Report Number: WUCS-87-11

1987-05-01

Graph Separation and Search Number

J. A. Ellis, H. Sudborough, and Jonathan S. Turner

We relate two concepts in graph theory and algorithmic complexity, namely the search number and the vertex separation of a graph. Lengauer has previously related vertex separation to progressive black/white pebble demand. Let a (G) denote the search number and vs(G) denote the vertex separation of a connected, undirected graph G. We show that vs(G) < s(G) < vs(G) +2 and we give a simple transformation from G to G^1 such that vs(G^1) = s(G). We give algorithm that, for any tree T, compute vs(T) in linear time and compute an optimal layout with respect to vertex separation in... **Read complete abstract on page 2.**

Graph Separation and Search Number

J. A. Ellis, H. Sudborough, and Jonathan S. Turner

Complete Abstract:

We relate two concepts in graph theory and algorithmic complexity, namely the search number and the vertex separation of a graph. Lengauer has previously related vertex separation to progressive black/white pebble demand. Let a (G) denote the search number and vs(G) denote the vertex separation of a connected, undirected graph G. We show that vs(G) < s(G) < vs(G) +2 and we give a simple transformation from G to G^1 such that vs(G^1) = s(G). We give algorithm 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). We characterize those trees having a given vertex separation and describe the smallest such trees. We give an algorithm which, for all fixed k>1, decides the problem: "Is vs(G)

GRAPH SEPARATION AND SEARCH NUMBER

J. A. Ellis, I. H. Sudborough and
J. S. Turner

WUCS-87-11

May 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

GRAPH SEPARATION AND SEARCH NUMBER

J. A. Ellis(1), I. H. Sudborough(2) and J. S. Turner(3)

(1) Department of Computer Science

University of Victoria

Victoria, British Columbia V8W 2Y2, Canada

(2) Computer Science Department

University of Texas at Dallas

Richardson, Texas 75083, U S A

(3) Computer Science Department

Washington University

St. Louis, Missouri 63130, U S A

Abstract

We relate two concepts in graph theory and algorithmic complexity, namely the ***search number*** and the ***vertex separation*** of a graph. Lengauer has previously related vertex separation to progressive black/white pebble demand. 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 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 $O(n \log n)$. We characterize those trees having a given vertex separation and describe the smallest such trees.

We give an algorithm which, for all fixed $k \geq 1$, decides the problem: "Is $vs(G) \leq k$?" in polynomial time. Because of the relationships mentioned, this algorithm implies that the search number and progressive black/white pebble demand problems are also decidable in polynomial time, for fixed k. All three problems are known to be NP-complete in general.

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 or pebble demand.

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 two disconnected sets of vertices. Small separators that divide the graph into roughly equal components were used to describe good VLSI layouts in [LEIS80] and to describe good divide and conquer algorithms in [LIPT80]. Theorems which guarantee the existence of small separators for planar graphs and graphs of fixed genus have been described by Lipton and Tarjan [LIPT79] and by Philipp and Prauss [PHIL80].

Lengauer [LENG81] 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 $L : V \rightarrow \{1, 2, \ldots, |V|\}$. A ***partial layout*** of G is a one to one mapping L from a subset V' of V to the set of integers $\{1, 2, \ldots, |V'|\}$. 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 $L(x) \leq i$ 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 $vs_L(G) = \max\{ |V_L(i)| \mid 1 \leq i \leq |\text{domain}(L)|\}$ 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

Given a partial layout L, whose domain is some subset V' of all the vertices, and a positive integer $i \leq |V'|$, the partial layout L_i is the mapping that agrees with L on the vertices $\{L^{-1}(1), L^{-1}(2), \ldots, L^{-1}(i)\}$ 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 [PARS76] [PARS78]. 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 [LAPA83] 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. [MEGI81] show that determining the search number of a graph is NP-hard, which implies it is **NP**-complete because of LaPaugh's result. They also show that the search number of a tree can be determined in $O(n)$ steps and that determining if a graph has search number 2 or search number 3 can be done in $O(n)$ steps. It is known that, for any graph G with maximum vertex degree 3, $s(G)$ is identical to the cutwidth of G [MAKE83]. 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 [LEIS80].

The progressive black/white pebble game, which does not allow repebbling, is played on directed graphs according to the following rules:

(1) All vertices start free of pebbles.

(2) All vertices end free of pebbles.

(3) Each vertex receives and loses a pebble exactly once.

(4) A white pebble can be placed on a vertex at any time.

(5) A black pebble can be placed on a vertex only if all the predecessors of the vertex are currently pebbled.

(6) A white pebble can be removed from a vertex only if all the predecessors of the vertex are currently pebbled.

(7) A black pebble can be removed from a vertex at any time.

A *pebbling strategy* is a legitimate sequence of moves that starts from and finishes with the empty graph and pebbles each node. The *progressive black/white pebble demand* of the graph is the minimum over all strategies of the maximum number of pebbles on the graph during the game. The black/white pebble game models the space requirements of non-deterministic, straight line programs. Lengauer [LENG81] showed that the vertex separation and progressive black/white pebble problems are polynomially reducible one to the other.

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 and describe the smallest such trees. In the last section we describe an algorithm which, for fixed k, decides the vertex separation problem in polynomial time. Because of the relations between vertex separation and progressive black/white pebble demand and between vertex separation and search number, it follows that these problems are also decidable in polynomial time, for fixed k.

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 $vs(G) \leq s(G) \leq vs(G) + 2$.

Proof From Lemmas 2.1 and 2.3 below. ■

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

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 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 V_L from 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$(L_i) = V_L^i \bigcup V_S^i$. 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 $s(G) \leq vs(G) + 2$.

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 $x$ do
    begin
        add a searcher to $y$;
        move a searcher from $y$ to $x$;
        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. The easiest way to demonstrate that the search number is five uses the fact that search number is identical to cutwidth for graphs with maximum vertex degree three [MAKE83], 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.

Kirousis and Papadimitriou [KIRO83] have recently extended Theorem 2.1 by comparing vertex separation with the number of searchers needed in a variation of the searching game, which they call "node searching". In the node searching version, an edge $\{x,y\}$ is considered to be become clear if x and y are simultaneously occupied by searchers. They show that, for all graphs G, the node search number of G is identical to the vertex separation of G plus 1.

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 $\{b,y\}$.

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') $\leq s(G')$. Clearly, subdividing edges does not change the search number, so s(G) = s(G'). Hence $vs(G') \leq s(G)$.

To show that $s(G) \leq vs(G')$ 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 $L(x) \leq L(y)$ 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 edge is 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.

```
      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;
```

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.

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 $vs(G)$ 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.

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. [MEGI81] give such an algorithm for computing the search number of a tree and Chung et al. [CHUN82] give such an algorithm for computing the cutwidth of trees of fixed vertex degree, d, in time $O(n \log^d n)$. Yannakakis [YANN85] 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 the graph seen in the previous section.

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. [MEGI81] give an $O(n \log n)$ algorithm for tree search number and indicate that it can be improved to linear time.

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 [PARS76] and [CHUN82]. These latter characterizations underlie the tree algorithms in [MEGI81] and [CHUN82]. 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. For convenience we adopt the convention that the empty tree has vertex separation -1. The following theorem is analogous to a theorem in [PARS76] and to Theorem 2.1 in [CHUN82].

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 vertex, 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. A graph G' is a homeomorphic image of a graph G if G' can be obtained from G 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 may differ by two. 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 [CHUN82]. 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 [MAKE83] 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. We first define some concepts, then we describe the algorithm and justify both its correctness and the claim of linear time complexity.

3.4.1. k-Criticality and Vertex Labelling

In the following we shall consider directed trees for convenience. The vertex separation of a directed tree is identical to the vertex separation of the underlying undirected tree. Let $T[x]$ denote the subtree of the directed tree T with the root vertex x.

Definition 3.1

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

Let $T[x, v_1, v_2, \ldots\ldots v_i]$ be the tree with root x from which the subtrees with roots v_1 through v_i have been removed. Let T be a directed tree with root u. It follows directly from Theorem 3.1 that if $vs(T) = k$ and there is a k-critical vertex x in T then $vs(T[u, x]) \leq k-1$.

Definition 3.2

The assignment to a vertex x in a directed tree T, of a *label* consisting of a vector of integers $(a_1, a_2, \ldots, a_d)$ from {-1, 0, 1, 2, ...} means that:

(1) $vs(T[x]) = a_1$ and

(2) for all i, $1 \leq i < d$, there exists an a_i-critical vertex v_i such that $v_i \in T[x, v_1, v_2, ..., v_{i-1}]$ and $vs(T[x, v_1, v_2, \ldots, v_i]) = a_{i+1}$ and

(3) there is no a_d-critical vertex in $T[x, v_1, v_2, ..., v_{d-1}]$.

For example, the label (2,0) on vertex x means that T[x] has vertex separation 2, that there is a 2-critical vertex, say v_1, in $T[x]$ and that T[x,v_1] has vertex separation 0, i.e. it is a single vertex. The label (2,-1) on a vertex x means that the vertex separation of $T[x]$ is 2 and that x is a 2-critical vertex so that T[x,x] is empty. This labelling technique is analogous to the techniques used in [YANN85], [CHUN82] and [MEGI81] on trees to compute search number and cutwidth.

3.4.2. The Vertex Separation Algorithm

```
procedure compute_vs($T$); {Computes the vertex separation of a tree T}
    Choose some vertex $u$ in $T$ and make $u$ the root of $T$;
    Invoke compute_label ($u$);
    $vs(T)$ := max(label of $u$);


procedure compute_label ($u$); {Computes the label of the vertex $u$ in the tree $T[u]$.
    if $u$ is the only vertex in the tree $T[u]$
    then give $u$ the label "(0)"
    else begin
        Compute the labels of all vertices $v_i$, children of $u$, by invoking compute_label ($v_i$).
        Invoke add_labels, which computes the label for $u$ by combining the labels of all $v_i$.
    end;
```

We are arbitrarily choosing some vertex to be the root of T. The label of the root is computed by recursively computing the labels of its children and combining them. The vertex separation of T is the largest number in the label of root. In order to combine labels we represent each label as a binary string. Each bit shows the presence or absence of an integer in the label set. For example, the label (4, 3, 1) is represented by the string 110100. Most significant bits are on the left. Note that zero and -1 are possible members of the label, so the rightmost two bits indicate their presence or absence. We combine these binary strings by working from right to left and applying the rules defined in Table 3.1 together with the actions defined in Table 3.2.

The process uses a carry bit and a string of sum bits, sum_d through sum_{-1}, where d is the largest number in the set of labels being combined. The result, i.e. the label of the root, is given by concatenating the carry bit with the string of sum bits. An example of the computation of the label of the root of a tree is given in Figure 3.7.

```
procedure add_labels;
    {Computes a new label for the vertex $u$ by combining the labels of its children}
    $carry$ := 0; $sum_d$ through $sum_{-1}$ := 0;
    $j$ := 0; {$j$ is used to avoid repeatedly setting the same bits to 0}
    {Let "column $i$" be the vector of bits $i$, one from each label}
    if there is one or more 1 in column 0 then $carry$ := 1 else $sum_0$ := 1;
    for $i$ := 1 to $d$ do Combine column $i$ according to the rules defined in Tables 3.1 and 3.2 below;
    label of $u$ := carry bit concatenated with the string $sum_d$ through $sum_{-1}$;
```

# of ones in column i	carry = 0	carry = 1
3 of any kind 2, both critical 1 critical, 1 not critical	Action 1	Action 1
2, both not critical	Action 2	Action 2
1, critical	Action 3	Action 1
1, not critical	Action 4	Action 4
0	nothing	Action 4

Table 3.1. Label Union Rules

Action 1	$carry$:= 1 sum_i thru sum_j := 0 j := j+1
Action 2	$carry$:= 0 sum_{i-1} thru sum_j := 0 sum_i := 1 sum_{-1} := 1 j := i
Action 3	sum_i := 1
Action 4	sum_{i-1} thru sum_0 := 0 $carry$:= 0 sum_i := 1 j := i

Table 3.2. The Label Union Actions

3.4.3. Correctness of the Algorithm

Suppose there are m labels with maximum value d, so that we can view the set of labels as rows in an m by $d+2$ matrix of bits. Each 1 in column i of the matrix, except for column -1, indicates the existence of a distinct subtree with vertex separation i. The bit in column -1 is used as a criticality indicator when the root vertex is critical. Let T be the tree formed by combining m subtrees with a new root, r. Let T_i denote the subtree of T from which have been removed all those subtrees indicated by all the ones in columns i through d. The combination process can then be seen as adding in subtrees with progressively larger vertex separations.

