Computing the Vertex Separation of Unicyclic Graphs

JOHN ELLIS AND MINKO MARKOV

Department of Computer Science

University of Victoria, P.O. Box 3055

Victoria, British Columbia, V8W 3P6, Canada

May 4, 2004

Abstract

We describe an $O(n \log n)$ algorithm for the computation of the vertex separation of unicyclic graphs. The algorithm also computes a linear layout with optimal vertex separation in the same time bound. Pathwidth, node search number and vertex separation are different ways of defining the same notion. Path decompositions and search strategies can be derived from linear layouts. The algorithm applies existing, linear time, techniques for the computation of the vertex separation of trees together with corresponding optimal layouts. We reformulate the earlier work on the linear time computation of optimal layouts. A polynomial time algorithm for the problem on unicyclic graphs can be inferred from existing more general methods for graphs of fixed treewidth, since unicyclic graphs have treewidth two, but the time complexity of the resulting method would seem to be an inordinately high order polynomial. Our algorithm we claim is "practical". The addition of one edge to a tree does seem to require a considerably more elaborate algorithm.

Keywords: algorithms, unicyclic graphs, trees, vertex separation, search number, pathwidth, linear layouts, layout extensibility.

1 Introduction

It is also well known that *vertex separation* is identical to *pathwidth* [Kin92] and that vertex separation is identical to *node search number* -1 [KP86] and to *interval thickness* [KP85]. The *pathwidth* of a graph is an important parameter that can significantly influence the time complexity of algorithmic graph problems. Computing this parameter is, in general, an NP-complete problem and remains NP-complete even for planar graphs of maximum degree three [MS88].

[BK96] gives a polynomial time algorithm for the computation of the pathwidth of graphs of bounded treewidth. However, this algorithm can not be considered practical because the bound given on its time complexity is $\Omega(n^{11})$ for treewidth two, which includes unicyclic and outerplanar graphs. Fast approximation algorithms have been described for outerplanar graphs [BF02, GLY98].

In [EST94] a linear time algorithm was described for the computation of the vertex separation of trees. That paper suggested an $O(n \log n)$ time algorithm for the computation of an actual layout with minimum vertex separation. Linear time algorithms for optimal layout computation have been described in [Sko00] and [PHH$^+$98].

In this paper we show how to use the methods described in [EST94] and [Sko00] to compute the vertex separation and an optimal layout for unicyclic graphs, i.e, trees with one extra edge. The time complexity of the algorithm is $O(n \log n)$, with a reasonable constant of proportionality, and so this result can be considered to be practical. However, it does seem that the addition of one edge to a tree necessitates a considerably more elaborate algorithm.

In Section 2 we review techniques for computing the vertex separation and optimal layouts for trees, emphasising the concepts and methods we use heavily in the rest of the paper. In Section 3 we set up some useful lemmas regarding properties of layouts for trees and unicyclic graphs that will be of general utility. Also we show an interesting "reversibility" property for graph layouts and "extensibilities" which we do not know to have been previously demonstrated. In Section 4 we work out solutions to a variation of the problem on restricted unicyclic graphs. In Section 5 we break the general problem down into cases some of which can be solved immediately whereas others are reduced to the problem solved in Section 4. In the last section we summarise the results by outlining an algorithm and establish its $O(n \log n)$ time complexity.

2 Computing vertex separation and optimal layouts for trees

2.1 Background

In this section we review the concepts and methods used to both compute the vertex separation of a tree in linear time and to compute an optimal layout in linear time. The former problem was solved in [EST94], but there the authors gave only an $O(n \log n)$ algorithm to compute an optimal layout. Linear time methods for the computation of optimal layouts have been described in [Sko00] and [PHH+98]. The later sections in this paper are crucially dependent on these methods, so we restate them here in a form which emphasises the concepts we need in this paper.

We consider graphs without loops or multiple edges, since these do not change vertex separation. A *linear layout* of a graph $G = (V, E)$ is a bijection $L : V \to \{1, 2, \cdots, |V|\}$. Let $V_L(i) = \{x | x \in V$ and there exists $y \in V$ such that $\{x, y\} \in E$ and $L(x) \leq i$ and $L(y) > i\}$. Where convenient, we will refer to $|V_L(i)|$ as the *weight* of vertex u at location i, and use the notation $w_L(u) = |V_L(i)|$, where $L(u) = i$.

The *vertex separation* of G with respect to L, denoted by $vs_L(G)$, is defined by

$$vs_L(G) = max\{|V_L(i)| \mid 1 \leq i \leq |V|\}$$

and the vertex separation of G is defined by

$$vs(G) = min\{vs_L(G) | L \text{ is a layout of } G\}.$$

Any layout which achieves minimum vertex separation we will refer to as *optimal.*

We use *vs-k-tree* to denote a tree whose vertex separation is k and we use *k-layout* to denote a vertex separation k layout. Envisioning a layout as a horizontal sequence of vertices, we will speak informally of the leftmost and rightmost vertices in the layout and of one vertex preceding or following or being to the left or right of another, with the obvious meanings.

In [EST94] the following theorem is established (Corollary 3.1, Section 3.1), where the trees *induced* by a vertex v in a tree T are the components of T remaining after the removal of v from T.

Theorem 2.1 ([EST94])
$vs(T) > k$ iff there exists a vertex which induces ≥ 3 subtrees T' such that $vs(T') \geq k$.

We will also find useful a small extension of this theorem, because we are dealing with graphs more general than trees. If G' is a subgraph of G we will say that a layout L for G *induces* a layout L' for G' when L' assigns an order to the vertices in G' that is identical to their relative order in L.

Lemma 2.1
If a graph G contains three connected subgraphs G_1, G_2 and G_3, whose vertex sets are pairwise disjoint, and such that for every pair G_i, G_j there exists a path in G between G_i and G_j that includes no vertex in the third graph, and such that $vs(G_1) = vs(G_2) = vs(G_3) = k$, then $vs(G) \geq k + 1$.

Proof
Let L be an optimal layout for G. Suppose L induces layouts L_1, L_2 and L_3 for G_1, G_2 and G_3 respectively. Then $vs_{L_i}(G_i) \geq k$ for $1 \leq i \leq 3$ and hence there exists a vertex h_i in each G_i such that $w_{L_i}(h_i) \geq k$. Without loss of generality, let h_1, h_2 and h_3 appear in that

order in L. Note that there is a path in G from h_1 to h_3 containing no vertex in G_2. Hence $w_L(h_2) \geq k+1$. □

The algorithm that computes the vertex separation of a tree T ([EST94], Section 3.4.2) begins by arbitrarily choosing a vertex u and transforming T into a tree rooted at u. The notation $T[x]$ is used to denote the subtree rooted at any vertex x within the rooted tree T. Let $u, v_1, v_2, \ldots, v_i$ be vertices in T. Then $T[u, v_1, v_2, \ldots v_i]$ denotes the tree with root u from which the subtrees with roots v_1 through v_i have been removed. Note that if some v_i in this definition is not in $T[u]$, the presence of v_i in the list is without significance.

The algorithm uses the concept of *criticality*, which is related to Theorem 2.1. A vertex x is *k-critical* in a rooted tree T iff $vs(T[x]) = k$ and there are two children u and v of x such that $vs(T[u]) = vs(T[v]) = k$. We will also say that a tree is *k-critical* if it contains a k-critical vertex.

The algorithm creates *labels* for each vertex x. The label gives information about the structure of $T[x]$ including a list of the vertex separations of critical subtrees contained within it. The full definition is given below. Labels for any vertex x are computed by combining the labels of the children of x. The correctness of the algorithm is established by applying Theorem 2.1. In an analogous manner, layouts for $T[x]$ can be constructed by combining layouts for the trees rooted at the children of x, as we will show shortly.

We need the concept of a vertex being *singular* in a layout. Informally, a vertex is singular in a particular layout if there are no edges passing over it, and its weight is one.

Definition 2.1
A vertex u is singular *in a layout L if $V_L(i) = \{u\}$, where $L(u) = i$, i.e., $w_L(u) = 1$.*

We also need the concepts of *right extensible* and *left extensible* layouts, introduced in [Sko00] and slightly reformulated here.

Definition 2.2
Suppose L is a layout for a graph G. Then L is left extensible *with respect to a vertex u and the number k if for all vertices v such that $L(v) < L(u)$, $w_L(v) < k$, i.e., for all vertices v to the left of u, $w_L(v) < k$.*
We define rightmost*(u) to be the rightmost vertex in L among u and its neighbours. Then L is* right extensible *with respect to u and the number k if for all vertices v such that $L(v) \geq L(rightmost(u))$, $w_L(v) < k$, i.e., $w_L(rightmost(u)) < k$ and for all vertices v to the right of $rightmost(u)$, $w_L(v) < k$.*

The idea is illustrated in Figure 1. On occasion the number k may be understood from context, e.g. it is the vertex separation of the graph under consideration. We may then omit explicit mention of the number.

The following lemma follows immediately from the definition of left extensibility.

Lemma 2.2
If L is a layout for any graph G such that L is left extensible with respect to some vertex u and $vs_L(G)$, and L' is identical to L except that u is placed in the leftmost position, then $vs_{L'}(G) = vs_L(G)$.

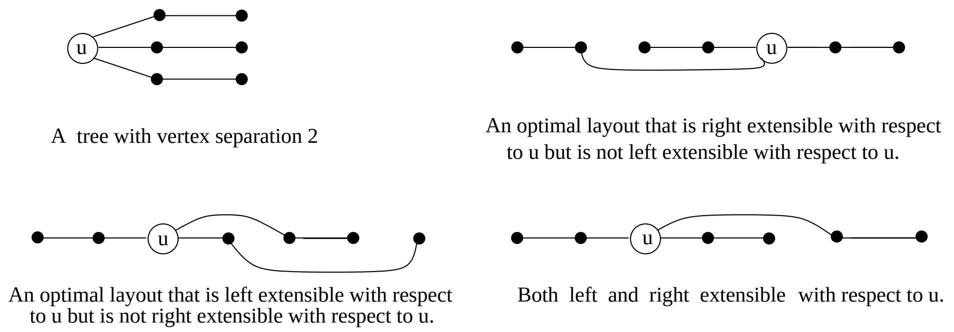


Figure 1: Extensible Layouts

Proof

The weight of all vertices to the left of u in L is less than $vs_L(G)$. The placement of u in the leftmost position adds no more than one to the weight of these and only these vertices. The weight of all other vertices is unchanged. $\square$

We repeat the definition of *label* from [EST94] and add the concept of *layout list.*

Definition 2.3 • *For any tree $T[u]$ the label λ of u is a list of integers $(a_1, \ldots, a_p)$, where $a_1 > a_2 > \ldots > a_p \geq 0$, and such that there exists a set of vertices $\{v_1, \ldots, v_p\}$ such that*

- $vs(T[u]) = a_1$.
- *For* $1 \leq i < p$, $vs(T[u, v_1, \ldots, v_i]) = a_{i+1}$.
- *For* $1 \leq i < p$, v_i *is an* a_i*-critical vertex in* $T[u, v_1, \ldots, v_{i-1}]$.
- v_p *is* u.

If a_p is marked, for example we use a prime ($\prime$), then there is no a_p-critical vertex in $T[u, v_1, \cdots, v_{p-1}]$. If a_p is not so marked then v_p is an a_p-critical vertex. In both cases $T[u, v_p] = T[u, u]$ is the empty tree.

• *A* layout list $\mathcal{L}$ *for $T[u]$ is a list of layouts $(L_1, L_2, \ldots, L_p)$ where,*

- *for* $1 \leq i \leq p$, L_i *is an optimal layout for* $T[v_i, v_1, \ldots, v_{i-1}]$.
- *for* $1 \leq i \leq p$, v_i *is singular in* L_i.
- *If* $T[u, v_1, \ldots, v_{p-1}]$ *is not critical, then* L_p *is either left or right extensible with respect to* u.

2.2 Primitive Operations on Layouts

In a manner analogous to the computation of the labels, we are going to compute a layout list for the tree rooted at any particular vertex by combining layout lists already computed for its children. The method we describe here is similar to that of [Sko00]. Our description extends the concepts used and results obtained in [EST94] and emphasises those concepts and constructions essential for the later sections of this paper. Although layouts have been defined abstractly as functions, we represent them in the obvious way, both in diagrams and algorithmically, as sequences of vertices. In all the diagrams, horizontal lines will represent layouts and curves will represent edges.

2.2.1 Methods for Combining Layouts

We describe four methods whereby optimal layouts can be combined so that the resulting layout is not only optimal but is also right or left extensible, if such layouts exist, with respect to some singular vertex. The methods are illustrated in Figure 2. In each case it follows immediately from the stated conditions and Theorem 2.1 that the defined construction yields the claimed results.

Method 1

Components: Optimal layouts for any number of subtrees T_i rooted at the children of u where $vs(T_i) \leq k$.

Arrangement: u is placed at the left end of the layout followed by the subtree layouts, in any order.

Result: A layout L for the tree $T[u]$, such that $vs_L(T[u]) \leq k+1$, L is both left and right extensible with respect to u and $k+1$ and u is singular. $T[u]$ does not contain a $(k+1)$-critical vertex.

Method 2

Components: Optimal layouts for two subtrees T_1 and T_2 rooted at children of u, where $vs(T_1) = vs(T_2) = k$, neither T_1 nor T_2 contains a critical vertex, the layout for T_2 is left extensible and the layout for T_1 is right extensible, with respect to their roots and k. Also, zero or more optimal layouts for other subtrees T_i where $vs(T_i) < k$.

Arrangement: The layout for T_1 is followed by u which is followed by the layouts for the other T_i in any order, which are followed by the layout for T_2.

Result: A layout L such that $vs_L(T[u]) = k$ and u is singular. $T[u]$ contains a k-critical vertex, namely u.

Method 3

Components: An optimal layout for a subtree T_1 rooted at a child of u where $vs(T_1) = k$, T_1 does not contain a k-critical vertex and both left and right extensible layouts for T_1 exist. Also, zero or more optimal layouts for other subtrees T_i where $vs(T_i) < k$.

Arrangement: To create a right extensible result, the right extensible layout for T_1 is followed by u which it followed by layouts for the other T_i, in any order.
To create a left extensible result, u is placed at the left end of the layout, followed by layouts for the other T_i in any order, followed by the left extensible layout for T_1.

Result: A layout L such that $vs_L(T) = k$ and L is either left or right extensible with respect u, which is singular. $T[u]$ does not contain a k-critical vertex.

Method 4

Components: An optimal layout L for a tree T where $vs_L(T) < k$, a layout L_i for a tree T_i where $vs_{L_i}(T_i) = k$, and where v is singular in L_i, there exists no more than one edge between T_i and T and that edge, if it exists, is incident with v.

Arrangement: The layout L for T is inserted immediately after v in L_i.

