Pruning graphs with digital search trees. Application to distance hereditary graphs.

Jean-Marc LANLIGNEL, Olivier RAYNAUD and Eric THIERRY
LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France.
Email: `(lanligne,raynaud,thierry)@lirmm.fr`

Abstract

Given a graph, removing pendant vertices (vertices with only one neighbor) and vertices that have a twin (another vertex that has the same neighbors) until it is not possible yields a reduced graph, called the "pruned graph". In this paper, we present an algorithm which computes this "pruned graph" either in linear time or in linear space. In order to achieve these complexity bounds, we introduce a data structure based on digital search trees. Originally designed to store a family of sets and to test efficiently equalities of sets after the removal of some elements, this data structure finds interesting applications in graph algorithmics. The computation of the "pruned graph" provides a new and simpler algorithm for the recognition of distance-hereditary graphs, and we improve the complexity bounds for the complete bipartite cover problem on bipartite distance-hereditary graphs.

Keywords: graph algorithms, distance-hereditary graphs, sets, digital search trees, amortized complexity.

1 Introduction

Distance-hereditary graphs have been introduced by E. Howorka [How77] in 1977. Originally defined as graphs in which every induced path is isometric, many other characterizations have been found for this class: forbidden subgraphs, properties of cycles, metric characterizations ... (see [BM86] for a survey). We focus on one of them: G is a distance-hereditary graph iff every subgraph of G with at least two vertices contains a pendant vertex (a vertex with only one neighbor) or a vertex that has a twin (another vertex that has the same neighbors). Thus repeatedly removing from a graph G a pendant vertex or a vertex that has a twin, whenever possible (a process called "to ***prune*** the graph"), leads to a graph reduced to one single vertex iff G is a distance-hereditary graph.

This pruning process has a special property, which enables to define a reduction on graphs thanks to the following theorem: given a graph, whatever the order you chose to remove pendant vertices and vertices with twins, it leads to a unique graph, up to an isomorphism. We will call this reduced graph the ***pruned graph*** and the problem of computing the "pruned graph" of any graph appears as a generalization of the recognition of distance-hereditary graphs (graphs with a single vertex as "pruned graph").

Figure 1 shows a sequence of deletions which leads to the "pruned graph".

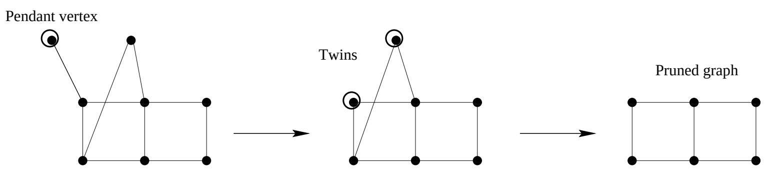

Figure 1: Pruning process

The aim of this article is to provide algorithms to compute the pruned graph of any graph, with best upper bounds on the complexity. To give a first bound, note that if the graph is given by its adjacency lists, the algorithm which consists of finding pendant vertices and twins by comparing adjacency lists and removing these vertices one by one, would lead to an $O(mn)$ time complexity, where n is the number of vertices and m the number of edges. Our algorithms are based on another representation of the graph: instead of keeping the adjacency lists, we consider the neighbourhoods of the vertices as a family of subsets on the set of all vertices, and we work on the digital search tree associated to this family. Storing the graph in such a data structure, also called ***trie***, enables to perform, with good complexities, the deletion of one element in all the subsets and the deletion of a subset (corresponding to the deletion of a vertex) as well as the detection of equal subsets (corresponding to the detection of twin vertices). We will present three slightly different variants of this data structure and the associated algorithms, which lead to the following upper bounds for the computation of the pruned graph (n is the initial number of vertices, m the initial number of edges):

Time bound	$m+n$	$m \log n$	n^2
Space bound	n^2	$m \log n$	$m+n$

In section 2, all the operations we need in order to compute the "pruned graph", are reinterpreted in terms of sets and operations on sets, so that we could describe our data structures and the associated algorithms for a family of sets. This point of view is chosen for commodity, and because the details of the algorithms and the complexity analyses we give apply to any family of sets.

In section 3, we give the applications to graph problems. The computation of the "pruned graph" provides a new algorithm for the recognition of distance-hereditary graphs, with the complexities mentionned above, depending of which variant is chosen. They aren't linear both in time and space. But the only known linear algorithm is very recent and complex (for instance it uses the modular decomposition of cographs as a subroutine), to the point that a first release was not fully functional ([HM90],[Dam97]).

Willing to take advantage of the graph structure to have faster algorithms, we prove that some good orderings of the vertices yield a linear algorithm both in time and space in some special pruning cases, for instance the computation of the "pruned graph" of bipartite graphs.

As a final example, we improve the complexity bounds for the complete bipartite cover problem on bipartite distance-hereditary graphs, with a linear algorithm, whereas the only known algorithm proposed in [Mül96] runs with a $O(mn)$ time complexity.