We can proceed by induction on i. We assert that for $1 \leq i$, after the addition of column $i - 1$ has been completed, the label formed by concatenating the carry bit with sum_{i-1} through sum_{-1} is the label for the root of T_i.

The procedure starts at column 0. A label including a zero value indicates that, after all subtrees rooted at critical vertices have been removed, a single vertex remains. Consequently, if any subtree has a zero in its label, T_1 has vertex separation 1, and there is no 1-critical vertex. If no subtree has a zero in its label then T_1 is a single vertex, with vertex separation 0. In both cases the initial step of the algorithm is correct.

We show that the assertion remains true as we progress from column $i - 1$ to i by examining each of the proposed rules and actions.

Action 1: The action is appropriate since, in all cases invoking this action, there exists a vertex, call it the significant vertex, with at least three subtrees T', such that $vs(T') = i$. Thus $vs(T_i) = i + 1$. Note also that there is no $(i + 1)$ critical vertex. Let the significant vertex be s. In each of the four occasions which stimulate Action 1, we justify the assertion:

If there are ≥ 3 subtrees T' with $vs(T') = i$, then s is the root of T_i.

If there are two subtrees T' with $vs(T') = i$, and each T' has a critical vertex, then s could be either of the critical vertices.

If one of the two subtrees has a critical vertex and the other not, then s is the critical vertex.

If there is one subtree with a critical vertex, and $vs(T_{i-1}) = i$ then the carry bit is one and again s is the critical vertex.

Action 2: The root r induces two subtrees T' such that $vs(T')=i$, thus $vs(T_i)=i$ but r is at this point an i-critical vertex. When $T(r)$ is removed, the graph is empty, so the label of T_i is $(i, -1)$. This argument holds whether or not $vs(T_{i-1}) = 1$, i.e. whether or not the carry bit is one.

Action 3: $vs(T_i) = i$ and there is an i-critical vertex, say x. If $T[x]$ is removed from T_i, the resulting tree is described by sum_{i-1} thru sum_{-1} previously computed.

Action 4: $vs(T_i) = i$ and there is no i-critical vertex in T_i, whether or not $vs(T_{i-1}) = 1$.

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.

Insert Figure 3.7 Here

3.4.4. Time Complexity of the Algorithm

We can refine the algorithm just given by representing the labels succinctly, and show that the computation can be done in linear time. We will represent a label by a bit string, as previously described, and represent the bit string more compactly so that each maximal homogeneous substring is represented by an integer, defining the length of the substring, e.g. the string "11100100011" would be represented by "3 2 1 3 2". Note that the first number must refer to a sequence of ones, so there is no ambiguity. Each label would be a linked list of numbers, and a set of labels would be a linked list of these. The order of the elements within a label is of course significant, but not the order within a set of labels.

Consider again the label combination process which, given a set of subtree labels, computes the label of the tree obtained by combining the subtrees. The process proceeds from least significant bit to most significant, possibly examining all bits in all labels. Notice though that once we reach a column beyond which only one label still has entries, the result can easily be determined without the need to examine every bit remaining in this last label.

Consider for example the two labels:

column	9	8	7	6	5	4	3	2	1	0	-1	Representation
	1	1	1	1	0	1	0	1	0	0	0	4 1 1 1 1 1 3
					1	1	1	1	1	1	0	6 1

Once we reach column 5, the final result can be computed in constant time no matter how many bits the first label has to the left of column 5.

To achieve this the new label is created by destroying the old labels. A new label is created, bit by bit, up to the point, as in column 5 in the example, at which all label lists but one are empty. The new label is then concatenated on the right of whatever remains of the longest label.

Consequently, it is easy to see that the proposed data structure can be used so that the work done in the label addition process is proportional to the number of labels times the length of the second longest label. We also note that the length of a label is bounded by the logarithm of the size of the subtree rooted at the node in question, and that the length of the second longest label is no longer than the logarithm of the size of the second largest subtree.

For any vertex u in $T = (V,E)$ define

$$k_2(u) = \max \{ i \mid \text{there are children } v, w \text{ of } u \text{ with } vs(T[v]) \geq vs(T[w]) \geq i \}$$

or 1 if u has less than two children. Similarly, let

$$n_2(u) = \max \{ i \mid \text{there are children } v, w \text{ of } u \text{ with } (\,|T[v]|\,) \geq (\,|T[w]|\,) \geq i \}$$

or 1 if u has less than two children.

We can deduce from the observations just made that the time complexity of the modified algorithm is

$$O\Big(\sum_{u \in V} d(u) k_2(u)\Big) = O\Big(\sum_{u \in V} k_2(u)\Big) = O\Big(\sum_{u \in V} \log n_2(u)\Big)$$

If i is any non-negative integer, the number of vertices u with $2^i \leq n_2(u) \leq 2^{i+1}$ is no more than $n/2^i$. Consequently,

$$\sum_{u \in V} \log n_2(u) \leq 2n \sum_{i \geq 1} \frac{i}{2^i} = 4n$$

So the time complexity of the modified algorithm is $O(n)$.

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 directed towards a root 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 directed 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 the original algorithm must be used to compute the vertex labels, because labels are required for every vertex and since the second algorithm computes labels in a destructive fashion, it does not produce a complete set. 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.

4. Recognizing Graphs with Fixed Vertex Separation Values

In [SAXE80], [GURA84] and [MAKE83] dynamic programming algorithms have been described which recognize graphs with fixed bandwidth or with fixed cutwidth in polynomial time. One might try the same approach for recognizing graphs with fixed values of vertex separation. However, applying the earlier methods to vertex separation does not produce a polynomial time algorithm. The reason is that, whereas graphs with bandwidth or cutwidth k have maximum vertex degree $2k$, no vertex degree constraint follows from the fact that a graph has small vertex separation. For example, for all $n \geq 1$, the star graph, i.e. the tree with a single internal vertex and n leaves, has vertex separation 1. So, if one defines an equivalence relation on partial layouts with vertex separation $\leq k$ by stipulating that layouts L and M are equivalent if and only if they have the same set of active vertices and the same set of dangling edges, as was done in the algorithms cited, the number of equivalence classes is not necessarily bounded by a polynomial in the number of vertices. We begin by describing an equivalence relation whose index is polynomial in the size of G.

4.1. Definitions

Let L be a partial layout of G and let A_L be the set of active vertices of L, as defined in Section 1. Let the ***inactive vertices*** be those in domain(L) - A_L. Let the ***active vertices at position*** j be the set of active vertices of L_j, i.e. the truncation of L to its first j vertices, as defined in Section 1.

(1) For all u in A_L let back(L, u) be the set of active vertices at position $L^{-1}(u) - 1$, i.e. the position just to the left of u.

(2) Let F_L be a function from the active vertices of L to the positive integers such that $F_L(u)$ is the smallest number of active vertices at any position in the range $L^{-1}(u)$ through |domain(L)|.

(3) Let best(L, u) be the smallest integer j, $L^{-1}(u) \leq j \leq$ | domain(L) |, such that the number of active vertices at position j is $F_L(u)$.

An example illustrating these definitions is given in Figure 4.1.

Insert Figure 4.1 here

Let $\vec{A}_L$ be the vector of vertices in A_L, say $(a_1, a_2, \cdots, a_m)$, such that $i<j$ implies $L(a_i) < L(a_j)$. Let $\vec{B}_L$ be the corresponding vector (back(L, a_1), back(L, a_2), $\cdots$, back(L, a_m)).

Removing the vertices in A_L, and all the edges incident to these vertices, from G results in a graph which we denote by $G(A_L)$. Consider a connected component C of $G(A_L)$. We will call C a *j-component* of $G(A_L)$, or simply a *j-component* when G and L are understood, if j is the smallest integer $(1 \leq j \leq m)$ such that C is a connected component of the graph obtained by deleting $a_1, a_2, \cdots, a_j$ from G. That is, C is a j-component if a_j is adjacent to some vertex in C, but no a_k, for $k > j$, is adjacent to a vertex in C.

Note that no component can contain vertices u and v such that u is in domain(L) and v is not. If so, u would be an active vertex, but these have, by definition, been deleted. Hence, if C is a j-component of $G(A_L)$, one of the following must be true:

(1) No vertex in C is in domain(L), in which case we call C an *unassigned component*,

(2) All vertices of C are in domain(L) and lie to the right of a_j, in which case we call C a *forward j-component*,

(3) All vertices of C are in domain(L) and at least one vertex of C lies to the left of a_j, in which case we call C a *back component*.

Note that, if C is a back j-component, at least one vertex of C must be in back(L, a_j). Hence, if, for two partial layouts L and L', $\vec{B}_L = \vec{B}_{L'}$, then an unassigned component of L can not be a back component of L'.

The function F_L describes the ability of a partial layout to absorb unassigned components as forward components at intervals between consecutive vertices, without the vertex separation increasing beyond some bound. $F_L(a_i)$ is the minimum number of active vertices at a position to the right of a_i. A position, such as best(L, a_i), at which there are $F_L(a_i)$ active vertices, is an optimal position for absorbing components.

Let $R(L)$ be the partial layout of G obtained from L by removing all vertices in forward components from domain(L) and leaving all remaining vertices in the same order as they occur in L. Note that the vertices removed from domain(L) in forming $R(L)$ are not removed from the graph, but are only moved from the set of assigned vertices to the set of unassigned vertices. That is, if $FC(L)$ denotes the set of all vertices in forward components of L, then the domain($R(L)$) = domain(L) - $FC(L)$ and, for every pair of vertices x, y in domain($R(L)$), $R(L)(x) < R(L)(y)$ iff $L(x) < L(y)$. We will say that a partial layout L is ***proper*** if, for all x in A_L, $F_L(x) = F_{R(L)}(x)$.

We define the equivalence relation $E(G)$ on partial layouts L and M of G by: (L, M) is in $E(G)$ if $(\vec{A}_L, \vec{B}_L, F_L) = (\vec{A}_M, \vec{B}_M, F_M)$. In constructing an algorithm to determine if a graph with n vertices has vertex separation k, we never need to consider partial layouts with more than k active vertices. There are $O(n^k)$ different ways to choose a vector of at most k active vertices, $O(n^k)$ different ways to choose a set back(L, a), for each active vertex, and hence $O(n^{k^2})$ different ways to choose a vector of back sets corresponding to all active vertices and $O(n^{k^2}) = O(1)$ ways to choose the function F from the chosen set of at most k active vertices to a non-negative integer $\leq k$. Consequently there are $O(n^m)$ different equivalence classes in the equivalence relation $E(G)$, where $m = k^2 + k$. Our vertex separation algorithm is going to examine no more than one layout per equvalence class in $E(G)$. As the number of equivalence classes is polynomial, we are able to derive a polynomial time algorithm. We show in the next section that the relation $E(G)$ satisfies the necessary properties for the construction of a dynamic programming algorithm.

