The Molecule Problem: Determining Conformation from Pairwise Distances

Bruce A. Hendrickson
Ph.D Thesis

90-1159
September 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

THE MOLECULE PROBLEM:

DETERMINING CONFORMATION

FROM PAIRWISE DISTANCES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Bruce Alan Hendrickson

January 1991

THE MOLECULE PROBLEM: DETERMINING CONFORMATION FROM PAIRWISE DISTANCES

Bruce Alan Hendrickson, Ph.D.
Cornell University 1991

The *molecule problem* is that of determining the coordinates of a set of points in space from a (usually sparse) set of pairwise distance measurements. As its name implies, it has applications in the determination of molecular conformation. Unfortunately, the molecule problem is NP-hard.

We present an approach to the molecule problem that uses a very specialized divide-and-conquer technique. Instead of solving a single large problem we try to solve a sequence of smaller, presumably easier ones. These smaller problems consist of subsets of points whose relative locations can be determined uniquely. Once such a subset is positioned, its points can collectively be treated as a rigid body. This can greatly reduce the number of degrees of freedom in the problem.

Identifying subsets of points whose relative locations can be uniquely determined requires exploiting some very special structure inherent in the molecule problem. We reduce this identification to a purely combinatoric characterization that ignores the actual distances. We develop necessary graph theoretic conditions for a set of points to have a unique solution, along with efficient algorithms to find subgraphs with these properties. These characterizations and algorithms combine ideas from matching theory, differential topology and matrix computations.

These ideas have been implemented in ABBIE, a program to solve three-dimensional instances of the molecule problem. ABBIE combines the recursive decomposition described above with a nonlinear global optimizer to perform the coordinate determinations. Details of this implementation are described, and numerical results of simulated chemical data are presented.

Biographical Sketch

Although the early years of the life of Bruce Hendrickson are shrouded in mystery, some of the more bizarre tales of his origins have been largely put to rest by recent biographers. The first reported sightings of him were in the early 1960s in Oak Park, Illinois, a town Ernest Hemingway once described as having wide lawns and narrow minds. The available information on this period, such as it is, indicates the subject to be a relatively normal child. There is nothing in the record to anticipate his later activities.

The first well documented period began in 1978 at Brown University in Providence, Rhode Island. Acquaintances there seem hesitant to come forward, although it is not clear whether this is out of embarrassment, or merely an indication that the subject was rather unmemorable. Typical comments, culled anonymously from the files include the following:

"He seemed like such a normal guy; not the kind of person you would expect to end up this way."

"He was kind of a loner; kept pretty much to himself. I figured he'd become a disgruntled postal employee."

"Who?"

Bruce received an AB in mathematics and a ScM in physics from Brown in 1982.

After college the trail disappears for several years in the earthly pleasures of Southern California. Documentation on this period is extremely sketchy, but some observers, particularly those with Freudian leanings, place a heavy emphasis on these years.

The subject resurfaced in the mid 1980s in Ithaca, New York where he received his MS in computer science in 1987, followed eventually by a PhD in 1991. His current whereabouts are unknown, but recurrent rumors place him in the Southwest, somewhere in the vicinity of Sandia National Laboratories in Albuquerque.

Acknowledgements

This thesis owes an enormous debt to two individuals, Tom Coleman and Bob Connelly. Their wisdom and judgement permeate this thesis. Tom nurtured this work in its infancy, encouraging me to follow the problem wherever it led. Bob helped lead me through the minefields of rigidity theory and differential topology with infinite patience and enthusiasm.

My other committee members, Steve Vavasis and David Caughey, assisted me in many different ways at various times. I am also indebted to John Gilbert who continued to be supportive long after leaving Cornell. Jim Renegar helped me learn differential topology, in the process teaching me about the right way to do mathematics. Maria Terrell always seemed to find the flaws in all my succession of flawed proofs. Kate Palmer's contribution of data and a chemist's judgement were crucial to this thesis. The Hertz Foundation was extremely generous with support, both financial and otherwise.