Result: A layout with vertex separation k, where v is singular.

2.2.2 Layout Nesting

Suppose we have a label $(a_1, ..., a_p)$ and a corresponding layout list $\mathcal{L} = (L_1, \ldots, L_p)$ for the tree $T[u]$, with the properties just defined for these entities. We show that we can compute a layout L for $T[u]$ such that $vs_L(T[u]) = a_1$ by applying Method 4 repeatedly.

Let L^i, $1 \leq i \leq p$, be an optimal layout for $T[u, v_1, \ldots, v_{i-1}]$ so that $L^p = L_p$. Note that L^1 is then an optimal layout for $T[u]$. For all i, $1 \leq i < p$, we can construct L^i from L^{i+1} and L_i by applying Method 4 with T in the diagram in Figure 2 representing L^{i+1}, the remainder of the diagram representing L_i and v representing v_i, the root of $T[v_i]$.

From the definitions of the label and layout lists we have that

$$vs_{L_i}(T[v_i, v_1, v_2, \ldots, v_{i-1}]) = a_i,$$

v_i is singular in L_i, there is one edge between $T[u, v_1, \ldots, v_i]$ and $T[v_i, v_1, v_2, \ldots, v_{i-1}]$ and that edge is incident with v_i. Consequently, the inductive hypothesis:

$$vs_{L^i}(T[u, v_1, \ldots, v_{i-1}]) = a_i$$

is preserved and L^1 is an optimal layout for $T[u]$. We will refer to this method as the *layout nesting procedure.*

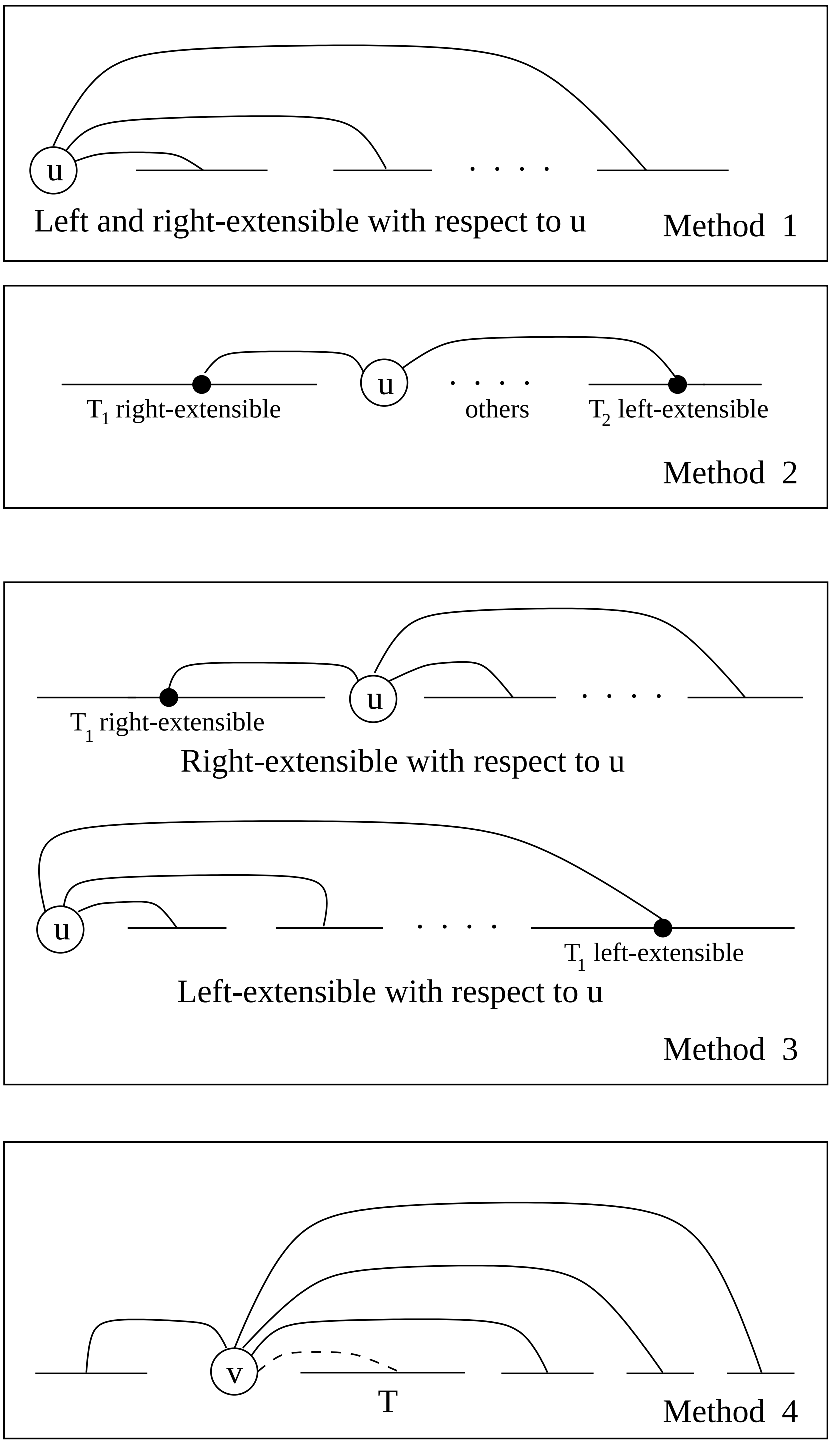


Figure 2: The Basic Constructions

2.3 Computing layout lists

We review the label generator in [EST94] Section 3.4.2. A depth first search procedure, *compute_label*, traverses the rooted tree. At each vertex, after labels are computed for the subtrees rooted at the children of that vertex, these labels are combined by the procedure *combine-labels* to create a label λ for the vertex. This latter procedure scans all subtree labels from smaller to larger values and proceeds according to the number of labels containing a particular value k.

By T^k we denote the tree from which all subtrees corresponding to label elements strictly greated than k in any subtree label have been removed from T. We do not repeat the justification for the claim that after the scanning of all label elements $\leq k$, λ is the correct label for T^k. There are six cases covering the construction of λ. In Case 5, "&" indicates list concatenation.

Case 1: There exist at least three labels containing the value k, from which it is deduced that $vs(T^k) = k+1$ and T^k does not contain a $(k+1)$-critical vertex.
Label action: $\lambda := (k+1')$;

Case 2: There exist exactly two labels containing the value k and at least one of them is associated with a critical vertex, from which it is deduced that $vs(T^k) = k+1$ and T^k does not contain a $(k+1)$-critical vertex.
Label action: $\lambda := (k+1')$;

Case 3: There exist exactly two labels containing the value k and neither of them is associated with a k-critical vertex, from which it is deduced that $vs(T^k) = k$ and T^k contains a k-critical vertex, namely the root vertex.
Label action: $\lambda := (k)$;

Case 4: There exists exactly one label containing the value k and it is associated with a critical vertex and $k \in \lambda$, from which it is deduced that $vs(T^k) = k+1$ and T^k does not contain a $(k+1)$-critical vertex.
Label action: $\lambda := (k+1')$;

Case 5: There exists exactly one label containing the value k and it is associated with a critical vertex and $k \notin \lambda$, from which it is deduced that $vs(T^k) = k$ and T^k contains a k-critical vertex.
Label action: $\lambda := (k)\&\lambda$;

Case 6: There exists exactly one label containing the value k, it is not associated with a critical vertex, and $k \notin \lambda$, from which it is deduced that $vs(T^k) = k$ and T^k does not contain a k-critical vertex.
Label action: $\lambda := (k')$;

We now show how to use these concepts and results from [EST94] to compute layout lists in linear time. We propose that at each level of recursion the computation of a label is followed immediately by the computation of a layout list via the combination of layout lists for the subtrees. The basis for the recursive computation of layout lists is the assignment of the layout (u) to the tree comprising the single vertex u. In general, suppose we are combining labels and layout lists for the subtrees $T_1, T_2, \ldots, T_d$, rooted at the children of u, in order to form a label λ and a layout list $\mathcal{L}$ for $T[u]$. We first run the combine_labels procedure to compute λ. Suppose the output is the label $\lambda = (a_1, \ldots, a_p)$. This information is used to compute $\mathcal{L} = (L_1, \ldots, L_p)$, the corresponding layout list.

We first note that all the elements in λ except a_p must have been generated by Case 5, because all the other cases create a label containing exactly one element. Consequently, among all the layout lists for all the subtrees there is exactly one layout L_i corresponding to a_i, for all i, $1 \leq i < p$. Hence, to form $\mathcal{L}' = (L_1, \ldots, L_{p-1})$, we need simply to insert these L_i, ordered by vertex separation.

Finally we note that a_p was generated by one of Cases 1 - 4 or by Case 6. We recall that a_p is the vertex separation of $T[u, v_1, \ldots, v_{p-1}]$, where $v_1, \ldots, v_{p-1}$ are the critical vertices corresponding to the elements $a_1, \ldots, a_{p-1}$ in λ, and that L_p should be a layout for that tree. This tree is the tree $T[u]$ from which all critical subtrees with vertex separation greater than a_p have been removed. Thus to compute L_p we take two steps:

Step 1: For each subtree T_i compute a layout for T_i minus all critical subtrees with vertex separation greater than a_p. This can be done by applying the layout nesting procedure, described in the previous section, for all layouts in the layout list for T_i conforming to the vertex separation constraint.

Step 2: Combine these layouts to form a layout for $T[u, v_1, \ldots, v_{p-1}]$ by using the appropriate method in Figure 2 as defined in the following cases. We may assume, for example, that some flag indicating which case was used to create a_p is associated with that value.

Case 1: Use Method 1 to produce a layout that is both left and right extensible with respect to u and with u singular;

Case 2: Use Method 1 to produce a layout that is both left and right extensible with respect to u and with u singular;

Case 3: Use Method 2 to produce a layout with u singular;

Case 4: Use Method 1 to produce a layout that is both left and right extensible with respect to u and with u singular;

Case 6: Use Method 3 to produce both right and left extensible layouts with respect to u and with u singular.

Finally we note that a layout for the entire tree can be formed by applying the layout nesting procedure to the layout list which has been computed for the root of the tree.

2.4 Correctness and Time Complexity

To demonstrate correctness, note that we have already pointed out that the elements of the layout list L_1 through L_{p-1} are simply certain of the existing layouts in the layout lists of the subtrees and hence just need to be linked in to the new list. It remains only to argue that the prescribed method for computing L_p is correct.

Let T^p denote $T[u, v_1, \ldots, v_{p-1}]$ so that $vs(T^p) = a_p$. Step 1 of the computation of L_p relies on the correctness of the layout nesting procedure, already demonstrated. Hence, at completion of Step 1, we have optimal layouts for all subtrees of T^p rooted at children of u.

In Step 2 we combine the layouts of these subtrees to form L_p, using one of Methods 1 - 3, depending on the case used to compute a_p. The correctness of Cases 1 - 4 is immediate. Case 6 uses Method 3 whose validity requires the assumption that if a rooted tree T contains no critical vertex, then there exist optimal layouts for T that are left extensible and optimal layouts that are right extensible with respect to the root. This fact is easily established by induction on tree height. See Lemmas 2.3 and 2.4 below.

To justify linear time complexity we must specify the way in which the layout lists are to be represented. Standard linked list techniques are adequate. If each layout is represented by a linked list of vertices, it is sufficient to maintain pointers to the beginning, the end and to the singular vertex, which always exists, in each layout for all operations defined in Figure 2, namely concatenation and insertion, to be done in constant time. A layout list can be represented by a linked list of layouts.

The collection of layouts L_1 through L_{p-1} and Steps 1 and 2 of the construction of L_p require that the layout lists for the subtrees be scanned, from lower to higher separations, just as is done in the computation of the label. Further, the manipulation of the layouts via Methods 1-4 require the concatenation of lists and the insertion of one list into another, but these are the only operations needed. We consider the time complexity of layout manipulations and list scanning separately.

Consider the number of layout concatenations and insertions. Suppose a vertex u is at the end of a layout that is concatenated with another. This can not occur more than twice to u, once on each end. List insertions are used only by Method 4, and that method is only used by the nesting procedure. Consequently no vertex is subject to an insertion immediately to its right, i.e., is v_i in Method 4, more than once. Consequently the time spent in layout operations over the entire process is linear in the number of vertices.

In order to achieve linear time during list scanning, it is necessary to introduce a further refinement, analogous to the representation by "intervals" of the labels, as described in Section 3.4.4 of [EST94]. The interval representation permits the label algorithm to terminate as soon as all lists, excepting perhaps one, are empty. This ensures that the work done is bounded by the number of lists times the length of the second longest list, where the length of a list means the maximum separation number in that list. This observation is the key element in the non-trivial proof of linear time complexity in [EST94] for the corresponding label computation.

We can achieve the same result for the scanning of the layout lists if we adopt a compact representation of the layout lists. Suppose a layout list contains adjacent layouts L_i and

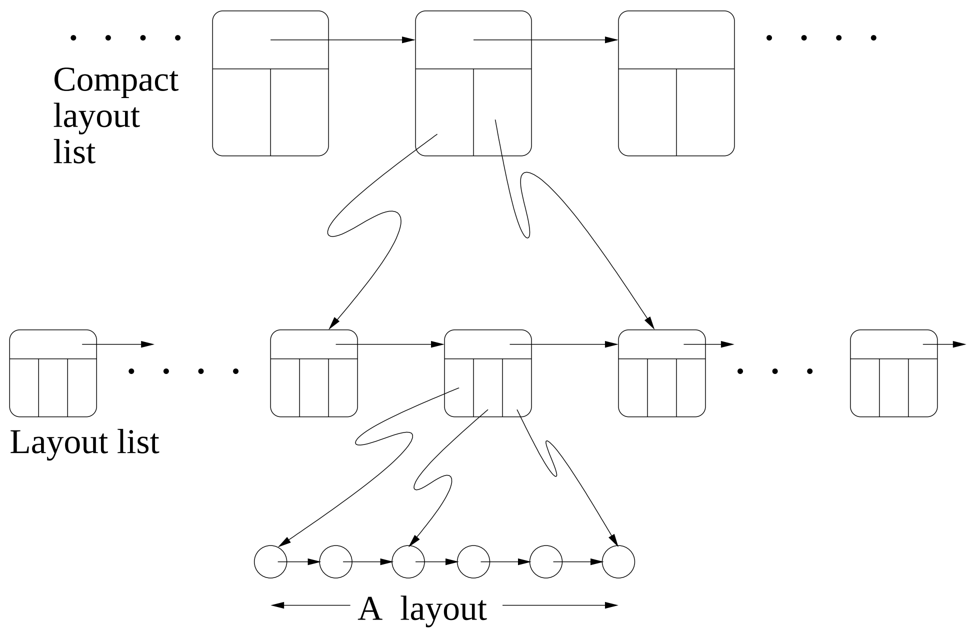


Figure 3: Layout List Representation

L_{i+1} such that $a_i = a_i + 1$. We will say such layouts are *contiguous*. Let $L_i, \ldots, L_j$ be a subsequence of $\mathcal{L}$ that is maximal with respect to contiguity. The compact layout list contains a list of pairs of pointers to corresponding L_i and L_j in the layout list for every such contiguous sequence, let us call them *layout intervals*. The principle is illustrated in Figure 3.