2 Data structure

2.1 The general problem

Let $G = (V, E)$ be a non oriented graph. If $x \in V$, $\mathcal{N}(x) = \{y \in V, y \neq x | (x, y) \in E\}$ is the open neighborhood of x in G, and $\mathcal{N}_c(x) = \mathcal{N}(x) \bigcup \{x\}$ its closed neighborhood. An $x \in V$ such that $\#\mathcal{N}(x) = 1$ is called a ***pendant vertex***. Some $x, y \in V$ such that $\mathcal{N}(x) \backslash \{y\} = \mathcal{N}(y) \backslash \{x\}$ are called ***twin vertices***. Thus twins share the same vertices as neighbors, and may or may not be linked together. We will call non-adjacent twins ***true twins***, and adjacent twins ***false twins***.

To compute the "pruned graph" of G, we mainly have to detect false twins, namely vertices x, y such that $\mathcal{N}(x) = \mathcal{N}(y)$, and true twins, namely vertices x, y such that $\mathcal{N}_c(x) = \mathcal{N}_c(y)$. The detection of pendant vertices only needs a data structure to keep track of the degree $\#\mathcal{N}(x)$ of each vertex x. Then we need to delete vertices from the graph, and try to detect new removable vertices, and so on.

This process may be expressed in another way. Let's consider the family of sets $\{\mathcal{N}(x) | x \in V\} \bigcup \{\mathcal{N}_c(x) | x \in V\}$ which completely characterizes the graph G. We mainly have to detect equal sets to detect twins (note that for any x, y, $\mathcal{N}_c(x) = \mathcal{N}(y)$ is never possible). After the deletion of a vertex x, we have to update the family of sets by deleting the sets $\mathcal{N}(x)$ and $\mathcal{N}_c(x)$, and suppress all the occurences of x in the other sets. In the whole section 2, we deal with these operations on sets, and then we describe the complete algorithm on graphs in section 3. We now reformulate our main problem in terms of sets.

Let X be a set of cardinality n. Suppose we are given k sets of X, $S_1, \ldots, S_k$ on X. The problem is to find a data structure that will enable us, with the best amortized complexity, to perform the following operations:

Operation Equality: Find (i, j) such that $S_i = S_j$, $i \neq j$, or detect that all sets are different.

Operation Set delete: Given i, delete S_i.

Operation Element delete: Given $x \in X$, delete all occurrences of x from all sets.

2.2 General survey of the data structure

The input thus consists of k sets, $S_1, \ldots, S_k$. l_i is the cardinal of S_i, or its length if we view sets as words. Let d be the size of the input. We suppose that:

$$d \in \Omega(k + \sum_{i=1}^{k} l_i).$$

This is obviously true if the sets are given as lists of elements.

We first need an order on X; This order will identify X and $[\![1, n]\!]$; thus we will able to identify the sets of X and the words of $[\![1, n]\!]$ whose letters are sorted. In some particular applications we will need to compute a special order to guarantee linear time algorithms; but the general results will hold with any order.

In a second time, we decide to store in a *trie* (a tree coding words thanks to labelled edges [Knu73]) the sets in their sorted-letters word form. Due to our presentation, we will call ***lexicographic tree*** this data structure.

Definition 1 The lexicographic tree

- *The root is labelled by the empty set.*
- *Each edge is labelled by elements of X.*
- *To each subset S_i of X corresponds a node labelled by S_i.*
- *The path from the root to each labelled node S_i has its edge labels forming the set S_i*
- *Edges are non-decreasing in a path and all the edges from a node are also ordered from left to right.*

The figure 2 represente the lexicographic tree built from the set $X = \{1, 2, 3, 4, 5\}$ with $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{1, 3\}$, $S_3 = \{2, 3, 5\}$, $S_4 = \{2, 3\}$, $S_5 = \{1, 3\}$.

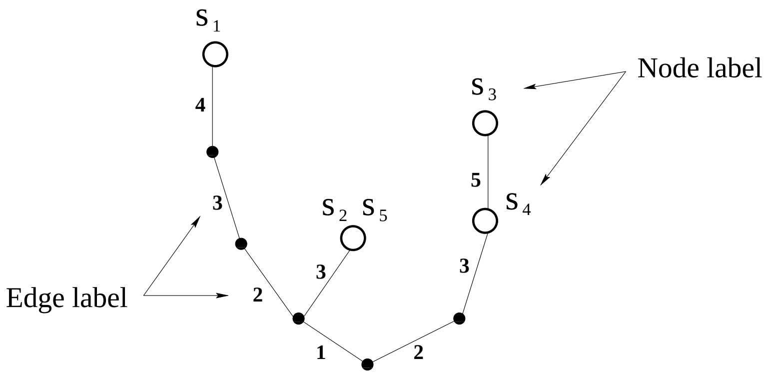

Figure 2: The lexicographic tree

Our data structure for the tree will have some variants, because the coding of the tree structure (parent, children) is a key point for the complexity analysis, and different choices for kinship will lead to different complexities in time and space.

The basic item in a tree is an edge together with the node it leads to, which we will call a ***nodge*** (see Figure 3) to avoid ambiguities. It holds the following information:

- A 0 (for instance) marking this structure as being a nodge.
- Two pointers at sets, the “first” and the “last” sets labelling this node. See below how the node labels in our tree are organized.
- The label $x \in X$ of its edge (say 0 for the root nodge).
- Pointers at the next and previous nodges with the same edge label (or NIL if this is the last nodge). The nodges with the same edge label are thus organized in a doubly linked list.
- A pointer at the parent nodge (if this nodge is not the root).
- Some more coding of the tree structure for kinship, to be described later.

