THE MOLECULE PROBLEM EXPLOITING STRUCTURE IN GLOBAL OPTIMIZATION*

BRUCE HENDRICKSON†

Abstract. The molecule problem is that of determining the relative locations of a set of objects in Euclidean space relying only upon a sparse set of pairwise distance measurements. This NP–hard problem has applications in the determination of molecular conformation. The molecule problem can be naturally expressed as a continuous, global optimization problem, but it also has a rich combinatorial structure. This paper investigates how that structure can be exploited to simplify the optimization problem. In particular, we present a novel divide–and–conquer algorithm in which a large global optimization problem is replaced by a sequence of smaller ones. Since the cost of the optimization can grow exponentially with problem size, this approach holds the promise of a substantial improvement in performance. Our algorithmic development relies upon some recently published results in graph theory. We describe an implementation of this algorithm and report some results of its performance on a sample molecule.

1. Introduction. Consider a set of objects in Euclidean three–space at unknown locations. We wish to determine the relative locations of the objects, but the only information available to us is some subset of their pairwise distances. How can we use this information to compute their positions? We call this the *molecule problem*. It has obvious applications in surveying and satellite ranging [19, 31], and a less obvious but potentially more important application in determining molecular conformation. It is possible to interpret the nuclear magnetic resonance spectra of a molecule to obtain pairwise interatomic distance information [10, 33, 32]. Solving the molecule problem in this context would determine the three–dimensional shape of the molecule, which is critical for understanding its chemical and biological properties.

The data in an instance of the molecule problem can be succinctly represented by a graph $G = (V, E)$. The vertices V correspond to the objects or atoms, and an edge $e_{ij} \in E$ connects vertices i and j if the distance between the corresponding objects is known. We will denote the number of vertices and edges by n and m respectively and the distance associated with edge e_{ij} by d_{ij}. A *realization* of a graph is a mapping p that takes each vertex to a point in Euclidean space. (Some authors prefer the term *embedding*, but a realization need not be an embedding in the strict topological sense).

The molecule problem can be naturally phrased as a nonlinear global optimization problem. Denoting the position of a vertex i as p_i, we can construct a simple cost function $F(p)$ that penalizes a realization for unsatisfied constraints. One simple such function is

$$F(p) = \sum_{e_{ij}} (|p_i - p_j|^2 - d_{ij}^2)^2, \tag{1}$$

where $|\cdot|$ denotes the Euclidean norm. This function is everywhere infinitely differentiable, and (assuming all the distances are correctly given) it has a global minimum of zero, attained when all the distance constraints are satisfied. In principle, $F(p)$

* To appear in the SIAM J. Opt.

† Mathematics and Computational Science Department, Sandia National Laboratories, Albuquerque, NM 87185; bahendr@cs.sandia.gov. Research performed while the author was at Cornell University, supported by a fellowship from the Fannie and John Hertz Foundation.

could be used with any global optimization technique to solve the molecule problem. Unfortunately, this naive approach is unlikely to work well in practice, due to the computational complexity of the problem. Saxe has shown that the molecule problem is strongly NP–complete in one dimension, and strongly NP–hard in higher dimensions [29], so it is unlikely that a general polynomial time algorithm exists. It can also be shown that $F(p)$ can have an exponential number of local minimizers, which makes the global optimization problem daunting.

In this paper, we describe an approach to the molecule problem that attempts to avoid having to solve a large global optimization by instead solving a sequence of smaller optimizations. Since the cost of an optimization can grow exponentially with the size of the problem, this approach holds the prospect of a substantial reduction in computational effort. To achieve this reduction, we will need to exploit some complex combinatorial structure inherent in the molecule problem, which will allow us to devise a novel divide–and–conquer algorithm. Although an important computer science paradigm, divide–and–conquer methods have not previously found many applications in optimization. The purpose of this paper is twofold: on the one hand we present a novel algorithm for a practically important optimization problem, and on the other we provide a case study of how divide–and–conquer ideas can be applied to optimization. It is our hope that the underlying ideas will find application to a variety of additional problems, a possibility we will reconsider in our conclusions.

A simple observation underlies our divide–and–conquer approach to the molecule problem. Within a large problem there are often subproblems that can be solved independently. If we can identify a subgraph that has many edges, it may be possible to determine the relative positions of the vertices in the subgraph by only considering the subgraph's induced edge constraints. Once this subproblem is solved, the entire subgraph can be treated as a rigid body. In three–space, a rigid body has six degrees of freedom, but considered independently each vertex has three. So by treating a set of vertices collectively the number of variables in the problem can be substantially reduced, greatly simplifying the original problem. Using this approach, the initial large optimization problem is replaced by a sequence of smaller ones.

If we are to treat a subgraph as a rigid body, we must be certain that the relative positions of the vertices in the rigid body match their relative positions in the solution to the full problem. This can only be guaranteed if the subgraph allows only a single, unique realization (modulo translations, rotations and reflections), a property we refer to as *unique realizability*. In addition to characterizing uniquely realizable graphs, we need to be able to find subgraphs that have this property within the larger graph that represents the full problem.

This approach to the the molecule problem has been implemented in a code named ABBIE. Since it decomposes a large global optimization into a sequence of smaller, more localized problems, the program is named in honor of Abbie Hoffman for his admonition to "think globally, act locally," although it is doubtful he had nonlinear optimization in mind! The structure of the ABBIE program is depicted in Fig. 1.

For the purposes of this paper we need to make two assumptions about the data, which make for an idealized problem. The first assumption is that we know edge distances with a high degree of accuracy. The second is that there is no special relationship among the locations of the vertices that generated the data for the problem. More formally, a realization is said to be *generic* if the vertex coordinates are algebraically independent over the rationals. We will assume the realization that generated our data was generic, but this is actually a much stronger condition than we

```
(1)   Find maximal uniquely realizable subgraphs
      For Each such subgraph
          If subgraph is small enough Then
(2)           Position graph with global optimization
(3)       Else Break into smaller pieces
              For Each piece call ABBIE
(4)           Combine pieces with global optimization
      Return (realized subgraphs)
```

Fig. 1. The structure of the ABBIE program.

need. There is only a small set of algebraic dependencies that we need to avoid in the uniqueness analysis. However, within the space of all realizations, the set of generic realizations is dense, and the non–generic realizations comprise a set of measure zero. These assumptions are unrealistic for true data, which can be noisy and imprecise, but they are necessary for the formal derivation of the algorithm. We believe these constraints can be relaxed somewhat in practice as we will discuss in the conclusions.

Most previous work on the molecule problem has been performed by chemists interested in molecular conformation. Various heuristics have been developed that rely in various ways upon knowledge about chemical structures. A survey of this previous work is beyond the scope of this paper, but a good overview can be found in Chapter 6 of [10]. A more detailed description of the current work is provided in [14].

This paper is structured as follows. The characterization of uniquely realizable graphs is the topic of the next section. This is followed in §3 by an algorithm to identify uniquely realizable subgraphs, step (1) in Fig. 1. In §4 we describe ABBIE's technique for breaking a large, uniquely realizable graph into pieces, step (3) in Fig. 1. To finally determine coordinates, steps (2) and (4), ABBIE uses a global optimization procedure which is described in §5. Experimental results are presented in §6, followed by discussion and conclusions in §7.

2. Conditions for unique realizability. Does an instance of the molecule problem have a unique solution? Saxe has shown this problem to be as difficult as the original molecule problem: strongly NP–complete in one dimension, and strongly NP–hard in higher dimensions [29]. However, these results depend upon very special realizations in which the locations of the vertices are not algebraically independent. Since we are assuming that our problem is generic, these cases can be excluded, and the uniqueness question becomes much easier. Strong results can be derived that depend only upon the underlying graph, independent of the edge lengths.

Two independent necessary conditions for unique realizability are established in [15], along with algorithms for their detection, and we briefly summarize these below. Unfortunately, in three and higher dimensions these conditions aren't sufficient. We present a sufficiency condition for unique realizability in §2.3 and an algorithm for identifying it in §2.3.1.