We define two ***expansion*** operations to be performed on partial layouts, called ***vertex addition*** and ***component absorption***. Let L be an arbitrary partial layout and y a vertex not in domain(L). The ***vertex addition*** process creates the partial layout $L+y$, by adding y to the right end of L. It is defined by: $(L+y)(x) = L(x)$, for all x in domain(L), and $(L+y)(y) = |\text{domain}(L)| + 1$.

Component absorption is a process whereby a component is inserted between nodes in a partial layout. Absorbing a component C may result in some of the active vertices of L becoming inactive. Consequently, for any k and m, it may be possible to absorb a component with vertex separation greater than m into an interval with vertex separation $k-m$ and create a partial layout with vertex separation $\leq k$.

Let I denote the subset of A_L which contains all vertices that become inactive through the absorption of C. Define the ***I-augmentation of C***, denoted by $C[I]$, to be the graph such that:

(1) the set of vertices is vertices $(C) \cup I$ and

(2) the set of edges is edges$(C) \cup \{ \{y,z\} \mid y$ in I and z in vertices (C) such that $\{y,z\}$ is an edge in $G\}$.

That is, $C[I]$ is the graph obtained from C by adding the vertices in I and adding those edges of G which connect vertices in I with vertices in C.

An *I-anchored layout of* C is a layout of $C[I]$ in which the vertices in are assigned arbitrarily to the first $|I|$ integers. The *I-anchored vertex separation of* C, denoted I-$vs(C)$, is $\min\{vs(C[I], L) \mid L$ is an I-anchored layout of $C\}$. Note that, as the vertices in I are not connected to each other in $C[I]$, their order relative to each other has no effect on vertex separation. Observe that the vertex separation of a graph G, as defined earlier, is simply the anchored vertex separation where the set I is empty, i.e. $vs(G) = \emptyset$-$vs(G)$. Observe that a component C whose absorption into a partial layout L makes the set of vertices I inactive, can have I-anchored vertex separation $m + |I|$ and be absorbed into an interval with vertex separation $k-m$ to produce a new partial layout with vertex separation k.

Component absorption can now be defined as follows:

(1) Choose an unassigned i-component C of G under L, for some $i \geq 1$,

(2) Choose position j = best(L, a_i),

(3) Increment the position of all vertices in domain(L) to the right of position j by $|C|$,

(4) Lay out the vertices in C, using positions in the range $j+1$ to $j + |C|$, by laying out the vertices of $C[I]$ in an order given by an I-anchored layout with vertex separation less than some given bound, where I is the subset of A_L that become inactive through the absorption of C.

Let $L{*}C$ denote the partial layout obtained from L by absorbing the component C.

Let L be a partial layout of G. A partial layout L' is obtained from L by *forward component realignment* if every forward component of L is removed from domain(L) and absorbed through component absorption. In this case we assume theat each forward component is laid out in the same order as in the layout L, which is not necessarily an optimal order. Observe that forward component realignment never increases the vertex separation of a partial layout, as component absorption always places a component into an optimal position for vertex separation. Let $CR(L)$ denote the partial layout obtained from L by performing forward component realignment.

Notice that if, for every forward *j-component* C, the vertices of C are in contiguous positions of the partial layout, then the partial layout must be proper. For in such a case, for each active vertex u, the positions just before and just after a component have the same number of active vertices, as does the common interval created by the component's deletion. It follows that, for any partial layout L, $CR(L)$ is proper.

4.2. Supporting Lemmas

We develop two lemmas. We say that a partial layout L can be ***expanded*** if there is a sequence of partial layouts $L_1, L_2, \cdots, L_p$ $(p \geq 1)$ such that

(1) $L_1 = L$,

(2) for all i $(1 \leq i < p)$, L_{i+1} is obtained from L_i by an expansion step, i.e. by vertex addition or component absorption,

(3) for all i $(1 \leq i \leq p)$, L_i has vertex separation at most k, and

(4) L_p is a layout of the entire graph.

Lemma 4.1 states conditions sufficient to guarantee that, if L and M are partial layouts and L can be expanded, then M can also be expanded. Lemma 4.2 states that at most one proper partial layout per equivalence class need be considered.

Lemma 4.1 Let L and M be partial layouts of G such that the following conditions are satisfied:

(1) domain(L) $\subseteq$ domain(M),

(2) $\vec{A}_M$ is a subsequence of $\vec{A}_L$, i.e. A_M is a subset of A_L and the vertices in A_M are in the same order in $\vec{A}_M$ as they are in $\vec{A}_L$,

(3) for all vertices u in A_M, $F_L(u) = F_M(u)$ + (the number of vertices in A_L to the left of best(L, u) that are not in A_M).

If L can be expanded then M can be expanded.

Proof Let $L = L(1), L(2), \cdots, L(p) = L'$ be a sequence of partial layouts with vertex separation at most k such that, for all i, $1 \leq i \leq p$, $L(i+1)$ is obtained from $L(i)$ by an expansion step and L' is a layout of all of G. We show that there is a sequence $M = M(1), M(2), \cdots, M(p) = M'$ of partial layouts with vertex separation at most k such that $M(i+1)$ is either equal to $M(i)$ or is obtained from $M(i)$ by an expansion step and M' is a layout of all of G. The sequence $M = M(1), M(2), \cdots, M(p) = M'$ is obtained by applying the same sequence of expansion steps used to obtain the sequence $L = L(1), L(2), \cdots, L(p) = L'$, except that expansion steps that add vertices or components already in domain(M) are omitted. We show that the following conditions hold for all i, $1 \leq i \leq p$, by induction:

(1) domain($L(i)$) $\subseteq$ domain($M(i)$),

(2) $\vec{A}_{M(i)}$ is a subsequence of $\vec{A}_{L(i)}$,

(3) for all vertices u in $A_{M(i)}$, $F_{L(i)}(u) = F_{M(i)}(u)$ + (the number of vertices in $A_{L(i)}$ to the left of best($L(i)$, u) that are not in $A_{M(i)}$).

By our hypotheses, these conditions are satisfied for $i = 1$. We assume now that the four conditions are satisfied for $M(i)$ and show that they are satisfied for $M(i+1)$. There are three cases. Either $M(i+1)$ is obtained from $M(i)$ by vertex addition or by component absorption or $M(i+1) = M(i)$.

Case 1 $M(i+1) = M(i) + y$.

Hence $L(i+1) = L(i) + y$. Domain($L(i)$) $\subseteq$ domain($M(i)$) implies domain($L(i+1)$) $\subseteq$ domain($M(i+1)$). Any vertex in $A_{L(i)}$ that becomes inactive because of the addition of y to $L(i)$ must also become inactive after the addition of y to $M(i)$. Hence, $\vec{A}_{M(i+1)}$ is a subsequence of $\vec{A}_{L(i+1)}$. As noted, active vertices in $L(i)$ that become inactive through the addition of y are also active vertices of $M(i)$ and become inactive through the addition of y. Thus, for every vertex u in $A_{M(i+1)}$, $F_{L(i+1)}(u) = F_{M(i+1)}(u)$ + (the number of active vertices in $L(i+1)$ to the left of best($L(i+1)$, u) that are not active in $M(i+1)$). It is possible that the position of best(L, u), for an active vertex u, changes with vertex addition. If so, it is because the number of active vertices in the partial layout being is smaller than at the previous best location. It follows, as $\vec{A}_{M(i+1)}$ is a subsequence of $\vec{A}_{L(i+1)}$, that in this case also we have the desired relation between $F_{L(i+1)}$ and $F_{M(i+1)}$.

If y is active in $M(i)+y$, then it is also active in $L(i)+y$, as domain $(L(i))$ is a subset of domain $(M(i))$. So, $\vec{A}_{M(i+1)}$ is a subsequence of $\vec{A}_{L(i+1)}$. Also $F_{L(i+1)}(y) = F_{M(i+1)}(y) +$ (the number of active vertices of $L(i+1)$ to the left of best($L(i+1)$, y) that are not active in $M(i+1)$), as best($L(i+1)$, y) is clearly the last interval in the layout.

Case 2 $M(i+1) = M(i)*C$.

Then, $L(i+1) = L(i)*C$. Domain($L(i)$) $\subseteq$ domain(M(i)) implies domain($L(i+1)$) $\subseteq$ domain($M(i+1)$). Any active vertex of $L(i)$ that becomes inactive because of the absorption of C into $L(i)$ must also become inactive after the absorption of C into $M(i)$. Hence, $\vec{A}_{M(i+1)}$ is a subsequence of $\vec{A}_{L(i+1)}$.

For every vertex u in $A_{M(i+1)}$, $F_{L(i+1)}(u) = F_{M(i+1)}(u) +$ (the number of active vertices in $L(i+1)$ to the left of best($L(i+1)$, u) that are not active in $M(i+1)$).

Case 3 $M(i+1) = M(i)$.

Then, $L(i+1)$ is obtained by either absorbing a component C or adding a vertex y that is already in domain($M(i)$). Consequently, domain($L(i+1)$) $\subseteq$ domain($M(i)$) $=$ domain($M(i+1)$). Let u be an active vertex of $L(i)$ that becomes inactive through the addition of y or through the absorption of C. It follows that all remaining unassigned neighbors of u are put into domain($L(i)$) to make $L(i+1)$. Since these neighbors of u that are added to $L(i)$ to obtain $L(i+1)$ are already in $M(i)$, the vertex u is already inactive in $M(i) = M(i+1)$. It follows that $\vec{A}_{M(i+1)}$ is a subsequence of $\vec{A}_{L(i+1)}$. By hypothesis, for every active vertex u in $A_{M(i)}$, $F_{L(i)}(u) = F_{M(i)}(u) +$ (the number of active vertices in $L(i)$ to the left of best($L(i)$, u) that are not active in $M(i)$). Active vertices in $L(i)$ that become inactive through the addition of y or the absorption of C simply decrease, for a given active vertex u, the number of active vertices of $L(i)$ to the left of best($L(i)$, u) that are not active in $M(i) = M(i+1)$. Thus, for every active vertex u in $A_{M(i+1)}$, $F_{L(i+1)}(u) = F_{M(i+1)}(u) +$ (the number of active vertices in $L(i+1)$ to the left of best($L(i+1)$, u) that are not active in $M(i+1)$). ■

Let $CR(L)$ denote the partial layout obtained from L by performing forward component realignment, as defined in Section 4.1. Remember that, for any partial layout L, $CR(L)$ is proper.

Corollary 4.1 For any expansible partial layout L, $CR(L)$ is expansible.

Proof Let $M = CR(L)$. Observe that (1) domain(M) = domain(L), (2) $\vec{A}_M = \vec{A}_L$, and (3) for all active vertices u, $F_M(u) \leq F_L(u)$. Then, by the proof of Lemma 4.1, using cases (1) and (2) only, it can be seen that, if L is expansible, then M is expansible. ■

Lemma 4.2 Let L and M be partial layouts of G in the same equivalence class of $E(G)$ with $vs(G, L) \leq k$ and $vs(G, M) \leq k$. Then, there is a collection of connected components $S = \{C_1, C_2, \cdots, C_m\}$ of $G(A_M)$ such that all components in S can be absorbed into M without exceeding vertex separation k resulting in a partial layout P with:

(1) domain(L) $\subseteq$ domain(P),

(2) $\vec{A}_P$ is a subsequence of $\vec{A}_L$, and

(3) for all $u \in A_P$, $F_L(u) = F_P(u)$ + (the number of active vertices of L to the left of best(L, u) that are not active in P)

Proof We show that vertices in domain(L) but not in domain(M) can be added to domain(M) through component absorptions to produce a partial layout P such that:

(1) domain(L) is a subset of domain(P)

(2) $\vec{A}_P$ is a subsequence of $\vec{A}_L$ and the active vertices of L that are not active in P are in domain(P),

(3) for all active vertices u in A_P, back(L, u) = back(P, u) $\cup$ $\{x \in$ back(L, u) $\mid x \in A_L\}$, and

(4) for all active vertices u in A_P, $F_L(u) = F_P(u)$ + (the number of active vertices of L to the left of best(L, u) that are not active in P).