Now consider the time required to scan the compact layout lists during Step 1 when the lists for all subtrees are scanned up to the point where a layout of separation greater than a_p is found. We observe that a_p can not be the separation of a layout internal to one of the contiguous layout sequences. This is because the label computation can be viewed as a kind of binary addition process. A "carry" ripples through any contiguous sequence just defined, as can be seen in the following. Since a_p is the last item in a label, it was not generated by Case 5. If a_p is internal to a contiguous sequence in one of the subtrees, then it was not computed by Case 3 or by Case 6 because in these cases the elements in the scanned labels refer to a non-critical vertex, i.e., they are end items in a label. So a_p was computed by one of Cases 1, 2 or 4. In each of these cases a "carry" is generated, i.e., the generated label is $(k+1)$.

It is also clear that no carry can jump the gap between adjacent layout intervals. Consequently, even if a_p is greater than any value in any of the labels excepting for one, only one of the remaining layout intervals in the longest of all labels needs to be examined, and that in constant time.

From the observations in the last two paragraphs, we deduce that the time complexity of the scan for a_p is bounded by the length of the second longest list. Consequently the recurrence relation set up in [EST94], Section 3.4.4 for the time complexity of the label computation is also valid for the scan time and the linear time solution is also valid.

2.5 Tree types and their layout properties

In the following sections, we make substantial use of the following four ways of classifying trees and the fact that certain classes admit layouts with certain properties.

Definition 2.4
Let T be a tree rooted at a vertex u and let $vs(T) = k$.

- *If there is no child v of u such that $vs(T[v]) = k$, then we say that T is of type NC (for "Not Critical").*
- *If there is a child v of u such that $vs(T[v]) = k$, and T is not k-critical, then we say that T is of type NCb (for "Not Critical, buried").*
- *If T is k-critical and u is the critical vertex, then we say that T is of type C (for "Critical").*
- *If T is k-critical and u is not the critical vertex, then we say that T is of type Cb (for "Critical, buried").*

Lemma 2.3
If a tree is of type NC, there exists an optimal layout in which the root is singular and which is both left and right extensible with respect to the root.

Proof
Because there is no child v of u such that $vs(T[v]) = k$, Method 1, Figure 2, can be used to produce a layout with the claimed properties. $\square$

Lemma 2.4
If a tree is of type NCb, there exists an optimal layout in which the root is singular and which is left extensible with respect to the root and there exists an optimal layout in which the root is singular and which is right extensible with respect to the root.

Proof
The proof proceeds by induction on the distance from the root of the tree to the root of the minimal proper subtree whose vertex separation is the same as that of the entire tree. If T is of type NCb, then no more than one of the subtrees rooted at a child of the root is of type NCb. For the basis of the induction, assume that no subtree rooted at a child of the root is of type NCb. Then, by Lemma 2.3, layouts which are left and right extensible with respect to their roots, exist for these subtrees. Then Method 3, Figure 2, suffices to construct the desired layout. If one of the subtrees rooted at a child of the root is of type NCb, then

we make the inductive assumption that layouts with the required properties exist for that subtree. Hence Method 3 can yield a layout for the entire tree with the desired properties. □

Lemma 2.5
If a tree is of type C, there exists an optimal layout in which the root is singular.

Proof
If T is of type C, then none of the children are critical, and exactly two of them are of separation k, and hence are of type NC, or NCb. Hence left and right extensible layouts exist, by the two previous lemmas. Hence Method 2, Figure 2, yields the desired layout. □

Lemma 2.6
Let T be any k-critical tree, i.e., of type C or Cb. Let u be the root of the subtree of T such that there exist two children of u, say x and y, such that $vs(T[x]) = vs(T[y]) = k$. Let L be any optimal layout for T and let L_x and L_y be the layouts induced by L for $T[x]$ and $T[y]$ respectively. Let h_x in $T[x]$ and h_y in $T[y]$ be any pair of vertices such that $w_{L_x}(h_x) = w_{L_y}(h_y) = k$. Then any vertex not in $T[x]$ and not in $T[y]$ lies between h_x and h_y in L.

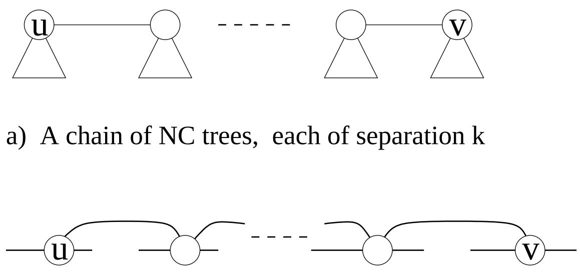
Proof
Without loss of generality suppose h_x precedes h_y in L. Let v be any vertex not in $T[x]$ and not in $T[y]$. If v precedes h_x in L, then there is a path from v to h_y containing no vertex in $T[x]$ that passes over h_x in L. Then $vs_L(T)$ would be at least $k + 1$.
If v follows h_y in L, then there is a path from h_x to v containing no vertex in $T[y]$ that passes over h_y in L. Then $vs_L(T)$ would be at least $k + 1$. □

Lemma 2.7
Let T be a chain of trees, each of type NC and of vertex separation k. Then there exists a vertex separation k layout for T which is left extensible with respect to the first tree root in the chain and right extensible with respect to the last tree root in the chain.

Proof
By Lemma 2.3, there is a vertex separation k layout for each of the trees in the chain that is both left and right extensible with respect to the tree root. Hence the layout defined in Figure 4 has the desired properties. □

3 Some useful lemmas

We set up some lemmas that we will use more than once in the next two sections.



a) A chain of NC trees, each of separation k

b) A k-layout

Figure 4: T is a chain of trees of vertex separation k and all of type NC

3.1 Layout reversal properties

Because we make heavy use of left and right extensibility properties, we point out an interesting "reversibility" property, possessed by any layout of any graph. We define a *reversal* process which rearranges any layout to produce another layout with the same vertex separation.

Firstly, we define a process which "reverses" a vertex search strategy. A vertex search strategy is a sequence of placements and removals of guards on and from the vertices of a graph. The edges of the graph are all initially "dirty". An edge is cleaned exactly when both ends of the edge are simultaneously occupied by guards. An "effective" strategy starts with all the edges being dirty and with no guards on the graph, and ends with all the edges being clean and no guards on the graph. A "progressive" strategy places a guard on a vertex exactly once.

A clean edge can be "recontaminated" if the removal of a guard opens a path containing no guard from a dirty edge to the clean edge. Consequently, a progressive strategy includes no recontamination. By [LaP93] and [KP86], if there exists a recontaminating strategy using a certain number of guards, then there exists a progressive strategy using that same number of guards. Hence, in any effective, progressive strategy there are exactly n placements and n removals of guards, where n is the number of vertices in the graph. In the following, we abbreviate "effective, progressive, vertex search strategy" to just "strategy".

Let the "occupancy" of the graph at any point during the strategy be the number of guards on the graph at that point. The search number of a strategy S on a graph G, denoted $sn_S(G)$, is the maximum occupancy during the strategy. The search number of a graph is the minimum of $sn_S(G)$ over all strategies S.

Definition 3.1
The reversal $\overline{S}$ *of a strategy* S *is the strategy obtained from* S *by:*

- *Reversing the order of the items in the sequence* S*.*
- *Replacing every removal of a guard from vertex* i *by a placement of a guard on vertex* i*.*
- *Replacing every placement of a guard on a vertex* i *by a removal of a guard from vertex* i*.*

The process is illustrated by the two central diagrams in Figure 5. Note that the construction of the reversal is a linear time process.

Lemma 3.1
If S *is an effective, progressive search strategy for a graph* G *such that* $sn_S(G) = k$ *then* $\overline{S}$*, the reversal of that strategy, is also an effective, progressive search strategy such that* $sn_{\overline{S}}(G) = k$*.*

Proof
From the definition of the reversal process it it clear that any pair of vertices are simultaneously occupied by guards in S iff they are simultaneously occupied by guards in $\overline{S}$. From this we deduce that $sn_S(G) = sn_{\overline{S}}(G)$ and that every edge is cleaned at some point in $\overline{S}$. Also, if S is progressive then so is $\overline{S}$ and if S begins and ends with no guards on the graph then $\overline{S}$ ends and begins with no guards on the graph.

Hence, if $\overline{S}$ is not effective, it must be that recontamination occurs, i.e., there is a guard removal from some vertex u that is incident with a dirty edge $\{u, v\}$. Consider the first such removal, if it exists, in $\overline{S}$. Since $\{u, v\}$ is dirty and $\overline{S}$ is progressive it must be that the placement of a guard on v follows the removal of a guard from u in $\overline{S}$. Consequently it must be that the removal of a guard from v precedes the placement of a guard on u in S, contradicting the fact that S is progressive and effective. □

The layout reversal process is based on the constructions described in [KP86, Section 4] which given a layout compute a search strategy, and given a search strategy compute a layout.

- To compute a search strategy S from a layout L we place guards on vertices in the order that they appear in the layout and remove a guard from a vertex as soon as all edges incident with that vertex are clean. If two or more vertices become free simultaneously, the order of guard removals is arbitrary. This construction ensures that $sn_S(G) \leq vs_L(G) + 1$. It also has the property, which we use below, that if the weight of all vertices preceding some vertex u in L is not greater than $k - 1$ then the occupancy in S at all times up to the placement of a guard on u is not greater than k.

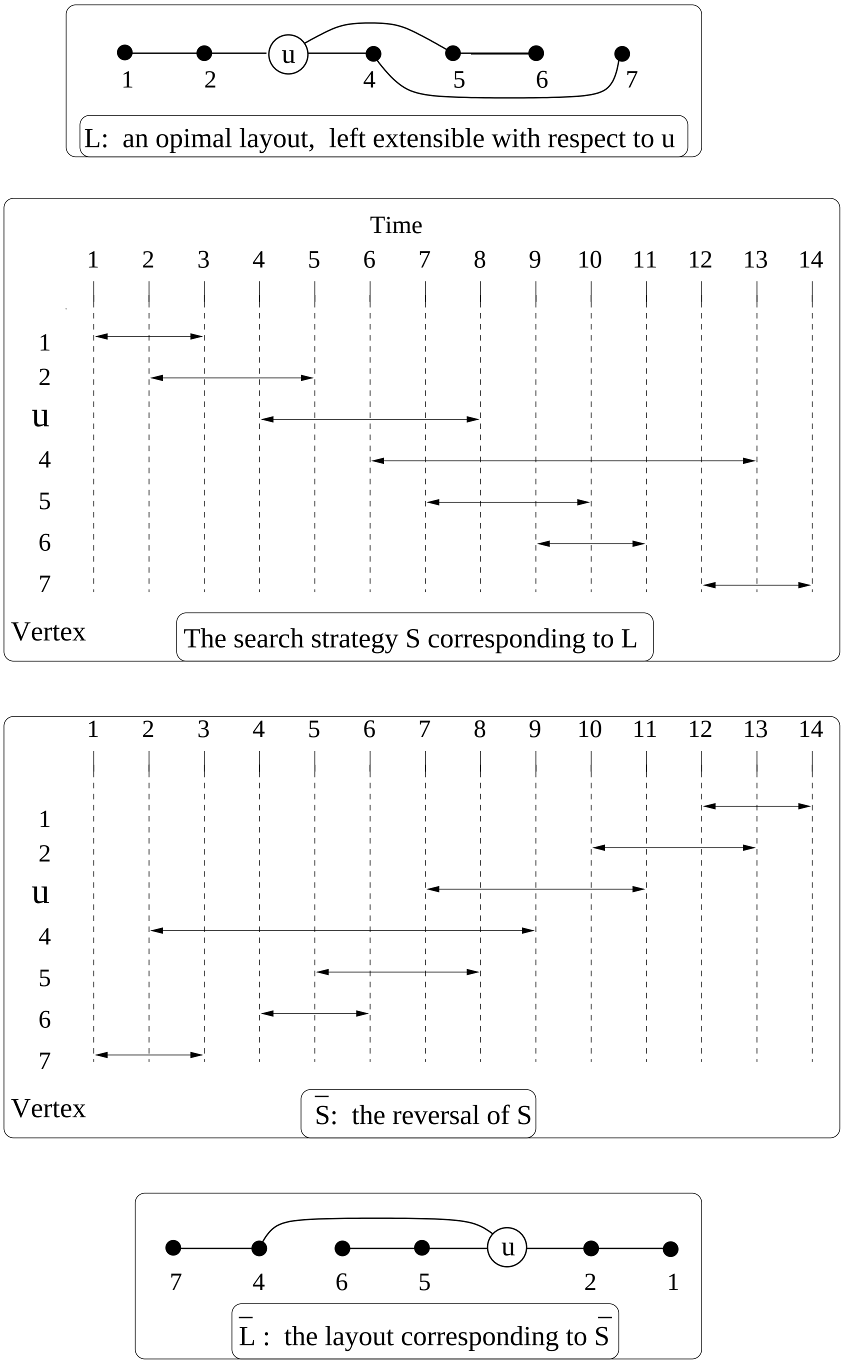


Figure 5: Reversing a layout and a search strategy

- To compute a layout L from a search strategy S we assign an order to the vertices in the layout that is identical to the order in which they receive a guard in S. This construction ensures that $vs_L(G) \leq sn_S(G) - 1$. It also has the property that if the occupancy up to and including the placement of a guard on u is not greater than k, then the weight of all vertices preceding u in L is less than k.

Definition 3.2
The reversal $\overline{L}$ *of a layout L is the layout obtained from L by:*

- *Computing a strategy S equivalent to the layout L as just described.*
- *Computing the reversal $\overline{S}$ of the strategy S as just described.*
- *Computing the layout $\overline{L}$ equivalent to the strategy $\overline{S}$ as just described.*

The entire process is illustrated in Figure 5. Note that this is a linear time process.

Lemma 3.2
If L is a layout for a graph G and $\overline{L}$ is the reversal of L, then $vs_{\overline{L}}(G) \leq vs_L(G)$.

Proof
From the argument given in [KP86, Section 4], the construction that creates a strategy S from a layout L guarantees that $sn_S(G) \leq vs_L(G) + 1$, and the construction that creates a layout $\overline{L}$ from a strategy $\overline{S}$ guarantees that $sn_{\overline{S}}(G) > vs_{\overline{L}}(G)$. We have just shown that $sn_S(G) = sn_{\overline{S}}(G)$. Hence $vs_{\overline{L}}(G) \leq vs_L(G)$. □

Corollary 3.1 *If L is an optimal layout for a graph G and $\overline{L}$ is the reversal of L, then $\overline{L}$ is also optimal.*

It is also true that by using the reversing construction we can reverse extensibilities.