The two independent necessary conditions derived in [15] for a graph to be uniquely realizable in d–dimensions are:
(1) vertex $(d+1)$–connectivity, and
(2) redundant rigidity.
Of these, vertex connectivity is a well studied graph property and, efficient algorithms for verifying $(d+1)$–connectivity have been developed [1, 5, 16, 18]. Redundant rigidity is less familiar, but efficient algorithms are known [15] and reviewed below. For

later reference we will need to review some simple rigidity theory in §2.1 and §2.2; a more complete discussion can be found in some of the references [2, 3, 9, 28]. In §2.3 a previously unpublished sufficient condition for unique realizability is derived and an algorithm to identify it is sketched.

2.1. Graph rigidity. We will call the the combination of a graph G and a realization p a *framework*, denoted by $p(G)$. A realization is *satisfying* if all the pairwise distance constraints are satisfied. Intuitively, a framework is flexible if the vertices can move while keeping all the edge distance constraints obeyed. More formally, a *finite flexing* of a framework $p(G)$ is a family of realizations of G, parameterized by t so that the location of each vertex i is a differentiable function of t and

$$\forall e_{ij} \in E, \quad (p_i(t) - p_j(t))^2 = \mathit{constant}. \tag{2}$$

Note that a motion of the Euclidean space itself, a rotation or translation, satisfies the definition of a finite flexing, and such flexings are said to be *trivial*. If the only finite flexings allowed by a framework are trivial, then the framework is said to be *rigid*. Otherwise it is *flexible*. In d–space there are $d(d+1)/2$ independent trivial flexings.

A linearized version of flexibility is more convenient, so thinking of t as time and differentiating (2) we find that

$$\forall e_{ij} \in E, \quad (v_i - v_j) \cdot (p_i - p_j) = 0, \tag{3}$$

where v_i is the instantaneous velocity of vertex i. An assignment of velocities that satisfies (3) for a particular framework is an *infinitesimal motion* of that framework. If a framework has a nontrivial infinitesimal motion it is *infinitesimally flexible*, and if not it is *infinitesimally rigid.*

Clearly the existence of a finite flexing implies an infinitesimal motion, but the converse is not always true. However, for generic realizations the converse is true [28], and since we are considering only generic realizations we will drop the prefix and refer to frameworks as either rigid or flexible. Whether a generic framework is rigid or flexible is purely a property of the underlying graph as indicated by the following theorem [13].

THEOREM 2.1 (GLUCK). *If a graph has a single infinitesimally rigid realization, then all its generic realizations are infinitesimally rigid.*

This theorem is crucial for a graph theoretic approach to the molecule problem. Since the frameworks built from a graph are either all infinitesimally flexible or almost all rigid, graphs can be characterized according to the typical behavior of their frameworks, without reference to a specific realization. This also allows us to be somewhat cavalier in the distinction between rigid frameworks and graphs that have rigid realizations. Henceforth such graphs will be referred to as *rigid graphs.*

In one dimension, rigidity is equivalent to connectivity. In two dimensions a combinatorial characterization of rigidity was first discovered by Laman [20], and several different $O(n^2)$ algorithms for rigidity testing have been developed [11, 15, 17].

In three and higher dimensions, no combinatorial characterization of rigidity is known. However, there is an efficient randomized algorithm based on Theorem 2.1 and (3). Begin by randomly positioning all the vertices. With probability one, the rigidity of the corresponding framework will be the same as that of the graph. Now construct the set of equations (3), where the velocities are the unknowns. The coefficients of the velocities can be formed into a matrix of size $m \times nd$, known as the *rigidity matrix*, denoted by M. Each row of M corresponds to the constraint imposed by a single edge. The null space of this matrix represents the allowed infinitesimal motions of the framework. Clearly the $d(d+1)/2$ trivial infinitesimal motions are in the null space. So if the rank of the rigidity matrix is $nd - d(d+1)/2$, then with probability one the graph is rigid, otherwise it is flexible.

2.2. Redundant rigidity. A graph is defined to be *redundantly rigid* if it is rigid after the removal of any single edge. Redundant rigidity is a necessary condition for a graph to be generically uniquely realizable [15]. We will define an edge of a rigid graph to be *redundant* if the graph remains rigid after the removal of the edge.

In one dimension, redundant rigidity is equivalent to edge biconnectivity. In two dimensions, an efficient algorithm built upon the combinatorial characterization of rigidity is described in [15]. In higher dimensions, since no graph theoretic characterization of rigidity is known, no characterization of redundant rigidity exists either. However, the randomized approach for rigidity testing described above can be extended to check for redundant rigidity.

Since rows of the rigidity matrix, M, correspond to edges of the graph, a framework is redundantly rigid if and only if M^T has maximal rank after the removal of any single column. A column of M^T is said to be *redundant* if the rank of M^T remains the same after its removal. If M^T has a set of $nd - d(d+1)/2$ linearly independent, redundant columns, then the framework is redundantly rigid.

Our algorithm for redundant rigidity builds upon a QR factorization of M^T. We maintain a list of linearly independent columns, and a new column is added to the list if it is linearly independent of the current set, otherwise it is discarded. A discarded column of M^T can be expressed as a linear combination of some set of the independent columns. The discarded column could replace any of the columns in the linear combination which forms it, without altering the span of the independent set. Hence, a discarded column identifies a set of redundant columns within the list.

The columns within the list whose linear combination equals a discarded column can be easily determined. Assume the algorithm has identified k independent columns of M^T, placed together to form an $nd \times k$ matrix, A_k. The QR factorization has been proceeding on these columns as they are identified, so there is a $k \times k$ orthogonal matrix Q_k and an $nd \times k$ upper triangular matrix R_k satisfying $Q_k R_k = A_k$. If a new column b of M^T is linearly dependent upon the columns of A_k then there must be a vector c satisfying $A_k c = Q_k R_k c = b$, or alternately $R_k c = Q_k^T b$. In the course of the QR factorization the column b has been overwritten with $Q_k^T b$, so it is easy to solve the upper triangular system for c. The nonzero elements of c identify which columns of A_k contribute to the linear combination composing b, that is, which columns are redundant.

This procedure requires the solution of $O(m)$ triangular systems, each of which requires $O(k^2)$ operations, where k is always $O(nd)$, so the total additional time is of the same order as the QR factorization itself, $O(mn^2d^2)$.

2.3. A sufficient condition for unique realizability. In one dimension, the necessary conditions for generic unique realizability discussed above reduce to edge biconnectivity, which can also be shown to be sufficient. In two dimensions, we know of no examples of graphs that satisfy the necessary conditions while not being unique, but the sufficiency of these conditions has not been proven.

In three and higher dimensions, Connelly has discovered a set of bipartite graphs that satisfy the necessary conditions from above, while still allowing multiple realizations [6, 8]. In three dimensions the only graph in this set is $K_{5,5}$, the complete bipartite graph with five vertices in each partition. This class of graphs was identified by the unusual properties of their *stress matrices*, an exploration of which will lead us to a sufficient condition for unique realizability.

Consider a framework $p(G)$ consisting of a graph G and a generic satisfying realization p. A *stress* for $p(G)$ is an assignment of scalars $\omega_{ij} = \omega_{ji}$ to every every pair of vertices of G in such a way that $\omega_{ij} = 0$ if $e_{ij} \notin E$, and

$$\sum_{j=1}^{n} \omega_{ij}(p_i - p_j) = 0, \qquad \forall i, \tag{4}$$

where p_k is the location in $\mathbb{R}^d$ of vertex k. Note that these are vector equations since each p_k has d coordinates, so each of the d dimensions must satisfy an identical set of equations.

The concept of a stress comes originally from mechanical engineering where the edges would be considered to be cables or struts under tension or compression. The framework will be in equilibrium exactly when the vector sum of all the stresses on each vertex is zero, which is the condition expressed by (4).

Equation (4) defines a stress for a particular realization p. In general, this same set of values ω_{ij} will not be a stress for a different realization. However, there is a very important exception to this general rule. Stresses are useful for our purposes because of the following result due to Connelly [7].

THEOREM 2.2. *Let p be a generic, satisfying realization of G in $\mathbb{R}^d$ in which the affine span of the locations of the vertices is d–dimensional. If ω is a stress for $p(G)$ then ω is a stress for any satisfying realization of G.*

This theorem allows us to greatly narrow down our search for alternate satisfying realizations. Once we generate a stress for p we only need to consider realizations q that satisfy the same stress equations.

