Dynamic Maintenance of Kinematic Structures*

D. HALPERIN J-C. LATOMBE R. MOTWANI

Abstract

We consider the following dynamic data structure problem. Given a collection of rigid bodies moving in 3-dimensional space and hinged together in a kinematic structure, our goal is to efficiently maintain a data structure that allows us to quickly answer range queries as the bodies move. This kinematic data structure problem arises in a variety of applications such as conformational search in molecular biology, simulation of hyper-redundant robots, collision detection, and computer animation. We study several models for dynamic maintenance of such structures and devise algorithms under these models. We obtain tight results on the worst-case, amortized, and randomized complexity of the data structure problem. For the offline version of the problem, we establish NP-hardness and provide efficient approximation algorithms.

Key words. data structure, approximation algorithm, kinematics, range queries, conformational search, molecular biology, robotics, collision detection, computer animation

AMS subject classifications. 68P05, 68Q25, 68U05, 70B15

Abbreviated Title: Dynamic Maintenance of Kinematic Structures

*Computer Science Department, Stanford University, Stanford, CA 94305. *E-mail:* {*halperin,latombe,motwani*}*@cs.stanford.edu*. This work was funded by ARO MURI Grant DAAH04-96-1-007. The application to molecular biology is supported by a grant from Pfizer Central Research. The third author is also being supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

1 Introduction

In this paper we study the following dynamic data structure problem: Given an articulated linkage in three-dimensional space, i.e., a collection of bodies (links) connected by joints, efficiently maintain a data structure allowing quick answers to range queries, as the bodies move (the joint parameters change). To be more concrete, let us assume that the data structure is a grid representation of the space occupied by the linkage, for some specified values of the joint parameters; for instance, it may store each grid cube intersected by the linkage in a hash-table. A query typically specifies a region in space (e.g., the region occupied by an obstacle) and asks whether the linkage intersects this region. Similarly, a query can ask whether the linkage lies within a certain distance from an obstacle, by enlarging the region occupied by the obstacle. The data structure is then used to quickly select small subsets of the bodies to which exact intersection/-distance algorithms will be applied in order to answer the query. The initial construction of the data structure, for a given set of values of the joint parameters, is a well-studied problem that we will not address here; in fact, we do not assume a particular data structure, nor a specific technique to construct it, but we assume that the operations they support have certain costs. A more difficult, and novel, problem is to efficiently update this data structure as the joint parameters change. This problem is the subject of our paper.

To get a better understanding of our problem, consider a serial linkage with n links, such as a classical manipulator arm. We can construct n data substructures, e.g., n hash-tables each representing a grid occupancy by one link, in a coordinate system attached to the link. These rigid substructures need not be updated when joint parameters change, since each update can be viewed as a transformation of the coordinate system and stored as such. However, when a query is received, each substructure must be queried separately, after having computed the position of the query region relative to each of the links. This takes time proportional to the number n of substructures, a cost that we wish to avoid. Indeed, suppose that we receive a long sequence of updates that all modify the same joint parameter. In this limited case, a much better strategy is to decompose the linkage into two sublinkages at the joint being modified and build one data structure for each sublinkage. The cost of a query is then only twice the cost we would have incurred if we had computed a single data structure for the entire linkage. If we later receive another sequence of updates modifying another joint parameter, we can break and merge substructures accordingly. For a more involved sequence of updates and queries, this approach leads us to represent the entire linkage by a data structure that is a *dynamic* collection of substructures representing distinct sublinkages. We study strategies for maintaining the global data structure such that a sequence of updates and queries is answered in minimal time.

A possible application of our data structure is to improving the efficiency of path planners for robots with many links. Except in very limited cases, it is computationally infeasible to compute an explicit representation of the set of collision-free configurations – the free space – of such a robot. Instead, as discussed in [2], one can approximate the geometry and connectivity of this set by randomly sampling the configuration space. The configurations picked at random are checked for collision and the collision-free ones are retained as milestones. The distance (in the workspace) between the obstacles and the robot placed at a milestone is used to decide which pairs of milestones can safely be connected by straight paths (in the configuration space), yielding a network of milestones called a probabilistic roadmap. Several successful planners are based on this general scheme [3, 5, 20, 22, 23, 34]. However, they all spend most of their running time (typically, over 90%) computing distances or checking collision. Hence, faster distance computation will directly benefit these planners. Potential fields are also used by many planners, randomized [3] or otherwise [13], as a heuristic cost function guiding the search for paths of multi-link robots. These fields usually depend on the distance between the robot and the obstacles, e.g., they grow to infinity as this distance tends to 0 [24]. More efficient distance computation will speed up the potential field calculation. Note that different data structures have been proposed to represent the space occupied by robots for the purpose of collision checking and distance computation operations: single-level and hierarchical occupancy grids (e.g., [12, 21]), bounding boxes (e.g., [35]), and spherical approximations (e.g., [36]). The framework presented in this paper can be applied to all these representations.

Another application domain, molecular biology, motivated our kinematic data structure problem in the first place. The geometry of a molecule can be described by a collection of spheres, each representing an atom [10, 31]. As noted by Halperin and Overmars [19], the spheres fulfill certain properties that allow the construction of an efficient data structure (an occupancy grid represented by a hash-table) to answer range queries, where the range is a sphere whose radius is at most c times larger than the radius of the largest sphere in the original set, for some fixed constant c. For a molecule of n atoms, the structure uses $O(n)$ storage, requires $O(n)$ randomized preprocessing time, and allows a query to be answered in $O(1)$ time. This structure was proposed [19] for answering queries when the molecule is at a fixed configuration. However, due to possible rotation about bonds connecting atoms, certain molecules are highly flexible, e.g., drug ligands. Our results are relevant to the representation of such molecules [18].

The kinematic data structure problem is related to dynamic data structure problems previously studied in the computational geometry literature [8, 30, 32, 33]. However, in most of the earlier work, the dynamization is caused by addition or removal of objects. The novel aspect of our problem is that the set of objects is fixed and the dynamization is due to their motion. Another related problem is the representation of multi-body systems (e.g., a collection of particles) for force calculation [1]. As in our problem, the bodies move but the set of bodies is fixed. However, due to the more involved mathematical form of the constraints on the relative motions of the bodies, prevailing solutions recompute the data structure from scratch at each step.

In the next section, Section 2, we present a formal model for our data structuring problem. In Section 3 we start by considering the case of path-like (serial) linkages and show that the complexity of this problem is $\Theta(\sqrt{n})$ for paths of length n when the measure of interest is the worst-case cost of an operation, i.e., we present an *optimal* strategy for maintaining the data structure so as to minimize over all input sequences of queries and updates the maximum time required by an operation. Our bounds apply unchanged to the amortized and randomized time measures. These results are extended to the case of trees in Section 4 based upon a novel tree decomposition strategy that may be of independent interest. We define a balance number κ for a tree T, and show that the worst-case and amortized complexity is $\Theta(\kappa)$. In some applications, such as sampling of configuration spaces in robot path planning or conformation search in molecular biology, the problem has an essentially offline nature, in that the entire sequence of queries and updates may be known in advance. In Sections 5 and 6 we discuss the *offline* version of the problem under two different cost measures. We show that already in the case of a path-like linkage, devising the best strategy is NP-hard, and we present efficient approximation algorithms for the problem. The approximation ratio achieved is bounded by 1.75 for one cost measure, and by $O(\log n)$ for the other. Some directions for further research are presented in Section 7. A particularly interesting direction of research is to apply the framework of competitive algorithms for online problems and we mention some preliminary results.

2 The Abstract Data Structure

In this section we give a formal description of our data structuring problem, henceforth referred to as the *kinematic data structure problem*. We are required to maintain a data structure D representing the space occupied by an articulated linkage L of n links with no closed loops. A real value, called the joint parameter, is associated with each joint and specifies the relative position of the two links connected by that joint. Since the linkage L is loop-free, it may be viewed as an abstract tree. It will be convenient in the later sections to switch to graph-theoretic terminology for the linkage, as follows. The linkage L is represented by a tree $T(V, E)$, where the links of L map to the vertices $V = \{v_1, v_2, \ldots, v_n\}$ and, for a joint connecting links i and j, there is an edge (v_i, v_j) in E connecting the corresponding vertices v_i and v_j. In the case of a serial linkage, the tree reduces to a single path.

The data structure D is a dynamic collection of substructures, each representing the space currently occupied by a connected subset of L, i.e., a subtree of T. Two distinct substructures represent disjoint subsets. Together, the substructures in D represent the entire linkage in that the corresponding subtrees are a decomposition of the tree T. The data structure D supports two operations, UPDATE and QUERY:

UPDATE(v_i, v_j, q) specifies a joint (edge) and requires that the corresponding joint parameter be changed to q.

QUERY requires a separate examination of all the substructures in D.

The details of the examination to be carried out by QUERY can vary from one task to another. A more general version of this operation would permit the specification of a subset of L and ask for the examination of only the substructures representing this subset. For simplification, we assume in this paper that QUERY always examines the entire structure. The goal of the algorithm maintaining D is to maintain the collection of substructures in D so as to be able to handle queries, given an input sequence with an arbitrary mix of UPDATE and QUERY operations.

