Dynamic Maintenance of Kinematic Structures

D. Halperin* J-C. Latombe† R. Motwani‡

Department of Computer Science
Stanford University
Stanford, CA 94305-2140.
E-mail: {halperin,latombe,motwani}@cs.stanford.edu

Abstract

We study the following dynamic data structure problem. Given a collection of rigid bodies moving in three-dimensional space and hinged together in a kinematic structure, our goal is to maintain a data structure that describes certain geometric features of these bodies, and efficiently update it as the bodies move. This data structure problem seems to be fundamental and it comes up in a variety of applications such as conformational search in molecular biology, simulation of hyper-redundant robots, collision detection and computer animation. In this note we present preliminary results on a few variants of the problem.

1 Introduction

We define an abstract data structuring problem and leave the motivation for the next section. We are required to maintain a data structure for a path graph consisting of n nodes $V = \{v_1, v_2, \ldots, v_n\}$ arranged in a path P so that for $1 \leq i \leq n-1$ there is an edge (v_i, v_{i+1}), and there are no other edges in the graph. At any time, each edge has a state that is either UNLOCKED or LOCKED. There are two operations that are required to be supported.

UPDATE(v_i, v_{i+1}) This operation specifies an edge and asks that it be UNLOCKED temporarily, unless the edge is already UNLOCKED.

QUERY(v_i, v_j) This operation specifies a subpath from v_i to v_j, and requires a report that involves examining all the UNLOCKED edges in the subpath.

*Partially supported by a grant from Pfizer Central Research.

†Partially supported by a grant from Pfizer Central Research.

‡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.

The algorithm's goal is to decide which edges to keep UNLOCKED and which to keep LOCKED between any two operations in an input sequence of operations with an arbitrary mix of UPDATE and QUERY. The algorithm is free to UNLOCK or LOCK any edge at any time in the processing of the sequence, and it has to pay for the change in state. It resorts to the following two primitive operations for this purpose: UNLOCK(v_i, v_{i+1}) and LOCK(v_i, v_{i+1}). Note that at any time, the set of UNLOCKED edges decompose the path P into an ordered sequence of subpaths P_1, P_2, ... that are linked together by UNLOCKED edges and do not contain any internal UNLOCKED edges. We call each such subpath a *rigid* subpath.

We define the cost function for these operations as follows:

UNLOCK(v_i, v_{i+1}) The cost is 0 when the edge (v_i, v_{i+1}) is already UNLOCKED. Otherwise, let P_j be the subpath containing the edge (v_i, v_{i+1}), and then the cost of the UNLOCK operation is the length of the path P_j.

LOCK(v_i, v_{i+1}) The cost is 0 when the edge (v_i, v_{i+1}) is already LOCKED. Otherwise, let P_j be the subpath created by LOCKING the edge (v_i, v_{i+1}), and then the cost of the LOCK operation is the length of the path P_j.

UPDATE(v_i, v_{i+1}) The cost is the same as that for UNLOCK(v_i, v_{i+1}), except that the cost is 1 (instead of 0) when the edge is already UNLOCKED.

QUERY(v_i, v_j) The cost of this operation is the number of rigid subpaths in the interval (v_i, v_j), namely the number of UNLOCKED edges plus 1.

We will refer to the cost of UNLOCK as described above as the TOTAL cost measure, because all the the nodes in the rigid path where we unlock are taken into consideration. An alternative cost measure, the MIN cost, will be the minimum of the length of the two subpaths resulting from the UNLOCK operation. The justification of both measures are given in the next section. A similar remark applies to the LOCK operation. In the TOTAL measure its cost will be the number of nodes in both subpaths and in the MIN measure, the cost will be that of the shorter of the subpaths.

We also observe that if we need to carry out a series of LOCK and UNLOCK operations on a contiguous subpath (not necessarily rigid) of length k, than the overall cost of these operations in the TOTAL measure is k. Again, this remark will be justified in the next section.

Unless otherwise stated, we will assume that a query visits the entire path, hence from this point on QUERY will stand for QUERY(v_1, v_n), and we call it a PATH query.

