THE VERTEX SEPARATION AND SEARCH NUMBER OF A GRAPH

J. A. Ellis*, I. H. Sudborough† and J. S. Turner‡

* Department of Computer Science
University of Victoria
Victoria, British Columbia V8W 3P6, Canada

† Computer Science Department
University of Texas at Dallas
Richardson, Texas 75083, U S A

‡ Computer Science Department
Washington University
St. Louis, Missouri 63130, U S A

Running head:

Vertex Separation and Search Number

Mailing address for proofs etc.:

John A. Ellis
Department of Computer Science
University of Victoria
P.O. Box 3055
Victoria
British Columbia V8W 3P6
CANADA

Abstract

We relate two concepts in graph theory and algorithmic complexity, namely the *search number* and the *vertex separation* of a graph. Let $s(G)$ denote the search number and $vs(G)$ denote the vertex separation of a connected, undirected graph G. We show that $vs(G) \leq s(G) \leq vs(G) + 2$ and we give a simple transformation from G to G' such that $vs(G') = s(G)$.

We characterize those trees having a given vertex separation and describe the smallest such trees. We also note that there exist trees for which the difference between search number and vertex separation is indeed 2. We give algorithms that, for any tree T, compute $vs(T)$ in linear time and compute an optimal layout with respect to vertex separation in time $O(n \log n)$.

Vertex separation has previously been related to *progressive black/white pebble demand* and has been shown to be identical to a variant of search number, *node search number*, and to *path width*, which has been related directly to *gate matrix layout cost*. All these properties are known to be computationally intractable. For fixed k, an $O(n \log^2 n)$ algorithm is known which decides whether a graph has path width at most k.

LIST OF SYMBOLS

Σ Greek upper case sigma
O Italic upper case oh
> greater than
< less than
≤ less than or equal to
≥ greater than or equal to

1. Introduction

We consider connected, undirected graphs. They may have multiple edges or loops, which can affect the search number, but not the vertex separation.

A *separator* of an undirected graph is a set of vertices, in the case of a vertex separator, or a set of edges, in the case of an edge separator, whose removal separates the graph into disconnected sets of vertices. Small separators that divide the graph into roughly equal components were used to describe good VLSI layouts in [Leiserson 1980] and to describe good divide and conquer algorithms in [Lipton 1980]. Theorems which guarantee the existence of small separators for planar graphs and graphs of fixed genus have been described in [Lipton 1979] and in [Philipp 1980].

Lengauer [Lengauer 1981] called this a static definition of separator and went on to define a ‘‘vertex separator game’’. We consider the same concept as Lengauer but describe it in terms of linear layouts. Let $G = (V, E)$ be a connected, undirected graph. A *linear layout*, or simply a *layout*, of G is a one to one mapping $L : V \rightarrow \{1, 2, ..., |V|\}$. Let a *partial layout* of G be a one to one mapping L from a subset V' of V to the set of integers $\{1, 2, ..., |V'|\}$. For a partial layout L, let $V_L(i) = \{x \mid x \in V$ and there exists $y \in V$ such that $\{x, y\} \in E$ and $L(x) \le i$ and either $L(y) > i$ or $L(y)$ is undefined$\}$. The *vertex separation* of G with respect to L, denoted by $vs_L(G)$, is defined by $vs_L(G) = \max \{ |V_L(i)| \mid 1 \le i \le |\mathrm{domain}(L)| \}$ and the vertex separation of G is defined by $vs(G) = \min \{ vs_L(G) \mid L$ is a layout of all of $G\}$. These concepts are illustrated in Figure 1.1.

****

Insert Figure 1.1 here

****

For a partial layout L, whose domain is some subset V' of all the vertices, and a positive integer $i \le |V'|$, the partial layout L_i is the mapping that agrees with L on the vertices $\{L^{-1}(1), L^{-1}(2), ..., L^{-1}(i)\}$ and is undefined elsewhere. An edge $e = \{x, y\}$ is *dangling* in a partial layout L, if x is in the domain of L and y is not. A vertex x is *active* in a partial layout L if it is incident to a dangling edge and in domain(L). The set of active vertices is denoted A_L. The active vertices and dangling edges of a partial layout are illustrated in Figure 1.2.

****

Insert Figure 1.2 here

****

The concept of *search number* was introduced by Parsons [Parsons 1976] [Parsons 1978]. Informally, the search number of a graph G, denoted by $s(G)$, is the minimum number of searchers necessary to guarantee the capture of a fugitive who can move with arbitrary speed about the edges of the graph.

A *search step* is one of the following operations: (a) the placing of a searcher on a vertex, (b) the movement of a searcher along an edge, (c) the removal of a searcher from a vertex. A *search sequence* is a sequence of search steps. Initially, all the edges of the graph are *contaminated*. We say that an edge $e = \{x, y\}$ becomes *clear* if either there is a searcher on x and a second searcher is moved from x to y or there is a searcher on x, all edges incident to x except e are clear, and the searcher on x is moved along e to y. A clear edge e could become contaminated again by the movement or deletion of a searcher which results in a path without searchers from a contaminated edge to e. A *search strategy* for a graph is a search sequence that results in all edges being simultaneously clear. The search number of a graph is the minimum number of searchers for which a search strategy exists.

LaPaugh [LaPaugh 1983] proved that recontamination can not help. That is, for every graph, there is a search strategy which does not recontaminate any edge and which uses the minimum number of searchers. A search strategy that does not recontaminate any edge will be called a *progressive search strategy*. The search number problem is then clearly in **NP** since it is easy to see a non-deterministic, polynomial time solution to the progressive search problem.

Meggido et al. [Megiddo 1988] showed that determining the search number of a graph is NP-hard, which implies it is **NP**-complete because of LaPaugh's result. They also showed that the search number of a tree can be determined in linear time. It is known that, for any graph G with maximum vertex degree 3, $s(G)$ is identical to the cutwidth of G [Makedon 1983]. Hence the search number problem has practical value, as well as theoretical interest, since finding the cutwidth of a graph is important in some VLSI layout applications [Leiserson 1980].

A variation on search number, called *node search number*, was defined by Kirousis and Papadimitriou [Kirousis 1986]. For node search number, a searcher does not need to move along an edge, looking along it is sufficient to catch the fugitive. The authors showed that node search number is identical to vertex separation $+ 1$.

Vertex separation is related, directly or indirectly, to several other important graph properties. Kinnersley [Kinnersley 1990] showed that vertex separation is identical to *path width*, a most important measure of graph structure in the theories of Robertson and Seymour which have been applied to fixed parameter algorithmic problems by Fellows and Langston [Fellows 1987], [Fellows 1988] and [Fellows 1989]. In [Fellows 1989] it is shown that *gate matrix layout cost* is equal to path width $+ 1$, for all graphs.

