A dynamic implicit adjacency labelling scheme for line graphs

David Morgan

Department of Computing Science
University of Alberta

TR05-03

January 2005

A dynamic implicit adjacency labelling scheme for line graphs

David Morgan

Department of Computing Science, University of Alberta, Edmonton AB, Canada T6G 2E8[1].

Abstract

This work is an introductory study of dynamic implicit informative labelling schemes, a topic suggested by Kannan et al. (Kannan et al., SIAM J Disc Mat, 1992) but not explored in their work. In particular, we define what is meant by a dynamic implicit informative labelling scheme and develop a dynamic implicit adjacency labelling scheme for line graphs.

1 Introduction

Consider a finite simple undirected graph $G = (V_G, E_G)$ with n vertices and m edges. Typically, we represent G using an adjacency matrix or a series of adjacency lists, labelling the vertices from 1 to n. These rudimentary labels serve only to distinguish between the vertices and do not tell us anything about the structure of G. In particular, the adjacency of any pair of vertices must be determined from the adjacency matrix or the adjacency lists, both of which are usually maintained as global resources.

What if we could determine the adjacency of two vertices of G in a more local manner, that is, by using only the labels given to them? One way to do this is by labelling each vertex with a unique prelabel from $\{1, \ldots, n\}$, along with its corresponding row of the adjacency matrix whose indices are based on these prelabels. Given this labelling scheme, we can determine the adjacency of two vertices having prelabels v_1 and v_2 using only their labels by looking up the bit corresponding to v_2 in the row of the adjacency matrix found in the label of v_1, or vice versa. In this labelling scheme each vertex has a label of size $\Theta(n)$, the sum of the sizes of all the vertex labels is $\Theta(n^2)$, and adjacency queries can be handled in $\Theta(1)$ time. Another approach is to label each vertex with a unique prelabel from $\{1, \ldots, n\}$, along with a list of the prelabels of the vertices to which it is adjacent. Given this labelling scheme, we can determine the adjacency of two vertices having prelabels v_1 and v_2 using only their labels by determining if v_2 is in the adjacency list found in the label of v_1, or vice versa. In this labelling scheme a vertex v has a label of size $\Theta(\deg(v) \log n) \subseteq O(n \log n)$, the sum of the sizes of all the vertex labels is $\Theta((m + n) \log n)$, and adjacency queries can be handled in $O(\log n)$ time providing the adjacency lists are sorted. Upon extension to families of finite graphs these labelling schemes based upon adjacency matrices and adjacency lists are examples of informative (adjacency) labelling schemes as defined by Peleg [18].

Definition 1.1 (Peleg) *Consider a function $f(S, G)$ defined on sets of vertices S of fixed but arbitrary finite graphs G. An implicit f-labelling scheme of a family $\mathcal{G}$ of finite graphs is a pair (M, D) defined as follows.*

- *M is a vertex labelling algorithm whose input is a graph G in $\mathcal{G}$. Note that M need not be deterministic; accordingly, let $\mathcal{M}_G$ be the set of all vertex labellings of V_G which can be assigned by M.*
- *D is a polynomial time deterministic evaluation algorithm whose input is a set of vertex labels. Given any labelling L_G of V_G, let $L_{S,G}$ denote the subset of these labels corresponding to a subset S of V_G. For any graph G in $\mathcal{G}$ we define L_G to be (D, f)-correct if $D(L_{S,G}) = f(S, G)$ for every subset S of V_G on which f is defined. Given this definition, we require that M_G be (D, f)-correct for all G in $\mathcal{G}$ and for all M_G in $\mathcal{M}_G$. Note that D is a function of the labels only.*

Allowing sufficiently large labels we can create informative labelling schemes for any such function f, however, in doing so we may overlook two desirable characteristics, namely, *space-optimality* and *balance*. If we define the size of a labelling of a graph to be the sum of the sizes of its vertex labels, then by a space-optimal f-labelling scheme of $\mathcal{G}$ we are referring to an f-labelling scheme which generates labellings of asymptotically smallest size over all f-labelling schemes of $\mathcal{G}$. By "asymptotically smallest size" we mean that the sizes of the graph labellings are considered asymptotically with respect to the number of vertices in the graph. If the size of a vertex labelling of a graph on n vertices is $\Theta(S)$, then by a balanced f-labelling scheme of $\mathcal{G}$ we are referring to an f-labelling scheme which generates vertex labels of size $O(\frac{S}{n})$, thus distributing the information about f across the graph. To date, balanced space-optimal informative labelling schemes have been developed for a variety of functions over certain graph classes, such as adjacency over interval graphs [12], distance over rings [17], and center of three vertices over trees [18].

[1]This research was also supported by NSERC (Natural Sciences and Engineering Research Council of Canada).

The seminal works of both Muller [15] and Kannan et al. [12] presented a narrower version of adjacency labelling schemes in the form of what Spinrad [22] defines as the the implicit representation problem. A family $\mathcal{G}$ of finite graphs is said to have an implicit representation if there is an adjacency labelling scheme for $\mathcal{G}$ such that the members on n vertices have vertex labels of size $\mathrm{O}(\log n)$. To date, implicit representations have been found for many classes of graphs including trees, bounded degree graphs, planar graphs, and interval graphs. In his recent text on graph representation, Spinrad [22] has generalized the idea of an implicit representation by asking if families of graphs with $2^{\Theta(\phi(n))}$ members on n vertices have adjacency labelling schemes using $\mathrm{O}(\frac{\phi(n)}{n})$ bits per vertex. We observe that an adjacency labelling scheme of a family of graphs provides a unique representation for each of the members of the family, so the number of bits required by an adjacency labelling scheme is at least the number of bits required to represent all of the members uniquely; in particular, a family of graphs with $2^{\Theta(\phi(n))}$ members on n vertices requires a labelling of size $\Theta(\phi(n))$ to uniquely represent each of the members on n vertices. Therefore, generalized implicit representations are balanced space-optimal adjacency labelling schemes. Note that the previously described adjacency labelling scheme devised from adjacency matrices is a generalized implicit representation, and hence a balanced space-optimal adjacency labelling scheme, for any family of graphs having $2^{\Theta(n^2)}$ members on n vertices. Such families include bipartite graphs, chordal graphs, and the class of all graphs.

The terms "generalized implicit representation" and "adjacency labelling scheme" used by Spinrad [22] and Peleg [18], respectively, do not precisely capture the essence of what they are intended to define when considered in the context of one another. The term "generalized implicit representation" directly references the ability to determine the adjacency of two vertices *implicitly* from their labels, however, put in the wider context of informative labelling schemes, it is no longer evident that adjacency is the property on which we are being informed. Moreover, the term "generalized implicit representation" is used to capture the properties of space-optimality and balance, neither of which are evident from the term itself. On the other hand, the term "adjacency labelling scheme" makes evident the underlying interest in vertex adjacency, but overlooks the fact that the adjacency of two vertices can be determined *implicitly* from their labels. As such, what is defined by Peleg to be an "adjacency labelling scheme" might more accurately be called an "implicit adjacency labelling scheme"; similarly, what Spinrad defines to be a "generalized implicit representation" might more accurately be termed a "balanced space-optimal implicit adjacency labelling scheme". Although longer, these terms better capture the properties of what is being defined and in doing so offer a unified terminology for researchers unfamiliar with the subject area. We will use this new terminology throughout the remainder of this article, except when referring to earlier works in a historical context.

In many applications the underlying topology is constantly changing and we desire algorithms which can accommodate these changes without having to process the new topology from scratch. At present, algorithms for finding implicit adjacency labelling schemes are static; that is, if the graph provided to the algorithm is changed then the algorithm must process the new graph from scratch. The dynamic version of this problem was mentioned by Kannan et al. [12] in their original work on implicit representations, however, no formulation of the problem is attempted. At most, the authors suggest that the addition or deletion of a vertex or an edge should require only a "quick" update of the labels in order to obtain an implicit representation of the new graph. To date, the paper of Brodal and Fagerberg [9] stands as the only publication on this dynamic problem. Their work handles the addition and deletion of single edges and vertices in graphs of bounded arboricity, providing the bounded arboricity is maintained. As a continuation of their work, there is a need for the development of dynamic implicit adjacency labelling schemes for more classes of graphs as well as a refinement of what is meant by a "quick" update of the labels of the vertices. Moreover, algorithms developed for dynamic implicit adjacency labelling schemes should incorporate some form of error detection; that is, the algorithms should recognize when the modified graph is no longer a member of the family under consideration. For example, Brodal and Fagerberg present an algorithm which handles an unspecified arboricity bound, however, the focus of the paper is on a more efficient algorithm which must be provided with a fixed bound on the arboricity of the graph that is being modified.

By further studying dynamic implicit adjacency labelling schemes we hope to expand the applicability of implicit labelling schemes to real world problems. In particular, implicit labelling schemes have direct applications to the efficiency of XML (Extensible Markup Language) search engines [13]. Web documents conforming to the XML standard can be viewed as a tree with nested nodes corresponding to individual words, phrases, or sections of the document. Using implicit informative labelling schemes, an XML search engine can assign labels to each of these nodes allowing relationships such as ancestor, parent, and sibling to be determined using only the labels of the nodes. This allows the search engine to answer web queries without repeatedly accessing the file itself. Moreover, by employing dynamic schemes the search engine will no longer have to recalculate the labels of the nodes when a small change is made to the XML document. Applications of implicit labelling schemes to communication networks have also been discussed in [13], [17], and [23].

2 Dynamic Implicit Adjacency Labelling Schemes

We begin by defining a dynamic implicit f-labelling scheme.

Definition 2.1 *Consider a function $f(S, G)$ defined on sets of vertices S of fixed but arbitrary finite graphs G. A dynamic implicit f-labelling scheme of a family $\mathcal{G}$ of finite graphs is a tuple (M, D, Δ, C) defined as follows.*

- *(M, D) is an implicit f-labelling scheme of $\mathcal{G}$.*
- *Δ is a set of functions which map graphs in $\mathcal{G}$ to other graphs.*
- *C is a polynomial time relabelling algorithm whose input is a pair (δ, L_G), where $\delta \in \Delta$, $G \in \mathcal{G}$, and L_G is a (D, f)-correct labelling of V_G from $\mathcal{L}_G$ (defined shortly); in particular, providing $\delta(G) \in \mathcal{G}$, C assigns a new (D, f)-correct labelling to $V_{\delta(G)}$ based upon the labelling L_G of G. Note that C need not be deterministic; accordingly, let $\mathcal{C}_{\delta,L_G}$ be the set of labellings of $V_{\delta(G)}$ which can be assigned by C on input (δ, L_G). For each G in $\mathcal{G}$ we define the family $\mathcal{L}_G$ of (D, f)-correct labellings of V_G by $L_G \in \mathcal{L}_G$ if and only if $L_G \in \mathcal{M}_G$ or there exists G^* in $\mathcal{G}$, δ in Δ, and L_{G^*} in $\mathcal{L}_{G^*}$ such that $\delta(G^*) = G$ and $L_G \in \mathcal{C}_{\delta,L_{G^*}}$.*

Moreover, we say that the dynamic implicit f-labelling scheme is error-detecting if, given any input (δ, L_G), C is able to determine when $\delta(G) \notin \mathcal{G}$.

In a less formal context, C can be considered as the composition of algorithms required by the graph operations found in Δ. For instance, if Δ permitted the addition or deletion of any edge from a graph, we might consider C to be comprised of two algorithms, InsertEdge(e,L_G) and DeleteEdge(e,L_G), which use a labelling L_G to relabel $G + e$ and $G - e$, respectively. Again, note that the algorithms InsertEdge and DeleteEdge are provided input about the graph only in the form of vertex labels; in turn, these algorithms output labellings of the vertices of $G + e$ and $G - e$, respectively. Moreover, in practice we are not interested in maintaining a labelling for every graph in the family, rather, we use the labelling of a graph to determine a labelling of a slightly modified graph, discarding the labelling of the original graph in the process. In this sense we can omit the labelling from the input of the algorithms as these algorithms are directly modifying the labelling of the graph under consideration; that is, the above algorithms might be presented as InsertEdge(e) and DeleteEdge(e).

We have seen how an implicit f-labelling scheme can be created for any function f when we allow sufficiently large labels; similarly, sufficiently weak choices of M, Δ, and C will result in a dynamic implicit f-labelling scheme for any function f. As a result, there are several ways in which one might judge the quality of a dynamic implicit f-labelling scheme. First of all, we might judge a dynamic scheme according to the time taken by C on input (δ, L_G) relative to the time taken to label $\delta(G)$ by the fastest labelling algorithm of a non-dynamic implicit f-labelling scheme. Specifically, the purpose of the dynamic scheme is to provide quick updates of the labels, thereby, if there is a non-dynamic scheme which can generate the labels in equal or better time, even from scratch, then there is no advantage gained by using the dynamic scheme. Secondly, since a dynamic implicit f-labelling scheme includes an implicit f-labelling scheme, we might also judge a dynamic scheme according to the size of the labels generated by M and C. For example, consider that the implicit adjacency labelling scheme developed using adjacency matrices can be further developed into a dynamic implicit adjacency labelling scheme. Since this dynamic scheme uses vertex labels of size $O(n)$, any other dynamic implicit adjacency labelling scheme using labels of size $\Omega(n)$ would only be advantageous if it permitted faster updates of the labels than can be achieved using the dynamic scheme developed from adjacency matrices. Finally, we might judge a dynamic scheme according to the operations contained in Δ. Preferably, Δ will contain the addition and deletion of a single edge or vertex (along with the incident edges with this vertex) which are four fundamental dynamic graph operations. Moreover, using the operations found in Δ, we would like to be able to transform any member of $\mathcal{G}$ into any other member of $\mathcal{G}$ without escaping the class $\mathcal{G}$; however, this may require more than these four fundamental dynamic graph operations.

It should be noted that if $\mathcal{G}$ is a hereditary graph class then these fundamental graph operations are sufficient to transform any member of $\mathcal{G}$ into any other member of $\mathcal{G}$ without escaping the class $\mathcal{G}$. For each member G of $\mathcal{G}$ there is a sequence $S_G = \{G_0 = \emptyset, G_1, \ldots, G_{|V_G|-1}, G_{|V_G|} = G\}$ of members of $\mathcal{G}$ for which $G_{i-1} = G_i - v_i$, where v_i is a vertex of G_i and $1 \leq i \leq |V_G|$. Thereby, given $G^{(1)}, G^{(2)} \in \mathcal{G}$ we can construct $G^{(2)}$ from $G^{(1)}$ by using the vertex deleting algorithm to transform $G^{(1)}$ into $\emptyset$ via the members of $S_{G^{(1)}}$, then using the vertex adding algorithm to transform $\emptyset$ into $G^{(2)}$ via the members of $S_{G^{(2)}}$.

Continuing with the idea of transforming one graph into another, there is a connection between error-detecting dynamic implicit f labelling schemes and the problem of recognizing whether a graph belongs to a certain family. Consider a family of graphs $\mathcal{G}$ for which there exists a dynamic implicit f-labelling scheme (M, D, Δ, C) and the recognition problem is polynomial on $\mathcal{G}$. If f allows us to determine the structure of a graph G from any labelling in $\mathcal{L}_G$, then on any input (δ, L_G) C can use f to determine the structure of G and, hence, the structure of $\delta(G)$, in polynomial time. In turn, C can apply a polynomial time recognition algorithm to determine if $\delta(G)$ is in $\mathcal{G}$; thereby, the dynamic implicit f-labelling scheme is error-detecting. For example, consider the family of line graphs which will be explored further in Section 3. Since the recognition problem has been shown polynomial for the class of line graphs [14] [19], any dynamic implicit adjacency labelling scheme for line graphs can be made error-detecting.

On the other hand, it is more complicated to develop recognition from error-detection. Consider a family of graphs $\mathcal{G}$ for which there exists an error-detecting dynamic implicit f-labelling scheme (M, D, Δ, C). If for any graph G in $\mathcal{G}$ there exists a graph G^* in $\mathcal{G}$ which can determine in polynomial time a polynomial length sequence $S_G = \{G_0 = G^*, G_1, \ldots, G_{k-1}, G_k = G\}$ of members of $\mathcal{G}$ and a polynomial length sequence $G^\Delta = \{\delta_0, \delta_1, \ldots, \delta_{k-1}\}$ of members of Δ such that $\delta_i(G_i) = G_{i+1}$, for $0 \leq i \leq k-1$ and there is a polynomial time function for determining a labelling of G^* which belongs to $\mathcal{L}_{G^*}$, then the recognition problem is polynomial on $\mathcal{G}$. The reason being that we can determine S_G, G^Δ, and the labelling for G^* in polynomial time, then transform the labelling of G^* into a labelling for G_{k-1} using a polynomial number of calls of the polynomial time algorithm C, namely $\{C_0, C_1, \ldots, C_{k-1}\}$, where $C_0 = C(\delta_0, L_{G^*})$ and $C_i = C(\delta_i, C_{i-1})$, for $1 \leq i \leq k-1$, and finally resolve the membership of G in $\mathcal{G}$ according to the action of C when it attempts to determine a labelling of G from the labelling of G_{k-1}. If $G \in \mathcal{G}$ then C will determine a labelling of G, otherwise, it will output that $G \notin \mathcal{G}$ since it is an error-detecting algorithm. For example, consider a hereditary graph class with an error-detecting dynamic implicit f-labelling scheme whose graph operation set includes the addition of vertices (along with incident edges). For each member G of the class there is a sequence $S_G = \{G_0 = \emptyset, G_1, \ldots, G_{|V_G|-1}, G_{|V_G|} = G\}$ of members of $\mathcal{G}$ for which $G_{i-1} = G_i - v_i$, where v_i is a vertex of G_i and $1 \leq i \leq |V_G|$. We have a polynomial time labelling for $\emptyset$ and a means to achieve $G_{|V_G|-1}$ from $\emptyset$ using operations in Δ, thereby, the recognition problem is polynomial for the hereditary class.

