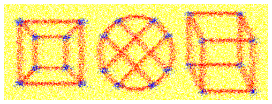
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 13, no. 2, pp. 219–232 (2009)

Towards an optimal algorithm for recognizing Laman graphs

Ovidiu Daescu Anastasia Kurdia

Department of Computer Science,
The University of Texas at Dallas, Richardson, TX 75080, USA.

Abstract

A graph G with n vertices and m edges is a generically minimally rigid graph (*Laman graph*), if $m = 2n-3$ and every induced subset of k vertices spans at most $2k-3$ edges. Laman graphs play a fundamental role in rigidity theory.

We discuss the **Verification problem:** *Given a graph G with n vertices, decide if it is Laman.* We present an algorithm that recognizes Laman graphs in $O(T_{st}(n) + n \log n)$ time, where $T_{st}(n)$ is the best time to extract two edge disjoint spanning trees from a graph with n vertices and $2n-2$ edges, or decide no such trees exist. So far, it is known that $T_{st}(n)$ is $O(n^{3/2}\sqrt{\log n})$.

Submitted:	Reviewed:	Revised:	Reviewed:
April 2008	December 2008	January 2009	May 2009
Revised:	Accepted:	Final:	Published:
May 2009	July 2009	July 2009	July 2009
Article type:		Communicated by:	
Concise paper		I. G. Tollis	

Daescu's research is supported in part by NSF award CCF-0635013. The authors would like to thank Ileana Streinu for organizing the 2005 and 2006 Barbados workshops, partially supported by NSF. This work would not have been possible without participation in those workshops.

E-mail addresses: daescu@utdallas.edu (Ovidiu Daescu) akurdia@utdallas.edu (Anastasia Kurdia)

1 Introduction

A graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges is a generically minimally rigid graph (also called *Laman graph* [10]), if $m = 2n - 3$ and every k-vertex subgraph has at most $2k - 3$ edges. Laman graphs play a fundamental role in rigidity theory: they characterize minimally rigid planar bar-and-joint systems that appear in robotics, sensor and network topologies and polymer physics. (A system of fixed-length bars and flexible joints connecting them is minimally rigid if it becomes flexible once one bar is removed).

In this paper we address the **Verification problem:** *Given a graph G with n vertices, decide if it is Laman.*

1.1 Previous work

Most existing verification algorithms take quadratic time in the number of input vertices to recognize Laman graphs [1, 7, 8, 11, 17]. A very elegant and simple algorithm is the *pebble game* algorithm, first proposed by Jacobs and Hendrickson [9], and generalized later on by Streinu, Lee, and Theran in a number of papers [6, 11, 15, 16]. The pebble game algorithm solves the verification problem in $O(n^2)$ time.

The characterizations of Laman graphs with tree partitions are due to Recski [14], Crapo [3] and Lovasz and Yemini [12]. Lovasz and Yemini proved that a graph $G = (V, E)$ is Laman if and only if, for each edge $e \in E$, the multigraph $G \cup \{e\}$ is the union of two edge disjoint spanning trees; Recski proved this statement also holds for any $e \notin E$. Crapo showed that graph is Laman if and only if it is decomposable into three disjoint trees such that every vertex is in every two of those trees and no nontrivial subrtrees of distinct trees have the same set of vertices. Verifying any of these conditions directly requires $\Omega(n^2)$ time.

A subquadratic time algorithm for the verification problem is due to Gabow and Westermann [5]. It requires $O(n^{3/2}\sqrt{\log n})$ time and solves the problem in two steps: (1) Find a 2-forest of $G \cup \{e\}$ (two edge disjoint spanning trees), which is done in $O(n^{3/2}\sqrt{\log n})$ time, and (2) Test if a data structure computed in step (1), called *top clump*, is empty: this is done in $O(n \log n)$ time. Thus, step (2) is coupled with step (1), in the sense that if two edge disjoint spanning trees are given to step (2), computed by some arbitrary method, then step (2) should be changed and could require asymptotically larger time.[1]

A different verification method was proposed recently by Bereg [1]. Bereg's algorithm performs a step-by-step decomposition of G, aiming to construct a hierarchical decomposition H of G, called a *red-black hierarchy* (RBH). It is proven in [1] that G is a Laman graph if and only if H is a RBH.