Of the many pebble games, the *black/white pebble game* models the space requirements of non-deterministic, straight line programs. The *progressive* black/white pebble game does not allow repebbling. Lengauer [Lengauer 1981] showed that the vertex separation and progressive black/white pebble problems are polynomially reducible one to the other.

Algorithms have been given for determining the vertex separation (tree width) of unrestricted graphs when k is taken to be constant. An algorithm due to Sudborough et al. [Ellis 1983, 1987] requires time $O(n^{2k^2+4k+2})$. That technique was also applied to the fixed parameter version of the hyper-graph cut width problem, [Miller 1990]. The algorithm described by Arnborg et al. [Arnborg 1987] requires time $O(n^{k+2})$. As a consequence of the work of Robertson and Seymour, we know there exists an $O(n^2)$ algorithm for the problem, although that theory does not show us how to construct an algorithm. However, the techniques described by Fellows and Langston [Fellows 1989], can be used to construct such an $O(n^2)$ algorithm, although it does have a very large constant of proportionality. The algorithm described by Bodlaender et al. [Bodlaender 1991] requires time $O(n \log^2 n)$ and has a constant which is only singly exponential in k.

In Section 2 we show the relations between vertex separation and search number. In Section 3 we give a recursive definition of the vertex separation of a tree in terms the vertex separations of its subtrees, a linear time algorithm for computing the vertex separation of trees, and a $O(n \log n)$ algorithm for computing an optimal layout. We also characterize trees of a particular vertex separation, describe the smallest such trees and note that there exist trees whose vertex separation differs from their search number by 2, i.e., the maximum possible.

2. Relationships Between Vertex Separation and Search Number

In this section we show that the search number of a graph is in the range $vs(G)$ through $vs(G)+2$. We then show a simple transformation from G to G' such that $s(G)=vs(G')$.

2.1. Relating Vertex Separation to Search Number

Theorem 2.1 Let $G=(V,E)$ be a graph. Then $vs(G) \le s(G) \le vs(G)+2$.

Proof From Lemmas 2.1 and 2.3 below. □

Lemma 2.1 $vs(G) \le s(G)$.

Proof We show how a layout, L, can be constructed from a search strategy, S, so that the vertex separation of L is no greater than the number of searchers used by S. The argument requires that the strategy be progressive, and so relies on LaPaugh's result [LaPaugh 1983] that there exist optimal, progressive strategies.

At any point in the execution of a strategy, let V_L be the set of vertices that are unoccupied by a searcher and incident to no contaminated edge. Let V_S be the set of vertices currently occupied by one or more searcher, and let V_R be the set of vertices remaining, i.e. those that are unoccupied and incident to some contaminated edge. We note that there can be no edge connecting a node in V_L to a node in V_R, else the node in V_L would, by definition of contamination, be incident to a contaminated edge. Hence, at all points in the strategy, the set V_S separates the vertices in V_L from those in V_R.

We want to consider strategies in which vertices pass only from V_R to V_S and from there to V_L. We note that no vertex can pass from V_R to V_L without passing through V_S, because an unoccupied vertex incident to a contaminated edge must receive a searcher if the edge is to be cleared. Since the strategy is progressive, no vertex will ever pass back from V_L to V_R because this would imply recontamination.

We now show that there exist optimal strategies in which no vertex ever passes back from V_L to V_S, i.e. no searcher is ever placed on an unoccupied node incident to no contaminated edges, and no vertex passes back from V_S to V_R, i.e. no searcher is removed from a vertex, leaving it unoccupied, unless all incident edges are clear. We call a strategy which has these properties *irredundant*. Optimal, irredundant strategies exist because we can remove redundant moves from an optimal progressive strategy without destroying its effectiveness, as we now show.

Firstly consider the history of a searcher who at some point arrives at a vacant vertex, none of whose incident edges are contaminated, causing the vertex to move from V_L to V_S. We can remove from the sequence of moves taken by this searcher any subsequence involving movement along a clear edge or placement on the empty vertex without affecting the remainder of the sequence. Of course removal and placement moves may have to be added at the beginning and end of the excised subsequence.

Secondly consider the case in which vertex x is vacated by a searcher and consequently moves from V_S to V_R. There are two possibilities, either x was incident to a clear edge before the searcher was moved or not. If x was adjacent to a clear edge, this edge would become contaminated, so the strategy was not progressive, as assumed. If x was adjacent only to contaminated edges, we can remove from the strategy the move which placed a searcher on x, without affecting the effectiveness of the strategy.

We define a layout based on any progressive, irredundant strategy as follows. For each vertex we consider the first step at which the strategy adds a searcher to that vertex. For all vertices x and y, if x and y are first occupied at steps i and j of S, then the constructed layout is such that $L(x) < L(y)$ iff $i < j$, i.e. the order of the vertices defined by L is exactly the order in which vertices enter V_S.

Let L_i be a partial layout with respect to L. L^{-1}(i) is the i^{th} vertex to enter V_S. Let V_L^i be the uncontaminated set and V_S^i the occupied set at the end of this move. It then follows from the observations above that domain(L_i) = $V_L^i \cup V_S^i$. Finally we note that no vertex in V_L is active because there are no edges connecting vertices in V_L to a vertex in V_R. Hence, at all steps in the strategy, $A_{L_i} \subseteq V_S^i$, and so the vertex separation of the constructed layout is no greater than the number of searchers used by the strategy. □

Lemma 2.2 $s(G) \le vs(G) + 2$.

Proof We show how a search strategy can be derived from a layout so that no more than two searchers over and above the vertex separation of the layout are used. The search strategy is the procedure defined below.

```
procedure search1(G, L);
for i := 1 to |V| do
begin
    x := L^{-1}(i);
    place a searcher on x;
    for each left neighbor y of x and each edge {x,y} do
    begin
        add a searcher to y;
        move the searcher from y to x;
        remove a searcher from x
    end;
    for each loop {x,x} do
    begin
        add a searcher to x;
        move a searcher around the loop;
        remove a searcher from x
    end;
    Remove searchers on vertices that are not active in L_i
end;
```

It can be shown, by induction on i, that at entry to the i^{th} iteration of the outer for loop, the following two conditions are satisfied:

(1) all edges connecting vertices in the domain of the partial layout L_{i-1} have been cleared, and

(2) there is exactly one searcher on each active vertex of the partial layout L_{i-1} and no searcher on any other vertex.

From this, it follows that the procedure clears all of the edges of G and that when the outer loop is entered the number of searchers on the graph is no more than $vs(G)$. Finally, note that no more than two extra searchers are added to the graph during the execution of the outer **for** loop. □

The bound in Theorem 2.1 is the best possible. The bipartite graph $K_{3,3}$ shown in Figure 1.1 has vertex separation three and search number five. To demonstrate that the search number is five we use the fact that search number is identical to cutwidth for graphs with maximum degree three [Makedon 1983], since it is easily seen that $K_{3,3}$ has cutwidth five. In Section 3.3 we give an example of a tree, Figure 3.6, which also shows a difference of two between vertex separation and search number.