For the years I have spent at Cornell I have been richly rewarded with friends. In rough order of appearance they included Andy Meltzer, Carolyn Turbyfil, Michael Schwartzbach, Sharon Himmelstein, Jennifer Colford, Hjalli Hafsteinsson, Ken Kane, Kim Taylor, Mike Quayle, Michael Slifker, Charlotte Lewis, Alex Panconesi and Dave Pearson. It is they who made the years spent here worth spending.

Table of Contents

List of Figures

List of Tables

Part I

The Molecule Problem

Chapter 1

Introduction

Imagine a set of objects at unknown locations. We would like to determine the locations of the objects, but all we know is some of the pairwise distances between them. How can we use this information to compute their positions? This is the *molecule problem.* As its name implies, it has applications in chemistry, where one wishes to determine the three-dimensional conformation of a molecule. It is possible to analyze the nuclear magnetic resonance spectrum of a molecule to obtain pairwise inter-atomic distance information [CH88,Wüt89b,Wüt89a]. Solving the molecule problem in this context would determine the shape of the molecule, of crucial importance in understanding the molecule's properties. Additional important applications of the molecule problem occur in surveying [Kol78]. For instance, a set of laser rangefinders may be distributed around a geologically active area in such a way that each detector can measure the distance to a few of its neighbors. A solution to the molecule problem in this setting would provide information about the movement of the terrain. Further applications occur in satellite ranging [KM69, Wun77a,Wun77b,Wun77c,Wun79].

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 points or atoms, and an edge $e \in E$ connects two vertices if the distance between the corresponding points is known. We will denote the number of vertices by n and edges by m. 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*). A realization is *satisfying* if all the distance constraints are obeyed. The combination of a graph and a realization is called a *framework*, denoted by $p(G)$.

The molecule problem does not conveniently lie within a single branch of mathematics. There are many different ways to approach it, each providing different insights. Perhaps the most natural way to think of the problem is in terms of geometry. If the distance between two points is known then the allowed positions for the second vertex define a sphere around a fixed location of the first. Solving the molecule problem corresponds to positioning all the points on the correct in-

tersections of spheres. This approach is useful for the intuition it provides, but it does not seem to generate practical computational techniques.

A second way to attack the problem is to use tools from continuous optimization. A cost function can be constructed that penalizes a realization for violating edge length constraints. A global minimizer of this function is a realization that satisfies all the constraints. There are many techniques for minimizing functions of this type, and most previous attempts to solve the molecule problem have used some variant of an optimization approach. The approach we are proposing also involves optimization, but in a very specialized way.

Another way to formulate the molecule problem is in terms of functional analysis. If the points are positioned in d dimensions, there is a simple *edge function* $f : \mathbb{R}^{nd} \rightarrow \mathbb{R}^m$ that maps from vertex locations to edge lengths. The molecule problem is really a calculation of the inverse of this function. (Of course, this inverse is multivalued since any rigid translation, rotation or reflection of a realization yields another set of coordinates with the same set of edge lengths.) We will use this formulation to investigate the set of possible solutions to an instance of the molecule problem. The solution set will turn out to have some very nice structure that can be analyzed with techniques from differential topology.

Still another formulation is as a set of algebraic equations. Each distance constraint defines a quadratic equation in the coordinates of the vertices. A solution to this coupled set of equations is a satisfying realization. Solving large numbers of quadratic equations directly is computationally infeasible, but we will use this approach to investigate the computational complexity of the problem.

There is an additional way in which the molecule problem refuses to be categorized into a particular branch of mathematics. It is inherently a continuous problem, as the constraints are real numbers and the vertices can be moved continuously in Euclidean space. However, it also has a rich combinatoric structure that is described by the underlying graph. The existence or absence of a particular edge length constraint is a discrete event. Remarkably, a great deal of insight into the molecule problem can be gained by ignoring the values of the edge lengths and instead concentrating solely on the underlying graph. This analysis will require tools from discrete mathematics.

We will combine techniques from various branches of discrete and continuous mathematics to gain a deeper insight into the problem than any of the approaches could provide in isolation.