Assume we have generated a stress for our initial satisfying realization p. We wish to find a q that can replace p in (4). It will be convenient to rewrite the stress equations. Let q_i^r denote coordinate r of the location of vertex i in realization q. For each $1 \leq i \leq n$ and each $1 \leq r \leq d$ we have the following equation.

$$\Big(\sum_{j=1}^{n} \omega_{ij}\Big)\, q_i^r - \sum_{j=1}^{n} \omega_{ij} q_j^r = 0. \tag{5}$$

This is just a set of n linear equations repeated for each of the d dimensions. Define the symmetric, $n \times n$ *stress matrix*, Ω, as follows.

$$\Omega_{i,j} = \begin{cases} -\omega_{ij} & \text{if } i \neq j \\ \sum_k \omega_{ik} & \text{if } i = j. \end{cases}$$

If we denote the n–vector consisting of coordinate r of each vertex by q^r, then (5) can be succinctly expressed as

$$\Omega q^r = 0, \tag{6}$$

for each dimension r. Any satisfying realization must satisfy these equations, so our search for alternate satisfying realizations is now reduced to an investigation of the null space of Ω.

Each row of the stress matrix sums to zero, so the vector of 1's is in Ω's null space. The product Ωp^r is identically zero by the construction of the stress. This is true for each of the d coordinates, so the nullity of the stress matrix is at least $d+1$. The linear combinations of these trivial null vectors are the affine linear maps of the vertices in realization p. That is, any realization in which q_i, the coordinates of vertex i, can be expressed as $Ap_i + b$ will satisfy the same stress equations as p, where A is any $d \times d$ matrix and b any d–vector. If there is nothing else in the null space of Ω then the only possible alternate satisfying realizations are these affine linear maps, which gives us the following theorem.

THEOREM 2.3. *Let p be a generic, satisfying realization of G in $\mathbb{R}^d$ in which the affine span of the locations of the vertices is d–dimensional. If ω is a stress for $p(G)$ such that Ω has nullity $d+1$, then any satisfying realization of G must be an affine linear map of p.*

Connelly has shown that in one, two and three dimensions these troublesome affine linear maps cannot lead to nonequivalent, satisfying realizations [7]. This gives us the following sufficient condition for a graph to have a unique realization.

THEOREM 2.4. *Let p be a generic, satisfying realization of G in $\mathbb{R}^d$ in which the affine span of the locations of the vertices is d–dimensional, where $d \leq 3$. If ω is a stress for $p(G)$ such that Ω has nullity $d+1$, then there is no nonequivalent, satisfying realization of G.*

Determining whether the stress matrix has the proper nullity is what we will call the *stress test* for unique realizability.

For a given realization, the stresses defined by (4) are solutions to a linear system of equations. As such they can be expressed as polynomials in the coordinates of the vertices. To determine whether or not the stress matrix has nullity 4, simply sum the squares of all the $(n-4) \times (n-4)$ subdeterminants of Ω. This polynomial will be zero if and only if the nullity of the stress matrix is greater than four. Thus we have a polynomial in terms of the coordinates of the vertices that describes our sufficiency condition. If this polynomial is nonzero for any generic realization, then it is nonzero for all generic realizations.

THEOREM 2.5. *The nullity of the stress matrix is a generic property; that is, it has the same value for for all generic realizations.*

COROLLARY 2.6. *If any generic realization passes the stress test, then all generic realizations will pass.*

In other words, the stress test is generic. Our necessary conditions were generic as well, which provides evidence that unique realizability may itself be a generic property. Whether or not this is the case is an open problem. Corollary 2.6 justifies using a random realization to generate the stresses. As we will see in the next section, a particularly convenient realization to use is the one that was utilized to generate the rigidity matrix for our redundant rigidity algorithm.

2.3.1. Forming the stress matrix. The sufficient condition for unique realizability expressed by Theorem 2.4 is not much use for us unless we can readily compute stresses. Fortunately, this is not a problem. In fact, most of the work has already been done in the QR factorization of the rigidity matrix M that was described in §2.2. Redundant edges of the graph were identified by linear dependence among the columns of M. Element $[e(i,j), di+r]$ of M is $p_i^r - p_j^r$ if the edge numbered $e(i,j)$ connects vertices i and j, and zero otherwise. Consequently, if the multipliers in a linear combination of columns of M summing to zero are denoted by $\alpha_{e(i,j)}$ for edge

$e(i,j)$, then for each $1 \le i \le n$ and $1 \le r \le d$

$$\begin{aligned} 0 &= \sum_e \alpha_{e(i,j)} M_{e(i,j),di+r} \\ &= \sum_j \alpha_{e(i,j)} (p_i^r - p_j^r) \end{aligned}$$

Equating $\alpha_{e(i,j)}$ with ω_{ij} in (4) we see that the multipliers in the linear combination constitute a stress. Consequently, the solution vector to the triangular systems solved in §2.2 identifies a stress.

In the course of a full redundant rigidity calculation many stresses may be found; one for every discarded row. Each of these stresses generates its own stress matrix, and any linear combination of stresses is also a stress. Since we are interested in identifying a stress that maximizes the rank of Ω, almost any linear combination of the stresses generated in the QR factorization will suffice. In practice we use a sum of all the stresses, scaled by random multipliers.

The determination of the rank of the stress matrix can be troublesome due to numerical roundoff problems. The entries in the stress matrix are the result of a previous factorization, so they may already have modest inaccuracies. For this reason it is important to determine the rank of Ω in as numerically stable a fashion as possible, so we recommend a singular value decomposition.

3. Finding uniquely realizable subgraphs. The preceding section described two necessary conditions and a sufficient condition for a graph to have a unique realization. Step (1) of the algorithm sketched in Fig. 1 requires a further step, the identification of subgraphs that are uniquely realizable. Ideally, we would like to find subgraphs that satisfy the sufficiency condition, but it is not clear how the stress test can be used directly for this purpose. However, the necessary conditions are well suited for identifying subgraphs, which suggests using the necessary conditions to find candidate subgraphs, and then confirming their uniqueness with the sufficiency test. An outline such an algorithm is presented in Fig. 2.

```
If Graph is K_{5,5} Then
    Return (No_unique_subgraphs)
Else If not four–connected Then
    Recurse on four–connected components
Else If not redundantly rigid Then
    recurse on redundantly rigid components
Else Perform sufficiency test
    If Pass Then
        Return (Graph_unique)
    Else Output interesting graph
```

Fig. 2. ABBIE's algorithm for finding maximal uniquely realizable subgraphs.

The only case that is not handled with this approach is a graph that passes the necessary conditions and fails the sufficiency test. We have yet to discover such a graph, although we would be very interested in finding one. In practice this approach seems to work very well, at least on the test problems that will be described in §6. Incidentally, our failure to find any graphs that pass the necessary conditions while failing the sufficiency test provides evidence that such graphs are uncommon if they exist at all.

The heart of the procedure described in Fig. 2, is finding $(d+1)$–vertex connected subgraphs and redundantly rigid subgraphs. The vertex connectivity problem is well studied, and good algorithms for finding maximal subgraphs are known [1, 5, 16, 18]. However, algorithms for finding redundantly rigid subgraphs have not been previously considered. In one dimension, this requires finding biconnected components, for which there are $O(m)$ algorithms [1]. In two dimensions, an $O(n^2)$ algorithm for finding maximal redundantly rigid components is given in [15]. In three and higher dimensions, an algorithm needs to rely upon the QR factorization of the transpose of the rigidity matrix. ABBIE's algorithm, summarized in Fig 3 below, relies upon the observation that any edge that is not redundant in the original graph will not be redundant in any subgraph. After removing these nonredundant edges, the flexes that remain will not effect the redundantly rigid subgraphs. These subgraphs can be identified by noticing which subsets of vertices preserve their relative locations under the remaining flexes, which requires the construction of a basis for the remaining flexes.

A basis for the space of flexes can be generated by the QR factorization of the columns of the transpose of the rigidity matrix that correspond to an independent set of redundant edges. It is helpful to exclude the trivial motions from the basis by explicitly adding them as columns at the end of the factorization. This reduces the size of the space of flexes and so speeds up the determination of subgraphs. If there are k redundant, independent columns, then the final $3n - 6 - k$ columns of Q form a basis for the flexes. Sets of vertices whose relative positions remain unchanged under these flexes are redundantly rigid subgraphs.