As mentioned above, every data substructure in D corresponds to a subtree of T. The various subtrees in D may be viewed as a decomposition of T induced by the removal of a set of edges in E. Indeed, we will assume that at any time each edge in E is labeled as being BROKEN or MERGED, and that the decomposition of T is implicitly defined as being induced by the removal of the BROKEN edges. However, we do not actually remove any of the tree edges since in our model the topology of the tree or the linkage remains fixed over time. To update D, the algorithm uses two primitive operations, BREAK and MERGE, that may be applied to any edge (v_i, v_j) in T; if applied to a non-existent edge, the operations report failure.

BREAK(v_i, v_j)**:** If v_i and v_j belong to the same substructure S of D, BREAK decomposes the sublinkage represented by S into two, by labeling the edge (v_i, v_j) as being BROKEN and partitioning S into two new substructures representing the resulting sublinkages. If v_i and v_j are not in the same substructure (i.e., (v_i, v_j) is already BROKEN), then BREAK leaves D unchanged.

MERGE(v_i, v_j)**:** If v_i and v_j belong to two distinct substructures S_i and S_j, MERGE combines them into a single substructure representing the union of the two corresponding sublinkages connected by the joint corresponding to (v_i, v_j). If v_i and v_j are represented in the same substructure (i.e., (v_i, v_j) is already MERGED), then MERGE leaves D unchanged.

Essentially, a strategy for maintaining this data structure will control the partition of D into the various substructures so as to minimize the overall cost of processing the UPDATE and QUERY operations. For example, if the input sequence contains only an extremely small number of queries, the best strategy would be BREAK all edges and thereby decompose the tree into isolated vertices; conversely, when the input sequence contains only an extremely small number of updates, the best strategy would be to not BREAK any edge at all. Of course, in the applications we need to support a possibly arbitrary mix of these two operations and our paper is concerned with devising optimal strategies for adapting to that situation.

In order to be able to state precise results, we will assume throughout the rest of the paper that the cost of the primitive operations is as follows. Note that the cost models will change with the precise application, but our results should carry over after suitable modifications to the bounds obtained.

- BREAK(v_i, v_j) takes time $O(1)$ if the edge (v_i, v_j) is already BROKEN, and time $O(n_{ij})$ if it is MERGED, where n_{ij} is the size (number of vertices) of the subtree containing the two end-points v_i and v_j.
- MERGE(v_i, v_j) takes time $O(1)$ if the edge (v_i, v_j) is already MERGED, and time $O(n_{ij})$ if it is BROKEN, where n_{ij} is the size (number of vertices) of the resulting subtree with the two end-points v_i and v_j.

We will refer to the cost of BREAK and MERGE as described above as the TOTAL cost measure. An alternative cost measure, called MIN, is obtained by defining n_{ij} as the size of the smaller substructure (subtree) produced by BREAK or merged by MERGE. Depending on the application at hand, one of these measures may be more appropriate than the other. The MIN measure corresponds to the situation where a structure can be decomposed into two substructures by deleting the elements of the smaller substructure and then computing its representation, and two substructures can be MERGED into a common structure by inserting the elements of the smaller substructure into the larger substructure. Conversely, the TOTAL cost measure corresponds to the situation where the BREAK and MERGE operations involve completely destroying the old structures and recomputing the new structures from scratch.

The given costs correspond exactly to the data structure described by Halperin and Overmars [19]. This data structure is an occupancy grid stored in a hash-table representing a molecule modeled as a collection of spheres. The same data structure could be used to represent a linkage. The costs defined above assume that the links, as well as the query region, have approximately the same shape and size, and that there is not too much steric overlap between the links. The condition on the links is fulfilled by some modular robots [9, 14, 37]; when it is not satisfied, one may arbitrarily cut the larger links into smaller ones connected by fixed joints. The condition on the query region is often reasonable, since one is rarely interested in computing the exact distance between the robot and an obstacle, when this distance is large. If the two conditions are not satisfied, the costs of the operations on the data structure may differ from those given above; then, the results presented in this paper would require to be reformulated appropriately.

The UPDATE operation can be implemented as follows: UPDATE(v_i, v_j, q) is implemented by first performing BREAK(v_i, v_j) and then storing the change q in the joint parameter at the edge (v_i, v_j). Thus, an UPDATE takes the same time as a BREAK. Of course, a series of updates not interspersed with any query can be accumulated and applied in a batch mode when a query finally arrives.

Finally, a QUERY takes time $O(k)$, where k is the number of BROKEN edges in T or the number of substructures in D. To justify this, we need to explain how a query is performed with our data structure. For clarity, let us assume that we are dealing with a serial linkage, i.e., the case where the corresponding tree is a path; the more general case can be dealt with in a similar fashion. Let S_1 denote the static structure (a hash table in the example above) containing (a representation of) the link v_1, let S_2 denote the next static structure along the path, and so on. We assume that the link v_1 is fixed in 3-space. Each static structure has a coordinate frame attached to it in which the links of the structure are described. The coordinate frame attached to S_1 is the universal frame in which the query regions will be given. For every pair of successive static structures S_i and S_{i+1}, we maintain a rigid transformation T_i which transforms points described in the frame of S_i so as to be described in the frame of S_{i+1}.

Given a query region Q, we query the structure S_1 with Q. Next, we transform Q into Q', using the transformation T_1, we query S_2 with Q', and so on. The final answer is easily deduced from the answers in all the structures S_i. For a path consisting of k static structures, the cost of the query is clearly $O(k)$.

Note that to update the joint value of an edge that lies between two static structures (i.e., not internal to any static structure), we simply update the transformation between the two structures. This takes $O(1)$ time, assuming that the update is applied to a BROKEN edge; as stated earlier, we will always BREAK an edge before applying an update.

3 Restriction to Paths

In this section we characterize the complexity of the update and query operations with respect to each of the following time measures: worst-case, amortized, and randomized. We begin by considering the case of paths and defer the extension to trees till the next section.

Theorem 1 *There is an algorithm for maintaining the kinematic data structure at a worst-case cost of $O(\sqrt{n})$ per operation, under both MIN and TOTAL cost measures.*

Proof: The idea is very simple: the algorithm chooses the initial state to be one where the BROKEN edges are spaced regularly along the path at intervals of $\sqrt{n}$. This state remains fixed throughout the processing of the input sequence. It is clear that any query can be answered in time $O(\sqrt{n})$ since that is the total number of BROKEN edges in the entire path. Further, any update operation has cost $O(\sqrt{n})$ under both MIN and TOTAL cost measures, since all subpaths are of length $\sqrt{n}$. Note that the algorithm will BREAK an edge for an update operation, but will then MERGE it right after that unless the edge is one of the initially BROKEN edges. The cost of the MERGE operation at most doubles the cost of the UPDATE operation. ■

As shown below, this bound is tight even for amortized and randomized time measures.

Theorem 2 *Any algorithm for maintaining the kinematic data structure must have a worst-case cost per operation that is $\Omega(\sqrt{n})$. The same holds for both* amortized *and* randomized *time measures, under both MIN and TOTAL cost measures.*

Proof: We first prove the worst-case lower bound using an adversarial approach. At any time, the adversary examines the state of the data structure being maintained by the algorithm. If the number of BROKEN edges exceeds $\sqrt{n}$, it requests a QUERY operation and this incurs a cost $\Omega(\sqrt{n})$. On the other hand, if the number of BROKEN edges is fewer than $\sqrt{n}$, then there exists a subpath with more than $\sqrt{n}$ vertices. In that case, the adversary inputs an operation which involves breaking the middle-most edge in this subpath. This costs $\Omega(\sqrt{n})$ regardless of whether we are using the MIN or the TOTAL cost measure.

Notice that this adversary is completely impervious to the strategy of the algorithm and can create an input sequence of arbitrary length where *each* operation costs $\Omega(\sqrt{n})$. Quite clearly then, the lower bound applies unchanged under the *amortized* time measure.

Finally, we extend our lower bound to the randomized case. Here we are allowing the algorithm to be randomized, and now the adversary can no longer look at the state of the data structure when choosing each operation in the input sequence. We modify the adversary strategy as follows: at each step, the adversary chooses to either supply an UPDATE operation or a QUERY operation, with equal probability; if it chooses an UPDATE operation, the edge to be updated is chosen uniformly at random. Suppose that the data structure has more than $\sqrt{n}$ BROKEN edges, then with probability $1/2$, the QUERY operation causes a cost of $\Omega(\sqrt{n})$. On the other hand, when the data structure has fewer than $\sqrt{n}$ BROKEN edges, the update operation is chosen with probability $1/2$ and the edge involved in this operation lies in a subpath of *expected* length $\Omega(\sqrt{n})$. It follows that the expected cost of each operation is $\Omega(\sqrt{n})$. ■

One way to get around the worst-case lower bound proved above is to consider the offline setting and the minimization of the total cost. Another approach would be to consider the online setting but employing competitive analysis, i.e., comparing the online algorithm's cost to the optimal offline cost. Both types of algorithms and analysis

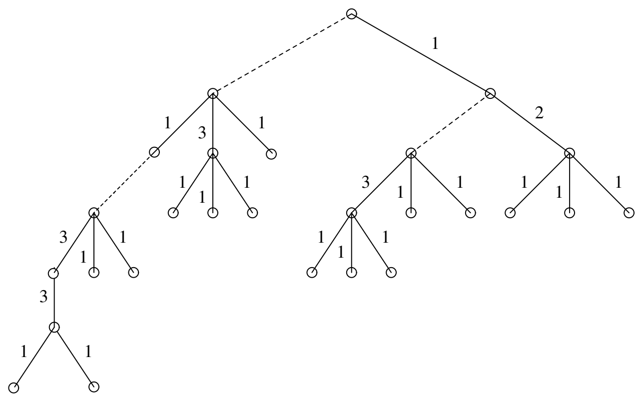


Figure 1: *A 4-balanced decomposition of a tree with balance number 4. The edges to be removed are shown as dashed lines and the resulting heaviness of each other edge is as shown.*

have their uses and applications. In Section 5 and 6 we will consider the offline setting in detail. The online setting is relatively open and we only have some preliminary results discussed in Section 7.

