müller and Rampon [34] proved that generalized interval orders were equivalent to the strict precedence of paths in a rooted tree. Garbe [17] presented an $O(m+n)$ recognition algorithm.

First, this chapter develops a forbidden induced-suborder characterization of rooted, strict path-precedence orders. This characterization gives a new proof that this class is equivalent to generalized interval orders. Moreover, the characterization gives a method to certify that a partial order is not a rooted, strict path-precedence order. This fact contributes to the development of an $O(m+n)$ certifying recognition algorithm for the class.

3.1 Forbidden Induced-Suborder Characterization

The definition of generalized interval order given by Faigle [12] specifies that the out-neighborhoods of the elements are either ordered by set containment or disjoint. This condition defines the set of forbidden induced suborders for the class. Let a and b be the elements with the out-neighborhoods that are not disjoint but cannot be ordered by set containment. As such, a and b must have a common out-neighbor d (or the out-neighborhoods would be disjoint), and each must have a private out-neighbor, c and e respectively. Since there are no restrictions on the out-neighborhoods of the children, two additional comparabilities may exist without violating transitivity: $c \prec d$ and $e \prec d$. The three forbidden induced suborders result from allowing neither, one, or both of these edges. These orders are shown in Figure 3.1.

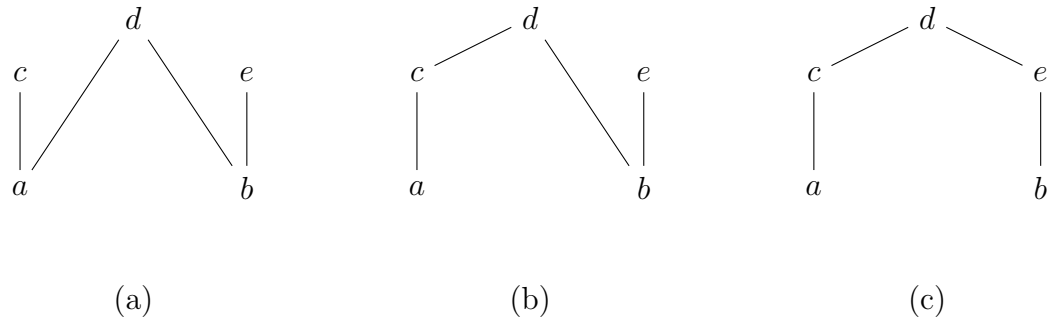


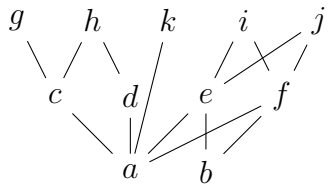
Figure 3.1: The three forbidden induced suborders for generalized interval orders and for rooted, strict path-precedence orders.

The equivalence of generalized interval orders and rooted, strict path-precedence orders is shown by demonstrating that these three partial orders are precisely the set of forbidden induced suborders for rooted, strict path-precedence orders. The first observation is that all of the predecessors of an element must end on the same root-to-leaf path in the tree, which forces the out-neighborhoods (i.e. successor sets) to be comparable.

Proposition 3.1. *All predecessors of any element in a rooted, strict path-precedence order must have comparable successor sets (by set containment).*

Proof. Let $P = (X, \prec)$ be a rooted, strict path-precedence order. Assume, for the sake of contradiction, that $a, b \in X$ have incomparable successor sets, yet share a common successor d. Let c be a's distinguishing successor, let e be b's distinguishing successor.

If a and b have incomparable neighborhoods, then the finishpoint of a and the finishpoint of b must not lie along the same root-to-leaf path in the tree, otherwise (as in interval orders) the set of one's successors would contain the others. This, though, is a contradiction of the host being a rooted tree, as that would make two different paths from the root to startpoint of d, one through the finishpoint of a, the other through the finishpoint of b. □



$ghijkdcefba$

Figure 3.2: A partial order to illustrate the procedure for building a tree model, and an ordering of vertices to build the model. For example, in the ordering d precedes c because d has lesser out-degree.

Proposition 3.1 forbids the three partial orders in Figure 3.1.

Lemma 3.1. *None of the partial orders:*  ,  , *or*  , *are rooted, strict path-precedence orders.*

Proof. Assume for the sake of contradiction, that there is a tree model for any of the three orders. By Proposition 3.1, the minimal elements, a and b share a common successor and therefore the finishpoints lie along the same root-to-leaf path. This is a contradiction because they have incomparable successor sets so the finishpoints of a and b cannot lie on the same root-to-leaf path. □

The sufficiency of forbidding these three partial orders is shown by construction of the tree model. The algorithm, called Tree Model Construction, shown in Algorithm 3.1, incrementally adds elements to the tree model in order of increasing out-degree. This ordering ensures that when adding an element e to the model, all successors of e are already in the model, and that no element currently in the model has out-degree in the partial order greater than that of e. Figure 3.2 shows the diagram of an example partial order and an ordering in which the elements could be considered.

An element with the finishpoint assigned to a tree vertex and the startpoint unassigned is called "open". This is depicted in Figure 3.3 with an downward pointing arrow at the root of the subtree. An element with both endpoints assigned is called

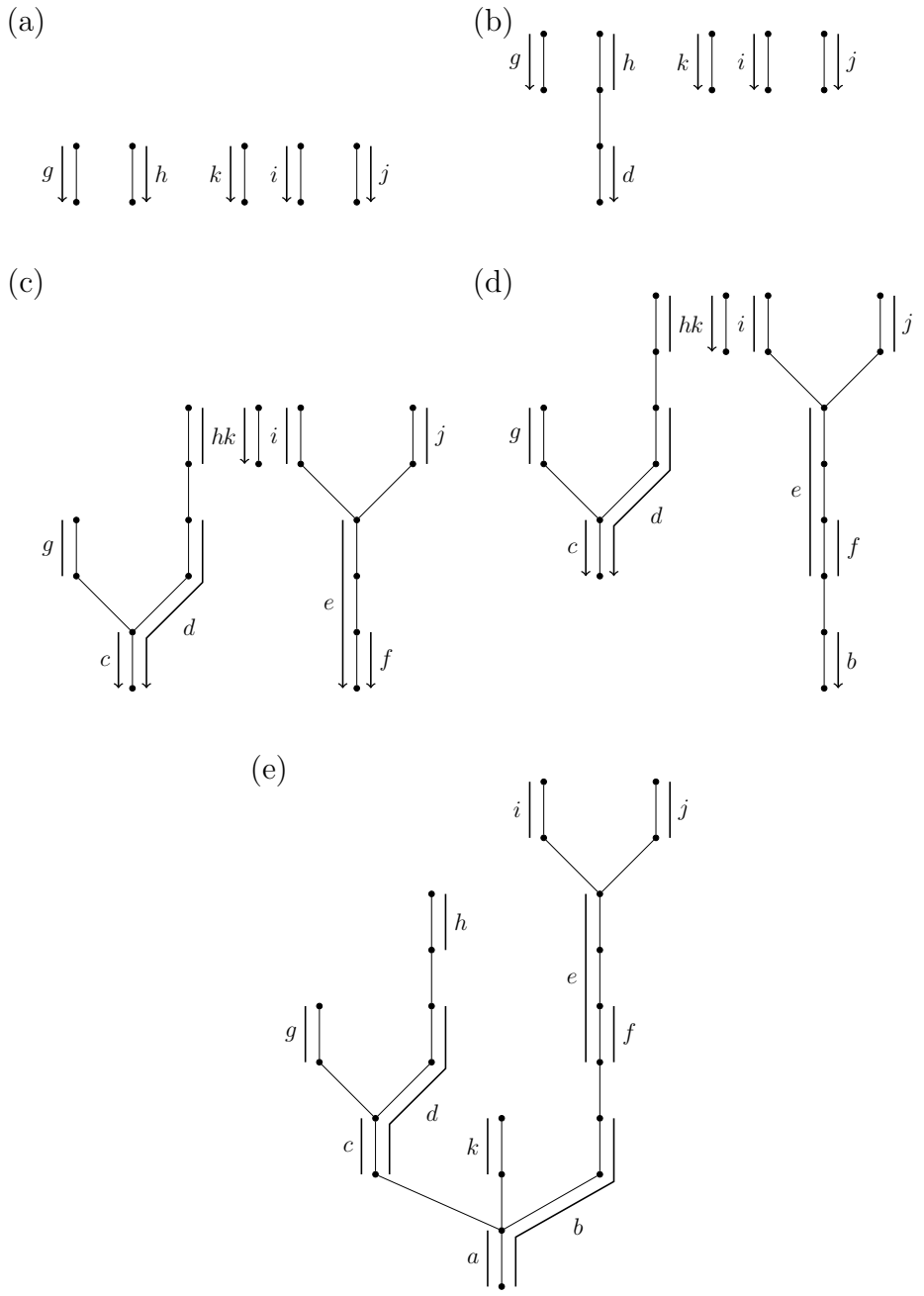


Figure 3.3: The building of the tree model for the partial order in Figure 3.2: (a) after adding the maximal elements, (b) after adding d, (c) after c, e, and f, c and e each joining two subtrees together, (d) after adding b, (e) after adding a and closing the remaining open elements a and b.

Algorithm 3.1: Tree Model Construction

input : A partial order $P = (X, \prec)$
output: If P is a rooted, strict path-precedence order, then Tree Model Construction returns the tree model. If not, it returns FAILURE

for each element $x \in X$ in order of increasing out-degree
  Let S be a new subtree with root r
  Add vertex l and edge (r, l) to S
  x.finishpoint $\longleftarrow l$

  for each subtree R containing a neighbor of x
    R.open-successors $\longleftarrow R$.open $\bigcap N(x)$
    R.closed-successors $\longleftarrow R$.closed $\bigcap N(x)$
    R.open $\longleftarrow R$.open $- N(x)$
    R.closed $\longleftarrow R$.closed $- N(x)$

    if R.closed $\neq \emptyset$
      return FAILURE

    for each $y \in R$.open-successors
      y.startpoint $\longleftarrow R$.root

    S.closed $\longleftarrow S$.closed $\bigcup R$.closed-successors $\bigcup R$.open-successors
    S.open $\longleftarrow S$.open $\bigcup R$.open
    Add tree R and edge $(l, R.\text{root})$ to the tree S
  S.open $\longleftarrow \{x\} \bigcup S$.open

Let T be a new subtree with root r
for each remaining tree S
  Add tree S and edge $(r, S.\text{root})$ to T
  T.closed $\longleftarrow S$.closed $\bigcup T$.closed
  T.open $\longleftarrow S$.open $\bigcup T$.open
  for each $x \in S$.open
    x.startpoint $\longleftarrow r$

return T and the start and finishpoint assignments for X

"closed." In Tree Model Construction, subtrees maintain two sets of elements that have endpoints assigned to their vertices, one of open and one of closed elements.

For each element x, a new subtree S is added to the model, consisting of a root and a leaf vertex. The finishpoint of x is the leaf vertex. If x is maximal, then all that remains is to add x to the new tree's open set. Figure 3.3a shows the tree model after all of the maximal elements from Figure 3.2 are added.

If x is not maximal, for each existing subtree in which x has successors, the open and closed sets are partitioned using the out-neighborhood of x. Those elements which are in closed and are out-neighbors of x are moved to the closed-successors set, and those out-neighbors in open are moved to open-successors similarly. Proposition 3.3 and Lemma 3.2 will show that if any subtree is partitioned in this manner and still has any elements left in its closed set, then Tree Model Construction fails to construct a valid tree model for the partial order. Moreover, the partial order contains one of the three forbidden induced suborders.

If a partitioned tree has an empty closed set, then the algorithm proceeds by first closing the elements of open-successors by setting their startpoints to the root of their current subtree. Next, open-successors and closed-successors are added to the closed set of S, and open to the open set of S. Then, the structure of the subtree is added to S along with a new tree edge from the leaf of S to the root of that subtree. Once all of the partitioned subtrees are processed, x is added to S's open set. Figure 3.3b shows an element added to a single subtree and Figure 3.3c shows elements with successors in multiple subtrees being added. The addition of element c in Figure 3.3c not only joins two subtrees, but illustrates how an open non-neighbor, in this case d, extends its path to the current root of the subtree.

Tree Model Construction ends by adding a new root and closing all those elements which are still open (minimal elements in the partial order) with their startpoints at that new root. If there is still more than one subtree (*i.e.* the partial order is not connected), then connect the subtrees to that new root.

Two observations about Tree Model Construction lead to the sufficiency argument:

Proposition 3.2. *During Tree Model Construction, an open element has no successors (*i.e. *a maximal element is open) only when it is the only path in its subtree.*

Proof. By the structure of the Tree Model Construction procedure, maximal elements become closed during the same step as they are added to a larger subtree.

For the remainder of the procedure, all open elements added to that larger subtree precede those maximal elements. □

In a subtree that has multiple elements, Proposition 3.2 implies that each open element has a closed successor. Furthermore, by transitivity, a new element cannot precede only an open element in such a subtree.

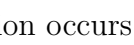
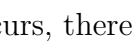
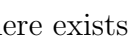
Proposition 3.3. *Tree Model Construction fails to construct a valid tree model only when there is an element that precedes a proper subset of the closed elements in a subtree.*

Proof. When Tree Model Construction creates a tree for a maximal element, there are no closed elements to consider. Similarly, when those maximal elements are joined to a larger subtree and are subsequently closed, there are no closed elements to consider.

When an element x precedes elements in a larger subtree, any open successors are closed, and all other open elements can be extended in the tree past the finishpoint of x. Therefore, open elements pose no problem. No closed successor poses a problem because x's path ends before any closed successor's path begins. Only those elements that are not successors of x but have had their path closed before x is added violate the precedence relation. Therefore, if an element x precedes all of the closed elements in every subtree in which it precedes any elements, then all precedence relations in the new tree are maintained. □

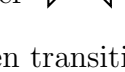
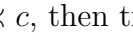
Lemma 3.2. *If a partial order does not contain as an induced suborder*  *,* *, or* *, then it is a rooted, strict path-precedence order.*

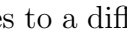
Proof. A partial order is a rooted, strict path-precedence order if and only if it admits a tree model. Proposition 3.3 tells us that Tree Model Construction will successfully build a tree model if, as it progresses, there is no point at which an

element precedes only a subset of the closed elements of a subtree. It remains to be shown that when this condition occurs, there exists a  ,  , or  as an induced suborder.

Let the partial order for which the tree model is being constructed be $P = (X, \prec)$, the element currently being added to the tree model be called x, the current subtree in which x has at least one successor be called S, and the last element that was successfully added to S be called a. The out-degree of x is greater than or equal to the out-degree of a, and since a was added previously, all of its successors are in S and closed.

There are two cases to consider: the case where x has strictly fewer successors in S than does a, and the case where x has an equal or greater number of successors in S.

For the former case, there exists some element c, that is a successor of a and incomparable to x. By Proposition 3.2, x precedes an element with a closed path in S, and therefore a and x share a common successor, d (the relationship of c and d is not specified). Since x has out-degree in the partial order at least as great as a, there must be some element e in another subtree, that x precedes, but is incomparable to a, c, and d. These five elements form either  , if c and d are incomparable, or  , if $c \prec d$. If $d \prec c$ and not $x \prec c$, then transitivity is violated.