2 Background and Motivation

Our study is motivated by applications where a large number of rigid bodies moving in three-dimensional space are hinged together in a kinematic structure. Our goal is to maintain a data structure that describes certain geometric features of these bodies, and update it as the bodies move. We will concentrate below on *intersection queries*. For an 'intersection query' with an additional object q, our data structure should report whether at the time of the query the object q intersects any of the bodies in the original set.

2.1 The Motivating Applications

The need to maintain data structures to answer geometric queries for large kinematic systems arises, among others, in the following application domains:

- molecular biology and rational drug design
- simulation of hyper-redundant robot manipulators
- computer animation.

Molecules can attain different conformations according to kinematic constraints [6]. When dealing with macromolecules the number of degrees of freedom can be in the hundreds or even thousands. In robotics, several researchers suggested to construct highly flexible robots that have a large number of degrees of freedom [3], [18]. Models of such robots have already been constructed in laboratories. In yet another area, in computer animation, living creatures with flexible motion capabilities need to be modeled and displayed [11]. Realistic kinematic models of humans and animals require a large number of degrees of freedom.

In all these areas, computation with the models, simulation and visualization require to maintain certain geometric properties of the bodies involved, as they move.

2.2 Related Work

The problem that we study here is related to dynamic data structure problems in computational geometry; see, for example, [15],[17]. However, most of the study of dynamic data structures deals with situations where the dynamization is in that objects are added to the set or removed from it. In our study the set of objects remains fixed, and the dynamization is in their motion while obeying kinematic constraints.

As we will see below, standard issues in dynamic data structures, namely insertions and deletions, will come to play in our analysis as well. However, in the model that we adapt, insertions and deletion are easy to perform and take $O(1)$ time each. This is justified in certain applications; see the next subsection.

Another related area is that of data structures for N-body force calculations [1]. The similarity to our problem is that in such calculations one aims to maintain a data structure for a set of N bodies that move, where the set of bodies does not change. However, because of the nature of change in the relative placements of the bodies, prevailing solutions to the problem (see, e.g., [1]) compute the data structure from scratch at each step.

Collision checking in robot motion planning and robot simulation, has been a prevailing area of research both theoretical and applied. However, since most of the prevailing robots to date have up to six degrees of freedom, little attention has been given to the effect of the number of degrees of freedom on the maintenance of geometric information on the robots as they move. Our current study focuses on systems that have a large number of degrees of freedom.

2.3 The Basic Model

We describe the static data structure for intersection queries in molecules. Our description is taken from [9] and is presented here to lay the ground for the discussion below. The structure represents the geometry of a molecule by representing each atom as a sphere in a fixed placement in 3-space and with radius that depends on the atom type. For more details on this so-called hard sphere model of a molecule see, for example, [4],[16].

Remark 1 *Our choice to concentrate on the kinematic model of molecules is merely for convenience: in molecules the description of the rigid links is especially simple. The same discussion would be applicable with minor changes to robots with many degrees of freedom, provided that their links are similar in shape and size [18]. The example of molecules also has the advantage that the static data structures have been implemented and experimented with to confirm the theoretical bounds that we cite below.*

As noted in [9], the spheres in the model of a molecule fulfill certain properties that make their manipulation easier than the manipulation of an arbitrary set of spheres in three-dimensional space. Informally the two properties of interest are (i) the radii of the spheres lie in a relatively narrow range, and (ii) the centers of any two spheres in the model cannot get too close to one another. These properties are formalized in [9] and used to construct efficient data structures and efficient visibility algorithms for this model.

It follows from these properties that one can construct a data structure for a molecule of n atoms for efficiently answering intersection queries with spheres of roughly the same size. The structure uses $O(n)$ storage, answers a query in $O(1)$ time, and requires $O(n)$ randomized preprocessing time. The structure uses a three-dimensional grid and stores in each grid cube those atom spheres that intersect it. The $O(n)$ non-empty grid cubes are then stored in a hash table. This structure is good for answering intersection queries when the molecule is at a fixed nuclear configuration.