The molecule problem is hard, in both an informal and a formal sense. More precisely, as we will see in Chapter 2, the one-dimensional version of the problem is NP-complete, and in higher dimensions it is NP-hard. Hence, there is unlikely to be a general propose algorithm to solve all instances of the problem efficiently. This implies that when expressed as an optimization problem, the number of local minimizers may grow exponentially with the problem size. For large problems this can result in prohibitive amounts of computation.

To mitigate this problem we propose an approach that tries to avoid having to solve large optimization problems. Our approach can be thought of as a divide-and-conquer scheme. Divide-and-conquer techniques appear in many different algorithmic settings throughout computer science [AHU74]. Their common idea is that a large problem is broken into smaller pieces, the pieces are solved separately, possibly recursively, and their answers are recombined into a solution for the whole. There are three distinct steps in this process. First is the division of a large problem into pieces. For many problems, like sorting, this step is trivial, but for the molecule problem it will prove to be quite difficult. The investigation of how to properly decompose a problem will be one of the main elements of this dissertation. The second step is the solution of the subproblems. This typically proceeds by recursively subdividing until the subproblems are acceptably small and can be solved directly. Lastly, the solutions to the pieces must be combined into a full solution. In our case, both the small problem solution and the recombination will involve a nonlinear global optimization.

The fundamental observation that allows us to use a divide-and-conquer approach to the molecule problem is that within an instance of the 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 edge constraints. Once this subproblem is solved, the entire subgraph can be treated as a rigid body. In three-space a rigid body has only six continuous degrees of freedom, but considered independently each vertex would have three. So by treating a set of vertices collectively as a rigid body, the number of variables in the problem can be greatly reduced, making the optimization problem tractable. Using this approach a large optimization problem can be reduced to a sequence of smaller ones. Since the computational cost of a problem can grow exponentially with the problem size, this can lead to a substantial reduction in overall computational effort.

Four our divide-and-conquer approach to the molecule problem the divide step is the most complicated. It consists of two phases. First, maximal solvable subproblems are identified. This identification is a difficult problem and will be the focus of the second part of this dissertation. This identification may decompose a graph into uniquely realizable subgraphs. If these subproblems are small enough they can be solved directly. Otherwise they must be further divided and solved by a recursive call to the algorithm. This further division constitutes the second second phase.

The second and third steps in a divide-and-conquer technique involve the solution of small subproblems and their recombination into a solution for the larger problem. Solving an instance of the molecule problem involves the assignment of coordinates to the vertices in such a way that all the distance constraints are satisfied. For both steps this involves fitting together a collection rigid bodies and single points to obey a set of distance constraints. Our approach to each of these steps

mixes combinatoric techniques with a nonlinear global optimization to determine the coordinates.

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 involving only a small number of variables. 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 very deep combinatoric structure to the molecule problem that allows solvable subproblems to be identified. We will have much more to say about this structure later. Whereas the first two properties are fairly common in optimization settings this third aspect is very special.

This dissertation is divided into three parts. In the next chapter we will discuss the computational complexity of the molecule problem and several closely related problems. The results of this analysis will justify our desire to avoid solving large problems with a single optimization. We will also briefly describe some of the approaches that other researchers, mainly chemists, have taken to solving the problem.

The second part of this dissertation will focus on a critical problem that is inherent to our recursive decomposition approach. If we wish to isolate a subgraph we must be certain that the answer we get by solving the subproblem is consistent with the answer we would get by solving the full problem. That is, we must be certain that the subgraph has a unique realization (modulo isometries of the entire space). It turns out that this question can be approached in a manner that ignores the edge lengths and focuses entirely on the underlying graph. This part will involve a combinatoric analysis of when a graph has a unique realization. Although we will not be completely characterize uniquely realizable graphs, we will develop three necessary conditions in Chapters 4, 5 and 6 and a sufficient condition in Chapter 7. Although neither the necessary conditions nor the sufficient one are ideally suited to our purposes, by combining them we will develop an efficient technique for identifying uniquely realizable subgraphs.