In the second case, x has an equal or greater number of successors in S than a. In this case, it is possible that all of x's successors are in S. If there are some number of closed elements (successors of a) that are incomparable to x, then x must have at least one open successor. Let c be that closed successor of a which is incomparable to x, and let e be the successor of x with an open path. The element e is therefore incomparable to a and (by transitivity) to c. By Proposition 3.2, there exists some closed element, d, that is necessarily a successor of x, e, and a. If c precedes d, then there is a  . If c branches to a different path and does not precede d, then there

is a  . □

This concludes the necessary arguments for the first result:

Theorem 3.1. *A partial order is a rooted, strict path-precedence order if and only if it does not contain as an induced suborder*  ,  , *or*  .

Proof. The result follows directly from Lemmas 3.1 and 3.2. □

Corollary 3.1 ([34])**.** *Generalized interval orders and rooted strict path-precedence orders are equivalent.*

Tree Model Construction builds a model which has $2n$ elements, leading to the counting result.

Theorem 3.2. *There are* $2^{O(n \log n)}$ *rooted, strict path-precedence orders on* n *elements.*

Proof. By Proposition 6.8, if a tree can be constructed, the size of the constructed tree is $O(n)$. Tree Model Construction builds a tree with $2n$ elements which can be stored in $O(n \log n)$ bits for tree vertices tree edges. Each element in the partial order stores the label of the tree vertices that are the endpoints. This takes an additional $O(n \log n)$ bits. Since the partial order can be stored uniquely $O(n \log n)$ bits, the result follows. □

3.2 Recognition in Linear Time

A partial order can be certified as a rooted, strict path-precedence order by giving a rooted tree and an assignment of the elements to paths in that tree. Theorem 3.1 shows that it can be certified as not a rooted, strict path-precedence order by showing that either  ,  , or  is an induced subgraph. In this section, Tree Model Construction is modified to additionally return the appropriate certificate for

the yes/no answer. Following the running-time analysis of the recognition algorithm, the methods for verifying certificates are discussed.

Achieving a linear time bound requires a set data structure that allows adding an element to the set, taking the union of sets, and, given a pointer to an element within the set, removing it all in constant time. The doubly-linked list used for partition refinement algorithms(see [20, 29]) described in the first chapter is an example of such a structure. The preprocessing necessary to construct the list structure takes $O(m+n)$ time.

3.2.1 Constructing the Forbidden Induced Suborder

Subroutine 3.2: Find Obstruction

input : An element x and a subtree R in which x has a neighbor but $R.\mathsf{closed} - N(x) \neq \emptyset$

output: A set of elements which induce one of the forbidden induced suborders

Let c be any element in $R.\mathsf{closed}$
Let a be the last element successfully added to $R.\mathsf{open}$
Let d be any element in $R.\mathsf{closed\text{-}successors}$
if $R.\mathsf{open\text{-}successors} = \emptyset$
  Let e be any successor of x not in R
else
  Let e be any element in $R.\mathsf{open\text{-}successors}$
return suborder induced by a, x, c, d, e

Subroutine 3.2 shows how to construct the forbidden induced suborder given an element x and the subtree R in which x precedes only some of the closed elements. It mirrors the proof of Lemma 3.2, and the naming of elements is consistent with the proof.

Three of the five elements, x, c, and d, are easily found in constant time. Finding e is bounded by the out-degree of x and is therefore $O(n)$. Finding a is not immediately a constant time operation, however, either by having each tree store a pointer to the last vertex added, or by always putting that element at the front of the open list, it

Subroutine 3.3: Partition

input : An element x whose successors have all been placed into subtrees
output: A set of subtrees that were partitioned by x

Let Q be an empty set of subtrees

for *each successor* y *of* x **do**
  if y *is closed* **then**
    Let $R \longleftarrow y.\mathsf{tree}$
    $R.\mathsf{closed} \longleftarrow R.\mathsf{closed}$ - $\{y\}$
    $R.\mathsf{closed\text{-}successors} \longleftarrow R.\mathsf{closed\text{-}successors} \bigcup \{y\}$
  else
    Let z be a successor of y Let $R \longleftarrow z.\mathsf{tree}$
    $R.\mathsf{open} \longleftarrow R.\mathsf{open}$ - $\{y\}$
    $R.\mathsf{open\text{-}successors} \longleftarrow R.\mathsf{open\text{-}successors} \bigcup \{y\}$
  $Q \longleftarrow Q \bigcup \{R\}$
return Q

can be implemented in constant time.

Since most of the relations between the elements are known, all that remains to complete the induced subgraph is to find the relation between c and d, and e and d. This requires looking through two adjacency lists and is therefore $O(n)$ also. Since this subroutine is run at most once, and is $O(n)$, it does not increase the asymptotic time complexity of the algorithm.

3.2.2 Partitioning Open and Closed Elements

For an $O(m + n)$ time bound, adding a new element to the model must take no more than time proportional to its out-neighborhood. In a standard partitioning algorithm(for example, lexicographic breadth-first search [20]), those elements that are not neighbors of the partitioning element do not change sets. In the first part of Tree Model Construction, those elements in $R.\mathsf{open}$ do not change sets, and therefore do not need to be updated; however, taking the union of open sets to make the new tree adds an extra complication: all elements are added to new sets. While the closed (and newly closed) elements can be updated in time proportional to the size of the

adjacency list, iterating through the list of open elements would cause this step to run in time proportional to the entire element set, not just the degree of the current vertex. Elements must keep accurate tree references so that only subtrees that have been partitioned are processed.

The solution follows from Proposition 3.2. If an open neighbor x is maximal, then x is alone in its subtree, and x's tree reference is valid because it has just been initialized. If x is not maximal, then all of x's successors are closed and therefore have correctly updated tree references. The first element in x's successor list references the correct subtree. Subroutine 3.3 shows the complete procedure.

This implementation allows for partitioning in time proportional to the degree of the vertex. Finding the host tree and performing the set operations are all constant time per edge. Furthermore, the number of trees that can be partitioned, the size of Q in Subroutine 3.3, is also bounded by the degree of the element. Each touched subtree causes only a constant amount of work, in assignments and set operations which take only constant time on the data structure, so adding an element to the model takes time proportional to the degree of the element. The time complexity for adding all of the elements is therefore $O(m)$.

3.2.3 Overall Time Complexity

Adding all of the elements to the tree model takes $O(m)$ time. In the case of a "no" answer, constructing the obstruction takes $O(n)$ time. Preprocessing the partial order to construct a list node for each element and to redirect adjacency list pointers takes $O(m + n)$ time. The final step of the algorithm, joining all the trees together, does constant work for each remaining subtree. An element, when added, creates at most one additional subtree and so the number of subtrees is bounded by the number of elements. This makes the joining phase of the algorithm $O(n)$, and so overall the complexity of the algorithm is $O(m + n)$. It follows from the recognition result that

the problem of constructing the tree model given a partial order is also $O(m+n)$.

3.2.4 Proving the Certificates

The certificate for a "no" answer is the instance of a forbidden induced suborder. To prove the certificate, one must scan the adjacency lists for the five elements of the suborder, taking $O(n)$ time.

The certificate of a "yes" answer is the tree model of the partial order. To prove that this model faithfully represents the partial order, perform a depth-first search on the tree. When retreating from the startpoint of an element, add that element to the list of current elements in the subtree succeeding the current vertex. When advancing from a branch vertex, leave that list at that branch vertex and then concatenate those lists when retreating. When retreating to a finishpoint, ensure that the out-neighbors of that element are exactly the elements in the list. If this check works for all elements, then the tree model is valid.

Using the same partitioning technique as the recognition algorithm, proving a "yes" certificate takes $O(m+n)$ time. Preprocessing the partial order elements into list nodes and adjacencies to pointers to list nodes takes $O(m+n)$ time. Since each element adds two vertices to the tree, the depth-first search itself is $O(n)$. Testing the adjacency list takes constant time for every edge, for $O(m)$ time.

CHAPTER 4

ROOTED, WEAK PATH-PRECEDENCE ORDERS

This chapter presents results on the partial orders defined by weak precedence in a rooted tree. These partial orders are shown to be exactly the intersection of a linear order and a forest of rooted-tree orders. This result leads directly to a class containment relationship. Weak path-precedence orders lie strictly between two- and three-dimensional partial orders.

This chapter also presents an algorithm for recognizing bipartite partial orders that are rooted, weak path-precedence orders. Recognition in the bipartite case essentially reduces to recognizing rooted, directed path graphs, and can therefore be done in $O(m + n)$ time [10]. Rooted, weak path-precedence orders are, therefore, a subclass of three-dimensional partial orders that can be recognized in polynomial time. Recognition of rooted, weak path-precedence orders that are not bipartite remains open.

4.1 Characterization

Characterizing rooted, weak path-precedence orders begins with a model transformation. Using the notion that a path "starts before" and "ends before," any tree-model can be transformed into one in which those aspects of precedence are essentially separated. To complete this separation, transform any tree model into a model where all of the startpoints precede all finishpoints and the first branch vertex in the tree. The key observation is that the finishpoints alone can confer the tree component of precedence, while startpoints can be reordered linearly as long as the result respects the relative ordering of startpoints along any root-to-leaf path.

For the moment, assume that each tree vertex contains at most one path endpoint. Once the general method is described, this assumption will be relaxed. A depth-first-

search ordering of the startpoints respects the relative ordering of startpoints on any root-to-leaf path, as it always processes ancestors before descendants. Since, by assumption, there is only one startpoint on any given vertex, there is no ambiguity of depth-first-search order. Once that ordering is computed, remove those startpoints from the tree. Next, construct a path of tree-vertices, one for each element, with the startpoints in depth-first-search order. Finally, connect the last tree-vertex in the path to the root of the tree with the finishpoints (the tree from the original model). If a path p preceded another path q in the original model, then p precedes q in the new model, because the finishpoints have the same position, and the depth-first-search ordering preserves relative ordering of the the startpoints along root-to-leaf paths. If p is incomparable to q in the original model, then there are two possible cases. If the finishpoint of p is incomparable to the finishpoint of q, then p and q are still incomparable because the position of finishpoints has not changed in the new model. Otherwise, the finishpoint of p precedes the finishpoint of q and the startpoint of q precedes that of p. Since the depth-first search preserves the ordering of the startpoints of p and q, so p and q are still incomparable.

To relax the constraint that tree-vertices contain only one endpoint, the first step is to protect against vertices that share both start and finishpoints. If elements share a startpoint but no finishpoints, then, in the new model, order the startpoints in inverse order of the finishpoints. This creates a containment relation and maintains their respective incomparability. In Figure 4.1, this approach can be seen with a, e, and d (it is assumed that the depth-first search considers branches in clockwise order). Similarly, if elements share a finishpoint, containment is maintained by ordering the elements inversely from their startpoints, shown in Figure 4.1 by elements a and b. If the elements share both endpoints, then the order of either the startpoints or finishpoints can be chosen arbitrarily, but the order of the other endpoints must be the opposite.

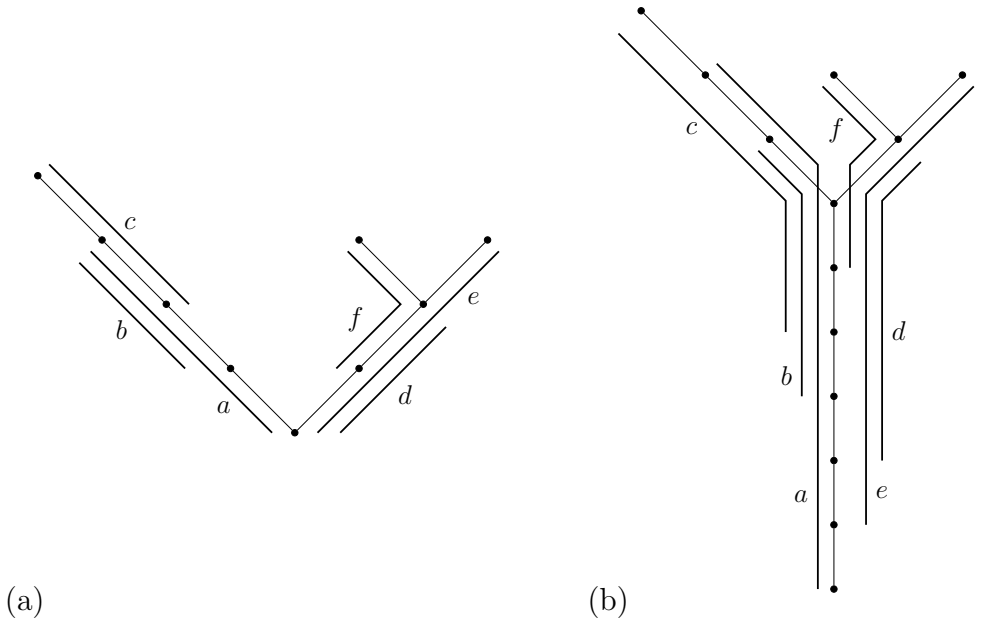


Figure 4.1: A weak precedence model (a) before and (b) after transformation

Lemma 4.1. *Any rooted, weak path-precedence model may be transformed into a model where all of the startpoints of elements precede any endpoints, and any branch vertices in the tree. Moreover, the new model has at most one endpoint per tree-vertex.*

Proof. This is a result of the transformation process described above. The transformation preserves relative order along root-to-leaf paths using depth-first search, so all precedence relations are maintained. Incomparable elements that do not share tree-vertices are still incomparable, as the tree structure is maintained in the endpoints, and path containment is maintained. Incomparable elements that do share endpoints remain incomparable by creating containment relations for the paths. □

This transformed model leads directly to the characterization result. For two-dimensional partial orders, a weak-precedence interval model can be constructed by using the two linear extensions of the realizer, arranging them one after the other, and making the instances of elements in the first linear extension the startpoints, and the second, the finishpoints. Rooted, weak path-precedence orders are similar; rather

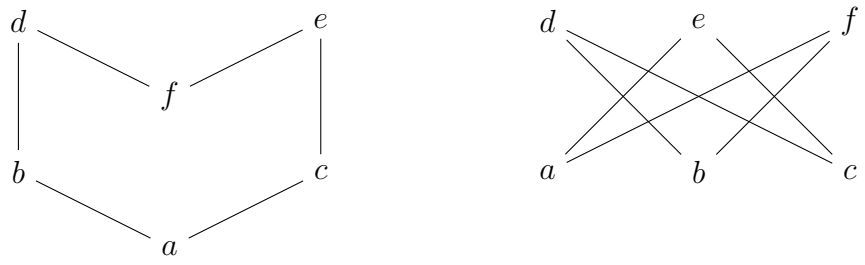
than a second linear extension, there is a forest of rooted trees.

Theorem 4.1. *A partial order is a rooted, weak path-precedence orders if and only if it is the intersection of a linear order and a forest of rooted-tree orders.*

Proof. Lemma 4.1 shows that any rooted, weak path-precedence model into can be transformed to a model where the startpoints precede the endpoints and any branches, and there is exactly one endpoint per tree vertex. The ordering on the startpoints is a linear order. The order on the finishpoints is a single rooted tree if there is a finishpoint of an element on or before the first branch vertex. Otherwise, it is a forest of rooted-tree orders. The linear order maintains the order of the startpoint and the forest of rooted-tree orders maintains the order of finishpoints. So, a rooted, weak path-precedence order is the intersection of this linear order and forest of rooted-tree orders.

If given a linear extension A and forest of rooted-tree orders T on the same ground set, a rooted, weak path-precedence model can be created by creating a two tree vertices, one for the element in A, one for T. Tree vertices for A will be assigned the startpoint of the elements, and the structure for the tree vertices will be a path. Tree vertices for T have the endpoints for the elements, and the structure matches the structure from the trees in T. Create a tree edge from each root in the T tree vertices to the tree vertex for the last element of A. A path p in this model starts before and ends before a path q precisely when the $p \prec q$ is in the intersection of A and T. □

This characterization of the class is often more convenient to manipulate than the model of paths in the tree. As such, the linear order A and the forest of rooted-tree orders T will be referred to as a standard model for this class. Given this representation, the class can be further characterized by containment relations with two- and three-dimensional partial orders.