Lemma 3.3
If L is an optimal layout for a graph G and L is right extensible with respect to some vertex u then there exists an optimal layout for G that is left extensible with respect to u.

Proof
Suppose $vs_L(G) = k$. Construct a strategy S from L using the construction in [KP86]. By construction, the number of guards on the graph from the time a guard is placed on rightmost(u) (as defined by L) is not greater than k. Construct S' from S by postponing the removal of the guard on u until the last move in S'. Then $sn_{S'}(G) \leq k + 1$. Construct the reversal of $\overline{S'}$ and the layout $\overline{L'}$ corresponding to this strategy as defined by the construction in [KP86]. By construction, u is the leftmost vertex in $\overline{L'}$ and $vs_{\overline{L'}(G)} \leq k$. □

Lemma 3.4
If L is an optimal layout for a graph G and L is left extensible with respect to some vertex u then there exists an optimal layout for G that is right extensible with respect to u.

Proof
Since L is left extensible with respect to u, we may assume, by Lemma 2.2, that u is the leftmost vertex in L. Suppose a strategy S is derived from the layout L. We deduce from the construction rules that u is the first vertex to receive a guard and that the $sn_S(G) \leq k+1$. Consider the reversal $\overline{S}$ of S. Construct $\overline{S'}$ from $\overline{S}$ by advancing the last move, the removal of a guard from u, to the point following the last guard placement on a neighbour of u. The occupancy of the graph after this removal is not greater than k. Finally construct the layout $\overline{L'}$ from $\overline{S'}$. Then $vs_{\overline{L'}(G)} \leq k$ and $\overline{L'}$ is right extensible with respect to u. □

Lemma 3.5
If L is an optimal layout for a graph G and L is left extensible with respect to some vertex u and right extensible with respect to some vertex v then there exists an optimal layout for G that is right extensible with respect to u and left extensible with respect to v.

Proof
The constructions described separately in Lemmas 3.4 and 3.3 can be carried out simultaneously. □

3.2 Some layout constraints on unicyclic graphs

We picture a unicyclic graph as a cycle in which each cycle vertex is the root of a tree, which we will refer to as a *constituent* tree.

Lemma 3.6
Let U be the unicyclic graph containing exactly five constituent vs-$(k-1)$-trees, each of type NCb, then $vs(U) \geq k+1$.

Proof
The situation is illustrated in Figure 6. Let L be any optimal layout for U. Let L_i, $1 \leq i \leq 5$, denote the layouts for the five constituent trees, T_i, induced by L. Let h_i, $1 \leq i \leq 5$, be a vertex in the corresponding tree such that $w_{L_i}(h_i) \geq k-1$. Note that these vertices are distinct from the tree roots, because each constituent tree contains a proper subtree of separation $k-1$. Without loss of generality, let h_3 be between two pairs of vertices, taken from the remaining four, in L. Considering the symmetry of the situation and noting that the order of the vertices within each pair is immaterial, we see that there are only three essentially distinct orderings. In each of these possible orderings we point to two vertex disjoint paths, neither including any vertex in T_3, that pass over h_3 in L. Hence, since the weight of h_3 in L_3 is at least $k-1$, $vs_L(U) \geq k+1$.

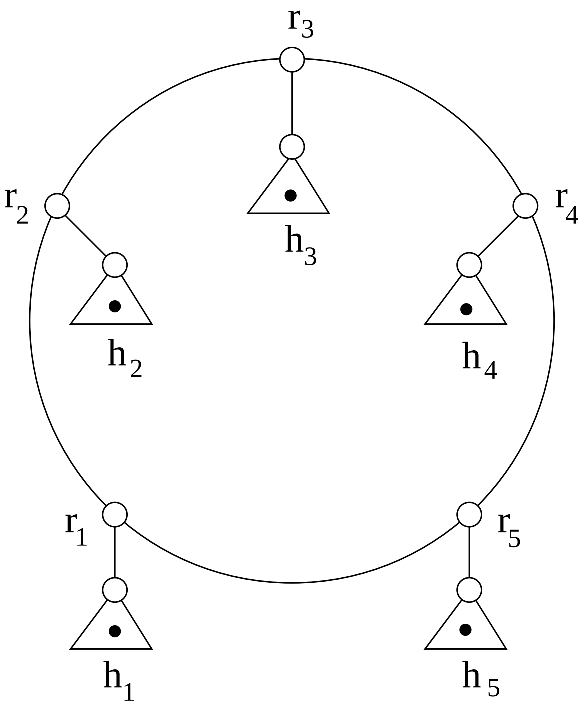


Figure 6: Five $(k-1)$-trees of type NCb

- If the order of the h vertices is h_1, h_2, h_3, h_4, h_5 then the necessary paths are $h_1, \ldots, r_1, r_5, \ldots, h_5$ and $h_2, \ldots, r_2, r_3, r_4, \ldots, h_4$.
- If the order of the h vertices is h_1, h_4, h_3, h_2, h_5 then the necessary paths are $h_1, \ldots, r_1, r_2, \ldots, h_2$ and $h_4, \ldots, r_4, r_5, \ldots, h_5$.
- If the order of the h vertices is h_1, h_5, h_3, h_2, h_4 then the necessary paths are $h_1, \ldots, r_1, r_2, \ldots, h_2$ and $h_4, \ldots, r_4, r_5, \ldots, h_5$. □

Lemma 3.7
Let U be a unicyclic graph containing a constituent vs-k-tree T and let L be a k-layout for U. Then one of the end vertices of L is in T.

Proof
Suppose neither of the end vertices of L, say α and ω, are in T. L induces a separation k layout L' for T and there exists a vertex u in T such that $w_{L'}(u) \geq k$. There exists a path from α to ω in U containing no vertex in T. Hence $w_L(u) \geq k+1$, contradicting our initial assumptions. □

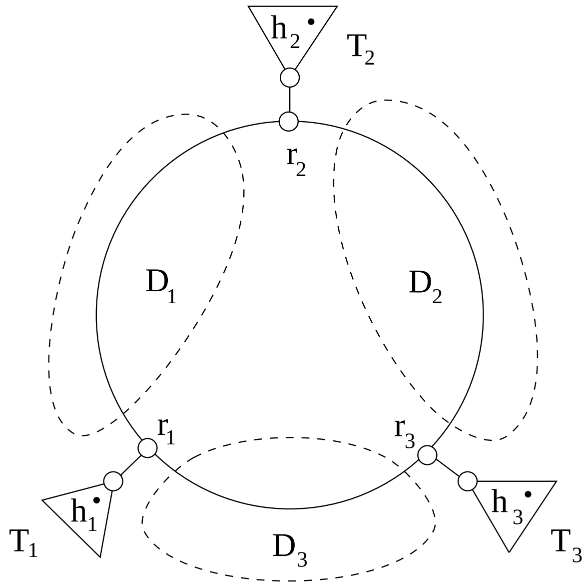


Figure 7: Three NCb vs-$(k-1)$-trees, none containing an endpoint of the layout

Lemma 3.8
Let U be the unicyclic graph containing four constituent vs-$(k-1)$-trees, each of type NCb, and let L be a k-layout for U, then the leftmost and rightmost vertices in L are in two distinct vs-$(k-1)$-trees.

Proof
We will refer to the four, type NCb, vs-$(k-1)$-trees, as the *heavy* trees. We establish that if the terminal vertices, say α and ω, of L are not both in distinct heavy trees, then $vs_L(U) \geq k+1$.

Suppose only one or neither of α, ω is in a heavy tree. Then there are at least three heavy trees, say T_1, T_2 and T_3, containing neither α nor ω. Let their roots be r_1, r_2 and r_3 respectively and let L_1, L_2 and L_3 respectively be the layouts for these trees induced by L. See Figure 7.

Let h_1, h_2 and h_3 be vertices in T_1, T_2 and T_3, such that $w_{L_1}(h_1), w_{L_2}(h_2), w_{L_3}(h_3) \geq k-1$. Note that such vertices exist which are distinct from the roots, because the trees are of type NCb. Without loss of generality, assume that h_2 lies between h_1 and h_3 in L.

We show that, no matter the location of α and ω in U, two paths, mutually vertex disjoint and each disjoint with T_2, exist which pass over h_2 in L, and that therefore $vs_L(U) \geq k+1$.

Taking symmetry into account, we need to consider just three cases: α and ω are both in D_i for some i, α is in D_1 and ω is in D_2, α is in D_1 and ω is in D_3.

- If α and ω are both in D_i for some i, then there is one path between α and ω that is confined to D_i. No matter which D_i is the subject, there is another path disjoint with the first between h_1 and h_3.

- If α is in D_1 and ω is in D_2, then the two paths are one between α and ω through D_1 and r_2 and D_2, and the other between h_1 and h_3 through D_3.
- If α is in D_1 and ω is in D_3, then let us first suppose that neither α nor ω is a tree root. Let r_α be the root of the tree containing α and let r_ω be the root of the tree containing ω.
  - If h_2 precedes r_α then one path lies between α and r_α, and the other between h_1 and h_3.
  - If r_ω precedes h_2 one path lies between r_ω and ω, and the other between h_1 and h_3.
  - Otherwise, the vertex order is: r_α, h_2, r_ω. If the order of the heavy vertices is h_1, h_2, h_3 then one path lies between h_1 and r_ω, and the other between r_α and h_3. If the order of the h vertices is h_3, h_2, h_1 then one path lies between h_3 and r_ω, and the other between r_α and h_1.

  If α is a tree root, it is necessary only to replace r_α by α in the path definitions. If ω is a tree root, it is necessary only to replace r_ω by ω in the path definitions. $\square$

Lemma 3.9
If U is a unicyclic graph containing three constituent vs-$(k-1)$-trees, each of type NCb, and L is a k-layout for U, then at least one of the end vertices in L is in one of the vs-$(k-1)$-trees.

Proof
Suppose neither of the endpoints is in one of the heavy trees. Then we can use exactly the same reasoning as in the proof of Lemma 3.8 to show that there is a vertex u from one of the heavy trees such that $w_L(u) \geq k+1$. $\square$

Definition 3.3
Let U be a unicyclic graph and let r_i and r_j be cycle vertices. A layout L for U will be said to be k-conforming *with respect to r_i and r_j if either $vs_L(U) = k$ and L is left extensible with respect to r_i and k, and right extensible with respect to r_j and k, or $vs_L(U) \leq k-1$.*

Lemma 3.10
Let U be the unicyclic graph containing (among others) two constituent trees, one a vs-$(k-1)$-tree of type NCb and one a vs-$(k-1)$-tree of type C, and two, degree two, cycle vertices r_i and r_j situated between the roots of these trees. Then there is no k-conforming layout for U with respect to r_i and r_j.

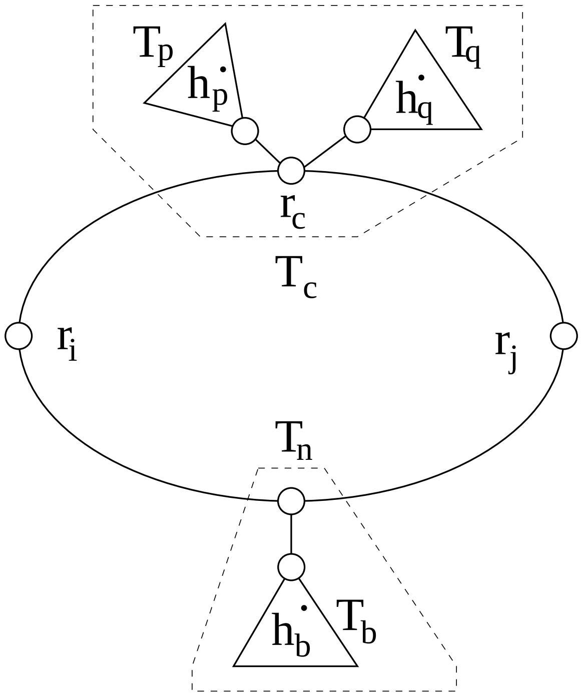


Figure 8: One vs-$(k-1)$-tree type NCb, one vs-$(k-1)$-tree type C

Proof
The situation is depicted in Figure 8. There T_n is the constituent vs-$(k-1)$-tree of type NCb and T_b is the minimum vs-$(k-1)$-subtree of type NCb in T_n. T_c is the constituent vs-$(k-1)$-tree of type C and T_p and T_q are the two vs-$(k-1)$-subtrees rooted at the children of the root, r_c, of T_c. Suppose a layout L for U exists with the prescribed separation and extensibilities. Consider the three minimal vs-$(k-1)$-trees that are proper subtrees of the two constituent trees. For each of the vs-$(k-1)$-trees, T_b, T_p and T_q, L induces layouts of separation at least $k-1$. So, in each of those layouts there exists a *heavy* vertex, not a cycle vertex, with weight at least $k-1$. Let h_p and h_q be the heavy vertices in T_p and T_q and let h_b be the heavy vertex in T_b.

Consider any sequence of these three heavy vertices and r_i, r_j in L.

- Suppose the third vertex, u, in that sequence is either h_p or h_q. Then it can be seen that, no matter the order of the other four vertices, there always exist two vertex disjoint paths between pairs chosen from these four that contain no vertex from T_p or T_q that pass over u. Hence $vs_L(U) = k+1$.

- Suppose the third vertex u is one of r_i, r_j, h_b.

  - If h_p and h_q are on opposite sides of the third vertex, then it can be seen that, for all arrangements of the vertices, there are two vertex disjoint paths between

h_p and h_q and between the other pair of heavy vertices that pass over the central vertex and contain no vertex in the minimal vs-$(k-1)$-tree containing that vertex. Hence $vs_L(U) = k+1$.

– Suppose h_p and h_q both precede u. By Lemma 2.6, if r_c is not between h_p and h_q, then the weight of one of them is at least k. Then L is not left extensible with respect to r_i.
  But if r_c is between h_p and h_q, there is a path from r_c to say r_j, containing no vertex in T_p or in T_q, that then passes over h_p or h_q. So the weight of one of these is at least k in L and L is not left extensible with respect to r_i.

– If h_p and h_q both follow u, a similar argument shows that L can not be right extensible with respect to r_j. □

4 The modified problem

In this section we work out a solution to a modified version of the general problem on a restricted form of unicyclic graph. This solution will be used in the next section to solve the general case. In the modified problem the unicyclic graph contains two, degree two, cycle vertices, say r_i and r_j. All the other constituent trees T are such that $vs(T) \leq k-1$. We show, for all restricted unicyclic graphs U and integers k, how to answer the question: "Does there exist a k-conforming layout for U?". If a conforming layout exists, the proofs are constructive, straightforwardly implying an algorithm for the construction of the layout.