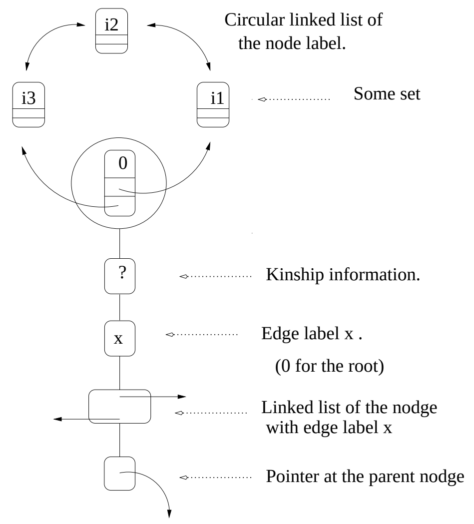

Figure 3: A nodge

Concerning kinship in the tree, we use the terms ***parent*** nodge, ***child*** of a nodge and ***brother*** nodges (for nodges that have the same parent nodge). There are two kinds of sets: the ones that are first label of their nodge, and the others, that are duplicates of the firsts. The duplicates are held in a doubly linked list, the entry of which we call D. For instance, if all the nodges of the tree are labelled by at most one S_i then the list D will be empty.

The information concerning the sets is stored in an $O(k)$-size array, with at entry i:

- An i marking this entry to be concerning S_i,
- Two pointers at the next and previous equal sets (or to the nodge it labels), as explained,
- Two pointers for the doubly linked list D if S_i is a duplicate (say two NIL's if not).

Finally we need an array of size $O(n)$ for the letters; the entry number x simply consists of a pointer at the first nodge of edge label x.

2.3 Operations

Note that if the sets are given by lists of elements, we first have to build the lexicographic tree from these lists. This can be done in linear time and space subject to the choice of the kinship coding : see the appendix for an algorithm based on partition refinement. The construction of the lexicographic tree from lists of elements will be called ***initialization***. We will check on section 2.4 that for three choices of kinship coding, the initialization remains linear in time and space.

We will need, as a base step for the operations we eventually want, to know how to delete a nodge. Given a nodge N, we have to delete it, and for every child to put it among N's brothers, with possible merging, which leads to recursive sub-tree merging. Therefore the low-level operation is the merging of a nodge into another one. An edge can be deleted in two ways: either through direct deletion (`delete-nodge`), or because of a merging into another edge (recursive calls to `merge-nodges`). Figure 4 shows the deletion of a nodge, and the resulting merging of nodges.

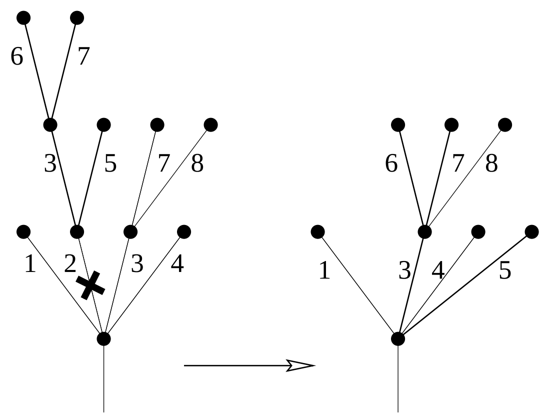

Figure 4: Deletion of a nodge

These two small procedures (algorithm 1 and 2) are self-describing. Only the "critical loop" of `merge-nodges` needs a more thourough description and a complexity analysis, since it strongly depends on the data structure to be described later. Everything else leads to $O(1)$ operations per deleted nodge, that is, $O(d)$ globally. The global operations are now easy to describe. (see appendix for the detailed procedures)

Operation Equality: Use the duplicates list D to know whether there are some equal sets, and if so, to find two equal ones;

Operation Set delete: Remove the set as a node label. Perhaps this creates a leaf without a node label, and a branch is to be deleted. Each nodge deleted this way (with a `delete-nodges` call) costs only $O(1)$ time, since it is childless;

Operation Element delete: While there are some nodges of the given edge label, call `delete-nodge` .

Algorithm 1: `delete-nodge(N)`

Data : A tree associated to a set of subset of X and N which points at a non-root nodge

Result: A new tree without the nodge N

begin

Let N' be a parent of N. ;

Remove N from the children of N'. ;

Removing a child *given by a pointer* will be an $O(1)$ operation;

`Merge-nodges(N,N')`.;

end

Algorithm 2: `Merge-nodges`(N, N')

Data : A lexicographic tree, a nodge N and a nodge N'

Result: The lexicographic tree where the subtree above N is merged with the subtree above N'

```
begin
    Merge the node labels of N into those of N' ;
    if both N and N' had labels then
        Add the set that is no longer first label to the duplicates list
    Remove N from its list of nodges with the same edge label ;
    foreach N1 child of N do
                                    /*Critical loop: to be fully described later*/;
        Let x be the edge label of N1;
        if N' has a child N'1 of edge label x then
            Merge-nodges(N1, N'1)
        else
            Insert N1 among the children of N'
end
```

2.4 Complexity analysis

Because of a possible long merging of subtrees and because of the repeated deletions of elements and sets, we are rather interested in amortized complexity for the operations. The core of the analysis being the "critical loop" of algorithm 2, we introduce the maximum time T spent on each child during this loop. We now give a time bound for any sequence of `element delete` and `set delete` operations.