The atoms in a molecule can move, and the molecule is said to attain different *conformations*. In a simplified model the relative motion capabilities of atoms in a molecule are described as if the atom spheres were rigid links in a robot and certain bonds between pairs of atoms allow a rotation around the line connecting the two atom centers as if they are rotational joints of the robot. Our extended model of the molecule, to include its capability to change conformations, is a collection of spheres, some pairs of which are rigidly attached to one another, while other pairs have a degree of freedom of rotation between them.

We will describe these kinematic constraints by a graph where each node corresponds to an atom sphere, and an edge between nodes describes a constraint. An edge can be either rigid—there is a fixed relative displacement between the atoms that it connects, or rotatable—when there is a degree of freedom of rotation along a fixed line between the two atoms. Molecules can often be described by graphs that are trees, although is some situations cycles do arise. However, to simplify our discussion, we will assume for the moment that the graph is a path. Some of the results below extend to trees and we will point these cases out when we discuss them.

Next, we are given a sequence of update operations that are aimed at changing the conformation of the molecule: this is done by giving a sequence of joint angles to which we need to update the rotatable bonds, and this sequence is interleaved with intersection queries: namely, does a given query sphere intersect any of the spheres in the molecule at its current conformation.

To get a feeling for the problem that we are addressing in this paper consider the following scenario. We are handling a molecule modeled as a path and having a large number of atoms, and we get a long sequence of updates that all modify the joint angle of a single rotatable bond somewhere near the middle of the path. To maintain our data structure valid for intersection queries we can simply rebuild the structure after each update of the joint angle. This will cost $O(n)$ per update. However, in this limited example there is a much better strategy to follow: break the path into two subpaths at the edge that represents the rotatable bond that is being updated, and build a static data structure for each subpath. When an update comes, all we have to update is the relative displacement between the two subpaths. For a query sphere, we need to query each of the substructures separately. The cost of each update is only $O(1)$ and the cost of a query (which originally is $O(1)$) has only doubled. For a more involved sequence of updates it may not be as obvious what is a good strategy in terms of breaking the structure into substructures or merging substructures into a larger structure.

The question that we address in this paper is: Given a set of objects and the kinematic constraints between them, what is the best strategy to maintain a data structure, such that a sequence of updates and queries will be answered in optimal time, where the algorithm has the freedom to break or merge substructures.

To further justify the models that we will propose below, we now explain more formally how each operation on our dynamic data structure, that consists of a series of static data structures, is performed. Our molecule consists of atom spheres that correspond to the nodes $v_1, v_2, \ldots, v_n$ in a path. Some of the edges of the path correspond to rotatable bonds. At any time our data structure consists of one or more static data structures each representing a contiguous set of nodes in the path. Let S_1 denote the static structure (a hash table in the example above) containing the sphere corresponding to v_1, S_2 is the next static structure along the path and so on. We assume that the sphere corresponding to v_1 is fixed in three-space. Each static structure has a coordinate frame attached to it in which the spheres of the structure are described. The coordinate frame attached to S_1 is the universal frame, in which the query spheres 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 to be described in the frame of S_{i+1}.

Given a query sphere 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 $O(k)$.

To update the joint value of an edge that lies between two static structures (that is, it is not internal to any static structure), we simple have to update the transformation between the two structures on both its sides. This takes $O(1)$ time.

To break a static structure consisting of t spheres into two substructures of sizes $t_1 \leq t_2$ we have two options: either to rebuild the two substructures from scratch at the cost of $O(t_1 + t_2) = O(t)$, or to delete elements from the original structure so that it becomes one of the desired substructures, and insert all the deleted elements into a new structure. This can be done in $O(t_1)$ time as we assume that the operations of insert or delete in our static structures each takes $O(1)$ time. (The assumption that insert or delete each takes $O(1)$ time is justified for the hard sphere model of a molecule, as long as we take care that the molecule does not roll into conformations that induce a large volume of steric interference. Roughly, this requires that in a sequence of updates whenever we detect steric interference beyond a preset threshold we do not bend further into this conformation.)

Similarly to merge two structures into one we can either build the new structure from scratch or move the smaller into the bigger. We will tackle a few slight variations as we go on in the analysis, but this should give the flavor of why we choose the cost measures that we describe in the paper.