The case, subcase structure of this section is quite complex. An outline of this structure, which may help the reader follow the arguments, is described in Figure 30 in Section 6.

The removal of r_i and r_j from U leaves a disconnected graph comprising two trees, say T_1 and T_2. One but not both of these may be empty. Without loss of generality, let $vs(T_1) \geq vs(T_2)$. We denote the vertices (if they exist) adjacent to r_i by a and c, and to r_j by b and d, where a and b are in T_1. The situation is illustrated in Figure 9. We consider three cases: $vs(T_1) = k$, $vs(T_1) < k-1$ and $vs(T_1) = k-1$.

Case 1: $vs(T_1) = k$.

Lemma 4.1
In this case, no k-conforming layout for U exists.

Proof
Any layout L for U induces a layout L_1 for T_1 that includes a vertex h such that $w_{L_1}(h) \geq k$. Suppose there exists a k-conforming layout L for U. Without loss of generality, let r_j be the rightmost of r_i and r_j in L. Rightmost(r_j) is either d or b or r_j itself. We require L to be left extensible with respect to r_i and right extensible with respect to r_j, hence h must lie strictly between r_i and rightmost(r_j) in L. If rightmost(r_j) is d or r_j itself, then there exists

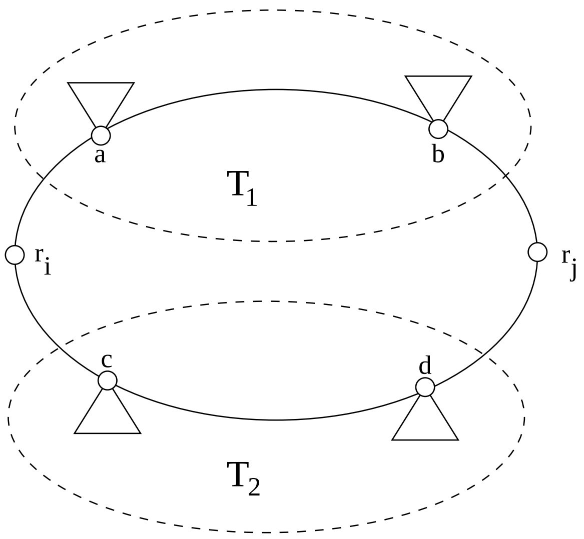


Figure 9: Degree two cycle vertices

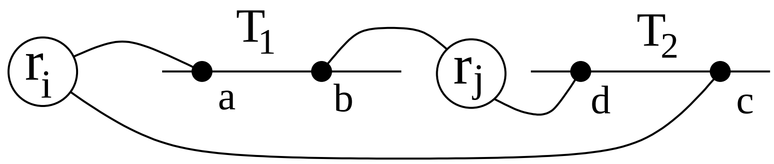


Figure 10: $vs(T_1) < k-1$ and $vs(T_2) < k-1$

a path in U from r_i to rightmost(r_j) through T_2, that is vertex disjoint with T_1 and which passes over h. Hence $w_L(h) > k$ and $vs_L(U) \geq k+1$.

If rightmost(r_j) is b and $h \neq b$ then there is a path in U from r_i to b through T_2, that is vertex disjoint with T_1, except for b, and which passes over h. Hence $w_L(h) > k$ and $vs_L(U) \geq k+1$.

If rightmost(r_j) is b and $h = b$, then L is not k-right extensible with respect to r_j. Thus in all cases we contradict the assumption that L is k-conforming. □

Case 2: $vs(T_1) < k-1$.

Lemma 4.2
In this case, a k-conforming layout for U exists.

Proof
By definition, $vs(T_2) < k-1$ and the construction illustrated in Figure 10 shows a k-conforming layout. Note that the relative orderings of a and b and of c and d are immaterial.

Case 3: $vs(T_1) = k - 1$.
We recall that although an extensible, optimal layout may exist with respect to a particular vertex, there may not exist an extensible, optimal layout with respect to some other vertex. $T_1[a]$ denotes T_1 when rooted at a and similarly for $T_1[b], T_2[c]$ and $T_2[d]$. We consider two cases: either both $T_1[a]$ and $T_1[b]$ are $(k-1)$-critical, or at least one of them is not $(k-1)$-critical.

Case 3.1 $T_1[a]$ is $(k-1)$-critical and $T_1[b]$ is $(k-1)$-critical.

In this case, either T_1 contains exactly one constituent vs-$(k-1)$-tree of type Cb or it contains exactly one constituent vs-$(k-1)$-tree of type C or it contains exactly two constituent vs-$(k-1)$-trees of type NCb. It can not be the case that there is one constituent vs-$(k-1)$-tree of type NCb whose removal from T_1 leaves two vs-$(k-1)$-trees, as then $vs(T_1) = k$. In all three valid cases, the vertex separation of any other constituent tree can not be greater than $k-2$, otherwise $vs(T_1) \geq k$. We consider the three possibilities separately.

T_1 contains one constituent vs-$(k-1)$-tree of type Cb

Let the buried $(k-1)$-critical tree be T', rooted at a vertex u, and containing subtrees T'_1 and T'_2 of vertex separation $k-1$. Note that T' is a proper subtree of some constituent tree, say T_w, rooted at some cycle vertex w. Let $U' = U - T'$. The situation is illustrated in Figure 11.

Lemma 4.3
In this case, there exists a k-conforming layout L for U iff there exists a $(k-1)$-conforming layout L' for U'.

Proof

- If there exists a $(k-1)$-conforming layout L' for U' then there exists a k-conforming layout for U by way of the construction illustrated in Figure 12. By Lemma 2.5 there exists a layout L for T' in which u is singular. Hence, inserting L' immediately to the right of u in L produces a vertex separation k layout for U, since there is only one edge between the two layouts, and that edge is incident with u. Note also that, in L', no vertex to the left of r_i has weight greater than $k-2$ and neither rightmost(r_j) nor any vertex to the right of rightmost(r_j) has weight greater than $k-2$.
- Suppose there exists a k-conforming layout L for U. Let L_w be the layout for T_w induced by L and let L' be the layout for T' induced by L. There exist at least two vertices in L_w, one in T'_1 and one in T'_2, of weight at least $k-1$. Let us call such vertices from T'_1 and T'_2 the *heavy* vertices. Let h_1 be the leftmost of all the heavy vertices in L_w and, without loss of generality, let h_1 belong to T'_1. Let h_2 be the rightmost heavy vertex belonging to T'_2.

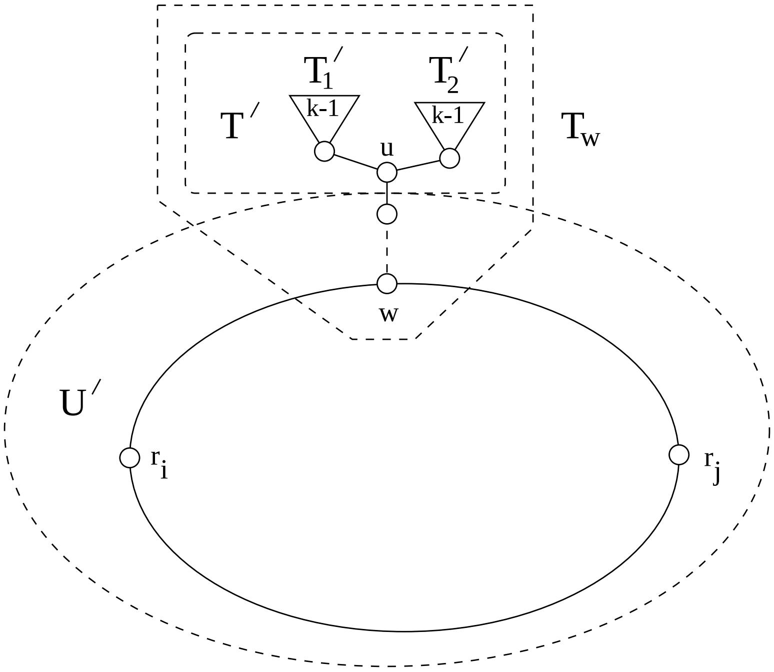


Figure 11: T_1 contains a constituent vs-$(k-1)$-tree, type Cb

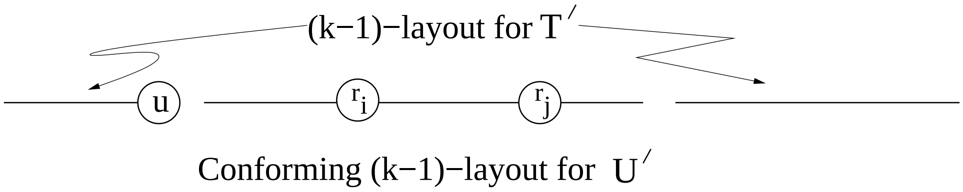


Figure 12: A layout for U using a $(k-1)$-conforming layout for U'

- By Lemma 2.2 we may assume that L is a k-conforming layout for U in which r_i is the leftmost vertex.

- We note that $vs_{L_w}(T_w) = k - 1$ because, if there is a vertex v in $T[w]$ such that $w_{L_w}(v) = k$, then v does not lie to the right of rightmost(r_j) in L, because L is k-conforming. But, if v lies between r_i and rightmost(r_j), its weight in L must be at least $k+1$, because there is a path from r_i to rightmost(r_j) that is vertex disjoint with T_w and that passes over v. The argument still holds when rightmost$(r_j) = w$.

- By the same argument, $vs_{L'}(T') = k - 1$.

- By Lemma 2.6, any vertex in T_w and not in either T'_1 or T'_2 must lie between h_1 and h_2 in L_w. Therefore w lies between h_1 and h_2 in L.

- Let v be any vertex not in T_w and not r_i. Then v can not lie to the left of h_1 in L, because there would be two vertex disjoint (except for w) paths, from r_i to w and from v to w that are also vertex disjoint with T_w and which pass over h_1. Then $vs_L(U)$ would be not less than $k+1$.

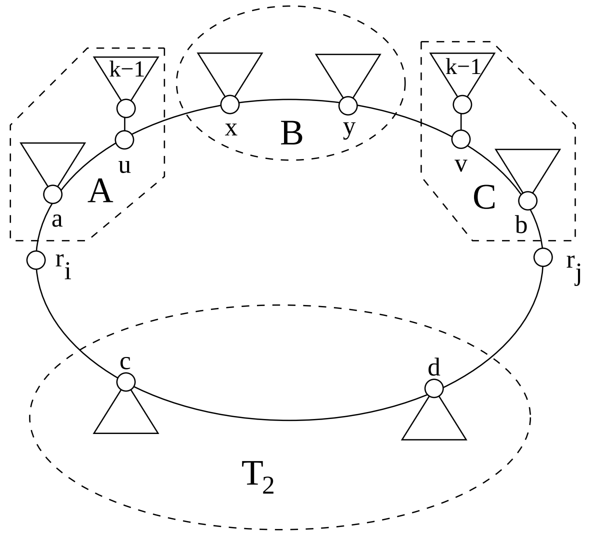
- Let c be any cycle vertex other than r_i and w. Then c can not lie to the right of h_2 in L, because then there would be two vertex disjoint (except for c) paths, from r_i to c and from w to c, that that are also vertex disjoint with T_w and pass over h_2. Then $vs_L(U)$ would be not less than $k+1$.

- Suppose v is any vertex, not r_i, not in the cycle and not in T_w. Then v can not lie to the right of h_2 by the following argument. There is a path from r_i to v, vertex disjoint with T_w, and passing over h_2. Hence $w_L(h_2) \geq k$. But rightmost(r_j) is a cycle vertex and so lies to the left of h_2. So then L is not right extensible with respect to r_j and k.

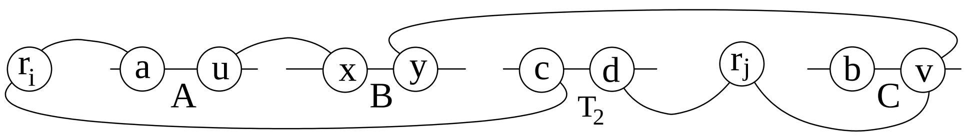
- We conclude that all vertices not in T'_1 and not in T'_2, except for r_i, lie between h_1 and h_2. Let α be the vertex immediately following r_i in L, and let ω be the rightmost vertex in L. Then α and ω must be in T'. There is a path from α and ω including no vertex in U'. Removing this path leaves a layout in which the weight of every remaining vertex, except for r_i whose weight is still one, is at least one less than its weight in L. Hence this reduced layout induces a $(k-1)$-conforming layout for U'. □

T_1 contains two constituent vs-$(k-1)$-trees of type NCb

We partition this case into two subcases: T_2 contains no constituent vs-$(k-1)$-



a) The graph



b) A k−conforming layout

Figure 13: $T_1[a]$ and $T_1[b]$ are both $(k-1)$-critical

trees of types other than NC and T_2 contains at least one constituent vs-$(k-1)$-tree of type other than NC.

- Suppose T_2 contains no constituent vs-$(k-1)$-trees of type other than NC.

  Lemma 4.4

  In this case, a k-conforming layout for U exists.

  Proof

  Let the type NCb, vs-$(k-1)$-trees in T_1 be rooted at cycle vertices u and v, where u is the closest to r_i in T_1. Let A be that part of T_1 comprising all the constituent trees rooted at cycle vertices in the path from a to u. Let B be that part of T_1 comprising all the constituent trees rooted at cycle vertices in the path from x to y, where x is the the cycle vertex adjacent to u and closest to r_j in T_1, and y is the cycle vertex adjacent to v and closest to r_i in T_1, if these vertices exist. Let C be that part of T_1 comprising all the constituent trees rooted at cycle vertices in the path from v to b. The situation is illustrated in Figure 13, part a).