Given that the algorithms which change the labellings are functions of the change and the labelling only, the vertex labels used in dynamic implicit labelling schemes must contain sufficient information to allow algorithms to update the labellings. In general, the labels used in implicit adjacency labelling schemes do not contain enough information to be used in dynamic implicit adjacency labelling schemes, however, the implicit adjacency labelling schemes of some classes are inherently dynamic.

For instance, consider the following implicit adjacency labelling scheme for trees. Let T be a tree on n vertices. We arbitrarily assign to T a root and give each vertex a unique prelabel from $\{1, \ldots, n\}$. We now obtain an implicit adjacency labelling scheme for T by giving each vertex v of T the label (prelabel(v), prelabel(parent(v))). The adjacency of two vertices v_1 and v_2 can be determined using only their labels by a polynomial time algorithm D which checks if prelabel(v_1) = prelabel(parent(v_2)) or prelabel(v_2) = prelabel(parent(v_1)). Moreover, each label is of size $\mathrm{O}(\log n)$, thus making the scheme space-optimal and balanced as the number of trees on n vertices is $2^{\Theta(n \log n)}$. If a new vertex is added to a tree such that the resulting graph is still a tree, then its only neighbour is its parent. Therefore, we can give it the label (prelabel, prelabel of parent) so that the labelling is still (D,*adjacency*)-correct on the new graph. If a vertex is deleted such that the resulting graph is still a tree, then the vertex must have been a pendant vertex. Therefore, it was not the parent of any other vertex and so deleting the vertex keeps the labelling (D,*adjacency*)-correct on the remaining tree.

Although the maintenance of this space-optimal implicit adjacency labelling scheme for trees seems straightforward, there are some underlying shortcomings. One such problem is that when a vertex is added and given a prelabel there must be some way of determining an acceptably small unused prelabel to assign to it. Another such problem is that it is possible to delete too many vertices causing the remaining prelabels to ruin the space-optimality of the labelling. In the work on line graphs presented in Section 3 we make assumptions which eliminate these problems.

As mentioned, the only work on dynamic implicit adjacency labelling schemes is by Brodal and Fagerberg [9] who develop such schemes for graphs of bounded arboricity. Fundamental to their work is the relationship between arboricity and outdegree orientations where, in particular, a graph with arboricity c has an outdegree-c orientation. Their algorithm keeps an outneighbourhood list for each vertex v, denoted by adj[v], however, it also includes a mechanism to handle outdegree lists which get too big. On a graph with n vertices and arboricity bounded by c Brodal and Fagerberg's representation supports adjacency testing in $\mathrm{O}(c)$ time, edge insertions in $\mathrm{O}(1)$ time, and edge deletions in $\mathrm{O}(c + \log n)$ time. We present their algorithms for handling the addition and deletion of a single edge from a graph of bounded arboricity c in Figure 1. Unfortunately, these algorithms are built on the assumption that the changes to the graph do not cause its arboricity to exceed c. In their article, Brodal and Fagerberg do describe modified algorithms which can handle unspecified arboricities, however, this results in increased time bounds.

```
INSERT$((u, v))$
 1  $\text{adj}[u] \leftarrow \text{adj}[u] \cup \{v\}$
 2  if $|\text{adj}[u]| = 4c + 1$
 3     then $S \leftarrow \{u\}$
 4            while $S \neq 0$
 5            do $w \leftarrow \mathsf{Pop}(S)$
 6                for $x \in \text{adj}[w]$
 7                do $\text{adj}[x] \leftarrow \text{adj}[x] \cup \{w\}$
 8                    if $|\text{adj}[x]| = 4c + 1$
 9                       then $\mathsf{Push}(S, x)$
10                $\text{adj}[w] \leftarrow \emptyset$

DELETE$((u, v))$
 1  $\text{adj}[u] \leftarrow \text{adj}[u] \setminus \{v\}$
 2  $\text{adj}[v] \leftarrow \text{adj}[v] \setminus \{u\}$
```

Figure 1: Algorithms for dynamic implicit representations of graphs of bounded arboricity c.

3 Error-Detecting Dynamic Implicit Adjacency Labelling Schemes for Line Graphs

In the remainder of this work we consider error-detecting dynamic implicit adjacency labelling schemes for line graphs which are defined as follows [6].

Definition 3.1 *Given a graph $G = (V, E)$, its line graph is the graph $L(G) = (E, E')$ for which $\{u, v\} \in E'$ if and only if u and v are incident edges in G.*

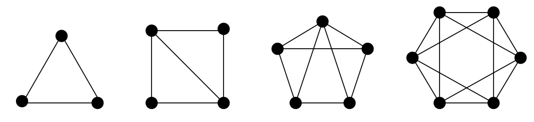
We observe that by adding isolated vertices we can obtain infinitely many graphs which give rise to the same line graph. As such, if a graph G has no isolated vertices we will refer to it as a base of $L(G)$. Just as an unlabelled graph "generates" an unlabelled line graph, we can say that an edge labelled graph "generates" a vertex labelled line graph. As such, the term base will also be used to refer to an edge labelled graph, with no isolated vertices, which "generates" a particular vertex labelled line graph.

In [24] Whitney has shown that every connected unlabelled line graph has a unique base, up to isomorphism, except for K_3 which has two bases, namely, K_3 and $K_{1,3}$. Our work on line graphs requires a concept similar to that of isomorphism, but involving edge labellings. Given an edge labelling ψ of a graph, for each edge label α let P^{ψ}_{α} denote the partition of the labels incident with α that is determined by endpoints of α. We define two bases of a vertex labelled line graph $L(G)$, having edge labellings ψ_1 and ψ_2, to be *partition isomorphic* if $P^{\psi_1}_{\alpha} = P^{\psi_2}_{\alpha}$, for all edge labels α. For example, consider the two bases shown in Figure 2(d); in one of these bases the partition corresponding to a is $\{\{b\}, \{c\}\}$, while in the other it is $\{\{b, c\}, \emptyset\}$. Therefore, these bases are not partition isomorphic. When we consider the above theorem of Whitney in the context of labelled line graphs we arrive at the following theorem which will be useful when we add a vertex to a line graph.

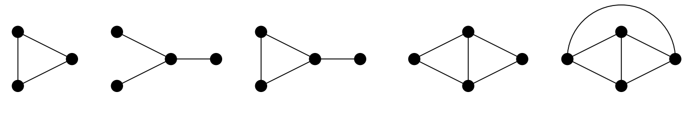
Theorem 3.2 *Every vertex labelled connected line graph, except for K_3 and the three line graphs shown in Figure 2(a), has a unique (edge labelled) base, up to partition isomorphism. For each of the four exceptions, a vertex labelled graph has two bases that are not partition isomorphic.*

Proof. Consider a vertex labelled connected line graph $L(G)$ which has two (edge labelled) bases, G_1 and G_2, that are not partition isomorphic. Fundamental to this proof is the fact that the edge incidences in G_1 and G_2 are identical; that is, two edges are incident in G_1 if and only if they are incident in G_2.

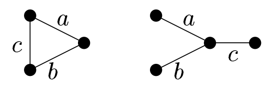
If G_1 is not isomorphic to G_2, then $L(G)$ is K_3. Therefore, it remains only to consider when G_1 and G_2 are isomorphic but not partition isomorphic. Let the labellings of G_1 and G_2 be ψ_1 and ψ_2, respectively, and let a



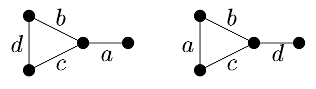
(a) The only connected line graphs with two (edge labelled) bases that are not partition isomorphic.



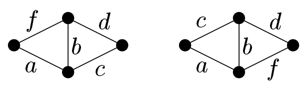
(b) The base graphs of the four line graphs pictured in Figure 2(a).



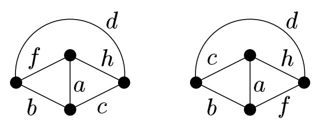
(c) Two edge labelled bases of a vertex labelled graph which are not partition isomorphic.



(d) Two edge labelled bases of a vertex labelled graph which are not partition isomorphic.



(e) Two edge labelled bases of a vertex labelled graph which are not partition isomorphic.



(f) Two edge labelled bases of a vertex labelled graph which are not partition isomorphic.

Figure 2: Partition isomorphism of graphs.

be a label for which $P_a^{\psi_1} \neq P_a^{\psi_2}$. Moreover, let $P_a^{\psi_1} = \{Q_{\psi_1}, R_{\psi_1}\}$ and $P_a^{\psi_2} = \{Q_{\psi_2}, R_{\psi_2}\}$. Trivially, observe that $|Q_{\psi_1}| + |R_{\psi_1}| \geq 2$, otherwise $P_a^{\psi_1} = P_a^{\psi_2}$.

Now consider when one of $|Q_{\psi_1}|$, $|R_{\psi_1}|$, $|Q_{\psi_2}|$, or $|R_{\psi_2}|$ is at least three; without loss of generality, let $b, c, d \in Q_{\psi_1}$. We first consider the case when $\{b, c, d\} \subseteq Q_{\psi_2}$ or $\{b, c, d\} \subseteq R_{\psi_2}$; without loss of generality, assume the former. Since $P_a^{\psi_1} \neq P_a^{\psi_2}$, there must be a label e in Q_{ψ_1} but not in Q_{ψ_2}, or vice versa; again, without loss of generality, assume the former. Given that $e \in Q_{\psi_1}$, e is incident with each of b, c, and d in both G_1 and G_2. Yet $e \in R_{\psi_2}$, so G_2 must contain each of the three cycles of edges abe, ace, and ade, which is not possible unless $b = c = d$. Next, consider the case when $\{b, c, d\} \not\subseteq Q_{\psi_2}$ and $\{b, c, d\} \not\subseteq R_{\psi_2}$; without loss of generality, assume that $\{b, c\} \subseteq Q_{\psi_2}$ and $\{d\} \subseteq R_{\psi_2}$. A similar argument gives that G_2 must contain both of the three cycles of edges abd and acd, which is not possible unless $b = c$.

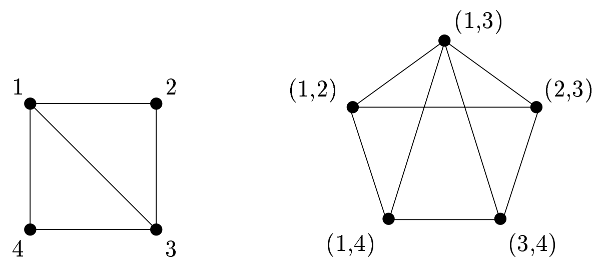
Having shown that neither Q_{ψ_1}, R_{ψ_1}, Q_{ψ_2}, nor R_{ψ_2} can contain three edges, we observe that the only way $P_a^{\psi_1} \neq P_a^{\psi_2}$ is if, without loss of generality, there exist edges b and c such that $\{b, c\} = Q_{\psi_1}$, $b \in Q_{\psi_2}$, and $c \in R_{\psi_2}$. Since b and c are adjacent in G_1, they are also adjacent in G_2; as such, the edges a, b, and c form a $K_{1,3}$ in G_1 and a K_3 in G_2.

Since the edges a, b, and c form a K_3 in G_2, each edge in R_{ψ_1} must be incident with exactly one of b or c. If R_{ψ_1} contains two edges, f and h, both of which are, without loss of generality, in Q_{ψ_2}, then both of these edges are incident with b. As such, G_1 contains both of the cycles of edge abf and abh, which is not possible unless $f = h$.

Finally, without loss of generality, consider if there exists some edge d incident with b but not a. Since $b \in Q_{\psi_2}$ and G_2 contains the three cycle of edges abc, d is incident with c. Therefore, G_1 contains the three cycle of edges bdc.

Given the constraints we have developed, G must be isomorphic to one of the graphs pictured in Figure 2(b). Therefore, $L(G)$ must be isomorphic to one of the graphs pictured in Figure 2(a). □

In [15], Muller presents the following balanced space-optimal implicit adjacency labelling scheme for the class of line graphs. Consider a line graph $L(G)$, with base graph G. To each vertex in G we assign a unique prelabel from $\{1, \ldots, |V_G|\}$. The implicit adjacency labelling scheme labels each vertex v of $L(G)$ as (e_1, e_2), where e_1 and e_2 are the prelabels of the endpoints of the edge of G corresponding to v. The evaluation algorithm can determine the adjacency of two vertices in polynomial time using only their labels by checking if the labels share a common entry, which happens if and only if the corresponding edges are incident in the base. Since G has no isolated vertices, $|V_G| \leq 2|E_G| = 2|V_{L(G)}|$, so each label is of size $\mathrm{O}(\log |V_G|) = \mathrm{O}(\log |V_{L(G)}|)$. Since there are $2^{\Theta(n \log n)}$ line graphs on n vertices, this labelling scheme is space-optimal and balanced. Presumably, the labelling algorithm, M, knows the structure of the base graph as well as the correspondence between edges in the base and vertices in the line graph; otherwise, it can use an algorithm like those found in [14] or [19]. An example of a line graph with such an implicit adjacency labelling is given in Figure 3.



(a) G.　　(b) Labels of $L(G)$.

Figure 3: An implicit adjacency labelling of a line graph.

Given a line graph $L(G)$, our dynamic scheme will assign each vertex a unique prelabel *pre* from $\{1, \ldots, |V_{L(G)}|\}$. Having assigned this prelabelling we select a edge labelled base G of the prelabelled graph from which we will derive the remainder of our labelling. Like the implicit adjacency labelling scheme of Muller [15], we assign each vertex of G a unique prelabel from $\{1, \ldots, |V_G|\}$, however, we label each vertex of $L(G)$ with the following information.

$pre.ep_0, pre.ep_1$: The edge of G corresponding to *pre* has two endpoints; $pre.ep_0$ and $pre.ep_1$ are the prelabels of these endpoints.

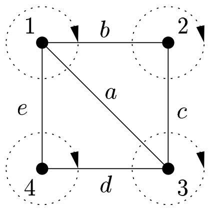
$pre.nn_0, pre.nn_1$: The values of $|N(pre.ep_0)|$ and $|N(pre.ep_1)|$ (in G), respectively, where $N(x)$ denotes the open neighbourhood of the vertex x.

$pre.nx_0, pre.nx_1$: Using the vertex labels we maintain a circular (singly) linked list of the edges incident with each vertex of G. With *pre* as the present edge, $pre.nx_i$, $i \in \{0, 1\}$, is the prelabel of the next edge in the circular linked list about $pre.ep_i$. These circular linked lists allow the information about the neighbourhood of a vertex in the line graph to be distributed over the labels of the neighbours themselves. This information is sufficient to allow both depth first and breadth first searches [6] on both G and $L(G)$.

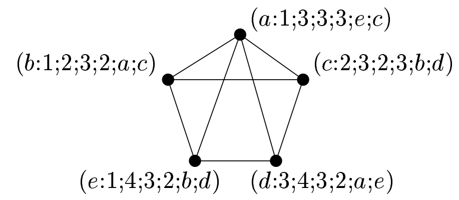
In particular, the label of a vertex with prelabel *pre* is (pre: $pre.ep_0$; $pre.ep_1$; $pre.nn_0$; $pre.nn_1$; $pre.nx_0$; $pre.nx_1$). As an example of this labelling consider the line graph presented in Figure 4. Observe that, the circular linked lists about the endpoints of an edge e of G uniquely partition the edges of G that are incident with e. Consequently, for any given graph labelling produced in our dynamic scheme, there is a unique edge labelled graph, up to partition isomorphism, from which our labelling is derived; in this sense we will refer to edge labelled base G as the *core* of our labelled graph. To compare, the core of a line graph labelled in our dynamic scheme is unique, up to partition isomorphism; however, an edge labelled base determined from the prelabelling of the vertices of the line graph may not be.

As with Muller's implicit adjacency labelling scheme for line graphs presented previously, we presume that the labelling algorithm, M, knows the structure of a base graph as well as the correspondence between edges in this base and vertices in the line graph; again, if this is not known then we may use an algorithm like those found in [14] or [19]. As well, given the labels of two vertices v_1 and v_2 of the line graph an evaluation algorithm can determine their adjacency in polynomial time by checking if $\{v_1.ep_0, v_1.ep_1\} \cap \{v_2.ep_0, v_2.ep_1\} = \emptyset$.

The success of our dynamic scheme lies in the ability to change the labelling of a graph to reflect a new core. In particular, if a vertex prelabelled line graph has a connected component with two edge labelled bases that are not partition isomorphic, it is possible that the selection of one of these bases as the core will permit certain dynamic operations while the other will not. For instance, consider the two bases depicted in Figure 2(d). If we wish to add a new vertex v to the corresponding line graph such that its neighbours are a and b, the equivalent operation in the base is the addition of an edge which is incident only with the edges labelled a and b. This can be done using one of the bases, but not the other. Again, we note that in any implicit informative labelling



(a) G (↻ denotes the order of the circular linked list).