Lemma 1 *The time complexity of all operations* `element delete` *and* `set delete` *is in $O(dT)$.*

The proof is given in the appendix.

This lemma is useful to evaluate the complexity of our two first variants of data structure: we simply compute T and check the initialization complexity.

We suppose that less than $O(d)$ operations `equality` are performed, each one being in $O(1)$). The three variants described next depend of the data structure chosen for kinship: array, binary search tree or linked list.

The time bounds correspond to the total of all operations `element delete` , `set delete` and `equality` whatever their order.

2.4.1 Time $O(d+n)$, Space $O(d+kn)$

It means linear time, but with extra space needed. Here we choose arrays to code kinship so that $T \in O(1)$, and the children of a nodge are linked, making a doubly linked list. We precisely use a somewhat redundant data structure for the kinship in the tree:

- An array A of size $k \times n$.
- A linked list of the unused lines of A, thus of size less than k.

- Each nodge N holds:
  - 4 more pointers, at the left brother and right brother which makes a linked list for brothers' structure, and at the first and last child (we already have the parent; of course NIL pointer are used for inexistent kin). The children of a nodge will not be kept sorted.
  - An index $a \in [\![0, k]\!]$, such that $a = 0$ if N has one child at most. If N has at least two children, then the a^{th} line of A consists of pointers at the children of N, that is, $A[a, x]$ points at the child of n of edge label x if any, or else contains NIL.

The main point is that there are at most k leaves, so at most k nodes have degree more than 1.

It is easy to delete or insert a given child (given by its edge label) in $O(1)$ thanks to these structures. In particular, changing a nodge with several children into a nodge with a single child as well as the inverse are $O(1)$ operations.

It remains to show that the extra data structures can be initialized in the required time and space; this is obvious if we are given an empty (filled with NIL) kn amount of space, in which to store A.

If not, we cannot afford the time to empty all that space, so we have to resort to the indirection trick [AHU71]. Each entry that is not void will hold, instead of its information, a pointer at a two-cell data structure, holding the information sought, and a pointer back at the entry. All information is thus held in extra data structures; this extra space is linear in the time complexity of the algorithm.

When accessing a given entry, we know it is void if it contains an illegal pointer (not pointing at the part of memory we are already using), or if it does not point at a pointer back.

Each read or write operation being in $O(1)$ time, we can thus simulate an empty amount of space.

2.4.2 Time and Space $O(n + d \log n)$

It is easy to find a structure for which $T \in O(\log n)$: for instance a binary search tree, with the edge label as a key. Each nodge simply needs two more pointers, left and right "sub-child". Figure 5 illustrates this relation between the parent nodge and its children.

Each child adds an $O(\log n)$ space, and that makes a global extra space $O(d \log n)$. Removing a child or inserting one with possible merging is easy in time $O(\log n)$.

2.4.3 Time $O(d + kn)$, Space $O(d + n)$

Here we choose to link the children of a nodge with a simple linked list, where the children are kept sorted, the smaller on the left. Each nodge has a pointer at his right brother and has other pointers at his first and last child. The initialization does not need extra time, as the children are appearing sorted (see the initialization algorithm in the appendix). This is the easiest algorithm, but the amortized cost is somewhat trickier to prove. The analysis relies on the ***holes***.

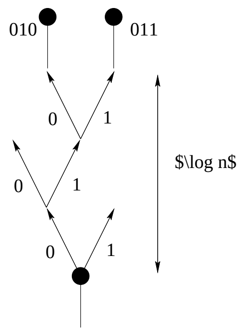

Figure 5: Kinship coding with binary search tree

Definition 2 *An* x-hole *in the lexicographic tree is a couple* (x, N) *such that* N *is a nodge with edge label* x_0*,* $x_0 \leq x$*,* N *has no son of edge label* x*, and* N *has a son of edge label* x_1*,* $x_1 > x$*.*

Removing a child given by a pointer is also easy in $O(1)$. Thus the global time complexity is $O(d+n)$ plus the time spent on the critical loop. We use the credits method to prove this time to be in $O(d+kn)$: we keep the invariant that each hole is credited with 1 credit, and each nodge with 2 credits plus 1 credit per descendant leaf.

Lemma 2 *This makes at most* $3d + kn$ *credits initially.*

The critical loop is a simple sweep of the children of N and N' at the same time, inserting or merging as you advance. All credits correspond to the same amount of time. We check precisely what credit is used for each operation, so that the invariant be maintained. As a result, the total time is bound by the initial number of credits (see appendix for the whole proof).

3 Application to graph problems

3.1 Pruning a graph

Let G be a graph with n vertices and m edges. The input is assumed to consist of the adjacency lists, it is of size $\Omega(n+m)$. To each $v \in [\![1, n]\!]$, we associate two sets, $S_{2v-1} = \mathcal{N}(v)$ and $S_{2v} = \mathcal{N}_c(v)$. We apply to our sets $(S_i)_{i \in [\![1,2n]\!]}$ the operations defined earlier. We will build the lexicographic tree, but also keep a copy of adjacency lists.

