Finding a Minimum Independent Dominating Set in a Permutation Graph

Mikhail J. Atallah

Glenn Manacher

J. Urritia

Report Number:
85-514
FINDING A MINIMUM INDEPENDENT DOMINATING SET IN A PERMUTATION GRAPH

Mikhail J. Atallah
Glenn K. Manacher
J. Urrutia

CSD-TR-514
April 1985
Revised
Finding a Minimum Independent Dominating Set in a Permutation Graph

Mikhail J. Atallah†
Glenn K. Manacher*
J. Urrutia†

Abstract

We give an $O(n \log^2 n)$ time algorithm for finding a minimum independent dominating set in a permutation graph. This improves on the previous $O(n^2)$ time algorithm known for solving this problem [4].

† Dept of Computer Sci., Purdue Univ., West Lafayette, IN 47907. Research supported by ONR Contract N00014-84-K-0502 and NSF Grant DCR-8451393, with matching funds from AT&T.
* Dept of Mathematics, University of Illinois, Chicago, IL 60614.
† Dept of Computer Science, University of Ottawa, Ottawa, Ontario, Canada.
1. Introduction

Let \( \Pi \) be a permutation on the set \( I_n = \{1,2, \ldots, n\} \). Then the permutation graph \( G(\Pi) \) is the undirected graph with vertex set \( V(G) = I_n \) such that vertex \( i \) is adjacent to vertex \( j \) in \( G(\Pi) \) if and only if \( i < j \) and \( \Pi^{-1}(i) > \Pi^{-1}(j) \). For any undirected graph \( G \), a subset \( S \) of the vertex set \( V(G) \) is called a dominating set if for every \( u \in V(G) \) there exists \( v \in S \) such that \( u \) is adjacent to \( v \). Set \( S \) is independent if no two vertices in \( S \) are adjacent. In [4] the problem of finding a minimum independent dominating set in a permutation graph (from now on called MIDS) was studied. In the same paper an \( O(n^3) \) time algorithm to solve the MIDS was presented. Our main result is an \( O(n \log^2 n) \) time algorithm for MIDS.

Given a sequence \( \alpha = a_1a_2 \cdots a_n \) of numbers, a subsequence of \( \alpha \) is a sequence \( \beta = a_{i_1}a_{i_2} \cdots a_{i_k} \) such that \( i_1 < i_2 < \cdots < i_k \). If, in addition, \( a_{i_1} < a_{i_2} < \cdots < a_{i_k} \), then we say that \( \beta \) is an increasing subsequence of \( \alpha \). An increasing subsequence of \( \alpha \) is maximal if it is not a proper increasing subsequence of another increasing subsequence of \( \alpha \). A maximum increasing subsequence is one of maximum length. Note that a maximum increasing subsequence is also maximal, but that a maximal increasing subsequence may not be maximum. For example, in the sequence 2,1,4,5,3 the increasing subsequence 1,3 is maximal but not maximum (for this example the length of a maximum increasing subsequence is three, e.g. 2,4,5).

Section 2 of this paper points out that MIDS can be viewed as the problem of computing a shortest maximal increasing subsequence (from now on called SMIS) of a sequence of \( n \) numbers. Section 3 then gives an \( O(n \log^2 n) \) time solution to SMIS. As a consequence, an \( O(n \log^2 n) \) time algorithm to solve the MIDS problem in permutation graphs is obtained, which compares favourably with the \( O(n^3) \) algorithm presented in [4]. It is interesting to notice that the problem of computing the maximum increasing subsequence of a sequence of numbers has been widely studied [2,3], while the problem of finding the shortest maximal increasing subsequence has not. Moreover, the known \( O(n \log n) \) algorithms for solving the former problem cannot be modified to solve the SMIS problem. So in spite of their apparent similarity, the two problems...
seem to be quite different. Our \(O(n \log^2 n)\) time algorithm for SMIS uses techniques and data structures that were originally developed to solve problems in Computational Geometry. In the weighted versions of the MIDS and SMIS problems, a non-negative weight is associated with every element of \(I_n\) or \(\alpha\). The problems then become those of finding the minimum weight independent dominating set and minimum weight maximal increasing subsequence, respectively. The algorithms we obtain for the unweighted cases can be easily modified to solve the weighted ones, so we shall no longer concern ourselves with the weighted cases.

2. Minimum Independent Dominating Sets in Permutation Graphs

The main objective of this section is to prove the following result.

Theorem 1. Given permutation \(\Pi\), the problem of finding a MIDS in the permutation graph \(G(\Pi)\) is reducible, in linear time, to the problem of finding a SMIS of a sequence of numbers.

Before proving Theorem 1, we shall obtain some easy properties of permutation graphs.

Lemma 1. Let \(I = \{i_1, i_2, \ldots, i_k\}\) be a subset of \(I_n\); \(i_1 < i_2 < \cdots < i_k\). Then \(I\) forms an independent set in \(G(\Pi)\) if and only if \(\Pi^{-1}(i_1) < \Pi^{-1}(i_2) < \cdots < \Pi^{-1}(i_k)\).