Identifying these sets of vertices requires the ability to determine whether the distance between any two vertices i and j changes under any of the allowed flexes. For each flex this involves the calculation of the inner product $(v_i - v_j) \cdot (p_i - p_j)$, where v_i is the velocity vector of vertex i under the flex, and p_i is its location. If this quantity is zero then the distance between i and j remains unchanged.

Pairs of vertices whose distance doesn't change under any of the allowed flexes could just as well be connected by an edge, so will consider such vertices to be joined by an *induced edge*. Finding sets of relatively fixed vertices corresponds to finding cliques in this graph of induced edges. A simple geometric observation simplifies this task. Let $\mathcal{S}$ be a set of at least three vertices whose relative positions don't vary. To determine whether a new vertex v should be added to $\mathcal{S}$ it is sufficient to check the change in the distance from v to any three vertices in $\mathcal{S}$. With three neighbors at fixed locations the position of v cannot vary continuously.

ABBIE's algorithm for identifying these cliques begins by looking for sets of three vertices whose relative locations are fixed. This requires $O(n^3)$ time. Once such a triangle is found, the unique clique containing it can be grown to maximal size by checking all other vertices against these three in $O(n)$ time. Although the resulting $O(n^4)$ algorithm is asymptotically the most expensive portion of the decomposition, for the problems discussed in §6, the cost of the entire component finding process less than 1% of the cost of the QR factorizations.

4. Breaking large graphs. A maximal uniquely realizable subgraph may be large, and consequently prohibitively expensive to try to realize directly. As described in Fig. 1, ABBIE breaks such a subgraph into pieces and recurses on the pieces, before trying to fit them back together. Ideally, smaller uniquely realizable subgraphs would

Use QR factorization to identify independent set of redundant edges
Use QR factorization to construct basis for remaining flexes
For All independent three–cliques (x, y, z) in induced graph
    For All other vertices v
        If v has induced edges to x, y and z **Then**
            add v to subgraph containing x, y and z.

Fig. 3. An algorithm for finding redundantly rigid subgraphs.

be found directly, but we don't know how to do this. Instead, as indicated by step (3) of Fig. 1, ABBIE breaks the graph by finding a small vertex separator and recurses on the induced pieces.

In selecting a value for how large a subgraph must be before being broken, a balance must be struck between two extremes. A small value results in a large number of small optimization problems, and potentially difficult optimizations fitting many small pieces together. On the other hand, a large value leads to a smaller number of large optimizations. For the calculations described in §6 a cutoff of 15 vertices was used. The value of this parameter seemed to have a very small impact on overall computation time. The most expensive optimization problems were typically those that occurred higher up in the chain of recursion, involving many more vertices. Decisions about how to handle these relatively small components were not of critical importance.

The fundamental unit of information in the molecule problem is an edge length, so when a graph is broken into pieces, any edge that does not lie in a single component is lost to the recursive positioning procedures. For this reason we would like a decomposition technique that ensures that any two vertices joined by an edge end up in the same component. We would also like the technique to divide the graph into approximately equally sized pieces as this will speed the recursive decomposition. To accomplish these goals, ABBIE was endowed with a procedure to find a small vertex separator, and when the separator set is added to each component no edges are lost.

Forces between atoms are strongly repulsive at small distances, so each atom effectively has an exclusion zone in which no other atoms are located. In addition, distances can only be measured if two atoms aren't too far apart. These geometric constraints place the underlying graphs in the class of k–overlap graphs, which are known to have $O(n^{2/3})$ separators [23]. For the problems described in §6, the separators found were uniformly small.

There are a number of different heuristics for finding small vertex separators, and for no compelling reason, ABBIE uses an algorithm described by Liu [22]. This algorithm uses a minimum degree ordering to generate an initial separator, which is then improved by a bipartite matching technique.

5. The optimization routines in ABBIE. Any program to solve the molecule problem must eventually assign coordinates to vertices. The combinatorial preprocessing described above merely delays this eventuality so that the actual positioning problems are smaller. The positioning problems that ABBIE needs to contend with involve fitting together two types of objects so that a set of distance constraints are satisfied: vertices, and subsets of vertices whose relative positions have already been determined which we will call *chunks*. These chunks can be treated as rigid bodies with at most six degrees of freedom in three–space.

ABBIE solves these problems using a three–phase approach: variable reduction, variable selection and global optimization. In the first phase the program uses a combinatorial analysis to try to merge chunks and vertices together. For instance, if a vertex has four edges connecting it to a chunk then the vertex will generally have a unique location relative to the chunk, so the vertex can be merged into the chunk. This phase reduces the size of the resulting optimization problems and will be described in greater detail in §5.1.

After exhausting its bag of combinatorial tricks ABBIE resorts to a nonlinear, global optimization. All the possible locations and orientations of the vertices and chunks are expressed by a set of translational and rotational variables. Several different sets of variables are possible and ABBIE attempts to select a set that will minimize the cost of the optimization. This selection process is described in §5.2. ABBIE constructs a cost function that penalizes a realization for not satisfying an edge length constraint so that the sum of the penalties will be zero only when all the constraints are satisfied. To find a realization where this function goes to zero ABBIE generates random values for all the variables and uses them as a starting vector for a local minimization. This process is repeated until a zero value is found, indicating that all the constraints are satisfied, and that the locations of the vertices constitute a satisfying realization. Details of this nonlinear optimization will be given in §5.3.

Much more sophisticated techniques to solve this global optimization are possible. In fact, most previous approaches to the molecule problem have focused exclusively on this aspect, as discussed in Chapter 6 of [10]. Our goal in this work was to test the feasibility of the divide–and–conquer approach to the molecule problem, and it is our expectation that the overall approach will be successful, even though the component optimization techniques are quite simplistic. Better global optimization methods like tunneling [21] or efficient stochastic algorithms [27] could transparently replace the routines described in §5.3.

5.1. Combinatorial positioning techniques. To reduce the number of variables in the global optimization ABBIE first tries to combine small numbers of chunks and vertices into larger chunks. ABBIE has five different heuristics for enlarging chunks which are synopsized in Fig. 4. The success of these techniques depends upon specific sets of vertices not being coplanar, which is ensured by the assumption that the final solution is generic. In the first technique, if two chunks have at least 4 vertices in common then they can only be combined uniquely, and ABBIE merges them. Second, if a vertex has four edges incident to a chunk then the vertex can be uniquely positioned relative to the chunk, and the chunk enlarged. ABBIE can use direct edge lengths that were given in the data, or induced lengths generated by a chunk that contains the two vertices.

The remaining three heuristics start with a *base chunk* and add pairs of objects to it. Consider a vertex with three direct or induced edges to the base chunk. We will call such a vertex *three–valent* to the base chunk. The location of the vertex relative to the chunk has only two possibilities, distinguished by a reflection of the vertex through the plane of its neighbors. If this ambiguity can be resolved then the vertex can be added to the chunk. A similar result applies to a chunk that shares three vertices with the base chunk. Such a chunk will also be called *three–valent* to the base chunk. The last three heuristics for enlarging chunks make use of this observation. The third technique allows two three–valent vertices to be added to a chunk if there is a direct or induced edge between the two vertices. The length of this edge is used to resolve the ambiguity of the reflections of the vertices. Note that this technique does not work if the two vertices have the same three neighbors in the base chunk.

The fourth heuristic involves adding a three–valent chunk and a three–valent vertex to the base chunk by resolving the reflections with a direct or induced edge between the vertex and the chunk. Again, if the three adjacent or shared vertices are the same then the ambiguity can not be resolved.

The last technique adds two three–valent chunks to the base chunk. There are several ways in which the reflection ambiguity can be resolved. The two chunks can share a vertex that is not in the base chunk, or there can be a direct or induced edge between vertices in the two chunks that does not involve a vertex in the base chunk. Again, if the the two sets of three shared vertices are the same then the ambiguity can not be resolved.

```
Until No Change
    For All Chunks X
        For All chunks Y, 4–valent to X
            Merge X and Y

        For All vertices v, 4–valent to X
            Merge v into X

        For All pairs of vertices v and w, 3–valent to X
            If valencies differ And reflections can be disambiguated Then
                Merge v and w into X

        For All pairs of chunks Y and Z, 3–valent to X
            If valencies differ And reflections can be disambiguated Then
                Merge X, Y and Z

        For All chunks Y and vertices v, 3–valent to X
            If valencies differ And reflections can be disambiguated Then
                Merge X, Y and v
```

Fig. 4. ABBIE's combinatorial positioning heuristics.