Deleting a vertex v from the current graph is equivalent to remove S_{2v-1}, S_{2v}, and every edge with label v, from the tree. This is exactly two `set delete` operations, and one `element delete` operation.

A `equality` operation is like scanning the graph for a couple of twins, so we simply lack the data structures necessary to detect pendant vertices. A way to do this is to create two arrays of size $O(n)$:

- An array Deg with, at entry i, $i \in [\![0, n-1]\!]$, the number of the first vertex of degree i, or a 0 if there are none. The idea is that the vertices with same degree are stored in a doubly linked list;

- An array with, at entry i, $i \in [\![1,n]\!]$, some information about the vertex i:
  - its degree,
  - a boolean value "dead or alive vertex", if we want the notion of current graph to be materialized,
  - and two vertices numbers for the doubly linked list of vertices with same degree.

When we delete a vertex, we can easily update these structures thanks to the adjacency lists. The total cost for updating these two arrays and adjacency lists is in $O(m+n)$. Algorithm 3 computes the "pruned graph". We suppose that the `equality` operation return $(0,0)$ if there are no equal sets.

Algorithm 3: Pruning a graph

```
Data   : A graph G
Result: The "pruned graph" of G
begin
    Initialize the data structures;
    repeat
        Let (i,j) be the output of an equality operation;
        if i ≠ 0 then  v ⟵ ⌈i/2⌉;
        else v ⟵ Deg[1] (No twins, try pendant vertices);
        if v ≠ 0 then
            Perform a set delete operation on S_{2v-1};
            Perform a set delete operation on S_{2v};
            Perform an element delete operation on v;
            Update the degree data structure;
    until v = 0;
end
```

Using the procedures of section 2, since $d = 4m + 2n$ and $k = n$, we have the following possible complexities for algorithm 3:

Time bound	$n+m$	$m \log n$	n^2
Space bound	n^2	$m \log n$	$n+m$

Remark 1 *The order chosen on the vertices may be important: for example, if this order happens to be a reversed pruning order, we might be deleting only leaf nodges, which would make the third algorithm linear in time too. We will thus be interested in finding a good order on the graph.*

3.2 Special cases with linear space and time

We will analyze again the complexity of the algorithm linear in space (the third variant) in the special case when we are not dealing with both true and false twins. We associate to each vertex v only one set S_v: $\mathcal{N}(v)$ if we just remove true twins, $\mathcal{N}_c(v)$ if we just remove false twins. To ensure linear time, we will need two slight modifications:

- Use of a special order on the vertices: a LexBFS order.
- When operation `equality` supplies us with $(i, j) \neq (0, 0)$, instead of deleting the vertex corresponding to i, we will delete the vertex corresponding to $\max(i, j)$. Each such vertex is guaranteed to have a smaller twin in the graph.

LexBFS means "lexicographic breadth-first search" [RTL76], and is best understood with the algorithm 4. See [HMPV97] for a simple way (based on partition refinement) to generate a LexBFS order.

Algorithm 4: Compute a LexBFS order on $G = (V, E)$, $\#V = n$

Data : a graph G where each vertex will be labelled with a "mark" (list of integers, kept sorted)

Result: a lexicographic order on G

begin

  Label each $v \in V$ with the singleton mark '$n + 1$';

  $i \longleftarrow 1$;

  while $i \neq n + 1$ **do**

    Choose v, an un-numbered vertex of minimal mark;

      (lexicographic order) as vertex number i;

    Mark every un-numbered neighbor of v with i;

    $i \longleftarrow i + 1$

end

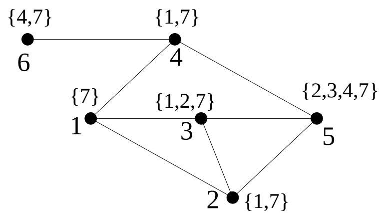

Figure 6: A lexBFS ordering of G with the mark of each vertex

Precomputing a LexBFS order on the graph is interesting for two reasons . First, it gives strong properties to the lexicographic tree. The main one concerns the removal of twins: for the `element deletion` operation, merging two nodges can be done in $O(1)$. And all the other operations also have good complexities (see appendix for the whole analysis). Secondly, after the removal of a pendant vertex or a vertex that has a smaller twin, we still have a LexBFS order on the graph.

Theorem 1 *The modified algorithm runs in time $O(m + n)$ (instead of n^2) if we just delete true twins or just delete false twins.*

Example: Bipartite Graphs. *In such graphs, two vertices are false twins iff they are pendant to each other. Thus we can use our algorithm (deleting pendant and true twin vertices) for an easy linear recognition of distance-hereditary bipartite graphs.*

4 Another application: complete bipartite cover

The "covering by complete bipartite subgraphs" problem is to find how many complete bipartite subgraphs we need (at the minimum) to cover every edge of a given bipartite graph. This problem is $\mathcal{NP}$-hard in the general case (Problem GT18 in [GJ79]); however [Mül96] showed that the complexity falls to $\mathcal{P}$ for some particular graph classes, like the distance-hereditary graphs. The algorithm solving this particular case is not studied further than being polynomial. The base step is to find a "bisimplicial" edge, which leads to a time complexity in $O(nm)$ on a bipartite graph with n vertices and m edges.