(a) (b)

Figure 4.2: (a) the chevron, which is not two-dimensional but is a rooted, weak path-precedence order. (b) the S_3 which is three dimensional but not a rooted, weak path-precedence order.

Corollary 4.1. *Rooted weak path-precedence orders lie strictly between two- and three- dimensional partial orders.*

Proof. All two-dimensional partial orders are rooted, weak path-precedence orders because T may be a linear order, which is a rooted-tree order. An example of a partial order that is not two dimensional but which is a rooted, weak path-precedence order is the chevron, shown in Figure 4.2(a). To see this, f and a are incomparable, and therefore would be reversed in the two linear extensions. As common out-neighbors, d and e would follow both in each. However, b and c must also be reversed with f in the two lists, and so there is no arrangement of two lists where b precedes only d and c precedes only e. The chevron is a rooted, weak path-precedence order, with $A = abcfde$ and T has f as the root, preceding a, which has two child branches, bd and ce.

Rooted, weak path-precedence orders are contained within three-dimensional partial orders because forests of rooted trees are two-dimensional, as proven by Wolk [46], and so the combination of A and a two-dimensional realizer for T is a realizer for the partial order. The containment is strict. For example, the standard example S_3, shown in Figure 4.2(b), is not a rooted, weak path-precedence order. Since a, b and c each share one out-neighbor with the other two, they must fall on the same root-

to-leaf path in T. Assume, without loss of generality, they are ordered abc in A. In T they must then be are ordered cba. As the source elements are symmetrical the particular labeling makes no difference. Immediately, c cannot precede f, so f must precede c in A. Also, d cannot succeed a, so it must branch. A therefore begins with $abfc$ and ends with d and e in some order. From here, there is no place to proceed, as e must follow a in T, however, that would force e to succeed b in both A and T. □

The number of members in the class is immediate from the containment result, as each element can store three $O(\log n)$ positions in the realizer, and completely store the partial order.

Theorem 4.2. *There are $2^{O(n \log n)}$ rooted, weak path-precedence orders on n elements.*

Proof. Follows directly from containment by three-dimensional partial orders. It can also be seen as a result of the tree-model representation. □

4.2 Recognition in the Bipartite Case

This section shows that if a partial order is bipartite, the question of whether it is a rooted, weak path-precedence order can be answered in linear time. The approach is to reduce the problem to one using the same machinery as recognition of rooted, directed path graphs.

Lemma 4.2. *A bipartite partial order P is a rooted, weak path-precedence order if and only if there exists a rooted tree R such that the vertices of tree correspond to sources in P and the in-neighbors of any sink are a directed path in R.*

Proof. Assume such an R exists, a model A, T for P can be built as follows. First, to start constructing A, the sources are ordered in reverse order from a depth-first search. This ensures all sources are incomparable. For each sink, the maximal and minimal sources of its neighbor path in the tree are noted. Then to create T, for a

sink k, add a tree vertex for k as a child to its maximal source. To add k to A, place it directly following its minimal source. The minimal source is k's maximal neighbor in A due to the reverse ordering. k is incomparable to elements after its maximal source in the tree because k is its own leaf in T. k is incomparable to elements preceding its minimal source, because k precedes them in A. Therefore this model respects the partial order.

Now, for a model A, T for P, create a rooted tree R by removing all of the sinks from T. Suppose in R, that there exists a consecutive path a, b, c, such that a and c are in-neighbors of a sink n but b is not. Since R contains only sources, a, b, and c are incomparable. In any A, the relative order of the three must be c, b, a so they are incomparable in the model. But n must be after a in A, and if there is no R without b between a and c, n would be after c, therefore n succeeds b in both orders. As b is incomparable to n, this is a contradiction. □

A graph is a rooted, directed path graph if and only if the maximal cliques of the graph can be arranged in a rooted tree such that the maximal cliques incident on a vertex are directed paths in a tree. Deitz *et al.* [10] showed that these graphs can be recognized in $O(m + n)$ time using a structure called a PQR-tree[1]. The algorithm begins by computing for each vertex the set of maximal cliques incident on that vertex. It then adds these constraint sets incrementally to the PQR-tree, until all are added, or it finds there is no conforming tree structure. In the case of bipartite, rooted, weak path-precedence orders, Lemma 4.2 implies that the in-neighbors of a sink are a constraint set for a PQR-tree. If there exists a rooted-tree configuration for those constraint sets, then there is a weak path-precedence model of the partial order.

[1]Pay close attention to the specific reference here. The usefulness of Booth and Leuker's PQ-tree is such that there are many generalizations, and the name PQR is sufficiently attractive that there are at least 3 different authors who have developed a structure with that name.

Theorem 4.3. *Bipartite, rooted, weak path-precedence orders are recognizable in* $O(m+n)$ *time.*

Proof. By Lemma 4.2, this reduces to finding whether there is a rooted-tree arrangement of sources in the partial order. Using the PQR-tree of Deitz *et al.* [10], this can be done in time proportional to the number of constraints plus the size of the constraint sets, which is $O(m+n)$. □

Corollary 4.2. *Construction of the model for bipartite, rooted, weak path-precedence orders is* $O(m+n)$.

Proof. Since the recognition is by construction of the tree of source elements, once the tree is built, the procedure used in the proof of Lemma 4.2 can be implemented to construct A and T in $O(m+n)$ time. □

CHAPTER 5

OPTIMIZATION PROBLEMS WITH ROOTED-TREE MODELS

This chapter looks at optimization problems on the classes of partial orders defined by the strict and weak path-precedence in rooted trees, specifically finding the size of the maximum clique or independent set in these orders. As discussed in Chapter 1, there are already efficient algorithms for these problems given a comparability graph. The maximum clique in a comparability graph can be found in $O(m + n)$ time, and the maximum independent set reduces to the time complexity of maximum flow, $O(m\sqrt{n})$. This chapter presents algorithms for these problems given the tree models as input. For strict precedence, these problems can both be solved in $O(n)$ time given the model as input. Given a rooted, weak path-precedence model, the size of the largest clique can be computed in $O(n \log n)$ time, and the size of the largest independent set can be computed in $O(n^2)$ time

For the algorithmic analysis, there are assumptions made about the input. For strict precedence models, it is assumed that the rooted tree contains $O(n)$ vertices, where n is the number of elements in the partial order. In Chapter 3, the algorithm which recognizes rooted, strict path-precedence orders builds such a model for any rooted, strict path-precedence order, so one does exist.

Chapter 4 discussed a standard form for rooted, weak path-precedence tree models as the intersection of a linear order A and a rooted tree T of elements. If a tree model is given in path form and the tree is of size $O(n)$, then this representation can be computed in $O(n)$ time using depth-first search. For simplicity, T is assumed to be a single rooted tree. If not, a special element r^* can be added to T to serve as the root. For computing the size of the largest clique, r^* is put at the end of A, so it is incomparable to every element. For the independent set problem, r^* is at the

beginning of A so it precedes every element. Therefore, the single-tree assumption does not change the time complexity of the algorithms.

For space-inefficient models with more tree vertices, the n in the time bounds refers to the size of the tree, but no longer to the number of elements in the partial order.

5.1 Maximum Clique

Chapter 1 showed an $O(n)$ maximum clique algorithm for interval orders given the model, and an $O(n \log n)$ algorithm for clique in two-dimensional partial orders, given the realizer as input. When generalizing the interval model to a rooted tree, it is still a necessary condition that the endpoints for the paths in a clique share a root-to-leaf path in the tree. This immediately gives algorithms for rooted, strict and weak path-precedence orders which increase the running time by a factor of n. For this approach, the algorithms for the smaller classes are run once on each root-to-leaf path, of which there are fewer than n. The algorithms presented in this section use simple techniques to avoid repeating work, and keep the same asymptotic time complexity as the smaller class.

5.1.1 Strict Precedence

Computing the size of the maximum clique for rooted, strict path-precedence orders takes only a very small change to the algorithm for interval orders. When computing the maximum clique for interval orders, each endpoint is processed in order, using the values of the preceding endpoints to compute size of the largest clique preceding the current endpoint. With the tree model, a tree vertex may have more than one successor, so a depth-first order is used. Tree vertices are, therefore, processed without repeating work.

Theorem 5.1. *Algorithm 5.1 correctly computes the size of the maximum clique for rooted, strict path precedence orders in $O(n)$ time.*

Algorithm 5.1: Rooted, Strict Path-Prec Order: Maximum Clique

input : v current tree vertex in T, and previous_max which is the size of the maximum clique ending at or before the parent of v.
output: the size of the largest clique ending on after v.

```
v.max ⟵ previous_max
current_max ⟵ v.max
for each finishpoint of x at v
    m ⟵ (startpoint of x).max + 1
    if m > current_max
        current_max ⟵ m
if v.next = ∅
    return current_max
else
    max_child ⟵ −1
    for each child c of v
        m ⟵ Maximum Clique(c, current_max)
        if m > max_child
            max_child ⟵ m
    return max_child
```

Proof. Assume, for the sake of contradiction, that the algorithm does not compute the maximum clique. Since Algorithm 5.1 computes the maximum clique for the path from the root to each tree vertex, there is some tree vertex for which the maximum clique path on the path from the root to that vertex is computed incorrectly. Let v be the first such vertex.

The last element in the maximum clique on the path from the root to v either ends at v or before v. If it ends before v, then the maximum clique was computed by the parent of v. Since v is the first vertex to compute the maximum clique incorrectly, the last element in the maximum clique must not end before v. If the last element in the maximum clique ends at v, then the size of the largest clique is the maximum value from the startpoints of the elements ending at v plus one. Since a single element can only increase the size of a clique by one, and all of the previous values are correct, the last element in the maximum clique ending at v cannot end at v. Since the maximum

clique on the path from the root to v can end neither before v nor on v, there is a contradiction. There can be no such v where a first mistake is made.

The algorithm does a constant amount of work for each tree vertex and each path endpoint, which gives an $O(n)$ running time. □

5.1.2 Weak Precedence

To compute the size of the maximum clique for rooted, weak path-precedence orders, the algorithm for two-dimensional partial orders is generalized. For two-dimensional partial orders, order in the first list is used to establish a notion of "increasing." Then, the algorithm processes the elements in order of the second list. An increasing subsequence corresponds to elements that each start and end before the next. Such a set of elements is a clique under weak precedence. The maximum such increasing subsequence is therefore the maximum clique. The details of how the algorithm works are given in Chapter 1, and the running time is $O(n \log n)$.

The existence of this algorithm quickly gives polynomial algorithms for clique in rooted, weak path-precedence orders. While there are branches in the tree, each clique must still fall on a root-to-leaf path in the tree. There are at most $O(n)$ leaves in the tree, and therefore at most $O(n)$ root-to-leaf paths. Running the two-dimensional algorithm on each of those paths would give an $O(n^2 \log n)$ algorithm. To avoid all of the repeated work, the array containing the partial solution can be copied at branch vertices. For a maximum clique of size k, this reduces the time complexity to $O(kn + n \log n)$. Since k is $O(n)$, for large cliques this is $O(n^2)$.

The bottleneck above is the copying of the array. This problem is avoided by slightly changing the data structure representing the partial solution. Instead of storing an array with the positions in the first list, an array of stacks is stored which holds positions in A. Then, when retreating during a depth-first search of the tree from a leaf, the entry made is popped off the stack, leaving the stack in the same state

Algorithm 5.2: Rooted, Weak Path-Prec. Order: Maximum Clique

input : v current tree vertex in T. Best an array of stacks, in which the top of the stack at index i stores the value of the last element in an increasing subsequence In A of length i, such that the value is minimum among all last elements in increasing subsequences in A of length i ending on the path from the root of the tree to v.

output: the size of the largest clique ending on or after v.

```
i ⟵ BinarySearchofTops(Best, v.value)
Best[i + 1].push(v.value)

current_max ⟵ Best.length
for each child c of v
    m ⟵ Maximum Clique(c,Best)
    if m > current_max
        current_max ⟵ m

Best[i + 1].pop()
return current_max
```

as it was given. Each element is pushed and popped only once, so stack operations add only a constant amount of work per element, so the overall running time remains the same as the two-dimensional algorithm, $O(n \log n)$.

The new algorithm works as follows. At each new element v, a binary search on the array is performed, using the position of v in A as the key. The values for the array positions are the values on top of the stacks at those positions. Then the value of v is pushed on the appropriate stack. If the value extends the longest clique, this could push on an empty stack which extends the length of Best. Each child is called recursively, and size of the largest clique found is returned.

Theorem 5.2. *Given the model in standard form, the size of the maximum clique for rooted, weak path-precedence orders is computed by Algorithm 5.2 in $O(n \log n)$ time.*

Proof. The correctness of the algorithm follows from two points. First, the maximum clique for two-dimensional partial orders is correctly computed using the longest increasing subsequence technique [20]. Second, weak precedence implies that cliques

must share a root-to-leaf path. This second point is trivially true given the definition of weak precedence.

The stack technique the algorithm employs does not require copies of the array to be made, and does so simply replacing a write operation with a push and a pop. Altogether, this computes the maximum clique on every root-to-leaf path, and does so in $O(\log n)$ time per tree vertex for the binary search, $O(n \log n)$ overall. □

5.1.3 Application: Longest Matching Substring

The weak-precedence model can be applied to the problem of finding the best match of a query. The problem of finding the longest common substring of two strings first introduced by Wagner and Fischer [44]. There have been several improvements, and there are currently several efficient algorithms with incomparable running times given in [2]. Of particular interest is an algorithm due to Hunt and Szymanski [24] with a running time of $O(r + n) \log n$, r is the number of ordered pairs of matching symbols, and n is the length of the input strings. In this section, it is shown that weak interval precedence models the longest common substring problem such that finding the largest clique finds the longest common substring. Then, using a rooted, weak path-precedence model, this result can be extended to finding the longest matching substring in a family of strings with the same time complexity as Hunt and Szymanski [24].

Given two input strings, the weak-precedence model is constructed in the following way. When every symbol appears only once in a string, constructing the interval model is simple. Order one string after the other. These positions are the possible endpoints. For every symbol which appears in both strings, create an interval which starts from the occurrence in the first string and ends at the occurrence in the second. Figure 5.1 shows an example of two strings and the resulting interval model.

When there are duplicate symbols in the strings, adjustments must be made to

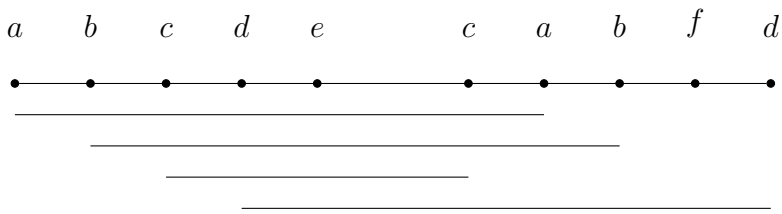


Figure 5.1: An example of an interval model created from strings without duplicate symbols.

ensure that no instance of a symbol from one string is matched with more than one position in the other string. Begin by ordering one string after the other, but instead of a single endpoint for each symbol, each symbol instance is associated with a number of endpoints, one for each matching symbol in the other string. When creating the intervals, pairs of duplicated symbols assign their endpoints to symbols in the other tree in reverse order of their appearance. This procedure, called *reverse-order matching*, causes the interval to have a containment relation with all intervals which use the same symbol. Thus, using the same symbol instance twice in the same chain is forbidden. Figure 5.2 shows an example of an interval model corresponding to two strings with duplicate symbols.