In Part III we will describe in some detail, ABBIE, an implementation of this approach to the molecule problem. This program combines the graph decomposition techniques developed in Part II, with a nonlinear global optimizer to actually calculate coordinates. Chapter 9 contains a description of the implementation of the unique realizability algorithms from Part II. The following chapter discusses the optimization techniques in ABBIE, which combine combinatoric and continuous elements. Features of the program which fail to conveniently fall into these two chapters are described in Chapter 11. ABBIE has been used to analyze simulated

molecular data, and our computational experiences are discussed in Chapter 12. Finally, conclusions and open problems are considered in Chapter 13.

Chapter 2

Background

2.1 Complexity

There are two aspects to the molecule problem that make it difficult to analyze using standard techniques from complexity theory. First, traditional computational complexity theory was formulated to deal with decision problems, those that have a yes or no answer. However, a solution to the molecule problem is a set of vertex coordinates. To apply tools from computational complexity we must first rephrase our problem. For instance, we could ask whether a particular instance of the molecule problem has a solution, or whether a particular solution is the only one. The complexity of these decision problems will be discussed below. Note that finding a solution to the molecule problem would resolve the decision problem of whether or not one exists.

The second drawback of the molecule problem is that in any dimension larger than one it is likely to involve non-integer values. The distance function is a square root, which generates algebraic numbers. Thus, there are bound to be numerical errors due to the finite ability of a computer to represent irrational values. From a complexity theoretic viewpoint there are several ways to deal with this problem. First, and easiest, we could use an unrealistic model of computation in which algebraic numbers are primitive data types and arithmetic operations on them (including square roots) require constant time. Alternately, we could require that all the vertices have rational coordinates and we could only work with the square of edge lengths, which would also be rational. Then complexity bounds could be developed that depend on the number of bits required to represent the rational data. A third, perhaps more intuitive approach, is to assume we have arbitrary precision in the data and in the numerical operations, but to not require distance constraints to be satisfied exactly. Instead, we can enquire as to the complexity of finding vertex coordinates at which all the distance constraints are satisfied within a small tolerance ϵ. These last two models are closely related, with the number of bits in ϵ from the third approach being related to the number of bits required to

store values in the second approach.

We will first address the decision problem of whether or not a solution to the molecule problem exists. To avoid the problems with real numbers we will pose the problem in one dimension, and require that all distances be integers. Consider the graph consisting of a simple cycle depicted in Figure 2.1. A one-dimensional solution to the molecule problem corresponds to an assignment of points in the line for each of the vertices of the graph. Since we have integer edge lengths the vertices can be assigned integer locations.

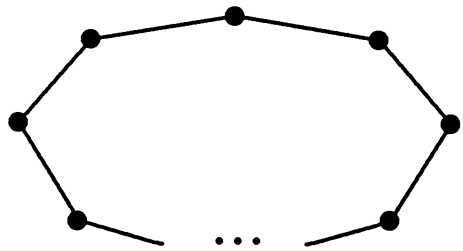


Figure 2.1: A cycle graph for the one-dimensional molecule problem.

Assume we have an algorithm that can solve this problem, mapping vertices to integers in the line. Once the graph is correctly realized, imagine walking along the edges of the graph. Some of the time you will be walking to the left, and some to the right. Since the graph is a cycle, you must end up back where you started. Thus the edge lengths get divided into two sets of equal sum, the left-walking edges and the right-walking edges. However, dividing a set of integers into two sets of equal sum is the *partition problem*, a classic NP-complete problem. (For a review of NP-completeness the reader is referred to the excellent book by Garey and Johnson [GJ79].) So if we can solve the one-dimensional version of the molecule problem we can also solve the partition problem, which is known to be difficult. This demonstrates that the molecule problem is NP-hard. In one dimension a nondeterministic Turing machine need only guess which direction each edge should point, indicating that the problem with integer edge lengths is in NP. This gives us the following theorem.

Theorem 2.1.1 *In one dimension, with integer edge lengths, deciding whether an instance of the molecule problem has a solution is NP-complete.*

In higher dimensions we must deal with the problem of non-integer distances. For now, make the generous assumption that we can work with arbitrarily precise data. To analyze the complexity of the two-dimensional molecule problem consider the graph fragment depicted in Figure 2.2.