However, it is possible to find an algorithm that runs in linear time and space. We will again prune the input graph.

If we are given an optimal complete bipartite cover on the pruned graph, it is easy to find an optimal one on the original graph. Suppose we add a vertex to a bipartite graph with an optimal cover:

Adding u as a true twin of v: every complete bipartite graph in which v occurs is changed by duplicating v into u. This is still a complete bipartite graph and we still cover all the edges.

Adding u as a pendant vertex: we cannot extend an existing complete bipartite graph so that it includes the new edge ending by u, for that would require u to have another neighbor. Thus we have to add a new graph in the cover, for instance the star centered in v, the unique neighbor of u.

Thus the number sought is simply increased by one for every "pendant operation".

Computing the number of "pendant operations" during the pruning gives a simple algorithm to solve the problem on the class of bipartite distance-hereditary graphs.

5 Conclusion

It is unclear whether it is possible to modify the data structures so that the algorithms become linear both in time and space in the general case.

Anyway, representing a graph by a lexicographic tree gives good results for the "pruned graph" problem. Since its computation from adjacency lists is linear in time and space, this tree appears as an interesting tool for graph problems.

Besides twin vertices, we may wonder what properties of graphs can be read on this data structure, knowing that special orders on vertices and special classes of graphs have a strong influence on the structure of his tree.

References

[AHU71] A. V. Aho, I. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Welsey, 1974, exercise 2.12 on page 71.

[BM86] H. J. Bandelt and H. M. Mulder. Distance-hereditary graphs. *J. Combin. Theory*, Ser. B, 41:182–208, 1986.

[Dam97] G. Damiand. Quelques propriétés des graphes distances héréditaires. Master's thesis, Université de Montpellier II, LIRMM, 1997.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* Freeman, 1979.

[HM90] P. Hammer and F. Maffray. Completely separable graphs. *Discrete Applied Mathematics*, 27:85–99, 1990.

[HMPV97] M. Habib, R. McConnell, C. Paul, and L. Viennot. LexBFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theor. Comp. Sci.*, 1997. to appear.

[How77] E. Howorka. A characterization of distance-hereditary graphs. *Quart. J. Math. Oxford*, Ser. 2, 28:417–420, 1977.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[Mül96] H. Müller. On edge perfectness and classes of bipartite graphs. *Discrete Mathematics*, 149:159–187, 1996.

[RTL76] D. J. Rose, R. E. Tarjan, and G. S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. of Computing*, 5(2):266–283, June 1976.

Appendix

We give here some further details that might enlighten the corresponding sections, along with the proofs that did not fit in the main article.

2 Data structure

2.1 The general problem

2.2 General survey of the data structure: initializing the data structure.

We show that the lexicographic tree associated with the sets $S_1, S_2, \ldots, S_k$ can be constructed in time and space $O(d+n)$ (where d is the sum of the sizes of the sets and n is the number of distinct elements) plus an additional amount of time and space depending on the precise structure chosen to store the kinship information. This section rather describes a constructing scheme, from which the full initialization (array in Section 2.4.1, binary search tree in Section 2.4.2, simple linked list in Section 2.4.3) will easily follow.

Step 1 in $O(d+n)$ – Initialize the data structures with a lexicographic tree consisting of a root only, and sets pointing at this root (the lists in which the sets are stored are not initialized yet, of course). Sets will be kept pointing at the node where their associated path currently stops in the current truncated lexicographic tree.

Step 2 in $O(d)$ – For each i, for each "letter" x of S_i, generate the couple (i, x).

Step 3 in $O(d+n)$ – Bucket sort the couple (i, x) according to x (with an array of size n); we get a linked list of length $O(d)$, with non-decreasing x.

Step 4 in $O(d)$ – For each couple (i, x) taken in order from the list, check if the current node of S_i has a child with edge label x (which is maximum at this time). If so, S_i is made to point at that child; if not, such a child is created (this is the place where extra structures for kinship will come into account), and S_i is made to point at it.

After that step we have the lexicographic tree, as it is easy to see. Note that the lexicographic tree obviously has $O(d)$ edges. We still have to label it:

Step 5 in $O(k)$ – For each i, insert S_i as first (and last) node label if its node is without label, or else right after the first label. In the latter case insert S_i in the list of duplicates.

This completes the initialization of our data structure.

2.3 Operations

We give here a more detailed description of the three basic operations `equality`, `set delete`, and `element delete`.