Theorem 5.3. *Given an interval model constructed from two strings using reverse-order matching, there exists a chain of length n in the weak-precedence ordering if and only if there is a common subsequence of length n in the strings.*

Proof. First, note that the reverse-order matching creates a containment relation between all intervals which start or end at the same symbol, so the same symbol cannot be the end-point of two intervals in a chain. Also, note that all intervals start in the first string and end in the second. For a given chain, the start points of the intervals map to strictly-increasing (by weak precedence) and unique (by reverse-order matching) symbol positions in the first string. The end points map similarly to positions in the second string. Since intervals start and end at matching symbols, this is a common subsequence. Inversely, a common subsequence leads to a set of

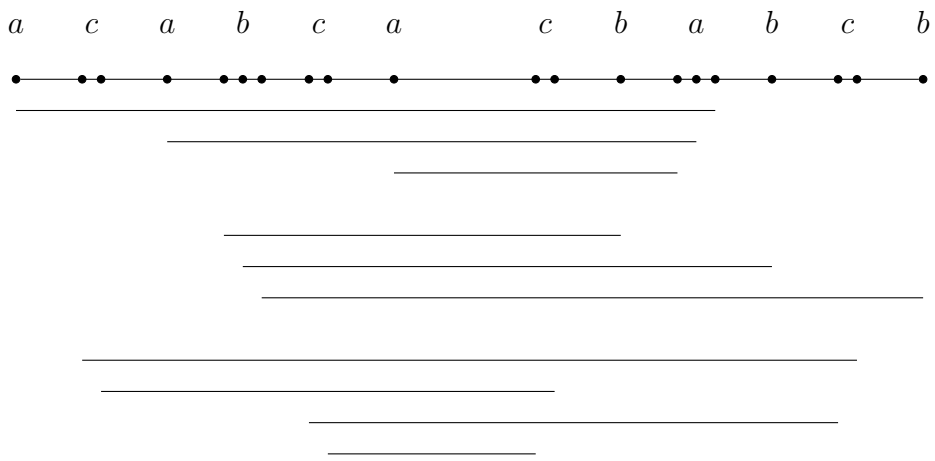


Figure 5.2: Example of an interval model created from strings with duplicate symbols. a has duplicates in the first string but not the second, b has duplicates in the second but not the first, and c has duplicates in both strings.

intervals totally ordered by weak-precedence, which is the definition of a chain. □

The interval model can be easily extended to trees. The tree models searching for the longest common subsequence in a family of strings. The tree model presented here avoids duplicating work by arranging the family of strings by common prefix. Each symbol then has a corresponding number of endpoints to the highest number of duplicates in any of the family of strings. The endpoints of the intervals are still governed by the reverse-order matching procedure given in the previous section. An example of such a model is given in Figure 5.3.

Theorem 5.4. *In a rooted, weak path-precedence model constructed from a string and a family of strings using reverse-order matching, there exists a clique of size n in the weak-precedence ordering if and only if there is common subsequence of length n between the string and one member of the family of strings.*

Proof. By the definition of weak precedence, cliques share a root-to-leaf path. Also, by the construction of the tree model, a root-to-leaf path represents a single member of the family of strings. By the same argument as in Theorem 5.3 it follows that there

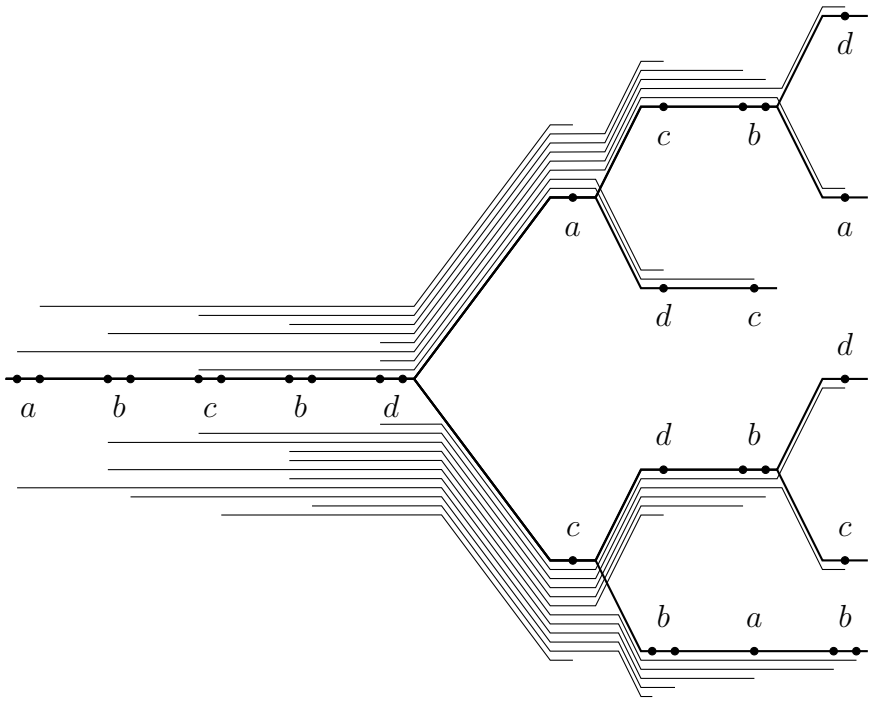


Figure 5.3: Example of a tree model created from matching the string *abcbd* with the family $\{acbd, acba, adc, cdbd, cdbc, cbab\}$. Notice that no two intervals share an endpoint in the right part of the tree, and if two intervals share an endpoint in the left half, the right endpoints are on different branches. The longest match is *acbd* along the top branch.

is a clique of size n in the weak-precedence model if and only if there is a common subsequence of length n in all strings that share that part of the root-to-leaf path. □

5.2 Maximum Independent Set

Independent sets in rooted path-precedence models are not constrained to be along a single root-to-leaf path. Elements in a tree may be non-adjacent either by containment along a single path, or by lying along different branches. The algorithms presented for computing the size of the maximum independent set compute optimal solutions for the children of a tree vertex, then combine these solutions working toward the root of the tree.

5.2.1 Strict Precedence

The algorithm for computing the size of the maximum independent set for strict precedence begins at the leaves of the tree, and proceeds as the interval order algorithm until reaching a branch vertex. At a branch vertex, the sizes of the independent sets of the children are summed, and that sum is compared to the number of paths intersecting at the vertex. This method yields the largest independent set.

Lemma 5.1. *The size of the maximum independent set of a tree model rooted at a branch vertex r is either the sum of the maximum independent sets of the children of r or the number of paths intersecting at r, whichever is larger.*

Proof. Consider the largest independent set among elements with finishpoints in the subtrees succeeding (and not including) r. There exists an independent set with size k, equal to the sum of the sizes of the maximum independent sets in each of the subtrees. However, there cannot be a larger independent set, as that would mean one of the subtrees had an element incomparable to its entire maximum independent set, which is a contradiction.

Algorithm 5.3: Rooted, Strict Path-Prec Order: Maximum Independent Set

input : v current tree vertex in T
output: a pair the size of the maximum independent set succeeding v, and the set of elements which have finishpoints at or succeeding v and startpoints preceding v.

```
max_IS ⟵ 0
open_paths ⟵ 0
for each child c of v
    ⟨m, o⟩ ⟵ MaxIndSet(c)
    max_IS ⟵ max_IS + m
    open_paths ⟵ open_paths + o
open_paths ⟵ open_paths +|{x|x finishes at v}|
if open_paths > max_IS
  max_IS ⟵ open_paths
open_paths ⟵ open_paths −|{x|x starts at v}|
return ⟨ max_IS , open_paths ⟩
```

It follows that, if there is a larger independent set when r is added, then at least one element in the independent set ends at r. If any element with a finishpoint of r is part of the independent set, then no element with a startpoint in any of the child subtrees is included in that independent set. In other words, if the largest independent set contains any element with a finishpoint at r then it consists entirely of elements that intersect at r. □

This observation allows generalization of the independent set algorithm for interval orders as shown in Algorithm 5.3.

Theorem 5.5. *The size of the maximum independent set for rooted, strict path-precedence orders, given the tree model, is computed by Algorithm 5.3 in $O(n)$ time.*

Proof. The correctness of the algorithm follows from Lemma 5.1, which shows that, at any tree vertex v, the maximum independent set of the elements finishing on or before v is either the sum of the maximum independent sets of the children or the

number of paths intersecting at v. This allows the incremental approach to computing the size of independent sets with successively larger rooted subtrees.

Algorithm 5.3 essentially performs a depth-first search on the tree, with added processing time for the start and finishpoints of each element in the partial order. Since the size of the tree is $O(n)$, this gives an overall time complexity of $O(n)$. □

5.2.2 Weak Precedence

The final algorithm for this chapter finds the size of the largest independent set in a rooted, weak path-precedence order. As with independent set in the strict precedence case, the approach is to generalize the algorithm for finding the largest independent set for the smaller class, starting at the leaves of the tree, and combining the answers to sub-problems at branch vertices. In this case, the smaller class is two-dimensional partial orders. The model is again assumed to be given in the standard form, a single total order A and a rooted-tree order T on the elements.

The algorithm for independent set for two-dimensional partial orders, based on the longest increasing subsequence algorithm, uses an array called Best. Best stores at each index i, the last in an independent set of size i such that the value is minimum in the first total order, among all independent sets of size i, which are induced by the elements that succeed the current element in the second total order. The order of processing elements in the second total order is reversed from the clique algorithm. In a two-dimensional partial order, the relationship of the paths in a clique would be containment, so "last" simply means the element contained by all the others. Once there are branches, the element at i need only be contained by all other elements which finish on the same path. For a given independent set I, the last element is later in A then all other elements in I that are comparable in T, and earliest in A among elements in I that are incomparable in T. The Best array at a branch vertex

is computed by merging, in the same sense as merge sort, the Best arrays of child branches.

Lemma 5.2. *The Best array at a branch element r in T before r is added is given by merging the Best arrays of the children of r in T.*

Proof. When considering the largest independent set in the elements succeeding r, its size can be at most the sum of the independent sets from each child of r, otherwise more elements come from one branch than its largest independent set. So the Best array at r must be at most the size of the sum of the Best arrays of the children.

The invariant of the Best array at r is it holds the last element in an independent set of size i that precedes, in A, the last element in all other independent sets of size i consisting of elements succeeding r in T. This allows a greedy choice, as any element which later extends any independent set of size i, must also extend the independent set recorded in Best.

At a branch vertex, when looking for the independent set of size 1 that ends first in A, that element must be the first element in the Best array of one of the children. The second element may then be the first element from the other child branches, or the second element from Best array which held the first element. This continues as long as there are elements, and this is precisely the standard merge operation on the child Best arrays. Since the length of the new array is the sum of the lengths of the child Best arrays, the length of the Best array remains the length of the largest independent set. □

Since this merge gives a valid Best array, at r, the algorithm can recursively compute the Best arrays of the children, merge those arrays, and add r. This computes the size of the largest independent set in the subtree rooted at r. If r is the root of T, this is the size of the largest independent set in the partial order.

Algorithm 5.4: Rooted Weak Path-Prec Order: Maximum Independent Set

input : v, the current tree vertex in T.
output: The Best array after adding v. The length of the Best array is the size of the maximum independent set.

```
Best ⟵ ∅
for each child c of v
    Best_c ⟵ MaxIndSet(c)
    Best ⟵ Merge(Best,Best_c)
i ⟵ BinarySearch(Best, v.value)
Best[i + 1] ⟵ v.value
return Best
```

Theorem 5.6. *The Algorithm 5.4 computes the size of the maximum independent set in a rooted, weak path-precedence order in $O(n^2)$ time.*

Proof. The correctness of the algorithm is implied by Lemma 5.2. Let us assume the size of the maximum independent set is size k. At each element in the tree order, there is a binary search and addition which takes $\log k$. Each edge in the tree may lead to a merge operation which takes k time. So Algorithm 5.4 is bounded by $O(k(n-1) + n \log k)$. Since k is $O(n)$, for large k the first term dominates and the algorithm is $O(n^2)$. □

CHAPTER 6

STRICT PATH-PRECEDENCE IN A GENERAL TREE

This chapter discusses the class of partial orders defined by strict precedence of paths in a general directed tree, *i.e.* a tree that does not necessarily have a root. These partial orders are called strict path-precedence orders. A forbidden induced-suborder characterization of the class and a polynomial-time, in this case $O(n^2)$, certifying, recognition algorithm are presented. While the classes arising from rooted and general tree models are intuitively similar, there are a key differences. There are two classes of forbidden orders, a finite class of six-element partial orders, and the infinite class of crowns. The finite class of forbidden suborders have an additional element. Essentially, this extra element restricts the possible placement of its out-neighbors to rooted subtree. The increase in the number of forbidden orders results from combinations including two additional "optional edges" with the extra element. The crowns force cycles in the tree model, and so are disallowed. The increase in the time complexity of the algorithm comes from searching the entire subtree for each new element added. In the rooted case, the structure of the model allowed for checking only the root of each subtree. With no single root for a subtree, an element can "branch in" at any single point in the tree to precede only a subset of the neighbors of previously added elements. This search increases the time complexity of the certifying recognition algorithm to $O(n^2)$.

The chapter begins by showing a set of orders which are not strict path-precedence orders. Next, a construction algorithm is outlined which shows that the absence of this set is sufficient to build a tree model. Finally, techniques are given for implementing the algorithm in $O(n^2)$ time.

6.1 Notation

Some additional notation is required to aid in referencing specific tree vertices. As before, trees (or subtrees) will be referenced using uppercase script, e.g. T. Elements in the partial order will be referred to using lowercase script, e.g. x. It will be convenient to refer to the start and finishpoints of the paths in the tree representing certain elements. The startpoint of an element x in the tree is subscript with an s, *e.g.* x_s. Similarly, the finishpoint of x is x_f. The least common descendant and the greatest common ancestor of two tree vertices also have special notation. In these trees, these elements need not exist, but the existence of such a tree vertex can be forced by comparabilities in the partial order. The lattice symbols are used for these structures, subscripted by the tree vertex which forces the existence. For example, if a and b both precede c, then in the tree there is an element $(a_f \vee_{c_s} b_f)$. Furthermore, that element must be less than or equal to c_s in the tree for the precedence relation to hold, $(a_f \vee_{c_s} b_f) \leq_T c_s$. Similarly, if y and z share a common ancestor x, then in the tree, their startpoints have a greatest lower bound greater than or equal to x_f, that is $(y_s \wedge_{x_f} z_s) \geq_T x_f$.

6.2 Strict Path-Precedence Orders

While the complete tree model may not be rooted, the neighborhoods of the elements are restricted to be in a rooted tree.

Proposition 6.1. *All of the descendants of an element x lie in a rooted subtree. Moreover x_f is the root of such a subtree.*

Proof. This follows immediately from the definition of precedence and the definition of root. □

This property also holds for the intersection of out-neighborhoods.