4 Generalization to Trees

To understand the case of trees, it is instructive to first examine the other extreme from paths, i.e., stars. Unlike in the case of paths, it is impossible to find a small number of edges whose removal decomposes the star into small subtrees and so it may seem that we will have to pay a significantly higher cost per operation. However, upon closer examination, it turns out that stars are much easier than paths provided we work with the MIN cost measure rather than the TOTAL cost measure (which will be considered later in Section 4.3). One reason to focus on the MIN cost measure is that the cost of an UPDATE operation in a star is $O(1)$ for each edge, even if it is MERGED. Thus, for stars, the right solution is to not BREAK any edges at all, leading to $O(1)$ cost for a QUERY and $O(1)$ cost for an UPDATE.

Motivated by this insight, we make the following definition:

Definition 1 *In a tree T, the two subtrees resulting from the removal of an edge (v_i, v_j) are denoted T_i and T_j, according to which of these contains the two endpoints v_i and v_j. The* heaviness *of the edge is defined to be* $\min\{|T_i|, |T_j|\}$*, where $|T|$ denotes the size (number of vertices) of a tree T. An edge of heaviness k is said to be* k'-heavy *for any $k' \le k$.*

Basically, the "heaviness" of an edge is the cost of breaking or updating the edge. We extend the notion of heaviness to the entire tree.

Definition 2 *The* heaviness *of a tree T is the maximum heaviness of an edge in it. A tree T of heaviness k is said to be* k'-heavy *for all $k' \le k$.*

Based on this, we define the notion of balance number of a tree.

Definition 3 *The* balance number *κ of a tree T is the smallest integer k such that the removal of $k-1$ edges from T decomposes it into k subtrees $\tau_1, \tau_2, \ldots, \tau_k$ none of which is k-heavy. Such a decomposition is called a κ-balanced decomposition of T.*

Note that an edge that is not k-heavy in some τ_i could be k-heavy in the original tree T. In the above definition we are considering the heaviness of the edges in each τ_i with respect to that tree itself. Refer to Figure 1 for an example of a 4-balanced decomposition of a tree.

We now relate the balance number to the complexity of the kinematic data structure problem.

Theorem 3 *Let T be a tree with balance number κ. There is an algorithm that maintains the kinematic data structure at a worst-case cost of $O(\kappa)$ per operation, under the MIN cost measure.*

Proof: Let U be a set of at most $\kappa - 1$ edges in T which gives a κ-balanced decomposition into trees that are not κ-heavy. The idea is to keep the edges in U BROKEN. The cost of a QUERY is clearly at most κ. An UPDATE is also going to cost at most κ since in each of the induced subtrees, none of the edges are κ-heavy, ■

This bound is tight, even for the amortized time measure.

Theorem 4 *Let T be a tree with balance number κ. Any algorithm for maintaining the kinematic data structure must have a worst-case cost per operation that is $\Omega(\kappa)$. The same holds for the* amortized *time measure.*

Proof: We first prove the worst-case lower bound using an adversarial approach. At any time, the adversary examines the state of the data structure being maintained by the algorithm. If the number of BROKEN edges is at least $\kappa - 1$, it inputs a QUERY operation and this incurs a cost $\Omega(\kappa)$. On the other hand, if the number of BROKEN edges is strictly less than $\kappa - 1$, then we claim that the resulting decomposition contains a subtree τ_j which is $(\kappa - 1)$-heavy. The claim follows from the observation that otherwise we would have at most $\kappa - 2$ deleted edges yielding a decomposition in which no resulting subtree is $(\kappa - 1)$-heavy, implying that the balance number of T is at most $\kappa - 1$ and thereby contradicting the assumption that T has balance number κ. By this claim, the adversary can identify a $(\kappa - 1)$-heavy subtree τ_j that must contain a $(\kappa - 1)$-heavy edge e — the adversary then inputs an UPDATE operation involving e and this costs $\Omega(\kappa)$.

Notice that this adversary is completely impervious to the strategy of the algorithm and can create an input sequence of an arbitrary length where *each* operation costs $\Omega(\kappa)$. Quite clearly then, the lower bound carries over to the *amortized* cost of operations without any changes. ■

We have related the balance number κ to the data structure problem but we still have to devise an algorithm for computing the balance number κ and, in fact, for finding the $\kappa - 1$ edges that induce a balanced decomposition. This is not immediately obvious. The following lemma provides some insight.

Lemma 1 *For any tree T and any k, the set of k-heavy edges in T form a connected subtree of T.*

Proof: Let $e_1 = (v_1, v_2)$ and $e_2 = (v_3, v_4)$ be two k-heavy edges in T. Assume that v_2 is closer to e_2 than v_1, and that v_3 is closer to e_1 than v_4. We will show that the edges on the (unique) path from v_2 to v_3 are all k-heavy, and this will imply the desired result.

Suppose that the removal of the edge e_1 from T creates a subtree T_1 containing v_1 and a subtree T_2 containing v_2; similarly, the removal of the edge e_2 from T creates a subtree T_3 containing v_3 and a subtree T_4 containing v_4. Clearly, each of T_1, T_2, T_3, and T_4 has size at least k, since e_1 and e_2 are both assumed to be k-heavy.

Let $e = (v_5, v_6)$ be an edge on the path from v_2 to v_3, and suppose that the removal of the edge e from T creates a subtree T_5 containing v_5 and a subtree T_6 containing v_6. Clearly, T_1 is contained in T_5 and T_4 is contained in T_6, implying that both T_5 and T_6 have size at least k; therefore, e is k-heavy. ■

4.1 Algorithm for Balanced Decomposition

We now describe a linear-time algorithm, Algorithm DFS-Decompose, for computing a κ-balanced decomposition of a tree with balance number κ. This algorithm assumes that the value of the balance number κ is provided along with the input tree; when κ is unknown, a binary search for the value of κ can be performed at the cost of increasing the running time to $O(n \log \kappa)$. Note that a slightly simpler version of this algorithm can be shown to compute a (2κ)-balanced decomposition of a tree with balance number κ; we omit the details.

Designate any arbitrary vertex r of T as its root. The algorithm is based on performing a depth-first search (dfs) of T starting at the root r. We assume that all edges are directed down from the root towards the leaves. The decision to delete an edge (u, v) directed from u to its child v is *usually* made when the dfs completes the traversal of all the vertices below u and is ready to leave u to move back up to the rest of the tree; in the sequel, we will refer to this as the *final return* to u. We assume throughout that the number of vertices remaining in the tree (as edges and subtrees are cut away) does not fall below 2κ; clearly, a tree with fewer than 2κ vertices cannot be κ-heavy and the algorithm can be terminated if the number of vertices ever falls below 2κ. Throughout the proof, the term "residual tree" will denote the subtree of T that remains at the point in time under consideration, with the removal of subtrees by deleting edges at earlier times in the dfs algorithm. The algorithm can track the total number of vertices remaining in the residual tree and can terminate whenever this falls below 2κ.

In the general case, suppose that the algorithm has just completed the traversal of the entire subtree rooted below a vertex u, and has just made a final return to u. Let the children of u be the vertices $v_1, \ldots, v_r$ and, for $1 \leq i \leq r$,

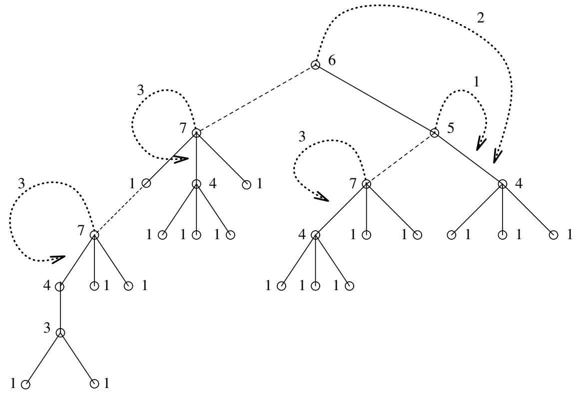


Figure 2: *Algorithm DFS-Decompose applied to a tree with balance number 4. The deleted edges are shown as dashed lines. For each vertex i we label it with the n_i value upon the return from that vertex to its parent. The pointers p_i at marked vertices are shown as dotted arrows labeled with the corresponding m_i values.*

let $e_i = (u, v_i)$. We will assume that the algorithm has recursively computed certain labels for each of the vertices $v_1, \ldots, v_r$. First, there is the label n_i for v_i which is the number of descendants of v_i (including itself) in the current residual tree. We assume that the children $v_1, \ldots, v_r$ are indexed in non-increasing order of n_i. In addition, some of these vertices may be "marked" and for each marked vertex, the algorithm associates a pointer to an edge p_i in the subtree rooted at that vertex and associates with the pointer an additional label m_i.

