Using Bounded Degree Spanning Trees in the Design of Efficient Algorithms on Claw Free Graphs

Marek Chrobak*
Joseph Naor**
Mark B. Novick***

TR 89-975
March 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Department of Mathematics and CS, University of California
**Supported by contract ONR N00014-88-K-0166.
***Research supported by NSF grant CCS-8806979.

Using Bounded Degree Spanning Trees in the Design of Efficient Algorithms on Claw Free Graphs

Marek Chrobak * Joseph Naor † Mark B. Novick ‡

Abstract

Claw-free graphs are graphs that do not have $K_{1,3}$ as an induced subgraph. Line graphs, a special case of claw-free graphs, are the intersection graphs of edges in simple graphs. We show how to compute efficiently in parallel a binary tree that will be a rooted spanning tree of the claw-free graph. Every binary tree contains at least one edge whose removal partitions the tree into two subtrees of nearly equal cardinality, and this separator can be found efficiently in parallel. We solve problems on claw-free graphs by a divide-and-conquer strategy. The advantage of our partition is that the vertices in each set induce a connected subgraph. The problems are solved for the two subgraphs, and then the results are combined to get a solution for the entire graph.

Both the problem of finding a perfect matching in claw-free graphs and the problem of reconstructing a root graph from a line graph are amenable to this approach. We present a nearly optimal parallel NC algorithm for computing a perfect matching that runs in time $O(\log^2 n)$ with $O(n+m)$ processors on an EREW PRAM. Also, we present an efficient parallel reconstruction of root graphs from line graphs. If $G = (V, E)$ denotes a line graph, then the algorithm runs in $O(\log |V|)$ time using $O(|E|)$ processors in the CRCW PRAM model. It is optimal up to a polylogarithmic factor. Previously, it was known how to reconstruct the root graph in NC using a large (though polynomial) number of processors; this is the first algorithm that employs a linear number of processors.

*Department of Mathematics and Computer Science, University of California, Riverside, CA 92521

†Computer Science Department, Stanford University, Stanford, CA 94305-2140. Supported by contract ONR N00014-88-K-0166.

‡Computer Science Department, Cornell University, Ithaca, NY 14853-7501. Supported by NSF grant CCS-8806979.

1 Introduction

We introduce a new technique for solving graph problems which is based on the existence of bounded degree spanning trees in special classes of graphs, such as claw-free graphs, i.e., graphs that do not have $K_{1,3}$ as an induced subgraph. Our method can be used in the design of efficient parallel algorithms for graphs in these classes.

A line graph, $L(G) = (V_l, E_l)$, of a graph $G = (V_r, E_r)$ has $V_l = E_r$ as its vertex set and two vertices are adjacent if their corresponding edges in G are adjacent. The graph G is also called the root graph of $L(G)$. This concept was introduced by Whitney [Wh] and it was later proved by various researchers that a line graph can also be characterized by a set of nine forbidden induced subgraphs (see [Ha] for details). One of the forbidden subgraphs is the claw and hence, line graphs are also claw-free graphs. Whitney also proved that if $L(G_1)$ and $L(G_2)$ are isomorphic and connected, then G_1 and G_2 are also isomorphic except for one case, the triangle graph.

The class of claw-free graphs has been studied in the literature, and it is known that some problems which are computationally hard for general graphs can be solved fast on claw-free graphs. The best-known result in this area is that a maximum independent set in these graphs can be found in polynomial time. This fact was shown independently by Minty [Mi] and Sbihi [Sb]. The maximum matching problem in a general graph corresponds to the maximum independent set problem in line graphs, and therefore the latter result can be looked upon as a generalization of Edmonds' celebrated result [Ed]. On the other hand, there are many computational problems that remain NP-complete even when restricted to claw-free graphs, e.g., computing the largest clique, (see [Jo1,Jo2]). This demonstrates that this class of graphs is non-trivial and has a rich combinatorial structure.

The two applications in which we are interested in this paper are the following:

- Computing a perfect matching in a claw-free graph. Las Vergnas [La] and Sumner [Su], independently proved that a claw-free graph always has a perfect matching (with a slight abuse of terminology, if a graph has an odd number n of vertices, a matching is called perfect if it covers $n - 1$ vertices). Our goal is to design efficient parallel algorithms for computing this matching. We present a parallel EREW PRAM algorithm for finding a perfect matching in claw-free graphs. This algorithm is nearly optimal: it runs in time $O(\log^2 n)$ and uses $O(n+m)$ processors. This improves a result of Naor [Na] who presented an NC algorithm using $O(n^3)$ processors to compute a perfect matching in a line graph.

- Reconstructing the root graph of a given line graph, or determining whether a given graph is not a line graph. We present an efficient NC algorithm that uses $O(|E_l|)$ processors and its running time is $O(\log |V_l|)$ in the CRCW model. Sequentially, this problem can be solved in linear time [Le,Ro] and hence our algorithm is optimal up to a polylogarithmic factor. In principal, the reconstruction problem can be solved in NC quite easily by brute force methods if the number of processors available is large enough, yet bounded by a polynomial. Our algorithm is the first one that employs a linear number of processors.

As already stated, our methods rely on computing a bounded degree spanning tree. It turns out that a DFS tree in a claw-free graph must be a rooted binary tree, and hence of bounded degree. We do not know how to compute a DFS tree in parallel, even in a line graph. However, we can efficiently transform any spanning tree of a claw-free graph into a suitable bounded degree spanning tree. In particular, we can obtain the desired tree in $O(\log n)$ time by using $O(n+m)$ processors on a CRCW PRAM.