Hierarchy. [1] A *hierarchy* $H(G, T_h, \alpha, \beta)$ for a given graph $G(V, E)$, $|V| = n$, is a graph $H(V_h, E_h)$, $E_h = T_h \cup \beta(E)$. T_h is a set of edges forming a rooted

[1] Very recently, it was suggested to us that a method presented in [16] can be adapted to speed up the top clump test to $O(n)$ time, assuming the data structures computed in step (1) are available.

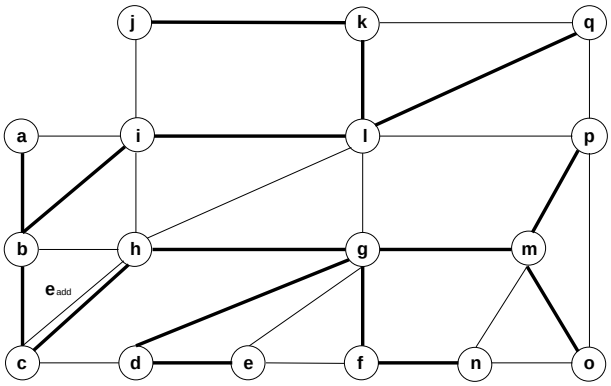

Figure 1: A graph $G^* = G \cup \{e_{add}\}$ and its two edge disjoint spanning trees (red tree is drawn with thick lines).

tree. The function $\alpha : V \to L(T_h)$, defines a one-to-one correspondence between the vertices of V and the leaves of the tree, denoted as $L(T_h)$. The function $\beta : E \to V(T_h) \times V(T_h)$ maps the edge (u, v) of G to the edge $\beta(u, v) = (\beta_1(u, v), \beta_2(u, v))$ of H (called *cross edge*), so that $\beta_1(u, v)$ and $\beta_2(u, v)$ are ancestors, but not common ancestors, of $\alpha(u)$ and $\alpha(v)$, respectively.

Red-black hierarchy. [1] A *red-black hierarchy* $H(G, T_h, \alpha, \beta)$ is a hierarchy satisfying the following conditions:

- The root of the tree T_h has exactly two children (root rule);
- A vertex is the only child of its parent if and only if it is a leaf (leaf rule);
- For any cross edge its endpoints have the same grandparent but different parents in the tree (cross-edge rule);
- Cross edges connect all grandchildren of a vertex and form a tree (tree rule).

The construction of the RBH in [1] has three major phases. (1) A copy of an edge of G, e_{add}, is added to G and two edge-disjoint spanning trees, *red tree* T^r and *black tree* T^b are computed for $G^* = G \cup \{e_{add}\}$ using a known method (if no such trees exist, then G is not Laman), see Figure 1. An $O(n^2)$ time algorithm is used to obtain the trees.

(2) A decomposition of G^* is performed and a characterizing hierarchy $H = H(G^*)$ is constructed in correspondence with the steps of the decomposition (Figure 2), which is done in $O(n^2)$ time.

(3) A certification whether H satisfies the rules of a RBH is performed in $O(n)$ time.

Since steps (2) and (3) do not depend on how step (1) is performed, this method decouples the computation of the two edge disjoint spanning trees in step (1) from the rest of the computation. (After the preliminary version of this