We will ensure in the algorithm that: *after the final return to a vertex u, the subtree rooted at u does not contain any κ-heavy edges.* This may have required deleting edges earlier while visiting the descendants of u. In fact, the algorithm's goal will be to try to ensure the following stronger condition: *after the final return to u, none of the subtrees rooted at a child v_i of u should have more than $\kappa - 1$ vertices.* It is easy to verify that if the root of a tree satisfies the second condition, then it satisfies the first condition, implying that the tree is not κ-heavy; however, the converse need not be true. As will soon become clear, the algorithm will sometimes be unable to achieve the second goal and then *exactly one* subtree (rooted at some v_i) with more than $\kappa - 1$ vertices will be allowed to remain, and that is precisely when a vertex v_i will be marked. At a marked vertex v_i, we store a pointer to an edge p_i directed from one of its descendants x_i to a vertex y_i such that y_i is the "lowest" descendant of v_i that has more than $\kappa - 1$ descendants (and hence is marked). The pointer's label m_i is the number of descendants of v_i that are non-descendants of y_i. The algorithm will ensure as an invariant that $m_i < \kappa - 1$.

If a child vertex v_i is marked, we were unable to meet our goal of ensuring that the final return to u should be with v_i's subtree having no more than $\kappa - 1$ vertices; however, we do not violate the requirement that upon final return to u the subtree rooted at u does not contain any κ-heavy edges. To understand how this is achieved, consider the decomposition of the edges of the subtree rooted at v_i into those contained in the tree rooted at y_i, and the remaining edges which span exactly m_i vertices. The former set of edges cannot contain any κ-heavy edges since by definition y_i is the lowest descendant of v_i with more than $\kappa - 1$ descendants and so cannot have any marked children. If the latter set of edges were to contain a κ-heavy edge, then we would delete the edge from u to v_i; observe that now the edges in the latter set could not be κ-heavy either since one of the two subtrees obtained by their deletion must have at most m_i vertices and $m_i < \kappa - 1$ by definition, implying that the subtree rooted at v_i is not κ-heavy. Note that, in general, it is entirely possible that an optimal solution would not delete the edge (u, v_i), preferring instead to delete an edge not contained in the subtree rooted at u to obtain a more fruitful deletion. As will soon become clear, our algorithm will only delete the edge (u, v_i) when absolutely essential for ensuring the invariants, and in that case we will be able to argue that some optimal solution must also delete this edge.

We are now ready to describe the recursive dfs-based computation. Suppose that at some point the algorithm has just made a final return to a vertex u. If $n_1 < \kappa - 1$, then the algorithm does not delete any of the edges $e_1, \ldots, e_r$; instead, it merely sets $n_u = 1 + \sum_{i=1}^{r} n_i$ and returns to u's parent. Otherwise, let s be the index such that $n_s \geq \kappa - 1$ and $n_{s+1} < \kappa - 1$; recall, the vertices v_i are in non-increasing order of number of descendants n_i. The algorithm has two stages of deletion: the first stage, Stage A, is invoked only in the case $s \geq 2$; the second stage, Stage B, is applied in all cases, including the case where $s = 1$ and Stage A is applicable. During Stage A, the algorithm will cut away $s - 1$ of the subtrees of u, and thus this stage always produces a residual tree with $s = 1$ to which the algorithm then applies Stage B.

In Stage A, we have $s \geq 2$. Then out of the edges $e_1, \ldots, e_s$, the algorithm deletes all but one, say e_k. The index k is chosen as follows: if one of the vertices $v_1, \ldots, v_s$ is unmarked, then choose k such that v_k is an unmarked vertex; otherwise, all of them are marked, and the algorithm selects k such that m_k is the smallest amongst $m_1, \ldots, m_k$. Then, the algorithm deletes the edges $\{e_1, \ldots, e_s\} \setminus \{e_k\}$ and does not delete e_k or the edges $e_{s+1}, \ldots, e_r$. Notice that the resulting situation at u is the same as in the case where $s = 1$.

It remains to describe Stage B which the algorithm enters with $s = 1$. Since $s = 1$ implies that $n_i < \kappa - 1$ for $i > 1$, only v_1 can be a marked vertex. The algorithm computes $d = 1 + \sum_{i=2}^{r} n_i$ as the number of descendants of u (including itself) that are not descendants of v_1 (including itself). When v_1 is indeed a marked vertex, it defines $p = p_1$ and $m = m_1$; on the other hand, when v_1 is not marked, it defines $p = (u, v_1)$ and $m = 0$. If $m + d < \kappa - 1$, then the algorithm does not delete any edges and, before proceeding upwards from u, it labels u as follows: the label n_u is computed as $1 + \sum_{i=1}^{r} n_i$; u is marked; the pointer at u is to the edge p; and, the additional label $m_u = m + d$ is associated with this pointer. Conversely, if $m + d \geq \kappa - 1$, then the algorithm deletes the edge $e_1 = (u, v_1)$, updates n_u to be $1 + \sum_{i=2}^{r} n_i$, leaves u unmarked, and returns to u's parent.

It is easily verified that if u is marked, then the associated labels p_u and m_u satisfy the properties mentioned earlier. We remark that the algorithm does not really need to associate the pointer p with a marked vertex as the value of the pointer is never used; however, it will facilitate our analysis to be able to refer to the edge to which p points and that is the only reason why we ensure that the algorithm maintains such pointers.

This completes the description of the Algorithm DFS-Decompose. In Figure 2, we illustrate the execution of this algorithm on the tree shown in Figure 1.

4.2 Analyzing the Algorithm

Our goal in this section is to establish the following theorem.

Theorem 5 *Algorithm DFS-Decompose runs in time $O(n)$ on an input tree T with n nodes, and for any given integer κ, it returns a κ-balanced decomposition of T if it has balance number κ, and returns* FAILURE *otherwise.*

We analyze the performance and running time of Algorithm DFS-Decompose in the following sequence of lemmas. It should be clear that the lemmas combine to imply the proof of Theorem 5.

Lemma 2 *Algorithm DFS-Decompose runs in time $O(n)$.*

Proof: It is easy to verify that this algorithm runs in linear time. The depth-first search by itself runs in linear time. The additional work performed at a vertex u in maintaining vertex labels and choosing edges to delete can be charged to u and its incident edges such that only a constant amount of work is assigned to each vertex and edge. ■

Next, we show that that this algorithm produces a valid decomposition, i.e., none of the resulting subtrees are κ-heavy.

Lemma 3 *Algorithm DFS-Decompose produces a decomposition of the tree T such that none of the subtrees in the decomposition are κ-heavy.*

Proof: To this end, we establish the invariant that: *when the algorithm returns from a vertex u, the subtree rooted at u cannot be κ-heavy.* The invariant is easily seen to be true when the vertex u is unmarked, since then none of its subtrees are of size more than $\kappa - 2$. When u is marked, then consider the edge $p_u = (x, y)$ associated with u with corresponding weight label $m_u < \kappa - 1$. Since y is defined to be the lowest descendant of u that is marked, none of the subtrees of y are of size more than $\kappa - 2$, implying that none of the edges *below* y are κ-heavy since their deletion produces at least one subtree of size less than $\kappa - 1$. Further, the edge p_u itself and the edges below u that are not below y also cannot be κ-heavy. This is because the total number of vertices below u but not below y is $m_u < \kappa - 1$, and the deletion of these edges produces one subtree that does not contain vertices that are descendants of y.

Given the invariant, it is clear that when the algorithm returns from the root, the residual tree is not κ-heavy. However, we still need to show that the subtrees that were cut away from this residual tree are also not κ-heavy. Consider first the trees cut away upon a final return to a vertex u in Stage A, assuming that $s \geq 2$ and Stage A is invoked in the first place. Since each of these trees is rooted at a child v_i of u, and since the invariant also applies when the algorithm return from v_i to u, it is clear the cut subtrees are not κ-heavy. The only issue remaining is that of

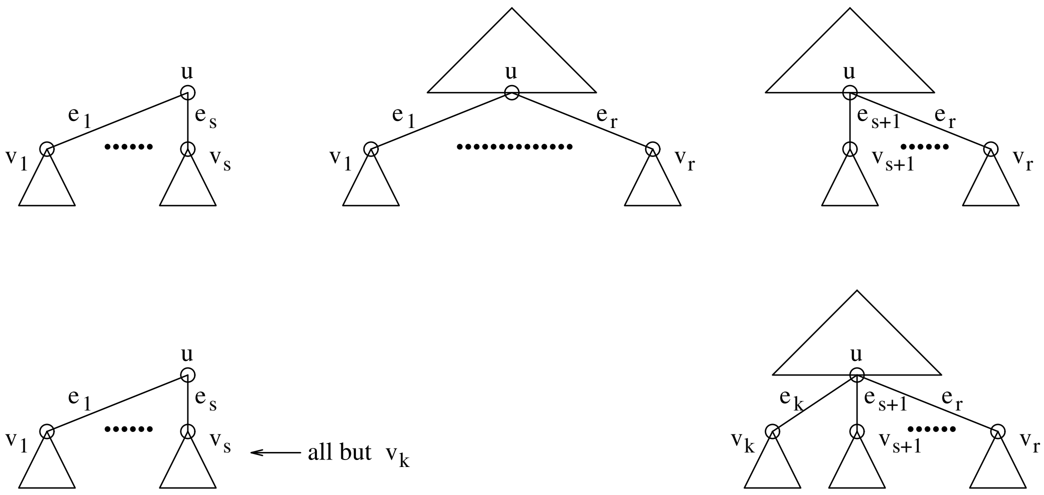


Figure 3: *An illustration for the subtrees referred to in the proof of Lemma 4. The top row shows the tree T (in middle), and the two subtrees T_1 (on left) and T_2 (on right). The bottom row shows the subtrees T_1' (on left) and T_2' (on right). Note that in T_1' the subtree rooted at v_k is* not *present.*

a subtree cut away in Stage B, assuming an edge is deleted at all in Stage B; the argument for this case is similar to that in the previous paragraph. The cut subtree is rooted at the marked vertex v_1 with pointer edge $p_1 = (x, y)$ and a label m_1. It is clear that none of the edges in the subtree rooted at y can be κ-heavy since no descendant of y has more than $\kappa - 2$ descendants of its own. The other edges cannot be κ-heavy since at least one of the subtrees resulting from their removal consists only of non-descendants of y; this subtree must have size at most $\kappa - 1$ since there are only $m_1 < \kappa - 1$ non-descendants of y in the subtree rooted at v_1. ■

Finally, the following lemma claims that the total number of edges deleted by this algorithm is no more than $\kappa - 1$.