2.2. A Simple Transformation

Let the 2-expansion of a graph G be the graph formed by replacing each edge $\{x,y\}$ of G by two new vertices, say a and b, and edges $\{x,a\}$, $\{a,b\}$ and $\{b,y\}$. We note that the 2-expansion of a graph contains no loops or multiple edges.

Theorem 2.2 For any graph G, s(G) is identical to the vertex separation of the 2-expansion of G.

Proof Let G' be the 2-expansion of G. By Theorem 2.1, vs(G') $\le s(G')$. Clearly, subdividing edges does not change the search number, so s(G) = s(G'). Hence $vs(G') \le s(G)$.

To show that $s(G) \le vs(G')$ we show how to construct a search strategy from a layout of a 2-expanded graph, such that the number of searchers used is no greater than the vertex separation of the layout. We distinguish the vertices in G' which are also in G from the vertices that have been added to create the 2-expansion. We call the former *original* vertices and the latter *added* vertices.

Let x and y be any pair of original vertices that were adjacent in G. Without loss of generality, suppose $L(x) \le L(y)$ in some layout L for G'. Let the added vertices for the edge $\{x, y\}$ be a and b, where a is adjacent to x and b to y. If the original edge was a loop, then x and y are the same vertex. We will call L a *standard* layout if, for all edges $\{x, y\}$, and added vertices a and b, $L(a) = L(b) - 1$ and, if x and y are distinct, $L(a) > L(x)$. Lemma 2.3 below shows that there exist standard layouts with optimal vertex separation. We construct a searching algorithm based on a standard layout L, see the procedure search2 below.

It can be shown by induction on i, that at entry to the i^{th} iteration of the **for** loop the following conditions are satisfied:

(1) all edges connecting vertices in the domain of the partial layout L_{i-1} are clear, and

(2) there is exactly one searcher on each active vertex of the partial layout L_{i-1} and no searcher on any other vertex.

The argument must show that, during the i^{th} iteration, all edges connecting vertices in domain(L_{i-1}) to L^{-1}(i) are cleared without recontamination occurring. We first note that all possible cases are covered. The case in which x is an added vertex and its two neighbors both lie to the right of x, case 3, implies we have a loop, else this arrangement would not be standard.

It is easy to see in each case that all new edges are cleared but the prevention of recontamination needs justification. The movement of searchers in line (1) does not allow recontamination, because neighbors of an original node are added nodes of degree 2. Since the layout is standard, these added nodes are adjacent to vertices which must be to the left of x. By the induction hypothesis, the edges connecting these added nodes to nodes to the left of x have already been cleared.

In line (3), moving a searcher from an added vertex to the left of x to x does not allow recontamination, since the added vertex is connected to another vertex to the left of x, because the layout is standard. By the induction hypothesis, all edges connecting nodes to the left of x have been cleared.

In line (4), since the neighbor to the left of x is not connected by an edge to any vertices to the right of x, all edges incident to this neighbor, except the one connecting it to x, have been cleared. So the searcher can be moved from this left neighbor to x without allowing recontamination.

The movement in line (5) does not allow recontamination, since a new searcher is added to the left neighbor before the move.

```
    procedure search2 (G, L)
    for i := 1 to |V| do
    begin
        x = L^-1(i);
        if x is an original vertex
        then
            if x has a neighbor placed to its left
(1)         then move a searcher from each of x's left neighbors to x;
(2)         else place a searcher on x
        else {x is an added vertex, with two neighbors. One is an
            added vertex, say y. The other is an original vertex, say z}
            begin
            Case 1: {both y and z are to the left of x}
(3)             move a searcher from y to x and then from x to z;
            Case 2: {exactly one of y or z is to the left of x}
                if there is no edge connecting this left
                  neighbor of x to a node to the right of x
(4)             then move the searcher on this node to x
(5)             else add a new searcher to the left neighbor and move it to x;
            Case 3: {both y and z are to the right of x}
(6)             place a searcher on x
            end;
        Remove searchers on vertices that are not active in L_i and
        remove duplicate searchers on vertices that are active in L_i
    end;
```

By the induction hypothesis, at entry to the i^{th} iteration of the loop, there is exactly one searcher on all and only the active vertices of L_{i-1}. Consequently, there are never more than $vs(G)$ searchers on G at entry to the loop. Since the movements described in lines (1), (3), and (4) do not introduce new searchers, it is clear that there are at most $vs(G)$ searchers on G during these steps. Only in lines (2), (5) and (6) is a new searcher added. Let line (2), (5) or (6) be executed in the i^{th} iteration of the loop. In all cases, the vertex x is an active vertex in the partial layout L_i, since at least one neighbor lies to its right. In addition, all vertices that were active in L_{i-1} are still active in L_i, because in line (2) we have that x has no left neighbors, in line (5) that the left neighbor of x is connected to a vertex to the right of x and in line (6) that x is an added node with both neighbors on its right. So the number of active vertices in L_i is one more than in L_{i-1}. Consequently, the number of searchers used in all steps of the algorithm is not larger than the number of active vertices in any partial layout, i.e. not larger than $vs(G)$. □

Lemma 2.3 Let G' be obtained from a graph by 2-expansion. If there is a layout for G' with vertex separation k then there is a standard layout for G' with vertex separation $\leq k$.

Proof Figure 2.1 (a) shows all possible positions of a and b with respect to distinct vertices x and y in which b precedes a. Figure 2.1 (b) shows all possible positions of a with respect to distinct vertices x and y in which a precedes b. We assume that other vertices could be positioned anywhere. Of the ten possible arrangements, #8 and #10 can be made standard by moving a right until it meets b, without increasing vertex separation. It is easy to see that the other arrangements can be transformed into either #8 or #10 by repositioning a or b, without increasing vertex separation:
#1 can be transformed to #7 by moving b to a position between x and y,
#2 can be transformed to #8 by moving b to a position between a and y,
#3 can be transformed to #8 by moving b to a position between a and y,
#4 can be transformed to #3 by moving a to a position between b and y,
#5 can be transformed to #10 by moving b to a position immediately following a,
#6 can be transformed to #7 by moving b to a position between x and y,
#7 can be transformed to #8 by moving a to a position between x and b,
#9 can be transformed to #8 by moving b to a position between a and y. □

We note that Theorem 2.2 together with Lengauer's transformation from vertex separation to progressive black/white pebble demand gives an explicit transformation from search number to progressive black/white pebble demand.