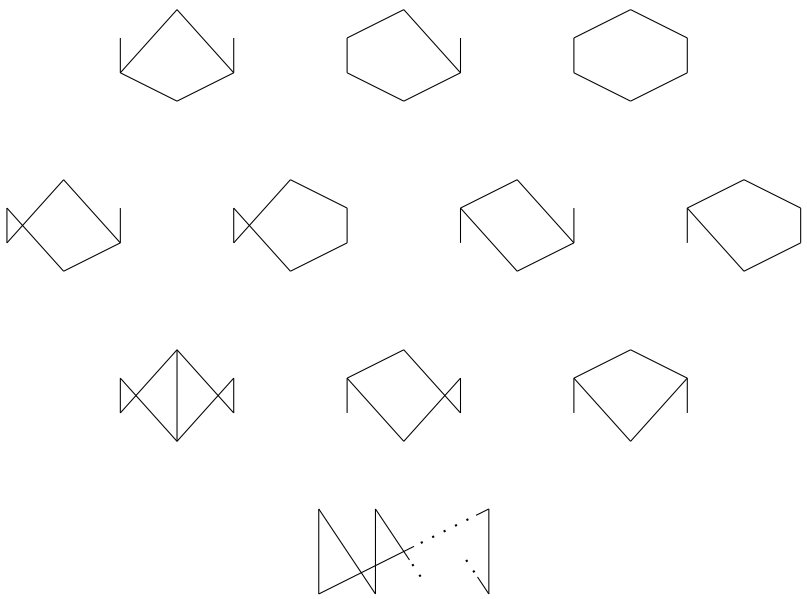


Figure 6.1: Diagrams for the complete list of forbidden induced suborders

Proposition 6.2. *For any two elements x and y with a common out-neighbor n, $(x_f \vee_{n_s} y_f) \leq_T n_s$. exists. Moreover, the startpoints of all common out-neighbors are greater than or equal to $(x_f \vee_{n_s} y_f)$.*

Proof. For one neighbor, there exists a path from x_f to n_s, and y_f to n_s, $(x_f \vee_{n_s} y_f) \leq_T n_s$, with equality holding when the two paths share no vertices (*i.e.* n_s must be a converging branch point). Since the model is a tree, there can be only one such least common descendant. Two minimal common descendants would imply a cycle.

□

Note that the symmetrical properties, for in-neighbors and in-rooted trees also apply, with similar proofs. These observations are sufficient to show that there are orders which are not strict path-precedence orders.

Lemma 6.1. *None of the partial orders whose diagrams are pictured in Figure 6.1 are strict path-precedence orders.*

Proof. Consider the cases as divided and labeled in Figure 6.2. For the first case, by Proposition 6.2, there must exist a vertex $(a_f \vee_{b_s} w_f)$ and a vertex $(a_f \vee_{b_s} x_f)$. These

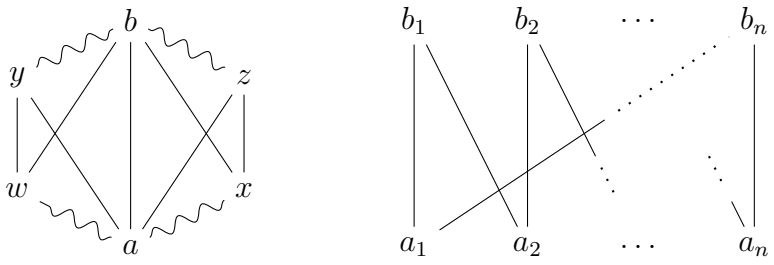
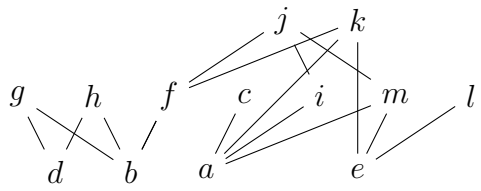


Figure 6.2: An abbreviated, labeled list of forbidden induced suborders. The wavy edges may or may not be present, giving rise to all 10 orders in the first class.

two tree vertices cannot be comparable or either x or w would contain the other's neighborhood. But as both follow a_f and precede b_s if they are not comparable there is a cycle in the underlying tree. The second case is the n-crown, where $n \geq 3$. Again by Proposition 6.2, for each successive pair a_i and $a_{(i+1)\%n}$ there is a vertex $(a_i \vee_{b_i} a_{(i+1)\%n})$ in the tree, including $(a_n \vee_{b_n} a_1)$. But this is, again, a cycle in the underlying tree as there would be two paths from a_1 to a_n. □

6.3 Constructing the tree model

The tree model is constructed incrementally, adding elements from the partial order in order of increasing out-neighborhood. Again, the term *open* denotes an element that has not yet had its startpoint in the tree fixed, and *closed* to denote those elements with both endpoints fixed. When adding an element x to a general tree model, within each subtree with any out-neighbors of x, there must be a tree vertex which precedes the startpoints of all closed out-neighbors of x and the finishpoints of all open out-neighbors of x in that subtree, but does not precede any of the closed non-neighbors of x in that subtree. The discussion of finding such a tree vertex efficiently, if it exists, is postponed until Section 6.4. This section presents a method to construct a new valid tree model with x added, if such a tree vertex can be found. Then it is proved that if no such tree vertex exists, then there exists a suborder from Figure 6.1. This will complete the characterization by forbidden induced suborder.



$jkghlcimfdeba$

Figure 6.3: A partial order to illustrate the procedure for building a tree model, and an ordering of vertices to build the model. The elements are ordered by increasing out-degree.

6.3.1 Adding an Element

As before, the tree model for a partial order is created incrementally by adding elements in order of increasing out-degree. For this algorithm, elements with identical out-neighborhoods are added together, at the same tree-vertex.

When adding an element x, first create a tree vertex for x's finishpoint, x_f. Then, for each existing subtree in which x has out-neighbors, a tree vertex is found which may be the out-neighbor of x_f such that the precedence of paths is maintained. If there is more than one tree vertex that precedes all of the neighbors but no non-neighbors, choose the minimum element in that set. The simplest case is adding a maximal element, as these elements have no out-neighbors. For a maximal element, add a single new tree vertex to the subtree, and set the finishpoint of the maximal element at that element. Figure 6.4(a) shows all of the maximal elements added to the tree model for the example partial order shown in Figure 6.3.

When the element x being added is not maximal, add a second tree vertex, directly succeeding x_f, called a join vertex. The join vertex branches to each subtree in which x has neighbors, and open neighbors from each of those subtrees are closed on the joining vertex[1]. Figure 6.4(b) shows the addition of two elements which only have

[1]For complete consistency, one could add a join vertex for maximal elements also, but these join vertices would never contain any endpoints. When an element is not maximal, even if there are no startpoints on it, the join vertex still represents a unique in-neighborhood, and therefore is not redundant. It may play a role in the refinement procedure introduced shortly.

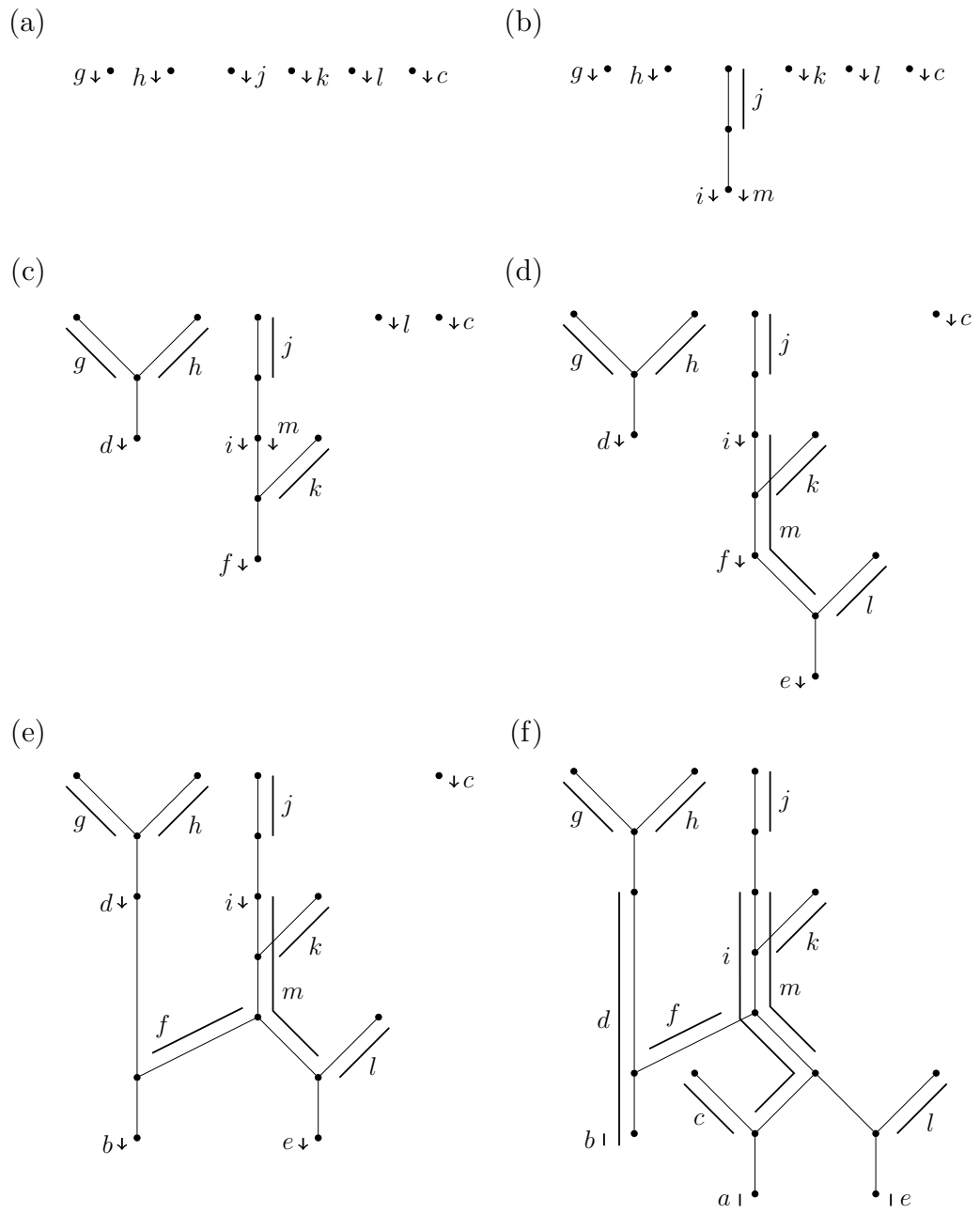


Figure 6.4: The process of constructing the tree model for the partial order in Figure 6.3: (a) shows the maximal elements being added; (b) shows elements i and m being added; (c) shows d and f being added; (d) shows after e; (e) after b; and (f) shows after a is added, and the remaining open elements are closed at a preceding minimal tree element.

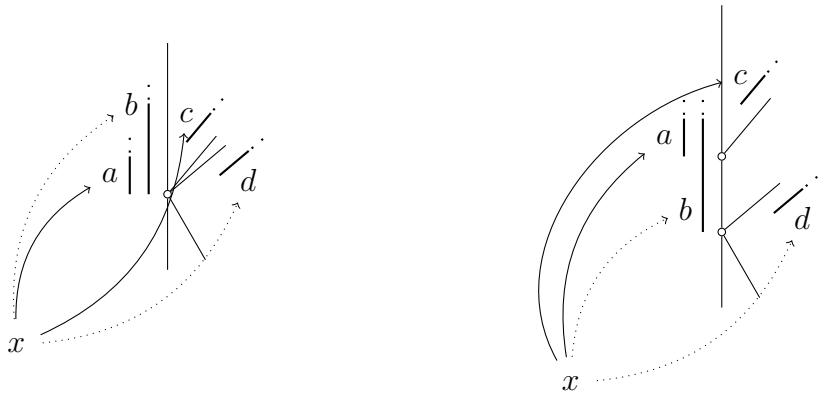


Figure 6.5: The element x refines a tree vertex based in its out-neighborhood.

out-neighbors (in this case only one out-neighbor) in a single subtree. The paths for elements i and m close the path for element j. Figure 6.4(b) is also an example of elements with the same out-neighborhoods, i and m being added at the same tree vertex. Figure 6.4(c) and (d) show the addition of elements (d,f, and e) which join subtrees together.

When the tree vertex preceding all out-neighbors of x is not minimal element in the subtree, it leads to a converging branch vertex in the tree. In Figure 6.4(e), b precedes a subset of e's closed neighbors in the subtree (j and k but not m or l), and so there is a converging branch. In this example, f is closed on the join vertex of b which directly succeeds b_f.

Ordering only by out-degree does not guarantee that an element is added before an element that contains a superset of its neighbors within a particular subtree. It may be necessary to refine the structure of a tree during the construction. If both neighbors and non-neighbors have start-points at the same tree vertex, the tree-vertex is split, with the neighbors placed at the greater tree vertex, and the non-neighbors placed at the lesser tree vertex. If the tree vertex is a diverging branch point, then there may be branches to move. Any such branch must contain only neighbors, or only non-neighbors, a branch with neighbors is placed at the greater tree vertex, and

a branch with non-neighbors at the lesser. Any converging branch must precede both points, and so the converging branch vertex must converge at the lesser tree vertex. Figure 6.5 shows a vertex x refining a tree vertex based on its neighborhood. In Figure 6.4(f), adding a causes the join vertex of e to be refined, because a precedes m but not l.

Before continuing with our proof that absence of the partial orders in Figure 6.1 is sufficient, there are some basic observations about the tree model resulting from the construction algorithm outlined above.

Proposition 6.3. *After any element is added to the model during construction, paths for elements in the model with the same out-neighborhood finish on the same tree vertex, and paths for elements in the model with the same in-neighborhood start on the same tree vertex.*

Proof. Since elements are added with the same out-neighborhood at the same tree vertex, the first part is trivially true. A path for an element is closed on the join vertex of the first in-neighbor to be added. This is only changed by refinement, and then only if there is a vertex which distinguishes the set, but then the elements no longer have the same in-neighborhood. □

Immediately by Proposition 6.3, if two elements have comparable but not equal startpoints, the greater startpoint must have an additional in-neighbor. Similarly, for two comparable but not equal finishpoints, the lesser must have an additional out-neighbor. The following propositions concern endpoints which precede or succeed common branches.

Proposition 6.4. *If the startpoints of two elements are incomparable but succeed a common diverging branch vertex, then each element has a private in-neighbor (i.e. an in-neighbor which is not an in-neighbor of the other element).*

Proof. Consider the element a, whose join vertex is the diverging branch vertex. Since each path was closed before the branch, each path was closed while the paths were in different trees. Since the same element cannot have paths in different trees, the paths were closed by different elements. □

Proposition 6.5. *If the finishpoints of two elements are incomparable but precede a common converging branch vertex, then each element has a private out-neighbor.*

Proof. Consider the process of creating a converging branch point. The element "branching in" has either a private neighbor closed on its join vertex, or a diverging branch on its join vertex because, by the ordering condition, there must be an additional out-neighbor. A converging branch is only necessary when an element precedes a proper subset of another element's out-neighbors. Any finishpoint on the existing branch must therefore precede its own private startpoint (which could be comparable to the converging branch, or after some diverging branch). □

Propositions 6.4 and 6.5 will establish the existence of certain paths when building forbidden orders after a failure. The existence of elements preceding diverging branches and succeeding converging branches must also be established.

Proposition 6.6. *Every diverging branch vertex has the finishpoint of a path preceding it in the tree.*

Proof. A diverging branch vertex can only be created on a join vertex, which by definition has the finishpoint of an element preceding it. If the join vertex has been refined, then the finishpoint may not be directly preceding the branch, but it still finishes at a predecessor. □

Proposition 6.7. *Every converging branch vertex has the startpoint of a path on or succeeding it in the tree.*

Proof. A converging branch vertex is created when one element x contains a non-empty, proper subset of another's, y's, out-neighbors. The startpoint of a neighbor therefore must be succeeding it. □

Finally, the construction algorithm builds tree models that grow linearly with the size of the input.