Observe that conditions (2) through (4) are satisified initially when M is substituted for the partial layout P, because M and L are in the same equivalence class of $E(G)$. We shall see that after each component is absorbed we get a new partial layout that satisfies conditions (2) through (4). Thus, the argument can be iterated until all vertices $x \in$ domain(L) - domain(M) have been absorbed.

Let x be any vertex in domain(L) - domain(M). Vertex x cannot be active, as all vertices active in L are in domain(M). Consider the connected component C_x containing x in the graph $G(A_L)$ obtained by deleting the active vertices of L from G. We have already observed that such a component C_x must be entirely in or entirely out of the domain of the partial layouts L and M, because it contains no active vertices. Hence, all vertices in C_x must be in domain(L) - domain(M).

Let C_x be a j-component of G under L, for some $j \geq 1$. C_x must be laid out in L so that all of its vertices lie to the right of the j^{th} active vertex, say a_j, of L. If any vertex of C_x lay to the left of a_j, then some vertex, say y, of C_x would be in back(L, a_j). However, since back(L, a_j) = back(M, a_j) $\cup$ $\{x \in$ back(L, a_j) $\mid x \in A_L\}$, it follows that y would be in domain(M). As no vertex of C_x is in domain(M), this is a contradiction. So, C_x is a forward component in the layout L.

Let L' be the partial layout obtained by removing C_x from the domain of L. As L is proper, $F_L(a_j) = F_{L'}(a_j)$. Consequently, as L has vertex separation at most k, C_x must have vertex separation at most $k - F_L(a_j)$. As $F_M(a_j) \leq F_L(a_j)$, C_x can be absorbed as a forward component into the layout M in the position best(M, a_j) which has $F_M(a_j)$ active vertices. $M' = M{*}C_x$ may have a smaller set of active vertices than L, as some of the active vertices of M may no longer be active when C is absorbed. However, no new active vertices are created by absorbing the connected component C_x, so $\vec{A}_{M'}$ is a subsequence of $\vec{A}_L$. All active vertices of M that become inactive through the absorption of C_x are no longer in the back sets of active vertices of M' to the right of the point of absorption. All other back sets are unchanged by the absorption of C_x. By hypothesis, for all active vertices u in M, back(L, u) = back(M, u) $\cup$ $\{x \in$ back(L, u) $\mid x \in A_L\}$. Thus, for all active vertices u in M', back(L, u) = back(M', u) $\cup$ $\{x \in$ back(L, u) $\mid x \in A_L\}$. By hypothesis, for every active vertex $x \in A_M$, $F_L(x) = F_M(x)$ + (the number of active vertices in L to the left of best(L, x) that are not active in M). The absorption of the component C_x into M may cause some of the active vertices of M to become inactive, but that will simply increase the second term on the right side of the equality. Thus, for every active vertex x in $A_M{}'$, $F_L(x) = F_{M'}(x)$ + (the number of active vertices in L to the left of best(L, x) that are not active in M').

So, conditions (2) through (4) are satisfied when M' is substituted for M. Thus, the argument can be repeated until obtaining a partial layout $P = M{*}C_1{*}C_2{*} \cdots {*}C_k$, where $C_1, C_2, \cdots, C_k$ are all the forward components of L that are not in domain(M). As all vertices in domain(L) - domain(M) will be absorbed in components to create P, domain(L) is a subset of domain(P). So, conditions (1) through (4) are satisfied.

Corollary 4.2 Let L and M be proper partial layouts of G in the same equivalence class of $E(G)$ with $vs(G, L) \leq k$ and $vs(G, M) \leq k$. If L can be expanded, then M can be expanded.

Proof By Lemma 4.2 we can expand M to a partial layout P such that:

(1) domain(L) $\leq$ domain(P),

(2) $\vec{A}_P$ is a subsequence of $\vec{A}_L$, and

(3) for all $u \in A_P$, $F_L(u) = F_P(u)$ + (the number of active vertices of L to the left of best(L, u) that are not active in P).

Then, by Lemma 4.1, as L can be expanded, P can be expanded. As P is an expansion of M, M can also be expanded. ■

4.3. A Vertex Separation Algorithm for Graphs

An algorithm is described which decides, for a graph G, a set of vertices I in G, and a positive integer k, whether G has an I-anchored layout with vertex separation at most k. The algorithm is defined below as a function, called separation(G, I, k). It uses the equivalence classes induced by the relation $E(G)$. Given some partial layout L of G the procedure generates new partial layouts both by absorbing unassigned components of $G(A_L)$ and by adding each unassigned vertex. The partial layouts produced by these processes are considered further only if they are in a previously unexplored equivalence class.

The algorithm uses a stack S, which contains partial layouts of G, and a Boolean array T such that $T(\vec{A}, \vec{B}, F)$ = true if and only if the equivalence class $(\vec{A}, \vec{B}, F)$ has been previously considered. Initially $T(\vec{A}, \vec{B}, F)$ = false, for all $(\vec{A}, \vec{B}, F)$, and the stack S contains only the partial layout L_I, which has the set I as its domain, the vertices in I being mapped to the integers 1, ..., $|I|$ arbitrarily.

```
function separation($G$, $I$, $k$): Boolean;
begin
        empty($S$); push($L_I$, $S$); successful := false;
        searchlayout($G$, $I$, $k$); separation := successful
end;


procedure searchlayout ($G$, $I$, $k$);
begin $L$ := pop($S$);
if $A_L \neq \emptyset$ then begin
    {Let $Y$ be the set of all unassigned connected components of $G(A_L)$}
    absorb := true;
    while $Y \neq \emptyset$ and absorb do {Absorb as many components as possible}
    begin X := Y; $Y := \emptyset$; absorb := false;
        while X $\neq \emptyset$ do
        begin Remove a connected component $C$ from $X$;
            {Let $I'$ be the set of vertices in $A_L$ that become inactive after the absorption of $C$}
            {Let $\vec{A}_L = (a_1, a_2, \cdots, a_m)$ and let $C$ be an $j$-component of $G$ under $L$}
            if $F_L(a_j) > |I'|$ and separation $(C[I'], I', k - F_L(a_j) + |I'|)$
            then begin absorb := true; $L := L*C$ end else $Y := Y \cup \{C\}$
    end end end;
if $L \neq L_I$ and $A_L = \emptyset$ then successful := true
else for each vertex $x$ not in domain($L$) do {Add each remaining vertex}
    begin $L' := L + x$; $M := CR(L')$;
    if not $T(\vec{A}_M, \vec{B}_M, F_M)$ and $|A_{L'}| \leq k$
    then begin $T(\vec{A}_M, \vec{B}_M, F_M)$ := true;
push($M$, $S$) end
end end;
```

Lemma 4.3 The algorithm separation(G , I , k) is correct.

Proof Let G be a finite undirected graph, I a set of vertices of G and k a positive integer. First we show that, if there is an I-anchored vertex separation $\leq k$ layout of G , then separation(G , I , k) will terminate with the value true. Observe that the algorithm on such arguments places the partial layout L_I on the stack S and calls the algorithm searchlayout(G , I , k). The initial partial layout on the stack, namely $L = L_I$, can be expanded by hypothesis. We show that when an expansible partial layout L is removed from the stack, the algorithm either halts, returning the value true, or places at some step an expansible proper partial layout, say M , on the stack such that domain(M) properly contains domain(L). Observe that the algorithm searchlayout first absorbs into L all unassigned components that can be absorbed without making the vertex separation greater than k . The result is a partial layout, say L^*, which together with L , satisfies Lemma 4.1. Thus, L^* is expansible.

Note that, for some unassigned j-component C and set of vertices I' which become inactive by the absorption of C, if $|I'| \geq F_L(a_j)$, then all active vertices at position best(L , a_j) become inactive. This is a contradiction, unless best(L , a_j) is the rightmost position in the layout, because, if we eliminate all active vertices then no edges can connect vertices to the right of the position to those to its left and the graph is assumed to be connected. Furthermore, if best(L , a_j) is the rightmost position, then we can use vertex addition and not component absorption. So, we assume that $F_L(a_j) > |I'|$ and hence $k - F_L(a_j) + |I'| < k$.

As no more component absorptions can be done to L^* and L^* is expansible, it must be possible to expand L^* by first doing a vertex addition step, say to obtain $L' = L^* + y$. The algorithm tries all possible vertex additions and therefore considers the partial layout L'. The algorithm then computes $M = CR(L')$ which is proper. Notice that, by Corollary 4.1, M is expansible, as L' is expansible. In particular, M has at most k active vertices. The algorithm does one of the following:

(1) it places M on its stack, or

(2) it observes that some other partial layout, say Q , in the same equivalence class as M , has already been placed on the stack.

In (1), as domain(L) is a proper subset of domain(M), placing M on the stack establishes our claim. In (2) we note that either Q is removed from the stack at some time or the algorithm halts with a positive answer. Observe that, by Corollary 4.2, Q is expansible. If Q is removed from the stack, then the algorithm absorbs all possible components into Q to obtain a partial layout Q^*. By Lemma 4.1, as Q is expansible, Q^* is expansible. As all components are absorbed into Q to form Q^* and M and Q are in the same equivalence class, by Lemma 4.2, domain(M) is a subset of domain(Q^*). As Q^* is expansible and all component absorptions have been done, Q^* can be expanded by a vertex addition, say to $Q' = Q^* + z$. The algorithm tries all possible vertex additions, so it generates $Q' = Q^* + z$. It then computes $M' = CR(Q')$, which is proper. Note that M' is expansible, as Q' is expansible. In particular, M' has at most k active vertices. Furthermore, domain(M') properly contains domain(M). Now, substituting M' for M, we can repeat the argument but with a partial layout M' with a larger domain than M. Hence, at some point we run out of vertices, and hence the algorithm halts with a positive answer, or an expansible partial layout with larger domain than L is placed on the stack. The reader may observe that the partial layout L' in the algorithm is not in the same equivalence class as L. This is because $L' = L + x$ and no more component absorptions can be done to L, hence x must be a new active vertex. That is, if x were not active, then it would be a component consisting of an isolated vertex and hence have vertex separation 0. Such components can always be absorbed, which would contradict the assumption that L has no more possible component absorptions.

It follows that, if there is a layout of all of G with vertex separation at most k, then the algorithm will terminate, returning the value true. On the other hand, if the algorithm returns the value true, then it has found a layout with vertex separation k. So, the algorithm returns the value true if and only if the graph has vertex separation k. ■

4.4. Implications

Because of the relationships between the vertex separation problem and the search number and pebbling problems previously described, the algorithm given for vertex separation can be used to measure these properties also. Hence we can say something about the complexity of all three problems.

Theorem 4.1 For each positive integer k, graphs with vertex separation at most k can be recognized in

$O(n^m)$ steps, where $m = k^2 + 2k + 4$.

Proof For reasons of efficiency we replace the recursive step, i.e. the call to separation($C[I']$, I', $k - F_L(a_j) + |I'|$) in the procedure searchlayout by a table look up. We create a table, call it Look(k), which contains, for every possible set V of at most k vertices in G, and every possible connected component C of $G(V)$, every possible integer $i \leq k$, and every subset V' of V, an indication of whether $C[V']$ has a V'-anchored layout with vertex separation at most i. Having Look(k) available, we can answer the question of whether an augmented component $C[I']$ has an I'-anchored layout with vertex separation at most $k - F_L(a_i) + |I'|$ $(\leq k)$ by looking in the table rather than by executing the procedure.