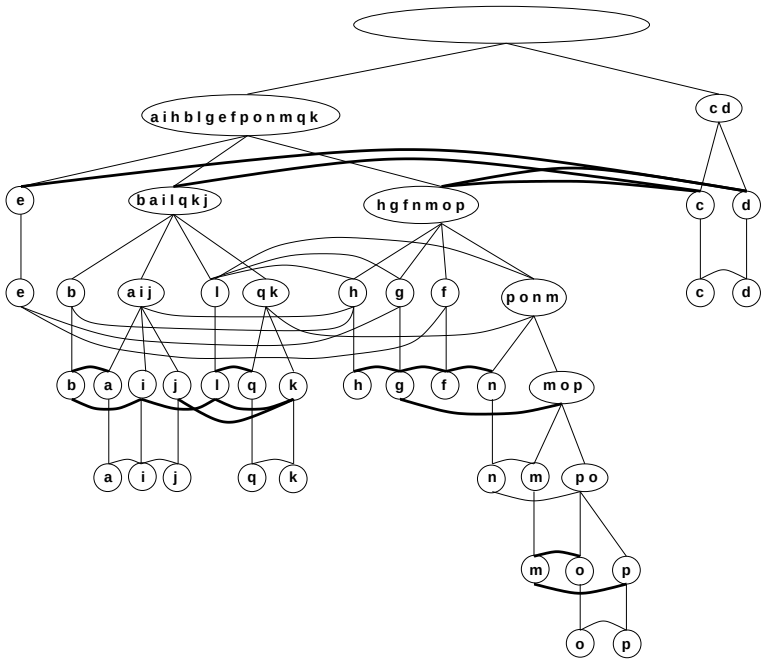

Figure 2: A corresponding red-black hierarchy H for graph G^*. Red edges are drawn with thick lines.

article was published, an improved, $O(n \log n)$ time solution for step (2), was posted at [2] that also uses $O(n^{3/2}\sqrt{\log n})$ time algorithm for step (1)).

1.2 Our results

Let $T_{st}(n)$ be the time to find two edge disjoint spanning trees in G. We present an $O(T_{st}(n)+n \log n)$ time verification algorithm based on a simple observation: from Corollary 4 in [1], it follows that it is not necessary to actually construct H to decide G is Laman; we only need to decide whether a RBH decomposition H *exists* for G. Thus, steps (2) and (3) from the above Bereg's algorithm become: (2) use the two spanning trees to decide whether G *admits* a RBH decomposition.

Our algorithm has two steps: (1) Compute two edge disjoint spanning trees by the best possible method. We use the $O(n^{3/2}\sqrt{\log n})$ time algorithm from [5] since this is the best we know (if, say, a simple $O(n \log n)$ time algorithm is discovered for this part, we will use that one). (2) Given two edge disjoint spanning trees for G, we give a solution for deciding whether G admits a RBH decomposition, that uses depth-first search and segment trees only, and takes $O(n \log n)$ time. This step is independent of how step (1) is done.

At the end of step (2) we know if G is Laman or not. Moreover, we also show that the RBH can be actually constructed in $O(n \log n)$ time using a two steps

procedure that is simple and easy to implement. Thus, our algorithm decouples step (1) from step (2), to take advantage of future improvements on step (1), and solves the second step of the verification in $O(n \log n)$ time instead of $O(n^2)$ time.

We summarize our results below.

Theorem 1 *Given a graph G with n vertices and m edges deciding whether G is a Laman graph can be done in $O(T_{st}(n) + n \log n)$ time, where $T_{st}(n)$ is the time to extract two edge disjoint spanning trees from G or decide no such trees exist.*

Theorem 2 *Given two edge-disjoint spanning trees for G^*, a red-black hierarchy for G^*, if it exists, can be constructed in $O(n \log n)$ time.*

In the rest of the paper we explore the properties of the RBH, provide the proofs for the above theorems and give implementation details for our algorithm. From now on, we assume familiarity of the reader with the decomposition and hierarchy construction processes described in [1].

2 A sufficient condition

We begin by showing that if all edges are removed from G during the decomposition process, the graph H constructed from the decomposition is always a RBH and thus G is a Laman graph.

Theorem 3 *(i) The graph H constructed from the decomposition always satisfies the four rules of red-black hierarchy and (ii) The graph G is a Laman graph if and only if all edges are removed from G during the decomposition process.*

The correctness of the above statement follows from Theorem 2 of [16], since the decomposition proceeding until all edges are removed is a special case of the (2,3)-pebble-game with colors decomposition [16]. We choose to prove it via red-black hierarchies to reveal some properties of the latter, described in the lemmas below.

Notations. Anything marked by the subscript h in what follows refers to H. Let $color(v_h)$ denote the color associated with node v_h (red or black). If $c = color(v_h)$ is red then $\overline{c}$ is black and vice versa. Vertices of H correspond to spanning trees of connected subgraphs of G. $C(v_h)$ denotes the connected subgraph of vertex v_h, $parent(v_h)$ denotes a parent of v_h in H. $V(v_h)$ denotes the set of vertices of $C(v_h)$. $V(T)$ denotes the set of vertices of G spanned by the tree T.

