1-1-2005

Finding and Maintaining Rigid Components

Audrey Lee
University of Massachusetts Amherst

Ileana Streinu
Smith College, istreinu@smith.edu

Louis Theran
University of Massachusetts Amherst

Finding and Maintaining Rigid Components

Audrey Lee[*‡] Ileana Streinu[†‡] Louis Theran[*‡]

Abstract

We give the first complete analysis that the complexity of finding and maintaining *rigid components* of planar bar-and-joint frameworks and arbitrary d-dimensional body-and-bar frameworks, using a family of algorithms called pebble games, is $O(n^2)$. To this end, we introduce a new data structure problem called *union pair-find*, which maintains disjoint edge sets and supports `pair-find` queries of whether two vertices are spanned by a set.

We present solutions that apply to generalizations of the pebble game algorithms, beyond the original rigidity motivation.

1 Introduction

Efficient algorithms for rigidity are important for practical applications, such as protein flexibility[6]. Rigidity of planar bar-and-joint frameworks is well-understood and characterized by Laman graphs. The *pebble game* of Jacobs and Hendrickson[5] is an elegant algorithm for deciding rigidity and finding the rigid components in the planar case. Despite its simplicity, its complexity has never been fully analyzed in terms of the necessary data structures, even in the more recent version by Berg and Jordan[1].

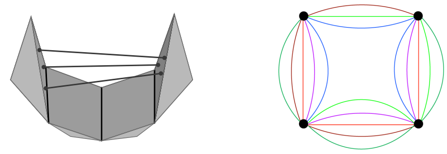
Tay's characterization[9] for d-dimensional body-and-bar frameworks is the only known combinatorial tool for handling rigidity in higher dimensions. In [7], the first two authors generalize [5] to a family of pebble games on a larger class of graphs called (k,l)-*sparse* graphs (defined below) including (k,l)-*arborescences*; the graphs needed to handle generic rigidity via the theorems of Laman and Tay are both instances of (k,l)-arborescences.

In this paper, we complete the analysis of the pebble game algorithms of [7] and [5], and show a clean $O(n^2)$ running time including data structure manipulation. Along the way, we abstract a general data structure problem called *union pair-find*; this differs from the classical union-find in that it maintains disjoint *edge sets*, which may not be vertex-disjoint. To the best of our knowledge, the need for such a data structure has not been previously identified.

*Department of Computer Science, UMass Amherst, `alee@cs.umass.edu, ltheran@cs.umass.edu`

†Department of Computer Science, Smith College, `streinu@cs.smith.edu`

‡Research partially funded by NSF grant CCR-0310661.



(a) (b)

Figure 1: (a) Generic minimally rigid body-hinge-and-bar framework in 3d: four rigid bodies joined along three hinges and three bars. (b) The corresponding graph decomposes into 6 edge-disjoint spanning trees.

1.1 Preliminaries

We call a multi-graph on n vertices (k,l)-*sparse* if every subset of $n' \le n$ vertices spans at most $kn' - l$ edges, $0 \le l < 2k$; this hereditary property was first identified and shown to be matroidal by White and Whiteley[10]. A multi-graph is a k-*arborescence* if it is the union of k edge-disjoint spanning trees and a (k,a)-*arborescence* if the addition of *any* a edges results in a k-arborescence. When $0 \le a < k$, Haas[3] proved equivalence of (k,a)-arborescences with $(k,k+a)$-sparse graphs. In [7], we show that the (k,l)-pebble games precisely characterize (k,l)-sparse graphs.

A *body-and-bar framework* is a structure built from n rigid bodies connected by rigid bars placed generically; it induces a graph, with a vertex associated to each body and an edge to each bar. A remarkable theorem of Tay [9] states that the structure is (generically) rigid in dimension d if and only if the associated graph is a k-arborescence, for $k = \binom{d+1}{2}$. See Figure 1 for an example of a 3d body-and-bar framework and its corresponding graph decomposable into 6 edge-disjoint spanning trees[1]. Recski's Theorem [8] states that a graph is Laman if and only if it is a $(2,1)$-arborescence. These geometric problems motivate our interest in the purely combinatorial (k,l)-arborescences.

If bars are removed from a rigid structure (and edges from the corresponding graph), the structure becomes flexible. Parts may still be connected together in a rigid fashion; maximal such substructures form *rigid com-*

[1]Here we use the observation that hinges are equivalent to 5 bars [9].