By joining the crossed rectangles into a cycle we reduce the two-dimensional problem to the one-dimensional example that was NP-hard. Saxe has shown that

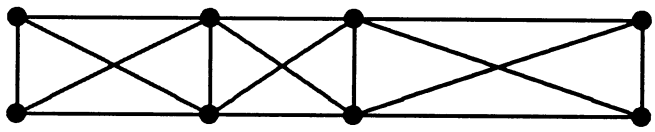


Figure 2.2: A framework for the two-dimensional molecule problem.

this trick can be performed while using only integer edge lengths. A similar construction will work in higher dimensions. This gives the following result.

Theorem 2.1.2 *In two and higher dimensions, deciding whether an instance of the molecule problem has a solution is NP-hard.*

Following Garey and Johnson we define a problem to be *strongly* NP-hard (or *strongly* NP-complete) if it remains NP-hard (or NP-complete) when the lengths of numerical values are bounded by a polynomial in the problem size. Saxe has shown the above complexity bounds to hold even if edge lengths are limited to the values 1 and 2 [Sax79]. This gives the following stronger theorem.

Theorem 2.1.3 *Even if edge lengths are integers, deciding whether an instance of the molecule problem has a solution is strongly NP-complete in one dimension and strongly NP-hard in higher dimensions.*

Saxe also considered the complexity of two closely related problems. First was the problem of *approximate realizability.* For an instance of the molecule problem can the vertices be positioned so that all the edge lengths are satisfied within some tolerance? This formulation is an attempt to deal with the problem of real numbers instead of integers. Saxe showed that this problem is just as hard as the original problem, strongly NP-complete in one dimension and strongly NP-hard in higher dimensions.

The second related problem was that of *ambiguous realizability.* Given an instance of the molecule problem and a solution, is there a second solution that is not congruent to the first. This problem was also shown to be strongly NP-complete in one dimension and strongly NP-hard in higher dimensions. However, it should be noted that this proof required a very special degeneracy in the locations of the vertices.

These results are extremely discouraging. They imply that a general purpose, polynomial time algorithm for the molecule problem is unlikely to be found. Any approach is likely to require more than a polynomial amount of time on some problems.

An upper bound on the cost of the computation can be deduced from the algebraic formulation of the problem. A decision problem is said to be in PSPACE if it can be decided by a Turing machine that requires only a polynomial amount of

space. It is easy to show that such a calculation can involve at most an exponential amount of time. We can assume infinite precision arithmetic on real values in constant time, with the distance constraints expressed as equalities. Alternately, we can consider a model with finite precision in which the constraints become two inequalities expressing upper and lower bounds. In either case, it is well known that the ideas developed by Canny lead to a PSPACE algorithm for solving sets of polynomial equations [Can86]. Applied to our algebraic formulation of the molecule problem this gives the following result.

Theorem 2.1.4 *In any dimension, the molecule problem is in PSPACE.*

2.2 Previous Work

The complexity results from the previous section are daunting. No matter how the molecule problem is reformulated, one is unlikely to find a general, polynomial time algorithm to solve it. In particular, when expressed as a nonlinear optimization problem there can be an exponential number of local minimizers. Chemists have observed this in practice and refer to it as the multiple minima problem.

Various approaches to the molecule problem have been developed, primarily by chemists. Most of these techniques use some optimization formulation of the problem, usually incorporating specific insight into the structure of the molecules they were designed to analyze. However, because of the NP-hardness of the problem it must be emphasized that none of the techniques will work well on all problems.

Real experimental data cannot be precisely accurate. Instead, the experimental data give upper and lower bounds for distances. For some applications of the problem, like surveying, these bounds can be very tight, but for problems of molecular conformation the uncertainties can be a significant fraction of the actual distances. Phrasing the problem in terms of upper and lower bounds on distances is a generalization of the molecule problem as we defined it in Chapter 1. Some of the previous approaches to the problem deal with this uncertainty in an explicit way, but most do not. Instead, they merely search for some solution in which the constrained distances fall between the bounds. Our presentation of some of these techniques is based upon Chapter 6.4 of the book by Crippen and Havel [CH88].