(b) Labels of $L(G)$.

Figure 4: An error-detecting space-optimal dynamic implicit adjacency labelling of a line graph.

scheme we have access to the vertex labels only; as such, the core is an artificial construct which help us interpret and manipulate the labels. Consequently, when we say that we change the core of a line graph we ultimately mean that we change the labelling of the graph so as to reflect a new core.

Even more critical to the success of our dynamic scheme is the inclusion of sufficient information in the labels to deduce, at least partially, the structure of the core. Upon modification of the line graph, our knowledge of the original core will allow us to determine the core of the new line graph and, in turn, the labels of the new line graph. To illustrate this need for knowledge about the core, consider the line graphs presented in Figure 5. Even though the line graph $\delta(L(G))$ is formed by deleting a single edge from $L(G)$, the change in the core, from G to G' is substantial; in particular, the required change affects much more than just the edges of the core which correspond to the endpoints of the deleted edge in the line graph. If the label of a vertex v were to use the labels of the non-dynamic scheme of Muller [15], then it would be impossible to deduce the neighbourhood of v without checking the label of every vertex u in the graph to see if the edges of the base corresponding to u and v share a common endpoint. For simplicity we will hereafter refer to $\delta(L(G))$ as $L(G')$, the line graph with base G', however, we implore the reader to recognize that it is the graph $L(G)$ to which the operation δ is being applied; we are not applying δ to G to get G', rather, G' is the resulting base graph of $L(G') = \delta(L(G))$.

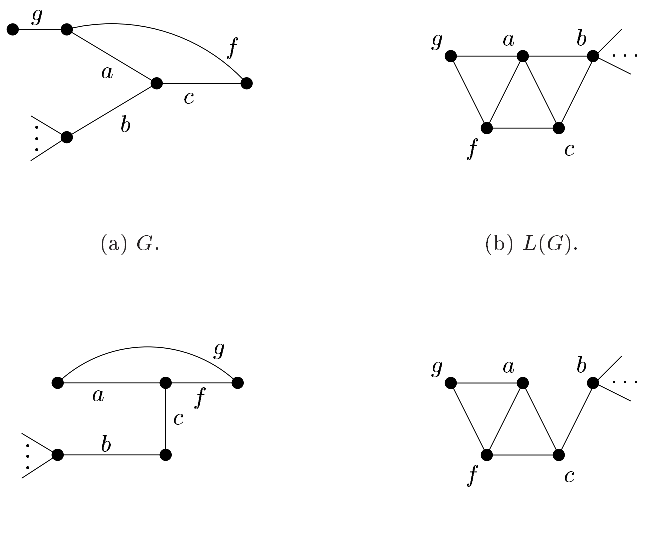


Figure 5: Deleting an edge from a line graph.

Consider a line graph which, through some series of modifications, now contains exactly n vertices. If $l(\mathit{string})$ is used to denote the size of the representation of *string* then the size of a label is

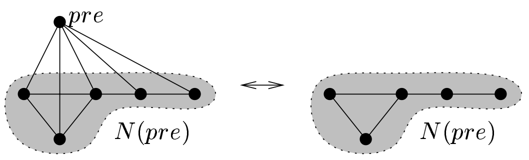
$$\begin{aligned} & \mathrm{O}(l(\mathit{pre}) + l(\mathit{pre.ep_0}) + l(\mathit{pre.ep_1}) + l(\mathit{pre.nn_0}) \\ & + l(\mathit{pre.nn_1}) + l(\mathit{pre.nx_0}) + l(\mathit{pre.nx_1})) \\ = \; & \mathrm{O}(l(\mathit{pre}) + l(\mathit{pre.ep_0}) + l(\mathit{pre.nn_0}) + l(\mathit{pre.nx_0})). \end{aligned}$$

In an earlier discussion on trees we have observed that if the deletion of vertices is permitted, it may result in the prelabels of the remaining vertices not being space-optimal. The same is true for any labelling in which prelabels are assigned. To this effect, let the largest prelabel of a vertex in the line graph be L and let the largest prelabel of a vertex in the core be B; thereby, $l(\mathit{pre}), l(\mathit{pre.nx_0}) \in \mathrm{O}(\log L)$ and $l(\mathit{pre.ep_0}) \in \mathrm{O}(\log B)$. Additionally, since the core has no isolated vertices, $|V_G| \leq 2|E_G| = 2|V_{L(G)}| = 2n$, so $l(\mathit{pre.nn_0}) \in \mathrm{O}(\log n)$ and the size of a vertex label is $\mathrm{O}(\log L + \log B + \log n)$. Providing $L, B \in \mathrm{O}(n)$, the size of a vertex label reduces to $\mathrm{O}(\log n)$, making the vertex labels space-optimal. We will assume that $L, B \in \mathrm{O}(n)$ hereafter.

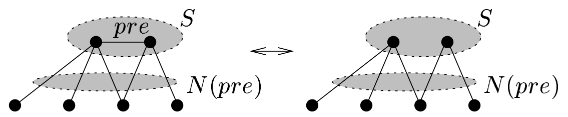
In the remainder of this work we discuss the graph operations included in our dynamic implicit adjacency labelling scheme for line graphs. In particular, the permitted operations are the deletion of a vertex (along with its incident edges), the addition of a vertex (along with some incident edges), the deletion of an edge, and the addition of an edge. Since the class of line graphs is hereditary, these four graph operations are sufficient to transform any line graph into any other line graph, without escaping the class of line graphs. It should be noted that we will only present algorithms to deal with the deletion of a vertex in the body of this work; the algorithms for the other operations will be contained in an appendix due to their length.

3.1 Deleting a vertex from the line graph

One change we can make to a line graph is to delete a vertex along with its incident edges. The resulting change in the core is the deletion of the corresponding edge as depicted in Figure 6. The algorithm DeleteVertex presented in Figure 7 can be used to maintain the vertex labels in this situation. Letting *pre* be the prelabel of the vertex of the line graph to be deleted, DeleteVertex traverses the circular linked lists at each of the endpoints of the edge corresponding to *pre* so as to decrement by one the the number of edges incident with these endpoints, then removes *pre* from the circular linked lists of its endpoints and frees the prelabel *pre* for future use. DeleteVertex runs in $\mathrm{O}(|N(\mathit{pre})|)$ time, which is $\mathrm{O}(1)$ per edge deleted from the line graph. We observe that any such algorithm is error-detecting as any induced subgraph of a line graph is a line graph.



(a) A vertex is deleted from (or added to) $L(G)$



(b) The corresponding change in G

Figure 6: A vertex is deleted from (or added to) the line graph.

DeleteVertex:

Input: *pre*, the prelabel of a vertex to be deleted from the line graph.

Output: None, however, DeleteVertex changes the labels of the present graph in order to produce a labelling of the modified graph.

1: The counter i is used for the endpoints of *pre* in the core. We must process both of these endpoints.

2,3: If $\mathit{pre.ep_i}$ is incident only with *pre*, then it will become an isolated vertex once *pre* is deleted. The function FreeCore frees the prelabel of $\mathit{pre.ep_i}$ for future use.

4-8: If $\mathit{pre.ep_i}$ is incident with more than just *pre*, then we must update the information about $\mathit{pre.ep_i}$ contained at each of these edges. We use the circular linked list structure to do this. The variable *end* tells us which

```
DeleteVertex($pre$)
1   for $i \leftarrow 0$ to 1 do
2       if $pre.nn_i = 1$ then
3           FreeCore($pre.ep_i$)
4       else $I \leftarrow$ GetIncidentNeighbors($pre, i$)
5           $(last, j) \leftarrow$ Pop($I$)
6           $last.nx_j \leftarrow pre.nx_i$
7           Push($I, (last, j)$)
8           DecrementNN($I$)
9   FreeLine($pre$)

FreeCore($corepre$)
1   free $corepre$ for future use as a prelabel of a vertex in the core

GetIncidentNeighbours($start, endpt$)
1   $S \leftarrow$ Nil
2   $u \leftarrow start$
3   $end \leftarrow endpt$
4   Push($S, (u, end)$)
5   while $u.nx_{end} \neq start$ do
6       $u \leftarrow u.nx_{end}$
7       $end \leftarrow$ FindEnd($u, start.ep_{endpt}$)
8       Push($S, (u, end)$)
9   return $S$

FindEnd($edge, vert$)
1   if $edge.ep_0 = vert$ then
2       $num \leftarrow 0$
3   else $num \leftarrow 1$
4   return $num$

DecrementNN($S$)
1   while $S \neq$ Nil do
2       $(u, end) \leftarrow$ Pop($S$)
3       $u.nn_{end} \leftarrow u.nn_{end} - 1$

FreeLine($linepre$)
1   free $linepre$ for future use as a prelabel of a vertex in the line graph
```

Figure 7: The algorithm DeleteVertex associated with the deletion of a vertex from a line graph. FindEnd is a function called by DeleteVertex; it has been isolated as it is also called by the algorithms associated with the other graph operations.

endpoint of the present edge in the circular linked list is $pre.ep_i$ and, as such, end has value 0 or 1. This value is determined by FindEnd, a function which will also be used in our other algorithms.

9: For each edge u in the circular linked list at $pre.ep_i$ we must decrement the number of edges incident with $u.ep_{end}$ by one.

10: At this point u is the edge preceding pre in the circular linked list for $pre.ep_i$. We eliminate pre from the circular linked list of $pre.ep_i$ by letting $u.nx_{end}$ be $pre.nx_i$.

11: Once the circular linked list has been processed we delete pre using FreeLine which frees its prelabel for future use.

FindEnd:

Input: $edge$ and $vert$. We wish to know which endpoint of $edge$ is $vert$.

Output: num, the endpoint of $edge$ which is $vert$.

1-4: If the "0" endpoint of $edge$ is $vert$ we return 0, otherwise, the "1" endpoint of $edge$ is $vert$ so we return 1.

3.2 Adding a vertex to the line graph

Adding a vertex, along with its incident edges, to the line graph is the same as adding an edge to the core; this is depicted in Figure 6. In this situation we use the algorithm AddVertex found in Appendix A in order to maintain the vertex labels. It should be noted that in adding a vertex with prelabel *pre* to the line graph we must specify the set of prelabels of neighbours of *pre*.

The endpoints of the edge in the core corresponding to *pre* must not only cover all of the edges of the core corresponding to the members of $N(\mathit{pre})$, but, they must be incident only with edges corresponding to vertices in $N(\mathit{pre})$. If $N(\mathit{pre}) = \emptyset$, then AddVertex creates two new vertices in the core and puts the edge corresponding to *pre* between them. If $N(\mathit{pre}) \neq \emptyset$ then we are looking for a set S of vertices in the core for which each of the following conditions hold.

- $1 \leq |S| \leq 2$.
- each edge of the core corresponding to a vertex of $N(\mathit{pre})$ has exactly one endpoint in S.
- no edge of the core corresponding to a vertex not in $N(\mathit{pre})$ has an endpoint in S.

We will call such a set *valid* and illustrate this concept in Figure 6.

To find a valid set, AddVertex relies on a subroutine called FindValid. First, FindValid selects an edge of the core, $\mathit{edge_0}$, from $N(\mathit{pre})$ and chooses endpoint 0 of $\mathit{edge_0}$ to be in the valid set. Letting R be the set of edges in $N(\mathit{pre})$ not incident on $\mathit{edge_0.ep_0}$, we observe that if there is another vertex in the valid set then it must come from an edge in R. We initially set R to $N(\mathit{pre})$, then use the circular linked list structure at $\mathit{edge_0.ep_0}$ to eliminate edges from R. If at any point we find a vertex which does not belong to $N(\mathit{pre})$ then $\mathit{edge_0.ep_0}$ cannot belong to the valid set, so we backtrack and try endpoint 1. If endpoint 1 is similarly problematic then we cannot find a valid set in the core. However, before concluding that the vertex cannot be added to the line graph we must determine if the graph has another base which is not partition isomorphic; in particular, the algorithm ChangeCore is used to change the core of the component containing the vertex corresponding to $\mathit{edge_0}$. If this component has another base then we must repeat our efforts on $\mathit{edge_0}$ using the new core.

Providing that $\mathit{edge_0}$ does not prevent us from finding a valid set, FindValid now selects an edge, $\mathit{edge_1}$, from R and attempts to include endpoint 0 of $\mathit{edge_1}$ in the valid set. Letting R_1 be the set of edges in R not incident on $\mathit{edge_1.ep_0}$, we observe that R_1 must be $\emptyset$ in order to have a valid set. We initially set R_1 to R, then use the circular linked list structure at $\mathit{edge_1.ep_0}$ to eliminate edges from R_1. If at any point we find a vertex which does not belong to R, or if R_1 does not finish as $\emptyset$, then $\mathit{edge_1.ep_0}$ cannot belong to the valid set, so we backtrack just as we did with $\mathit{edge_0}$. In this manner, FindValid exhausts all possible combinations of bases and endpoints in its attempt to find a valid set, where, in particular, backtracking is accomplished by first trying a new endpoint and then, if necessary, by changing the core.

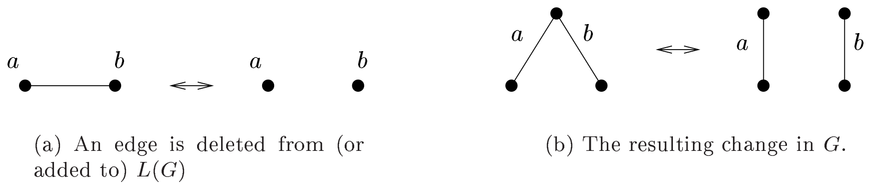
If a valid set is found then we use the members of the valid set to add the edge corresponding to *pre* to the core. If the valid set contains two vertices then these become the endpoints of the edge corresponding to *pre*; if it contains only one vertex then AddVertex makes the edge of the core corresponding to *pre* incident with this vertex as well as a new vertex which it creates. Regardless of the size of the valid set, AddVertex concludes by inserting *pre* into the circular linked lists of its endpoints, then updating the labels of the vertices in these circular linked lists by incrementing the number of edges incident with these endpoints.

From Theorem 3.2 we see that each component of the line graph has at most two bases which are not partition isomorphic; moreover, if a component has two bases which are not partition isomorphic then the size of the component is O(1). Therefore, FindValid requires only O(1) backtracks, where each core change takes O(1) time, so the running time of FindValid is dominated by the time taken to traverse the circular linked lists when eliminating edges from R and R_1. Consequently, the running time of FindValid is $O(|N(\mathit{pre})|)$ time and, as such, AddVertex runs in $O(|N(\mathit{pre})|)$ time as well.

3.3 Deleting an edge from the line graph

Consider the act of deleting an edge from a line graph as depicted in Figure 8. This is equivalent to "pulling apart" two incident edges in the core. If there are additional edges incident with w, the vertex of the core at which these two incident edges were joined, then it becomes increasingly difficult to determine the new core. Fortunately, there are a finite number of cases to be considered; we enumerate these cases as a corollary of the theorem that follows.

Theorem 3.3 *Let $L(G)$ and $L(G')$ be line graphs where $L(G') = L(G) - e$, $e = \{a, b\}$. Moreover, let the edges of G corresponding to a and b be $\{w, x\}$ and $\{w, y\}$, respectively. We make the following observations on the graph G.*



(a) An edge is deleted from (or added to) $L(G)$

(b) The resulting change in G.

Figure 8: An edge is deleted from (or added to) a line graph.

1. *The degree of w is at most four.*
2. *If $d = \{x, y\}$ is an edge of G, then $deg(w) \leq 3$.*
3. *Let $c = \{w, z\}$ be an edge of G, where $c \neq a, b$. If $f = \{z, t\}$ is an edge of G, $f \neq c$, then either $t = x$ or $t = y$.*
4. *If $d = \{x, y\}$ and $c = \{w, z\}$ are edges of G, where $c \neq a, b$, then there can be no edges incident with c other than a and b.*
5. *If $c = \{w, z\}$ and $f = \{z, x\}$ are edges of G, where $c \neq a, b$, then $deg(x) \leq 3$. Moreover, if $deg(x) = 3$ then x is incident with an edge $g = \{x, p\}$, where $p \neq w, z, y$, such that $deg(p) \leq 2$; if $deg(p) = 2$ then the edge $i = \{p, w\}$ belongs to G.*
6. *Let $c = \{w, z\}$ be an edge of G, where $c \neq a, b$. If there exist distinct edges $f = \{z, t_1\}$, $h = \{z, t_2\}$, where $c \neq f, h$, then $\{t_1, t_2\} = \{x, y\}$. Moreover, $deg(x) = deg(y) = 2$.*
7. *If H is a subgraph of G then $L(H) - e$ is a line graph.*

Proof. Fundamental to the proofs of each of these observations is the additional observation that the only edge incidence that changes from G to G' is that of a and b which are incident in G but not in G'. To aid in the visualization of these proofs the reader is encouraged to consult the diagrams in Table 1. It should be noted that, although there is a direct correspondence between edges of G and G', the same correspondence cannot be made between the vertices as it is only the edge incidences which are important, not the specific vertices at which edges are incident. Consequently, any references to vertices in the following arguments will be in the context of the graph G.