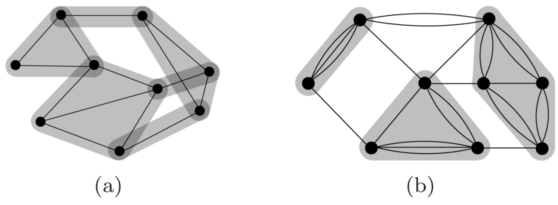


Figure 2: Rigid components of (a) a Laman graph and (b) a 3-arborescence.

ponents and correspond to maximal sub-arborescences. See Figure 2 for examples of $(2, 1)$- (Laman) and 3-arborescence components.

We identify four fundamental problems on graph rigidity. The **Decision** problem asks if G is minimally rigid. The **Extraction** problem asks for a maximal, minimally rigid subgraph of G. When weights are given for the edges of G, the **Optimization** problem asks for the maximum weight, minimally rigid subgraph of G. Given a graph with some flexibility, the **Components** problem asks for G's maximal rigid subgraphs, or components.

The pebble game. The algorithm maintains, as an additional data structure, a directed graph with pebbles placed on its vertices, on which the game is played. The edges of the input graph are considered in an arbitrary order, with each edge inserted into the additional data structure if and only if the resulting graph is (k, l)-sparse. An edge is *rejected* exactly when both endpoints lie in a common component; otherwise, it is inserted, and several existing components may combine to form a new one.

The correctness of this algorithm relies on a structure theorem in [7] which states that components are edge-disjoint, but may intersect in at most 1 vertex. For k-arborescences, components are vertex-disjoint, leading to a simple marking scheme[7] for component maintenance. However, in the general case, vertices may belong to more than one component, raising the question of whether the rejection test can be performed in $O(1)$ time. Efficient component maintenance requires additional data structures, and is the topic of this paper.

1.2 Related work

Gabow and Westermann study k-arborescences using matroid sum algorithms in [2], achieving $O(n^{3/2})$ time. Their techniques can also be applied Laman graphs; however, the running time increases to $O(n^2)$.

The pebble game algorithm for Laman graphs was devised by Jacobs and Hendrickson[5] as an elegant, easy to implement alternative to a previous algorithm of Hendrickson [4], based on bipartite matchings, and is the basis of the family of pebble games in [7]. While [5] describes the complete algorithm, it does not provide all correctness proofs. These are given in a recent paper by Berg and Jordan[1], where the only missing details pertain to the data structure needed to maintain the components. The vertex marking scheme employed in [4] (not fully analyzed there) is a special case of the approach we present in Section 3.

1.3 Union pair-find

We formally present the data structure necessary for maintaining the disjoint edge sets corresponding to components. The data structure must support a `union` operation as well as a `pair-find` query that determines if two vertices are spanned by a common component. This is a different problem from the classical union-find on disjoint sets and is presented here as *union pair-find*.

Union pair-find
Input:
- Set $V = [1..n]$ of n elements
- Set $E \subseteq \{\{u, v\} | u, v \in V\}$, where $m = |E|$

Requirements:
Dynamically maintain disjoint subsets $E_1, \dots, E_l$ of E, supporting the following operations:
- `union(`E_i, E_j`)` unions sets E_i and E_j and returns the result
- `find(`v`)` returns a list of E_i such that $v \in C_i$, where $C_i = \{x \in V | \exists y \in V \text{ such that } \{x, y\} \in E_i\}$.
- `pair-find(`u, v`)` returns `true` if there exists E_i with $u, v \in C_i$; creates and returns a new $E_{l+1} = \{\{u, v\}\}$ otherwise.

In our context (including rigidity applications), V and E are the sets of vertices and edges, respectively, of a (k, l)-sparse graph. Because components are induced subgraphs on a set of vertices, we refer to C_i as a component with edge set E_i.

2 Bounded union pair-find

The structure theorem from [7] states that components of (k, l)-arborescences may pairwise intersect in at most one vertex; thus, we first consider a restricted version of union pair-find, which we refer to as *bounded union pair-find*. Formally, the **Bounded** property requires $|C_i \cap C_j| \leq 1$, for all $C_i \neq C_j$.

For the bounded union pair-find problem, we achieve $O(1)$ for each `pair-find` operation and $O(m^2 + n^2)$ total time for all `union` operations. This will imply an $O(n^2)$ running time for all four fundamental pebble game problems, including the **Extraction** and **Components** problems on a graph with potentially $O(n^2)$ edges.