3 Worst-Case, Amortized, and Randomized Analysis

In this section we characterize the complexity of the update and query operations in each of the following settings: worst-case, amortized, and randomized. We begin by considering the case of paths, and then extend our results to trees.

Theorem 1 *There is an algorithm that maintains this data structure at a worst-case cost of $O(\sqrt{n})$ per operation, whether we use the MIN or the TOTAL cost measures.*

Proof: The idea is very simple: the algorithm chooses the initial state to be one where the UNLOCKED edges are spaced regularly at distance of $\sqrt{n}$. This state remains fixed throughout the execution 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 UNLOCKED edges in the entire path. Further, any update operation has cost $O(\sqrt{n})$ since all subpaths are of length $\sqrt{n}$. Note that the algorithm will break an edge for an update operation, but will then LOCK it right after that unless the edge is one of the initially UNLOCKED edges. The cost of the LOCK operation at most doubles the cost of the UPDATE operation. ■

This bound is tight, even in the amortized and randomized settings.

Theorem 2 *Any algorithm for maintaining this 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, whether we use the MIN or the 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 UNLOCKED edges exceeds $\sqrt{n}$, it inputs the operation QUERY(v_1, v_n) and this has a cost $\Omega(\sqrt{n})$. On the other hand, if the number of UNLOCKED edges is fewer than $\sqrt{n}$, then there exists a subpath P_j with more than $\sqrt{n}$ edges. 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 the above adversary strategy is completely impervious to the strategy of the algorithm and can create an input sequence of an arbitrary length where *each* operation costs

$\Omega(\sqrt{n})$. Quite clearly then, the lower bound carries over to the *amortized* cost of operations without any changes.

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, while the QUERY operation refers to the entire path. Suppose that the data structure has more than $\sqrt{n}$ UNLOCKED 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 $\sqrt{n}$ UNLOCKED 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 $\Omega(\sqrt{n})$. ■

Remark 2 *The latter theorem illustrates the need for applying competitive analysis to this problem. It is fairly easy to see that this problem is a special case of the* metrical task systems *formulation of Borodin, Linial, and Saks [2]. However, it also appears that the special structure of this problem should lead to far more efficient solutions than the ones given by them.*

3.1 Extension to Trees

To understand the case of trees, it is instructive to first examine the other extreme from paths, i.e., stars. By analogy with paths, we may at first think that the goal is to find $\sqrt{n}$ edges which decomposes the star into in subtrees of size $O(\sqrt{n})$ each. Clearly, this is impossible and so it may seem that we will have to pay a linear worst-case cost for the UPDATE/QUERY operations. 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. This is because under the MIN cost measure, the cost of an UPDATE operation is $O(1)$ for each edge, even if it is LOCKED. Thus, for stars, the right solution is to not UNLOCK 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 *k of the edge is defined to be $k = \min\{|T_i|, |T_j|\}$. An edge of heaviness k is said to be k'*-heavy *for any $k' \leq k$.*

Basically, the "heaviness" of an edge is the cost of unlocking or updating the edge.

We now extend the notion of heaviness to the entire tree.

Definition 2 *The* heaviness *k 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 $k' \leq k$.*

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 $T_1, T_2, \ldots, T_k$ (some of which may be empty) none of which is k-heavy. Such a decomposition is called a k-balanced decomposition of T.*

Note that an edge that is not k-heavy in some T_i could be k-heavy in the original tree T, and in the above definition we are considering the heaviness of the edges in each T_i with respect to that tree itself.

Before relating the balance number κ to the data structure problem, we will need to determine the value of κ and, in fact, to find the $\kappa - 1$ edges that induce the balanced decomposition. This is not immediately obvious. The following lemma seems useful.

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, and therefore e is k-heavy. ■

We now turn to the task of determining the balance number κ and the set of κ edges whose deletion gives a κ-balanced decomposition into subtrees that are not κ-heavy. Note that we can do binary search on the value of κ, so it basically boils down to the issue of deciding whether a tree has balance number $\kappa = k$ for some given value k. We prove the following theorem.