1. Assume that w is incident with at least five edges in G, say a, b, c, i, and j. Now c, i, and j must be incident with both a and b in G' because they had been so in G. However, in a simple graph, any set of three edges between two disjoint edges, such as a and b in G', must also contain two disjoint edges. Without loss of generality, let these disjoint edges be c and j. This implies c and j were not incident in G, which is a contradiction, as they were both incident with w.

2. Assume that w is incident with at least four edges in G, say a, b, c, and i. Now d must be incident with both a and b in G' because it had been so in G. Yet a and b are disjoint in G', so G' must contain the path a, d, b of edges. Similarly, c must be incident with both a and b in G' because it had been so in G. Yet a and b are disjoint in G', so G' must contain the path a, c, b of edges. Additionally, c must be disjoint from d in G' as it had been so in G; thereby, G' must contain the four cycle a, d, b, c of edges. However, G' must contain the four cycle a, d, b, i of edges as the arguments made for c can also be made for i. Thereby, $c = i$, which is a contradiction.

3. Using an argument identical to that for the edge d in (2), G' must contain the path a, c, b of edges. Yet f must be incident with c in G' because it had been so in G. Therefore, f is incident with at least one of a or b in both G' and G. Without loss of generality, let f be incident with a. If $t = w$, then $f = c$; therefore, $t = x$ as desired. This scenario is depicted in case E of Table 1.

4. By (2) the only edges incident with c at w (in G) are a and b, so it remains to show that $deg(z) = 1$. Assume that there is another edge incident with c at z, namely $f = \{z, t\}$. By (3), $t \in \{x, y\}$, so without loss of generality let $t = x$. Using the argument found in (2) we know that G' contains the four cycle a, d, b, c of edges as shown in case B of Table 1. However, G' must contain the four cycle f, d, b, c of edges as the arguments made for a can also be made for f. Thereby, $f = a$, which is a contradiction.

5. Consider when $deg(x) > 2$, that is, there exists some edge $g = \{x, p\}$ for which $p \notin \{w, z\}$ (i.e. $g \neq f, a$). Since $p \notin \{w, z\}$, g is not incident with c in G, nor in G'. Yet g must be incident with both a and f in G' as it had been so in G, thereby, g must exist in G' as shown in case F of Table 1. Since g is required to be a specific edge of G' we get $deg(x) \leq 3$. Moreover, from (4) we observe that the edge $\{x, y\}$ cannot exist in G, so $p \neq y$ as desired.

   Finally, consider when $deg(p) > 1$, that is, there exists some edge $i = \{p, s\}$ for which $i \neq g$. Now i must be incident with g in G' as it had been so in G. But G' contains the three cycle g, a, f of edges as depicted in case F of Table 1, so i must be incident with either a or f in G' and, subsequently, in G. If i is incident with f in G then $i \neq g$ gives $s = z$, so i is incident with c and f but not b in G and, subsequently, in G'. However, i is also incident with g in G', thereby forcing $i = a$ as can be seen in case F of Table 1, which is a contradiction. On the other hand, if i is incident with a in G then $i \neq g$ gives that $s = w$ as seen in case J of Table 1. Since i is required to be the specific edge $\{p, w\}$ we get $deg(p) \leq 2$.

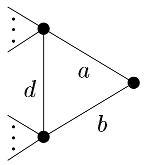
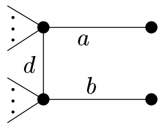
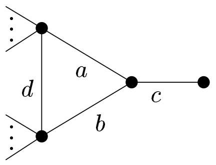
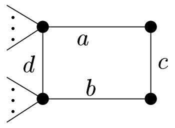
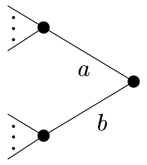
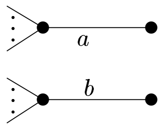
6. The first part of this statement follows directly from (3); this in turn gives that $deg(x), deg(y) \geq 2$. Without loss of generality, let $f = \{z, x\}$ and let $h = \{z, y\}$. Now f is incident with a and h, but not b, in G' as it had been so in G; similarly, h must be incident with b and f, but not a, in G' as it had been so in G. Thereby, G' must contain the path a, f, h, b of edges. As well, c must be incident with each of a, b, f, and h as it had been so in G, so G' is as depicted in case G of Table 1. Note, in particular, that the edges c, f, and h form a triangle in G'.

   Assume that $deg(x) > 2$, that is, x is incident with some edge $l = \{x, q\}$ for which $q \notin \{w, z\}$ (i.e. $l \neq a, f$). Now $l \notin \{w, z\}$ gives that l is not incident with c in G and, subsequently, in G'. As well, from (4) we observe that the edge $\{x, y\}$ cannot exist in G, so $q \neq y$ which combined with $q \neq z$ gives that l is not incident with h in G and, subsequently, in G'. Since the edges c, f, and h form a triangle in G', l cannot be incident with f in G' as it is incident with neither c nor h in G'. Thereby, l is not incident with f in G, which is a contradiction as l and f are incident in G at x. A similar argument gives that $deg(y) \leq 2$.

7. Given that H is a subgraph of G, $L(H)$ is an induced subgraph of $L(G)$. However, $L(G') = L(G) - e$, so $L(H) - e$ is an induced subgraph of $L(G')$. Thereby, $L(H) - e$ is a line graph.

□

Corollary 3.4 *Table 1 enumerates all of the possible core graphs, up to symmetry, for which the edge $\{a, b\}$ can be deleted from the corresponding line graph to produce a new line graph. For each core G the core of the new line graph is given by G'.*

Case	G	G'
A		
B		
C		

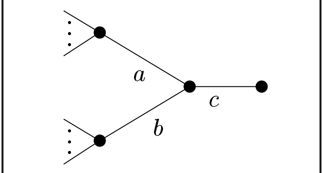
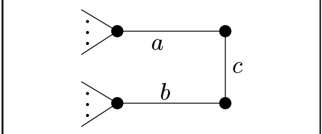
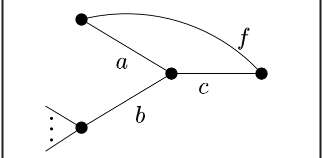
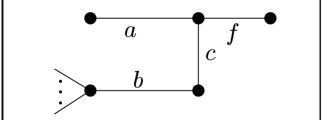
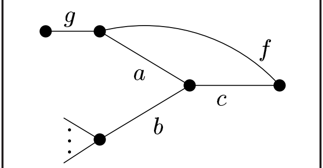
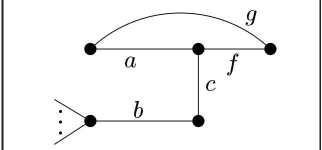
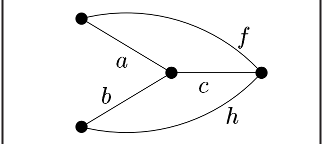
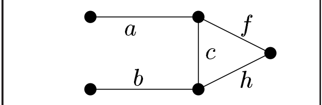
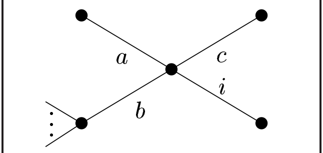
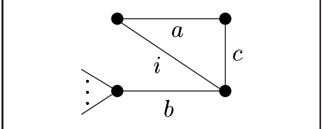
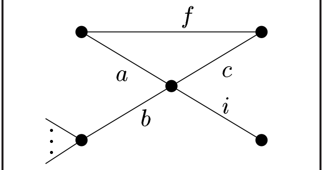
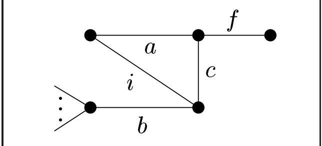
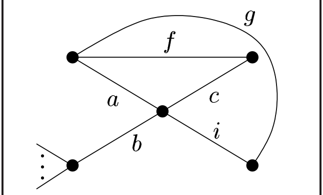
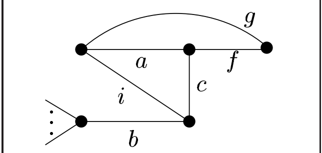
D		
E		
F		
G		
H		
I		
J		

Table 1: Possible cases for the deletion of an edge from a line graph.

In light of Corollary 3.4 the reader should observe that, unlike the addition of a new vertex, the choice of base as core of a line graph is irrelevant when it comes to deleting an edge from the line graph. Specifically, given a component of a line graph with two bases that are not partition isomorphic, if one base has a pair of incident edges that lead to one of the configurations presented in Table 1 then so too will these edges in the other base. In Table 2 we present all such pairs of bases that are not partition isomorphic in which the edge $\{a, b\}$ is to be deleted from the line graph, as well as the corresponding case of Table 1 to which each base belongs so that the reader can confirm that the choice of base is irrelevant.

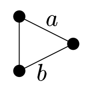
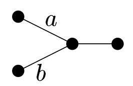
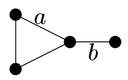
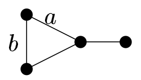
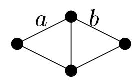
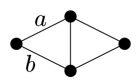
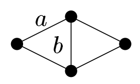
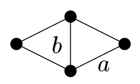
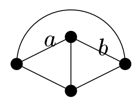
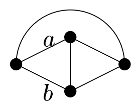
Case in Table 1	Base 1	Base 2	Case in Table 1
A			D
E			A
G			A
none			none
none			none

Table 2: Pairs of bases in which the edge $\{a, b\}$ is to be deleted from the line graph.

Since the circular linked list structure distributes the information about the neighbourhood of a vertex across its neighbourhood, the vertex labels are sufficient to determine the local structures as depicted in Table 1 in O(1) time. As such, an algorithm used to update the labels upon the deletion of an edge need only identify the structure of the core then alter the labels to represent the new core. The algorithm DeleteEdge presented in Appendix A does this in O(1) time.

3.4 Adding an edge to the base graph

The act of deleting an edge from a line graph is depicted in Figure 8. Since the process of adding an edge is exactly the reverse of deleting an edge, as such, Table 1 enumerates all the possibilities. Just as we saw with the deletion of an edge, the choice of base as core of a line graph is irrelevant when it comes to adding a new edge to the line graph. Specifically, if a component of a line graph has two bases that are not partition isomorphic, no pair of non-incident edges in the core will lead to one of the configurations presented in Table 1. A judicious examination of the bases shown in Figure 2(b) will confirm this.

Again, the labels of the vertices in the line graph are sufficient to determine the local structures as depicted in Table 1 so the algorithm for updating the labels need only identify the structure of the core, then alter the labels to represent the new core. Such an algorithm would be similar to that presented for the deletion of an edge from a line graph and could be designed to run in O(1) time.

4 Conclusion

Over the last fifteen years the concept of an implicit informative labelling scheme has evolved through the efforts of several authors, including Muller [15], Kannan et al. [12], Spinrad [22], and Peleg [18]. We define what is meant by a dynamic implicit f-labelling scheme and present a dynamic scheme for the class of line graphs. In particular, the graph operations which are permitted in this dynamic scheme for line graphs are the deletion of a vertex (and its incident edges), the addition of a vertex (and some incident edges), the deletion of an edge, and the addition of an edge. It is hoped that by studying the dynamic versions of these schemes we will increase the applicability of implicit informative labelling schemes to their real world problems.

Future research on dynamic implicit informative labelling schemes will reveal dynamic schemes for additional classes of graphs. It is expected that the circular linked list structure employed herein will also prove to be useful for these other classes of graphs as it allows the information about the neighbourhood of a vertex to be distributed over the labels of the neighbours.