Proposition 6.8. *The construction procedure outlined above builds a tree model with $O(n)$ tree vertices.*

Proof. Clearly there are at most n tree vertices added for the endpoints of elements, and there are at most n join vertices. When an element is closed on a join vertex, due to a converging branch, it is possible that a refinement was necessary, so each element may cause a refinement for its closing. Combined, the number of per-element vertices charged in this way is $\leq 3n$.

It is possible that a refinement may occur without closing the path on an element. When this occurs, an element has strictly fewer neighbors in a subtree than a previously added element. By the ordering condition, there must be neighbors in a different tree. Conceivably, such an element could cause a number of refinements equal to its out-degree, however such refinements happen only when trees are joined together. There are $O(n)$ trees to start, one for each maximal element. The number of vertices added in this manner is equal to the number of edges in a tree that tracks the joining of minimal elements, $2n' - 2$, for n' minimal elements.

This list of ways to add a tree vertex is exhaustive given the construction procedure, so the total number of tree vertices is $O(n)$. □

6.3.2 Failing to Add an Element

The previous section discussed the necessary condition for adding a new element to the tree. In order to add an element x, there must be, in each subtree that contains a

single out-neighbor of x, a tree-vertex that precedes all of the neighbors of x (complete paths for closed elements, finishpoints for open elements), and no startpoint of a non-neighbor. This section shows, given the construction procedure outlined above, if no such tree vertex exists, then the partial order contains one of the forbidden orders in Figure 6.1. Throughout these proofs, the naming of the elements will follow as closely as possible to the labeling in Figure 6.2.

Even when x cannot be added, there is a minimal out-neighbor of x in the subtree S, otherwise, S would not be considered. In this case, one maximal neighbor is shown, and a rooted subtree called R, is grown from that neighbor, by extending R to a maximal rooted tree. R, then, has the following properties: (1) R contains the path of at least one out-neighbor of x; (2) R does not contain the startpoint of a non-neighbor of x; (3) R contains all tree vertices descending from any tree vertex in R, and R is maximal in size, i.e. adding another element would violate either (2) or the fact that R is a rooted subtree; (4) R does not contain the startpoints of all closed out-neighbors and the finishpoints of all open out-neighbors of x in S. By the construction procedure, if R has properties (1)-(3) but not (4) and edge could be made from x_f to the root of R and the precedence relation would be maintained. R exists only when no suitable branch vertex can be found. The existence of R implies the existence of one of the orders from Figure 6.1.

The existence of R implies that there is at least one neighbor of x in S outside of R. Since the host is a tree, in the underlying undirected graph, there is a unique path between a tree vertex in R and an endpoint of a path corresponding to any other out-neighbor. Of particular interest is the number of vertices in this path where the direction of the edges changes. These vertices are called alternation points. A tree-vertex is called a sink in the path if all of the directed edges in the tree on the path are in-edges, and a source if all edges on the path are out-edges. As paths with more alternation points are considered, the number of alternation points along the path

is minimal among all paths to out-neighbors outside R. Minimality ensures no tree vertex in the path precedes the endpoint of a out-neighbor outside of R until the end of the path. The endpoint that has the fewest alternation points between itself and R is called a *minimum-distance endpoint*.

The proof is divided into four cases with zero, one, two, and many alternation points along the path to the minimum-distance endpoint. First is the case where there are zero alternation points. This means that a neighbor not in R has an endpoint that precedes some element in R. This case is further divided into the sub-case where the endpoint precedes the root of R, and the sub-case where the endpoint precedes elements in R but not the root. Next, the case with one alternation point is handled. By definition, in this case, there is a neighbor whose endpoint precedes a tree vertex that is zero alternation points away, but not in R. The end of the path in R is always a sink, however, the other end of the path can be either a source or a sink. Cases for paths with more than one alternation point are divided into two sub-cases: the sub-case where the second endpoint of the path is a sink, and the sub-case where it is a source.

Since the finishpoint of an out-neighbor of x preceding the startpoint of a non-neighbor would be a violation of transitivity, only the startpoint of a neighbor preceding the root of R is considered.

Lemma 6.2. *If, when adding x to S, there is the startpoint of an out-neighbor of x preceding R, then there exists one of the forbidden orders from 6.1.*

Proof. Let the neighbor of x outside of R be z. Since R is maximal, the element directly preceding the root of R is either the startpoint of a non-neighbor or a diverging branch vertex with a branch containing a non-neighbor. Let that non-neighbor be y. Let w be either the element that closed y above the root of R or the element that formed the diverging branch vertex.

The path of z must be closed, since it has a startpoint. Let a be the element

that closed z. The path of z contains w_f; otherwise w would precede z, or z would precede w, violating transitivity. A maximal element is closed as soon as it joins a larger subtree; and so its path does not contain the endpoints of the path of any other element, so z cannot be maximal. Therefore, there must exist an element b in R which must be an out-neighbor of z, x, w and a, and may be an out-neighbor of y.

This gives the forbidden suborders in which a to x is absent, as a was added before x; z to b is present (for some b, it may not be present for every b); and a to w and y to b are optionally present. □

For the following lemma, the element being added is called a instead of x to match the labeling scheme used in Figure 6.2.

Lemma 6.3. *If when adding a to S, an endpoint of a out-neighbor of a precedes some tree vertex in R, but is incomparable to the root of R, then there exists one of the forbidden orders from Figure 6.1.*

Proof. There is a converging branch, as both the root of R and the endpoint precede a common vertex. Let b be the element whose path starts on or after the converging branch by Proposition 6.7. For the side of the branch in R, construction of a converging branch implies there is a startpoint either preceding the branch or succeeding the tree vertex preceding the branch, and a finishpoint which closed that path or created the diverging branch. Call the element with the startpoint in R and preceding the converging branch z. Call the element that closed it x.

For the other side of the converging branch, there is also a startpoint at the tree vertex directly preceding it, or after a diverging branch at the tree vertex. If this is not a neighbor, but there is the startpoint of a neighbor above it, then there is the same situation as Lemma 6.2, so the obstruction may be constructed as in that case, with an R' rooted at the converging branch. Otherwise, it is a neighbor, call it y. The element which closed y may or may not be a neighbor of a is called w.

In this situation, all of the forbidden configurations that are not crowns are possible. w may or may not be a neighbor of a, but since x may not be closed, x_s may not be in R, and so x may or may not be an out-neighbor of a. The elements y and z may be on diverging branches directly preceding the converging branch vertex, and therefore they may or may not precede b. □

That concludes the consideration of zero alternation points. Now, one-alternation-point paths are considered.

Lemma 6.4. *If, when adding x to S, there is a diverging branch which precedes some neighbor in R and a minimum-distance endpoint of a neighbor of x outside R, then there exists one of the forbidden orders from Figure 6.1.*

Proof. If there is any endpoint below the branch, there is a startpoint. Suppose there is a finishpoint but no startpoint. Then the finishpoint must belong to a minimal element. If a minimal element has an finishpoint below a branch but not a startpoint, either the startpoint is on the branch itself, which is a violation of the minimum distance, or the path of a minimal element spans some branch vertex, which is a violation of the construction algorithm.

Again, let z be the neighbor of x which is outside R. Let a be the element whose join vertex is the diverging branch. Call the maximum tree vertex in R which succeeds the branch vertex r^{max}. Either r^{max} is the root of R or is a converging branch vertex, so there exists some element b whose startpoint is on or after r^{max}. Clearly a precedes b and z.

y and w are found as in the earlier lemmas. If r^{max} is the root of R, then either some startpoint or another diverging branch with a non-neighbor below keep R from continuing until a_f. The non-neighbor which causes the problem is y, and w is either the closer of y or the creator of the branch. This case is depicted in Figure 6.6(a).

If r^{max} is not the root of R, then there is a converging branch vertex. In this

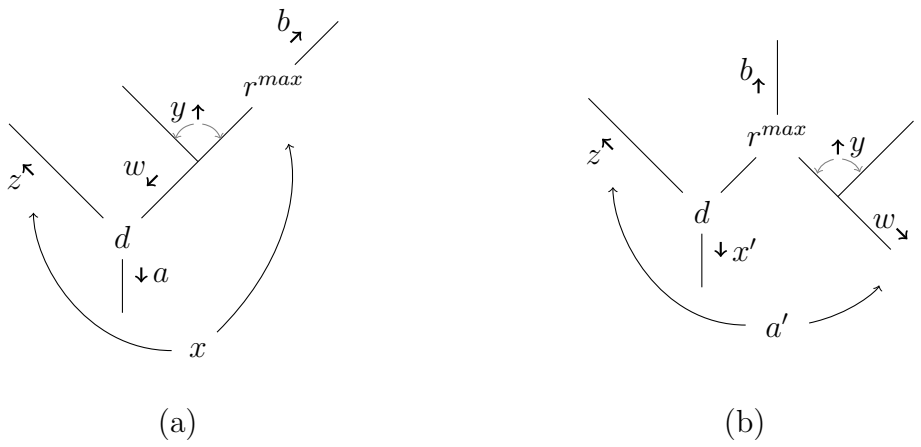


Figure 6.6: The two cases for Lemma 6.4. (a) depicts the case When r^{max} is the root of R. On the right, the figure shows when r^{max} is a converging branch vertex within R. In both cases, the rounded arrows for y indicate that the startpoint of y may be on either branch.

case, to match Figure 6.2, x is renamed a', and a is renamed x'. Above that tree vertex, outside of R, there is either the startpoint of an element, which must be a non-neighbor, or a branch with a non-neighbor. Again, call the non-neighbor y and the closer or branch-creator w. Such a w must exist before the diverging branch, by Proposition 6.4, y must have an in-neighbor private from z.

In this configuration, the only "optional" adjacency is a before w. While in the Figure 6.6 it appears y before b is optional, y is not maximal, since so there is always some vertex b which succeeds the startpoint of y, as well as all the other necessary elements. The forbidden orders for this case are the orders second from the left in columns two and three in Figure 6.1. □

Lemma 6.5. *If when adding a to S, some neighbor b_0 in R and a minimum-distance endpoint of a neighbor of a outside R, c, are separated by two alternation points, there exists a 3-crown, which is one of the forbidden orders from Figure 6.1.*

Proof. If there are two alternation points, the minimum-distance neighbor c has its startpoint on a source in the path. If c's endpoint were on a sink in the path, it would

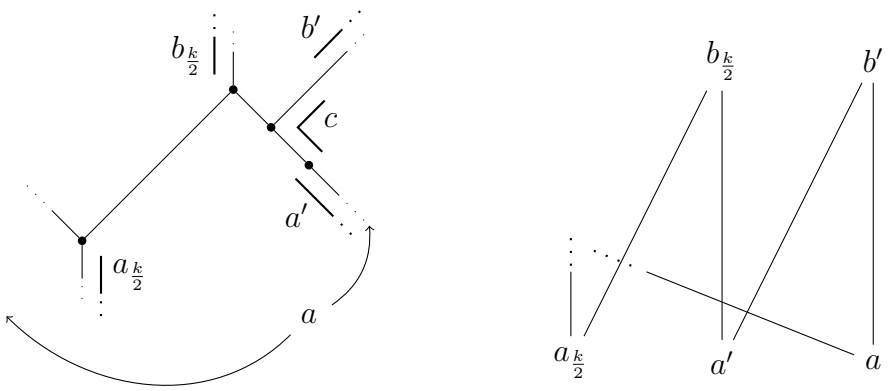


Figure 6.7: When a minimum distance neighbor c is a source in the alternating path, there exists an out-neighbor b' which can be used in constructing a crown.

precede the element which starts after the converging branch, and so c would not be a minimum-neighbor. Since c cannot be minimal (its startpoint precedes a converging branch), it also cannot start after the converging branch at the second alternation point. Any out-neighbor would then be a distance-1 neighbor of a. Therefore, while it starts on the path, there must be a branch to a different path before the converging branch on the alternating path. Therefore, c has an out-neighbor b' ending after that branch. Let the element that created the branch be a'. By the same minimum-distance-neighbor constraint, a cannot precede a'. A more general version of this configuration (one with an unspecified, but even number of alternation points) can be seen in Figure 6.7.

Let the element whose startpoint succeeds the second alternation point by Proposition 6.7 be b_1, and the element whose finishpoint precedes the first alternation point by Proposition 6.6 be a_0.

The set a_0, a', a, b_0, b_1, b' induces a 3 crown in the partial order. □

Lemma 6.6. *If when adding a to S, some neighbor b_0 in R and a minimum-distance endpoint of a neighbor of a outside R are separated by $k > 1$ alternation points, and both ends of the path are sinks, then there exists a $(\lfloor \frac{k}{2} \rfloor + 2)$-crown, which is one of the forbidden orders from Figure 6.1.*

Proof. When both ends can only be sinks, on the path, there are $\lfloor \frac{k}{2} \rfloor$ converging branches. The neighbor in R is b_0. Label the element below each converging branch by Proposition 6.7 $b_0, \ldots, b_{\lfloor \frac{k}{2} \rfloor}$, increasing the subscript as the distance from R increases. The label of the neighbor at the end of path is $b_{\lfloor \frac{k}{2} \rfloor}$. It can be assumed a precedes a startpoint here, because if there is only a finishpoint, then the element is open element, and not it is not minimal. Label the elements preceding diverging branches, required by Proposition 6.6, a_0 to $a_{\lfloor \frac{k}{2} \rfloor + 1}$. The element a completes the $(\lfloor \frac{k}{2} \rfloor + 2)$-crown. □

Lemma 6.7. *If when adding a to S, (1) some neighbor b_0 in R and a minimum-distance endpoint of a neighbor of a outside R, c, are separated by $k > 2$, and (2) the end of the path outside of R is a source, then there exists a $(\frac{k}{2} + 2)$-crown, which is one of the forbidden orders from Figure 6.1.*

Proof. Just as described in the proof of Lemma 6.5 and shown in Figure 6.7, the a' and b' are implied by c. The rest of the b_i's are the elements which much succeed the converging branch vertices by Proposition 6.7, and the a_i's are preceding the diverging branch vertices as required by Proposition 6.6. There are then $\frac{k}{2} + 2$ b's including b_0 in R and b', and $\frac{k}{2} + 2$ a's including a and a'. These elements create a $(\frac{k}{2} + 2)$-crown. □

These technical lemmas provide the necessary results to characterize and count the class.

Theorem 6.1. *A partial order is a strict path-precedence order if and only it does not contain as an induced subgraph any of the orders from Figure 6.1.*

Proof. Lemma 6.1 shows that the absence of those orders is a necessary condition. The construction algorithm given is a procedure that succeeds to incrementally build a tree model when a rooted subtree with all of the neighbors and no non-neighbors exists in each subtree with any neighbors. Lemmas 6.2-6.7 show that if no such subtree exists, there is one of the orders from Figure 6.1. Thus, the absence of those orders is also sufficient. □

Theorem 6.2. *There are* $2^{O(n \log n)}$ *strict path-precedence orders on* n *elements.*

Proof. By Proposition 6.8, if a tree can be constructed, the size of the constructed tree is $O(n)$. Storing $O(n)$ tree vertices and $O(n)$ tree edges takes $O(n \log n)$ bits. Each element in the partial order stores the label of the tree vertices that are the endpoints. This takes an additional $O(n \log n)$ bits. Since the partial order can be stored uniquely $O(n \log n)$ bits, the result follows. □

6.4 Implementing the Construction Algorithm

6.4.1 Ordering and Preprocessing the Elements

The construction algorithm calls for a element ordering with two properties. First, the elements must be ordered by increasing out-degree. Since the degrees of vertices are in the range from 0 to n, this may be easily done with a bucket sort. Second, elements with the same out-neighborhoods must be added together. This can be done with the partition refinement data structure introduced in Chapter 1. An initial partitioning the elements in order of increasing out-degree, then partitioning using the in-neighbors of the elements gives an order with both necessary properties in $O(m+n)$ time. Elements with the same out-neighborhoods will always remain in the same partition.