Lemma 4 *The four RBH rules always hold for the graph H that characterizes decomposition of any graph $G^* = G \cup e_{add}$, if the edge set of G^* can be partitioned into two edge-disjoint spanning trees.*

Proof. *Root rule.* At the very first step, H is empty and a node r_h of color c, corresponding to the spanning tree T^c that does not contain the added edge e_{add}, is created in H. The node r_h is the root of H. Then, e_{add} is deleted from the other tree $T^{\overline{c}}$, which necessarily creates exactly two trees of color $\overline{c}$ in G^* and exactly two nodes of color $\overline{c}$ in H that are children of r_h, corresponding to these two trees. Thus, the root rule always holds.

Leaf rule. If a vertex v_h is the only child of its parent then v_h is a leaf. At the step when v_h was created, the decomposition process has stopped for $C(v_h)$: there was just one tree of color $color(v_h)$ in $C(parent(v_h))$ and just one tree of color $\overline{color(v_h)}$ (otherwise $C(parent(v_h))$ would have been partitioned further and v_h would have siblings). Hence, the vertex v_h corresponding to $C(v_h)$ is a leaf in H.

A leaf vertex cannot have any siblings. Suppose there is a vertex y_h having $k > 1$ children $x_h^i, i = 1, \ldots, k$ and x_h^j is a leaf. The vertex y_h corresponds to a connected subgraph spanned by a tree of color $c = color(y_h)$ and a spanning forest of k trees of color $\overline{c}$; x_h^j corresponds to a connected subgraph $C(x_h^j)$, spanned by a tree of color $\overline{c}$ and a forest of color c. If the forest contains more than one tree, at the next step of the decomposition the edges of color $\overline{c}$ connecting the trees of the spanning forest will be deleted, the spanning tree of color $\overline{c}$ will split into at least two different trees and corresponding vertices will be created in H as children of x_h^i. Hence, x_h^j cannot be a leaf vertex. If the spanning forest of $C(x_h^j)$ contains just one tree, then a vertex corresponding to that tree, of color c, is created in H as a child of x_h^j and x_h^j cannot be a leaf vertex, a contradiction.

Cross-edge rule. A cross edge is added between any two vertices u_h and v_h at step i if their corresponding vertex sets $V(u_h)$ and $V(v_h)$ previously belonged to one connected subgraph $C_{u,v}$ and got separated at step $i-1$ by removing the edge with endpoints in $V(u_h)$ and $V(v_h)$. At level $i-2$ of H there is always a vertex that corresponds to $C_{u,v}$. The vertices at the same level of H correspond to connected subgraphs that are disjoint subgraphs of G. Hence, no other vertex at level $i-2$ of H can correspond to a connected subgraph containing $V(u_h)$, $V(v_h)$, their subsets, or the union of their subsets. The vertex corresponding to the connected subgraph $C_{u,v}$ is a common grandparent of u_h and v_h.

According to the construction rules, parents of u_h and v_h in H correspond to different connected subgraphs and cannot coincide.

Tree rule. If k edges are removed from the tree $T \subset E(G)$ spanning the vertex set $V(v_h)$ that corresponds to some vertex v_h of H, $k+1$ new trees result from T and $k+1$ nodes are created as grandchildren of v_h in H. For each edge e deleted from T, exactly one cross edge is added between the grandchildren of v_h. Each grandchild of v_h gets a cross edge incident to it, thus cross edges form a tree spanning all the grandchildren of v_h. □

Lemma 5 *If all edges are removed from G during the decomposition process then the characterizing graph H of G satisfies the definition of hierarchy.*

Proof. There is a cross edge $e_h = (u_h, v_h)$ in H for each edge $e = (u, v)$ of G. The edge e is deleted from G when it crosses the cut separating u from v, a cross edge is then added between the vertices of H corresponding to the connected components of u and v at the current step.

There is one-to-one correspondence between the leaves of T_h and the vertices of G: If the decomposition continues until all edges are removed, each vertex v of G is eventually disconnected from the rest of the graph by deleting an edge of some color c. A vertex l_h corresponding to a tree of color c spanning the connected subgraph $C_v = \{v\}$ is then created in H. Since C_v cannot be split further, the decomposition stops for C_v and l_h becomes a leaf vertex of T_h. Also, there is no leaf vertex in H that does not correspond to a vertex of G. Suppose there exists such vertex in H. Then, it corresponds to a tree spanning a connected subgraph C_x, with $|C_x| > 1$, i.e. C_x contains edges that were not deleted during the decomposition of G, a contradiction.