Appendix A

A.1 Adding a vertex to the line graph

```
AddVertex($N(pre)$)
 1  $pre \leftarrow$ GetPrelabelLine()
 2  switch
 3   case $|N(pre)| = 0$:
 4      NoNeighbours()
 5   case $|N(pre)| \geq 1$:
 6      $(valid_0, vpt_0, valid_1, vpt_1, good) \leftarrow$ FindValid($N(pre)$)
 7      if $good = 1$ then
 8          EstablishEdgeInBase($valid_0, vpt_0, valid_1, vpt_1$)
 9      else error "this is no longer a line graph"

EstablishEdgeInBase($edge_0, end_0, edge_1, end_1$)
 1  if $edge_1 =$ NIL then
 2      $size \leftarrow 1$
 3      ValidSetOfSizeOne()
 4  else $size \leftarrow 2$
 5  for $i \leftarrow 0$ to $size - 1$ do
 6      $pre.ep_i \leftarrow edge_i.ep_{end_i}$
 7      $pre.nx_i \leftarrow edge_i.nx_{end_i}$
 8      $pre.nn_i \leftarrow valid_i.nn_{vpt_i}$
 9      $edge_i.nx_{end_i} \leftarrow pre$
10      $I \leftarrow$ GetIncidentNeighbors($pre, i$)
11      IncrementNN($I$)

IncrementNN($S$)
 1  while $S \neq$ Nil do
 2      $(u, end) \leftarrow$ Pop($S$)
 3      $u.nn_{end} \leftarrow u.nn_{end} + 1$

NoNeighbours()
 1  $pre.ep_0 \leftarrow$ GetPrelabelCore()
 2  $pre.ep_1 \leftarrow$ GetPrelabelCore()
 3  $pre.nn_0 \leftarrow 1$
 4  $pre.nn_1 \leftarrow 1$
 5  $pre.nx_0 \leftarrow pre$
 6  $pre.nx_1 \leftarrow pre$

ValidSetOfSizeOne()
 1  $pre.ep_1 \leftarrow$ GetPrelabelCore()
 2  $pre.nx_1 \leftarrow pre$
 3  $pre.nn_1 \leftarrow 1$

FindValid($N$)
 1  $done \leftarrow 0$
 2  $edge_0 \leftarrow$ some member of $N$
 3  $end_0 \leftarrow 0$
 4  $ischanged_0 \leftarrow 0$
 5  while $done = 0$ and $end_0 \leq 1$ do
 6      $(R, done) \leftarrow$ Eliminate($N, edge_0, end_0$)
 7      if $done = 1$ then
 8          if $R = \emptyset$ then
 9              $edge_1 \leftarrow$ NIL
10          else $done \leftarrow 0$
11              $edge_1 \leftarrow$ some member of $R$
```

```
12                  $end_1 \leftarrow 0$
13                  $ischanged_1 \leftarrow 0$
14                  while $done = 0$ and $end_1 \leq 1$ do
15                      $(R_1, done) \leftarrow$ Eliminate$(R, edge_1, end_1)$
16                      if $done = 0$ or $R_1 \neq \emptyset$ then
17                          $done \leftarrow 0$
18                          $end_1 \leftarrow end_1 + 1$
19                          if $ischanged_1 = 0$ and $end_1 = 2$ then
20                              $(C, ch) \leftarrow$ ChangeCore$(edge_1)$
21                              $ischanged_1 \leftarrow 1$
22                              if $ch = 1$ then
23                                  if $edge_0 \in C$ then
24                                      $(C, ch) \leftarrow$ ChangeCore$(edge_1)$
25                                  else $end_1 \leftarrow 0$
26      if $done = 0$ then
27          $end_0 \leftarrow end_0 + 1$
28          if $ischanged_0 = 0$ and $end_0 = 2$ then
29              $(C, ch) \leftarrow$ ChangeCore$(edge_0)$
30              $ischanged_0 \leftarrow 1$
31              if $ch = 1$ then
32                  $end_0 \leftarrow 0$
33  return $(edge_0, end_0, edge_1, end_1, done)$
```

```
Eliminate$(R_2, edge, i)$
 1  $val \leftarrow 1$
 2  $u \leftarrow edge$
 3  $end \leftarrow i$
 4  $R_2 \leftarrow R_2 - \{u\}$
 5  while $u.nx_{end} \neq edge$ and $val = 1$ do
 6      $u \leftarrow u.nx_{end}$
 7      $end \leftarrow$ FindEnd$(u, edge.ep_i)$
 8      if $u \in R_2$ then
 9          $R_2 \leftarrow R_2 - \{u\}$
10      else $val \leftarrow 0$
11  return $(R_2, val)$
```

```
ChangeCore$(a)$
 1  $changed \leftarrow 0$
 2  switch
 3   case $a.nn_0 = 2$ and $a.nn_1 = 2$:
 4      $b \leftarrow a.nx_0$
 5      $eb \leftarrow 1 -$ FindEnd$(b, a.ep_0)$
 6      $c \leftarrow a.nx_1$
 7      $ec \leftarrow 1 -$ FindEnd$(c, a.ep_1)$
 8      if $b.ep_{eb} = c.ep_{ec}$ then
 9          if $b.nn_{eb} = 2$ then
10              SwitchK3ToK13()
11              $comp \leftarrow \{a, b, c\}$
12              $changed \leftarrow 1$
13          elseif $b.nn_{eb} = 3$ then
14              if $c = b.nx_{eb}$ then
15                  $d \leftarrow c.nx_{ec}$
16              else $d \leftarrow b.nx_{eb}$
17              $ed \leftarrow 1 -$ FindEnd$(d, b.ep_{eb})$
18              if $d.nn_{ed} = 1$ then
19                  Switch$(a, d)$
20                  $comp \leftarrow \{a, b, c, d\}$
```

```
              $changed \leftarrow 1$
case ($a.nn_0 = 3$ and $a.nn_1 = 1$) or ($a.nn_0 = 1$ and $a.nn_1 = 3$):
   if $a.nn_0 = 3$ then
      $ea = 0$
   else $ea = 1$
   $b \leftarrow a.nx_{ea}$
   $eb \leftarrow 1 - \textsc{FindEnd}(b, a.ep_{ea})$
   $c \leftarrow b.nx_{1\text{-}eb}$
   $ec \leftarrow 1 - \textsc{FindEnd}(c, a.ep_{ea})$
   if $b.nn_{eb} = 1$ and $c.nn_{ec} = 1$ then
      $\textsc{SwitchK13ToK3}()$
      $comp \leftarrow \{a, b, c\}$
      $changed \leftarrow 1$
   elseif $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ then
      $d \leftarrow b.nx_{eb}$
      $ed \leftarrow 1 - \textsc{FindEnd}(d, b.ep_{eb})$
      if $d.ep_{ed} = c.ep_{ec}$ then
         $\textsc{Switch}(a, d)$
         $comp \leftarrow \{a, b, c, d\}$
         $changed \leftarrow 1$
case ($a.nn_0 = 3$ and $a.nn_1 = 2$) or ($a.nn_0 = 2$ and $a.nn_1 = 3$):
   if $a.nn_0 = 3$ then
      $ea = 0$
   else $ea = 1$
   $b \leftarrow a.nx_{ea}$
   $eb \leftarrow 1 - \textsc{FindEnd}(b, a.ep_{ea})$
   $c \leftarrow b.nx_{1\text{-}eb}$
   $ec \leftarrow 1 - \textsc{FindEnd}(c, a.ep_{ea})$
   $f \leftarrow a.nx_{1\text{-}ea}$
   $ef \leftarrow 1 - \textsc{FindEnd}(f, a.ep_{1\text{-}ea})$
   if $f.ep_{ef} = b.ep_{eb}$ then
      if $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$ then
         $\textsc{Switch}(c, f)$
         $comp \leftarrow \{a, b, c, f\}$
         $changed \leftarrow 1$
      elseif $b.nn_{eb} = 3$ and $c.nn_{ec} = 2$ then
         $d = c.nx_{ec}$
         $ed \leftarrow \textsc{FindEnd}(d, c.ep_{ec})$
         if $d.ep_{1\text{-}ed} = b.ep_{eb}$ then
            $\textsc{Switch}(c, f)$
            $comp \leftarrow \{a, b, c, d, f\}$
            $changed \leftarrow 1$
   elseif $f.ep_{ef} = c.ep_{ec}$ then
      if $b.nn_{eb} = 1$ and $c.nn_{ec} = 2$ then
         $\textsc{Switch}(b, f)$
         $comp \leftarrow \{a, b, c, f\}$
         $changed \leftarrow 1$
      elseif $b.nn_{eb} = 2$ and $c.nn_{ec} = 3$ then
         $d = b.nx_{eb}$
         $ed \leftarrow 1 - \textsc{FindEnd}(d, b.ep_{eb})$
         if $d.ep_{ed} = c.ep_{ec}$ then
            $\textsc{Switch}(b, f)$
            $comp \leftarrow \{a, b, c, d, f\}$
            $changed \leftarrow 1$
case $a.nn_0 = 3$ and $a.nn_1 = 3$:
   $b \leftarrow a.nx_0$
   $eb \leftarrow 1 - \textsc{FindEnd}(b, a.ep_0)$
   $c \leftarrow b.nx_{1\text{-}eb}$
```

```
        $ec \leftarrow 1 - \textsc{FindEnd}(c, a.ep_0)$
        $f \leftarrow a.nx_1$
        $ef \leftarrow 1 - \textsc{FindEnd}(f, a.ep_1)$
        $h \leftarrow f.nx_{1\text{-}ef}$
        $eh \leftarrow 1 - \textsc{FindEnd}(h, a.ep_1)$
        **if** $f.ep_{ef} = b.ep_{eb}$ and $h.ep_{eh} = c.ep_{ec}$ **then**
            **if** $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ **then**
                $\textsc{Switch}(c, f)$
                $comp \leftarrow \{a, b, c, f, h\}$
                $changed \leftarrow 1$
            **elseif** $b.nn_{eb} = 3$ and $c.nn_{ec} = 3$ **then**
                **if** $f = b.nx_{eb}$ **then**
                    $d \leftarrow f.nx_{ef}$
                **else** $d \leftarrow b.nx_{eb}$
                $ed \leftarrow 1 - \textsc{FindEnd}(d, b.ep_{eb})$
                **if** $d.ep_{ed} = c.ep_{ec}$ **then**
                    $\textsc{Switch}(c, f)$
                    $comp \leftarrow \{a, b, c, d, f, h\}$
                    $changed \leftarrow 1$
        **elseif** $f.ep_{ef} = c.ep_{ec}$ and $h.ep_{eh} = b.ep_{eb}$ **then**
            **if** $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ **then**
                $\textsc{Switch}(c, h)$
                $comp \leftarrow \{a, b, c, f, h\}$
                $changed \leftarrow 1$
            **elseif** $b.nn_{eb} = 3$ and $c.nn_{ec} = 3$ **then**
                **if** $h = b.nx_{eb}$ **then**
                    $d \leftarrow h.nx_{eh}$
                **else** $d \leftarrow b.nx_{eb}$
                $ed \leftarrow 1 - \textsc{FindEnd}(d, b.ep_{eb})$
                **if** $d.ep_{ed} = c.ep_{ec}$ **then**
                    $\textsc{Switch}(c, h)$
                    $comp \leftarrow \{a, b, c, d, f, h\}$
                    $changed \leftarrow 1$
    **return** $(comp, changed)$
```

```
$\textsc{SwitchK3ToK13}()$
    $\textsc{FreeCore}(a.ep_0)$
    $\textsc{FreeCore}(a.ep_1)$
    $\textsc{FreeCore}(b.ep_0)$
    $\textsc{FreeCore}(b.ep_1)$
    $\textsc{FreeCore}(c.ep_0)$
    $\textsc{FreeCore}(c.ep_1)$
    $a.ep_0 \leftarrow \textsc{GetPrelabelCore}$
    $a.ep_1 \leftarrow \textsc{GetPrelabelCore}$
    $b.ep_0 \leftarrow a.ep_0$
    $b.ep_1 \leftarrow \textsc{GetPrelabelCore}$
    $c.ep_0 \leftarrow a.ep_0$
    $c.ep_1 \leftarrow \textsc{GetPrelabelCore}$
    $a.nx_0 \leftarrow c$
    $c.nx_0 \leftarrow b$
    $b.nx_0 \leftarrow a$
    $a.nx_1 \leftarrow a$
    $b.nx_1 \leftarrow b$
    $c.nx_1 \leftarrow c$
    $a.nn_0 \leftarrow 3$
    $b.nn_0 \leftarrow 3$
    $c.nn_0 \leftarrow 3$
```

```
22   $a.nn_1 \leftarrow 1$
23   $b.nn_1 \leftarrow 1$
24   $c.nn_1 \leftarrow 1$
```

```
SwitchK13ToK3()
 1   FreeCore($a.ep_0$)
 2   FreeCore($a.ep_1$)
 3   FreeCore($b.ep_0$)
 4   FreeCore($b.ep_1$)
 5   FreeCore($c.ep_0$)
 6   FreeCore($c.ep_1$)
 7   $a.ep_0 \leftarrow$ GetPrelabelCore
 8   $a.ep_1 \leftarrow$ GetPrelabelCore
 9   $b.ep_0 \leftarrow a.ep_0$
10   $b.ep_1 \leftarrow$ GetPrelabelCore
11   $c.ep_0 \leftarrow a.ep_1$
12   $c.ep_1 \leftarrow b.ep_1$
13   $a.nx_0 \leftarrow b$
14   $b.nx_0 \leftarrow a$
15   $b.nx_1 \leftarrow c$
16   $c.nx_1 \leftarrow b$
17   $a.nx_1 \leftarrow c$
18   $c.nx_0 \leftarrow a$
19   $a.nn_0 \leftarrow 2$
20   $b.nn_0 \leftarrow 2$
21   $c.nn_0 \leftarrow 2$
22   $a.nn_1 \leftarrow 2$
23   $b.nn_1 \leftarrow 2$
24   $c.nn_1 \leftarrow 2$
```

```
Switch($e_0, e_1$)
 1   for $j \leftarrow 0$ to 1 do
 2       for $i \leftarrow 0$ to 1 do
 3           $u \leftarrow e_j$
 4           $end \leftarrow i$
 5           while $u.nx_{end} \neq e_j$ do
 6               $u \leftarrow u.nx_{end}$
 7               $end \leftarrow$ FindEnd($u, e_j.ep_i$)
 8           $u.nx_{end} \leftarrow e_{1-j}$
 9   $temp \leftarrow e_0$
10   $e_0 \leftarrow e_1$
11   $e_1 \leftarrow temp$
```

AddVertex:

Input: $N(pre)$, the set of prelabels of neighbours of a vertex to be added to the line graph.

Output: None, however, AddVertex changes the labels of the present graph in order to produce a labelling of the modified graph.

1: As the new vertex does not yet have a prelabel, the function GetPrelabelLine is used to assign one. This prelabel will be *pre*.

3-9: The new vertex, *pre* is isolated. Consequently, a new isolated edge corresponding to *pre* must be added to the core.

4,5: In adding an isolated edge to the core we must assign prelabels to its endpoints. GetPrelabelCore determines a prelabel for each new vertex of the core.

6,7: As *pre* is an isolated edge in the core, the number of neighbours of each of the endpoints of *pre* is one.

8,9: Since each of the endpoints of *pre* is incident only with *pre*, the next edge in the circular linked lists at these endpoints of is *pre*.

10-30: The new vertex *pre* has at least one neighbour; this is depicted in Figure 6 of Section 3. In order to add the corresponding edge to the core we must determine a valid set for $N(pre)$.

11-13: FindValid is the routine used to determine a valid set for $N(pre)$. If $good = 0$ then no valid set can be found, so we cannot add *pre* to the graph. If $good \neq 0$ then the valid set can be obtained using the values $valid_0$, vpt_0, $valid_1$, and vpt_1 which are returned by FindValid. At this point the reader may wish to skip ahead to the description of FindValid.

14-18: Given that there is a valid set, if $valid_1 =$ NIL then the valid set contains only one vertex ; the variable *size* is used to represent the size of the valid set. Since the valid set is of size one we must create a new vertex in the core for one of the endpoints of *pre*, namely, $pre.ep_1$. This endpoint is incident only with *pre*, therefore, $pre.nx_1 = pre$ and $pre.nn_1 = 1$

19: Given that there is a valid set, if it is not of size one then it must be of size two.

20-30: For each vertex in the valid set we set one of the endpoints of *pre* to be that vertex. Lines 22 and 23 insert *pre* into the circular linked list of each vertex in the valid set, while lines 24 through 29 increase by one the size of the neighbourhoods of each vertex in each of these circular linked lists. Finally, in line 30 we assign the number of neighbours of endpoints of pre. The addition of *pre* to the core is illustrated in Figures 9 and 10.

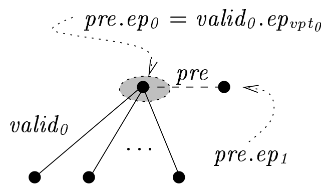


Figure 9: Having found a valid set of size one (this diagram is in the core).

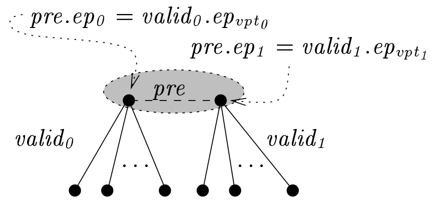

$pre.ep_0 = valid_0.ep_{vpt_0}$
$pre.ep_1 = valid_1.ep_{vpt_1}$
pre
$valid_0$
$valid_1$

Figure 10: Having found a valid set of size two (this diagram is in the core).

FindValid

Input: N, a set of prelabels for which we must find a valid set. Recall that the members of N can be considered as vertices in the line graph or as edges in the core.

Output: The five-tuple $(edge_0, end_0, edge_1, end_1, done)$ with values as follows.

- If N has a valid set then *done* will have value 1, otherwise it will have value 0.
- If N has a valid set then $edge_0$ is a member of N and $edge_0.ep_{end_0}$ is a vertex in the valid set.
- If N has a valid set of size two then $edge_0$ and $edge_1$ are distinct members of N, where $edge_0.ep_{end_0}$ and $edge_1.ep_{end_1}$ are the vertices in the valid set. If the valid set is of size one then $edge_1 =$ NIL.

1: The variable *done* is used to indicate if we have found a valid set. Although there are some exceptions, if $done = 1$ then we have found a valid set; otherwise, $done = 0$ and we must continue searching.

2: If N has a valid set then every member of N will have exactly one endpoint in the valid set. As such, we choose a member of N, namely $edge_0$, so as to include one of its endpoints in the valid set.

3: We first try to include the 0th endpoint of $edge_0$ $edge_0.ep_0$, in the valid set. If we later determine that $edge_0.ep_0$ cannot be included in any valid set, then we will try to include $edge_0.ep_1$. The value of end_0 indicates whether we are considering $edge_0.ep_0$ or $edge_0.ep_1$.

4: In Section 3 we discussed how a component of a line graph can have two bases which are not partition isomorphic; in particular, for a given set of prelabels, one of the bases may yield a valid set while the other may not. It may be necessary to change the core of a component in order to find a valid set. The variable $ischanged_0$ is used to indicate if we have attempted to change the core of the component containing $edge_0$. If $ischanged_0 = 1$ then we have previously attempted to change the core; otherwise, $ischanged_0 = 0$ and we have not tried to change the core. Recall that when we say that a core is changed we ultimately mean that the labelling of the line graph has been changed so as to reflect the new core.

5-32: As previously mentioned, we continue to look for a valid set providing $done = 0$ and providing there is at least one of $edge_0.ep_0$ and $edge_0.ep_1$ which we have not tried to include in a valid set.

6: Letting R be the subset of edges in N that are not incident with $edge_0.ep_{end_0}$, we observe that if there is another vertex in the valid set then it must come from an edge in R; in order to determine R, FindValid uses the function Eliminate. If the circular liked list at $edge_0.ep_{end_0}$ contains an edge which is not in N then Eliminate will set $done$ to 0; otherwise, it will set $done$ to 1 and return R. At this point the reader may wish to skip ahead to the description of Eliminate.

7-25: If $done = 1$ then the circular liked list at $edge_0.ep_{end_0}$ did not contain any edges not in N. As such, we may continue trying to place $edge_0.ep_{end_0}$ in the valid set.

8,9: Given that we have not yet found any reason to exclude $edge_0.ep_{end_0}$ from the valid set, if $R \neq \emptyset$ then all of the edge in N are incident with $edge_0.ep_{end_0}$; therefore, $\{edge_0.ep_{end_0}\}$ is a valid set. As such, FindValid sets $edge_1$, the second edge from which we might draw a vertex for the valid set, to NIL, then returns the appropriate values to AddVertex.

10-25: If $R \neq \emptyset$ then there are members of N which are not incident with $edge_0.ep_{end_0}$ so we must include a second vertex in the valid set.

10: Although Eliminate has set $done$ to 1 on line 6, we have not yet found a valid set so we assign $done$ the value 0.

11-13: As we did with $edge_0$, we choose an edge $edge_1$ from R and try to include one of its endpoints in the valid set. Like $edge_0$, $edge_1$ has corresponding variables end_1 and $ischanged_1$.

14-25: As we did with $edge_0$, we continue to look for a valid set providing $done = 0$ and providing there is at least one of $edge_1.ep_0$ and $edge_1.ep_1$ which we have not tried to include in the valid set. We first check if $\{edge_0.ep_{end_0}, edge_1.ep_0\}$ is a valid set, if it is not then we will try $\{edge_0.ep_{end_0}, edge_1.ep_1\}$.

15: Letting R_1 be the subset of edges in R that are not incident with $edge_1.ep_{end_1}$, we observe that $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is a valid set if and only if $R_1 = \emptyset$ and the circular linked list at $edge_1.ep_{end_1}$ does not contain any edges which are not in R. To determine R_1 FindValid uses the routine Eliminate just as it did to determine R.

16-25: If $done = 1$ and $R_1 = \emptyset$ then $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is a valid set so FindValid does nothing else before returning the appropriate values to AddVertex. Otherwise, $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is not a valid set, so we will need to try another endpoint of $edge_1$ or perhaps another base for the core component containing $edge_1$.

17: If Eliminate determined that all of the edges adjacent to $edge_1.ep_{end_1}$ were in R, then it would set $done$ to 1; however, if $R_1 \neq \emptyset$ then we have not found a valid set so we should set $done$ to 0 accordingly.

18: As we have determined that $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is not a valid set, we try to include a different endpoint of $edge_1$ in the valid set; this is achieved by incrementing end_1 by one. If end_1 now has value 1 then return to line 15 and try to include $edge_1.ep_1$ in the valid set.

19-25: If $end_1 = 2$ then we have already tried to include $edge_1.ep_1$ in the valid set so we must now try changing the core of the component containing $edge_1$. This is only allowed if the core has not been changed, that is, if $ischanged_1 = 0$.

20: To change the core of the component containing edge_1, FINDVALID relies on a function called CHANGECORE. C is the set of prelabels of all the vertices in the same component as edge_1; ch is a variable used to represent whether or not the core was changed. If $\mathit{ch} = 1$, the core was changed; otherwise, $\mathit{ch} = 0$ and the core was not changed.

21: We set $\mathit{ischanged}_1$ to 1 in order to indicate that an attempt was made to change the core of the component containing edge_1. The reader should note the distinction between ch and $\mathit{ischanged}_1$; $\mathit{ischanged}_1$ merely indicates that an attempt was made to change the core, whereas ch indicates if a change was actually made.