6.4.2 Finding the Minimal Neighbors

The construction algorithm used only the existence or absence of a single minimal neighbor to construct the tree. This section presents a method for finding such a vertex. The algorithm uses a "painting" technique to label vertices in the subtree that holds out-neighbors. Then, the tree can be refined by splitting a vertex, if that resolves all precedence constraints.

During the running of the construction algorithm, each subtree must store the elements in that tree. When adding an element x, begin painting with the non-neighbors of x with startpoints defined in the tree. Open paths of non-neighbors can be extended, so those elements do not contribute to the painted tree. For each closed non-neighbor of x, paint red the tree vertices above and including the startpoint, marking that the finishpoint of a non-neighbor may not precede this vertex. When a vertex that has already been colored red is reached, stop coloring on that path. By stopping at a painted vertex and not repeating work, each tree vertex contributes at most a constant amount of work to the painting process, and each non-neighbor contributes a constant amount of other work, so this painting the tree red takes $O(n)$ time.

After the the tree has been painted by non-neighbors, the neighbors are considered. The goal is to find the root of the subtree with all neighbors and no non-neighbors, or to construct the rooted subtree R (as described in Section 6.3.2). The search begins with the finishpoint of a maximal neighbor. If the current tree vertex is not red, examine the endpoints at that vertex. Closed elements are marked as "seen" when their startpoint is found, open elements when their finishpoint is found. Mark the appropriate neighbors at the current tree vertex as seen. If this tree vertex is a diverging branch, before continuing to a predecessor, search down each branch. Since the vertex is not red, there are no non-neighbors down any of the diverging branches. Once all the diverging branches are searched (only searching descendants

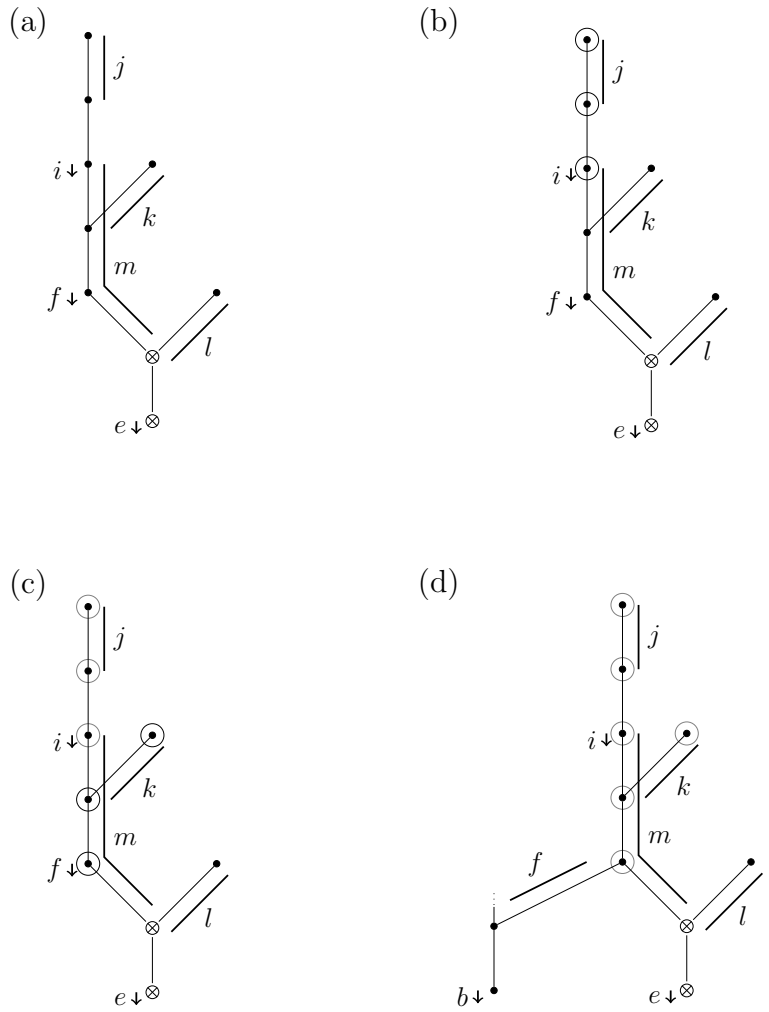


Figure 6.8: Coloring the subtree to add the element b: (a) shows the non-neighbor l painting tree vertices; (b) starts the neighbor search with j; (c) searches the other branch at a diverging branch vertex; and (d) shows b being added at the root of the search tree.

of the current tree vertex), search the predecessors.

If the tree vertex is a converging branch, examine next the preceding tree vertex for each of the branches. As discussed earlier, this next tree vertex is either the startpoint of a path, or a diverging branch vertex. To continue, there must be at least one path which is not painted red. If more than one path is not painted red, then there is no single tree vertex before every neighbor. Choose one of the branches to

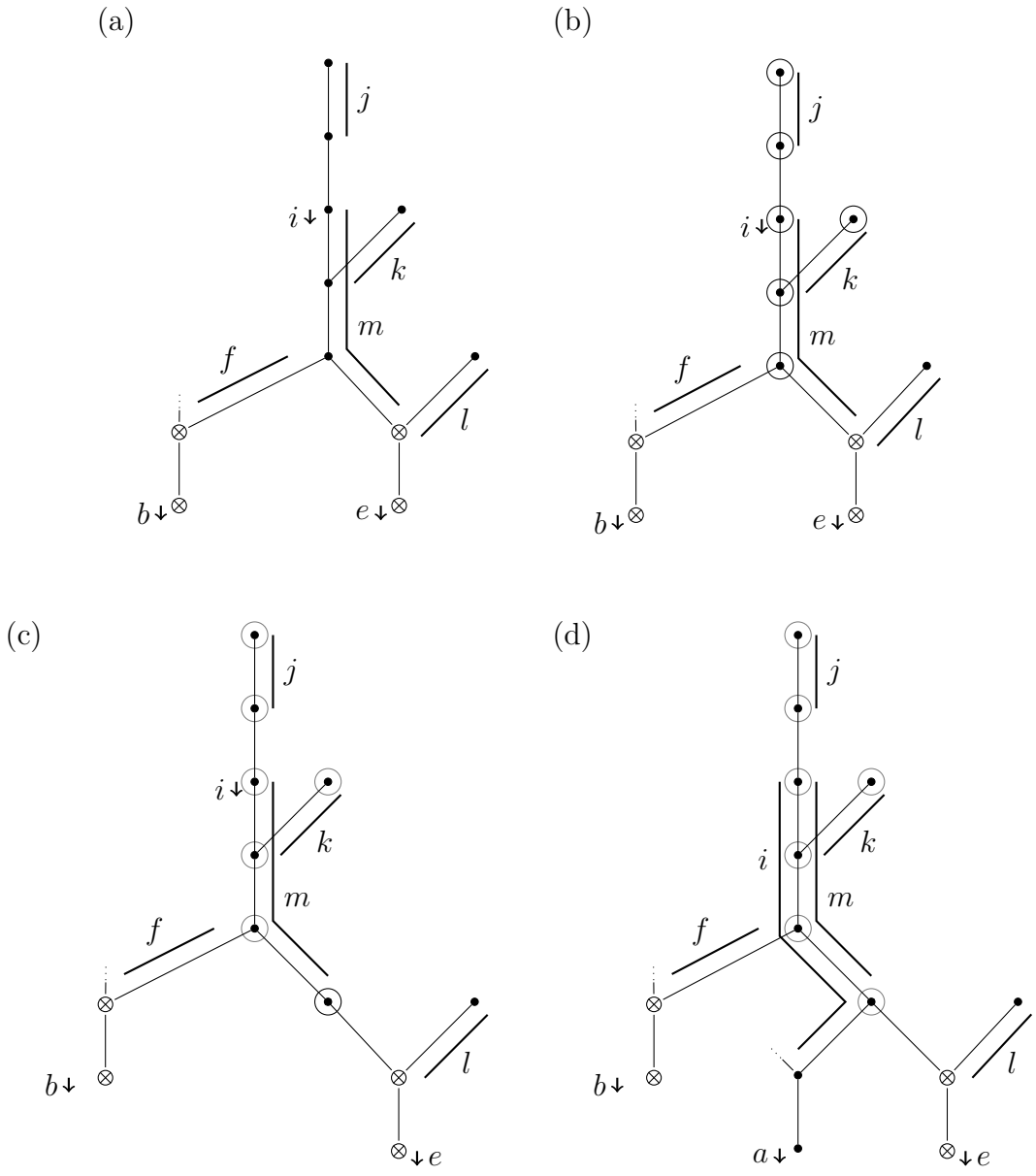


Figure 6.9: Coloring the subtree to add the element a: (a) shows the non-neighbors l and f painting tree vertices; (b) starts the neighbor search with j; (c) shows the refinement of the startpoint of l and m (d) shows a being added at the root of the search tree.

continue with the search, which now can only be creating R.[2] The search terminates whenever a minimal or red tree vertex is reached. If all neighbors have been seen, then this is the vertex to precede with the new join vertex and finishpoint. If there are unseen neighbors, then the searched tree is an R that can be used to construct an obstruction. Tree refinement works by adding at most one tree vertex, in the case where not all neighbors of x have been found before reaching the red tree vertex v that caused the neighbor-search to stop. After tree painting, a red tree vertex can be refined into a red vertex preceding an unpainted vertex. The precess is illustrated in Figure 6.9(b)-(c), where the addition of a requires the tree vertex containing the startpoints of l and m to be refined.

6.4.3 Finding an Obstruction

When a failure is detected, the subtree R found in the neighbor search and the element x being added (sometimes x is relabeled to a to match Figure 6.2) are used to find a forbidden configuration in the subtree S. As in Lemmas 6.2-6.7, the minimum-distance neighbor and the number of alternation points between R and that minimum distance neighbor are used to determine where to find the remaining elements of the obstruction.

The first step is finding the the neighbor the fewest alternation points from R. Conceptually, this is equivalent to finding the shortest path in the undirected comparability graph underlying the transitive closure of the tree. The search process outlined below searches that graph without explicitly constructing the transitive closure. The algorithm starts by putting all ancestors of vertices in R into an array of lists A at index 0. Then at $A[1]$, place the descendants of those tree vertices in $A[0]$ which have not been discovered. The search proceeds in breath-first manner, searching all tree vertices at a particular distance before searching those at a greater

[2]At this point, failure is assured, however, since R must be a maximal rooted tree, the search must continue to certify the failure)

Alternation Points	Element Location	
0,	x	element being added
t_R is root of R	z	neighbor of x outside of R
	a	element with endpoint above z
	w	element which ends on the path between the root of R and startpoint of z
	y	private neighbor of w with respect to x, either between endpoint of w and root of R or on another branch succeeding w
	b	an element within R
0,	a	element being added
otherwise	z	element with startpoint on or succeeding (on a different branch) the tree vertex immediately preceding t_R, in R
	x	element which closed z, or made the branch vertex preceding z
	y	element with startpoint on or succeeding (on a different branch) the tree vertex immediately preceding t_R, outside of R
	w	element which closed y, or made the branch vertex preceding y
	b	element with startpoint at tree vertex succeeding t_R

Table 6.1: The positions of the elements in the tree that lead to a forbidden configuration, for a neighbor 0 alternation points from R

Alternation Points	Element Location	
1	x	element being added
	z	neighbor outside of R
	a	element with endpoint the created the branch to z
	w	element which ends on the path between t_R and a_f
	y	private neighbor of w with respect to x, either between endpoint of w and t_R, or on another branch succeeding w
	b	element with startpoint at tree vertex succeeding t_R

Table 6.2: The positions of the elements in the tree that lead to a forbidden configuration, for a neighbor 1 alternation point from R

distance. If a vertex was discovered through an ancestor, and it is a converging branch in the tree, then all ancestors on other branches will be one additional edge (in the comparability graph) away. For tree vertices discovered through a descendant, it is the other descending branches that are an additional edge away.

With each the tree vertex in the list, store the "discovering" vertex, the vertex in the previous list whose expansion lead to finding that vertex. For tree vertices in $A[0]$, this is the minimum neighbor in R, call it t_R. For vertices in the list at $A[i]$ for $i > 0$, this vertex is called t_{i-1}. For any vertex, k alternation points from R these labels $t_R, t_0, ...t_{k-1}$ can be traced back to find the complete path.

For zero-alternation-point paths, there are two cases: when t_R is the root of R and otherwise. This distinction mirrors the cases from Lemmas 6.2 and 6.3. For both cases, the locations of the elements for the forbidden suborder are found in Table 6.1. The locations of the forbidden-suborder elements with one alternation point, as described by Lemma 6.4, are shown in Table 6.2. Finally, for longer paths, the locations of the elements are given in Table 6.3. The existence of these elements is proven in Lemmas 6.5-6.7.

The search to find the minimum-distance neighbor takes time proportional to the

Alternation Points	Element Location	
k	a_0	element being added
$k > 1$, odd	b_0	element succeeding t_R
	$a_1 - a_{\lfloor \frac{k}{2} \rfloor - 1}$	elements preceding the t_i's for $1 \leq i \leq k$, and i is odd
	$b_1 - b_{\lfloor \frac{k}{2} \rfloor - 1}$	elements succeeding t_i's for $1 < i < k$ and i is even
	$b_{\lfloor \frac{k}{2} \rfloor}$	neighbor of a_0 outside of R
k	a_0	element being added
$k > 1$, even	b_0	element succeeding t_R
	$a_1 - a_{\frac{k}{2} - 1}$	elements preceding the t_i's for $1 \leq i \leq k$, and i is odd
	$b_1 - b_{\frac{k}{2} - 1}$	elements succeeding t_i's for $1 < i < k$ and i is even
	$b_{\frac{k}{2}}$	neighbor of a_0 outside of R, starting immediately preceding starting succeeding a branch immediately preceding t_k.

Table 6.3: The positions of the elements in the tree that lead to a forbidden configuration, for a neighbor k alternation points from R

size of the tree plus the size of all of the endpoint lists, therefore $O(n)$. Once the neighbor is found, the tree is searched again for specific elements in the locations given by Tables 6.1-6.3. As no tree vertex needs to be visited twice, this search is also $O(n)$. Therefore the entire algorithm for finding the obstruction, given R and the element x, is $O(n)$.

6.4.4 Overall Running Time

To summarize, the algorithm for certifying recognition works as follows. First, order the elements using partitioning into partitions of elements with increasing out-degree, such that elements with the same out neighborhood are in the same partition. This partitioning step takes $O(m + n)$ time, $O(n)$ for sorting by out-degree, creating the initial partition, and then $O(m + n)$ for the partitioning. Next, the partitions are added to the model incrementally. This possibly involves searching each of the existing

subtrees, which gives $O(n)$ for each element added, $O(n^2)$ for adding all the elements. If there is a failure, the $O(n)$ time to find the obstruction does not increase the time-complexity. If there is no failure, then the algorithm returns a valid tree model, certifying the success.

Theorem 6.3. *Certifying-recognition of strict path-precedence orders given a partial order is* $O(n^2)$.

Proof. The analysis above shows that the dominating step in the certifying recognition algorithm, constructing the tree model, is $O(n^2)$. □

Corollary 6.1. *The construction problem for strict path-precedence orders given a partial order is* $O(n^2)$.