Proof. Follows immediately from the definition of \(G(\Pi)\). \(\square\)

Lemma 2. Let \(I = \{i_1, i_2, \ldots, i_k\}\) be an independent set in \(G(\Pi)\), with \(i_1 < i_2 < \cdots < i_k\). Then \(I\) is a dominating set in \(G(\Pi)\) if and only if the sequence \(\beta = \Pi^{-1}(i_1)\Pi^{-1}(i_2) \cdots \Pi^{-1}(i_k)\) forms a maximal increasing subsequence of \(\alpha = \Pi^{-1}(1)\Pi^{-1}(2) \cdots \Pi^{-1}(n)\).

Proof. First observe that Lemma 1 implies that sequence \(\beta\) is increasing, so that it suffices to prove that \(I\) is dominating iff \(\beta\) is maximal.

For the "if" part of the proof, suppose that \(\beta\) forms a maximal increasing subsequence of \(\alpha\). To prove that \(I\) forms a dominating set in \(G(\Pi)\), note that otherwise there is a vertex \(j \in I\) such that \(I \cup \{j\}\) is also an independent set. Therefore (by Lemma 1) we can insert \(\Pi^{-1}(j)\) in \(\beta\) obtaining a new increasing subsequence of \(\alpha\) which properly contains \(\beta\), contradicting the maximality of \(\beta\).
For the "only if" part of the proof, assume that \( I \) is dominating in \( G(\Pi) \). If \( \beta \) is not maximal, then there exists a \( j \in I_\pi \) such that insertion of \( \Pi^{-1}(j) \) in \( \beta \) results in a \( \beta \) which is an increasing subsequence of \( \alpha \), e.g. \( \beta = \Pi^{-1}(i_1) \cdots \Pi^{-1}(i_{k-1}) \Pi^{-1}(j) \Pi^{-1}(i_{k-1}) \cdots \Pi^{-1}(i_k) \) where \( \beta \) is increasing and \( i_1 \cdots \cdots i_{k-1} < j < i_k < \cdots < i_k \). By Lemma 1, this implies that \( I \cup \{j\} \) is independent, which contradicts the fact that \( I \) is dominating in \( G(\Pi) \). \( \square \)

Proof of Theorem 1. By Lemma 2 every independent dominating set in \( G(\Pi) \) generates a maximal increasing subsequence of \( \Pi^{-1}(1) \Pi^{-1}(2) \cdots \Pi^{-1}(n) \), and vice-versa. Then finding a minimum independent dominating set in \( G(\Pi) \) is equivalent to finding a minimum maximal increasing subsequence in \( \Pi^{-1}(1) \Pi^{-1}(2) \cdots \Pi^{-1}(n) \). \( \square \)

3. Finding Shortest Maximal Increasing Subsequences

This section gives an \( O(n \log^2 n) \) time algorithm for SMIS. Before giving the algorithm, we need some preliminaries.

Let \( P \) be a set of points in the plane. We use \( X(p) \) and \( Y(p) \) to denote the \( x \) and (respectively) \( y \) coordinates of a point \( p \). Point \( p_i \) is said to dominate \( p_j \) iff \( X(p_i) > X(p_j) \) and \( Y(p_i) > Y(p_j) \). We use \( \text{DOM}(p_i) \) to denote the subset of points in \( P \) that are dominated by point \( p_i \); i.e. \( \text{DOM}(p_i) \) contains the points of \( P \) that are below and to the left of \( p_i \). A point of \( P \) is a maximum iff no other point of \( P \) dominates it. From now on we use \( \text{MAX}(P) \) to denote the set of maxima of \( P \).

Our algorithm makes use of the following elegant result of Overmars and Van Leeuwen:

There exists a data structure for dynamically maintaining the maxima of a set of points in the plane, such that insertions and deletions take time \( O(\log^2 n) \) per operation. Such an augmented tree structure (as it is called in [5]) takes \( O(n) \) storage space, and can initially be created in time \( O(n \log n) \). At any time, the maxima are available at the root, in a concatenateable queue "attached" to the root. An augmented tree structure can also support \( \text{SPLIT} \) and \( \text{CONCATENATE} \) operations in time \( O(\log^2 n) \) per operation (even though this is not mentioned explicitly in reference
it easily follows from it). In other words, if the points are stored in the augmented tree structure according to (say) their \( y \)-coordinate, then a \textit{SPLIT} operation about any horizontal line \( y = y_0 \) can be implemented in time \( O(\log^2 n) \). Such a \textit{SPLIT} operation results in two augmented tree structures: One for the points above the horizontal line, and one for those below it. A \textit{CONCATENATE} operation also takes \( O(\log^2 n) \) time and has the reverse effect of a \textit{SPLIT}. In the context of this paper, every point will have a \textit{label} associated with it, and we will need to maintain the smallest-labeled maximum at the root of the augmented tree structure (more precisely, at the root of the concatenable queue attached to the root). It is not hard to show that this can be done without losing the \( O(\log^2 n) \) time-per-operation performance (this is done using standard data structure techniques, such as those described in reference [1]).