Theorem 3 *Given a tree T with balance number κ, the κ-balanced decomposition of T can be computed in $O(n^2\kappa)$ time.*

Proof: Consider an edge $e = (v_1, v_2)$ and 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. Assume, without loss of generality, that $|T_1| \leq |T_2|$. We will say that e is *κ-critical* if the tree T_1 is not κ-heavy, but the addition of the vertex v_2 (and the edge e) to T_1 gives a tree T_1' that is κ-heavy.

Consider the following greedy algorithm: pick any κ-critical edge e and delete it; set aside the tree T_1 that is not κ-heavy, and recurse on the residual tree T_2. We terminate in κ stages with a κ-balanced decomposition, or possibly report that the tree is *not* κ-balanced. The analysis of the greedy algorithm is based on the following monotonicity property: if T' is a subtree of a tree T and T' is a k-heavy tree, then T must also be k-heavy.

The claim is that this algorithm deletes the smallest possible number of edges so as to decompose T into subtrees that are not κ-heavy. This can be established (inductively) as follows. Let OPT be some optimal set of edges to delete to ensure that T decomposes into subtrees that are not κ-heavy. If OPT contains the edge e chosen at the first step, then we are done. Otherwise, OPT must delete some edge in T_1; clearly, removing all such edges from OPT and adding e to the remaining set gives a set OPT' of (at most) the same size as OPT. We claim that OPT' must be also be an optimal set of edges to delete to obtain a κ-balanced decomposition. Since $OPT' \setminus e$ must be an optimal solution for the residual subtree T_2, we can apply induction to complete the proof.

We now show that the greedy algorithm can be implemented in $O(n^2\kappa)$ time. At any stage, we consider each edge in turn and determine whether it is κ-critical. This requires $O(n)$ time since the heaviness of a tree can be computed in linear time using depth-first search. The total work for each stage of the greedy algorithm is $O(n^2)$, and since the number of stages cannot exceed κ, the time bound follows. ■

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

Theorem 4 *Let T be a tree with balance number κ. There is an algorithm that maintains a kinematic data structure at a worst-case cost of $O(\kappa)$ per operation, using 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 unlocked. The cost a QUERY is clearly at most κ. An UPDATE is also going to cost at most κ since in each of the connected subtrees none of the edges are κ-heavy, ■

This bound is tight, even in the amortized setting.

Theorem 5 *Let T be a tree with balance number κ. Any algorithm for maintaining a 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 UNLOCKED edges exceeds κ, it inputs a QUERY operation and this has a cost $\Omega(\kappa)$. On the other hand, if the number of UNLOCKED edges is fewer than κ, then there exists a subtree T_j which is κ-heavy and contains a κ-heavy edge e. The adversary then inputs an UPDATE operation involving e and this costs $\Omega(\kappa)$.

Notice that the above adversary strategy 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. ■

4 The TOTAL Problem

In this section, we focus on the TOTAL problem. Interpreting the problem in geometric terms, we obtain that the it is NP-complete. We also indicate briefly the known results for approximations to the problem.

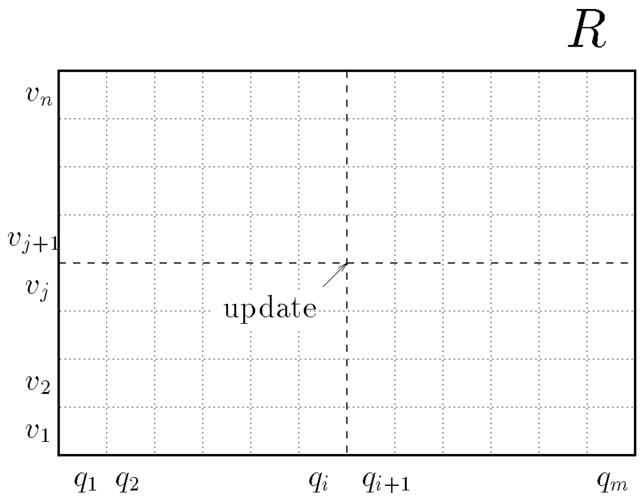

Figure 1: A geometric interpretation of the problem with PATH and TOTAL

4.1 The TOTAL Problem is NP-complete

Let $q_1, q_2, \ldots, q_m$ denote the queries and recall that $v_1, v_2, \ldots, v_n$ denote the nodes 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 node 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}; see Figure 1. 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 $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ 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 unlocked interior edge along the path $v_i, v_{i+1}, \ldots, v_j$ was locked, and each of the exterior edges of the path, namely the edges (v_{i-1}, v_i) and (v_j, v_{j+1}), was unlocked (if it was previously locked), and all the interior edges of this path remain locked until the completion of query q_l.