****

Insert Figures 2.1 (a) and (b) here

****

3. The Vertex Separation of Trees

Properties of trees can often be computed recursively and in polynomial time by computing the property for subtrees and combining the results. Meggido et al. [Megiddo 1988] give such an algorithm for computing the search number of a tree and Chung et al. [Chung 1982] give such an algorithm for computing the cutwidth of trees of fixed vertex degree, d, in time $O(n \log^d n)$. Yannakakis [Yannakakis 1985] gives an $O(n \log n)$ cutwidth algorithm for arbitrary trees that can be extended to compute the black/white pebble demand of trees. Transformations from the vertex separation problem to the search number problem, or to the pebble or cutwidth problems, that preserve treeness are not known, so a polynomial time algorithm for computing the vertex separation of a tree does not follow from the algorithms of Megiddo et al. and Yannakakis.

We present a linear time algorithm for computing the vertex separation of arbitrary trees. It depends on a recursive characterization of the vertex separation of a tree in terms of the vertex separation of the subtrees induced by the root. We also give an algorithm that constructs a layout with optimal vertex separation. The characterization allows us to describe the form and size of the smallest trees with a given vertex separation. We note also, in Section 3.3, that there exists a tree T for which $s(T) = vs(T) + 2$, so the difference can be as large as for graphs in general.

Because the search number of a graph is equal to the vertex separation of its 2-expansion, one can use the vertex separation algorithm to compute the search number of a tree. The number of edges in a tree is $O(n)$, where n is the number of vertices, so the transformation takes linear time and the entire process is still linear. Megiddo et al. [Megiddo 1988] have already given a linear time algorithm for tree search number.

3.1. A Recursive Characterization of Trees with Vertex Separation k

We present a recursive characterization of trees with vertex separation k which is analogous to the characterizations of search number and cutwidth of trees found respectively in [Parsons 1976] and [Chung 1982]. These latter characterizations underlie the tree algorithms in [Megiddo 1988] and [Chung 1982]. Let the subtrees *induced* by a vertex x be those subtrees in the forest resulting from the deletion of x from the tree. Figure 3.1 shows a tree, a vertex x and the subtrees induced by the vertex x.

****

Insert Figure 3.1 Here

****

We note that the only tree with vertex separation 0 is the tree with one vertex. In Section 3.3, Theorem 3.2, we show that a tree has vertex separation 1 if and only if it contains at least one edge and does not contain the subtree with vertex separation 2, shown in Figure 3.4. The following theorem is analogous to a theorem in [Parsons 1976] and to Theorem 2.1 in [Chung 1982].

Theorem 3.1 Let T be a tree and let $k \geq 1$. $vs(T) \leq k$ if and only if for all vertices x in T at most two of the subtrees induced by x have vertex separation k and all other subtrees have vertex separation $\leq k-1$.

Proof For any integer $k \geq 1$, let $P(k)$ denote the following property of T:

For all vertices x in T there are at most two subtrees induced by x such that the vertex separation of these subtrees is k and the vertex separation of all remaining subtrees is $\leq k-1$.

We first show that if T satisfies $P(k)$ then there is a layout L of T such that $vs_L(T) \leq k$. Let V_k be the set of vertices which induce two subtrees each with vertex separation k. Any vertex on a path connecting two members of V_k must also be a member of V_k. Further, there must exist a single path containing all members of V_k, because, if not, there exists a member x of V_k that is a part of two such paths. But then x induces three subtrees each with vertex separation k, which contradicts $P(k)$.

We show in the following paragraphs that there always exists a path, call it S, containing all the members of V_k, such that for each member x of S, the subtrees induced by x and not containing members of S all have vertex separation $\leq k-1$. Given the existence of S, there is a layout, L, with vertex separation k. It is a layout which assigns integers to vertices in a manner consistent with the following rules:

(1) If x precedes y in S then $L(x) < L(y)$.

(2) If x is not a member of S, then let T' be the induced subtree of which x is a member, let u be the member of S that induces T' and let v be next vertex after u in the sequence S. Assign an integer to all y in T' consistent with a layout of vertex separation $\leq k-1$, such that $L(u) < L(y) < L(v)$ and such that the layout of T' does not overlap the layout of any other subtree induced by u.

Since the vertex separation of all subtrees in this layout is $\leq k-1$ and no more than one vertex from S lies to the left of any induced subtree and is connected to it or to a vertex to the right of it, the vertex separation of the whole layout is $\leq k$. Let S_i be the i^{th} vertex in the sequence S. The arrangement is illustrated in Figure 3.2, in which the straight lines represent subtrees S_i laid out with minimum vertex separation.

****

Insert Figure 3.2 Here

****

To show that S exists we examine first the case for which V_k is not empty. Let x_1 and x_p be members of V_k having at most one neighbor in V_k, i.e. the ends of the path formed by the members of V_k. There is only one vertex if $|V_k| = 1$. Let x_0 and x_{p+1} be the neighbors of x_1 and x_p respectively that are not members of V_k but are part of a subtree of vertex separation k induced by x_1 or x_p respectively. Since x_0 and x_{p+1} are not in V_k they induce no more than one subtree with vertex separation k. Also, the subtrees with vertex separation k and induced by x_0 and x_{p+1} must include x_1 and x_p respectively. Let S be the sequence x_0 followed by the path formed by the vertices in V_k followed by x_{p+1}. It is evident that every subtree induced by a vertex in S and not containing a vertex in S has vertex separation $\leq k-1$. So we have exhibited the sequence as claimed above.

Now suppose that V_k is empty, i.e. there are no vertices inducing two subtrees with vertex separation k. We can form a sequence S with the desired property as follows. Take any vertex x and let it be the initial element of S. Repeatedly add a vertex to S as long as it is a neighbor of the last added vertex and induces a subtree, not containing a member of S, with vertex separation k. It is evident that this process terminates and yields an S with the desired properties.

We now show that if there is a layout L such that $vs_L(T)$ is at most k, then $P(k)$ must be true. Let vertices a and b be the first and last vertices in a layout with vertex separation $\leq k$. Let x be any vertex in T. There must be paths from both a and b to x. If a and b are members of distinct subtrees induced by x, then these paths are disjoint, else not. Now remove from the layout the vertex x, all edges incident to x, and the one or two subtrees containing the vertices a and b. What remains are all the subtrees induced by x, except those that contained a and b. Note that, because of the removal of the paths from a

and b to x, for every remaining vertex y, some vertex that was to the left of y and connected to a vertex to the right of y has been removed. Thus for all remaining subtrees T' induced by x, $vs(T') \leq k-1$. No more than two subtrees were removed and these had vertex separation $\leq k$. Note that the same argument applies even if the vertex x is a or b. The argument is illustrated by Figure 3.3.

****

Insert Figure 3.3 Here

****

Corollary 3.1 $vs(T) > k$ iff there exists a vertex which induces ≥ 3 subtrees T' such that $vs(T') \geq k$.