Before describing some of the previous approaches in detail, it will be helpful to provide a very limited introduction to the geometry and chemistry of proteins. For a more detailed discussion the reader is referred to any good physical chemistry text, like [CS80]. Several of the techniques to be described exploit knowledge of protein geometry to improve their performance. ABBIE, the implementation of the approach described in this dissertation, is quite general, and makes no a priori assumptions about the solution it is trying to compute.

A *protein* is a molecule composed of a linear chain of smaller units known as *amino acids*. There are about 20 common amino acids and their chemical and

geometric structures are well known. There exist efficient techniques for sequencing proteins, determining the order of the constituent amino acids. However, the conformation of proteins is very difficult to determine. Many thousands of proteins have been sequenced, but only a few hundred three-dimensional structures have been described. Chemical activity depends critically on geometry, so the conformation problem is a serious bottleneck in the understanding and development of new molecules. Nature manages to find the appropriate geometrical structure purely from the underlying chemistry, but mankind has not yet been so successful. Attempts to compute the conformation directly from the sequence have so far met with very limited success.

The conformation of a protein is traditionally described on three scales. The *primary* structure simply consists of a list of the amino acids in the order in which they occur in the protein. This is sufficient to understand only very local geometry. The protein can twist and fold into very complicated shapes that can not currently be predicted from the sequence alone. Several patterns seem commonly to occur within the complicated structure, including α-helices and coils. The identification of these intermediate sized patterns is known as the *secondary* structure. The *tertiary* structure describes how these pieces combine to comprise the full protein.

Historically, the most popular technique for determining the conformation of molecules has been x-ray crystallography. To perform this experiment a quantity of the molecule is purified and crystallized. The crystal is bombarded with x-rays and the diffraction pattern is recorded. Without monochromatic x-rays it is difficult to unambiguously interpret the results, but various clever techniques have been developed that work well in practice. X-ray crystallography has several shortcomings that limit its usefulness. First, it can be extremely difficult to understand a diffraction pattern, requiring many experiments and extensive computational effort. This is particularly true for large molecules. Second, and more problematic, many molecules are difficult to crystallize, and even if they do it is not clear that they have the same shape in a crystal that they do in solution. This is a particular concern for biological molecules as their activity typically occurs in solution. A related problem is that the close packing of molecules in a crystal means that atoms on the outside of each molecule may be distorted by their crystalline neighbors. This is a potentially serious problem for molecules with many of their atoms on the surface, like DNA.

Partly in response to these shortcomings an alternate approach to determining molecular conformation has been proposed [Wüt89b, Wüt89a]. The idea is to determine as many constraints on the geometry as possible and from these to compute the conformation. The most important constraints are distances between atoms. Distances can be determined in several ways. Atoms that are bonded together have a well defined bond length, typically 1–2 Å. This provides the bulk of the distance information, but not enough to determine secondary and tertiary structure. Additional distance measurements come from two-dimensional nuclear

magnetic resonance experiments, particularly those that measure the nuclear Overhauser enhancement. These experiments are performed in solution, and can detect spin-spin coupling of hydrogen atoms, which can only occur when the atoms are physically close together, typically less that 5 Å. This allows the detection of pairs of hydrogens that are geometrically near each other even if they are far apart in the sequence. This reveals a great deal about how the molecule is folded.

There are many additional sources of geometric constraints that chemists can make use of including bond angles, chiralities, and hard sphere radii. ABBIE is not specialized to use any of this information, just pairwise distances.

Chemists have long been interested in simulating the dynamics of molecules using energy considerations. A natural way to determine conformations is to include the geometric constraints as additional terms in the energy minimization equations. If violating a constraint imposes a heavy penalty then the simulation should prefer configurations that satisfy the constraints. It is difficult to perform these calculations in practice because of the excessive number of local minimizers. As we saw in the previous section the distance constraints alone can generate an exponential number of local minimizers. When coupled with energy functions the problem only gets worse. However, when this is combined with a technique to escape from non-optimal minima the problem is somewhat mitigated.

Simulated annealing is a popular global optimization technique for discrete problems which attempts to avoid local minima [vA87]. It has been applied to the problem of minimizing the combined energy and constraint function described above with some success [CBKG86,dVBS+86]. This approach works best when a relatively good starting configuration is available, and it can be used to improve the solutions generated by other approaches. One drawback is that with such a complicated penalty function the calculation of derivatives can be expensive.