ABBIE applies these techniques to all combinations of chunks and vertices until no more merging is possible. The vast majority of positioning problems encountered in the test problems were resolved this way, without any need for the nonlinear optimizer. Additional, more complicated heuristics are possible, and would probably be worth implementing. As the numerical results in §6 will show, the nonlinear optimizations dominate the execution time of ABBIE. Hence, it is our expectation that the additional cost of more complex techniques would be more than compensated for by the reduction in size, and consequently cost, of the optimization problems.

5.2. Selecting optimization variables. The optimization problems ABBIE must solve involve sets of chunks and vertices. Vertices have three translational degrees of freedom, and the location and orientation of a chunk can be described by three translational and three rotational variables. To describe rotations of chunks, ABBIE defines an axis of rotation using a standard (θ, ϕ) system, and an amount of rotation. This representation has fewer singularities than the more familiar Euler angles.

There are many possible ways to parameterize the motions of the vertices and chunks. For instance, any of the chunks can be held fixed while the others are allowed to move. The selection of optimization variables in ABBIE was designed to minimize the computational effort required by the global optimization. ABBIE solves the global optimization problem by a sequence of local minimizations, so there are two factors which determine the total cost of the global optimization: the cost of each local minimization and the number of local minimizations required to find the global optimum. We need to analyze these two quantities before we can explain the procedure for selecting optimization variables.

To find a local minimizer from a random starting point ABBIE uses a trust region approach, repeatedly forming and factoring the Hessian matrix. This factorization tends to dominate the cost of each iteration, requiring $\Theta(q^3)$ floating point operations, where q is the number of variables. It is difficult to estimate the number of iterations each local minimization will require, as this depends in a complicated way on the local topography of the penalty function, so ABBIE assumes that the cost of each local minimization is simply proportional to the cube of the number of variables.

The number of local minimizations required to find the global optimum depends on the size of the region of attraction of the global minimizer relative to size of the entire domain. Assuming this region of attraction is of average size, the number of local minimizations will be proportional to the number of local minimizers in the problem. Since this number can grow exponentially with the number of vertices, the number of local minimizers can be approximated as $2^{q/\beta}$, where β is an empirical parameter. After some experimentation, ABBIE was given a value of 8 for β for the test problems described in §6, but an appropriate value for this parameter depends on the class of problems under consideration.

There is one additional factor to consider in estimating the cost of an optimization. Note that edge lengths remain unchanged if a chunk is replaced by its mirror image, but there is no continuous rigid body motion to transform between these two realizations. ABBIE cannot know in advance which of these two *parities* is the correct one, so it has to try them both. Since only one will fit properly with the remainder of the graph, if a particular chunk is assigned an arbitrary parity then all the others must be made consistent. Since parities are selected randomly as a local optimization is started, this adds an additional factor of 2^{k-1} to the number of local minimization attempts, where k is the number of chunks in the optimization problem.

Combining these three factors, we can approximate the total cost of a global optimization as $\Theta(q^3 2^{k-1} 2^{q/\beta})$. ABBIE tries to select a set of optimization variables that minimizes this estimated cost function.

In performing a local optimization, one *base* chunk, is held fixed at the origin to remove translational and rotational ambiguities. (If no chunks can be found, a single edge is used, and an additional vertex is constrained to lie in the x–y plane.) ABBIE tries all of the chunks in turn as candidate base chunks, and selects the one that minimizes the estimated cost of the optimization. If a second chunk shares three vertices with the base chunk then it has no continuous degrees of freedom. If a chunk shares two vertices with the base chunk its motions can be described with a single rotational variable. If a chunk has a single vertex in common with the base chunk then it has three degrees of freedom, and if it shares none it has six. All chunks add a factor of two to the parity consideration.

In evaluating the candidate base chunks ABBIE adds the remaining chunks in a greedy manner, trying to minimize the estimated optimization cost. This process is sketched in Fig. 5. At each step ABBIE selects the remaining chunk with the largest number of vertices that are not yet contained in an accepted chunk. This chunk is accepted and the variables describing its motion included in the optimization, or rejected and ignored, depending upon which action reduces the estimated optimization cost. If the chunk is accepted it increases the number of chunks by one, and it can increase the number of variables in the optimization. If it is rejected, its vertices that are not yet in an accepted chunk are assumed to wander freely, each adding three to the number of variables. ABBIE processes all of the remaining chunks in this way, determining the cost of selecting this base chunk, as well as generating a set of variables for the optimization. ABBIE selects the base chunk which generates the lowest estimated optimization cost, and the corresponding variables are used in the global optimization.

```
Best_Cost := ∞
For All vertices v, free(v):= TRUE
For All Candidate base chunks X
    k := 1
    q := 0
    For All vertices v in X, free(v) := FALSE
    While any chunks remain
        Select next chunk Y having maximal free vertices
        t := Number of free vertices in Y
        If Y 3–valent to X Then r := 0
        Else If Y 2–valent to X Then r := 1
        Else If Y 1–valent to X Then r := 3
        Else If Y 0–valent to X Then r := 6
        Accept_Cost := (q + r)^3 2^k 2^((q+r)/β)
        Reject_Cost := (q + 3t)^3 2^(k−1) 2^((q+3t)/β)
        If (Accept_Cost < Reject_Cost) Then
            q := q + r
            k := k + 1
            For All vertices v in Y, free(v) := FALSE
        Else discard Y
    End While
    t := Number of remaining free vertices
    q := q + 3t

    Cost := q^3 2^(k−1) 2^(q/β)
    If (Cost < Best_Cost) Then
        Best_Cost := Cost
        Best_Base_Chunk := X
        Optimization_Variables := those identified above
```

Fig. 5. ABBIE's process for selecting optimization variables

5.3. ABBIE's global optimization technique. To finally position the set of chunks and vertices ABBIE finds the global minimizer of a function that penalizes a realization for violating constraints. Many different penalty functions are possible and ABBIE uses a particularly simple one. For each edge e_{ij} that needs to be satisfied the program computes a value $P^1_{ij} = (|p_i - p_j|^2 - d^2_{ij})^2$, where p_i is the location of vertex i, and d_{ij} is the desired distance between vertices i and j. This is the simplest possible function that has continuous derivatives of all orders and is greater than zero whenever a constraint is violated. The full penalty function for edges is then $F_1 = \sum P^1_{ij}$, where the sum is taken over all edges in the graph which aren't contained in any chunks.

Positioning problems in ABBIE can involve multiple chunks other than the base chunk which share one or more vertices. These vertices must be forced to coincide in a satisfying realization, so ABBIE needs a penalty term to enforce this constraint. The obvious candidate would have the same functional form as that for edges but with a zero distance. Unfortunately, this function has a singular Hessian. For this reason the program uses a simpler penalty $P^2_{ij} = |p_i - p_j|^2$. Summing all of these types of constraints together gives F_2, a second component to the cost function. The full penalty function is then $F = F_1 + F_2$. We note that the full penalty function is composed of both quartic and quadratic functions. For large deviations from satisfiability the quartic terms should dominate the quadratic ones, while near the solution the opposite should occur. In practice this seemed to cause no problems.

To find a zero of this penalty function ABBIE generates a random starting value for each of the optimization variables, including random parities for each chunk. The program then performs a local minimization, and this process is repeated until a functional value of zero (or almost zero) is found or until a limit on the number of trials is exceeded. This is an extremely simple global optimizer and much more sophisticated techniques could easily be used instead.

For local optimization ABBIE uses a modified version of the NTRUST code of Moré and Sorensen that is based upon the trust region method described in [24]. This approach was selected for ABBIE because trust region techniques tend to be robust, and our function and its derivatives are fairly inexpensive to evaluate explicitly. NTRUST treats the Hessian as dense, which can be inefficient, but this is not a serious problem for ABBIE since the divide–and–conquer approach avoids large problems. The ability to scale variables was added to NTRUST to cope with the different ranges associated with translational and rotational variables.