For example, the subtree indicated in Figure 3.4 has vertex separation 2 because its degree 3 vertex induces 3 subtrees with vertex separation 1. The trees shown in Figures 3.4 and 3.5 have vertex separation 3 because the indicated vertex x induces three subtrees, each isomorphic to the subtree with vertex separation 2.

****

Insert Figure 3.4 Here

Insert Figure 3.5 here

****

3.2. Smallest Trees with a Given Vertex Separation

Let the set of smallest trees, i.e. the trees with the least number of vertices, with vertex separation k be called $T(k)$. There is just one tree in $T(1)$, namely the tree with a single edge, and one in $T(2)$, namely the subtree in Figure 3.4. There are many in $T(3)$. Two of them are shown in Figures 3.4 and 3.5. We can deduce immediately from Theorem 3.1 that to construct a tree with vertex separation $k+1$ we can take any three members from $T(k)$ and link any one vertex from each of these to a new vertex. Furthermore, from Corollary 3.1, any tree with vertex separation $k+1$ must have a vertex which induces three subtrees with vertex separation k. So the constructed trees are among the smallest in their class.

Let $m(k)$ denote the number of vertices in a smallest tree with vertex separation k. By the rules for the construction of $T(k)$ we obtain the recurrence relation $m(k) = 3m(k-1)+1$ and $m(1) = 2$. It follows that $m(k) = \lfloor 5.3^k/6 \rfloor$ for all $k \geq 1$. Hence, for example, since $m(5) = 202$, no tree has vertex separation 5 unless it has at least 202 vertices. It also follows that for any tree T, $vs(T) = O(\log n)$, where n is the number of vertices in the tree.

3.3. The Difference between Vertex Separation and Search Number for Trees

The operation of replacing an edge $\{x,y\}$ by a new vertex z and two edges $\{x,z\}$ and $\{z,y\}$ is called edge subdivision. Two graphs are said to be homeomorphic if both can be obtained from the same graph by a finite number of edge subdivisions. Let T be a tree and let $S(T)$ denote the set of trees obtained from T by any single edge subdivision operation. If F is a family of trees, then let $S(F)$ denote the family of trees $\bigcup \{S(T) \mid T \in F\}$. For all $i \geq 1$, let $F(i)$ be the family of trees defined by:

$F(1) = T(1)$, where $T(k)$ is defined in Section 3.2,

$F(i+1)$ = the set of all trees that are formed by taking three trees in $F(i) \cup S(F(i))$
and a new vertex x and joining x by an edge to an arbitrary vertex in each tree.

The following theorem can be proved without difficulty by induction on k, by applying Theorem 3.1 and Corollary 3.1.

Theorem 3.2

For all $k \geq 1$, a tree has vertex separation $\geq k$ if and only if it contains a subtree that is a homeomorphic image of a tree in $F(k)$.

We have already seen that the vertex separation and the search number of a graph can differ by two (Figure 3.2). The tree shown in Figure 3.6 has vertex separation 3 and cutwidth 5. This tree is a smallest tree with cutwidth 5. It is constructed by uniting three trees of cutwidth 4 by sharing a vertex as shown in [Chung 1982]. That it has vertex separation 3 can be seen by applying Theorem 3.2. Let the central vertex be the root. Note that the black vertices each induce two subtrees of vertex separations 1 and 2. Consequently the vertex separation of the subtrees rooted at the black vertices is 2 and that of the entire tree is 3.

In [Makedon 1983] it is shown that for all graphs with maximum vertex degree 3, cutwidth and search number are identical. Hence the example of Figure 3.6 shows that the vertex separation and search number of trees can differ by two.

****

Insert Figure 3.6 Here

****

3.4. Computing the Vertex Separation of a Tree

We now describe a linear time algorithm for computing the vertex separation of arbitrary trees.

3.4.1. k-Criticality and Vertex Labelling

We need to view the trees as being rooted at some vertex. The parent and children of any node are then well defined, with respect to a given root. We will call the trees that are rooted at the children of a node u the subtrees *yielded* by u. In what follows, "tree" will always mean "rooted tree". The vertex separation of the rooted tree is obviously identical to the vertex separation of the underlying unrooted tree. Let $T[u]$ denote the tree, with root u, within the rooted tree T. Let $T[u, v_1, v_2, ... v_i]$ denote the tree with root u from which the subtrees with roots v_1 through v_i have been removed.

Definition 3.1
A vertex x is *k-critical* in a rooted tree T iff $vs(T[x]) = k$ and there are two children y and z of x such that $vs(T[y]) = vs(T[z]) = k$.

We observe, by Theorem 3.1, that in any tree with vertex separation k, there can be no more than one k-critical node. The following corollary also follows immediately from Theorem 3.1.

Corollary 3.2. Let $T[u]$ be a tree with root u within the rooted tree T having children $v_1, \ldots, v_d$ and let $k = \max_i \{vs(T[v_i])\}$.

(1) If more than two of the trees $T[v_i]$ have vertex separation k, then $vs(T) = k+1$.

(2) If exactly two of the trees $T[v_i]$ have vertex separation k and at least one contains a k-critical vertex, then $vs(T) = k+1$.

(3) If exactly two of the trees $T[v_i]$ have vertex separation k and neither one contains a k-critical vertex, then $vs(T) = k$.

(4) If exactly one of the trees $T[v_i]$ has vertex separation k and it contains a k-critical vertex x and $vs(T[u,x]) = k$, then $vs(T) = k+1$.

(5) If exactly one of the trees $T[v_i]$ has vertex separation k and it contains a k-critical vertex x and $vs(T[u,x]) < k$, then $vs(T) = k$.

(6) If exactly one of the trees $T[v_i]$ has vertex separation k and it does not contain a k-critical vertex, then $vs(T) = k$.

Proof Each statement can be derived immediately from Theorem 3.1. □

Our algorithm is going to compute a *label* for each vertex.

Definition 3.2

For any tree $T[u]$ define the *label* of u to be the list of integers $(a_1, \ldots, a_p)$, where $a_1 > a_2 > \cdots > a_p \geq 0$, and such that there exists a set of vertices $\{v_1, \ldots, v_p\}$ such that

(1) $vs(T[u]) = a_1$

(2) For $1 \leq i < p$, $vs(T[u, v_1, \ldots, v_i]) = a_{i+1}$.

(3) For $1 \leq i < p$, v_i is an a_i-critical vertex in $T[u, v_1, \ldots, v_{i-1}]$.

(4) v_p is u. If a_p is marked with a prime ($'$) then there is no a_p-critical vertex in $T[u, v_1, \ldots, v_{p-1}]$. If a_p is not marked with a prime then v_p is an a_p-critical vertex. In both cases $T[u, v_p] = T[u, u]$ is the empty tree.