For each edge e_h its endpoints u_h and v_h are ancestors of $\alpha(u)$ and $\alpha(v)$, respectively, but they are not their common ancestors. (Recall that $\alpha(u)$ and $\alpha(v)$ are the leaf vertices of H corresponding to vertices u and v of G). The leaf vertices of H that correspond to vertices in $V(u_h)$ are the descendants of u_h in H. Since $u \in V(u_h)$, u_h is an ancestor of $\alpha(u)$. Similarly, v_h is an ancestor of $\alpha(v)$. Since u_h and v_h belong to the same level of the hierarchy, u_h cannot be an ancestor of $\alpha(v)$, v_h cannot be an ancestor of $\alpha(u)$. □

Lemma 6 *If G has edges left at the end of the decomposition process, the characterizing graph H of G does not satisfy the definition of hierarchy.*

Proof. If there are non-deleted edges of G when the decomposition stops, then there are no corresponding edges for them in H. In addition, we do not have a one-to-one map from V to $L(T_h)$: some leaves of T_h correspond to connected subgraphs containing several vertices. □

This concludes the proof of Theorem 3.

Thus, building H is not required for certifying Laman graphs. It is sufficient to perform the decomposition of G according to the rules from [1] and then check whether there are edges left in G when the decomposition ends.

3 Decomposition

Our main goal now is to speed up the decomposition process. At each step of the decomposition edges of only one color are deleted. The groups of red and black edges are deleted in turns. At each step, except the first and the last ones, at least one edge is deleted from G. Let $g = (g_2, g_3, \ldots, g_k)$ be a grouping of some (possibly all) edges of G, such that all edges of a group g_i were deleted from G at step i. Instead of H, we use g to characterize the graph decomposition.

3.1 Algorithm

We slightly alter the graph decomposition algorithm from [1]. The edges to be deleted at the next step are identified at the end of the preceding step and are marked for deletion. At the first step, e_{add} is marked for deletion (and no other action is performed). Each iterative step in G consists of removing the marked edges of some color c and identifying and marking the edges (of the opposite color $\overline{c}$) crossing the cuts induced by removing the marked edges.

We note that once the original graph has split into several connected subgraphs, the decomposition proceeds independently on each subgraph, and the problem of finding the edges to be deleted at the subsequent step can be viewed as several independent subproblems, each on a distinct connected subgraph. We also note that deletion of any edge $e = (u, v)$ from its tree (of color $color(e)$), where u is a parent of v in a depth-first-search (DFS) ordering of the tree of color $color(e)$, always forms two trees such that one of them is rooted at v and all nodes in that tree are descendants of v.

Consider the graph G^* and its two edge disjoint spanning trees T^c and $T^{\overline{c}}$, rooted at vertices r^c and $r^{\overline{c}}$, respectively. Let $DFS(c)$ be the depth-first search traversal of G^* starting at r^c and using only edges of color c, where c is either red or black. We assign each vertex of G two DFS order numbers, one from $DFS(red)$ and another one from $DFS(black)$. New edges are never added to the trees, so the numbers never change. Whenever an edge e is mentioned in the following text as a vertex pair, the first vertex is always the parent of the second vertex in $DFS(color(e))$.

When an edge $e = (u, v)$ of color c is deleted from a tree T_k^c rooted at some r^c and spanning a connected subgraph C_k, two trees emerge: T_i^c rooted at r^c and T_j^c rooted at v. Only the vertices of T_j^c are descendants of v in $DFS(c)$. The ancestor/descendant relationship can be established in the $DFS(c)$ tree by looking at the discovery and finish times ($d^c[\cdot]$ and $f^c[\cdot]$, respectively) of the vertices.

Lemma 7 *An edge (x, y) of color $\overline{c}$ crosses the cut $(V(T_i^c), V(T_j^c))$ induced by the deletion of the edge (u, v) of color c if and only if one of its endpoints is a descendant of v and the other one is not, i.e., exactly one of its endpoints discovery times is in $t = [d^c[v], f^c[v]]$.*

Proof. If $d^c[x] \notin t$ and $d^c[y] \in t$, then $x \in T_i^c$ and $y \in T_j^c$, so (x, y) clearly crosses the cut. A symmetric argument applies if $d^c[x] \in t$ and $d^c[y] \notin t$.