Our algorithms employ a divide-and-conquer scheme. The vertex set of the graph is split into two sets such that each induces a connected graph. This is done by finding an edge in the bounded degree spanning tree whose removal results in two trees of nearly equal size. In order to improve the efficiency of our algorithms, we precompute the set of separating edges used in the recursion. We split the vertex set of the graph into the vertex sets of these two subtrees. The algorithm recurses on the subgraphs induced by these two vertex sets. Finally, we combine the results from the subgraphs to get the answer in the original graph. This technique is applied both to compute a perfect matching in claw free graphs and to reconstruct the root graph of a line graph.

As already mentioned, our first application is computing a perfect matching in claw-free graphs. Computing a maximum matching in a graph has recently attracted much interest because of its parallel computational difficulty. An outstanding open question is whether the computation of a maximum matching is in the complexity class NC. The current approaches to this problem involve either randomization [KUW,MVV], or a restriction of the problem to special families of graphs such as bipartite planar graphs [MN], or if the number of perfect matchings is bounded by a polynomial ([GK] and see also [BT]).

The second application with which we are concerned is reconstructing the root graph of a given line graph. The line graph and the root graph may be viewed as different representations of a given connected graph. Computing the line graph of a graph is easy, both sequentially and in parallel. In this paper we are concerned with the parallel complexity of the inverse transformation, namely reconstructing the root graph of a given line graph. Optimizing the transfer cost from one representation of a combinatorial object to another is a typical computer science problem.

Note that if the vertex set of a line graph is divided into two arbitrary sets that not necessarily induce a connected graph, then in general, the root graph of each set is not uniquely defined [Ha]. This would result in an exponential blowup in terms of the number of processors. In our case, the root graph of each set is recursively computed (it is uniquely defined), and the two root graphs are merged using a complicated case analysis.

The sequential complexity of computing the root graph has been studied in the literature, and algorithms whose running time is $O(|E_l|)$ were given by [Le, Ro]. These results do not seem to lend themselves to efficient parallelization in terms of the number of processors.

2 Computing the bounded degree spanning tree

Let T be a spanning tree in a graph $G = (V, E)$. We denote the parent of a vertex v in T by $f_T(v)$, or simply $f(v)$ if it does not lead to a confusion. By $T(v)$ we denote the subtree of T with root v. Let the level of a vertex in a tree be its distance from the root. If G is claw-free, and T is a DFS tree, then there are no edges between any sons of a vertex in T, and we obtain easily that T is a binary tree. Unfortunately, we do not know how to compute a DFS tree efficiently in parallel, even for claw-free graphs. However, as it turns out, it is possible to construct a bounded-degree spanning tree without using DFS.

We first compute any spanning tree D and label its vertices by their depth. This tree is computed by the algorithm of [SV] that runs in $O(\log n)$ time with $O(n)$ processors. The tree is then converted into another spanning tree D'. Each vertex u chooses as its parent $f_{D'}(u)$ in D' the highest ascendant v of u in D which is adjacent to u. Since in D' a vertex u cannot be adjacent to its grandparent, we have the following.

Proposition 2.1 *The children of every vertex $v \in D'$ (except for the root) induce a clique in G.*

The tree D' is transformed into a bounded-degree spanning tree T as follows:

1. Let $v_1, \ldots, v_k$ be an arbitrary permutation of the children of a vertex v. We do the following for each v: for $1 \leq j < k$, $f(v_i) \leftarrow v_{i+1}$. Also $f(v_k) \leftarrow f_{D'}(v_k)$, that is, the parent of v_k does not change.
2. for $i = 0$ to $\log^* n$ do
   (a) for $j = 1$ to k let L_j be the children of vertices $v_1, \ldots, v_k$ in level i with degree greater than 2.
   (b) Repeat (c) and (d) until for all j, $|deg(v_j)| = 2$:
   (c) Divide each set L_j arbitrarily into $\lfloor |L_j|/3 \rfloor$ triples, and one set which has $|L_j| \bmod 3$ elements (if non-empty).
   (d) For each triple $\{x, y, z\}$, at least two of its vertices are adjacent and without loss of generality, suppose they are x and y. Then, $f(x) \leftarrow y$; $L_j \leftarrow L_j \setminus \{x\}$.

Proposition 2.2 *There is an edge e that divides T into two trees of at most $\lceil 2n/3 \rceil$ vertices each.*

To compute such an edge e, every edge has to know the size of the two tree components that are generated by its deletion. One way of doing it is by the method of parallel tree contraction of Miller and Reif [MR]. To implement that, T is rooted and each vertex counts the number of vertices in the subtree beneath it. When this information is available, the set of separators can be precomputed in $O(\log n)$ time on a EREW PRAM.

3 The matching algorithm

3.1 Terminology and theoretical background

Given a matching M, a vertex v is called exposed if none of the edges adjacent to it is in M. M is perfect iff there is at most one exposed vertex. An alternating path is a path whose edges alternate between M and $E \setminus M$ and that starts and ends at edges in $E \setminus M$. We show that, for claw-free graphs, any path between exposed vertices can be transformed into an alternating path.

Theorem 3.1 [La,Su] *A connected claw-free graph G has a perfect matching.*

Proof: Let M be any matching in G. If M is not perfect, then there are at least two exposed vertices u and v. We extend M to a matching M' that covers u and v as well.

Let $P : u = v_0, v_1, \ldots, v_k = v$ be a path from u to v. If P is an alternating path then M can be extend to M' by taking $M' := (M \setminus P) \cup (P \setminus M)$ (somewhat informally, in this formula we denote by P the set of edges on P).

Suppose that P is not alternating. Without loss of generality, we can assume that all the vertices in P are matched and that P does not have chords (edges (v_i, v_j), for $j > i + 1$). There are some vertices in P that are matched to vertices outside the path. Let us call these vertices alien. We show how to reduce the number of alien vertices to zero, and obtain in this way an alternating path.