6. Results. ABBIE has been tested on simulated molecular data provided by Palmer [25, 26]. The input data consisted of simulated distance constraints, corresponding to measurements that could be made in a typical NMR experiment. However, in our case the distances were given precisely, whereas true experimental data has limited precision. The molecule that generated our test problems was bovine pancreatic ribonuclease A, a typical small protein consisting of 124 amino acids and, after discarding end chains, 1849 atoms. The three–dimensional conformation of ribonuclease is known, so all pairwise distances could be determined. For our purposes, the data set consisted of all distances between pairs of atoms in the same amino acid, along with 1167 additional distances corresponding to pairs of hydrogen atoms that were within 3.5 Å of each other. The former set of values can be deduced from the chemical structure, and the latter could in principle be measured by two–dimensional NMR spectroscopy experiments. Combined this made for a total of 15,046 edges.

Proteins are constructed of chains of amino acids. Since the shapes of the amino acids are well known, the conformation of the protein is determined by the angular parameters where the amino acids are joined. In fact, one common approach to the analysis of protein conformation is to treat these angles as the only degrees of freedom [12]. Under this assumption, if the locations of any four non–coplanar atoms in an amino acid can be determined, the locations of the remaining atoms in that amino acid can be easily computed. This allowed us to reduce the size of the graphs that were passed to ABBIE. Within each amino acid, we discarded vertices that had no edges to vertices outside of that amino acid, until there were only four vertices left.

In addition, any amino acid that had six or fewer edges to other amino acids could not be uniquely positioned. These amino acids were removed, further reducing the size of the graph.

A single problem would give only limited insight into the strengths and weakness of ABBIE, so we generated a set of related test problems of varying sizes by extracting leading subchains of amino acids from the ribonuclease. The six different problem sizes used are presented in Table 1. The second column presents the number of vertices and edges in the initial, unadulterated graphs. These graphs were reduced in size by exploiting protein structure as discussed above, resulting in the graph sizes described in the third column. These are the graphs that were passed to the unique realizability algorithms. The final column presents the size of the largest uniquely realizable subgraph that was found within each of the reduced graphs.

Table 1. Sizes of the test problems; vertices (edges).

Amino Acids	Initial Graph	Reduced Graph	Largest Unique Subgraph
20	292 (2263)	63 (236)	57 (218)
40	604 (4902)	186 (828)	174 (786)
60	902 (7264)	310 (1392)	287 (1319)
80	1193 (9556)	405 (1804)	377 (1719)
100	1491 (12038)	504 (2272)	472 (2169)
124	1849 (15046)	698 (3292)	695 (3283)

It is worth noting that the edge density for the full molecule is greater than that for any of the leading subchains. This is a consequence of the tendency of proteins to form compact structures. Leading subchains need not be as compact, and so the number of pairs of atoms that are close together is reduced.

For the runs described below, subgraphs were considered too big to directly realize if they contained more than 15 vertices. All larger subgraphs were divided into pieces using the small vertex separator heuristic from §4. Also, the stress test to verify unique realizability was turned off. Besides the intrinsic reduction in effort, this allowed for some economy in the redundant rigidity calculation [14]. If a subgraph passed the necessary tests, but wasn't truly uniquely realizable, disabling the stress test could lead to incorrect coordinates being computed for the subgraph. However, this would be detected when the subgraph would be used in later optimizations since it would be unable to fit properly with the remainder of the full graph.

6.1. Performance of the unique realizability algorithms. ABBIE's algorithm for finding uniquely realizable subgraphs consists of alternate phases of a four–connected components routine and a redundant rigidity code. The redundant rigidity algorithm requires a QR factorization as described in §2.2. The four–connectivity algorithm in ABBIE removes two vertices at a time and checks for biconnectivity, requiring $\Theta(mn^2)$ time. Although there are asymptotically more efficient algorithms for this step 3-c,4-c, the QR factorization requires $O(n^3)$ time, so a more complex algorithm for four–connectivity was deemed unnecessary. The total time spent in these portions of the code as a function of the reduced graph sizes is presented in Table 2 for the six different problems. These times are all a small fraction of the optimization time. These and all subsequent timings are for CPU time on a Sparcstation 1^+. As expected, both the four–connectivity and the redundant rigidity times grow roughly as the cube of the number of vertices.

Table 2. Total minutes spent in unique realizability routines.

Amino Acids	Redundant Rigidity	Four–Connectivity
20	0.16	0.11
40	4.22	5.36
60	18.82	48.78
80	45.96	57.84
100	84.47	115.04
124	333.09	323.98

Whereas the four–connectivity routines are entirely deterministic, there is a degree of randomness involved in the redundant rigidity calculations. The values in the rigidity matrix come from a random realization of the graph. For some realizations this can lead to numerical problems in the QR factorization. This was observed in practice for the largest problem, involving the factorization of a 1085×3283 matrix. In particular, the factorization had a difficult time determining when a value should be considered to be zero. After several attempts with different random number seeds a realization was generated with excellent numerical properties. For this factorization there was a gap of nearly 5 orders of magnitude between the smallest value that was accepted as nonzero, and the largest that was rejected. Additional runs demonstrated that as long as there was a reasonable gap between these values the redundant rigidity calculations were essentially deterministic.

6.2. The small vertex separator. The algorithm for identifying small separators ran rapidly and produced good separators. For the largest problem the total time spent in the separator routines was only 1.55 minutes, a minuscule fraction of the total running time. A plot of the size of the separator set versus the size of the graph is shown in Fig. 6 for all the invocations of the algorithm in the set of test problems. Except for the smallest graphs, the vast majority of separators have between 5 and 10 percent of the total number of vertices. Note that no separator smaller than 4 could ever be found, for it would imply that the graph was not four–connected, and hence not uniquely realizable.

The idea of using a small separator heuristic was based on our hope that it would typically divide the graph into two halves, each of which had a good chance of being uniquely realizable. The technique succeeded in dividing the graphs into two pieces of approximately equal size, but unfortunately they were not always uniquely realizable. Often each half would contain a large uniquely realizable subgraph along with a few smaller unique subgraphs and maybe some isolated vertices. These various pieces must eventually be combined with an invocation of the global optimizer, and the cost of an optimization depends critically on the number of subgraphs and isolated vertices being combined. When this number is large the optimization problems are difficult. As the results in the next section indicate, the total cost of each of the problems was dominated by the cost of a few large optimization problems generated in this way. In this sense the small vertex was a disappointment. It would be preferable to have an alternate technique for dividing large problems into smaller ones that is more successful at generating a small number of uniquely realizable subproblems.

6.3. The optimization routines. As expected, the global optimization routines dominated ABBIE's running time. This is partially a consequence of the NP–hardness of the molecule problem, but it is also a reflection of the simplicity of the

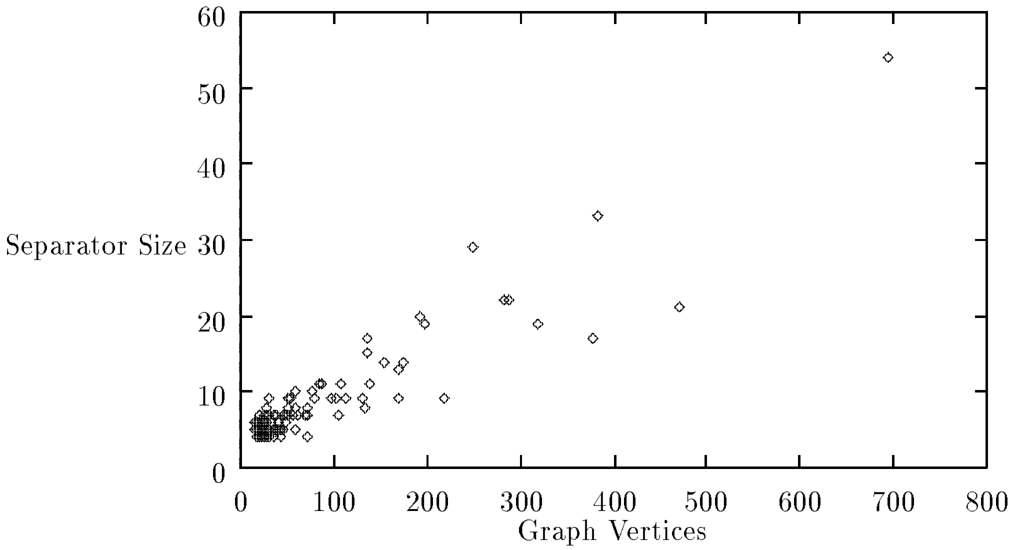

Fig. 6. Separator size as a function of graph size.

optimization routines encoded in ABBIE. A sophisticated optimizer should be able reduce the running time substantially, so the times presented below should be taken as only a rough measure of the relative complexity of the optimization problems.