We claim that an optimal (i.e., minimum cost) plan for the problem with PATH and TOTAL 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 (resp. w_i) is the length in unit grid size of the horizontal (resp. vertical) edge of R_i, and such that no rectangle R_i contains a point of $\mathcal{P}$ in its interior.

To see why this is true 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, as we assumed that the cost of a path query is equal to the number of rigid components in the path. So we charge the cost of the query per rectangle R_i, to the portion of the lower

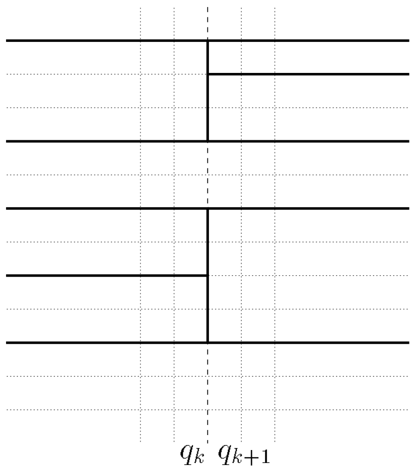

Figure 2: The cost of restructuring between queries q_k and q_{k+1}

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} (see Figure 2). 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 and LOCK 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 $\mathcal{P}$ in its interior. If a rectangle contains a point $P_j \in \mathcal{P}$ in its interior this means that our data structure has not been updated by the update operation corresponding to the point P_j.

Lingas et al. [13] have shown that partitioning an axis-parallel rectangle R with N point holes into axis-parallel rectangles with minimum total edge length, and such that no rectangle in the partitioning contains a point hole in its interior, is an NP-complete problem. Hence we can state the following theorem

Theorem 6 *The problem with PATH and TOTAL is NP-complete.*

4.2 Approximation Algorithms for the TOTAL Problem

A number of approximation algorithms have been proposed over the years 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 [5]. Finding the optimal guillotine partition is solvable in polynomial time, and it has been shown [5] that the optimal guillotine partition has edge length no greater than 1.75 times the length of the optimal rectangular partition.

Gonzalez and Zheng [5] 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 [8] give a simple $O(N \log N)$ algorithm that obtains a factor 4 approximation for the same problem.

5 Further Results and Work in Progress

We currently exploring several different aspects of the basic problem discussed above. In the near future, we intend to prepare a revision to this report describing our efforts. The following are some of the issues we have considered.

- Providing fast approximation algorithms for the offline problem under the MIN cost measure.
- The online version of these problems can be viewed from the perspective of competitive analysis. There are several ways to model the online setting in this respect, including metrical task systems [2], but we are particularly interested in a model we have developed that is a strict generalization of the k-server problem [14]. We will report this model and our observations concerning it in a future version of this article.

References

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

[2] 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.

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

[4] M.L. Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221 (1983), pp. 709–713.

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

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

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

[8] 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.

[9] 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.

[10] F.K. Hwang. An $O(n \log n)$ algorithm for rectilinear minimal spanning tree. *J. ACM* 26 (1979), pp. 177–182.

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

[12] C. Levcopoulos and A. Lingas. Bounds on the Length of Convex Partitions of Polygons. In *Proceedings of the 4th Conference Found. Softw. Tech. Theoret. Comput. Sci.*, Lecture Notes in Computer Science 181, Springer-Verlag, pp. 279–295.

[13] 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.

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

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

[16] P.G. Mezey. Molecular surfaces, in *Reviews in Computational Chemistry*, Volume I. K.B. Lipkowitz and D.B. Boyd, Eds., VCH Publishers, 1990.

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

[18] M. Yim. *Locomotion with a Unit-Modular Reconfigurable Robot*. PhD thesis, Stanford Univ., December 1994. Stanford Technical Report STAN-CS-94-1536.