Take any alien vertex v_j from P, which is matched to a vertex $x \neq v_{j-1}, v_{j+1}$. Since P does not have chords, $(v_{j-1}, v_{j+1}) \notin E$. This, in turn, implies that either $(v_{j-1}, x) \in E$ or $(v_{j+1}, x) \in E$. In the first case we set $P := v_0, \ldots, v_{j-1}, x, v_j, \ldots, v_k$. In the second case, $P := v_0, \ldots, v_j, x, v_{j+1}, \ldots, v_k$. In both cases the number of alien vertices is decreased by one. ∎

3.2 Overview of the algorithm

We first compute a bounded degree spanning tree in G. In such a tree, there exists an edge that separates the tree into two components of linear size. Each component induces a connected claw-free graph and recursively a maximum matching (M_1 and M_2) is computed in each one. $M_1 \cup M_2$ can be augmented by at most one edge in the graph G.

It is important to divide the vertices into two components through the tree and not in an arbitrary fashion, for two reasons: (i) Each component in the tree is a connected claw-free graph and by Theorem 3.1 contains a perfect matching, and (ii) When the solutions for two components are being combined, a path between the two exposed vertices has to be computed. If we have a spanning tree, expensive search for such a path can be avoided.

The algorithm looks as follows:

1. Compute in G a spanning tree T of maximum degree 3, as explained below.

2. Let e be an edge in T that separates it into two parts T_1 and T_2 of at most $\lceil 2n/3 \rceil$ vertices each. Compute recursively a maximum matching in the graphs induced by the vertices in T_1 and T_2.

3. If both T_1 and T_2 contain exposed vertices, then augment the matching.

3.3 Augmenting the matching

Consider the path $P : v = v_0, \ldots, v_{k+1} = w$ in T, from v to w. We eliminate alien vertices from this path in two rounds to avoid conflicts. The algorithm below is due to Xin He [He] and it improves the original method of the authors which runs in time $O(\log^2 n)$.

begin
  transform P into a chord-free path;
  label the vertices on P as odd or even according to
    their distance from v;
  for each even i, if v_i is alien **pardo**
  begin
    let $(v_i, x) \in M$;
    let $j \in \{i-1, i+1\}$ be such that $(v_j, x) \in E$;
    $y := v_j$
    $P := P \setminus \{(y, v_i)\} \cup \{(y, x), (v_i, x)\}$
  end
  for each odd i, if v_i is alien **pardo**
  begin
    let $(u, v_i), (v_i, u') \in P$;
    if $(u, u') \in E$ **then** $P := P \setminus \{v_i\} \cup \{(u, u')\}$
    else begin
      let $(v_i, x) \in M$;
      let $y \in \{u, u'\}$ be such that $(y, x) \in E$;
      $P := P \setminus \{(y, v_i)\} \cup \{(y, x), (v_i, x)\}$
    end
  end;
$M := (M \setminus P) \cup (P \setminus M)$
end.

3.4 Analysis of algorithm

Theorem 3.2 *The algorithm above finds a perfect matching in a connected claw-free graph. It runs in time $O(\log^2 n)$ using $O(n+m)$ processors on an EREW PRAM.*

Proof: The number of phases of the algorithm is $O(\log n)$ because the subgraphs at each phase have $\lceil 2n/3 \rceil$ vertices at most. Therefore, it remains to show that the running time of one phase is $O(\log n)$.

Step 1, the construction of a bounded-degree spanning tree, costs time $O(\log^2 n)$, using the method described at the beginning of this section since anything a CRCW PRAM can do an EREW PRAM can do with $O(\log n)$ times as much time. Step 1 is only performed once. In Step 2, the graph is divided into two connected subgraphs of size at most $\lceil 2n/3 \rceil$ each. In order to do that, the spanning tree T is used.

Let us analyze the last part, augmenting the matching. To transform P into a chordless path, we use a similar trick to the one in the construction of T: each vertex u on P chooses as its descendant the neighbour closest to w. This yields a tree S rooted at w. P is set to be the path from v to w in S. The correctness of the procedure which eliminates alien vertices can be proved as follows. First, the vertices which are in the matching remain in P. If an alien vertex v was even, it will not be alien in the second phase. Also, the vertices addded to a path in the first phase are not alien. So all vertices which are alien at the beginning of the first phase were labelled odd. After the second phase they will not be alien anymore. Eliminating the alien vertices clearly takes $O(\log n)$ time with a linear number of processors and therefore augmenting a path takes $O(\log n)$ time using $O(n + m)$ processors assigned to vertices and edges. So the total time complexity of the whole algorithm is $O(\log^2 n)$. ∎

4 The reconstruction algorithm

4.1 Terminology and theoretical background

Let $G = (V, E)$ denote an undirected graph. Let the line graph of G be denoted by $L(G)$, and its root graph (if it exists) by $R(G)$. Line graphs are characterized by a set of nine forbidden subgraphs [Ha] where each contains at most six vertices. One of them is the graph $K_{1,3}$, also called a claw or a star.

Property 4.1 [Ha] *An induced subgraph H of a line graph G is also a line graph.*

Property 4.2 [Wh] *Let G and H be connected and isomorphic line graphs that contain more than three vertices. Then $R(G)$ and $R(H)$ are also isomorphic.*

Let U be a subset of V; The induced subgraph defined by U is denoted by G_U.

4.2 Overview

For simplicity we assume that G is indeed a line graph as long as this assumption is not contradicted. An overall view of the algorithm is the following:

1. If $|V| < 14$ then compute $R(G)$. Otherwise, partition V into disjoint sets U and W such that:
   (a) The graphs G_U and G_W are connected.
   (b) $|W| \approx |U|$ up to a constant factor.
2. Compute $R(G_U)$ and $R(G_W)$ recursively.
3. Merge $R(G_U)$ and $R(G_W)$ to $R(G)$.
4. Compute $L(R(G))$ and compare it with G. If they differ, then G is not a line graph.