We now describe the data structures used; also see Figure 3. Elements in V are stored in vector `VV`, indexed by value; each element has a doubly-linked list pointing

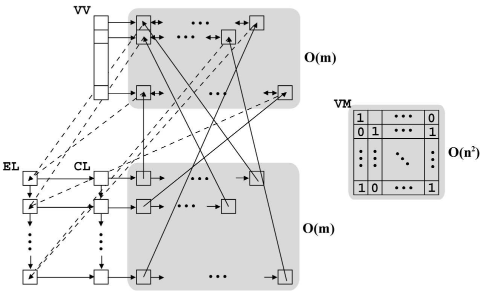


Figure 3: A representation of the data structures VV, CL, EL and VM. For clarity, only the pointers from EL to CL are included; the actual linked lists for each E_i are omitted. Dashed lines indicate pointers from VV to EL.

to the edge sets of its spanning components. The edge sets $E_1, E_2, \dots$ are maintained in linked list EL; in addition, each E_i has a pointer to the corresponding C_i. Each C_i, stored in linked list CL, is a linked list pointing to spanned elements; the pointer for spanned element v points to the entry in VV[v]'s linked list for E_i (see Figure 3). Finally, VM is an $n \times n$ matrix, whose rows and columns are indexed by the elements in V. VM[u][v] = 1 if and only if element u and element v are spanned by some common component. Initially, all entries are set to 0.

Supporting union, find and pair-find operations. For a union(E_i, E_j) operation, we must update all data structures. Since EL stores each edge set as a linked list, we simply update the pointers between the last element of E_i and the first element of E_j; E_j's entry in EL is removed. Maintenance of VV, CL and VM is slightly more complicated. First, a *marking stage* is performed, in which elements of C_i are marked. Updating VM is now accomplished by changing entries of pairs $v_i \in C_i$ and $v_j \in C_j$, where v_j is unmarked, from 0 to 1. Finally, we update CL by first walking down C_j. Entries of marked C_j elements are removed, as are the corresponding entries in VV; entries of unmarked elements are left in C_j, but corresponding pointer entries in VV are updated to point at E_i. The final step updates the last element of C_i and first element of C_j to point to each other; the linked list CL is updated to remove C_j's entry.

The find(v) operation simply returns the list of edge sets pointed to by v's entry in VV.

A pair-find(u, v) operation starts with a simple lookup of matrix VM. If the entry is 1, true is returned. Otherwise, a new singleton edge set is formed from $\{u, v\}$; this requires additional entries to EL and CL and simple updates to VV.

Time complexity analysis. We analyze the time complexity for the union operation. EL is maintained in $O(1)$ as it is a simple update of pointers to merge the corresponding linked lists. Updating CL and VV can be done in $O(m)$ time. The marking stage is a simple pass over one element of CL; this requires $O(m)$ time. Merging and updating CL and VV can also be done in $O(m)$ time by pointer updates.

As a consequence of the **Bounded** property, two vertices can be in at most one common component. Because the marking stage removes the one vertex common to C_i and C_j (if such a vertex exists), this implies that entries of VM are accessed only when a value is changed from 0 to 1. Thus, the time for updating VM over the lifetime of all union operations is $O(n^2)$. In the worst case, there are $\Theta(m)$ union operations; then the total time is $O(m^2 + n^2)$.

The find operation simply returns an entry from VV and can be performed in output-sensitive $O(t)$ time, where t is the number of components spanning the query element.

Since the pair-find operation is a simple lookup in VM, the time for one such operation is $O(1)$. Note that creation of a new edge set and corresponding component can easily be done in $O(1)$.

Space complexity analysis. There is a 1-1 correspondence between entries in the linked lists of VV and entries in the linked lists of CL. Since the edge sets are disjoint and CL maintains lists of vertices spanned by each edge set, the total size of these lists can be at most twice the size of E. Thus, the total size of CL is $O(m)$; then, the total size of VV is also $O(m)$. Finally, since VM is an $n \times n$ matrix, its size is $O(n^2)$. The total space of this data structure, then, is $O(n^2 + m) = O(n^2)$.

Pebble game analysis. Given a graph with e edges, the pebble game must maintain a dynamic set of *successfully inserted* edges, i.e., edges of a (k, l)-sparse graph. Since a (k, l)-sparse graph has $O(n)$ edges, the union pair-find maintains $m = O(n)$ edges; this implies $O(n^2)$ total time for union operations. Since a pair-find operation is performed for each of the $e = O(n^2)$ edges in the input graph, the total time for pair-find operations is $O(n^2)$ as well.

3 Reducing the space complexity

In this section, we present a compact approach that removes the **Bounded** restriction and uses only $O(m+n)$ space. In the worst case, the compact implementation requires time $\Theta(m)$ for each pair-find operation; if we require a specific ordering on $\Omega(n^2)$ pair-find queries, we retain an amortized time of $O(1)$ for each query. The running time of union remains unchanged.

Compact data structures and space complexity. The compact implementation removes CL and replaces VM with a value LV and a vector MV indexed by V. LV represents the left operand of the most recent pair-find operation; MV maintains the vth row of VM. The space complexity is reduced to $O(n + m)$.