For example, the label $(2,0')$ on vertex u means that $T[u]$ has vertex separation 2, that there is a 2-critical vertex, say v_1, in $T[u]$ and that $T[u, v_1]$ has vertex separation 0, i.e., it is a single vertex. The label (2) on a vertex u means that the vertex separation of $T[u]$ is 2 and that u is a 2-critical vertex so that $T[u, u]$ is empty. We will refer to an element of the label that is associated with a critical vertex as a *critical element*. Note that the prime marker is used, if ever, only on the last element, since all others are necessarily critical. This labelling technique is similar in style to the techniques used in [Yannakakis 1985], [Chung 1982] and [Megiddo 1988] on trees to compute search number and cutwidth.

3.4.2. The Vertex Separation Algorithm

We arbitrarily choose a vertex to be the root of T. The label of the root is computed by recursively computing the labels of its children and then combining them. The vertex separation of T is the largest element in the label of root. Figure 3.7 shows an example of the labels produced by the labelling algorithm on a particular tree and for a particular choice of root. We use a list concatenation operation, denoted by "&".

****

Insert Figure 3.7 here

****

```
function vs(T: tree) : integer;  {Computes the vertex separation of a tree T}
   Choose some vertex u in T and make u the root of T;
   vs := largest element of compute_label (T, u);

function compute_label (T : tree, u : vertex) : label;
   {Computes the label of the vertex u in the tree T[u].
   if u is the only vertex in the tree T[u]
   then compute_label := (0)
   else begin
      for all vertices v_i, the d children of u, do λ_i := compute_label (T, v_i);
      {Compute the label for u by combining the labels λ_i}
      compute_label := combine_labels(λ_1, λ_2, · · · , λ_d)
   end;


function combine_labels(λ_1, λ_2, · · · , λ_d : label) : label;
   {Computes a new label, λ, by combining a set of labels}
   if there is one or more label containing 0 then λ := (1) else λ := (0);
   {Let m be the largest element in any label}
   for k := 1 to m do
   begin
     {Let n be the number of labels containing an element k}
     Case 1: {n ≥ 3} λ := (k+1′);
     Case 2: {n = 2 and at least one element k is critical)} λ := (k+1′);
     Case 3: {n = 2 and neither element k is critical} λ := (k);
     Case 4: {n = 1 and element k is critical and k ∈ λ} λ := (k+1′);
     Case 5: {n = 1 and element k is critical and not (k ∈ λ)} λ := (k) & λ;
     Case 6: {n = 1 and element k is not critical} λ := (k′)
   end;
   combine_labels := λ;
```

3.4.3. Correctness of the Algorithm

Theorem 3.2

The function combine_labels computes the label of a vertex u in the rooted tree T whose d children have the labels λ_1 through λ_d.

Proof

We have already defined in the algorithm: $m = \max_i \{vs(T[v_i])\}$.

For each $T[v_i]$, let $T^i_m, T^i_{m-1}, \ldots, T^i_0$ be the sequence of trees defined by:

$T^i_m = T[v_i]$ and, for $0 < k \le m$,

if k is equal to some element a_j of the label of v_i

then T^i_{k-1} is obtained from T^i_k by removing $T[v_j]$, i.e. the tree rooted at the vertex associated with a_j

else $T^i_{k-1} = T^i_k$.

That is, we successively remove trees rooted at the nodes corresponding to the elements in the label of the root, v_i. We observe that, for all k less than the smallest element in the label of v_i, T^i_k is the empty tree but that, if 0 is in the label, then T^i_0 is a single vertex.

Finally we define T_k, $0 \le k \le m$, to be the tree with root u yielding subtrees T^i_k, $1 \le i \le d$. Hence $T_m = T[u]$ and, if all the T^i_0 are empty, T_0 is a single vertex, else it is a star graph.

The argument proceeds by induction on k. Let Λ denote the label of u in $T[u] = T_m$ and, for $0 \le i \le m$, let Λ_i denote the label of u in T_i. Our induction hypothesis is that, at the top of the loop, $\lambda = \Lambda_{k-1}$, where k is the current value of the variable k, and that $\lambda = \Lambda_k$ at the bottom of the loop. If so, then certainly $\lambda = \Lambda_m = \Lambda$ at exit from the loop.

If T_0 is a single vertex, then $\Lambda_0 = (0)$, else it is a star graph, in which case $\Lambda_0 = (1)$. Since the if statement checks to see if any T^i_0 is non-empty, our hypothesis is true at first entry to the loop. Assume then that the induction hypothesis holds and consider the various cases.

Case 1: More than two of the T^i_k have vertex separation k. By case 1 of Corollary 3.2, $vs(T_k) = k+1$. Clearly there can be no $(k+1)$-critical vertex. Hence $\Lambda_k = (k+1)'$.

Case 2: Exactly two of the T^i_k have vertex separation k and at least one has a k-critical vertex. By case 2 of Corollary 3.2, $vs(T_k) = k+1$. Clearly there can be no $(k+1)$-critical vertex. Hence $\Lambda_k = (k+1')$.

Case 3: Exactly two of the T^i_k have vertex separation k and neither has a k-critical vertex. By case 3 of Corollary 3.2, $vs(T_k) = k$ and, since u is a k-critical vertex, $\Lambda_k = (k)$.

Case 4: Exactly one of the T^i_k has vertex separation k and it contains a k-critical vertex and $vs(T_{k-1}) = k$. By case 4 of Corollary 3.2, $vs(T_k) = k+1$. Clearly there can be no $(k+1)$-critical vertex. Hence $\Lambda_k = (k+1)$.

Case 5: Exactly one of the T^i_k has vertex separation k and it contains a k-critical vertex and $vs(T_{k-1}) < k$. By case 5 of Corollary 3.2, $vs(T_k) = k$. By the definition of a label, $\Lambda_k = (k)$ concatenated with Λ_{k-1}.

Case 6: Exactly one of the T^i_k has vertex separation k and it does not contain a k-critical vertex. By case 6 of Corollary 3.2, $vs(T_k) = k$. If $vs(T_{k-1}) < k$, then clearly there can be no k-critical vertex in T_k. Suppose $vs(T_{k-1}) = k$. Then the current value of λ was created by case 1, 2 or 4 in the previous iteration of the loop. By observing these cases, we see that there is no k-critical vertex in T_{k-1}. So, if there is a k-critical vertex in T_k, it must be the root. By observing cases 1, 2 and 4, we see that no subtree yielded by u in T_{k-1} has vertex separation k. Hence the root can yield no more than one subtree in T_k of vertex separation k and can not be k-critical. Hence T_k has no k-critical vertex and $\Lambda_k = (k)$.