We now elaborate on the steps of the algorithm. In Step 1, if G contains fewer than 14 vertices, then we can run a sequential algorithm to find $R(G)$ (or discover that G is not a line graph). Otherwise, we break up G into G_U and G_W so that each of these subgraphs contains at least five vertices. The vertices are divided into the two sets by computing a bounded degree spanning tree in G by using the algorithm given in section 2. Furthermore, by using the methods of that section we can precompute the set of edge separators necessary in Step 1.

In Step 2, the root graphs of G_U and G_W are recursively computed, and each vertex in $R(G_U)$ and $R(G_W)$ receives a different label. In Step 3, the root graphs are merged via a complicated case analysis. We decide for each vertex in $R(G_U)$ whether to identify it with some other vertex in $R(G_W)$ and vice versa. The vertices that are not identified keep their old label.

In Step 4 we check whether the solution that we arrived at Step 3 is consistent with G. This is necessary because not all the edges in G_U and G_W were examined in the previous steps. It turns out that each vertex in $R(G_U)$ has to examine at most three of its neighbors in order to decide whether to identify itself with some other vertex in $R(G_W)$.

4.3 Merging the root graphs

We now show how to merge the root graphs $R(G_U)$ and $R(G_W)$ computed in Step 2 of the reconstruction algorithm. The subgraphs G_U and G_W are connected because they each have a connected spanning tree, namely their half of the degree bounded spanning tree. This property is important as it insures us that $R(G_U)$ and $R(G_W)$ are uniquely defined [Ha] and we take advantage of it in the merging step. The vertices of G_U will be labelled by pairs of letters from $\{a, b, c, \ldots\}$. For example, the pair ab corresponds to an edge between vertices a and b in the root graph of G_U. Similarly, the vertices of G_W will be labelled by pairs of letters from $\{z, y, x, \ldots\}$.

In merging $R(G_U)$ and $R(G_W)$, we decide which vertices from the two root graphs should be identified. Vertices that are identified receive the same label. When the merging is completed, a vertex that was not identified with some other vertex, remains in $R(G)$ with its original label. Initially, the vertices of $R(G_U)$ and $R(G_W)$ are labelled when Step 2 is completed.

Proposition 4.1 *A vertex of $R(G_U)$ can be identified with at most one vertex of $R(G_W)$, and vice versa.*

Proposition 4.2 *There is a unique way to identify the vertices of the root graphs $R(G_U)$ and $R(G_W)$.*

The following case analysis is performed for each vertex $a \in R(G_U)$ (and symmetrically for each vertex in $R(G_W)$). It is applicable only for vertices $a \in R(G_U)$ such that for some there exists an edge $xy \in R(G_W)$ which for all $b \in R(G_U)$ adjacent to a satisfies ab is adjacent to xy in G. If a does not obey this property, then there is no vertex in $R(G_W)$ with which it can be identified.

Case A: Vertex a has degree at least three in $R(G_U)$. There are vertices ab, ac, ad, etc. in G_U. Choose three of these vertices and call them ab, ac, and ad. Call this set of vertices A.

1. Vertex xy is adjacent to all the vertices in A. However, vertex xz is not adjacent to at least three of these vertices of G_U.

Because xz is not adjacent to every vertex in A, x and a must be different vertices in $R(G)$. We do identify a and y as the same vertex in $R(G)$ since this is the only way xy can be adjacent to every vertex containing a.

2. Vertex xy is adjacent to all vertices in A, and xz is adjacent to only one of them, namely ab.

   In this case, we still identify a with y. Also z and b are identified so that xz is adjacent to ab.

3. Vertex xy is adjacent to all vertices in A, and xz is adjacent to exactly two of these vertices, namely ab and ac.

   This graph contains an induced subgraph (see G_3 in [Ha], page 75) that is not a line graph. Therefore, G is not a line graph.

4. Vertices xy and xz are both adjacent to all vertices containing a.

   Then a and x are identified. Otherwise one of the vertices (in G_U) containing a would not be adjacent to both xy and xz.

Case B: Vertex a has degree exactly two in $R(G_U)$. The edges containing a in $R(G_U)$ are ab and ac.

1. Vertices xy and xz of G_W are adjacent to both ab and ac.

   In this case, a and x are identified. The other vertices of the root graphs are not identified.

2. Vertex xy is adjacent to both ab and ac, but xz is adjacent to neither of them.

   Vertex x cannot be identified with a, b, or c. Thus y must be identified with a.

3. Vertex xy is adjacent to both ab and ac, but xz is adjacent to ab, and not ac. The case where xy is still adjacent to both vertices of G_U and xz is adjacent to ac and not ab proceeds symmetrically.

   This case is not completely resolved without more information. One possibility is that we identify a and y, and b and z, while leaving c and x unidentified. The other possibility is that we identify b with x, and c with y, while not identifying a or z. In the former case, ab, ac, and xy form a claw, but in the latter case these vertices form a triangle in $R(G)$.

   We distinguish between these two cases by looking at a larger portion of G_W. One of x, y, or z must be in another vertex in G_W because G_W is a connected graph with more than three vertices. There is another vertex u in $R(G_W)$ which is a neighbor of at least one of these three vertices. We now classify according to which of the three vertices is adjacent to u.