Supporting `union`, `find` and `pair-find` operations. For a `union(`E_i`,`E_j`)` operation, when `LV` is spanned by the resulting set, updating `MV` is accomplished by a single pass over the new set. A `pair-find(`u`,`v`)` first performs a check to determine if `LV` $= u$. If so, we return `true` when `MV`$[u] = 1$ and a new component otherwise. If `LV` $\neq u$, we update `MV` by walking over `EL`, then answer the query.

Time complexity. The running times of `union` and `find` remain unchanged.

We call a `pair-find(`u`,`v`)` query a *miss* when `LV` $\neq u$. It is straightforward to see that the running time of `pair-find` is $O(m)$ for a miss and $O(1)$ otherwise. It follows that, for a sequence of p `pair-find` queries with s misses, the total running time is $O(ms + (p - s)) = O(ms+p)$. When the `pair-find`s have $O(n)$ misses, the total cost becomes $O(mn + p)$; consider, for instance, restricting the queries to be ordered by left operand.

Pebble game analysis. The analysis on the `union` operations remains unchanged. On an input graph with e edges, we can satisfy the restriction of $O(n)$ misses by attempting to insert the edges in an order corresponding to breadth-first exploration. Recall that, for the pebble game, the *inserted* edges form a (k, l)-sparse graph with $O(n)$ edges, resulting in a union pair-find data structure with $m = O(n)$; then the total cost for the `pair-find` queries is $O(n^2)$.

4 Conclusion

We have presented a new data structure problem called union pair-find. Motivated by achieving efficient and simple algorithms for the **Decision**, **Extraction**, **Optimization** and **Components** problems for rigid graphs, union pair-find maintains disjoint edge sets corresponding to rigid components. While `union` operations of the disjoint sets must be supported, the application requires efficient time complexity for `pair-find` queries. Therefore, this paper proposes two approaches to union pair-find which concentrate on the complexity of `pair-find`.

Both approaches result in $O(n^2)$ time pebble games for the **Decision**, **Extraction** and **Components** problems. While Section 3's approach requires the queries to be given by breadth first exploration, the matroidal properties of rigid graphs[10] imply that this additional requirement does not affect the correctness of the pebble games. The **Optimization** problem can be solved by the greedy algorithm, thus dictating an order on the `pair-find` queries. Section 2's approach is then required to achieve $O(n^2)$ complexity, as it is efficient for any sequence of `pair-find` queries.

4.1 Open problems

The introduction of union pair-find and two approaches for its solution leads to several interesting open problems. Section 2 gives a quadratic space solution to the bounded version, while providing constant time `pair-find` queries. The compact approach of Section 3 solves the general problem, but is only time efficient when the ordering of `pair-find`s is flexible. Whether a linear space, constant time `pair-find` solution exists for the general union pair-find is an open problem.

For applications in rigidity, we have focused on efficient `pair-find` queries at the expense of `union` operations. If we relax the efficiency requirements for `pair-find`, can we reduce the complexity for `union` operations? What sort of tradeoff is there between the two operations?

References

[1] A. R. Berg and T. Jordan. Algorithms for graph rigidity and scene analysis. In G. D. Battista and U. Zwick, editors, *ESA*, volume 2832 of *Lecture Notes in Computer Science*. Algorithms - ESA 2003, 11th Annual European Symposium, Budapest,Hungary, Springer, 2003.

[2] H. Gabow and H. Westermann. Forests, frames, and games: algorithms for matroid sums and applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 407–421. ACM Press, 1988.

[3] R. Haas. Characterizations of arboricity of graphs. *Ars Combinatorica*, 63:129–137, 2002.

[4] B. Hendrickson. *The molecule problem: determining conformation from pairwise distances.* PhD thesis, Cornell University, 1991.

[5] D. J. Jacobs and B. Hendrickson. An algorithm for two dimensional rigidity percolation: The pebble game. *J. Comput. Phys.*, 137:346–365, 1997.

[6] D. J. Jacobs, L. A. Kuhn, and M. F. Thorpe. Flexible and rigid regions in proteins. In *Rigidity Theory and Applications*, pages 357–384. Kluwer Academic/Plenum Publishing, NY., 1999.

[7] A. Lee and I. Streinu. Pebble game algorithms and (k, l)-sparse graphs. *Accepted to EuroComb '05*, 2005.

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

[9] T.-S. Tay. Rigidity of multigraphs I: linking rigid bodies in n-space. *Journal of Combinatorial Theory Series*, B 26:95–112, 1984.

[10] W. Whiteley. Some matroids from discrete applied geometry. In J. O. J. Bonin and B. Servatius, editors, *Matroid Theory*, volume 197 of *Contemporary Mathematics*, pages 171–311. American Mathematical Society, 1996.