If $d^c[x] \notin t$ and $d^c[y] \notin t$, neither x nor y are in T_j^c, so both endpoints of (x, y) are in T_i^c and (x, y) does not cross the cut. If $d^c[x] \in t$ and $d^c[y] \in t$, both endpoints of (x, y) are in T_j^c and (x, y) does not cross the cut. □

From Lemma 7 it follows that if we associate an interval $[d^c[u], d^c[v]]$ with every edge (u, v) of color $\overline{c}$, the intervals corresponding to the edges crossing the cut have exactly one endpoint in t.

We identify such intervals using a segment tree data structure enhanced with two lists at each internal node, one sorted by the start time of the intervals stored at the node and one sorted by their finish time. A segment tree [13] is a

balanced binary search tree that stores a set of intervals with endpoints from a finite set of abscissae (intervals corresponding to edges of color $\bar{c}$, for example). Each of its nodes u has an interval $I(u)$ associated with it and stores a list of input intervals intersecting $I(u)$. Binary search in a segment tree allows to report the intervals containing a query point.

In our case, the endpoints of the intervals are integer numbers, so an interval containing a point $p+\Delta$ or $p-\Delta$, for any $0 < \Delta < 1$ and integer p, contains the point p as well. First, we find the intervals with one endpoint before $d^c[v]$ and the other endpoint in t by querying for intervals containing the point $d^c[v] - \Delta$ (first query). Then, we find the intervals with one endpoint in t and the other endpoint after $f^c[v]$ by querying for intervals containing the point $f^c[v] + \Delta$ (second query).

To ensure that each returned interval has exactly one endpoint in t we augment the standard segment tree by storing two sorted lists at each node, instead of just one list. With each node u, we store a list $L_{finish}(u)$ of intervals that intersect $I(u)$ that is sorted by the finish time of the intervals in non-decreasing order; similarly, the list L_{start} stores the same intervals sorted by their starting time in non-increasing order. Both queries are given an additional parameter: $f^c[v]$ for the first one and $d^c[v]$ for the second one. The first query only scans the lists L_{finish} and reports the intervals that have their right endpoint no greater than $f^c[v]$. The second query only scans at the lists L_{start} and reports the intervals that have their starting point no later than $d^c[v]$. Thus, this data structure allows us to return intervals with exactly one endpoint in t.

We also maintain an auxiliary list L for each segment tree T_S. Each entry in that list corresponds to one interval and stores a list of pointers to interval's positions in L_{finish} and L_{start} (the pointers are obtained when the interval is inserted into L_{finish} and L_{start}). Additionally, the entries of L_{finish} and L_{start} should point to their corresponding entry in L. For each interval, $O(\log n)$ auxiliary data is stored.

3.2 Running time

To efficiently identify edges crossing the cuts at each step of the decomposition, we maintain two segment trees, one for the red intervals $[d^{red}[u], d^{red}[v]]$ associated with the black edges and the another one for the black intervals $[d^{black}[u], d^{black}[v]]$ associated with the red edges.

Each query with an edge (interval t) takes $O(\log n + k)$ time, where k is the number of intervals (crossing edges) reported: the query scans the lists L_{finish} (L_{start}) at each level of the segment tree, stopping each time when an interval with a right (left) endpoint greater than $f^c[v]$ ($d^c[v]$) is encountered. To avoid reporting an interval more than once, the interval is deleted from the segment tree (including the sorted lists associated with the nodes that store it) right after it is returned by a query.

When an interval is deleted from T_S a) a corresponding entry should be located in L, and b) all the entries in T_S associated with that interval should be deleted. If we keep a pointer from each interval to the corresponding entry

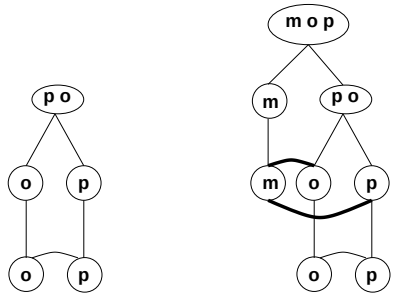

Figure 3: (a) H after considering edges of $g_8 = \{(o, p)\}$. (b) H after considering edges of $g_8 \cup g_7 = \{(m, o), (m, p)\}$.

in L, part a) can be done in constant time. Part b) is completed by removing from the linked lists L_{finish} and L_{start} all entries pointed to by entries in the list of L and takes $O(\log n)$ per interval deletion. Finally, the entry of L for that interval should be deleted, which takes constant time.