Look(k) is constructed iteratively. One first removes each subset V of at most k vertices from G and determines which connected components in the resulting graph have vertex separation 0. This takes $O(1)$ time for each component, as a connected graph has vertex separation 0 iff it is an isolated vertex. Next we remove each subset V of at most $k-1$ vertices and for each subset V' of V and each connected component C of $G(V)$ we run our algorithm separation($C[V']$, V', 1), as written, except that recursive calls for components with vertex separation 0 are replaced by table look up using the values already computed. Continuing in this way we add to Look(k) information about whether components obtained by deleting at most $k-j$ vertices have vertex separation j, by using information already in the table instead of recursive calls to the procedure. That is, any recursive call will be for a component obtained by deleting some set of $k - j + t$ vertices, for some $1 \leq t < j$, and some vertex separation value less than j. All such information will already be in the table Look(k).

We thus consider for removal all possible vertex sets V of size j, for $j < k$. There are $O(n^k)$ such sets. For each such set V deleted there are $O(2^k)$, i.e. $O(1)$, subsets V' and at most $O(n)$ connected components C in the graph $G(V)$. For each such component C of $G(V)$, each subset V' of V, and each integer i, $(0 \leq i <+ k)$, we call the procedure separation($C[V']$, V', i). We assume that the recursive calls stated in the procedure are made by table look up and take $O(1)$ time. Running the algorithm results in the consideration of no more than one partial layout from each equivalence class and for each representative considered some time is used in computing new partial layouts. There are $O(n^m)$ equivalence classes, where $m = k^2 + k$. We calculate now the amount of time to compute new partial layouts for each representative. For a given partial layout L there are at most $O(n)$ unassigned connected components and each component is considered at most $O(n)$ times for absorption. The algorithm then considers all possible vertex additions to the partial layout L' obtained after all possible component absorptions. There are at most $O(n)$ unassigned vertices possible for vertex addition. For each partial layout $L' + y$ the algorithm computes $CR(L' + y)$, which takes $O(n^2)$ time. Thus, $O(n^3)$ steps are needed for computing new partial layouts from a given representative. Consequently, an upper bound on running the algorithm given the table Look(k) is $O(n^p)$ time, where $p = k^2 + k + 3$. Therefore, as the algorithm is run at most $O(n^k + 1)$ times for computing the table Look(k), the entire process requires at most $O(n^q)$ steps, where $q = k^2 + 2k + 4$.

Corollary 4.3 For each fixed value k, the problem of recognizing graphs with search number at most k can be done in $O(n^m)$ time, where $m = 2k^2 + 4k + 8$.

Proof In Section 2 we showed that a graph G has search number k iff the graph G' has vertex separation k, where G' is the graph obtained from G by subdividing each of the edges of G with two, degree 2 vertices. Since the number of vertices in G' is bounded by $O(n^2)$, where n is the number of vertices in G, the indicated bound follows from Theorem 4.1. ■

Corollary 4.4 For each fixed value k, the problem of recognizing dags with progressive black/white pebble demand at most k can be done in $O(n^m)$ steps, where $m = k^2 + 3$.

Proof The corollary follows from Theorem 4.1 and a transformation described by Lengauer [Leng81] from the black/white pebble demand problem to the vertex separation problem. Lengauer's transformation takes a directed acyclic graph G and creates an undirected graph G' whose edges are those of G, without their direction, plus edges connecting every pair of vertices that are predecessors of the same vertex in G. Lengauer shows that G has progressive black/white pebble demand k iff G' has vertex separation k-1. Since G and G' have the same number of vertices, the result follows. ■

Although we have shown a polynomial time bound, the degree of the polynomial is too large for the algorithm to be practical. We leave open the question of whether graphs with vertex separation at most k can be recognized in $O(n^k)$ steps.

References

[CHUN82] M. J. Chung, F. Makedon, I. H. Sudborough and J. Turner, "Polynomial Algorithms for the Min-Cut Linear Arrangement Problem on Degree Restricted Trees", SIAM J. Computing, 14, 1, pp. 158-177, 1985.

[GURA84] E. M. Gurari and I. H. Sudborough, "Improved Dynamic Programming Algorithms for Bandwidth Minimization and the Min-Cut Linear Arrangement Problem", J. Algorithms, 5, 4, pp. 531-546, 1984.

[KIRO83] L. M. Kirousis and C. H. Papadimitriou, "Searching and Pebbling", Technical Report, National Technical University, Athens, Greece, 1983.

[LAPA83] A. S. LaPaugh, "Recontamination does not Help to Search a Graph", Technical Report, Electrical Engineering and Computer Science Department, Princeton University, 1983.

[LEIS80] C. E. Leiserson, "Area-Efficient Graph Layouts, for VLSI", Proc. 21st Annual Symp. Foundations of Computer Science, pp. 270-281, 1980.

[LENG81] T. Lengauer, "Black-White Pebbles and Graph Separation", Acta Informatica, 16, pp. 465-475, 1981.

[LIPT79] R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs", SIAM J. Appl. Math. 36, 2, pp. 177-189, 1979.

[LIPT80] R. J. Lipton and R. E. Tarjan, "Applications of a Planar Separator Theorem", SIAM J. Computing, 9, 3, pp. 615-627, 1980.

[MAKE83] F. Makedon and I. H. Sudborough, "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, 1983.

[MEGI81] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson and C. H. Papadimitriou, "The Complexity of Searching a Graph (Preliminary Version)", Proc. 22nd Annual Symp on Foundations of Computer Science, pp. 376-385, 1981.

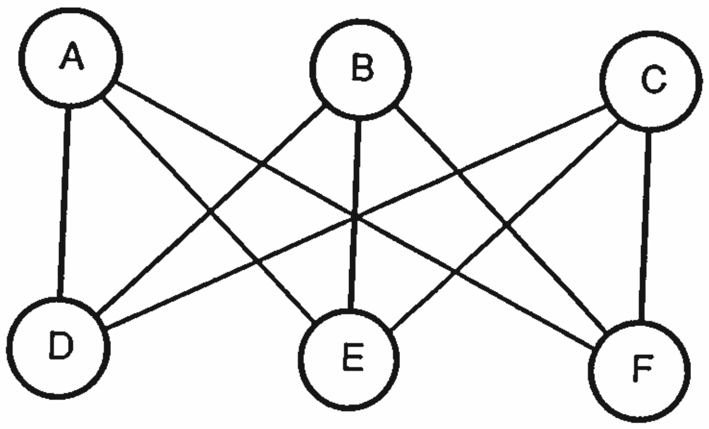
[PARS76] T. D. Parsons, "Pursuit-Evasion in a Graph", in Theory and Application of Graphs, Y. Alavi and D. R. Lick, Springer-Verlag, pp. 426-441, 1976.

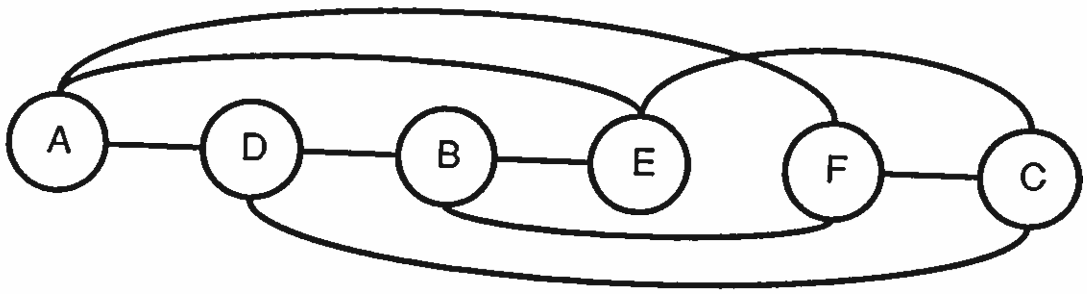
[PARS78] T. D. Parsons, "The Search Number of a Connected Graph", Proc. 9th Southeastern Conf. on Combinatorics, Graph Theory, and Computing, Utilitas Mathematica Publishing, Winnipeg, pp. 549-554, 1978.

[PHIL80] R. Philipp and E. Prauss, "Uber Separatoren in planaren Graphen", Acta Informatica, 14, 1, pp. 87-97, 1980.

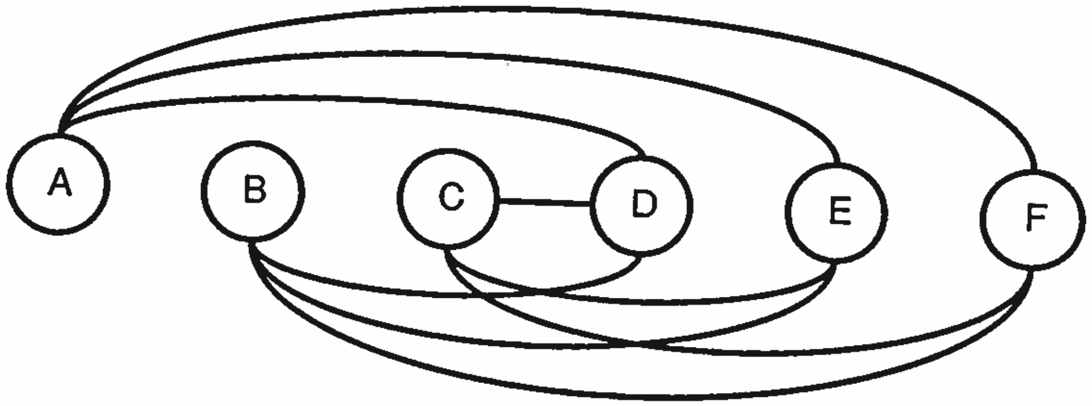
[SAXE80] J. B. Saxe, "Dynamic-Programming Algorithms for Recognizing Small Bandwidth Graphs in Polynomial Time", SIAM J. Algebraic and Discrete Methods, 1, 4, pp. 363-369, 1980.

[YANN85] M. Yannakakis, "A Polynomial Algorithm for the Min-Cut Linear Arrangement of Trees", JACM, 32, 4, pp. 950-988, 1985.