Lemma 4 *Algorithm DFS-Decompose produces a decomposition of T into at most κ subtrees.*

Proof: Let OPT be some optimal set of edges to delete to ensure that T decomposes into subtrees that are not κ-heavy, and let $opt = |OPT|$. Since we assume that T has balance number κ, it must be the case that $opt < \kappa$. We establish inductively that the total number of edges deleted is at most opt, where the induction is over the number of steps in which the algorithm deletes one or more edge upon final return to a vertex.

We first establish the base case which corresponds to a tree in which no deletion is required, i.e., a tree T that is not κ-heavy. For this, it is easily verified that the algorithm does not delete any edges, since it never deletes until it has identified at least one κ-heavy edge in the entire tree.

To complete the induction, suppose that the *very first* edge deletion step involves deleting edges upon final return to a vertex u. There are only two possible stages of deletion to consider: Stage A and Stage B.

In Stage A, upon final return to u, the algorithm finds that $s \geq 2$ of u's children have at least $\kappa - 1$ descendants. Let T_1 denote the subtree of T *rooted at* u after deleting the edges $e_{s+1}, \ldots, e_r$ and the subtrees below them, and let T_2 denote the subtree of T obtained by deleting the edges $e_1, \ldots, e_s$ and the subtrees below them. (Refer to Figure 3.) Note that the trees T_1 and T_2 partition the edges of T, and u occurs in both of them. Let OPT_1 and OPT_2 be the partition induced on OPT by the partition of the edges of T into T_1 and T_2, respectively. Assume that the optimal solution deletes opt_1 edges in T_1 and opt_2 edges in T_2. We begin by establishing that $opt_1 \geq s - 1$. If this were not the case, then at least two of the edges in $\{e_1, \ldots, e_s\}$ are not deleted by OPT and do not have any deleted edges below them; without loss of generality, let these surviving edges be e_1 and e_2. But then, the subtree of T rooted at u after deleting the edges $e_3, \ldots, e_r$ (and the subtrees below them) is a κ-heavy subtree (with κ-heavy edges e_1 and e_2) that survives intact after the deletion of all the edges in OPT, a contradiction.

In fact, it is fairly easy to verify that if OPT deletes an edge in the subtree rooted at a vertex v_i in T_1, then modifying OPT to exclude this edge and include the edge (u, v_i) still gives a κ-balanced decomposition. Consequently, we can assume without loss of generality that OPT_1 deletes only the edges in $\{e_1, \ldots, e_s\}$, deleting either $s - 1$ or possibly even all s of them. The residual tree after the end of Stage A is defined to be the subtree of T denoted T_2' obtained by

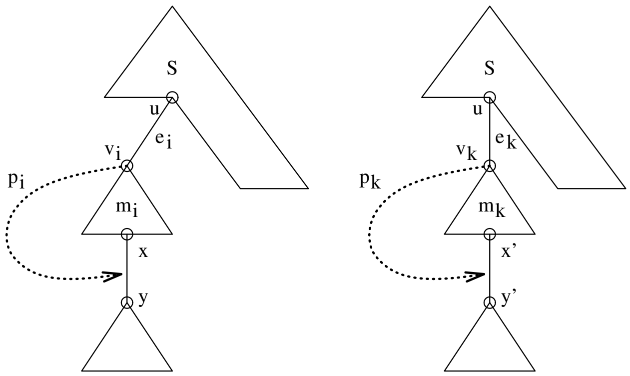


Figure 4: *The subtrees τ_i (on left) and τ_i' (on right), in the case when both v_i and v_k are marked during edge deletions in Stage A.*

deleting the edges in $\{e_1, \ldots, e_s\} \setminus \{e_k\}$, where e_k is the edge not deleted during Stage A; conversely, the cut trees are clubbed together into the other subtree T_1' obtained by taking the subtree rooted at u and deleting the edges e_k and $e_{s+1}, \ldots, e_r$. (Refer to Figure 3.) Since T_1' and T_2' partition T's edges, there is a corresponding partition of OPT into OPT_1' and OPT_2'.

Suppose first that OPT deletes all s edges from $\{e_1, \ldots, e_s\}$. This implies that $opt_1' = s - 1$, and our algorithm also deletes exactly that many edges in T_1'. The tree T_2' is the residual tree at the end of Stage A, and by the induction hypothesis our algorithm will delete exactly opt_2' edges in it. That is, the number of edges deleted by our algorithm is opt. Suppose instead that OPT deletes only $s - 1$ edges from $\{e_1, \ldots, e_s\}$, choosing to leave e_i undeleted out of this set, where e_i need not be the same as the edge e_k left undeleted by our algorithm in Stage A. In the decomposition of T induced by the optimal solution, let τ_i be the subtree containing e_i and the subtree rooted at v_i (which did not have any of its edges deleted by OPT). We claim that replacing e_k by e_i in OPT still gives a κ-balanced decomposition. The only possible source of κ-heaviness due to this modification of OPT is that, in the associated decomposition of T, the subtree τ_i containing e_i (and the subtree rooted at v_i) is modified to the subtree τ_i' obtained from τ_i by deleting the edge e_i and the subtree rooted at v_i while adding the edge e_k and the subtree rooted at v_k. Refer to Figure 4 for an illustration of the two subtrees τ_i and τ_i' in the case where both v_i and v_k are marked.

We will show below that provided τ_i is not κ-heavy, the tree τ_i' is also not κ-heavy. But before that, observe that we can now assume without loss of generality that OPT surely deletes the edges in $\{e_1, \ldots, e_s\} \setminus \{e_k\}$, implying that our algorithm deletes the optimal number, opt_1', of edges in T_1'. By the induction hypothesis, our algorithm also deletes the optimal number, opt_2', of edges in the residual tree T_2', and so it deletes the optimal number overall.

Claim 1 *If the tree τ_i is not κ-heavy then the tree τ_i' is also not κ-heavy.*

Proof: Consider first the case when v_k is unmarked, implying that while v_k has more than $\kappa - 2$ descendants, none of its children has more than $\kappa - 2$ descendants. From this it follows that all edges of τ_i' that lie in the subtree rooted at v_k have at most $\kappa - 2$ vertices *below* them and hence are not κ-heavy. The remaining edges in τ_i' include the edge e_k and the set of edges $S = \tau_i \cap \tau_i'$. Note that the edges in S are such that upon their removal the two subtrees obtained can be classified as: subtree S_1 that is totally contained in S and is the same in both τ_i and τ_i'; and, tree S_2 which in τ_i completely contains e_i and the subtree rooted at v_i, and in τ_i' completely contains e_k and the subtree rooted at v_k. Since in τ_i the tree S_2 has at least $\kappa - 1$ vertices, it must be the case that tree S_1 has no more than $\kappa - 2$ vertices, implying that the edges in S cannot be κ-heavy in τ_i'. By a similar argument, the edge e_k cannot be κ-heavy since that would imply that the edge e_i in τ_i is κ-heavy. Thus, τ_i' is not κ-heavy provided τ_i is not κ-heavy.

Suppose now that v_k is marked. We know that v_i must be marked since otherwise the algorithm would not have chosen a marked vertex to be v_k. Also, by the algorithm's choice of v_k, we know that $m_k \leq m_i$. Suppose that the pointer p_i at v_i is to the edge (x, y) and the pointer p_k at v_k is to the edge (x', y'). (Refer to Figure 4.) Observe that while y and y' both have more than $\kappa - 2$ descendants, none of their children has more than $\kappa - 2$ descendants. From this it follows that all edges of τ_i' that lie in the subtree rooted at y' have at most $\kappa - 2$ vertices *below* them and hence are not κ-heavy. Note that in τ_i, removing the edge p_i gives a subtree containing the more than $\kappa - 2$ descendants of y and another subtree containing the vertices in $S = \tau_i \cap \tau_i'$ and the remaining m_i vertices that are descendants of v_i

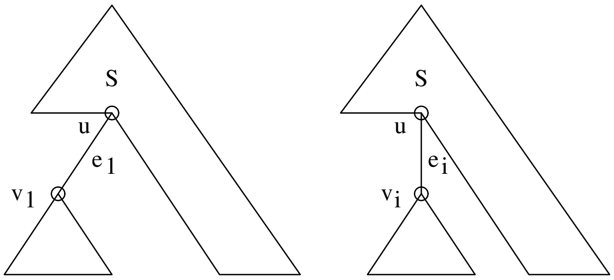


Figure 5: *The subtrees τ_i (on left) and τ_i' (on right), during the edge deletions in Stage B.*

but not of y. Therefore, since e_i is not κ-heavy in τ_i, it must be that case that $|S| + m_i \leq \kappa - 2$. Consider any edge in τ_i' that does not lie in the subtree rooted at y'. Upon the removal of such an edge from τ_i', one of the two resulting subtrees does not contain any vertex from among the descendants of y', and hence has at most $|S| + m_k$ vertices. Since $m_k \leq m_i$ and $|S| + m_i \leq \kappa - 2$, it follows that the edge cannot be κ-heavy. Thus, τ_i' is not κ-heavy provided τ_i is not κ-heavy. ■