We can check that $m = 2n - 3$ in $O(n)$ time. Finding two edge disjoint spanning trees or deciding no such trees exist takes $T_{st}(n)$ time. The best known algorithm so far for this task has $T_{st}(n) = O(n^{3/2}\sqrt{\log n})$ time [5]. The decomposition takes $O(n \log n)$ time: $O(n \log n)$ for the maintenance of segment trees, $O(n \log n)$ to answer all queries, and $O(n)$ to check if G has any edges left at the end of the decomposition. This concludes the proof of Theorem 1.

4 The hierarchy reconstruction algorithm

The order in which edges are deleted from G during the decomposition determines the structure of the corresponding red-black hierarchy H, so given g, one can unambiguously construct H in top-down fashion according to the rules from [1]. In the original approach, to construct the i-th level of H, one has to know the spanning sub-trees at step $i - 2$ of the decomposition and to spend $O(n)$ time figuring out what trees appear after removal of edges at the beginning of step i.

We consider the decomposition process in reverse order (i.e. start from n red and n black disjoint trees of one vertex each and add edges to them until two spanning trees of G are formed). The last group g_k of $g = (g_2, g_3, \ldots, g_k)$ contains edges of some color c deleted at the very last step of the decomposition. Each endpoint v of edges of g_k corresponds to a subtree of G of color c spanning only the vertex v. A leaf node $v_h = \alpha(v)$ is added to the k-th level of H for each such vertex v. Only one leaf vertex is created for the endpoint shared by multiple edges from g_k. For every edge (u, v) of g_k a corresponding cross edge $\beta(u, v) = (\alpha(u), \alpha(v))$ is added to H. For every leaf vertex $\alpha(v)$ of H, its parent

should be at level $k-1$ of H, corresponding to a subtree in G that is of color $\bar{c}$ and spans only the vertex v. Such parent vertex $v_h^p = parent(\alpha(v))$ is added to level $k-1$ of H along with a tree edge connecting v_h^p and $\alpha(v)$ (we call this the *parent creation rule*). The vertices of H connected by a cross edge have the same grandparent. For every cross edge tree T_k^j formed at the k-th level, a vertex v_h^g is added to level $k-2$ of H, as well as a tree edge connecting v_h^g and v_h^p, for every $v_h \in T_k^j$. We have completed level k of H as well as added some elements to the two upper levels (see Figure 3).

At the i-th iterative step for each cross edge (x, y) of g_i of color c two vertices v_h^x and v_h^y on the i-th level of H are identified. They correspond to trees in G of color c that contained x and y respectively at the i-th step of the decomposition. If for some endpoint x of an edge from g_i v_h^x does not exist on the i-th level of H, a new vertex v_h^x should be created at the i-th level and a parent for it should be added following the parent creation rule. Then the cross edge corresponding to (x, y) is added to H between v_h^x and v_h^y. After all edges of g_i are considered, all cross-edges of the i-th level of H are in place. For each cross edge tree T_i^j formed at the i-th level of H, a node is added to level $i-2$ of H. That grandparent node becomes a parent of the parents of the vertices of H spanned by the cross-edge tree T_i^j. At this time the i-th level of H is complete and levels $i-1$ and $i-2$ of H are partially constructed. Repeating these steps for all g_i, $i > 2$, yields the RBH H.