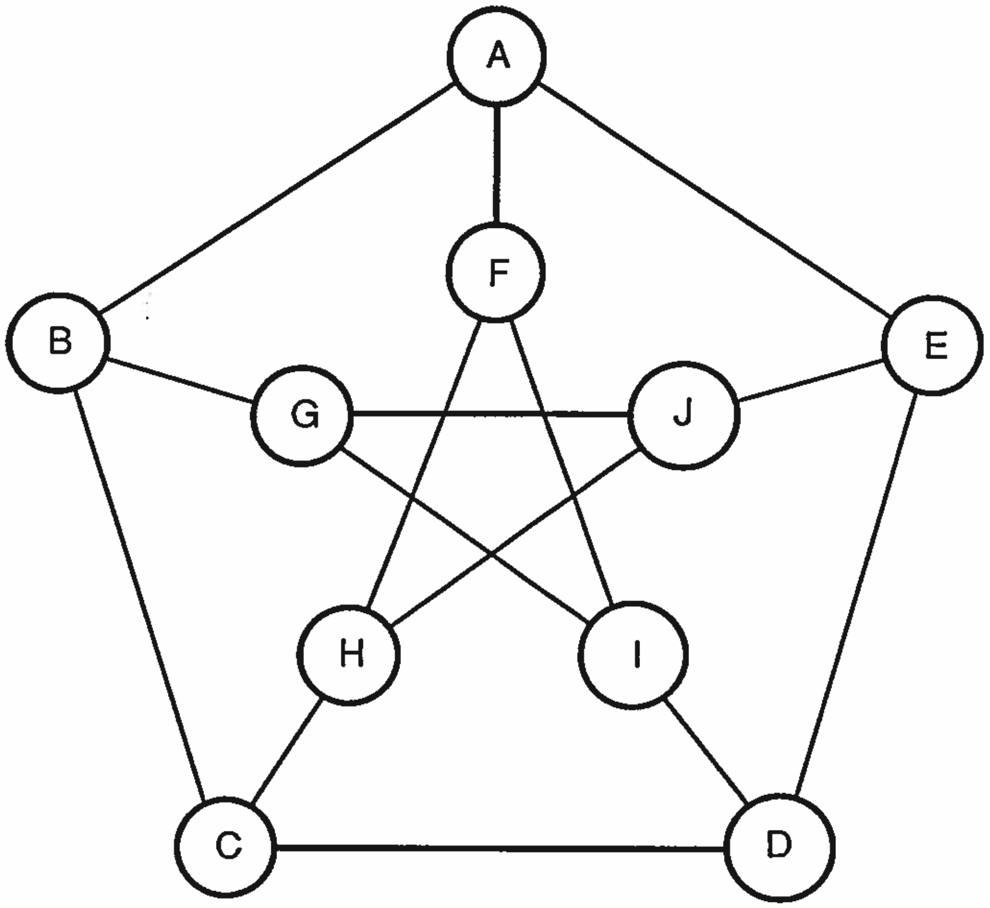


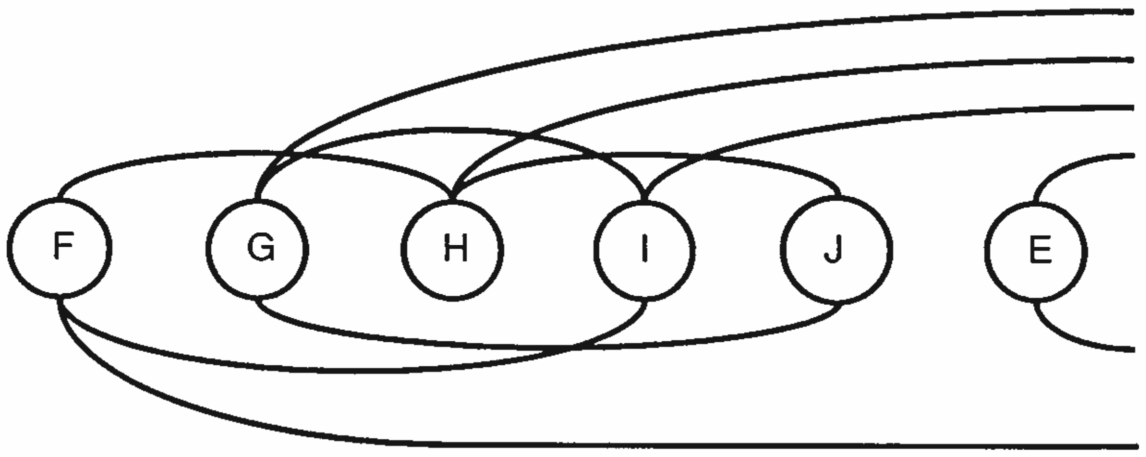
The vertex separation of this layout is 4



The vertex separation of this layout is 3

Figure 1.1 $K_{3,3}$ and Two Linear Layouts





The vector of active vertices is (F, G, H, I, E).

There are 6 dangling edges incident to these vertices.

Figure 1.2 A Partial Layout of the Petersen Graph

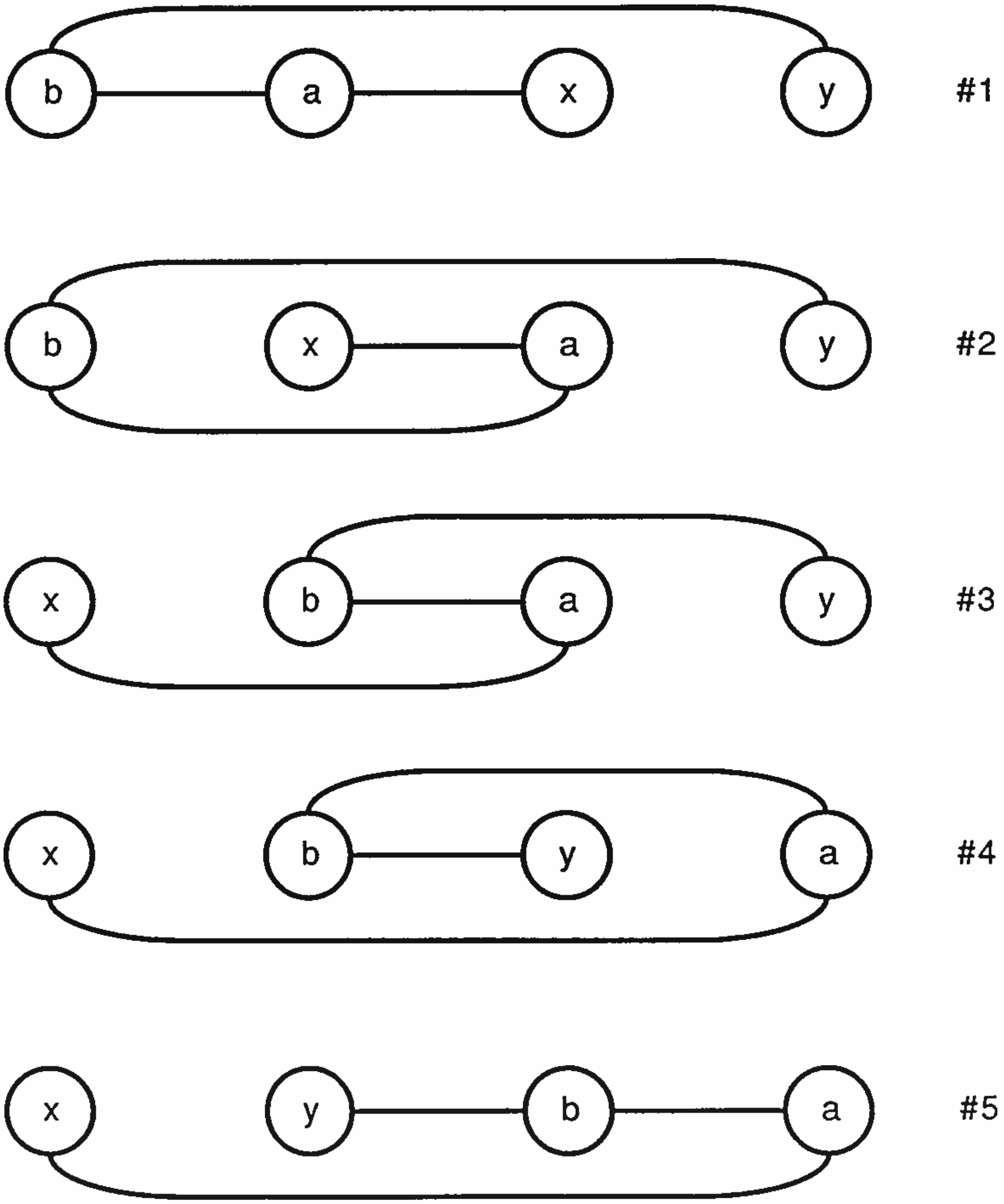


Figure 2.1. (a) Added (a, b) and Original (x, y) Vertices and Their Possible Relative Positions

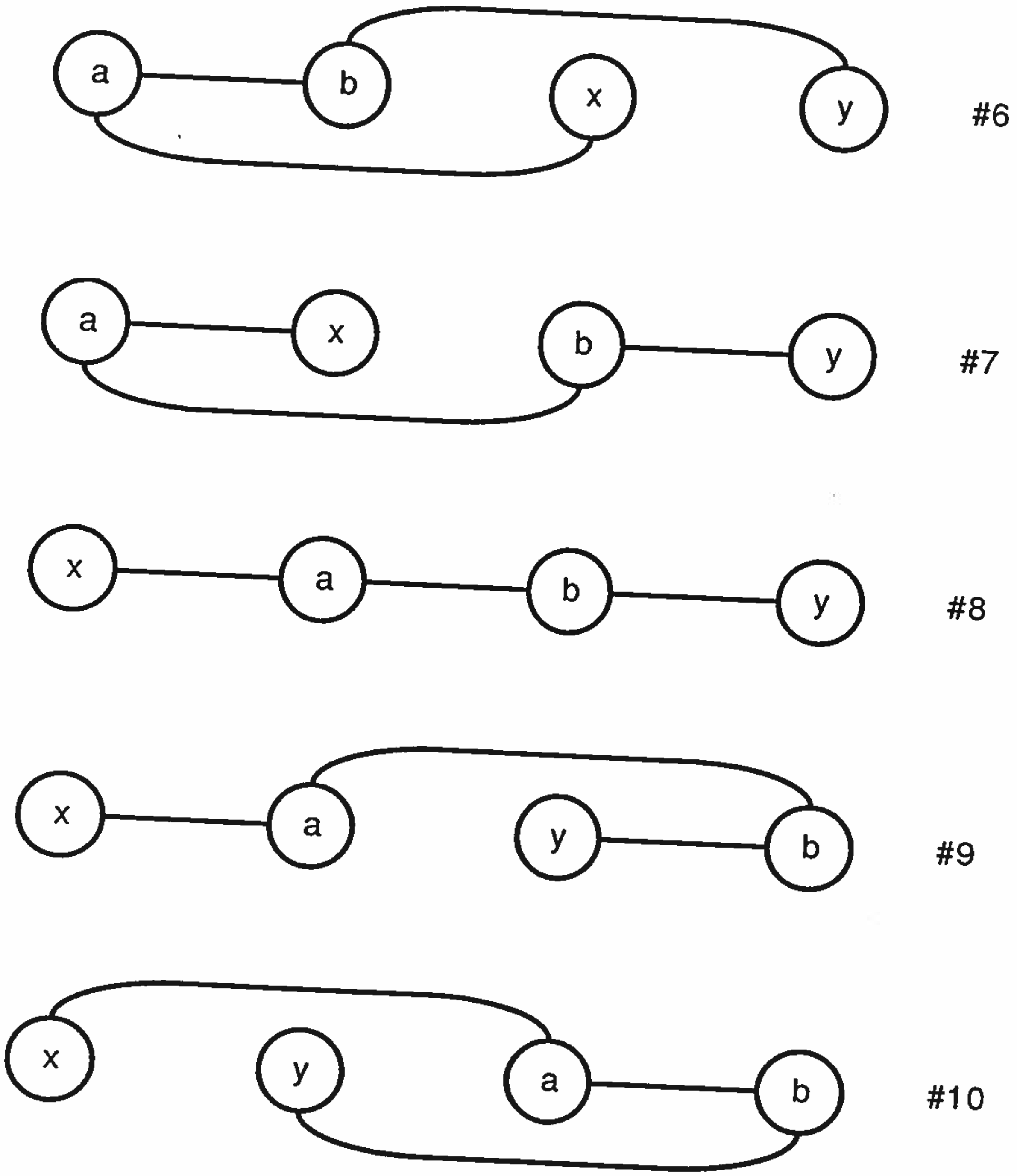


Figure 2.1. (b) Added (a, b) and Original (x, y) Vertices and Their Possible Relative Positions