We still have to consider the case where the very first deletion occurred in Stage B. Recall that when Stage B is invoked, the algorithm is at some vertex u with $s = 1$. Assume that the algorithm indeed deletes the edge e_1, since otherwise there is nothing to prove. Let T_1 denote the subtree of T rooted at u, and let T_2 denote the subtree of T obtained by deleting the edges $e_1, \ldots, e_r$ and the subtrees below them. Let OPT_1 and OPT_2 be the partition induced on OPT by the partition of $T's$ edges into T_1 and T_2, respectively. Note that the edge p (in the description of stage B) is κ-heavy in T_1 since that is precisely when deletion takes place in Stage B. We claim that OPT_1 contains at least one edge. This is easy to see since otherwise the tree T_1 is a κ-heavy subtree that survives after the deletion of all the edges in OPT, a contradiction. Thus, the optimal solution deletes at least one edge in T_1 and opt_2 edges in T_2.

If OPT deletes any edge in the subtree rooted at any v_i, then that edge can be replaced in OPT by the edge e_i since the subtree rooted at v_i is not κ-heavy. That is, without loss of generality we may assume that OPT_1 deletes only the edges in $\{e_1, \ldots, e_r\}$. Suppose first that the optimal solution does not delete the edge e_1. Then, it must delete some edge e_i, for $i \geq 2$. In the decomposition of T induced by the optimal solution, consider the subtree τ_i that contains edge e_1 and the subtree rooted at v_1. Suppose we were to replace e_i by e_1 in OPT; since the subtrees rooted at v_1 and v_i are not κ-heavy, the only possible source of κ-heaviness due to this modification is that, in the associated decomposition of T, the tree τ_i gets modified into a tree τ_i' by the deletion of e_1 and the descendants of v_1 which are replaced by e_i and the descendants of v_i. (Refer to Figure 5.) If we can establish that τ_i' is not κ-heavy, then it would follow that we can assume without loss of generality that OPT deletes e_1. Note that an edge in $S = \tau_i \cap \tau_i'$ cannot become κ-heavy since the removal of at least $\kappa - 1$ vertices and the addition of at most $\kappa - 2$ vertices in going from τ_i to τ_i' can only reduce the heaviness of that edge. At the same time, any remaining edge in $\tau_i' \setminus S$ lies below the edge e_i and has less than $\kappa - 1$ vertices below it, so it cannot be κ-heavy. We are left with the case where the optimal solution does contain the edge e_1. Now, since the algorithm does not delete any edges below v_1 and by the induction hypothesis deletes the optimal number of edges in the residual tree after the deletion of e_1, we obtain that our algorithm also computes an optimal solution overall. ■

4.3 Extension to Trees with TOTAL Cost

We can show that essentially the same results as for paths carry over to the tree case with the TOTAL cost measure, provided we are willing to modify the notion of breaking an edge into that of breaking a vertex; clearly, this is not as satisfactory a resolution as in the case of the MIN measure with trees.

Note that now the star graph is a really bad case if we do not BREAK any edges, since each update has a linear cost. Also, breaking a sublinear number of edges is of no use since there will remain a subtree of linear size whose edges will require linear time to update, and of course breaking a linear number of edges will drive up the cost of a QUERY to a linear quantity.

The trick here is to modify our model slightly. Instead of breaking edges to decompose trees into small subtrees, we will use a different primitive operation that we will refer to as *breaking a vertex*. The idea is to replace a vertex v by two copies of itself, say v_1 and v_2, with an edge between them and such that each edge incident on v is assigned to

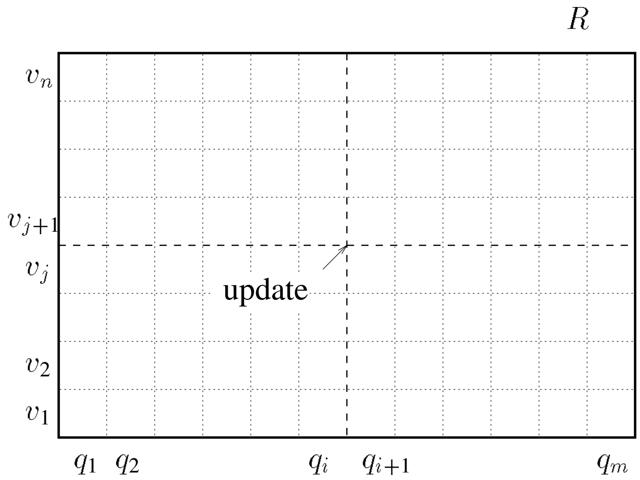


Figure 6: *A geometric interpretation of the TOTAL problem*

exactly one of the two copies v_1 and v_2. Then, breaking vertex v corresponds to breaking the (dummy) edge between v_1 and v_2. A possible physical interpretation of this is in terms of decomposing a link into a pair of sublinks. It can be argued that, at least in some applications, the resulting cost measures are an accurate reflection of the actual implementation discussed earlier.

It is now easy to see that there exists a choice of $O(\sqrt{n})$ vertices to break (and a choice of the assignment of incident edges to the two copies of a broken vertex) such that the tree decomposes into $O(\sqrt{n})$ subtrees, each of size $O(\sqrt{n})$. It is now possible to maintain the data structure at a cost of $O(\sqrt{n})$ per operation.

5 Offline Setting with TOTAL Cost

In this section, we focus on the offline version of the kinematic data structures when the linkage L has a path topology and using the TOTAL cost measure, henceforth referred to as the TOTAL problem. (We will consider the MIN version of this problem in the next section.) Interpreting the problem in geometric terms, we obtain that it is NP-complete. We also indicate briefly the known results for approximation to the problem.

5.1 The TOTAL Problem is NP-complete

Let $q_1, q_2, \ldots, q_m$ denote the ordered sequence of queries and recall that $v_1, v_2, \ldots, v_n$ denote the vertices in the path. We consider an $n \times m$ grid, lying inside a rectangle R, where each column stands for a query and each row stands for a vertex in the path. With a slight abuse of notation we will refer to the columns as $q_1, q_2, \ldots$ and to the rows as $v_1, v_2, \ldots$

The queries are interleaved with UPDATE operations. Suppose that between queries q_i and q_{i+1} we have an update operation UPDATE(v_j, v_{j+1}). In our geometric model this update operation translates into a point that lies at the intersection of the vertical grid line between columns q_i and q_{i+1} and the horizontal grid line between rows v_j and v_{j+1}. (Refer to Figure 6.) If there is more than one update operation between the queries q_i and q_{i+1}, they all translate into points on the same vertical grid line lying on the appropriate horizontal lines. Let $U = \{u_1, u_2, \ldots, u_N\}$ be the set of all the points corresponding to update operations.

Consider an axis parallel rectangle whose height spans rows $v_i, v_{i+1}, \ldots, v_j$ and whose width spans columns $q_k, q_{k+1}, \ldots, q_l$. This rectangle represents the following situation: immediately before query q_k any BROKEN interior edge along the path $v_i, v_{i+1}, \ldots, v_j$ was MERGED, and each of the exterior edges of the path, namely the edges (v_{i-1}, v_i) and (v_j, v_{j+1}), was BROKEN (if it was previously MERGED), and all the interior edges of this path remain MERGED until the completion of query q_l.

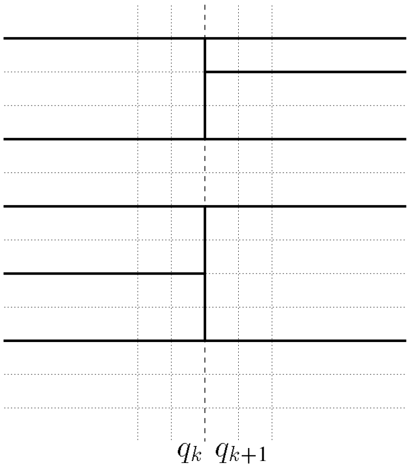


Figure 7: *The cost of restructuring between queries q_k and q_{k+1}*

We claim that an optimal (i.e., minimum cost) solution for the problem corresponds to partitioning the entire grid rectangle R into rectangles $R_1, R_2, \ldots, R_t$ such that $\sum_{i=1}^{t}(h_i + w_i)$ is minimized, where h_i (w_i) is the length in unit grid size of the vertical (respectively, horizontal) edge of R_i, and such that no rectangle R_i contains a point of U in its interior.

To see this, consider first the intersection of a single column q_i with the rectangles R_i. The cost of the query q_i is the number of rectangles in the intersection, since we assumed that the cost of a query is equal to the number of substructures in the path. So we charge the cost of the query per rectangle R_i, to the portion of the lower horizontal edge of R_i that intersects the column q_i. Next consider the vertical grid line between the query columns q_k and q_{k+1}. (Refer to Figure 7.) The rectangle edges that appear on this vertical line correspond to the subpaths that have undergone change. The cost of these changes is exactly the length of these vertical rectangle edges. Recall that, if there are several update operations between queries q_k and q_{k+1} our planner executes them all at the same time, and the cost of restructuring a subpath of length s, by a collection of MERGE and BREAK operations is s.