We now have all the ingredients which we use in our algorithm.

3.1. The algorithm

Let \( a_1 \ldots a_n \) be the input sequence. Let \( \beta_i \) be a shortest maximal increasing subsequence of \( a_1 \ldots a_i \) which ends with \( a_i \). Let \( \text{label} \ (i) \) be the length of \( \beta_i \). If \( \text{label} \ (i) > 1 \), let \( \text{predecessor} \ (i) \) be the index of the predecessor of \( a_i \) in \( \beta_i \), i.e. \( \beta_i \) ends with \( a_{\text{predecessor} \ (i)} a_i \). If \( \text{label} \ (i) = 1 \) then \( \text{predecessor} \ (i) = 0 \).

Algorithm \textbf{MINMAX}

\textbf{Input}: Sequence \( a_1 \ldots a_n \)

\textbf{Output}: A minimum-length maximal increasing subsequence of \( a_1 \ldots a_n \)

\textbf{Method}: The algorithm sets \( \text{label} \ (1) = 1 \) and \( \text{predecessor} \ (1) = \emptyset \). Next, the algorithm creates points \( p_1 \ldots p_n \) in the plane, where \( p_i = (i, a_i) \), \( 1 \leq i \leq n \). Then the algorithm sweeps a vertical line \( L \) from left to right, maintaining the maxima of the set of points to the left of \( L \) in an augmented tree structure \( T \). When the left-to-right sweeping line \( L \) encounters a point \( p_i \), the following steps 1-3 are taken:

1) The algorithm splits \( T \) about the horizontal line \( y = Y(p_i) \), obtaining two augmented tree structures \( T_{up} \) and \( T_{down} \). Note that \( T_{down} \) contains the set \( \text{MAX} (\text{DOM}(p_i)) \) in a
concatenable queue attached to its root, and the smallest-labeled point of $\text{MAX} (\text{DOM} (p_i))$
is attached to the root of this concatenable queue.

2) If $\text{MAX} (\text{DOM} (p_i))$ is empty then the algorithm sets $\text{label} (i) := 1$ and $\text{predecessor} (i) := \emptyset$.Otherwise it sets $\text{predecessor} (i)$ equal to the index $j$ of the smallest-labeled point $p_j$ of$\text{MAX} (\text{DOM} (p_i))$, then it sets $\text{label} (i) := \text{label} (j) + 1$.

3) Rebuild $T$ by concatenating $T_{up}$ and $T_{down}$, then insert $p_i$ in $T$.

After the line $L$ sweeps past $p_n$ (the rightmost of the $p_i$'s), the algorithm chooses asmallest-labeled point in $\text{MAX} ([p_1, \cdots, p_n])$, let $p_k$ be this point. The algorithm then sets $\beta := a_k$and then, so long as $\text{predecessor} (k) \neq \emptyset$, it does $\beta := a_{\text{predecessor} (k)} \beta$ followed by$k := \text{predecessor} (k)$. When $\text{predecessor} (k) = \emptyset$ the algorithm outputs $\beta$.

End of Algorithm MINMAX

That $\text{label} (i)$ and $\text{predecessor} (i)$ are computed correctly by the algorithm follows from thedefinitions of these two functions. That the $\beta$ produced by the algorithm is the desired subsequence follows from the definitions of the $\text{label}$ and $\text{predecessor}$ functions and the observation that any maximal increasing subsequence must end with an $a_k$ such that $p_k \in \text{MAX} ([p_1, \cdots, p_n])$.

That the algorithm runs in $O(n \log^2 n)$ time is an immediate consequence of the fact that each of the operations $\text{INSERT}$, $\text{SPLIT}$, $\text{CONCATENATE}$ takes $O(\log^2 n)$ time in an augmented tree structure, and that the smallest-labeled maximum is readily available at the root.

This completes the proof of the following:

**Theorem 2.** Given a sequence of integers $a_1 \cdots a_n$, it is possible to find a shortest maximal increasing subsequence in time $O(n \log^2 n)$ and space $O(n)$.

Using theorems 1 and 2, we have:

**Theorem 3.** Finding a minimum independent dominating set in a permutation graph can be done in $O(n \log^2 n)$ time and $O(n)$ space.
4. Conclusion

In this paper we gave a $O(n \log^2 n)$ time algorithm to find a shortest maximal increasing subsequence of a sequence of $n$ numbers. Using this algorithm, we can find a minimum independent dominating set of a permutation graph in $O(n \log^2 n)$ time, an improvement over the previously known $O(n^2)$ time algorithm. These results can be easily extended to the weighted cases.

References