To determine the coordinates of the vertices ABBIE first employs a combinatoric approach to combine chunks and vertices as described in §5.1. Most of the optimization problems encountered in the set of test problems were completely solved this way. For all our problems the combinatorial operations were extremely efficient relative to the optimizations. For the largest problem all the combinatorial phases consumed a total of less than 50 CPU seconds, while the optimizations required many days.

The optimizer in ABBIE searches for a global minimizer by repeated local minimizations from random starting points. To mitigate concerns of particularly lucky or unlucky sequences of starting points, each of the six problems was run three times. The cost of the decomposition routines remained virtually unchanged, but the optimization time varied by as much as two orders of magnitude, as indicated by Table 3.

Table 3. Total minutes spent in global optimizer.

Amino Acids	Trial 1	Trial 2	Trial 3	Average
20	1.9×10^3	9.0×10^2	4.3×10^2	1.1×10^3
40	4.5×10^4	7.5×10^5	1.2×10^6	6.6×10^5
60	6.6×10^6	2.2×10^6	4.7×10^6	4.5×10^6
80	8.8×10^5	3.6×10^5	3.5×10^3	4.1×10^5
100	1.3×10^5	3.9×10^4	2.8×10^5	1.5×10^5
124	2.5×10^5	1.1×10^5	1.8×10^5	1.8×10^5

As discussed in §5, the cost of an optimization problem should grow as 2^{k-1}, where k is the number of chunks being combined. However, for our test problems we already knew the correct answer, and hence the appropriate parity for each chunk. We exploited this knowledge to reduce the actual computational effort by ensuring that all the parities for each optimization were correct. The resulting running time was then multiplied by 2^{k-1} to approximate a more realistic, unbiased run. The results in Table 3 were generated this way.

The total optimization time presented in Table 3 shows an unexpected dependence on the size of the graph being realized. Except for the smallest problem, the two largest problems are the least expensive ones. This result is especially surprising since we expect larger problems to have to perform more optimizations. This expectation is borne out by the results presented in the second column of Table 4. Clearly, the optimization problems for the 40 and 60 amino acid problems are more difficult than those for the larger problems. We believe this is a consequence of using leading subchains of the protein to generate the intermediate test problems. As remarked above, unlike a full protein, a leading subchain will not generally form a compact structure. With a less dense conformation, there is less geometric data to work with.

Another way to consider the problem complexity is to look at the number of difficult optimization problems. We will consider an optimization problem to be large if it involves at least 25 variables. (Recall that if the vertices are treated individually each of them contributes 3 variables.) Not surprisingly, the number of large optimization problems increases with problem size, as indicated by the last column of Table 4.

Table 4. Number of optimizations.

Amino Acids	Total Optimizations	Large Optimizations
20	2	1
40	7	1
60	15	1
80	21	2
100	22	3
124	32	7

For each of the test problems, the total optimization time is dominated by this subset of large problems; they always consume more than 99% of the total optimization time. A breakdown of these large problems is given in Table 5, in which the number of trials necessary to find the global optimum for the three trials is shown as a function of the number of variables and number of chunks in the optimization problem. The trials always had the parities of the chunks correct, so for a more correct measure of the difficulty of the problems the number of attempts should be multiplied by 2^{k-1}. With this dependence on parities removed, the number of trials should depend solely on the topography of the penalty function. Generally, we would expect the penalty function to become more complex as the number of variables increases, but the experimental data reveals a much more complicated situation. For example, the problems encountered in the test problem with 100 amino acids show exactly the opposite behavior. The optimization with 30 variables is much more difficult than those with 39.

The values in Table 5 reveal why the test problems with 40 and 60 amino acids were so difficult. The optimization problems they encountered were the largest of any in the test set. This led to a large number of trials, each of which involved large Hessians. In addition, these problems involved many chunks which further increased the running time. However, the examples in the table indicate that size alone is a poor predictor of computational difficulty.

Without exception, the large, expensive problems all occurred while trying to combine a large number of chunks and vertices that were created by the small ver-

Table 5. Breakdown of large optimization problems.

Amino Acids	Number of Variables (Chunks)	Number of Starting Attempts			
		Trial 1	Trial 2	Trial 3	Average
20	34 (7)	241	101	55	132
40	46 (8)	1425	17451	25443	14773
60	54 (7)	183092	61871	129258	124740
80	25 (5)	40	387	124	184
80	42 (7)	77470	25085	209	34255
100	39 (8)	6	6	14	9
100	30 (5)	57013	17088	101326	58476
100	39 (6)	1391	147	924	821
124	33 (7)	5395	417	2226	2679
124	39 (8)	30	1960	575	855
124	37 (5)	2745	2213	447	1802
124	28 (5)	632	364	502	499
124	39 (6)	22917	518	12261	11899
124	31 (5)	5	1	2	3
124	39 (10)	238	829	498	522

tex separator. An alternate technique that decomposed a large graph into a small number of uniquely realizable subgraphs would reduce the incidence of such difficult optimizations with a corresponding dramatic improvement in run time.

Having said this, it is still true that the cost of an optimization problem tends to increase sharply as the problem size grows. This justifies the divide–and–conquer idea underlying ABBIE.

7. Conclusions and future work. The divide–and–conquer approach to the molecule problem exemplified by ABBIE shows promise at solving large, practically interesting instances of an NP–hard problem. This technique should work on a large class of instances of the molecule problem. Instances with many extra edges should decompose easily into manageable pieces, while those with very few will quickly be broken into uniquely realizable subgraphs. It is in the intermediate region where the decomposition approach may fail, when there are just enough edges for a unique solution but not enough for subgraphs to be unique.

Our recursive decomposition has several distinct advantages over other approaches to the molecule problem. First, if there is not enough information to uniquely solve the problem (the typical situation in chemical applications) ABBIE will identify and solve unique subproblems. The remaining degrees of freedom in the problem describe the range of solutions that are compatible with the data, and investigating this solution space is now reduced to a much smaller problem. Chemists are often interested in this information for its own sake. The range of solutions may be related to the actual flexibility of the molecule, in which case, the motions identified by ABBIE may have important physical significance.

Second, for many applications it is only a small portion of the molecule that is of interest, like a binding site. Even if there is not enough data to uniquely position the full molecule, ABBIE may be able to solve for the subproblem of interest. ABBIE will automatically identify the portions of the molecule that can be solved uniquely.

Third, the graph algorithms in ABBIE determine whether or not there is sufficient data to solve an instance of the problem. This knowledge can be used to direct further experiments. In this way, poorly posed problems can be readily identified and avoided.

Fourth is the problem of inconsistent data. In any physical experiment there can be some measurements that are in error. This is a difficult problem for all of the approaches to the molecule problem, and they typically find a solution that nearly, but not exactly, satisfies all the constraints. If there are a few bad values that are causing the confusion, identifying them would be extremely useful as they could then be discarded. The only previous techniques for identifying bad data involved repeated attempts to solve the full problem, each time discarding a few edges. If one of the runs produced an acceptable answer then a discarded edge must have been causing the confusion. Our recursive decomposition has the potential to simplify this task. Inconsistent data would be indicated by the inability to solve a particular subproblem, narrowing the location of the erroneous data to the values in this subproblem.