Algorithm. The algorithm takes g as an input and returns the corresponding RBH H. It relies on classical UNION-FIND data structure augmented with a few simple operations. A pointer to a vertex of H is associated with each set, and a function GET-VERTEX(s) returns the vertex pointed to by the set s. The resulting set of the UNION(s,t) operation points to the vertex that s used to point to. A canonical MAKE-SET(s) is modified to take an additional parameter v, the vertex of H that the newly created set should point to. A function REPRESENTATIVE(s) returns an element belonging to the set s. Two instances of UNION-FIND data structure are used. The sets of one instance point to vertices of some level i of H, the sets of another instance point to the parents of the vertices of the i^{th} level. In the pseudocode below these instances are distinguished by subscripts a and $\overline{a}$, such that if $a = 0$ then $\overline{a} = 1$ and vice versa.

```
RECONSTRUCTION(g)
 1  a ← 0
 2  i ← k
 3  while i > 2
 4     do for e = (x, y) ∈ g_i
 5           do for u ∈ {x, y}
 6                 do if FIND_a(u) = NIL
 7                       then v_h ← H.createVertex()
 8                            MAKE − SET_a(u, v_h)
 9                            w_h ← H.createVertex()
10                            MAKE − SET_ā(u, w_h)
```

```
11                              H.addParentEdge(w_h, v_h)
12                  S_x ← FIND_a(x)
13                  S_y ← FIND_a(y)
14                  H.addCrossEdge(GET − VERTEX(S_x), GET − VERTEX(S_y))
15                  UNION(S_x, S_y)
16          for s ∈ UNION − FIND_a
17              do v_h ← H.createVertex()
18                  SET − POINTER(s, v_h)
19          for s ∈ UNION − FIND_ā
20              do
21                  v_h ← GET − VERTEX(s)
22                  r ← REPRESENTATIVE(s)
23                  w_h ← GET − VERTEX(FIND_a(r))
24                  H.addParentEdge(v_h, w_h)
25          a ← ā
26          i ← i − 1
```

Running time. Obtaining g for G^* takes $O(n \log n)$ time. The order of edges in g uniquely determines the sequence of $UNION()$ operations during the hierarchy reconstruction phase, so the algorithm for maintaining UNION-FIND data structure in time linear in the number of operations can be applied [4]. The time spent on reconstructing one level is proportional to the number of cross edges at that level. The total number of cross edges is $O(n)$. This allows to complete the reconstruction phase in $O(n)$ time, so the total time for constructing the RBH is $O(n \log n)$. This concludes the proof of Theorem 2.

5 Conclusions

In this paper we discussed the problem of recognizing whether a given graph G with n vertices is Laman. We presented an algorithm that recognizes Laman graphs in $O(T_{st}(n) + n \log n)$ time, where $T_{st}(n)$ is the best time to extract two edge disjoint spanning trees from a graph with n vertices and $2n - 2$ edges, or decide no such trees exist. So far, it is known that $T_{st}(n)$ is $O(n^{3/2}\sqrt{\log n})$.

We notice that improvements to $T_{st}(n)$ would result in improvements to our algorithm. We leave improving $T_{st}(n)$ as an open problem.

References

[1] S. Bereg. Certifying and constructing minimally rigid graphs in the plane. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 73–80, 2005.

[2] S. Bereg. Faster algorithms for rigidity in the plane. http://arxiv.org/abs/0711.2835, 2007.

[3] H. H. Crapo. On the generic rigidity of plane frameworks. *Technical Report 1278, Institut de recherche d'informatique et d'automatique*, 1988.

[4] H. Gabow and R. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[5] H. Gabow and H. Westermann. Forests, frames, and games: algorithms for matroid sums and applications. *Algorithmica*, 7(1):465–497, 1992.

[6] R. Haas, A. Lee, I. Streinu, and L. Theran. Characterizing sparse graphs by map decompositions. *Journal of Combinatorial Mathematics and Combinatorial Computing (JCMCC)*, 62:3–11, 2007.

[7] B. Hendrickson. Conditions for unique graph realizations. *SIAM J. Comput.*, 21(1):65–84, 1992.

[8] H. Imai. On combinatorial structures of line drawings of polyhedra. *Disc. Appl. Math.*, 10:79–92, 1985.

[9] D. Jacobs and B. Hendrickson. An algorithm for two dimensional rigidity percolation: The pebble game. *Journal on Computational Physics*, 137(2):346–365, 1997.

[10] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathmatics*, 4:331–340, 1970.

[11] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437, 2008.

[12] L. Lovasz and Y. Yemini. On generic rigidity in the plane. *SIAM J. Algebraic and Discrete Methods*, 3(1):91–98, 1982.

[13] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[14] A. Recski. A network theory approach to the rigidity of scheletal structures II: Laman's theorem and topological formulae. *Discrete Applied Math*, 8:63–68, 1984.

[15] I. Streinu and L. Theran. Sparse hypergraphs and pebble game algorithms. *European Journal of Combinatorics*, 2009.

[16] I. Streinu and L. Theran. Sparsity-certifying graph decompositions. *Graphs and Combinatorics*, 25:219–238, 2009.

[17] K. Sugihara. On some problems in the design of plane skeletal structures. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):355–362, 1983.