Operation `equality` in $O(1)$

Data : The lexicographic tree, with the doubly linked list of duplicates D

Result: (i, j) such that $S_i = S_j$ if there are equal sets, or else $(0, 0)$.

begin

  if $D = NIL$ **then** all sets are distincts;

  else

    let i be the set at which D points, we can find S_j a set equal to S_i in time $O(1)$ thanks to the node label structure.

end

Operation `set delete` in $O(1)$ per deleted edge

Data : i a set index

begin

  Remove i from its node labels circular list;

  if *i was not the only node label* **then**

    if *i was the first node label* **then**

      Let j be the next node label;

      Remove j from the duplicates list   /*j is the new first label*/;

    else

      Remove i from the duplicates list   /*i no longer exists*/;

  else

    Let N be the nodge holding i as node label   /*We know N*/;

    while *N is childless, without node label, and is not the root* **do**

      Let N' be the parent of N /*N is useless now;*/;

      `delete-nodge`(N) /*this is $O(1)$ since there is no child*/;

      $N \longleftarrow N'$;

end

Operation `element delete`.

Data : x the element of X to be deleted

begin

  while *$N \neq NIL$, where N is the first nodge of edge label x* **do**

    `delete-nodge`(N);

end

2.4 Complexity analysis

Lemma 1 *The time complexity of all operations* `element delete` *and* `set delete` *is in* $O(dT)$.

Proof: At most $O(d)$ operations can be performed; each of them needs an $O(1)$ time, plus, for each disappearing edge:

- $O(1)$, for operation `element delete`, or
- $O(1)$ plus T for each child (at time of deletion), for operation `set delete`.

Since we have $O(d)$ edges initially (and $T \geq 1$), we simply have to show that the sum of the number of children, at time of deletion, of the deleted nodges through operation `element delete`, is in $O(d)$.

Let N be a nodge being deleted after an operation `element delete` performed on a letter x_0. Let N_0 be the nodge responsible for the deletion of N: a first call to `delete-nodge`N_0 resulted in a subsequent call to `merge-nodge`N,N', either because $N = N_0$, or because N is part of a subtree rooted in N_0 being merged into a subtree rooted in the parent of N_0.

Let $N_1, \ldots, N_r$ be the children of N (if N has no child, there is nothing to prove). Let S_i be a set labelling a leaf that is a descendant of N_i. The path associated with S_i thus goes through N_0, corresponding to a letter x_0 of S_i (the current set). This letter corresponds to a letter x_0 of the original set S_i. We associate in this way some letter of the input with every child of a nodge being deleted.

Since there are less than d such letters, it is sufficient to show that each letter can be picked up this way only once at most. This is obvious, since after the first time a given letter x_0 in a given S_i has been picked up, that occurence of x_0 in S_i has disappeared from the lexicographic tree, and thus cannot be picked up again after the deletion of N_0. And while N_0 is being removed, a given descendant edge will be merged at most once. $\square$

2.4.1 Time $O(d+n)$, Space $O(d+kn)$

The following two procedures are rather straightforward; we nevertheless give here a more formal description than in the main text.

Remove N from the children of N' in $O(1)$

begin
 Update the pointer structure;
 let a be the index of N';
 if $a \neq 0$ **then**
  let x be the edge label of N;
  $A[a, s] \longleftarrow NIL$;
  if N' *has only one child left (last child is first child)* **then**
   let t be the edge label of N';
   $A[a, t] \longleftarrow NIL$ /*this line is free now*/;
   add a to the list of free lines;
   set the index of N' to be 0;
end

```
Critical loop on nodges N and N', with T ∈ O(1) per child
while N_1 ≠ NIL, where N_1 is the first child of N, do
    Remove N_1 from the children of N /*This eventually frees the line of N in A*/;
    Let x be the edge label of N_1, a be the index of N';
    if a ≠ 0 then
        if N'_1 = A[a, s] ≠ NIL then merge-nodges(N_1,N'_1);
        else  A[a, s] ⟵ N_1;
    else
        if N' has one child only then
            Let a be the first free line in the list      /*the list cannot be empty*/;
            Remove a from the list, let it be the index of N';
            Update the a^th line of A                       /*two entries to be filled*/;
    Insert N_1 as first child of N' with the pointer structure;
```

2.4.2 Time and Space $O(k + d \log n)$

2.4.3 Time $O(d + kn)$, Space $O(d + n)$

We now give the proof of the lemma counting credits, we describe the critical loop for the kinship data structure and how the credits are used to perform the elementary operations.

Lemma 2 *There are at most $3d + kn$ credits initially.*

Proof:
$2d$ **credits** are given to each nodge.

kn **credits** are sufficient to supply each hole with 1 credit. The easiest way to see this is perhaps to supply each leaf with n credits (there are indeed less than k leaves), and then let each hole (x, N) pick up its credit from a leaf that lies "to its right" (a descendant of N_1, the son of N of edge label x_1, for some $x_1 > x$). We could refer to the credit in question as the credit number x of the leaf.

This credit can be picked up only once, since nodes that are descendant of N_1 have no x-hole, and nodes that are ascendants of N have their x-holes to the right of this leaf.

d **credits** are sufficient to supply each nodge with 1 credit per descendant leaf: each leaf supplies 1 credit to each nodge on the way to the root. □

The critical loop is a simple sweep of the children of N and N' at the same time, inserting or merging as one goes along; we have to check precisely what credit is used for each operation.

A hole is possibly created at a call to `delete-nodge`(N): the first credit of N is transfered to that hole. Then comes the critical loop.