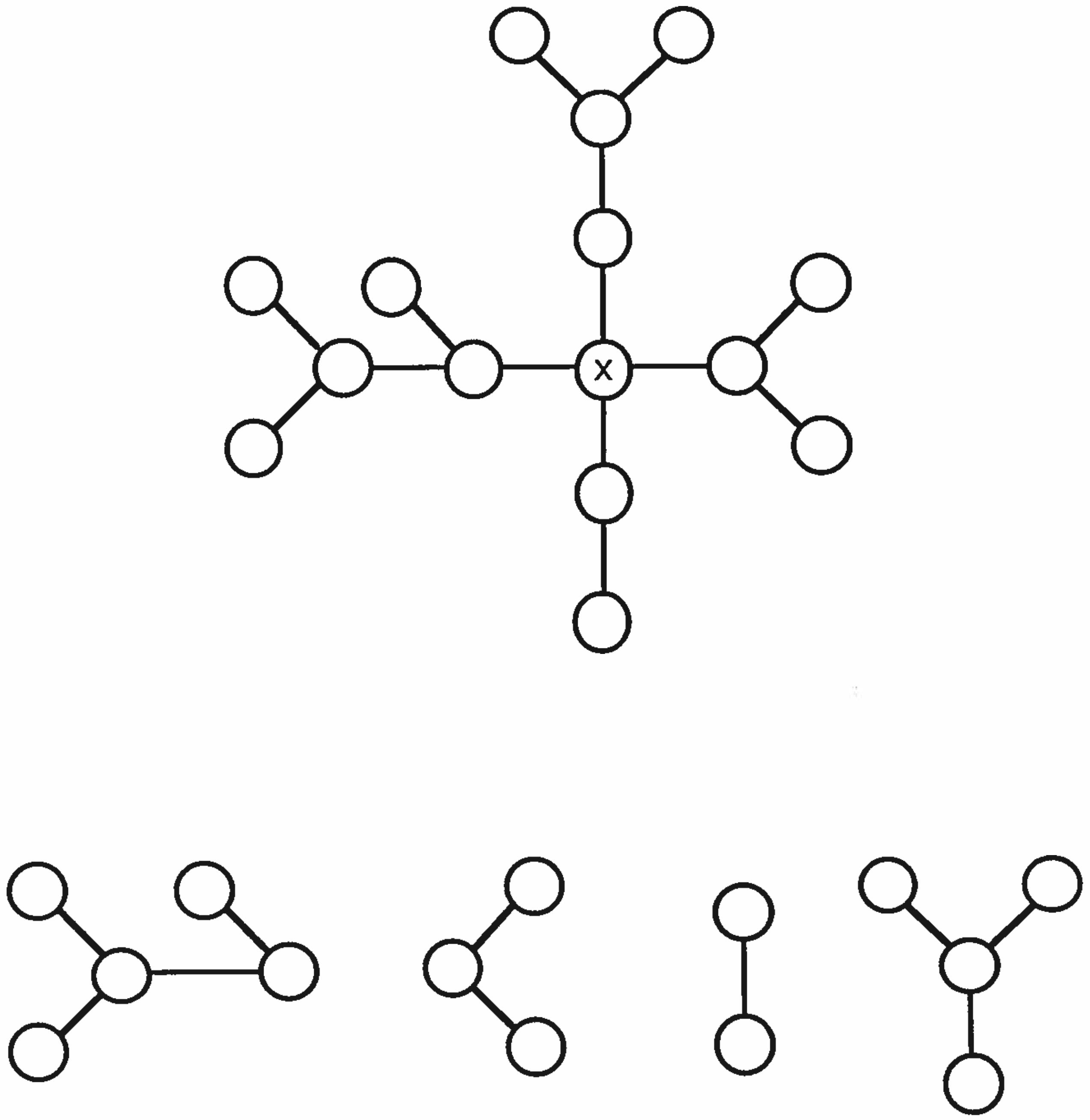


Figure 3.1 A Tree and the Trees Induced by the Vertex x.

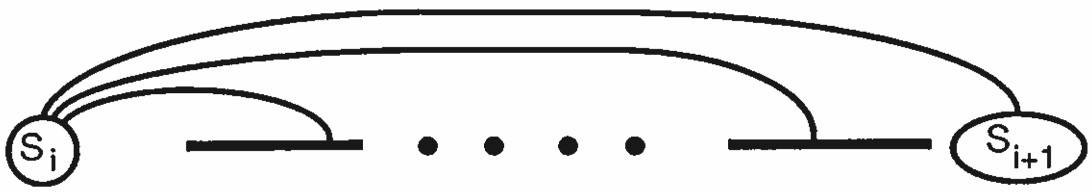


Figure 3.2 Laying Out the Subtrees of Sequence Members

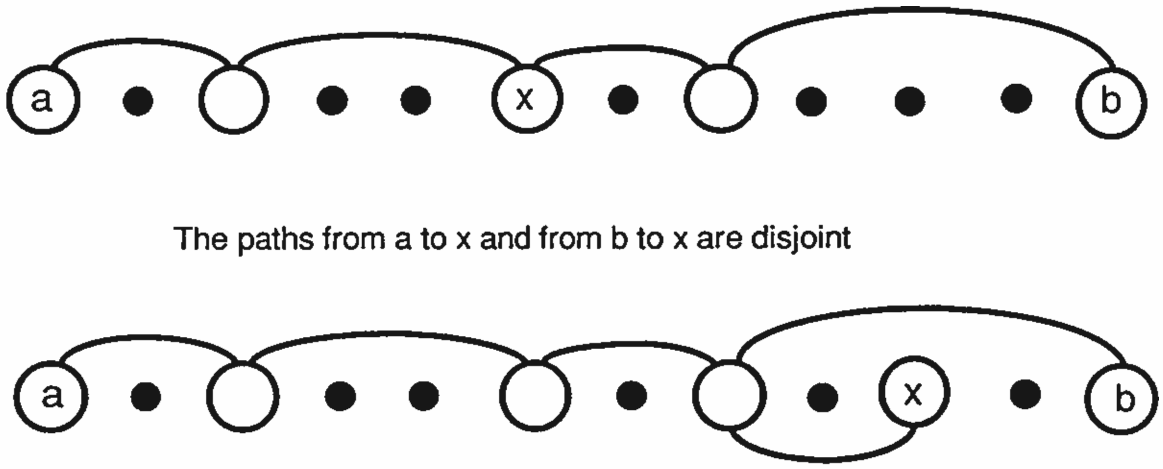
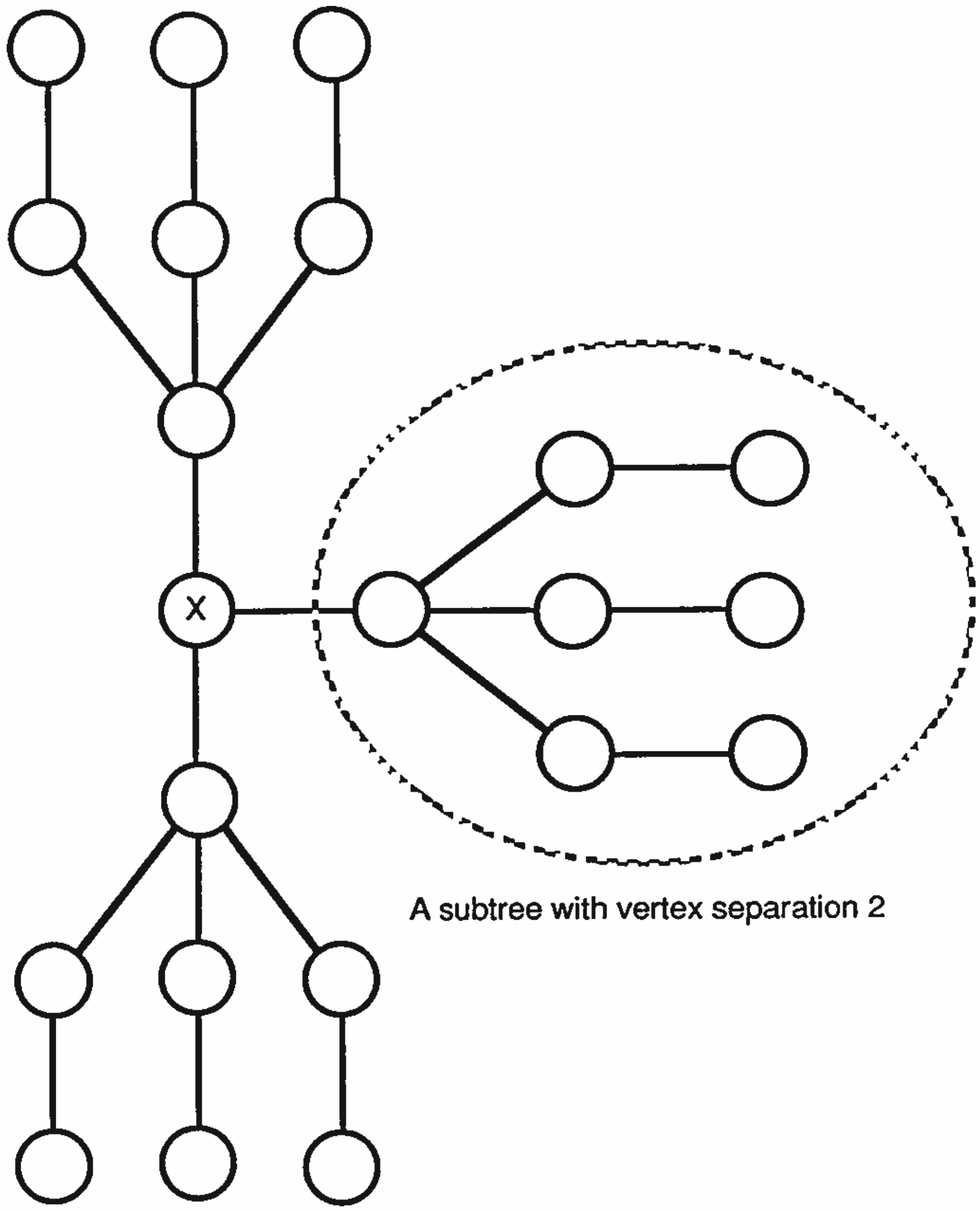
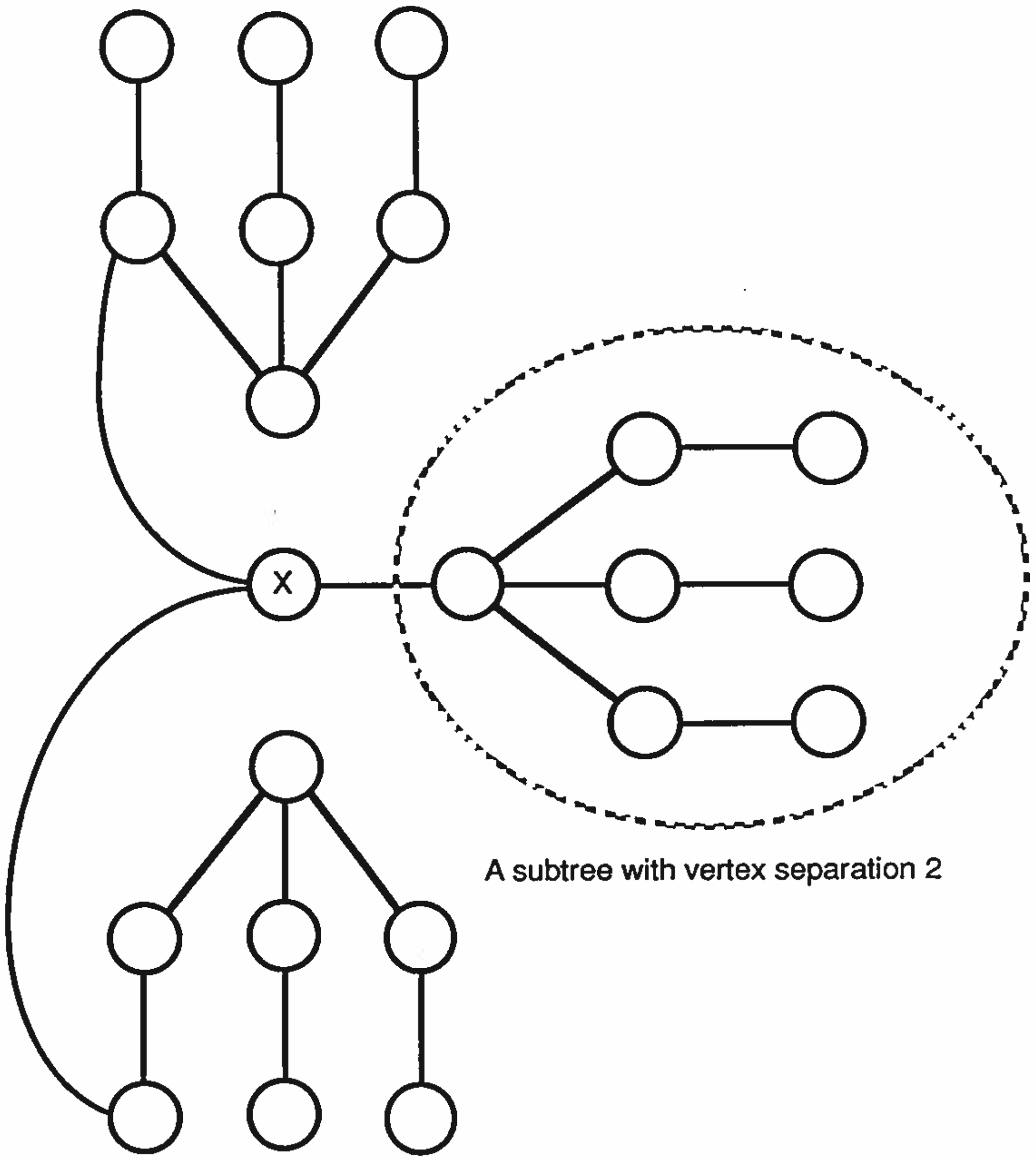

a
x
b
The paths from a to x and from b to x are disjoint
a
x
b
The paths from a to x and from b to x are not disjoint