In addition to inconsistent data, we would like to be able to deal with the realistic problem in which distances are not known exactly. Much of the theory about unique realizations will no longer apply in the presence of measurement uncertainty. We believe that the underlying ideas will still be applicable in this case, but in a more heuristic way. For instance, a graph that violates the necessary conditions for uniqueness will still have multiple realizations in the presence of data uncertainty, which can still permit the decomposition into smaller subproblems. However, uniqueness is now harder (and probably impossible) to guarantee. But as long as the range of satisfying conformations of a subgraph remains relatively small, the decomposition approach is still appropriate. Instead of treating a solution to a subproblem as a rigid body, simply use it as a starting conformation for the subset of vertices, allowing their relative locations to change as the optimization proceeds. If the solution from the subproblem is near to the correct solution, then this should provide a good starting point for the succeeding optimization. This should significantly reduce the optimization effort. By treating the solution of subproblems as intelligent starting points for later optimizations, the overall difficulty of the problem should be reduced.

The algorithms in ABBIE could be improved in a number of ways. One of the asymptotically faster four–connectivity algorithms could be used, and sparse matrix algorithms could be used in the redundant rigidity calculations. Much greater savings could be realized by improving the optimization phase, by far the most time consuming portion of the code. The number of optimization variables could be reduced using more sophisticated combinatorial heuristics than those described in §5.1. A more sophisticated global optimization routine could be employed, like the stochastic technique of Rinnooy Kan and Timmer [27]. Stochastic techniques also have the advantage of being easy to parallelize [4]. Additional possible approaches to the global optimization problem would be the tunneling algorithm of Levy and Montalvo [21], or a simulated annealing approach [30]. In addition, there are various optimization tools that could improve the performance of the local optimizations. A quasi–Newton approach could be used so that instead of refactoring the Hessian matrix at each step, the factorization would be approximated and updated in linear time. Also, sparsity within the Hessians could be exploited.

An alternate way to substantially reduce the cost of the optimizations would be to reconsider the way in which uniquely realizable subgraphs are decomposed into smaller problems. As mentioned in §6, the cost of the optimizations was dominated by problems involving many subgraphs. These problems were induced by the proliferation of smaller subgraphs generated by ABBIE's vertex separator algorithm. An alternate method that identified small uniquely realizable subgraphs more directly could have a dramatic impact on runtime.

More generally, we believe the basic ideas described in this paper have applicability beyond the molecule problem. Divide–and–conquer techniques have not been commonly used in optimization, primarily because it is difficult to figure out how they can be applied. There are three aspects to the molecule problem that make a recursive decomposition possible. First, the penalty function describing an instance of the problem expresses equality constraints, since each edge must achieve a specific distance. Second, the penalty function consists of a sum of simple subfunctions, each relating only a small number of variables; that is, it is partially separable. This allows for the identification of subproblems that completely contain a set of constraints. If instead, the subfunctions coupled many variables then it would be difficult to decompose the problem. Third, there is a deep combinatorial structure to the molecule problem that allows solvable subproblems to be identified. The first two of these properties are fairly common in optimization settings. Although the specific structure we have exploited is unique to the molecule problem, it is likely that other optimization problems have analogous structure that can be similarly utilized. Any problem that contains subproblems which can be solved independently should be amenable to the type of divide–and–conquer approach described here. The challenge is to identify this structure.

Acknowledgements. The ideas in this paper have been developed and refined in innumerable discussions with Tom Coleman and Bob Connelly. I am also indebted to Kate Palmer for many helpful conversations.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] L. Asimow and B. Roth, *The rigidity of graphs*, Trans. Amer. Math. Soc., 245 (1978), pp. 279–289.

[3] ———, *The rigidity of graphs, II*, J. Math. Anal. Appl., 68 (1979), pp. 171–190.

[4] R. H. Byrd, C. L. Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel, *Concurrent stochastic methods for global optimization*, Math. Programming, 46 (1990), pp. 1–30.

[5] J. Cheriyan and R. Thurimella, *On determining vertex connectivity*, Tech. Rep. UMIACS-TR-90-79, CS-TR-2485, Dept. of Computer Science, University of Maryland at College Park, 1990.

[6] R. Connelly, *Personal communication*, April 1989.

[7] ———, *Personal communication*, September 1989.

[8] ———, *On generic global rigidity*, in Applied Geometry and Discrete Mathematics, the Victor Klee Festschrift, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 4, P. Gritzmann and B. Sturmfels, eds., AMS and ACM, 1991, pp. 147–155.

[9] H. Crapo, *Structural rigidity*, Topologie Structurale, 1 (1979), pp. 26–45.

[10] G. M. Crippen and T. F. Havel, *Distance Geometry and Molecular Conformation*, Research Studies Press Ltd., Taunton, Somerset, England, 1988.

[11] H. N. Gabow and H. H. Westermann, *Forests, frames and games: Algorithms for matroid sums and applications*, in Proc. 20th Annual Symposium on the Theory of Computing, Chicago, 1988, pp. 407–421.

[12] L. M. Gierasch and J. King, eds., *Protein folding: deciphering the second half of the genetic code*, AAAS, 1989.

[13] H. Gluck, *Almost all simply connected closed surfaces are rigid*, in Geometric Topology, Lecture Notes in Mathematics No. 438, Springer-Verlag, Berlin, 1975, pp. 225–239.

[14] B. Hendrickson, *The Molecule Problem: Determining Conformation from Pairwise Distances*, PhD thesis, Cornell University, Dept. of Computer Science, Ithaca, NY, 1990. Technical Report 90-1159.

[15] ———, *Conditions for unique graph realizations*, SIAM J. Comput., 21 (1992), pp. 65–84.

[16] J. E. Hopcroft and R. E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.

[17] H. Imai, *On combinatorial structures of line drawings of polyhedra*, Discr. Appl. Math., 10 (1985), pp. 79–92.

[18] A. Kanevsky and V. Ramachandran, *Improved algorithms for graph four-connectivity*, in Proc. 28th IEEE Annual Symposium on Foundations of Computer Science, Los Angeles, October 1987, pp. 252–259.

[19] K. Killian and P. Meissl, *Einige grundaufgaben der räumlichen trilateration und ihre gefährlichen örter*, Deutsche Geodätische Komm. Bayer. Akad. Wiss., A61 (1969), pp. 65–72.

[20] G. Laman, *On graphs and rigidity of plane skeletal structures*, J. Eng. Math., 4 (1970), pp. 331–340.

[21] A. V. Levy and A. Montalvo, *The tunneling algorithms for the global minimizer of functions*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 15–29.

[22] J. W. H. Liu, *A graph partitioning algorithm by node separators*, ACM Trans. Math. Software, 15 (1989), pp. 198–219.

[23] G. L. Miller, S.-H. Teng, and S. A. Vavasis, *A unified geometric approach to graph separators*, in Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, October 1991, pp. 538–547.

[24] J. Moré and D. Sorensen, *Computing a trust region step*, SIAM J. Sci. Stat. Comput., 4 (1983), pp. 553–572.

[25] K. A. Palmer, *Personal communication*, March 1990. Chemistry Department, Cornell University.

[26] K. A. Palmer and H. A. Scheraga, *Standard–geometry chains fitted to X–ray derived structures: Validation of the rigid–geometry approximation. ii. systematic searches for short loops in proteins: applications to bovine pancreatic ribonuclease A and human lysozyme*, J. Comp. Chem., 13 (1992), pp. 329–350.

[27] A. H. G. Rinnooy Kan and G. T. Timmer, *A stochastic approach to global optimization*, in Numerical Optimization, P. Boggs, R. Byrd, and R. B. Schnabel, eds., SIAM, Philadelphia, 1984, pp. 245–262.

[28] B. Roth, *Rigid and flexible frameworks*, Amer. Math. Monthly, 88 (1981), pp. 6–21.

[29] J. B. Saxe, *Embeddability of weighted graphs in k-space is strongly NP-hard*, in Proc. 17th Allerton Conference in Communications, Control and Computing, 1979, pp. 480–489.

[30] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, Boston, MA, 1987.

[31] W. Wunderlich, *Untersuchungen zu einem trilaterations problem mit komplaneren standpunkten*, Sitz. Osten. Akad. Wiss., 186 (1977), pp. 263–280.

[32] K. Wütrich, *The development of nuclear magnetic resonance spectroscopy as a technique for protein structure determination*, Accounts of Chemical Research, 22 (1989), pp. 36–44.

[33] ———, *Protein structure determination in solution by nuclear magnetic resonance spectroscopy*, Science, 243 (1989), pp. 45–50.