   (a) The adjacent vertex is x.

      i. Vertex xu is adjacent to neither ab nor ac.
      We identify a with y, and b with z. Don't identify x with a vertex of G_U.

ii. Vertex xu is adjacent to ab, but not to ac.
We have a triangle. Pair b with x, and c with y. Don't identify any of the other vertices.

iii. Vertex xu is adjacent to ac, but not to ab.
This time we have a claw. Match a with y, b with z, and c with u.

iv. Vertex xu is adjacent to both ab and ac.
The graph is not a line graph. We are not in the claw case because xu would not be adjacent to ab then. Likewise, we are not in the triangle case because ac adjacent to xu would mean that a and u would be identified. The only possibility is to identify a with u; this is impossible as it implies that $ac \in G_U$ and $uy \in G_W$ are the same vertex. In fact, this graph also contains an induced subgraph (see G_3 in [Ha], page 75) that is not a line graph.

(b) The adjacent vertex is y.

i. Vertex yu is not adjacent to ac. It does not matter whether yu is adjacent to ab. This is impossible if G is a line graph because it contradicts both the triangle and the claw cases. The vertices xy, ac, yu, and xz form a claw with xy at the center of the claw.

ii. Vertex yu is adjacent to ac, and not ab.
The triangle case applies here. We use the claw case because yu is not adjacent to ab.

iii. Vertex yu is adjacent to both ab and ac.
Now the claw case is used. We don't use the triangle case because yu is adjacent to ac.

(c) The adjacent vertex is z. Without loss of generality we may assume that neither x nor y is adjacent to another vertex in $R(G_W)$ because otherwise it could be handled above. We can have the vertex yz in G_W, but since there are at least five vertices in G_W there must be another vertex unaccounted for.

i. Vertex zu is not adjacent to neither ab nor ac.
We are in the triangle case. The claw case does not apply because ab is not adjacent to zu.

ii. Vertex zu is adjacent to ac but not to ab.
The graph is not a line graph. It contains an induced subgraph (see G_2 in [Ha], page 75) that is not a line graph.

iii. Vertex zu is adjacent to ab but not to ac.
The only possibility is the claw case where a is identified with y and b with z.

iv. Vertex zu is adjacent to both ab and ac.
This is the last, but toughest part of our case analysis. If we are in the claw case, then we identify a with y, b with z, and c with u. On the other hand, if the triangle case applies, then identify a with u, b with x, and c with y. One of z or u will be adjacent to another vertex v in G_W. We can relabel the vertices of $R(G_W)$ by interchanging x with z and y with u. Then apply the preceding case analysis. This way we completely resolve the identification of vertices from G_U and G_W.

Case C: Vertex a has degree exactly one in $R(G_U)$. The edge containing a in $R(G_U)$ is ab.

Vertex ab in G_U is adjacent to vertex xy in G_W. If x has degree greater than one in $R(G_W)$ then it will take care of this case. Otherwise, first merge all the vertices whose degree is greater than one. If b is not identified with x or y, and a is not identified with y, then the two vertices of degree one, a and x, must be identified.

4.4 Proof of correctness

Theorem 4.1 *The reconstruction algorithm computes $R(G)$ or determines that G is not a line graph.*

Proof: Property 4.2 insures that $R(G_U)$ and $R(G_W)$ are well-defined. The case analysis described in the previous section is exhaustive and is performed for both $R(G_U)$ and $R(G_W)$. Notice that in all the cases, either we discover that G is not a line graph, or we obtain a **unique** solution for identifying vertices from $R(G_U)$ and $R(G_W)$, although at most three neighbors of vertices from G_U are examined, even if the degree is larger than three. One may assume that if a chose different neighbors (which may also have different neighbors in G_W), then a different identification scheme would result. This contradicts Proposition 4.2. It can be checked that if this last assumption were true, then G must contain one of the forbidden subgraphs. Hence, if the solution is inconsistent with the other neighbors of a, then G is not a line graph and this will be discovered in Step 4 of the algorithm.

By processing all graphs with fewer than 14 vertices in the base case of the recursion, we insure that the case analysis applies for all larger graphs. In case B.3.c.iv, we need at least five vertices in G_W. This is the largest number of vertices required for any of the cases. Therefore, each half of the graph must contain at least five vertices. In general, the smaller half of the separated graph must contain at least $\lceil (n-1)/3 \rceil$ vertices. Choosing $n = 14$ guarantees the existence of a suitable edge separator.

During the course of the algorithm, we may discover that G is not a line graph if our assumptions are violated. For example, this could happen if the degree of the spanning tree that was computed is not bounded, or a forbidden subgraph was discovered when merging. Also we need to check that there is at most one edge between two vertices of the root graph, because we are only looking at the line graphs of simple graphs. Finally, we check that no vertex has been identified with two different vertices, and we also compute the line graph of the graph we claim to be $R(G)$, and compare it with G itself. If these graphs differ, then G is not a line graph. Step 4 can be easily implemented if we maintain a correspondence between the edges in $R(G)$ and the vertices in G. This information can be obtained during the merging step and is passed upwards in the recursion tree. ∎

In the line graph that we get as input, let n and m respectively denote the vertex and edge cardinality.

Theorem 4.2 *The running time of the algorithm is $O(\log n)$ when $O(m)$ processors are available in the CRCW PRAM model.*

Proof: The number of phases of the algorithm is $O(\log n)$ because the subgraphs at each phase have $\lceil 2n/3 \rceil$ vertices at most. In section 2 we explained how to find the bounded degree spanning tree and precompute the set of separators in $O(\log n)$ time and $O(n)$ processors on a CRCW PRAM. Therefore, it remains to show that each merging step can be implemented in constant time and $O(m)$ processors.