22-25: If the core of the component containing edge_1 was changed there are two possibilities; either C contains edge_0 or it does not. If it does, then we change the core of the component containing edge_1 back to its original state as we do not wish to change the core of the component containing edge_0 at this time; if it does not, then we set end_0 to 0 and repeat the process of trying to include an endpoint of edge_1 in the valid set along with $\mathit{edge}_0.\mathit{ep}_{\mathit{end}_0}$.

26-32: If $\mathit{done} = 0$ then either the call of ELIMINATE in line 6 found an edge in the circular linked list at $\mathit{edge}_0.\mathit{ep}_{\mathit{end}_0}$ which was not in N or, in choosing an edge edge_1 from R, neither endpoint of edge_1 could be put in a valid set with $\mathit{edge}_0.\mathit{ep}_{\mathit{end}_0}$ regardless of the base used to represent the component containing edge_1. Either way, $\mathit{edge}_0.\mathit{ep}_{\mathit{end}_0}$ cannot belong to a valid set using the present base. This segment of the algorithm is similar to that involving edge_1 in lines 17 though 25.

33: FINDVALID returns the values of edge_0, end_0, edge_1, end_1, and done to ADDVERTEX.

ELIMINATE

Input: A triple (R_2, edge, i), where R_2 is a set of prelabels from which we will eliminate the edges belonging to the circular linked list about $\mathit{edge}.\mathit{ep}_i$.

Output: The pair (R_2, val), whose values are as follows.

- If all of the members of the circular linked list about $\mathit{edge}.\mathit{ep}_i$ are in R_2 then $\mathit{val} = 1$; otherwise, $\mathit{val} = 0$.
- Providing $\mathit{val} = 1$, the output R_2 is the input R_2 less the edges of the circular linked list about $\mathit{edge}.\mathit{ep}_i$.

1: As mentioned above, the variable val is used to indicate if any of the members of the circular linked list about $\mathit{edge}.\mathit{ep}_i$ are not in R_2. The default value of val is set to 1 and will be changed to 0 if there is an edge in the circular linked list that is not in R_2.

2,3: The variables u and end are placeholders that allow us to traverse the circular linked lists about $\mathit{edge}.\mathit{ep}_i$.

4: Since edge is in the circular linked list about $\mathit{edge}.\mathit{ep}_i$ we eliminate it from R_2.

5-10: We proceed through the circular linked list eliminating prelabels from R_2. If an edge in the circular linked list is found not to be in R_2 we set val to be 0, which causes us to break out of the while loop.

11: We return the values of R_2 and val to FINDVALID as described above.

CHANGECORE:

Input: A prelabel a. Recall that a can be considered as a vertex in the line graph or as an edge in the core; in the majority of the description of this algorithm a is considered as an edge in the core. If possible, CHANGECORE will change the labelling of the line graph such that it corresponds to a new core in which the component of the core containing a has been changed to another base which is not partition isomorphic. Observe that we may consider either the core of the component containing a or the component of the core containing a; the preciseness of each phrasing depends on whether a is considered as a vertex in the line graph, or as an edge in the core.

Output: The pair $(\mathit{comp}, \mathit{changed})$, where comp is a set of prelabels and $\mathit{changed}$ is a value, either 0 or 1. The variable $\mathit{changed}$ will have value 1 and comp will be the set of prelabels belonging to the vertices of the component containing a if and only if the component of the core containing a is one of the graphs listed in Theorem 3.2. Again, if possible, CHANGECORE changes the component of the core containing a to another base which is not partition isomorphic. In particular, CHANGECORE need only modify the labels of the vertices in the component containing a.

1: As mentioned, the variable *changed* is used to indicate if the core has been changed. The default value of *changed* is 0 and will be set to 1 when the core is changed.

2-157: In determining if the core can be changed we consider a series of cases based upon the degrees of the endpoints of a.

3-44: In this case each endpoint of a is incident with exactly one other edge besides a.

4-7: We let b and c be the edges, other than a, incident with $a.ep_0$ and $a.ep_1$, respectively. The endpoints $b.ep_{eb}$ and $c.ep_{ec}$ are set to be the endpoints of b and c, respectively, which are furthest from a.

8-44: Either $b.ep_{eb}$ and $c.ep_{ec}$ are the same vertex or they are not. If they are not, then the condition $a.nn_0 = a.nn_1 = 2$ guarantees that the component of the core containing a has an induced P_4 or C_4; since none of the bases of the graphs found in Theorem 3.2 (i.e. the graphs in Figure 2(b)) has an induced P_4 or C_4, the core cannot be changed.

9-35: If $b.nn_{eb} = 2$ then $b.ep_{eb}$ is incident only with b and c. Thereby, the component of the core containing a is the K_3 shown in Figure 11(a) so we change it to the $K_{1,3}$ shown in Figure 11(b).

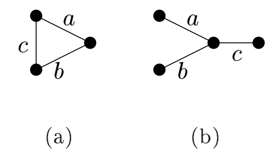


Figure 11: Two bases of the same graph which are not partition isomorphic.

36-44: If $b.nn_{eb} = 3$ then $b.ep_{eb}$ is incident with b, c, and another vertex which we will call d. Observe that if $b.nn_{eb} > 3$ then the core cannot be changed as none of the graphs found in in Figure 2(b) has a vertex of degree greater than three.

37-40: We ensure that d is distinct from b and c, then set $d.ep_{ed}$ to be the endpoint of d that is furthest from b.

41-44: If $d.nn_{ed} = 1$ then the core is as shown in Figure 12(a); using the function SWITCH we change the core to the graph depicted in Figure 12(b). Furthermore, observe that if $d.nn_{ed} > 1$ then the conditions $a.nn_0 = 2$ and $b.nn_{eb} = 3$ guarantee that the component of the core containing a has an induced P_4 which prevents the core from being changed.

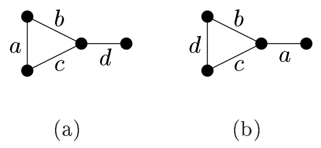


Figure 12: Two bases of the same graph which are not partition isomorphic.

45-86: We now consider the case when one endpoint of a is incident with two additional edges besides a, and the other endpoint is incident with only a itself.

46-48: We set $a.ep_{ea}$ to be the endpoint of a with degree three.

49-52: We let b and c be the edges, other than a, that are incident with $a.ep_{ea}$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a.

53-86 Given that $a.nn_{1\text{-}ea} = 1$ and $a.nn_{ea} = 3$, the only way that the core can be changed is if $b.nn_{eb} = c.nn_{ec} = 1$ or if $b.nn_{eb} = c.nn_{ec} = 2$. Otherwise, the component of the core containing a has an induced P_4 which prevents the core from being changed.

53-79: In this case the core is the $K_{1,3}$ shown in Figure 11(b) so we change it to the K_3 shown in Figure 11(a).

80-86: In this case both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

81,82: We let d be the edge, other than b, that is incident with $b.ep_{eb}$. Moreover, we let $d.ep_{ed}$ be the endpoint of d that is furthest from b.

83-86: If $d.ep_{ed} \neq c.ep_{ec}$ then the condition that $a.nn_{1\text{-}ea} = 1$ guarantees that the component of the core containing a has an induced P_4 which prevents the core from being changed. On the other hand, if $d.ep_{ed} = c.ep_{ec}$ then the core is as shown in Figure 12(b) so we change it to the base shown in Figure 12(a).

87-120: We now consider the case when one endpoint of a is incident with two additional edges besides a, and the other endpoint is incident with one additional edge besides a.

88-90: We set $a.ep_{ea}$ to be the endpoint of a with degree three.

91-94: We let b and c be the edges, other than a, that are incident with $a.ep_{ea}$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a.

95-96: We let f be the edge, other than a, that is incident with $a.ep_{1\text{-}ea}$. Moreover, we let $f.ep_{ef}$ be the endpoint of f that is furthest from a.

97-120 If neither $b.ep_{eb} = f.ep_{ef}$, nor $c.ep_{ec} = f.ep_{ef}$, then the component of the core containing a has an induced P_4 which prevents the core from being changed.

97-108 Given that $a.nn_{ea} = 3$, the only way that the core can be changed is if $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$ or if $b.nn_{eb} = 3$ and $c.nn_{ec} = 2$. Otherwise, the component of the core containing a has an induced P_4 which prevents the core from being changed.

98-101: If $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$ then there are no additional edges in the graph; the core is as shown in Figure 13(a) so we change it to the base shown in Figure 13(b).

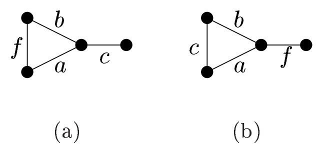


Figure 13: Two bases of the same graph which are not partition isomorphic.

102-108: In this case both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

103-104: We let d be the edge, other than c, that is incident with $c.ep_{ec}$. Moreover, we let $d.ep_{1\text{-}ed}$ be the endpoint of d that is furthest from c.

105-108: If $d.ep_{1\text{-}ed} \neq b.ep_{eb}$ then the condition that $a.nn_{ea} = 3$ guarantees that the component of the core containing a has an induced P_4 which prevents the core from being changed. On the other hand, if $d.ep_{1\text{-}ed} = b.ep_{eb}$ then the core is as shown in Figure 14(a) so we change it to the base shown in Figure 14(b).

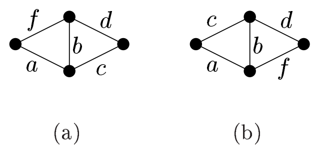


Figure 14: Two bases of the same graph which are not partition isomorphic.

109-120: This case is analogous to that found in lines 97 through 108, except that $c.ep_{ec} = f.ep_{ef}$ not $b.ep_{eb} = f.ep_{ef}$.

121-157: We now consider the case when both endpoints of a are incident with two additional edges besides a.

122-129: We let b and c be the edges, other than a, that are incident with $a.ep_0$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a. The edges f and h are defined similarly for $a.ep_1$.

130-157 If $b.ep_{eb} \neq f.ep_{ef}$ or $c.ep_{ec} \neq h.ep_{eh}$, and $b.ep_{eb} \neq h.ep_{eh}$ or $c.ep_{ec} \neq f.ep_{ef}$ then the component of the core containing a has an induced P_4 or C_4 which prevents the core from being changed.

130-143 Given that $a.nn_0 = a.nn_1 = 3$, the only way that the core can be changed is if $b.nn_{eb} = c.nn_{ec} = 2$ or if $b.nn_{eb} = c.nn_{ec} = 3$. Otherwise, the component of the core containing a has an induced P_4 which prevents the core from being changed.

131-134: If $b.nn_{eb} = c.nn_{ec} = 2$ then there are no additional edges in the graph; the core is as shown in Figure 14(a) so we change it to the base shown in Figure 14(b).

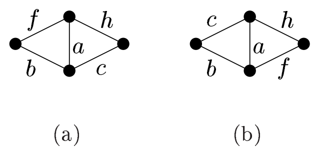


Figure 15: Two bases of the same graph which are not partition isomorphic.

135-143: In this case both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

135-139: We let d be the edge, other than b and f, that is incident with $b.ep_{eb}$. Moreover, we let $d.ep_{ed}$ be the endpoint of d that furthest from b.

140-143: If $d.ep_{1\text{-}ed} \neq c.ep_{ec}$ then the component of the core containing a has an induced P_4 which prevents the core from being changed. On the other hand, if $d.ep_{1\text{-}ed} = c.ep_{ec}$ then the core is as shown in Figure 16(a) so we change it to the base shown in Figure 16(b).

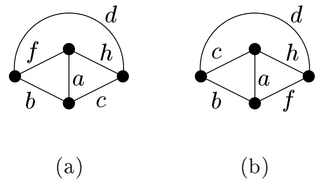


Figure 16: Two bases of the same graph which are not partition isomorphic.

144-157: This case is analogous to that found in lines 130 through 143, except that $c.ep_{ec} = f.ep_{ef}$ and $b.ep_{eb} = h.ep_{eh}$ not $b.ep_{eb} = f.ep_{ef}$ and $c.ep_{ec} = h.ep_{eh}$.

158: When ChangeCore is finished it *comp* and *changed* as described in the output section above.

Switch:

Input: A pair of prelabels (e_0, e_1). We will change the labellign of the graph to reflect a new core in which the edges e_0 and e_1 are switched. Recall that both e_0 and e_1 can be considered as vertices in the line graph or as edges in the core.

Output: None, however, we will change the labelling of the graph to reflect a new core in which the edges e_0 and e_1 are switched.

1-3: We switch the two labels.

4-11: The critical part of SWITCH, we ensure that the circular linked lists about the endpoints of e_0 and e_1 contain the correct information. For each endpoint of each of e_0 and e_1 we determine the edge in the circular linked list at the endpoint which points to the corresponding edge, then replace it with the other.

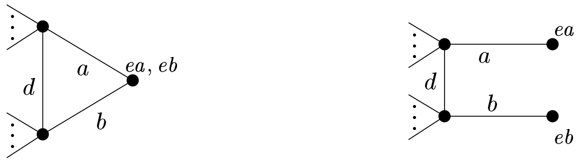
A.2 Deleting an edge from the line graph

```
DELETEEDGE($a, b$)
 1   for $i \leftarrow 0$ to 1 do
 2       for $j \leftarrow 0$ to 1 do
 3           if $a.ep_i = b.ep_j$ then
 4               $ea \leftarrow i$
 5               $eb \leftarrow j$
 6   switch
 7    case $a.nn_{ea} = 2$:
 8       CASEAC()
 9    case $a.nn_{ea} = 3$:
10       if $a.nx_{ea} = b$ then
11           $c \leftarrow b.nx_{eb}$
12       else $c \leftarrow a.nx_{ea}$
13       $ec \leftarrow 1 -$ FINDEND($c, a.ep_{ea}$)
14       switch
15        case $c.nn_{ec} = 1$
16           CASEBD()
17        case $c.nn_{ec} = 2$
18           $f \leftarrow c.nx_{ec}$
19           $ef \leftarrow 1 -$ FINDEND($f, c.ep_{ec}$)
20           switch
21            case $f.ep_{ef} = a.ep_{1\text{-}ea}$
22               switch
23                case $f.nn_{ef} \geq 4$
24                   error "this is no longer a line graph"
25                case $f.nn_{ef} = 3$:
26                   if $f.nx_{ef} = a$ then
27                       $g \leftarrow a.nx_{1\text{-}ea}$
28                   else $g \leftarrow f.nx_{ef}$
29                   $eg \leftarrow 1 -$ FINDEND($g, f.ep_{ef}$)
30                   if $g.nn_{eg} = 1$ then
31                       CASEF()
32                   else error "this is no longer a line graph"
33                case $f.nn_{ef} = 2$:
34                   CASEE()
35            case $f.ep_{ef} = b.ep_{1\text{-}eb}$
36               switch
37                case $f.nn_{ef} \geq 4$
38                   error "this is no longer a line graph"
39                case $f.nn_{ef} = 3$:
40                   if $f.nx_{ef} = b$ then
41                       $g \leftarrow b.nx_{1\text{-}eb}$
42                   else $g \leftarrow f.nx_{ef}$
43                   $eg \leftarrow 1 -$ FINDEND($g, f.ep_{ef}$)
44                   if $g.nn_{eg} = 1$ then
45                       CASEFSYMMETRIC()
46                   else error "this is no longer a line graph"
47                case $f.nn_{ef} = 2$:
48                   CASEESYMMETRIC()
49            case $f.ep_{ef} = a.ep_{1\text{-}ea}$ and $f.ep_{ef} = b.ep_{1\text{-}eb}$:
```

```
            error "this is no longer a line graph"
    case $c.nn_{ec} = 3$:
        if $a.nn_{1\text{-}ea} \neq 2$ or $b.nn_{1\text{-}eb} \neq 2$ then
            error "this is no longer a line graph"
        else $f \leftarrow a.nx_{1\text{-}ea}$
            $h \leftarrow b.nx_{1\text{-}eb}$
        $ef \leftarrow$ FindEnd($f, a.ep_{1\text{-}ea}$)
        $eh \leftarrow$ FindEnd($h, b.ep_{1\text{-}eb}$)
        if $f.ep_{1\text{-}ef} \neq c.ep_{ec}$ or $h.ep_{1\text{-}eh} \neq c.ep_{ec}$ then
            error "this is no longer a line graph"
        else CaseG()
    case $c.nn_{ec} > 3$:
        error "this is no longer a line graph"
case $a.nn_{ea} = 4$:
    if $a.nx_{ea} \neq b$ then
        $c \leftarrow a.nx_{ea}$
    else $c \leftarrow b.nx_{eb}$
    $ec \leftarrow 1 -$ FindEnd($c, a.ep_{ea}$)
    if $a.nx_{ec} \neq b$ or $a.nx_{ec} \neq c$ then
        $i \leftarrow a.nx_{ea}$
    elseif $b.nx_{eb} \neq a$ or $b.nx_{eb} \neq c$ then
        $i \leftarrow b.nx_{eb}$
    else $i \leftarrow c.nx_{1\text{-}ec}$
    $ei \leftarrow 1 -$ FindEnd($i, a.ep_{ea}$)
    switch
    case $c.nn_{ec} > 2$ or $i.nn_{ei} > 2$
        error "this is no longer a line graph"
    case $c.nn_{ec} = 1$ and $i.nn_{ei} = 1$
        if $a.nn_{1\text{-}ea} = 1$ then
            CaseH()
        elseif $b.nn_{1\text{-}eb} = 1$ then
            CaseHSymmetric()
        else error "this is no longer a line graph"
    case $c.nn_{ec} = 2$ and $i.nn_{ei} = 2$
        $f \leftarrow c.nx_{ec}$
        $g \leftarrow i.nx_{ei}$
        $ef \leftarrow 1 -$ FindEnd($f, c.ep_{ec}$)
        $eg \leftarrow$ FindEnd($g, i.ep_{ei}$)
        if $f.ep_{ef} = g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$ and $f.nn_{ef} = 3$ then
            CaseJ()
        elseif $f.ep_{ef} = g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$ and $f.nn_{ef} = 3$ then
            CaseJSymmetric()
        else error "this is no longer a line graph"
    case $i.nn_{ei} = 1$ and $c.nn_{ec} = 2$:
        $f \leftarrow c.nx_{ec}$
        $ef \leftarrow 1 -$ FindEnd($c, c.ep_{ec}$)
        if $f.ep_{ef} = a.ep_{1\text{-}ea}$ and $f.nn_{ef} = 2$ then
            CaseI()
        elseif $f.ep_{ef} = b.ep_{1\text{-}eb}$ and $f.nn_{ef} = 2$ then
            CaseISymmetric()
        else error "this is no longer a line graph"
    case $i.nn_{ei} = 2$ and $c.nn_{ec} = 1$:
        $g \leftarrow i.nx_{ei}$
        $eg \leftarrow$ FindEnd($g, i.ep_{ei}$)
        if $g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$ and $g.nn_{1\text{-}eg} = 2$ then
            CaseISymmetric()
        elseif $g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$ and $g.nn_{1\text{-}eg} = 2$ then
            CaseISymmetric()
```