CHAPTER 7

SUMMARY AND OPEN PROBLEMS

This dissertation has covered problems associated with the precedence of paths in trees, using two different notions of precedence. Strict precedence uses the same notion as in interval orders, where one path x precedes another y if the path x ends before y begins. Weak precedence requires only that x start before and end before y, but the paths may intersect. Partial orders given by the weak precedence of intervals are exactly equal to the two-dimensional partial orders. Both of these notions of precedence in a tree, therefore, generalize well-known and well-studied classes of partial orders.

Chapter 2 studied the problem of finding a realizer for partial orders with a tree diagram and presents a linear-time algorithms for both the rooted and general case. The partial orders of strict precedence in a rooted tree were first studied as generalized interval orders by Faigle *et al.* [12], the equivalence proven by Müller and Rampon [34]. Chapter 3 presented the first certifying recognition algorithm for the class, matching the $O(m + n)$ running time of Garbe's recognition algorithm [17]. For the orders of weak precedence in a rooted tree, an intersection characterization of the class is proven in Chapter 4, as well as a $O(m + n)$ recognition algorithm for the bipartite, rooted, weak path-precedence orders. Chapter 5 studied the independent set and clique problems for the precedence orders with rooted models, and gave algorithms for these problems, given the model as input, that are asymptotically more efficient then the well-known algorithms for these problems on comparability graphs. Finally, Chapter 6 presented a forbidden induced-subgraph characterization of general, strict path-precedence orders, and an $O(n^2)$ certifying recognition algorithm.

7.1 Tree Orders

Chapter 2 presented a algorithm for computing the realizer of a tree order. Topological depth-first search has a clear geometric interpretation. UndoUnder, also has a geometric interpretation, reversing the order in which different children are visited, based on the tdfs. Re-Right does not have a clear geometric interpretation, it simply rearranges a list. The question then is this: Is there a geometric interpretation of the changes that Re-Right makes to the tdfs order? There is another unsatisfying part of the realizer algorithm for general trees. One would like to have an algorithm which, when the tree is only two-dimensional[1], gives the two-dimensional realizer (or at least gives a realizer with two identical lists). These two questions seem related. It seems especially likely that understanding how an embedding of the Re-Right order relates to the tdfs can offer insight for a single minimum-realizer algorithm for all trees.

7.2 Recognition of Weak Path-Precedence Orders

For weak precedence, this work only presents a recognition algorithm for bipartite, rooted, weak path-precedence orders. The approach for recognizing the bipartite case was to use restrictions on certain elements which must be consecutive in the tree order. Removing the restriction to bipartite graphs, this approach does not lead as readily to a result. For an element x, there are two sets of elements which must be on the same root-to-leaf path as x. The first is the set of all in-neighbors; the second set is the set of elements which share an out-neighbor with x (these sets are obviously not mutually exclusive).

While the location of elements in these sets is necessary, it is not sufficient. There exist rooted, weak path precedence orders for which other elements (not from the set of in-neighbors or the set sharing out-neighbors) are forced to precede an element in the tree model. Consider the chevron (directed oppositely from a previous example)

[1]Rooted-trees are two-dimensional, but recall that any tree which does not contain a transitive orientation of the graphs in Figure 2.2 is two-dimensional.

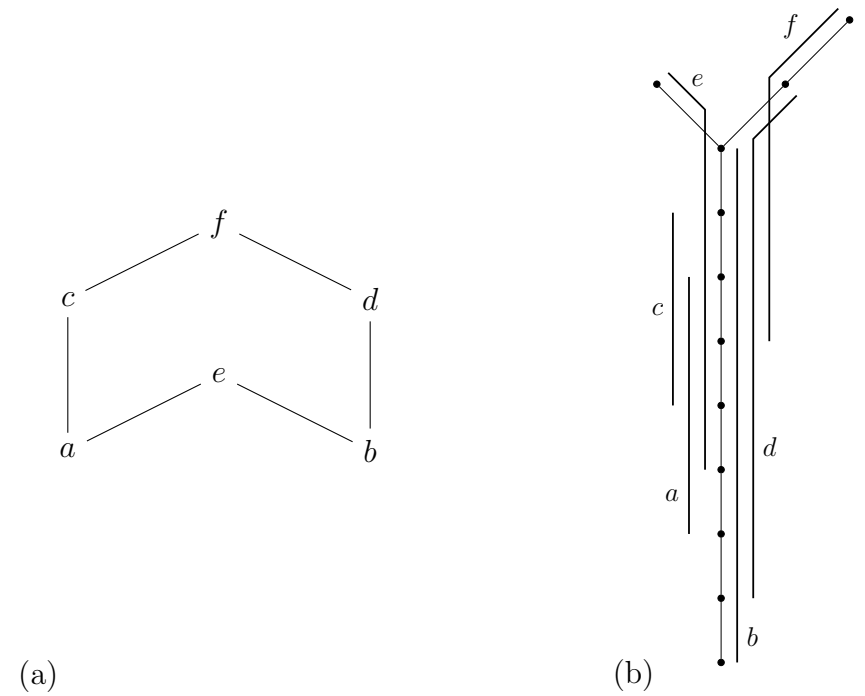


Figure 7.1: A weak precedence model (a) before and (b) after transformation

in Figure 7.1. Elements a, b, c, and d precede f, so all must end on the same path in the tree. If a and b precede both c and d in the path, for example $abcd$ the second minimal element, in this case b must precede both c and d in both lists, since a and b must be reversed in the second list. That is a contradiction. Therefore, the tree ordering must be either $acbd$ or $bdac$. Due to the symmetry of the partial order, we can choose $acbd$ without loss of generality. Since e must succeed both a and b in the tree, e also succeeds c in the tree, even though c neither precedes c nor shares an out-neighbor with e. The tree model is shown in Figure 7.1. This example shows the difficulty of extending the basic approach of finding consecutive sets for the bipartite case to general rooted, weak path-precedence orders. The recognition question of these partial orders remains open.

Open Question 7.1. Can rooted, weak path-precedence orders be recognized in polynomial time?

As with strict precedence, after the rooted-tree case is solved, the natural next

question is what happens in general trees. This is the class of (general) weak path-precedence orders. All pertinent questions remain open for weak-precedence in these general trees. The characterization of rooted, weak path-precedence orders as the intersection of a linear order with a forest of rooted-tree orders offers a second avenue for generalization. What partial orders result from the intersection of a linear order with a forest of tree-orders? A good starting point for studying this class is to prove or disprove a relationship with weak path-precedence orders.

Open Question 7.2. Are weak path-precedence orders equivalent to the intersection of a linear order with a forest of tree orders?

If the answer is yes, then weak path-precedence orders are clearly four-dimensional, and there is a nice representation (using the tree realizer) for which adjacency testing can be done in constant time. If no, then the new class may be worthy of study in its own right.

7.3 Optimization Problems on Path-Precedence Orders

Chapter 5 studied how generalization of algorithms on the interval model could be extended to efficient algorithms on rooted path-precedence orders. These problems are still open for general tree models. The clique algorithms used the property that all cliques lie in a single root-to-leaf path in the tree. For general trees, one can simply run the algorithm from each minimal tree-vertex, increasing the running time by at most a factor of n. For strict precedence, processing the host tree in topological sort order seems likely to find the maximum clique again in $O(n)$ time. For weak precedence, the approach used for rooted models could be extended by running the rooted algorithm at every minimal vertex. This approach gives a running time of $O(n^2 \log n)$, which is slower than the standard algorithm for comparability graphs. It may be possible to reduce this time bound with judicious caching to return to $O(n \log n)$.

Open Question 7.3. Can the size of the maximum clique in a weak path-precedence order be found faster than that of a comparability graph when given the model as input?

The algorithms for the independent set problem relied on the structure of interactions between incomparable paths. It is not clear how this approach can be extended to a general tree model. The question, then, is whether the tree model can be used to find independent sets in path-precedence orders in less time than on comparability graphs.

Open Question 7.4. Can the size of the maximum independent set in a strict path-precedence order be found faster than that of a comparability graph when given the model as input?

Open Question 7.5. Can the size of the maximum independent set in a weak path-precedence order be found faster than that of a comparability graph when given the model as input?

7.4 Intervals in a Series-Parallel Partial Order

In this work, we generalized a rooted tree to a general tree. In that generalization, the partial order dimension of the host increased from 2 to 3. There is another class which contains rooted trees and also has dimension 2: the series-parallel partial order. Series-partial orders are defined by a construction procedure. A single element is a series-parallel partial order. A larger series-parallel partial order can be constructed from two series-parallel partial orders, P and Q in two ways. A series construction is one where every element in P precedes every element in Q. In a parallel construction, elements from P and Q are incomparable in the combined partial order.

Series-parallel partial orders are well-known to be N-free [42], meaning if an incomparable a and b both precede c, then any out-neighbor of b either precedes c, or succeeds a. In particular, this excludes series-parallel partial orders from con-

taining crowns. It is not difficult to show that series-parallel strict path-precedence orders do not contain crowns, as their representation would require a crown in the host. However, forbidding crowns alone cannot be sufficient by a counting argument. Chordal bipartite graphs are free from the cycles that when directed would become crowns, and there are $2^{\Theta(n \log^2 n)}$ chordal bipartite graphs, see [37]. However, by the same counting argument used for tree hosts, there are $2^{O(n \log n)}$ series-parallel strict path-precedence orders. Crowns, therefore, are not the only obstruction. In rooted, strict path-precedence orders, crowns were implicitly forbidden because  was forbidden. With series-parallel strict path-precedence orders, this may be the case as well, or there may be something in addition to crowns.

Open Question 7.6. Is there a simple forbidden induced subgraph characterization of series-parallel, strict path-precedence orders?

Open Question 7.7. Can series-parallel, strict path-precedence orders be recognized in polynomial time?

Just as with tree models, members of a clique are still restricted to a single directed path in the host model. This suggests that a similar approach will yield a clique algorithm given the model which is faster than the comparability graph algorithm. Series-parallel orders can be used to model finite regular expressions, so cliques under weak precedence has an application in finding the longest match including finite regular expressions [30].

REFERENCES

[1]Belfer, A. and Golumbic, M. (1993). Counting endpoint sequences for interval orders and interval graphs. *Discrete Mathematics*, 114(1-3):23–39.

[2]Bergroth, L., Hakonen, H., and Raita, T. (2000). A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 39. IEEE Computer Society.

[3]Bodlaender, H., Kloks, T., Kratsch, D., and Müller, H. (1998). Treewidth and minimum fill-in on d-trapezoid graphs. *Journal of Graph Algorithms and Applications*, 2(5):1–23.

[4]Bogart, K. (1994). Intervals and orders: What comes after interval orders? *Orders, Algorithms, and Applications*, pages 13–32.

[5]Bogart, K., Bonin, J., and Mitas, J. (1995). Interval orders based on weak orders. *Discrete Applied Mathematics*, 60(1-3):93–98.

[6]Bogart, K., Mö hring, R., and Ryan, S. (1998). Proper and unit trapezoid orders and graphs. *Order*, 15(4):325–340.

[7]Booth, K. and Lueker, G. (1976). Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379.

[8]Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill.

[9]Dagan, I., Golumbic, M., and Pinter, R. (1988). Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, 21(1):35–46.

[10]Dietz, P., Furst, M., and Hopcroft, J. (1979). A linear time algorithm for the generalized consecutive retrieval problem. Technical report, Cornell University.

[11]Dushnik, B. and Miller, E. (1941). Partially ordered sets. *American Journal of Mathematics*, 63(3):600–610.

[12]Faigle, U., Schrader, R., and Turan, G. (1992). The communication complexity of interval orders. *Discrete Applied Mathematics*, 40(1):19–28.

[13]Felsner, S., Müller, R., and Wernisch, L. (1997). Trapezoid graphs and generalizations, geometry and algorithms. *Discrete Applied Mathematics*, 74(1):13–32.

[14]Fishburn, P. (1985). *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*. John Wiley & Sons.

[15]Fulkerson, D. and Gross, O. (1965). Incidence matrices and interval graphs, Pacific J. *Pacific Journal of Math*, 15:835–855.

[16]Gallai, T. (1967). Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18(1):25–66.

[17]Garbe, R. (1994). *Algorithmic aspects of interval orders*. PhD thesis, University of Twente, The Netherlands.

[18]Garey, M. and Johnson, D. (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness*. WH Freeman and Company.

[19]Gavril, F. (1974). The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56.

[20]Golumbic, M. (2004). *Algorithmic Graph Theory and Perfect Graphs*. North-Holland.

[21]Golumbic, M. and Monma, C. (1982). A generalization of interval graphs with tolerances. In *Proceedings of the 13th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium*, volume 35, pages 321–331.

[22]Gupta, U., Lee, D., and Leung, J. (1982). Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467.

[23]Hashimoto, A. and Stevens, J. (1971). Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Design Automation Workshop*, pages 155–169. ACM.

[24]Hunt, J. and Szymanski, T. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):353.

[25]Kratsch, D. and Rampon, J. (1998). Tree-visibility orders. *Discrete Mathematics*, 190(1-3):163–175.

[26]Ma, T. and Spinrad, J. (1994). On the 2-chain subgraph cover and related problems. *Journal of Algorithms*, 17(2):251–268.

[27]McConnell, R., Mehlhorn, K., Nä her, S., and Schweitzer, P. (2010). Certifying algorithms. *Computer Science Review*.

[28]McConnell, R. and Spinrad, J. (1999). Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241.

[29]McConnell, R. and Spinrad, J. (2000). Ordered vertex partitioning. *Discrete Mathematics & Theoretical Computer Science*, 4(1):45–60.

[30]McConnell, R. and Spinrad, J. (2011). unpublished work.

[31]Mehlhorn, K. and Näher, S. (1998). From algorithms to working programs: On the use of program checking in leda. *Mathematical Foundations of Computer Science 1998*, pages 84–93.

[32]Mitas, J. (1992). *The structure of interval orders*. PhD thesis, Technische Universitä t Darmstadt.

[33]Monma, C. L. and Wei, V. K. (1986). Intersection graphs of paths in a tree. *Journal of Combinatorial Theory. Series B*, 41(2):141–181.

[34]Müller, H. and Rampon, J. (1997). Partial orders and their convex subsets. *Discrete Mathematics*, 165:507–517.

[35]Paige, R. and Tarjan, R. (1987). Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989.

[36]Service, T. (2011). personal communication.

[37]Spinrad, J. (2003). *Efficient Graph Representations*. American Mathematical Society.

[38]Tarjan, R. (1983). *Data Structures and Network Algorithms*, volume 14. SIAM.

[39]Trotter, W. (2001). *Combinatorics and Partially Ordered Sets: Dimension theory*. Johns Hopkins Univ Pr.

[40]Trotter, W. and Moore, J. (1977). The dimension of planar posets. *Journal of Combinatorial Theory, Series B*, 22(1):54–67.

[41]Tsoukias, A. and Vincke, P. (1999). A generalization of interval orders. *Electronic Notes in Discrete Mathematics*, 2:122–133.

[42]Valdes, J., Tarjan, R., and Lawler, E. (1979). The recognition of series parallel digraphs. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 1–12. ACM.

[43]van Emde Boas, P. (1975). Preserving order in a forest in less than logarithmic time. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 75–84. IEEE.

[44]Wagner, R. and Fischer, M. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

[45]Wiener, N. (1914). A contribution to the theory of relative position. In *Proc. Camb. Philos. Soc*, volume 17, pages 441–449.

[46]Wölk, E. (1965). A note on the comparability graph of a tree. In *Proc. Am. Math. Soc*, volume 16, pages 17–20.

[47]Yannakakis, M. (1982). The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3:351.