A careful examination of the case analysis will reveal that each edge is examined at most twice. If the degree of a vertex $a \in R(G_U)$ is more than two, then b, c and d are the first three vertices appearing in the adjacency list of a. So we can assume that the degree of a is at most three. To find vertices x, y and z, the most expensive operation performed is computing the intersection of three sets, namely the neighborhood sets. This can be done in time logarithmic in their size by merging the sets and throwing out everything that is not duplicated. We can reduce this to constant time if we use an adjacency matrix. For each vertex $ab \in G_U$ we keep a matrix telling us what vertices in G_W are adjacent to ab. Suppose b, c, and d are the vertices adjacent to a in $R(G_U)$. Also suppose the label of b is less than that of c or d. These labels can be integers with $\log n + O(1)$ bits. In parallel, using one processor for each edge of the graph, we set matrix entry (ab, xy) to 0 if ab, ac, or ad is adjacent to xy. Next, in parallel, we increment the matrix entry (ab, xy) by 1 if ab is adjacent to xy. Similarly we increment (ab, xy) by 1 if ac is adjacent to xy, and we also do the same for ad. Vertex xy is in all three neighborhood sets if and only if matrix entry (ab, xy) contains 3, and furthermore, there is a processor assigned to (ab, xy). Each vertex in the intersection tries to write its name in the same space. Only one of them can succeed. Include it in the intersection set, set its entry to 0, and repeat at most twice so that we get either all the neighboring vertices or at least three of them. This process takes constant time using $O(m)$ processors, although it uses $O(n^2)$ space. We also delay comparing $L(R(G))$ with G until we have finished the recursion. This way this $O(\log n)$ step is done only once. The case analysis can still proceed without this check being done at every step as long as we make sure that no two vertices in some case receive the same label. We claim that each neighborhood set will not be scanned more than once. Assume for example that vertex $b \in R(G_U)$ is adjacent to both $a_1, a_2 \in R(G_U)$. The neighborhood set of the vertex $a_1 b \in G_U$ will be scanned once for the case analysis of a_1, and the neighborhood set of $a_2 b \in G_U$ will also be scanned only once for the case analysis of a_2. Hence, each edge is examined at most twice and these edges and vertices can be quickly retrieved if G and $R(G)$ are represented by adjacency lists. The same procedure is used on G_W with respect to G_U. ∎

Acknowledgment

We would like to thank Xin He for improving the time complexity of the algorithm that eliminates alien vertices, and for allowing us to include it in this paper. We would also like to thank Samir Khuller for pointing out an error in the original algorithm for computing the bounded degree spanning tree and helping to correct it.

References

[BT] M.Ben-Or and P.Tiwari, A deterministic algorithm for sparse multivariate polynomial interpolation, *Proc. 20th Symposium on Theory on Computing*, 1988, pp. 301-309.

[Ed] J. Edmonds, Paths, trees and flowers, *Canad. J. Math.* 17 (1965) 449-467.

[GK] D. Y. Grigoriev and M. Karpiński, The matching problem for bipartite graphs with polynomially bounded permanents is in *NC*, *28th Annual IEEE Symposium on Foundations of Computer Science*, Los Angeles, CA, 1987, pp. 166-172.

[Ha] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, 1972.

[He] Xin He, Private communication.

[Jo1] D.S.Johnson, The NP-completeness column: an ongoing guide, *J. Algorithms* 6 (1985) 434-451.

[Jo2] D.S.Johnson, The NP-completeness column: an ongoing guide, *J. Algorithms* 8 (1987) 438-448.

[KUW] R. M. Karp. E. Upfal and A. Wigderson, Constructing a perfect matching is in random *NC*, *Combinatorica* 6, 1 (1986) 35-48.

[La] M. Las Vergnas. A note on matchings in graphs, *Colloque sur la Théorie des Graphes (Paris 1974)*, Cahiers Centre Etudes Rech.Opér. 17, 1975, 257-260.

[Le] P. G. H. Lehot, An optimal algorithm to detect a line graph and output its root graph, *JACM* **21**, (1974) pp. 569-575.

[Mi] G.J.Minty, On maximal independent sets of vertices in claw-free graphs, *J. Comb. Theory B* 28 (1980) 284-304.

[MN] G.Miller and J.Naor, Flow in planar graphs with mulitple sources and sinks, *manuscript.*

[MR] G. L. Miller and J. Reif, Parallel tree contraction and its applications, *Proceedings 26th Annual IEEE Symposium on Foundations of Computer Science*, (1985) pp. 478-489.

[MVV] K. Mulmuley, U. V. Vazirani and V. V. Vazirani, Matching is as easy as matrix inversion, *Combinatorica* 7, 1 (1987) 105-113.

[Na] J. Naor, Computing a perfect matching in a line graph, *3rd International Workshop on Parallel Computation and VLSI Theory, Aegean Workshop on Computing*, Corfu Island, Greece (1988).

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

[Sb] N.Sbihi, Algorithme de recherche d'un stable de cardinalité maximum dans un graphe, *Disc. Math.* 29 (1980) 53-76.

[Su] D. P. Sumner, On Tutte's factorization theorem, *Proceedings of the Capital Conference on Graph Theory and Combinatorics,* George Washington University, Washington D.C., 1973, pp. 350-355. Lecture Notes in Math., Vol. 406, Springer, Berlin, 1974.

[SV] Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3** (1982) 57-67.

[Wh] H. Whitney, Congruent graphs and the connectivity of graphs, *Amer. J. Math.* **54** (1932), 150-168.