  Note that $A[u]$ is not $(k-1)$-critical because then $vs(T_1) = k$. Likewise $B[v]$ is not $(k-1)$-critical. Hence, by Lemma 2.4, there is an $(k-1)$-layout for

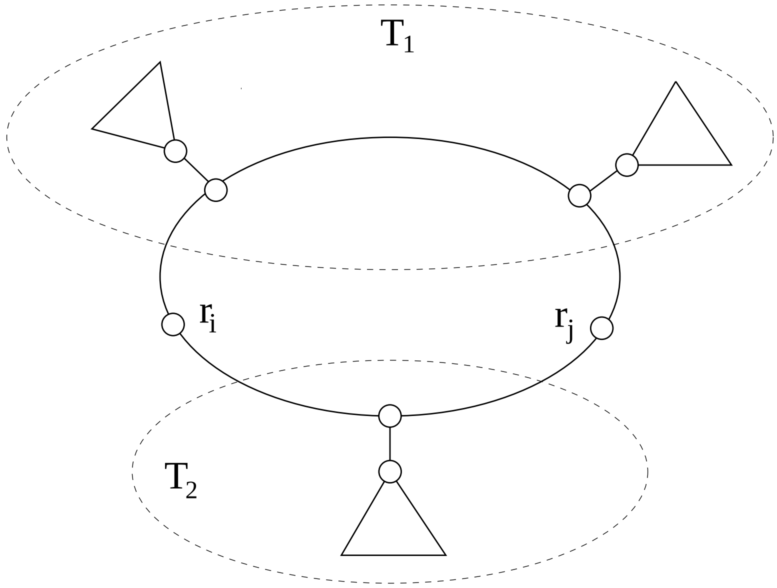


Figure 14: T_2 contains a constituent vs-$(k-1)$-tree not of type NC

A that is right extensible with respect to u and there is a $(k-1)$-layout for C that is left extensible with respect to v. Note also that, since both B and T_2 contain no vs-$(k-1)$-trees of types other than NC, by Lemma 2.7, there is a $(k-1)$-layout for T_2 that is left extensible with respect to c and right extensible with respect to d and there is a $(k-1)$-layout for B that is left extensible with respect to x and right extensible with respect to y.
These observations imply that the layout described in Figure 13, part b) is k-conforming with respect to r_i and r_j. Note that the relative orderings of a and u in the layout for A and of v and B in the layout for C are immaterial.
□

- Suppose T_2 contains at least one constituent vs-$(k-1)$-tree of type other than NC.

  Lemma 4.5
  In this case, no k-conforming layout for U exists.

  Proof
  Consider U' in which we replace r_i and r_j in U by trees of type NCb and vertex separation $k-1$ rooted at r_i and r_j. Suppose a k-conforming layout exists for U. Figure 15 defines the construction of a layout L for U', using this k-conforming layout. Since r_i and r_j are each in two of the constituent layouts, we use the dotted circles to indicate that one of the positions of r_i and r_j not used in the compound layout. By Lemma 2.4, left and right extensible layouts exist for $T[r_j]$ and $T[r_j]$. Hence $vs_L(U') = k$.

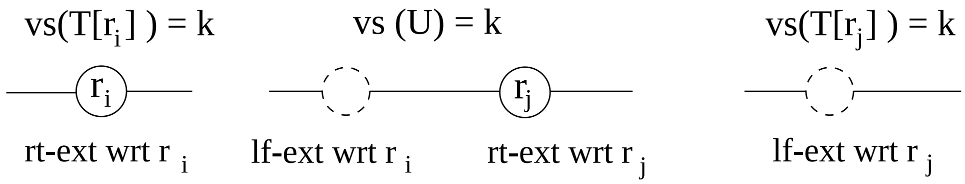


Figure 15: An impossible result

But U' contains a subgraph homeomorphic to the unicyclic graph containing five constituent trees of type NCb and vertex separation $k-1$, as shown in Figure 6. Hence, by Lemma 3.6, no such L exists. □

T_1 contains one constituent vs-$(k-1)$-tree of type C

We partition this case into two subcases: T_2 contains no constituent vs-$(k-1)$-trees of types other than NC and T_2 contains at least one constituent vs-$(k-1)$-tree of type other than NC.

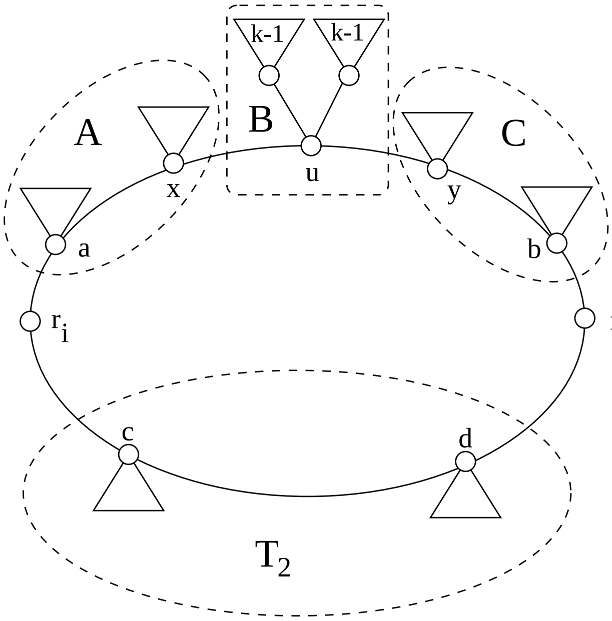
- Suppose T_2 contains no constituent vs-$(k-1)$-trees of types other than NC. The situation is illustrated in Figure 16.

  Lemma 4.6
  In this case, there exists a k-conforming layout for U.

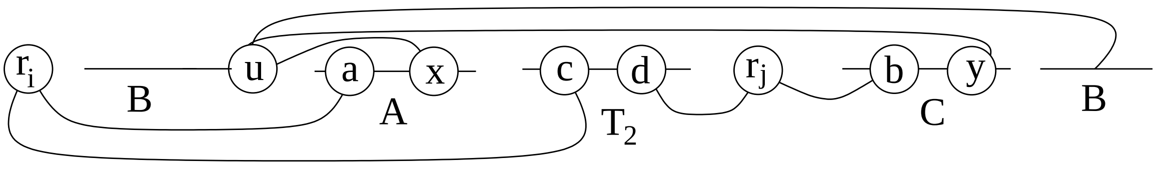
  Proof
  We divide the graph into four trees, plus the vertices r_i, r_j, as illustrated in Figure 16, a):

  - A is the tree comprising all the constituent trees rooted at vertices strictly between r_i and u, plus the incident cycle edges. We note that $vs(A) \leq k-2$, else $vs(T_1) \geq k$, contrary to the assumptions for this case.
  - B is the vs-$(k-1)$-tree of type C. By Lemma 2.5 there exists a layout L_B for B such that $vs_{L_B}(B) \leq k-1$ in which u is singular.
  - C is the tree comprising all the constituent trees rooted at vertices strictly between r_j and u, plus the incident cycle edges. We note that $vs(A) \leq k-2$, else $vs(T_1) \geq k$, contrary to the assumptions for this case.
  - T_2 is as previously defined. We are considering the case where all constituent trees in T_2 are of separation $k-1$ and of type NC, or have separation strictly less than $k-1$. Hence, by Lemma 2.7, there exists a layout L_2 for T_2 such that $vs_{L_2}(T_2) \leq k-1$, L_2 is left extensible with respect to c and right extensible with respect to d, and c precedes d in L_2.

  Consequently the construction illustrated in Figure 16 b) defines a k-conforming layout for U. □



a) The graph



b) The layout

Figure 16: T_1 contains one constituent vs-$(k-1)$-tree of type C

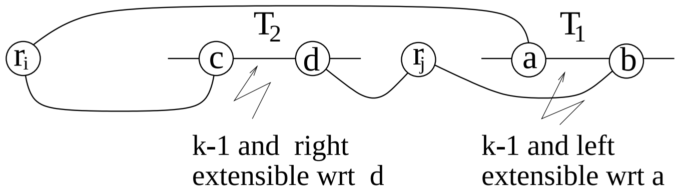


Figure 17: Complementary Extensibilities

Suppose T_2 contains at least one constituent vs-$(k-1)$-tree of type other than NC.

Lemma 4.7
In this case, there does not exist a conforming layout for U.

Proof
U contains a subgraph homeomorphic to the graph illustrated in Figure 8. Hence, by Lemma 3.10, no k-conforming layout exists for U. $\square$

Case 3.2 At least one of $T_1[a]$, $T_1[b]$ is not critical.
We may assume that not both $T_2[c]$, $T_2[d]$ are not $(k-1)$-critical, else we have Case 3.1 with T_1 and T_2 exchanged.

We will say that T_1 and T_2 have *complementary extensibilities* if

- either $T_1[a]$ is not $(k-1)$-critical and $T_2[d]$ is not $(k-1)$-critical
- or $T_1[b]$ is not $(k-1)$-critical and $T_2[c]$ is not $(k-1)$-critical.

Lemma 4.8
In this case, there exists a k-conforming layout L for U iff T_1 and T_2 have complementary extensibilities.

Proof

Suppose that $T_1[a]$ is not $(k-1)$-critical and $T_2[d]$ is not $(k-1)$-critical. Then by Lemmas 2.3 and 2.4, there exists a layout for T_1 that is left extensible with respect to a and there exists a layout for T_2 that is right extensible with respect to d. Figure 17 describes a way of arranging the components of U so as to construct a k-conforming layout. Note that it matters not whether b is left or right of a, or whether c is left or right of d. If $T_1[b]$ is not critical and $T_2[c]$ is not critical then we just have the mirror image of the previous situation.

To demonstrate the implication in the other direction we show that if T_1 and T_2 do not have complementary extensibilities and L is a layout for U and L is left extensible with respect to r_i and right extensible with respect to r_j then $vs_L(U) \geq k+1$.

Without loss of generality, let both $T_1[a]$ and $T_2[c]$ be $(k-1)$-critical. Consider the layouts L_1 and L_2 for T_1 and T_2 induced by L. If $vs_{L_1}(T_1) = k$ or $vs_{L_2}(T_2) = k$ then $vs_L(U) \geq k+1$, by Lemma 4.1. If $vs_{L_1}(T_1) < k$ and $vs_{L_2}(T_2) < k$, then, by Lemma 2.6, there exist vertices h_1 and h_2 in T_1 and T_2 such that $w_{L_1}(h_1), w_{L_2}(h_2) \leq k-1$ and h_1 precedes a and h_2 precedes c in L.

There are six possible relative orderings of these four vertices, consistent with these constraints:

$$\ldots h_2 \ldots c \ldots h_1 \ldots a \ldots \tag{1}$$
$$\ldots h_2 \ldots h_1 \ldots c \ldots a \ldots \tag{2}$$
$$\ldots h_2 \ldots h_1 \ldots a \ldots c \ldots \tag{3}$$
$$\ldots h_1 \ldots h_2 \ldots c \ldots a \ldots \tag{4}$$
$$\ldots h_1 \ldots h_2 \ldots a \ldots c \ldots \tag{5}$$
$$\ldots h_1 \ldots a \ldots h_2 \ldots c \ldots \tag{6}$$

In the first case the path c, r_i, a implies that the weight of h_1 in L is at least k. If r_j precedes h_1 then b and d must follow h_1, because L is right extensible with respect to r_j. But then either the edge $\{r_j, d\}$ of the edge $\{r_j, b\}$ implies that the weight of h_1 is at least $k+1$. Note that if $b = h_1$ and $b = \text{rightmost}(r_j)$, then L is not right extensible. If r_j follows h_1, then the path from h_2 to r_j through T_2 also implies that the weight of h_1 is at least $k+1$. Hence $vs_L(U) \geq k+1$.

In the second and third cases there is a path from h_2 to c in T_2 that passes over h_1, and hence $w_L(h_1) \geq k$. Hence r_i must precede h_1 in L. But then the edge $\{r_i, a\}$ also contributes to the weight of h_1. Hence $vs_L(U) \geq k+1$.

In the fourth and fifth cases there is a path between h_1 and a in T_1 that passes over h_2, and hence $w_L(h_2) \geq k$. Hence r_i must precede h_2 in L. But then the edge $\{r_i, c\}$ also contributes to the weight of h_2. Hence $vs_L(U) \geq k+1$.

In the sixth case the path a, r_i, c implies that the weight of h_2 in L is at least k. If r_j precedes h_2 then b or d must follow h_2, because L is right extensible with respect to r_j. But then either the edge $\{r_j, d\}$ or the edge $\{r_j, b\}$ implies that the weight of h_2 is at least $k+1$. Note that if $d = h_2$ and $d = \text{rightmost}(r_j)$, then L is not right extensible. If r_j follows h_2, then the path from h_1 to r_j through T_1 also implies that the weight of h_2 is at least $k+1$. Hence $vs_L(U) \geq k+1$. $\square$

5 Computing vertex separation and optimal layouts for unicyclic graphs

We propose a process comprising two phases. The first phase consists of two steps, parts of which could be combined, but which we describe separately for clarity.

1. For each constituent tree in the graph, compute its vertex separation, type and an optimal layout.

2. Remove some cycle edge from U yielding a tree T. Compute the vertex separation of and an optimal layout L for T. Let $vs(T) = k$.

The problem then is to determine whether $vs(U) = k$ or $vs(U) = k + 1$. In this section we describe a procedure which computes the answer to this question. If $vs(U) = k$, the procedure also computes an optimal layout. If $vs(U) = k + 1$ then L, computed in Phase 1, is an optimal layout.

In the second phase of the process, we either complete the solution or we reduce the problem to the problem of a finding a k-conforming layout for a unicyclic graph U', where U' is obtained from U by replacing two constituent trees by their roots alone. This latter is the problem we worked out in the previous section. We divide the analysis into five cases.

Case 1 There exists exactly one k-critical constituent tree.

Case 2 There exist three or more vertex separation k, constituent trees, none of them critical.

Case 3 There exist exactly two vertex separation k, constituent trees, none of them critical.

Case 4 There exists exactly one vertex separation k, constituent tree, and it is not critical.

Case 5 There do not exist any vertex separation k, constituent trees.

We note that there can not exist more than one k-critical constituent tree, else $vs(U - e)$, computed in Phase 1, would be $k + 1$, for any edge e in the cycle. Hence the five cases divide the set of problem instances into disjoint sets such that any instance is in exactly one set. We proceed by analysing each case. Note that in all cases, if $vs(U) = k$, a constructive proof is given, which defines how an optimal layout can be built.

We use the following lemma several times. By the *body* of a rooted tree we mean the entire tree minus the root and edges incident with the root.

Lemma 5.1
Let L be a layout for the unicyclic graph U such that $vs_L(U) = k$. Let the leftmost vertex in L be α. Let the cycle vertex which is the root of the constituent tree containing α, say T_α, be r_α. Let L' be the layout obtained by removing the vertices in the body of T_α from L.
Let the rightmost vertex in L' be ω, and let the cycle vertex which is the root of the constituent

tree, say T_ω, containing ω be r_ω. Let L'' be the layout obtained by removing the vertices in the body of T_ω from L'.
Let U'' be U minus the bodies of T_α and T_ω. Then L'' is a k-conforming layout for U'' with respect to r_α and r_ω.

Proof
By definition, L'' is a layout for U'' and certainly $vs_{L''}(U'') \leq k$. Let U' be U minus the body of T_α. If α is r_α then L' is left extensible with respect to r_α. If α is not r_α then there is a path from α to r_α in T_α containing no vertex in U', except for r_α. The removal of the body of T_α from L removes this path. Hence, the weight of all vertices in L' to the left of r_α is at least one less than its weight in L, and this must also be true for L''.