Critical loop of `merge-nodges`(N,N')

Initialize a pointer p at the first child of N;
if N' *is the parent of* N **then**
  Let p' point at the right brother of N /*First call to `merge-nodges`*/;
else
  Let p' point at the first child of N' /*Subtree merging*/;
/*p' points at the first relevant child of N'; charge: second credit of N*/;
while $p \neq NIL$ *and* $p' \neq NIL$ **do**
  Let x and x' be the edge label of p and p' respectively;
  if $x < x'$ **then**
    Insert p as the left brother of p' (update some pointers);
    $p \leftarrow$ right brother of p;
    /*Charge: credit of the x hole of N' that is now filled*/;
  else
    if $x = x'$ **then**
      `merge-nodges`(p, p') /* Only the cost of the call is currently taken into account*/;
      $p \leftarrow$ right brother of p;
      $p' \leftarrow$ right brother of p';
      /*Charge: first credit of p (p leaves no hole after it has disappeared)*/
    else
      $p' \leftarrow$ right brother of p' /*"Jump". Charge: on the x' hole of N*/ (p' has been carefully initialized so that x' be a hole);

/*If $p = NIL$ it is over*/;
if $p' = NIL$ **then** Insert the remaining children of N after the last child of N';
/*Charge: see below*/;

The last line treats the case when we have to insert the last children of N after the last child of N'; we need a constant time for each remaining child, mainly to update the parent pointer. These credits are charged on the nodge originally deleted (`delete-nodge`), on the "leaves budget", since each such child leads to at least one such leaf. If this nodge happens to be N, then N' is the parent of N; thus N' has gained no leaf, and the invariant is preserved (note that if N' has gained holes, they have come with their credits). If not, N' might have gained some leaves, but in this case the "leaves budget" of N is left untouched, and can be transferred to N'. This restores the invariant, and completes the proof.

3 Application to a graph problem

3.1 Pruning a graph

3.2 Special cases with linear space and time

Theorem 1 *The modified algorithm runs in time $O(m + n)$ if we do not delete both true and false twins.*

Proof: We will separately study the case when we are deleting a pendant vertex (using the special order), a true or a false twin (using the smaller twin). We only have to take care that the number of attempts we need in the critical loop to find where to put a child, is less than a constant, since this makes $T \in O(1)$, and we can apply the complexity lemma. We will call an unsuccessful try a **jump** (the "jump" line in the algorithm).

A subtree containing only one node label will be called a **branch**, and a nodge generating a branch as a subtree a **branch nodge**. This kind of nodge looks easy to remove: at each depth of the recursion, we simply have one child to deal with, either to be inserted (which ends the process), or to be merged (deeper into the recursion). We will call **x-nodge** a nodge with x as edge label.

Deleting a pendant vertex:
Let u be a pendant vertex to be deleted, u' be its neighbor. After the set S_u has been removed, there is only one edge with label u, and this edge leads to only one labelled node, the leaf labelled u': the only u-nodge is a branch nodge.

The connected components are independent in the lexicographic tree, since they are visited one at a time by a LexBFS order, and since they lead to no crossed neighborhood relationships: they simply share the root as a common nodge, and otherwise behave as distinct lexicographic trees. Thus we can make the assumption that u belongs to the first connected component (this simplifies the notations). Things are different depending whether u was chosen first:

- **u is number one, open neighborhood lexicographic tree.** Then u' is 2, and the neighbors of u' are $3, 4, \ldots, k$. When the nodge 1 is deleted, its child (labelled 3) jumps over 2 to find its place; then (in the case of a merging) 3's child 4 finds his place at first try since it is leftmost; and the same argument applies to further descendants in the case of further mergings.

- **u is number one, closed neighborhood lexicographic tree.** In this case the child of the nodge 1 is a nodge 2, so we are directly in the situation when each child is leftmost.

- **u is not the first vertex of its connected component.** Then $u > u'$ (its neighbor must be visited first), and since u has u' as only mark, u is numbered after every neighbor of every vertex $v < u'$ has been numbered. We show that there cannot be a jump in this case: suppose β' jumps over β somewhere. Then $\beta > u$, and β is not a neighbor of u': this means that β is not marked by a vertex $v \leq u'$. But β' is a neighbor of u': thus $\beta' < \beta$. So β' cannot jump over β.

Thus each child needs at most 2 attempts.

Deleting a twin:
Let $u' < u$ be two twins, such that u is to be deleted, let N be an u-nodge (after S_u has been removed). We prove that N is rightmost:

Suppose N is not rightmost; let v be a node label of a descendant of N, and w be a node label of a descendant of a right brother of N. w cannot be a neighbor of u (there is no u-nodge on the path to the root), and thus is not a neighbor of u's twin u'.

If v is not u', v is a neighbor of u ($v \neq u$ since S_u has been removed), so it must also be a neighbor of u': there is an u'-nodge that is an ascendant of N. This nodge is also in the path to w, which is not a neighbor of u'. The only possibility is that we are in the closed neighborhood lexicographic tree, and that $w = u'$. But then u' is a false twin of u, so there should be a u-nodge on the way to the root, and there is not.

If v is u', u' is a neighbor of u, so we are in the closed neighborhood lexicographic tree. But then there should be an u'-nodge on the way to the root, and there is not.

Thus N is rightmost, and for each child the right place is always found at the first try. □