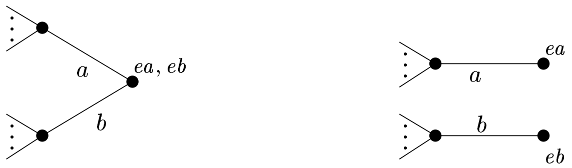
```
108            else error "this is no longer a line graph"
109      case $a.nn_{ea} > 4$
110         error "this is no longer a line graph"
```

```
CaseAC()
  1   FreeCore($a.ep_{ea}$)
  2   FreeCore($b.ep_{eb}$)
  3   $a.ep_{ea} \leftarrow$ GetPrelabelCore()
  4   $b.ep_{eb} \leftarrow$ GetPrelabelCore()
  5   $a.nx_{ea} \leftarrow a$
  6   $b.nx_{eb} \leftarrow b$
  7   $a.nn_{ea} \leftarrow 1$
  8   $a.nn_{1\text{-}ea} \leftarrow a.nn_{1\text{-}ea}$
  9   $b.nn_{eb} \leftarrow 1$
 10   $b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
```



(a) G (b) G'

Figure 17: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case A of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseAC.

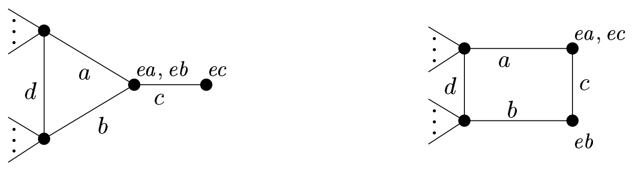


(a) G (b) G'

Figure 18: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case C of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseAC.

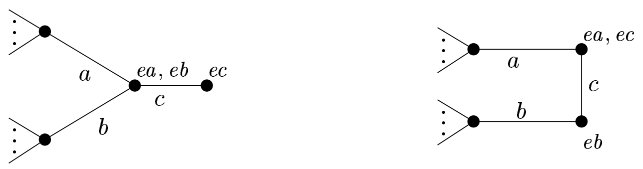
```
CaseBD()
  1   FreeCore($a.ep_{ea}$)
  2   FreeCore($b.ep_{eb}$)
  3   FreeCore($c.ep_{ec}$)
  4   FreeCore($c.ep_{1\text{-}ec}$)
  5   $a.ep_{ea} \leftarrow$ GetPrelabelCore()
  6   $b.ep_{eb} \leftarrow$ GetPrelabelCore()
  7   $c.ep_{ec} \leftarrow a.ep_{ea}$
```

```
 8   $c.ep_{\mathit{1}\text{-}ec} \leftarrow b.ep_{eb}$
 9   $a.nx_{ea} \leftarrow c$
10   $c.nx_{ec} \leftarrow a$
11   $b.nx_{eb} \leftarrow c$
12   $c.nx_{\mathit{1}\text{-}ec} \leftarrow b$
13   $a.nn_{ea} \leftarrow 2$
14   $a.nn_{\mathit{1}\text{-}ea} \leftarrow a.nn_{\mathit{1}\text{-}ea}$
15   $b.nn_{eb} \leftarrow 2$
16   $b.nn_{\mathit{1}\text{-}eb} \leftarrow b.nn_{\mathit{1}\text{-}eb}$
17   $c.nn_{ec} \leftarrow 2$
18   $c.nn_{\mathit{1}\text{-}ec} \leftarrow 2$
```



(a) G (b) G'

Figure 19: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case B of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseBD.

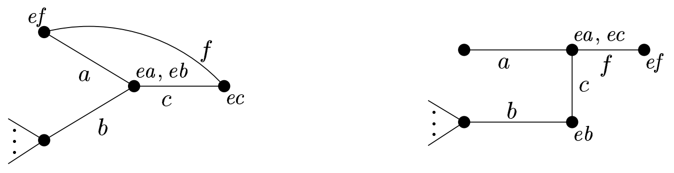


(a) G (b) G'

Figure 20: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case D of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseBD.

```
CaseE()
 1   FreeCore($a.ep_{ea}$)
 2   FreeCore($a.ep_{\mathit{1}\text{-}ea}$)
 3   FreeCore($b.ep_{eb}$)
 4   FreeCore($c.ep_{ec}$)
 5   FreeCore($c.ep_{\mathit{1}\text{-}ec}$)
 6   FreeCore($f.ep_{ef}$)
 7   FreeCore($f.ep_{\mathit{1}\text{-}ef}$)
 8   $a.ep_{ea} \leftarrow$ GetPrelabelCore()
 9   $a.ep_{\mathit{1}\text{-}ea} \leftarrow$ GetPrelabelCore()
10   $b.ep_{eb} \leftarrow$ GetPrelabelCore()
11   $f.ep_{ef} \leftarrow$ GetPrelabelCore()
```

```
12    $c.ep_{ec} \leftarrow a.ep_{ea}$
13    $c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
14    $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
15    $b.nx_{eb} \leftarrow c$
16    $c.nx_{1\text{-}ec} \leftarrow b$
17    $c.nx_{ec} \leftarrow a$
18    $a.nx_{ea} \leftarrow f$
19    $f.nx_{1\text{-}ef} \leftarrow c$
20    $f.nx_{ef} \leftarrow f$
21    $a.nx_{1\text{-}ea} \leftarrow a$
22    $a.nn_{ea} \leftarrow 3$
23    $a.nn_{1\text{-}ea} \leftarrow 1$
24    $b.nn_{eb} \leftarrow 2$
25    $b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
26    $c.nn_{ec} \leftarrow 3$
27    $c.nn_{1\text{-}ec} \leftarrow 2$
28    $f.nn_{ef} \leftarrow 1$
29    $f.nn_{1\text{-}ef} \leftarrow 3$
```

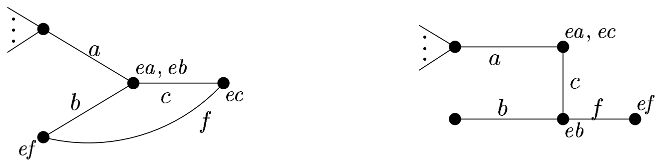


(a) G (b) G'

Figure 21: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case E of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseE.

```
CaseESymmetric()
1     FreeCore($a.ep_{ea}$)
2     FreeCore($b.ep_{eb}$)
3     FreeCore($b.ep_{1\text{-}eb}$)
4     FreeCore($c.ep_{ec}$)
5     FreeCore($c.ep_{1\text{-}ec}$)
6     FreeCore($f.ep_{ef}$)
7     FreeCore($f.ep_{1\text{-}ef}$)
8     $a.ep_{ea} \leftarrow$ GetPrelabelCore()
9     $b.ep_{eb} \leftarrow$ GetPrelabelCore()
10    $b.ep_{1\text{-}eb} \leftarrow$ GetPrelabelCore()
11    $f.ep_{ef} \leftarrow$ GetPrelabelCore()
12    $c.ep_{ec} \leftarrow a.ep_{ea}$
13    $c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
14    $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
15    $b.nx_{eb} \leftarrow c$
16    $c.nx_{1\text{-}ec} \leftarrow f$
17    $f.nx_{1\text{-}ef} \leftarrow b$
18    $c.nx_{ec} \leftarrow a$
19    $a.nx_{ea} \leftarrow c$
20    $f.nx_{ef} \leftarrow f$
21    $b.nx_{1\text{-}eb} \leftarrow b$
```

```
$a.nn_{ea} \leftarrow 2$
$a.nn_{1\text{-}ea} \leftarrow a.nn_{1\text{-}ea}$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1\text{-}eb} \leftarrow 1$
$c.nn_{ec} \leftarrow 2$
$c.nn_{1\text{-}ec} \leftarrow 3$
$f.nn_{ef} \leftarrow 1$
$f.nn_{1\text{-}ef} \leftarrow 3$
```

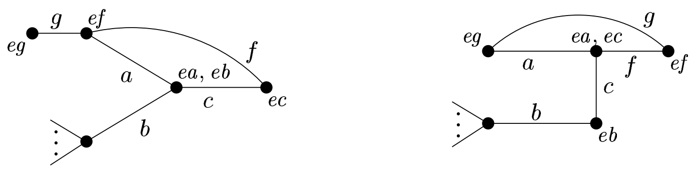


(a) G (b) G'

Figure 22: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (symmetric to case E of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseESymmetric.

```
CaseF()
    FreeCore($a.ep_{ea}$)
    FreeCore($a.ep_{1\text{-}ea}$)
    FreeCore($b.ep_{eb}$)
    FreeCore($c.ep_{ec}$)
    FreeCore($c.ep_{1\text{-}ec}$)
    FreeCore($f.ep_{ef}$)
    FreeCore($f.ep_{1\text{-}ef}$)
    FreeCore($g.ep_{eg}$)
    FreeCore($g.ep_{1\text{-}eg}$)
    $a.ep_{ea} \leftarrow$ GetPrelabelCore()
    $a.ep_{1\text{-}ea} \leftarrow$ GetPrelabelCore()
    $b.ep_{eb} \leftarrow$ GetPrelabelCore()
    $f.ep_{ef} \leftarrow$ GetPrelabelCore()
    $c.ep_{ec} \leftarrow a.ep_{ea}$
    $c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
    $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
    $g.ep_{eg} \leftarrow a.ep_{1\text{-}ea}$
    $g.ep_{1\text{-}eg} \leftarrow f.ep_{ef}$
    $b.nx_{eb} \leftarrow c$
    $c.nx_{1\text{-}ec} \leftarrow b$
    $c.nx_{ec} \leftarrow a$
    $a.nx_{ea} \leftarrow f$
    $f.nx_{1\text{-}ef} \leftarrow c$
    $f.nx_{ef} \leftarrow g$
    $g.nx_{1\text{-}eg} \leftarrow f$
    $g.nx_{eg} \leftarrow a$
    $a.nx_{1\text{-}ea} \leftarrow g$
    $a.nn_{ea} \leftarrow 3$
    $a.nn_{1\text{-}ea} \leftarrow 2$
    $b.nn_{eb} \leftarrow 2$
    $b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
```

```
32   $c.nn_{ec} \leftarrow 3$
33   $c.nn_{1\text{-}ec} \leftarrow 2$
34   $f.nn_{ef} \leftarrow 2$
35   $f.nn_{1\text{-}ef} \leftarrow 3$
36   $g.nn_{eg} \leftarrow 2$
37   $g.nn_{1\text{-}eg} \leftarrow 2$
```

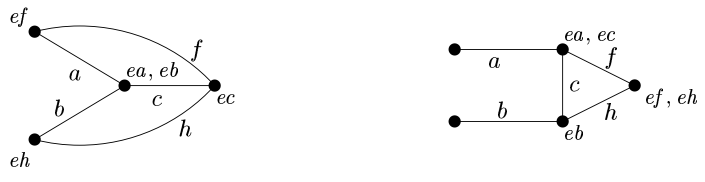


(a) G (b) G'

Figure 23: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case F of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseF.

```
CaseG()
1    FreeCore($a.ep_{ea}$)
2    FreeCore($a.ep_{1\text{-}ea}$)
3    FreeCore($b.ep_{eb}$)
4    FreeCore($c.ep_{ec}$)
5    FreeCore($c.ep_{1\text{-}ec}$)
6    FreeCore($f.ep_{ef}$)
7    FreeCore($f.ep_{1\text{-}ef}$)
8    FreeCore($h.ep_{eh}$)
9    FreeCore($h.ep_{1\text{-}eh}$)
10   $a.ep_{ea} \leftarrow$ GetPrelabelCore()
11   $a.ep_{1\text{-}ea} \leftarrow$ GetPrelabelCore()
12   $b.ep_{eb} \leftarrow$ GetPrelabelCore()
13   $b.ep_{1\text{-}eb} \leftarrow$ GetPrelabelCore()
14   $f.ep_{ef} \leftarrow$ GetPrelabelCore()
15   $c.ep_{ec} \leftarrow a.ep_{ea}$
16   $c.ep_{1\text{-}ec} \leftarrow b.ep_{1\text{-}eb}$
17   $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
18   $h.ep_{eh} \leftarrow f.ep_{ef}$
19   $h.ep_{1\text{-}eh} \leftarrow b.ep_{eb}$
20   $b.nx_{eb} \leftarrow c$
21   $c.nx_{1\text{-}ec} \leftarrow h$
22   $h.nx_{1\text{-}eh} \leftarrow b$
23   $c.nx_{ec} \leftarrow a$
24   $a.nx_{ea} \leftarrow f$
25   $f.nx_{1\text{-}ef} \leftarrow c$
26   $f.nx_{ef} \leftarrow h$
27   $h.nx_{eh} \leftarrow f$
28   $a.nx_{1\text{-}ea} \leftarrow a$
29   $b.nx_{1\text{-}eb} \leftarrow b$
30   $a.nn_{ea} \leftarrow 3$
31   $a.nn_{1\text{-}ea} \leftarrow 1$
32   $b.nn_{eb} \leftarrow 3$
33   $b.nn_{1\text{-}eb} \leftarrow 1$
```

```
34  $c.nn_{ec} \leftarrow 3$
35  $c.nn_{1-ec} \leftarrow 3$
36  $f.nn_{ef} \leftarrow 2$
37  $f.nn_{1-ef} \leftarrow 3$
38  $h.nn_{eh} \leftarrow 2$
39  $h.nn_{1-eh} \leftarrow 3$
```

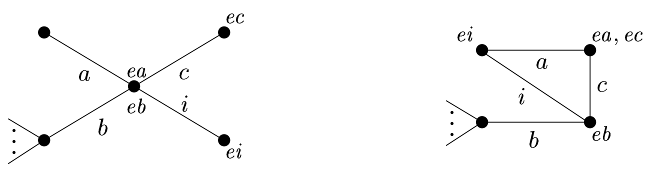


(a) G (b) G'

Figure 24: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case G of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseG.

```
CaseH()
 1  FreeCore($a.ep_{ea}$)
 2  FreeCore($a.ep_{1-ea}$)
 3  FreeCore($b.ep_{eb}$)
 4  FreeCore($c.ep_{ec}$)
 5  FreeCore($c.ep_{1-ec}$)
 6  FreeCore($i.ep_{ei}$)
 7  FreeCore($i.ep_{1-ei}$)
 8  $a.ep_{ea} \leftarrow$ GetPrelabelCore()
 9  $a.ep_{1-ea} \leftarrow$ GetPrelabelCore()
10  $b.ep_{eb} \leftarrow$ GetPrelabelCore()
11  $c.ep_{ec} \leftarrow a.ep_{ea}$
12  $c.ep_{1-ec} \leftarrow b.ep_{eb}$
13  $i.ep_{ei} \leftarrow a.ep_{1-ea}$
14  $i.ep_{1-ei} \leftarrow b.ep_{1-eb}$
15  $b.nx_{eb} \leftarrow i$
16  $i.nx_{1-ei} \leftarrow c$
17  $c.nx_{1-ec} \leftarrow b$
18  $c.nx_{ec} \leftarrow a$
19  $a.nx_{ea} \leftarrow c$
20  $i.nx_{ei} \leftarrow a$
21  $a.nx_{1-ea} \leftarrow i$
22  $a.nn_{ea} \leftarrow 2$
23  $a.nn_{1-ea} \leftarrow 2$
24  $b.nn_{eb} \leftarrow 3$
25  $b.nn_{1-eb} \leftarrow b.nn_{1-eb}$
26  $c.nn_{ec} \leftarrow 2$
27  $c.nn_{1-ec} \leftarrow 3$
28  $i.nn_{ei} \leftarrow 2$
29  $i.nn_{1-ei} \leftarrow 3$
```

```
CaseI()
 1  FreeCore($a.ep_{ea}$)
 2  FreeCore($a.ep_{1-ea}$)
```