The only constraint that our original problem imposes on the partitioning of R is that no rectangle in the partitioning contains a point of U in its interior. If a rectangle contains a point $u_j \in U$ in its interior, this implies that our data structure has not been updated by the update operation corresponding to the point u_j.

Lingas et al [28] have established the NP-completeness of partitioning an axis-parallel rectangle R with N point holes into axis-parallel rectangles with minimum total edge length, such that no rectangle in the partitioning contains a point hole in its interior. Hence, we obtain the following theorem

Theorem 6 *The TOTAL problem is NP-complete.*

5.2 Approximation Algorithms for TOTAL

A number of approximation algorithms have been proposed for the rectangular partition problem. Most of these algorithms rely on the connection between the rectangular partition as above and the so-called "guillotine" partition [11]. Finding the optimal guillotine partition is solvable in polynomial time, and it has been shown [11] that the optimal guillotine partition has edge length no greater than 1.75 times the length of the optimal rectangular partition.

Gonzalez and Zheng [11] give a 1.75 approximation bound for partitioning a rectangle with N point holes into axis-parallel rectangles, using dynamic programming. The running time of their algorithm is $O(N^5)$. Gonzalez, Razzazi, and Zheng [17] give a simple $O(N \log N)$ algorithm that obtains a 4-approximation for the same problem.

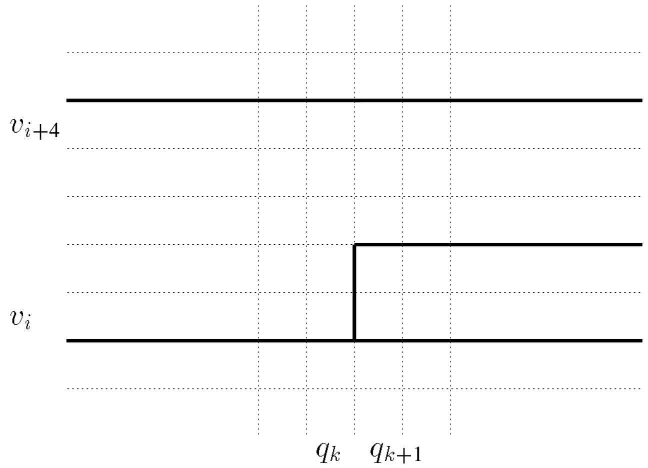


Figure 8: *A geometric interpretation of the MIN measure*

6 Offline Setting with MIN Cost

We now turn to the offline setting for path topologies under the MIN cost measure, which we call the MIN problem. As for the TOTAL measure, we interpret the MIN problem geometrically. We have a grid bounded inside a rectangle R. Each row corresponds to a vertex in the path; the order of the rows corresponds to the order of the elements in the kinematic chain. The columns correspond to queries and points on grid vertices correspond to updates.

Consider one column (i.e., one horizontal unit slab of the rectangle), say column q_i and suppose there are k horizontal line segments crossing this column besides R's edges. We interpret this as follows: at time q_i, the data structure consists of $k+1$ static structures describing rigid subpaths of the chain—therefore the cost of the query here is proportional to $k+1$. The collection of all horizontal segments inside the rectangle and the length of one horizontal edge of the rectangle can be charged for all queries.

In between queries the structure may be reconfigured. The rebuilding of structures after a query depends on how they were organized just before the query. There are two ways to rearrange the structures:

- We can take any set of contiguous structures and rebuild them from scratch; the cost is obviously proportional to the number of elements in all these sets together and this will be expressed as a vertical line segment through all the rows corresponding to the elements.
- We can be more careful: in the rebuilding stage, recompute the new structures by removing elements from one structure and adding them to a neighboring structure. The cost is now proportional to the number of elements that are being moved around, and geometrically this is expressed as a vertical line segment through the rows corresponding to the moved elements. For an illustration, consider Figure 8: after query q_k the structure describing the vertices $v_i, \ldots, v_{i+4}$ is split into two. The cost of this restructuring is proportional to the shorter subpath.

Definition 4 *A* rectilinear polygon *is a polygon with each side parallel to a coordinate axis.*

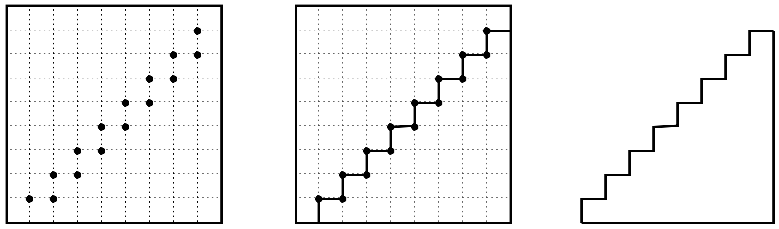
Definition 5 *A rectilinear polygon is* vertically convex *if every vertical line intersects it in at most one connected component.*

We will abbreviate *vertically convex rectilinear polygon* to VCRP.

Lemma 5 *In the geometric interpretation of the MIN problem, each structure is described by a VCRP.*

Proof: At each stage (column) a structure consists of a contiguous set of grid squares. The two restructuring rules above imply that there are no dangling edges inside the rectangle R. ■

Now our partitioning has two constraints: it consists of VCRPs, and no VCRP contains a point hole in its interior.



(a) (b) (c)

Figure 9: *An instance where the MIN measure is $\Omega(\log k)$ better than the TOTAL measure*

Remark 1 *We have overloaded the vertical axis with two slightly different meanings: the cost per element of building from scratch and the cost of deletion or insertion. This leads to a constant factor error (2 in the model for molecules as described above). It may still be helpful to use this model for obtaining approximation algorithms.*

We now discuss the relation between the TOTAL and MIN measures. We show that the gain in using the MIN measure instead of the TOTAL measure is at most a factor of $O(\log N)$ in the total time to process a sequence of updates and queries, where N is the number of updates. We also show that there are instances of the problem where this gain is obtained. Our analysis will imply an $O(\log N)$ factor approximation algorithm for the MIN problem. We remark that we do not know whether the MIN problem is NP-complete.

Let R be the rectangular grid as above, containing N point holes $U = \{u_1, u_2, \ldots, u_N\}$ on interior vertices. For a given instance (R, U) of the problem, let $\mathcal{R}_{\text{opt}}(R, U)$ denote the length of the minimum-length *rectangular* partitioning, and let $\mathcal{V}_{\text{opt}}(R, U)$ denote the length of the minimum-length partitioning of R into VCRPs; in both cases we consider the edges of R to be a part of the partitioning. Also, in both cases the points of P need to lie on the boundaries of the partitioning objects.

Theorem 7 *Given a rectangular grid R and a set U of N points on grid vertices inside R,*

$$\mathcal{R}_{\text{opt}}(R, U) = O(\mathcal{V}_{\text{opt}}(R, U) \log N).$$

Moreover, there is a family of problems (depending on N) such that $\mathcal{R}_{\text{opt}}(R, U) = \Omega(\mathcal{V}_{\text{opt}}(R, U) \log N)$.

Proof: Let c be a grid vertex on the boundary of R that is closest to a point in U. We construct a minimum-length rectilinear Steiner tree S on the points in $U \cup \{c\}$. The total edge length of S is clearly not greater than $\mathcal{V}_{\text{opt}}(R, U)$. This tree together with the boundary of R defines a degenerate rectilinear polygon Q without holes; it is degenerate in the sense that some edges bound the polygon on both sides. The total edge length of Q is no greater than twice $\mathcal{V}_{\text{opt}}(R, U)$. We now turn Q into a simple polygon Q' by substituting each edge that bounds the polygon on two sides by two parallel edges very close to each other. The polygon Q' has at most $O(N)$ concave vertices; this is guaranteed for a minimum-length rectilinear Steiner tree, and that is why we did not use the optimal partition into VCRPs directly.

By a result of Levcopoulos and Lingas [27], for a simple rectilinear polygon with perimeter π and k concave vertices, there is a rectangular partitioning of length $O(\pi \log k)$. Hence Q' can be partitioned into rectangles with total length at most $O(\log N)$ times the length of Q. Thus, we have obtained a rectangular decomposition of (R, U) which is of length at most $O(\mathcal{V}_{\text{opt}}(R, U) \log N)$. This implies the asserted upper bound.

To show that this bound is tight, we adapt a lower bound construction in Levcopoulos and Lingas [27]; see Figure 9 for an illustration. In a square grid of size $k \times k$, with the same unit resolution along both coordinates, we arrange the $2(k-2)+1$ points of U along a "staircase." A minimum length VCRP partition is obvious — it consists of the edges of the staircase (see Figure 9(b)), and has length $\Theta(k)$. Next, we consider a rectangular partition for the same setting, and we restrict our attention to the polygon W that is below the staircase (see Figure 9(c)). By the results of Levcopoulos and Lingas [27], any rectangular decomposition of W will be of length $\Omega(k \log k)$. ■

Similarly, we obtain the following algorithmic result.