If ω is rightmost(r_ω) in L'', then L'' is right extensible with respect to r_ω. If ω is not rightmost(r_ω) then there is a path from r_ω to ω, containing no vertex in U'' except r_ω. In L', this path passes through or over rightmost(r_ω) and over every vertex in U' to the right of rightmost(r_ω). The removal of the body of T_ω from L' removes this path. Hence, in L'', the weight of rightmost(r_ω) and of any vertex remaining to the right of rightmost(r_ω) is reduced by at least one. Hence L'' is a k-conforming layout for U'' with respect to r_α and r_ω. □

The case, subcase structure of this section is quite complex. An outline of this structure, which may help the reader follow the arguments, is described in Figure 29 in Section 6.

5.1 Case 1

There exists exactly one k-critical constituent tree. Denote this tree by T_c and the two cycle edges incident with its root by e_1 and e_2. Consider the tree $T' = U - T_c$. See Figure 18.

If $vs(T') = k$ then, by Lemma 2.1, $vs(U) \geq k+1$. So let us assume that $vs(T') \leq k-1$.

- Suppose r is the k-critical vertex, i.e., T_c is of type C. Figure 19 shows that there is a layout L such that $vs_L(U) = k$. By Lemma 2.5 there exists an optimal layout L' for T_c in which r is singular. Consequently we can insert an optimal $(k-1)$-layout for T' immediately to the right of r in L' and achieve a k-layout for U, because the two edges e_1 and e_2 add only the one vertex r to the left of any vertex in the layout for T'.

- Suppose some vertex u, not the root, is the k-critical vertex in T_c. If $vs(U-T[u]) \leq k-1$ then, because there exists a k-layout for $T[u]$ in which u is singular, we can insert the $(k-1)$-layout for $(U - T[u])$ immediately to the right of u to obtain a k-layout for U. See Figure 20.
  If $vs(U - T[u]) = k$, then $vs(U) \geq k+1$, by Lemma 2.1.

5.2 Case 2

There exist three or more vertex separation k, constituent trees, none of them critical. Consequently there is a subgraph of U homeomorphic to the unicyclic graph with exactly 3

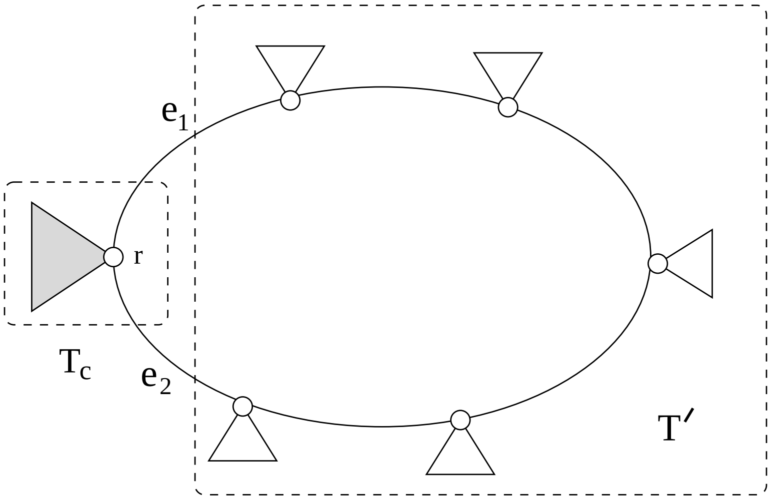


Figure 18: T_c is k-critical

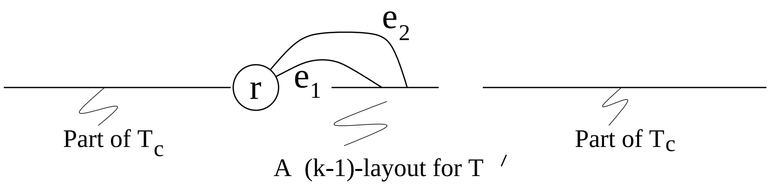


Figure 19: A k-layout when r is the k-critical vertex

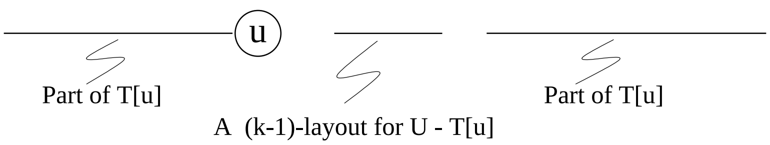


Figure 20: Case 1, a k-layout when r is not the k-critical vertex and $vs(U - T[u]) = k - 1$

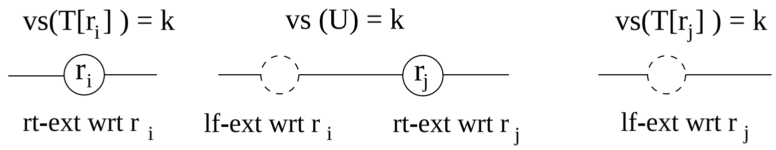


Figure 21: Case 3, two constituent vs-k-trees, not critical

constituent trees, each of vertex separation k, and of any type. By Lemma 2.1, this graph has vertex separation at least $k+1$, hence $vs(U) = k+1$.

5.3 Case 3

There exist exactly two constituent vs-k-trees, say T_i and T_j, neither of them critical. In this case, let $U' = U$ minus the bodies of T_i and T_j and let r_i be the root of T_i and r_j be the root of T_j.

Lemma 5.2
In this case, $vs(U) = k$ iff there exists a k-conforming layout for U' with respect to r_i and r_j.

Proof
If there is a k-conforming layout for U' then Figure 21 defines the construction of a layout L for U such that $vs_L(U) = k$. Since r_i and r_j are each in two of the three layouts, we use the dotted circles to indicate that one of the positions of r_i and r_j is not used in the complete layout. Note that, because T_j and T_j are not critical, left and right extensible layouts exist, by Lemmas 2.3 and 2.4.

To see the implication in the other direction, suppose L is a layout for U such that $vs_L(U) = k$. Let α and ω be the leftmost and rightmost vertices respectively in L. Let L_i and L_j be the layouts for T_i and T_j induced by L. We show that α must be in one of T_i, T_j and that ω must be in the other.

Let h_i and h_j be vertices in T_i, T_j such that $w_{L_i}(h_i) = w_{L_j}(h_j) = k$ and, without loss of generality, let h_i precede h_j in L. Suppose α is not in T_i. Then there is a path in U from α to h_j, containing no vertex in T_i, that passes over h_i, and hence $w_L(h_i) > k$.

Suppose ω is not in T_j. Then there is a path in U from ω to h_i, containing no vertex in T_j, that passes over h_j, and hence $w_L(h_i) > k$.

Hence, since we assume that $vs_L(U) = k$, it must be that α is in T_i, and that ω is in T_j. Let L' be the layout for U' obtained by removing the bodies of T_i and T_j from L. By Lemma 5.1, L' is k-conforming with respect to r_i and r_j. $\square$

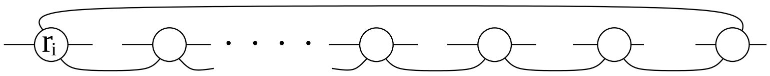


Figure 22: Case 4, type NC constituent trees only

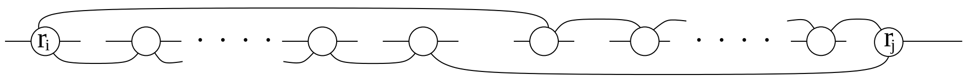


Figure 23: Case 4, one constituent vs-$(k-1)$-tree of type not NC

5.4 Case 4

There exists exactly one constituent vs-k-tree, say T_i, and it is not critical. Let r_i be the root of T_i. All other constituent trees have separation not greater than $k-1$. Let q be the number of constituent trees of separation $k-1$ and not of type NC.

$q = 0$

For all constituent vs-$(k-1)$-trees of type NC there exist $(k-1)$-layouts that are both left and right extensible with respect to their roots. Hence, since there are no constituent vs-$(k-1)$-trees not of type NC, the construction shown in Figure 22 creates a k-layout for U. A right extensible layout for the k-tree is placed as the leftmost with root r_i.

$q = 1$

Let r_j be the cycle vertex root of the one vs-$(k-1)$-tree, say T_j, not of type NC. There exists a k-layout for T_j which can be obtained from any $(k-1)$-layout by moving r_j to the leftmost position in that layout. Placing this layout as the rightmost of all the constituent tree layouts can produce a k-layout for U, as illustrated in Figure 23.

$q = 2$ **Lemma 5.3**

Let the two vs-$(k-1)$-trees, T_u and T_v, not of type NC be rooted at cycle vertices u and v. In this case, there exists a k-layout for U iff either there exists a k-conforming layout for U minus the bodies of T_i and T_u with respect to r_i and u or there exists a k-conforming layout for U minus the bodies of T_i and T_v with respect to r_i and v.

Proof

The situation is illustrated in Figure 24. S_1 is the tree formed by all constituent trees rooted at cycle vertex strictly between r_i and u and the adjacent cycle edges. S_2 is the tree formed by all constituent trees rooted at cycle vertex strictly between u and v and the adjacent cycle edges. S_3 is the tree formed by all constituent trees rooted at cycle vertex strictly between v and r_i and the adjacent cycle edges.

If either there exists a k-conforming layout for U minus the bodies of T_i and T_u with respect to r_i and u or there exists a k-conforming layout for U minus the bodies of T_i

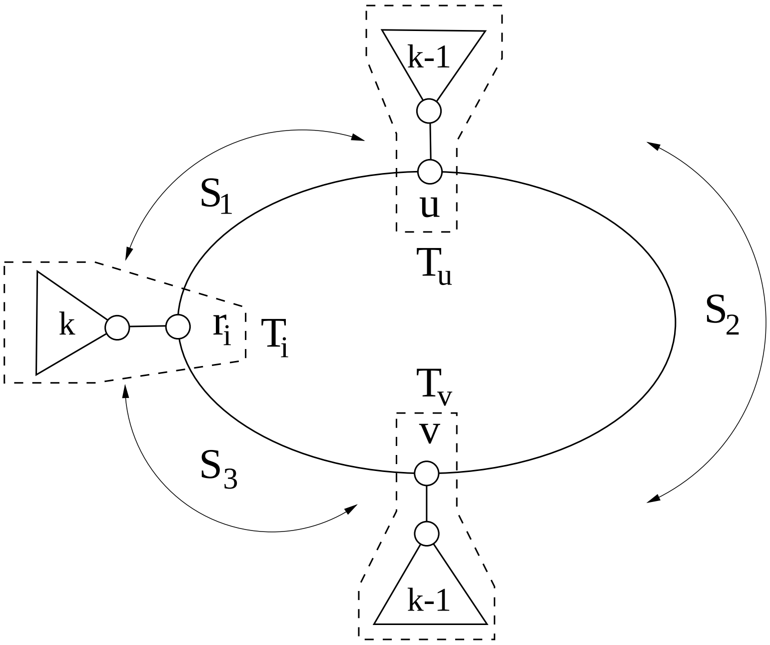


Figure 24: Case 4, two constituent vs-$(k-1)$-trees of type NCb

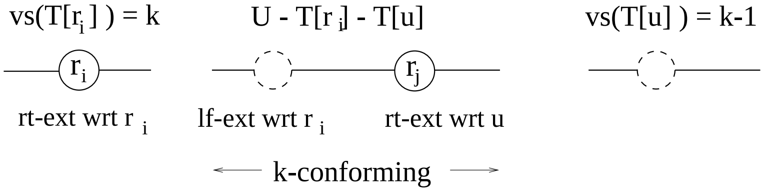


Figure 25: Case 4, $q = 2$, a layout in the k-conforming case

and T_v with respect to r_i and v, then the construction shown in Figure 25 shows that a k-layout exists for U.

To show the implication in the other direction, let L be a k-layout for U.

- We first show that either the leftmost, α, or the rightmost, ω, vertex in L is in T_i. If L_i is the layout for T_i induced by L, then there is a vertex h in L_i such that $w_{L_i}(h) = k$. Suppose neither α nor ω is in T_i. Then there is a path from α to ω, containing no vertex in T_i, that passes over h. Hence $w_L(h) > k$, contrary to our assumption.

  Of the two possibilities, let us assume that α is in T_i. The following argument is easily modified to cover the other possibility, namely that ω is in T_i.

- If ω is in T_u or in T_v then the implication follows immediately from Lemma 5.1. So in the remainder of the argument, let us assume the contrary.

- Next we show that it is not the case that $vs(S_i) = k-1$ for all i.
  Now let ω be the rightmost vertex in the layout obtained from L by removing all vertices in the body of T_i. Let r_j be the cycle vertex that is the root of the constituent tree T_j containing ω. Let U' be U minus the bodies of T_i and T_j and let L' be the layout obtained by removing the bodies of T_i and T_j from L. By Lemma 5.1, L' is a k-conforming layout for U' with respect to r_i and r_j.
  Suppose, contrary to our claim, that $vs(S_i) = k-1$ for all i. Let T_1 and T_2 be as defined at the beginning of Section 4, with respect to r_i and r_j, and recall that, without loss of generality, we let $vs(T_1) \geq vs(T_2)$.
  - If r_j is in S_2, then both T_1 and T_2 are $(k-1)$-critical and the criticalities are not complementary. Hence, by Lemma 4.8, no k-conforming layout with respect to r_i and r_j exists for U'.
  - If r_j is in S_1 or in S_3, then $vs(T_1) = k$ or $vs(T_2) = k$ by Theorem 2.1. Hence, by Lemma 4.1, no k-conforming layout with respect to r_i and r_j exists for U'.
- For the remainder of the argument we note that the situation is symmetrical with respect to u and v, and so the argument holds with the names u and v exchanged. We show that there must exist a conforming layout for U' with respect to r_i and u. We are already assuming that r_j is not u. We show that then neither T_u nor T_v can be $(k-1)$-critical.
  - Suppose r_j lies in S_1. If one of T_u, T_v is critical then $vs(T_1) = k$ and hence, by Lemma 4.1 no k-conforming layout exists for U'.
  - Suppose r_j lies in S_2 and that T_u is $(k-1)$-critical and of type Cb. Then, by Lemma 4.3, there is a k-conforming layout for U with respect to r_i and r_j iff there exists a $(k-1)$-conforming layout for U minus the $(k-1)$-critical subtree in T_u. Since $vs(T_2) = k-1$ by virtue of T_v, no such $(k-1)$-conforming layout exists, by Lemma 4.1.
  - Suppose r_j lies in S_2 and that T_u is $(k-1)$-critical and of type C. Since $vs(T_v) = k-1$, no such $(k-1)$-conforming layout exists, by Lemma 4.7.
- At this point we have that $vs(S_i) < k-1$ for some i, and that neither T_u nor T_v is $(k-1)$-critical.
  Suppose $vs(S_1) < k-1$. Then we may choose v to be r_j, making T_1 $(k-1)$-critical in one direction only, i.e., either $T_1[a]$ or $T_1[b]$ is critical, but not both. Since S_3 $= T_2$ is not $(k-1)$-critical, there is a k-conforming layout for U minus the bodies of T_i and T_v, by Lemma 4.8. The same reasoning leads to the same result for the case $vs(S_3) < k-1$, by choosing u to be r_j.
- Suppose $vs(S_2) < k-1$. Then we may choose v to be r_j and again we make T_1 $(k-1)$-critical in one direction only and the same positive result is derived. □

$q = 3$

Lemma 5.4
In this case, there exists a k-conforming layout for U iff, for some choice of T_j, with root r_j, from among the three vs-$(k-1)$-trees not of type NC, there is a k-conforming layout L' for $U' = U$ minus the bodies of T_i and T_j, with respect to r_i and r_j.