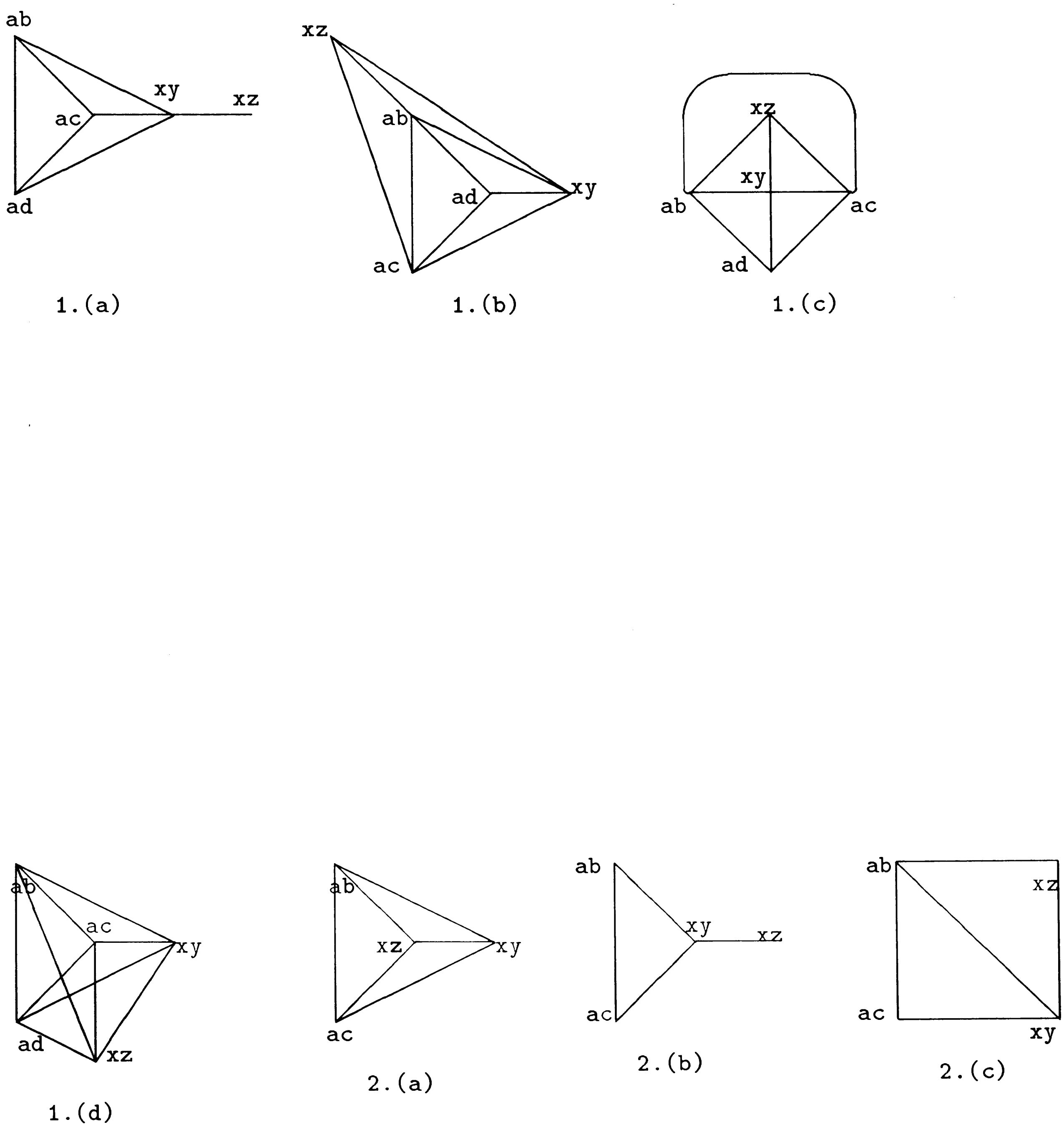

Figure 1: Graphs used in the case analysis

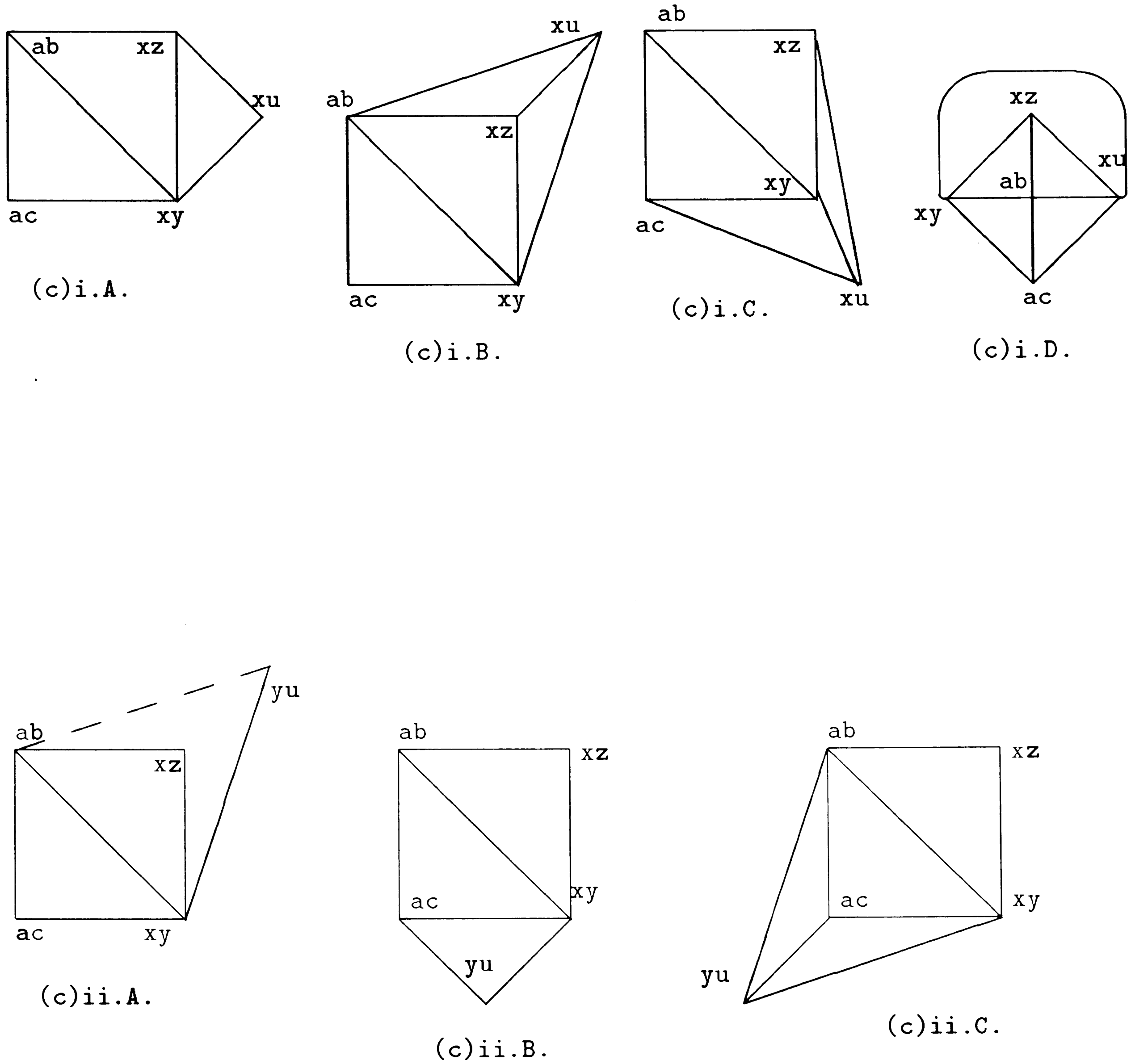