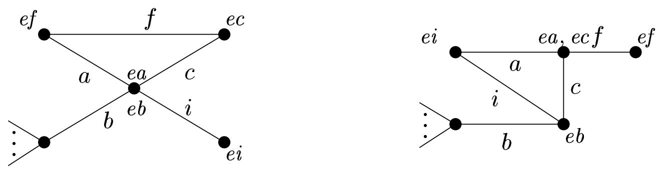


(a) G (b) G'

Figure 25: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case H of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseH.

```
 3  FreeCore($b.ep_{eb}$)
 4  FreeCore($c.ep_{ec}$)
 5  FreeCore($c.ep_{1\text{-}ec}$)
 6  FreeCore($f.ep_{ef}$)
 7  FreeCore($f.ep_{1\text{-}ef}$)
 8  FreeCore($i.ep_{ei}$)
 9  FreeCore($i.ep_{1\text{-}ei}$)
10  $a.ep_{ea} \leftarrow$ GetPrelabelCore()
11  $a.ep_{1\text{-}ea} \leftarrow$ GetPrelabelCore()
12  $b.ep_{eb} \leftarrow$ GetPrelabelCore()
13  $f.ep_{ef} \leftarrow$ GetPrelabelCore()
14  $c.ep_{ec} \leftarrow a.ep_{ea}$
15  $c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
16  $i.ep_{ei} \leftarrow a.ep_{1\text{-}ea}$
17  $i.ep_{1\text{-}ei} \leftarrow b.ep_{1\text{-}eb}$
18  $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
19  $b.nx_{eb} \leftarrow i$
20  $i.nx_{1\text{-}ei} \leftarrow c$
21  $c.nx_{1\text{-}ec} \leftarrow b$
22  $a.nx_{1\text{-}ea} \leftarrow i$
23  $i.nx_{ei} \leftarrow a$
24  $c.nx_{ec} \leftarrow a$
25  $a.nx_{ea} \leftarrow f$
26  $f.nx_{1\text{-}ef} \leftarrow c$
27  $f.nx_{ef} \leftarrow f$
28  $a.nn_{ea} \leftarrow 3$
29  $a.nn_{1\text{-}ea} \leftarrow 2$
30  $b.nn_{eb} \leftarrow 3$
31  $b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
32  $c.nn_{ec} \leftarrow 3$
33  $c.nn_{1\text{-}ec} \leftarrow 3$
34  $f.nn_{ef} \leftarrow 1$
35  $f.nn_{1\text{-}ef} \leftarrow 3$
36  $i.nn_{ei} \leftarrow 2$
37  $i.nn_{1\text{-}ei} \leftarrow 3$
```

```
CaseJ()
 1  FreeCore($a.ep_{ea}$)
 2  FreeCore($a.ep_{1\text{-}ea}$)
 3  FreeCore($b.ep_{eb}$)
 4  FreeCore($c.ep_{ec}$)
```



(a) G (b) G'

Figure 26: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case I of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseI.

```
 5  FreeCore($c.ep_{1\text{-}ec}$)
 6  FreeCore($f.ep_{ef}$)
 7  FreeCore($f.ep_{1\text{-}ef}$)
 8  FreeCore($g.ep_{eg}$)
 9  FreeCore($g.ep_{1\text{-}eg}$)
10  FreeCore($i.ep_{ei}$)
11  FreeCore($i.ep_{1\text{-}ei}$)
12  $a.ep_{ea} \leftarrow$ GetPrelabelCore()
13  $a.ep_{1\text{-}ea} \leftarrow$ GetPrelabelCore()
14  $b.ep_{eb} \leftarrow$ GetPrelabelCore()
15  $f.ep_{ef} \leftarrow$ GetPrelabelCore()
16  $c.ep_{ec} \leftarrow a.ep_{ea}$
17  $c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
18  $i.ep_{ei} \leftarrow a.ep_{1\text{-}ea}$
19  $i.ep_{1\text{-}ei} \leftarrow b.ep_{1\text{-}eb}$
20  $f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
21  $g.ep_{eg} \leftarrow f.ep_{ef}$
22  $g.ep_{1\text{-}eg} \leftarrow i.ep_{ei}$
23  $b.nx_{eb} \leftarrow i$
24  $i.nx_{1\text{-}ei} \leftarrow c$
25  $c.nx_{1\text{-}ec} \leftarrow b$
26  $c.nx_{ec} \leftarrow a$
27  $a.nx_{ea} \leftarrow f$
28  $f.nx_{1\text{-}ef} \leftarrow c$
29  $i.nx_{ei} \leftarrow g$
30  $g.nx_{1\text{-}eg} \leftarrow a$
31  $a.nx_{1\text{-}ea} \leftarrow i$
32  $g.nx_{eg} \leftarrow f$
33  $f.nx_{ef} \leftarrow g$
34  $a.nn_{ea} \leftarrow 3$
35  $a.nn_{1\text{-}ea} \leftarrow 3$
36  $b.nn_{eb} \leftarrow 3$
37  $b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
38  $c.nn_{ec} \leftarrow 3$
39  $c.nn_{1\text{-}ec} \leftarrow 3$
40  $f.nn_{ef} \leftarrow 2$
41  $f.nn_{1\text{-}ef} \leftarrow 3$
42  $g.nn_{eg} \leftarrow 2$
43  $g.nn_{1\text{-}eg} \leftarrow 3$
44  $i.nn_{ei} \leftarrow 3$
45  $i.nn_{1\text{-}ei} \leftarrow 3$
```

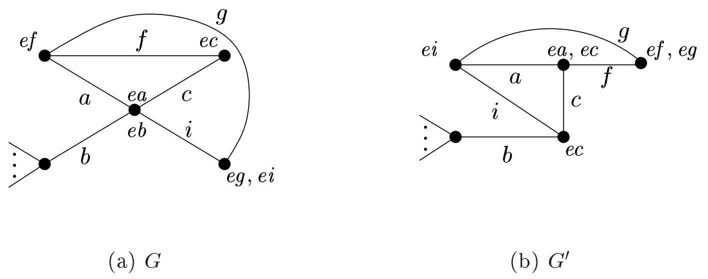


Figure 27: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case J of Table 1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseJ.

DeleteEdge

Input: a and b, the prelabels of endpoints of the edge to be deleted.

Output: None, however, DeleteEdge changes the labels of the present graph in order to produce a labelling of the modified graph.

1-5: We must first determine the endpoint in the core at which a and b intersect. In particular, we need to know the value of ea and eb, for which $a.ep_{ea} = b.ep_{eb}$. Later we will introduce the variables ec, ef, eg, eh, ei, ej, to denote particular endpoints of c, f, g, h, i, and j, respectively.

6-110: We determine the structure of the core graph around a and b through a series of case statements, these will allow us to determine which of the cases in Table 1 we must deal with.

7,8: Recall that $a.ep_{ea}$ is the endpoint of a at which a intersects with b. If the only edges of the core incident with $a.ep_{ea}$ are a and b themselves then we are dealing with either case A or case C. Both cases are handled by the function CaseAC.

9-62: There is exactly one additional edge incident with $a.ep_{ea}$ other than a and b themselves. We call this edge c and let ec be the value (0 or 1) for which $c.ep_{ec} \neq a.ep_{ea}$.

10-12: We determine which edge is c using the circular linked list at $a.ep_{ea}$.

13: As in the algorithm DeleteVertex we use the function FindEnd to determine the value of ec.

15,16: If c is the only edge incident with $c.ep_{ec}$, then we are dealing with either case B or case D. Both cases are handled by the function CaseBD.

17-50: There is exactly one other edge incident with $c.ep_{ec}$ other than c itself. We call this edge f and let ef be the value for which $f.ep_{ef} \neq c.ep_{ec}$. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with cases E or F, or symmetric variants thereof.

21-34: The edge f is incident with both c and a where, in particular, $f.ep_{ef} = a.ep_{1\text{-}ea}$. Again, the possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with cases E or F.

23: If there are more than three edges incident with $f.ep_{ef}$ then the modified graph is not a line graph.

25-32: There is exactly one other edge incident with $f.ep_{ef}$ other than a and f themselves. We call this edge g and let eg be the value for which $g.ep_{eg} \neq f.ep_{ef}$. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with case F.

30-32: If g is the only edge incident with $g.ep_{eg}$, then we are dealing with case F which is handled by the function CaseF. Otherwise, the modified graph is not a line graph.

33-34: If the only edges incident with $f.ep_{ef}$ are a and f themselves, then we are dealing with case E which is handled by the function CaseE.

35-48: The edge f is incident with both c and b where , in particular, $f.ep_{ef} = b.ep_{1\text{-}eb}$. This case is similar to lines 21 through 34.

49,50: The edge f is incident with c but neither of a nor b; consequently, the modified graph is not a line graph.

51-60: There are exactly two additional edges incident with $c.ep_{ec}$ other than c itself. If the modified graph is still a line graph, then the original core must have been as pictured in case G of Table 1.

52,53: The base graph is as pictured in case G of Table 1 only if $a.nn_{1\text{-}ea} = b.nn_{1\text{-}eb} = 2$. Otherwise, the modified graph is not a line graph.

54-57: Providing $a.nn_{1\text{-}ea} = b.nn_{1\text{-}eb} = 2$, we let f be the edge incident with $a.ep_{1\text{-}ea}$ other than a and we let h be the edge incident with $b.ep_{1\text{-}eb}$ other than b. Moreover, we let ef and eh be the values for which $f.ep_{ef} = a.ep_{1\text{-}ea}$ and $h.ep_{eh} = b.ep_{1\text{-}eb}$.

58-60: The core is as pictured in case G of Table 1 if and only if $f.ep_{1\text{-}ef} = h.ep_{1\text{-}eh} = c.ep_{ec}$. Providing this condition holds it is handled by the function CaseG.

61,62: If there are more than three edges incident with $c.ep_{ec}$ then the modified graph is not a line graph.

63-108: There are exactly two additional edges incident with $a.ep_{ea}$ other than a and b themselves. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with cases H, I, or J, or symmetric variants thereof.

64-73: We denote by c and i the edges incident with $a.ep_{ea}$ other than a and b themselves. The circular linked list at $a.ep_{ea}$ is used to determine c and i where, moreover, we denote by ec and ei the values for which $c.ep_{ec}, i.ep_{ei} \neq a.ep_{ea}$.

74-76: We require that $c.ep_{ec}$ and $i.ep_{ei}$ be incident with at most one edge other than c and i themselves, otherwise, the modified graph is not a line graph.

77-82: We first consider when $c.ep_{ec}$ and $i.ep_{ei}$ are incident only with c and i, respectively. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with case H or a symmetric variant thereof.

78-82: The modified graph is a line graph if and only if at least one of $a.nn_{1\text{-}ea} = 1$ or $b.nn_{1\text{-}eb} = 1$. If $a.nn_{1\text{-}ea} = 1$, then we are dealing with case H which is handled by the function CaseH. If $a.nn_{1\text{-}ea} > 1$, but $b.nn_{1\text{-}eb} = 1$, then the situation is symmetric to case H.

83-92: We now consider when $c.ep_{ec}$ and $i.ep_{ei}$ are both incident with exactly one edge in addition to c and i, respectively. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with case J or a symmetric variant thereof.

84-87: We denote by f and g the edges incident with c and i, respectively, other than c and i themselves. The circular linked lists at $c.ep_{ec}$ and $i.ep_{ei}$ are used to determine f and g where, moreover, we let ef and eg be the values for which $f.ep_{ef} \neq c.ep_{ec}$ and $g.ep_{eg} = i.ep_{ei}$.

88-92: The modified graph is a line graph if and only if either $f.ep_{ef} = g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$, where $f.nn_{ef} = 3$, or $f.ep_{ef} = g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$, where $f.nn_{ef} = 3$. If the latter holds then we are dealing with Case J which is handled by the function CaseJ. If the latter does not hold but the former does, then the situation is symmetric to Case J.

93-100: We now consider when $i.ep_{ei}$ is incident only with i itself and $c.ep_{ec}$ is incident with exactly one edge other than c. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with case I or a symmetric variant thereof.

94,95: We denote by f the edge incident with c, other than c itself. The circular linked list at $c.ep_{ec}$ is used to determine f where ef is the value for which $f.ep_{ef} \neq c.ep_{ec}$.

96-100: The modified graph is a line graph if and only if either $f.ep_{ef} = a.ep_{1\text{-}ea}$, where $a.nn_{1\text{-}ea} = 2$, or $f.ep_{ef} = b.ep_{1\text{-}eb}$, where $b.nn_{1\text{-}eb} = 2$. If the latter holds then we are dealing with Case I which is handled by the function CaseI. If the latter does not hold but the former does then the situation is symmetric to Case I.

101-108: We now consider when $c.ep_{ec}$ is incident only with c itself and $i.ep_{ei}$ is incident with exactly one edge other than i. This case is similar to that found in lines 93 through 100.

109,110: Finally, we consider when there are exactly at least three additional edges incident with $a.ep_{ea}$ other than a and b themselves. In this case the modified graph is not a line graph.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo, *Compact labeling schemes for ancestor queries*, Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (Washington, D.C., USA), 2001, pp. 547–556.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe, *Nearest common ancestors: a survey and a new distributed algorithm*, Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (Winnipeg, Canada), 2002, pp. 258–264.

[3] S. Alstrup and T. Rauhe, *Small induced-universal graphs and compact implicit graph representations*, 43rd Annual Symposium on Foundations of Computer Science (Vancouver, Canada), IEEE, 2002, pp. 53–62.

[4] S. R. Arikati, A. Maheshwari, and C. Zaroliagis, *Efficient computation of implicit representations of sparse graphs*, Discrete Applied Mathematics **78** (1997), no. 1, 1–16.

[5] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg, *Compact distributed data structures for adaptive routing*, Proceedings of the 21st Annual ACM Symposium on Theory of Computing (Seattle, USA), 1989, pp. 467–478.

[6] A. Branstädt, V. B. Le, and J. P. Spinrad, *Graph classes: A survey*, SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia, 1999.

[7] M. A. Breuer, *Coding the vertexes of a graph*, IEEE Transactions on Information Theory **12** (1966), 148–153.

[8] M. A. Breuer and J. Folkman, *An unexpected result in coding the vertices of a graph*, Journal of Mathematical Analysis and Applications **20** (1967), 583–600.

[9] G. S. Brodal and R. Fagerberg, *Dynamic representation of sparse graphs*, Algorithms and Data Structures, Proceedings of the 6th International Workshop (Vancouver, Canada), Lecture Notes in Computer Science, vol. 1663, Springer-Verlag, 1999, pp. 342–351.

[10] W. R. Franklin, *Compressing elevation data*, Advances in Spatial Databases, Proceedings of the 4th International Symposium (Portland, USA), Lecture Notes in Computer Science, vol. 951, Springer-Verlag, 1995, pp. 385–404.

[11] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz, *Distance labeling in graphs*, Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (Washington, D.C., USA), 2001, pp. 210–219.

[12] S. Kannan, M. Naor, and S. Rudich, *Implicit representation of graphs*, SIAM Journal on Discrete Mathematics **5** (1992), no. 4, 596–603.

[13] H. Kaplan and T. Milo, *Short and simple labels for small distances and other functions*, Algorithms and Data Structures, Proceedings of the 7th International Workshop (Providence, USA), Lecture Notes in Computer Science, vol. 2125, Springer-Verlag, 2001, pp. 246–257.

[14] P. G. H. Lehot, *An optimal algorithm to detect a line graph and output its root graph*, Journal of the Association of Computing Machines **21** (1974), no. 4, 569–575.

[15] J. H. Muller, *Local structure in graph classes*, Ph.D. thesis, Georgia Institute of Technology, March 1988.

[16] M. Naor, *Succinct representation of general unlabeled graphs*, Discrete Applied Mathematics **28** (1990), 303–307.

[17] D. Peleg, *Proximity-preserving labeling schemes and their applications*, Graph Theoretic Concepts in Computer Science, Proceedings of the 25th International Workshop (Ascona, Switzerland), Lecture Notes in Computer Science, vol. 1665, Springer-Verlag, 1999, pp. 30–41.

[18] ———, *Informative labeling schemes for graphs*, Mathematical Foundations of Computer Science 2000, Proceedings of the 25th International Symposium (Bratislava, Slovakia), Lecture Notes in Computer Science, vol. 1893, Springer-Verlag, 2000, pp. 579–588.

[19] N. D. Roussopoulos, *A $max\{m,n\}$ algorithm for determining the graph H from its line graph g*, Information Processing Letters **2** (1973), 108–112.

[20] N. Santoro and R. Khatib, *Labelling and implicit routing in networks*, The Computer Journal **28** (1985), 5–8.

[21] E. R. Scheinerman, *Local representations using very short labels*, Discrete Mathematics **203** (1999), 287–290.

[22] J. Spinrad, *Efficient graph representation*, Fields Institute Monographs, AMS, Providence, 2003.

[23] M. Thorup and U. Zwick, *Compact routing schemes*, Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (Heraklion, Greece), 2001, pp. 1–10.

[24] H. Whitney, *Congruent graphs and the connectivity of graphs*, American Journal of Mathematics **54** (1932), 150–168.