Figure 3.3 Removing a Path from a Layout



The vertex x induces 3 subtrees with vertex separation 2

Figure 3.4 A Tree with Vertex Separation 3.



The vertex x induces 3 subtrees with vertex separation 2

Figure 3.5 A Tree with Vertex Separation 3.

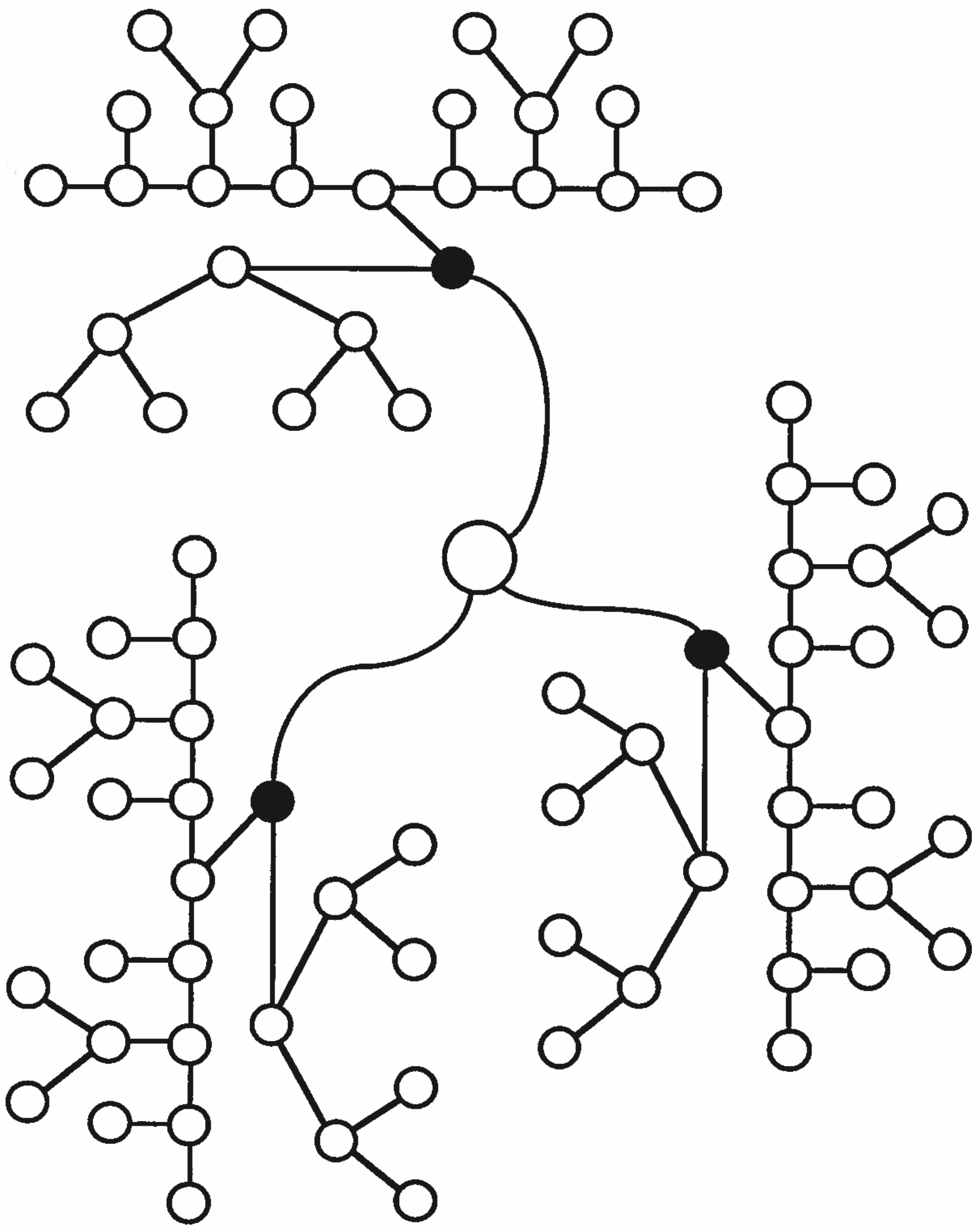


Figure 3.6 A Degree 3 Tree with Cutwidth 5 and Vertex Separation 3.

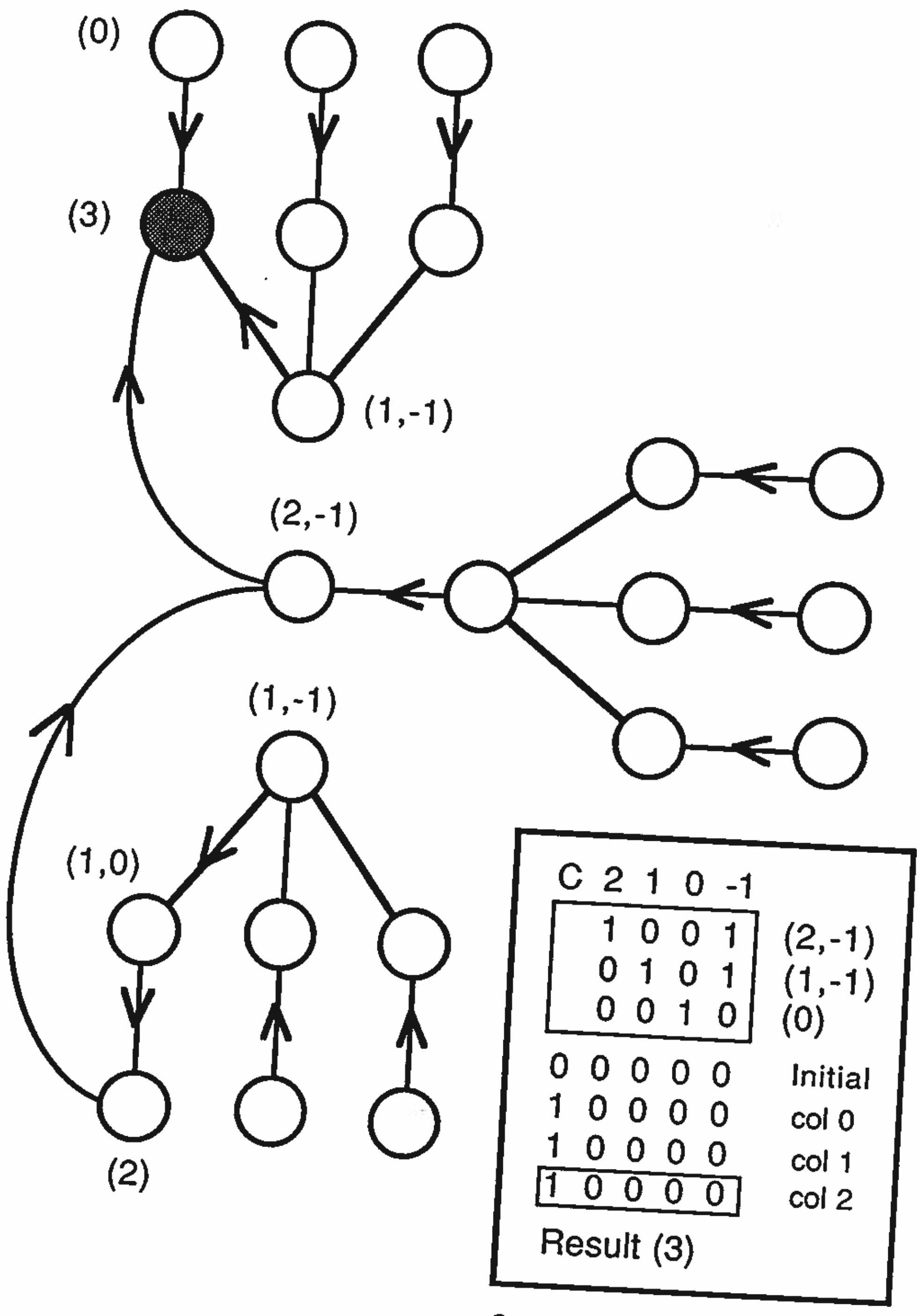
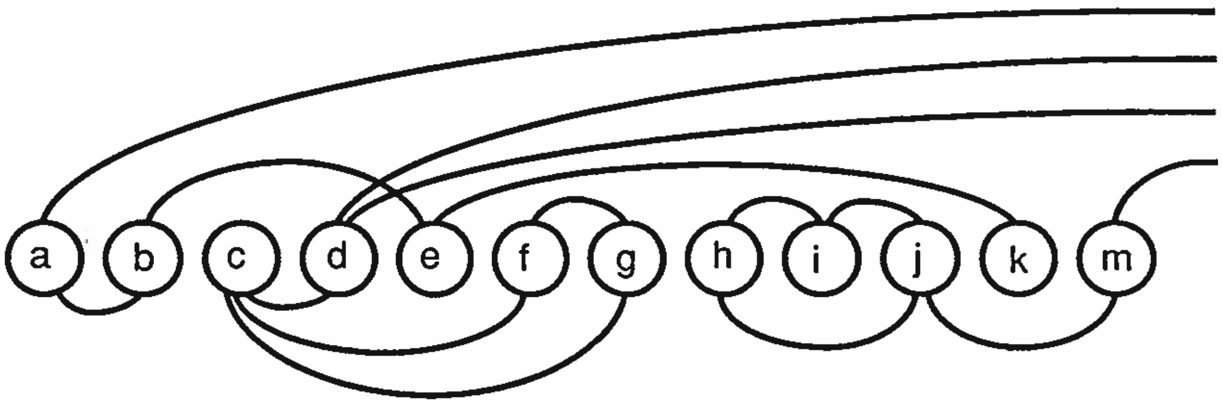


Figure 3.7 An Example of the Effect of the Labelling Algorithm



$A_L = \{a, d, m\}$

back(L, a) = { }   back(L, d) = {a, b, c}   back(L, m} = { a, d, j}

$F_L(a) = 1$ $F_L(d) = 3$ $F_L(m) = 3$

best(L, a) = 1   best(L, d) = 7   best(L, m) = 12

The graph ({b, e, k}, {{b, e},{e, k}}) is a forward 1-component

Figure 4.1 A Partial Layout, L