Theorem 8 *For a given maintenance problem (R, U), an $O(\log N)$-approximation for the MIN problem can be obtained in $O(N \log N)$ time, where N is the number of points in U.*

Proof: We apply the algorithm of Gonzalez et al [17] mentioned above. It is an $O(N \log N)$ algorithm that obtains a 4-approximation for the rectangular partition problem. By Theorem 7, the length of the resulting partition will be at most a factor $O(\log N)$ larger than the optimal partition into VCRP's. Thus it gives an $O(\log N)$-approximation for the MIN problem. ■

7 Conclusion and Further Work

In this paper, we have initiated the study of a novel type of data structure for kinematic structures that efficiently supports intersection queries. We formulated an abstract model that captures a variety of settings ranging from collision detection for articulated robot arms to conformation search in molecular biology. Our results shed light on the complexity of efficient kinematic data structures for path and tree topologies, but a whole multitude of questions remain open at this point. We outline some of the issues worthy of further exploration.

- An important issue is that of an empirical testing of the ideas outlined in this paper. We are currently working on implementation of these kinematic data structures for conformation search in molecular biology and collision detection for articulated robot arms.
- In some application, particularly in randomized path planning and conformation search for molecular biology, there is considerable flexibility in the order in which the operations are performed on the kinematic data structure. This raises the issue of extending the offline results to the case where the algorithm can at least partially reorder the sequence of operations so as to minimize the total computational cost.
- In the offline setting, our results are concerned primarily with the path topology, where we show that the problem is NP-complete and there are some reasonably good approximation algorithms. Improving the approximation bounds obtained here seems rather difficult, while extending the results to tree topologies seems much more feasible.
- Another approach would be to consider the online setting employing competitive analysis, i.e., comparing the online algorithm's cost to the optimal offline cost. It is fairly easy to see that this problem is a special case of the *metrical task systems* formulation of Borodin, Linial, and Saks [7], but this leads to a fairly weak bound on the competitive ratio. We believe that the special structure of this problem should lead to significantly better competitive ratios. For example, the (artificial) situation where we fix the number of substructures at some value k corresponds exactly to the well-known k-server problem [29] for which a $(2k-1)$-competitive algorithm is known [26]. In general, we can formulate the online version of our problem as a variant of the k-server problem that we call the *dynamic servers* problem. This is a server problem where the number of servers varies over time, and the online algorithm is required to pay a "rental" cost depending upon the number of servers in use at any given time. This bears some similarity to the data migration and replication problem studied in the online literature [6]. Another aspect of the online problem is that servers are allowed to perform excursions [4, 29]. We expect to give our model and results for the online setting in a subsequent paper.
- In Section 4.3 we outlined a strategy for maintaining the kinematic data structure for tree topologies under the TOTAL cost measure. This was based on applying the BREAK operation to vertices rather than edges. It would be interesting to further explore this notion and its validity in various applications.
- It is possible that in some applications even the topology of the linkage could undergo some change [14, 37]. It would be interesting to extend our results to such dynamic situations too.

References

[1] J. Barnes and P. Hut. A Hierarchical $O(N \log N)$ Force-Calculation Algorithm. *Nature*, 324 (1986), pp. 446–449.

[2] J. Barraquand, L. Kavraki, J.C. Latombe, T.Y. Li, R. Motwani, and P. Raghavan. A Random Sampling Scheme for Path Planning. To appear in *Robotics Research*, G. Giralt and G. Hirzinger (eds.), Springer Verlag (1996).

[3] J. Barraquand and J.C. Latombe. Robot Motion Planning: A Distributed Representation Approach, *The International Journal of Robotics Research*, 10 (1991), pp. 628–649.

[4] M. Bern, D.H. Greene, A. Raghunathan, and M. Sudan. Online Algorithms for Locating Checkpoints. *Algorithmica*, 11 (1994), pp. 33–52.

[5] P. Bessière, E. Mazer, snd J.M. Ahuactzin. Planning in a Continuous Space with Forbidden Regions: The Ariadne's Clew Algorithm. In *Algorithmic Foundations of Robotics*, K. Goldberg et al (eds.), A.K. Peters, Wellesley, MA (1995), pp. 39–47.

[6] D.L. Black and D.D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1983.

[7] A. Borodin, N. Linial, and M.E. Saks. An Optimal On-Line Algorithm for Metrical Task Systems. *Journal of the ACM*, 39 (1992), pp. 745–763.

[8] Y.-J. Chiang and R. Tamassia. Dynamic Algorithms in Computational Geometry. In *Proceedings of the IEEE*, 80 (1992), pp. 1412–1434.

[9] G.S. Chirikjian and J.W. Burdick. Kinematics of Hyper-Redundant Manipulators. In *Proceedings of the 2nd International Workshop on Advances in Robot Kinematics*, 1990, pp. 392–399.

[10] M.L. Connolly. Solvent-accessible Surfaces of Proteins and Nucleic Acids. *Science*, 221 (1983), pp. 709–713.

[11] D.-Z. Du, L.-Q. Pan, and M.-T. Shing. Minimum Edge Length Guillotine Rectangular Partition. Technical Report MSRI 02418-86, MSRI, Berkeley, 1986.

[12] B. Faverjon. Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1984, pp. 504–512.

[13] B. Faverjon and P. Tournassoud. A Practical Approach to Motion Planning for Manipulators with Many Degrees of Freedom. In *Robotics Research 5*, H. Miura and S. Arimoto (eds.), MIT Press, Cambridge, MA (1990), pp. 65–73.

[14] T. Fukuda and S. Nakagawa. Dynamically Reconfigurable Robotic Systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1988, pp. 1581–1586.

[15] N. Go and H.A. Scheraga. Ring Closure and Local Conformation Deformations of Chain Molecules. *Macromolecules*, 2 (1980), pp. 178–187.

[16] T. Gonzalez and S.-I. Zheng. Improved Bounds for Rectangular and Guillotine Partitions. *Journal of Symbolic Computation*, 7 (1989), pp. 591–610.

[17] T. Gonzalez, M. Razzazi, and S.-I. Zheng. An Efficient Divide-and-Conquer Approximation for Hyperrectangular Partitions. In *Proceedings of the 2nd Canadian Conference on Computational Geometry*, 1990, pp. 214–217.

[18] D. Halperin, L. Kavraki, J.C. Latombe, R. Motwani, C. Shelton, and S. Venkatasubramanian. Geometric Manipulation of Flexible Molecules. To appear in *Proceedings of the ACM Workshop on Applied Computational Geometry*, 1996.

[19] D. Halperin and M.H. Overmars. Spheres, Molecules, and Hidden Surface Removal. In *Proceedings of the 10th ACM Symposium on Computational Geometry*, 1994, pp. 113–122.

[20] T. Horsch, F. Schwarz, and H. Tolle. Motion Planning for Many Degrees of Freedom — Random Reflections at C-Space Obstacles. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1994, pp. 3318–3323.

[21] S. Kambhampati and L.S. Davis. Multiresolution Path Planning for Mobile Robots. *IEEE Transactions on Robotics and Automation*, 2 (1986), pp. 135–145.

[22] L. Kavraki and J.C. Latombe. Randomized Preprocessing of Configuration Space for Fast Path Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1994, pp. 2138–2145.

[23] L. Kavraki, P. Švestka, J.C. Latombe, and M. Overmars. Probabilistic Roadmaps for Fast Path Planning in High Dimensional Configuration Spaces. To appear in *IEEE Transactions on Robotics and Automation*.

[24] O. Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *International Journal of Robotics Research*, 5 (1986), pp. 90–98.

[25] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe. Planning Motions with Intentions. In *Proceedings of SIGGRAPH*, 1994, pp. 395–408.

[26] E. Koutsoupias and C.H. Papadimitriou. On the k-server conjecture. In *Proceedings of 26th Annual ACM Symposium on Theory of Computing*, 1994, pp. 507–511.

[27] C. Levcopoulos and A. Lingas. Bounds on the Length of Convex Partitions of Polygons. In *Proceedings of the 4th Conference on the Foundations Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 181, Springer-Verlag, pp. 279–295.

[28] A. Lingas, R.Y. Pinter, R.L. Rivest, and A. Shamir. Minimum Edge Length Partitioning of Rectilinear Polygons. In *Proceedings of the 20th Annual Allerton Conference on Communication, Control, and Computing*, 1985, pp. 53–63.

[29] M. Manasse, L. McGeoch, and D.D. Sleator. Competitive Algorithms for Server Problems. *Journal of Algorithms*, 11 (1990), pp. 208–230.

[30] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*. Volume 3 of *Data Structures and Algorithms*, Springer-Verlag, New York (1985).

[31] P.G. Mezey. Molecular Surfaces. In *Reviews in Computational Chemistry*, Volume I, K.B. Lipkowitz and D.B. Boyd (eds.), VCH Publishers (1990).

[32] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, New York (1993).

[33] M.H. Overmars. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science 156, Springer-Verlag, Berlin (1983).

[34] M. Overmars and P. Švestka. A Probabilistic Learning Approach to Motion Planning. In *Algorithmic Foundations of Robotics*, K. Goldberg et al (eds.), A.K. Peters, Wellesley, MA (1995), pp. 19–37

[35] M. Ponamgi, D. Manocha, and M.C. Lin. Incremental Algorithms for Collision Detection Between Solid Models. In *Proceedings of the 3rd ACM Symposium on Solid Modeling and Applications*, 1995, pp. 293–304.

[36] S. Quinlan. Efficient Distance Computation Between Non-Convex Objects. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1994, pp. 3324–3330

[37] M. Yim. *Locomotion with a Unit-Modular Reconfigurable Robot*. PhD thesis, Stanford Technical Report STAN-CS-94-1536, Department of Computer Science, Stanford University, December 1994.