Finally we note that if there is no k element in any of the labels, the algorithm does nothing. In this case, $T_k = T_{k-1}$, so $\Lambda_k = \Lambda_{k-1}$. □

3.4.4. Time Complexity

Clearly the time complexity of the algorithm just defined will be determined by the data structure used to represent the labels and sets of labels. Suppose each label is represented by an ordered linked list of integers and that a set of labels is represented by a linked list of labels. An element in the label can easily be associated with a non-criticality indicator, equivalent to the prime in our notation. The order of the labels in the list is immaterial. This arrangement would require time $O(d \log n)$, where d is the number of children of a given vertex, to combine d labels, since the length of any label is $O(\log n)$, as shown in Section 3.2. Hence the sum of the label combination work over all vertices would be $O(n \log n)$.

To achieve linear time we refine the label representation by representing a sub-list of consecutive integers in a label by an interval denoted by its endpoints, i.e., the label consists of a linked list of pairs of integers. For example, the vertex label (8, 7, 6, 5, 3, 2′) would be represented as ((8, 5), (3, 2′)). A concatenation operation on lists, would compare the last interval on the first list and the first interval on the second list and merge these intervals if they overlap so as to maintain the proper form. For example, ((9, 6), (4, 3)) concatenated with ((2, 2), (0,0)) would evaluate to ((9, 6), (4, 2), (0,0)). Clearly concatenation can be done in constant time.

The purpose of this representation by intervals is to allow us to terminate the label combination process once all elements of the second largest label have been scanned, where largest means containing the largest element. At this point a new label λ has been constructed which represents the correct addition of all label elements except those in the remaining segment of the largest label. The final step need only combine this remaining segment with λ. This can be done in constant time, independent of the number of items remaining in the last label, because the effect of the combination can not extend beyond the smallest interval pair in the remaining segment, as we now show.

Let us assume that the algorithm proceeds by scanning the smallest elements in each label, and removes them as they become equal to the current value of k. As a label becomes exhausted it disappears from the list of labels. First we let m be, not necessarily the maximum, but the second number in an ordered list of the vertex separations of the subtrees of the root. Consequently the loop terminates as soon as no more than one label contains elements yet to be processed. The following statement then achieves the final step.

if some segment of some label remains unprocessed
then begin {Denote the remaining label segment by λ_q and let it be of the form: (... , (a, b)) }
  Case 1: {b is critical and $b \in \lambda$} Remove (a, b) from λ_q; $\lambda := \lambda_q$ & $((a+1, a+1'))$;
  Case 2: {b is critical and not $(b \in \lambda)$} $\lambda := \lambda_q$ & λ;
  Case 3: {b is not critical} $\lambda := \lambda_q$
end;

These statements can be justified by reapplying cases 3, 4 and 5 of the proof of Theorem 3.2. So long as the assignment and concatenation operations are done, not by copying labels, but by renaming existing structures, constant time suffices for the entire statement. However, we note that since we destroy labels in the set being combined, the procedure does not leave a label on each node.

We now demonstrate that this algorithm - data structure combination has linear time complexity. Let u be a vertex in a rooted tree T and let $q_{u,i}$ be the number of subtrees in $T[u]$ with vertex separation at least i and for which there exists a sibling with vertex separation at least i. Note that these subtrees are not required to be disjoint.

Lemma 3.1
The time required by the algorithm to compute a label for u in T is:

$$t(n) \le c_1 n + c_2 \sum_{i=0}^{vs(T[u])} q_{u,i}$$

for some constants c_1 and c_2 and where n is the number of vertices in $T[u]$.

Proof
The proof proceeds by induction on the height of $T[u]$. For the basis of the induction we note that, for trees of height 0, u is the only vertex in the tree and the algorithm computes its label in constant time, which is consistent with the claim.

Now consider a tree, with root u, of height $h+1$ and assume that the claim is true for all trees of height $\le h$, i.e. for all the subtrees yielded by u. The time required to compute a label for u is the sum of the times required to compute a label for each of the subtrees plus the time required to combine these labels.

Let c_1 be an upper bound on the time required to carry out the final stage of the label combination process, i.e., the combination of the remaining segment of the largest label with the new label λ under construction. Let s_i be the largest integer in the i^{th} largest label in the combination process. We have shown that the label combination process requires time $t_c \le c_1 + c_2(s_2 + s_2 + \cdots + s_d)$, where we note that s_1 does not appear, but s_2 appears twice for the reasons given. If we define r_i to be the number of subtrees yielded by u with vertex separation at least i, then this equation can be rephrased as: $t_c \le c_1 + c_2 \sum_{i=0}^{s_2} r_i$.

Let u have d children, v_1 through v_d. The sum of the times required to compute the labels of the subtrees of u is, by the inductive assumption:

$$t_s \leq (n-1)c_1 + c_2 \sum_{j=1}^{d} \sum_{i=1}^{vs(T[v_j])} q_{v_j,i}$$

The sum $t_c + t_s$ gives:

$$t(n) \leq c_1 n + c_2 \sum_{i=0}^{vs(T[u])} q_{u,i}$$

thus confirming our hypothesis. □

Lemma 3.2

If a rooted tree has p leaves and r internal vertices which have at least one sibling, then $r \leq p$.

Proof

We prove that either $r = 0$ or $r \leq p-2$. The proof is by induction on the height of the tree. The lemma is vacuously true for a single vertex. As a basis, consider trees of height 1, for which the statement is true, since $r = 0$. Now suppose the hypothesis is true for all trees of height $\leq h$, and consider the root of a tree of $h+1$. This root yields subtrees for which we have assumed the hypothesis is true. Let r_i and p_i be the values of interest for the i^{th} subtree.

There are two cases. If there is just one subtree, then the numbers r_1 and p_1 remain unchanged. If there are two or more subtrees we have:

$$r \ \leq \ \sum r_i + d \ \leq \ \sum p_i - 2d + d \ \leq \ p - d \ \leq \ p - 2$$

which confirms the hypothesis. □

Theorem 3.2

The time complexity of the algorithm is $O(n)$, if the data structure just described is used.

Proof

For a given i, $0 \leq i \leq vs(T)$, consider those vertices that are the roots of subtrees of vertex separation $\geq i$, but yield no subtrees of vertex separation $\geq i$. Call these distinguished vertices. If there are p such vertices, Lemma 3.2 tells us that there are no more than another p vertices that are both ancestors of a distinguished vertex and have a sibling which is also the ancestor of a distinguished vertex. Hence, there are at least $q_i/2$ disjoint subtrees with vertex separation at least i, namely, those rooted at the distinguished vertices.

From Section 3.2, we know that a subtree with vertex separation at least i, must have at least 2^i vertices. Since there are at least $q_i/2$ such subtrees, $q_i \leq n/2^{i-1}$. Hence:

$$\sum_{i=0}^{\log_2 n} q_{u,i} \leq 2(n + n/2 + \cdots) \leq 4n$$

Consequently, from Lemma 3.1, the time complexity is linear. □

3.5. Computing an Optimal Layout

Once labels have been computed for all vertices, an optimal layout can be computed. The layout procedure assumes, for the sake of convenience, that the tree is rooted at r. We use the same root as used in the labelling procedure, so the vertex separation of the tree is the largest number in the label of this root. The procedure is invoked initially on the root. It will assign a unique integer, pos, $1 \leq pos \leq |V|$, to each vertex. This number is the position of the vertex in an optimal layout.