Figure 2: Graphs used in the case analysis

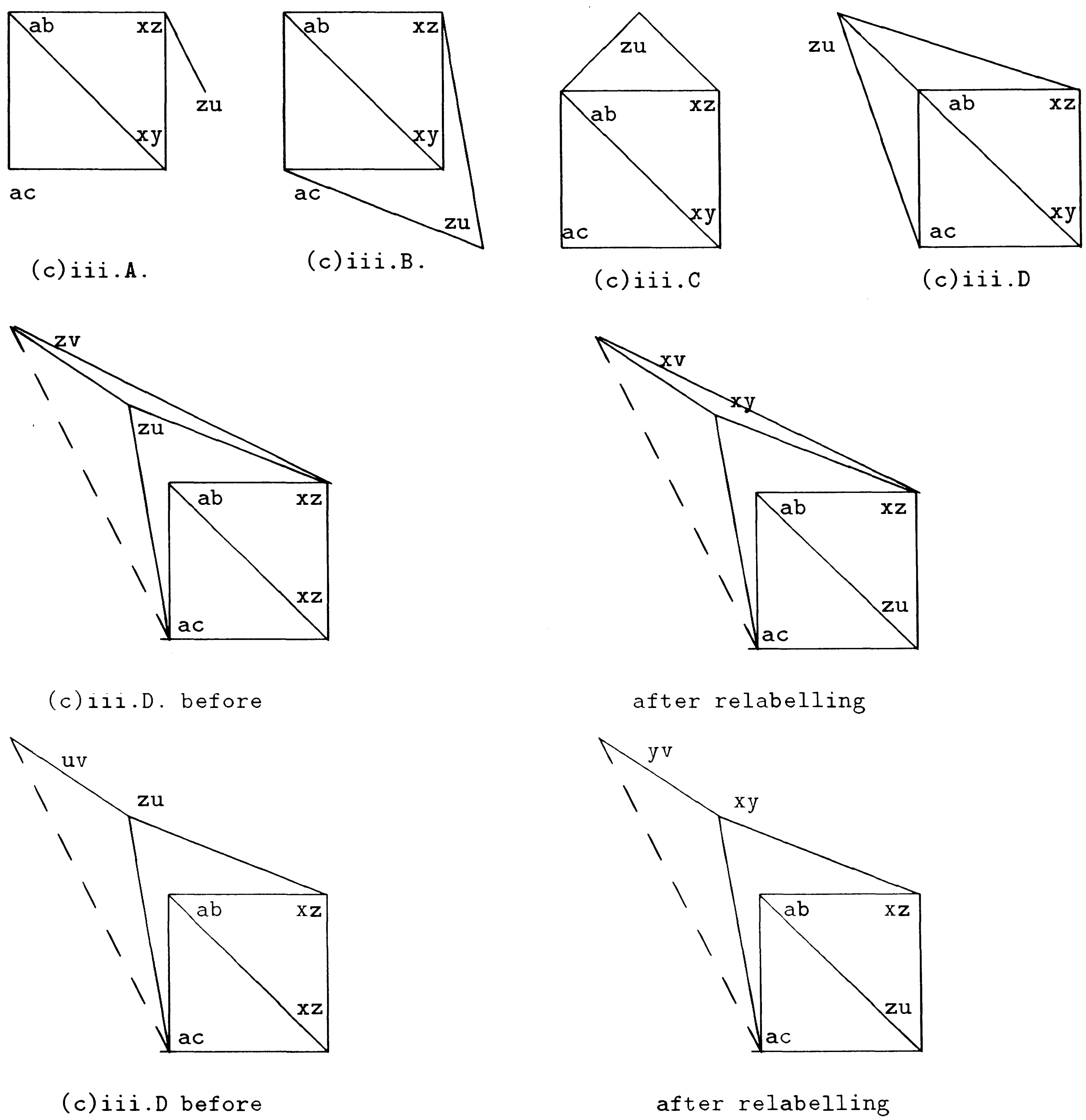

Figure 3: Graphs used in the case analysis