Proof
If there exists a k-conforming layout for U', the construction illustrated in Figure 27 shows that a k-layout for U exists.

To see the implication in the other direction, consider any optimal layout L for U. By Lemma 3.7, one of the end vertices of L must be in the constituent vs-k-tree, T_i with root r_i. By Lemma 3.9, one of the end vertices of L must be in one of the constituent vs-$(k-1)$-trees, say T_j with root r_j. Let L' be the layout obtained be removing the bodies of T_i and T_j from L. By Lemma 5.1, L' is a k-conforming layout for U' with respect to r_i and r_j. □

$q \geq 4$

In this case, U contains a subgraph homeomorphic to the graph in Figure 6. Hence, by Lemma 3.6, no k-layout exists for U.

5.5 Case 5

All constituent trees are of vertex separation no greater than $k-1$. Note that there must be at least one constituent tree of vertex separation $k-1$ and not of type NC, else by Lemma 2.7 there is a $(k-1)$-layout for any of the trees formed by removing one cycle edge, and we have assumed that this is not the case. Let q be the number of constituent trees of vertex separation $k-1$ and not of type NC. We consider four sub-cases, $1 \leq q \leq 2, q = 3, q = 4$ and $q \geq 5$.

$1 \leq q \leq 2$

Let T_i be one of the constituent trees of vertex separation $k-1$ and not of type NC, and let T_j be the other such, if it exists, or any of the other constituent trees if not. Let r_i and r_j be the roots of these trees and let U' be U minus the bodies of T_i and T_j. All the remaining constituent trees are of separation $k-1$ and type NC, or are of separation less than $k-1$. Therefore, by Lemma 2.3, each such tree has an optimal layout that is both left and right extensible with respect to its root. Hence there is a k-conforming layout of U' as illustrated in Figure 26. Consequently, we can construct a k-layout for U by concatenating first an optimal layout for T_i, then the layout for U' minus r_i, and then an optimal layout for T_j, minus r_j. This construction is illustrated in Figure 27, where the dotted circles indicate the unused, duplicate positions of r_i and r_j.

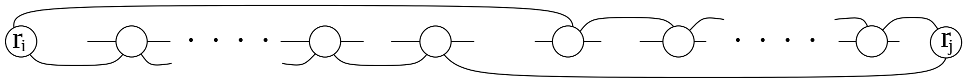


Figure 26: Case 5, $q \leq 2$, a k-conforming layout for U'

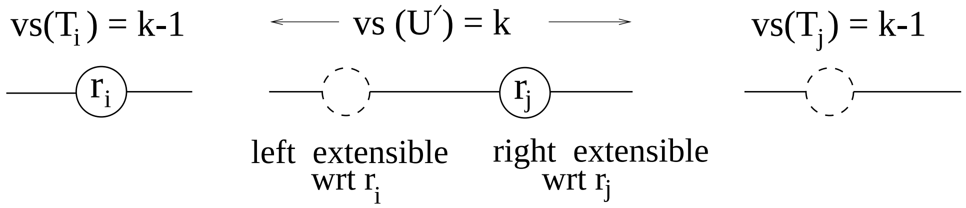


Figure 27: Case5, $q \leq 2$, a k-layout for U

$q = 3$ **Lemma 5.5**

There exists a k-layout for U iff for some choice of two trees, say T_i and T_j, from the three vs-$(k-1)$-trees there exists a k-conforming layout for U', where U' is U minus the bodies of T_i and T_j.

Proof

For each of the vs-$(k-1)$-trees there exist optimal layouts that are either left or right extensible with respect to their roots and k, even if the tree is $(k-1)$-critical. Hence, if there exists a k-conforming layout for some U' as just defined then the construction illustrated in Figure 27 shows that a k-layout for U exists.

To show the implication in the other direction, suppose a k-layout exists for U. By Lemma 3.9, either the leftmost or the rightmost vertex in L is in one of the three vs-$(k-1)$-trees. We may then proceed via the same argument as used in Lemma 5.3. □

$q = 4$ **Lemma 5.6**

In this case, $vs(U) = k$ iff there exist two heavy trees, T_i and T_j, among the four vs-$=(k-1)$-trees such that there exists a k-conforming layout L' for $U' = U$ minus the bodies of T_i and T_j.

Proof

If there exists a k-conforming layout for some U' as just defined then the construction illustrated in Figure 27 shows that a k-layout for U exists.

To see the implication in the other direction, consider any optimal layout L for U. By Lemma 3.8, the leftmost and rightmost vertices, α and ω, in L must be in distinct vs-$(k-1)$-trees. Let T_i and T_j be the trees containing α and ω with roots r_i and r_j. Let L' be the layout obtained be removing the bodies of T_i and T_j from L. By Lemma 5.1, L' is a k-conforming layout for U'. □

$q \geq 5$

In this case U contains a subgraph which is homeomorphic to the graph in Figure 6, i.e., the unicyclic graph containing exactly 5 constituent trees, each one of which is a vs-$(k-1)$-tree of type NCb. Hence, by the Lemma 3.6, $vs(U) \geq k+1$.

6 An algorithm and time complexity

6.1 An algorithm

We can summarise the results of Sections 4 and 5 in algorithmic form. Here the meaning of the various symbols is as defined in the corresponding parts of the previous sections. The main program carries out Phase 1, as described at the beginning of Section 5, and then invokes *vs_uni(U, k)*. We assume that the vertex separation, an optimal layout and the type of each constituent tree is recorded during Phase 1.

program main;
  For all constituent trees, compute their vertex separation and type;
  For some cycle edge e chosen at random, compute the vertex separation k
  and an optimal layout L for the tree U - e and let e be a global variable;
  if vs_uni(U, k)
    then output(k, the layout created by vs_uni)
    else output($k+1$, L);

Figure 28: The Main Program

At the completion of Phase 1 we know that $k \leq \text{vs}(U) \leq k+1$. The function *vs_uni* takes as input a unicyclic graph U and the integer k, and returns true/false indicating whether or not $\text{vs}(U) \leq k$. This function summarises the results of Section 5. We assume that if the function returns true, it also returns a layout of separation $\leq k$, using the constructions defined in Section 5. Note that there is a recursive invocation and invocations of the function *vs_reduced_uni*.

Finally, the function *vs_reduced_uni* summarises the results of Section 4. It takes a unicyclic graph U, two degree 2 cycle vertices r_i and r_j and the integer k as input. It returns true/false indicating whether or not there exists a k-conforming layout for U, with respect to r_i and r_j. We further assume that, using all the constructions defined in Section 4, if the function returns true, it also returns a k-conforming layout.

The correctness of the program, if it terminates, follows from the analysis in Sections 5 and 4. To see that the program always terminates we note that *vs_uni* can invoke itself or *vs_reduced_uni*. If *vs_uni* invokes itself, it does so with a smaller graph and a reduced k. If it invokes *vs_reduced_uni*, it does so with a smaller graph. The procedure *vs_reduced_uni* only ever invokes itself, and when it does it is with a smaller value of k.

```
function vs_uni (U : unicyclic_graph, k : integer) : Boolean;

{The function assumes that vs(U − e) = k.
The cases are identical to those defined in Section 5, as also is the notation}

Case 1: {One k-critical constituent tree}
    if vs(T') = k then return(false)
    else if T_c is of type C then return(true)
        else compute vs(T_c − T[u]);
            if vs(U − T[u] − e) < k − 1 then return(true)
                else return (vs_uni(U − T[u], k − 1));

Case 2: {Three or more non-critical vs-k-trees}
    return(false);

Case 3: {Exactly two non-critical vs-k-trees}
    return (vs_reduced_uni(U', k));

Case 4: {Exactly one non-critical vs-k-trees, q = number of vs-(k − 1)-trees, not type NC}
    Case 4.1: {0 ≤ q ≤ 1}
        return(true);
    Case 4.2: {q = 2}
        for all choices of tree T_j from among the two vs-(k − 1)-trees do
            if vs_reduced_uni(U', k) {where U' = U minus the bodies of T_i and T_j}
            then return (true);
        return(false);
    Case 4.3: {q = 3}
        for all choices of two trees T_i and T_j from among the vs-k-tree and the three vs-(k − 1)-trees do
            if vs_reduced_uni(U', k) {where U' = U minus the bodies of T_i and T_j}
            then return (true);
        return(false);
    Case 4.4: {q ≥ 4}
        return(false);

Case 5: {No vs-k-trees, q = number of vs-(k − 1)-trees, not type NC}
    Case 5.1: {1 ≤ q ≤ 2} return(true);
    Case 5.2: {q = 3}
        for all choices of two trees T_i and T_j from among the three vs-(k − 1)-trees do
            if vs_reduced_uni(U', k) {where U' = U minus the bodies of T_i and T_j}
            then return (true);
        return(false);
    Case 5.3: {q = 4}
        for all choices of two trees T_i and T_j from the four (k − 1)-trees do
            if vs_reduced_uni(U', k) {where U' = U minus the bodies of T_i and T_j}
            then return (true);
        return(false);
    Case 5.4: {q ≥ 5}
        return(false);
```

Figure 29: The Unicyclic Function

function vs_reduced_uni (U, k) : Boolean;

{The cases and the notation used are identical to those defined in Section 4
We assume that $vs(T_1) \geq vs(T_2)$}

Case 1: $\{vs(T_1) = k\}$
    return(false);

Case 2: $\{vs(T_1) < k - 1\}$
    return(true);

Case 3: $\{vs(T_1) = k - 1\}$
    if $T_1[a]$ is $(k-1)$-critical and $T_1[b]$ is $(k-1)$-critical
    then if T_1 contains one constituent vs-$(k-1)$-tree, type Cb
        then return (vs_reduced_uni$(U', k-1)$);

        if T_1 contains two constituent vs-$(k-1)$-trees, type NCb
            or T_1 contains a constituent vs-$(k-1)$-tree, type C
        then return (T_2 contains no vs-$(k-1)$-trees of type other than NC);

    else {At least one of $T_1[a]$, $T_1[b]$ is not $(k-1)$-critical}
        return (T_1 and T_2 have complementary extensibilities);

Figure 30: The Reduced Graph Function

Since there are no loops and all recursive calls are on smaller graphs or smaller k, the procedures must terminate. We have omitted explicit reference to any base terminating condition, to minimise clutter, but this is easy to add. For example, "if all the constituent trees are single vertices" would suffice, since any cycle has vertex separation 2.

6.2 Time complexity

In Section 3.2 of [EST94] the number of vertices in a smallest tree with vertex separation k is given by: $m(k) = 5(3^{k-1})/2 - 1/2$. Since the vertex separation of a unicyclic graph is no more than one greater than the vertex separation of any tree obtained by removing an edge from the cycle, the vertex separation of a unicyclic graph is $O(\log n)$, where n is the number of vertices in the graph.

We have noted that *vs_reduced_uni* only ever invokes itself, and that it does so with a reduced k value. Since k is $O(\log n)$, the depth of recursion is then $O(\log n)$. The work done in *vs_reduced_uni* involves, at worst, computing the vertex separation and types of T_1, T_2, $T_1[a]$ and $T_1[b]$, i.e. the time complexity, excluding the recursive calls, is $O(n)$. Hence the time complexity of the entire procedure is $O(n \log n)$.

The work done by *vs_uni*, outside of the recursive calls or invocations to vs_reduced_uni, is $O(n)$ since it nowhere involves more than computing the vertex separation of trees together with optimal layouts.

- In Case 1 *vs_uni* can invoke itself, but only with a smaller graph and a reduced k. Hence this case can not lead to more than $O(n \log n)$ work.
- In Case 3 *vs_uni* invokes *vs_reduced_uni* on a smaller graph. We have just shown that the time complexity of *vs_reduced_uni* is $O(n \log n)$.
- In Cases 4.2, 4.3 and 5.2, 5.3 *vs_uni* invokes *vs_reduced_uni* more than once. But in all cases the the number of invocations is limited by a constant no bigger than $\binom{4}{2} = 6$. Each such invocation can lead to no more than $O(n \log n)$ work.

Consequently the entire procedure has time complexity bounded by $O(n \log n)$.

7 Conclusions

We have given an alternative formulation of a linear time algorithm to compute layouts for trees, that are optimal with respect to vertex separation. We have pointed out that optimal layouts can be "reversed" yielding another optimal layout.

Using the linear time computations and layout constructions, and also the concepts of extensibilty, we have shown that vertex separation together with optimal layouts can be computed for unicyclic graphs in $O(n \log n)$ time by an arguably "practical" algorithm. The addition of one edge to a tree seems to considerably complicate the task.

Perhaps these same tools can be applied to other families of graph with some tree-like structure to obtain useful polynomial time algorithms.

References

[BF02] H. Bodlaender and F. Fomin. Approximation of pathwidth of outerplanar graphs. *Journal of Algorithms*, 43:190–200, 2002.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.

[EST94] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Information and Computation*, 113(1):50–79, Aug 1994.

[GLY98] Rajeev Govindan, Michael A. Langston, and Xudong Yan. Approximating the pathwidth of outerplanar graphs. *Information Processing Letters*, 68(1):17–23, 1998.

[Kin92] Nancy G. Kinnersley. The vertex separation number of a graph equals its pathwidth. *Information Processing Letters*, 42(6):345–350, 1992.

[KP85] L. M. Kirousis and C. Papadimitriou. Interval graphs and searching. *Discrete Math.*, 55(2):181–184, 1985.

[KP86] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, 47(2):205–218, 1986.

[LaP93] A. LaPaugh. Recontamination does not help to search a graph. *Journal of the Association for Computing Machinery*, 40(2):224–245, 1993.

[MS88] B. Monien and I. H. Sudborough. Min Cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, 58(1-3):209–229, 1988.

[PHH+98] Peng, Ho, Hsu, Ko, and Tang. A linear-time algorithm for constructing an optimal node-search strategy of a tree. In *Annual International Conference on Computing and Combinatorics*, volume 1449 of *Lecture Notes in Computer Science*, pages 279–288. Springer, 1998.

[Sko00] K. Skodinis. Computing optimal strategies for trees in linear time. *Proceedings of the 8th Annual European Symposium on Algorithms*, pages 403–414, 2000.