The efficacy of the procedure follows immediately from the proof of Theorem 3.1. The procedure finds the sequence S described in this proof, recursively computes the layout of each subtree induced by members of S and places them to the right of this member of S. As in the proof, there are two possibilities, either there are critical vertices or not, and this is indicated by the label on the root. If there are critical vertices and r is not critical, a path is found from r to the nearest critical vertex. *Pos* must be given the initial value 1. $Label(x)$ means the label of x and L is the layout function.

procedure layout (x); $\{x$ is the root of a tree$\}$
  $k := max(label(x))$; $c := x$;
  if $T[c]$ has a critical vertex
  then while c is not a k-critical vertex **do**
    begin delete k from $label(c)$; c := the child of c with k in its label **end**;
  {Let $(v_1, v_2, \cdots v_s)$ be the sequence S containing all vertices x in $T[c]$ such that $label(x)$ contains k}
  for i := 1 **to** s **do**
  begin $L(V_i) := pos$; $pos := pos + 1$; delete v_i from T;
    for all children y of v_i **do** layout(y);
    if $(v_i = c$ **and** $x \neq c)$ **then** layout(x)
  end;

The last **if** statement lays out the subtree of $T[x,c]$, since vertex c has been deleted. We observe that $T[x,c]$ is a subtree induced by vertex c with vertex separation less than k. Hence, by Theorem 3.1, it should be laid out after c but before the next vertex in S. The time complexity of the procedure is $O(n \log n)$ since no vertex is visited more than k times and k is $O(\log n)$.

Also, notice that we must use a straightforward version of the vertex separation algorithm to compute a label for each vertex. The refined, linear time version, did not do this since it computed new labels by destroying old ones. It is an open question, whether or not there exists a linear time layout algorithm. Note that such an algorithm could not represent the labels explicitly for each vertex since this requires $\Omega(n \log n)$ space in the worst-case. It seems likely however that a variation on our scheme using an implicit representation for the labels can be developed, leading to a linear time layout algorithm.

References

Arnborg, S., Corneil, D.G. and Proskurowski, A. (1987) ‘‘Complexity of finding embeddings in a k-tree’’ SIAM J. Alg. Disc. Meth., 8, pp. 277 - 284.

Bodlaender, H. L. and Kloks, T. (1991), ‘‘Better Algorithms for the Pathwidth and Treewidth of Graphs’’ Proceedings 18^{th} International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 510, Springer-Verlag, pp. 544 - 555.

Chung, M. J., Makedon, F., Sudborough, I. H. and Turner, J. (1985), ‘‘Polynomial Algorithms for the Min-Cut Linear Arrangement Problem on Degree Restricted Trees’’, SIAM J. Computing, 14, 1, pp. 158-177.

Ellis, J., Sudborough, I. H. and Turner, J. (1983), ‘‘Graph Separation and Search Number’’ Proceedings 21st Allerton Conference on Communication Control and Computing.

Ellis, J., Sudborough, I. H. and Turner, J. (1987), ‘‘Graph Separation and Search Number’’ Technical Report DCS-66-IR, Department of Computer Science, University of Victoria, British Columbia, Canada.

Fellows, M. R. and Langston, M. A. (1987), ‘‘Nonconstructive Advances in Polynomial-Time Complexity’’, Info. Proc. Letters, 26, pp. 157-162.

Fellows, M. R. and Langston, M. A. (1988), ‘‘Nonconstructive Tools for Proving Polynomial-Time Decidability’’, JACM, 35, pp. 727-739.

Fellows, M. R. and Langston, M. A. (1989), ‘‘On Search, Decision and the Efficiency of Polynomial Time Algorithms’’, Proc. 21st ACM Symposium on the Theory of Computing, pp. 501-512, 1989.

Kinnersley, N. G. (1990), ‘‘The Vertex Separation Number Equals its Path-width’’, manuscript, Department of Computer Science, University of Kansas, Lawrence, KS, USA.

Kirousis, L. M. and Papadimitriou, C. H. (1986), ‘‘Searching and Pebbling’’, Theoretical Computer Science, 47, pp. 205-216.

LaPaugh, A. S. (1983), ‘‘Recontamination does not Help to Search a Graph’’, Technical Report, Electrical Engineering and Computer Science Department, Princeton University.

Leiserson, C. E. (1980), ‘‘Area-Efficient Graph Layouts, for VLSI’’, Proc. 21st Annual Symp. Foundations of Computer Science, pp. 270-281.

Lengauer, T. (1981), ‘‘Black-White Pebbles and Graph Separation’’, Acta Informatica, 16, pp. 465-475.

Lipton, R. J. and Tarjan, R. E. (1979), ‘‘A Separator Theorem for Planar Graphs’’, SIAM J. Appl. Math. 36, 2, pp. 177-189.

Lipton, R. J. and Tarjan, R. E. (1980), ‘‘Applications of a Planar Separator Theorem’’, SIAM J. Computing, 9, 3, pp. 615-627.

Makedon, F. and Sudborough, I. H. (1983), ‘‘Minimizing Width in Linear Layouts’’, Proc. 10th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 154, pp. 478-490, Springer Verlag.

Megiddo, N., Hakimi, S. L., Garey, M. R., Johnson, D. S. and Papadimitriou, C. H. (1988), ‘‘The Complexity of Searching a Graph ’’, JACM, 35, pp. 18-44.

Miller, Z. and Sudborough, I. H. (1990), ‘‘A Polynomial Algorithm for Recognizing Bounded Cutwidth in Hypergraphs’’, to appear in Math. Systems Theory.

Parsons, T. D. (1976), ‘‘Pursuit-Evasion in a Graph’’, in Theory and Application of Graphs, Y. Alavi and D. R. Lick, Springer-Verlag, pp. 426-441.

Parsons, T. D. (1978), ‘‘The Search Number of a Connected Graph’’, Proc. 9th Southeastern Conf. on Combinatorics, Graph Theory, and Computing, Utilitas Mathematica Publishing, Winnipeg, pp. 549-554.

Philipp, R. and Prauss, E. (1980), ‘‘Uber Separatoren in planaren Graphen’’, Acta Informatica, 14, 1, pp. 87-97.

Saxe, J. B. (1980), ‘‘Dynamic-Programming Algorithms for Recognizing Small Bandwidth Graphs in Polynomial Time’’, SIAM J. Algebraic and Discrete Methods, 1, 4, pp. 363-369.

Yannakakis, M. (1985), ‘‘A Polynomial Algorithm for the Min-Cut Linear Arrangement of Trees’’, JACM, 32, 4, pp. 950-988.