Another optimization approach that tries to avoid local minima is the ellipsoid algorithm, initially developed for convex feasibility problems [BGT81,EK85]. The distance constraints in the molecule problem are not convex, but the ellipsoid technique can still be used as a heuristic [BHW86]. The ellipsoid algorithm becomes more difficult to apply as a heuristic as the number of variables increases. For this reason the authors chose to treat the amino acids as rigid bodies and to let the variables be the dihedral angles about bonds between adjacent amino acids. This reduces the number of variables while increasing the complexity of the function to be optimized. Unfortunately, this approach becomes infeasible as the number of dihedral angles becomes large.

The EMBED algorithm due to Crippen and Havel is unusual in several respects [CH88,HKC83,HW84]. For one thing, it makes an attempt to sample the range of conformations consistent with the distance bounds, instead of just finding a single solution. It begins by shrinking the intervals defined by the distance bounds by enforcing triangle inequalities [DH87]. This idea is applicable to all approaches to the molecule problem. In the process, the algorithm identifies upper and lower

bounds for all pairwise distances. Next, a random distance is selected from the allowed interval for each pair of atoms. These distances are entered into an $n \times n$ distance matrix.

This set of distances will not generally correspond to a three-dimensional conformation. The algorithm proceeds by looking for a new distance matrix that does come from a three-dimensional solution. This new matrix is required to be optimally close to the old one in a particular norm. This calculation involves the determination of the three largest eigenvalues of the old distance matrix and their corresponding eigenvectors. The three-dimensional conformation identified in this way will not generally satisfy all the distance constraints, so it is locally improved by a continuous optimization.

One shortcoming of this approach is that it needs to assign distance bounds to all pairs of atoms, even those with no direct experimental values. This has two drawbacks. First, these induced bounds are unlikely to be very tight, but they are treated identically with the directly measured bounds, which de-emphasizes the true experimental data. Second, although the data is sparse, typically with $O(n)$ distance constraints, this technique extends the bounds to all n^2 pairs, requiring substantial additional computer memory.

Fundamentally, the EMBED algorithm is a technique to generate realizations that are fairly good at satisfying the distance constraints. Thus, it can be used as a starting procedure for many of the other approaches. Unfortunately, it is an expensive technique because of the need to work with $O(n^2)$ data values and compute eigenvectors of a dense $n \times n$ matrix.

An alternate approach has been proposed that relies upon heuristic techniques from artificial intelligence [LCBJ87]. The central feature of this approach is an attempt to determine a family of structures which broadly approximate the topology of the protein. These structures can then be refined using any of the other techniques. An unusual aspect of this approach is that it does not involve the minimization of a penalty function. Instead, a discretized version of the parameter space is searched for consistency with the geometric constraints. Secondary structures are presented as input and treated as rigid bodies. The algorithm grows the protein by adding new secondary structures one at a time, retaining the set of conformations that do not violate any constraints.

This approach has the potential to sample more of the conformation space than other techniques. However, its performance requires that many secondary structures have been accurately identified. Thus it is a technique to determine approximate tertiary structure from secondary structure, but not directly from the distance constraints.

One very specialized technique is the *variable target method* due to Braun and Go [BG85,Bra87]. Like the ellipsoid algorithm discussed above, the variable target method uses dihedral angles as variables. The key idea behind this method is to vary the penalty function as the optimization proceeds. The distance constraints

are divided into *short range*, those which occur among atoms that are near in sequence, and *long range*, the remainder. By first performing an optimization involving only the short range constraints the method tries to identify secondary structure. The long range constraints are gradually added in to force the tertiary structure to be correct. In effect this heuristic determines secondary structure first, and uses it as a starting conformation for finding tertiary structure. This approach seems to work well when there are many accurate short range constraints. Unfortunately, it is dependent upon the special structure of proteins and is unlikely to generalize.

A very different approach to solving the two-dimensional version of the molecule problem was proposed by Yemini [Yem79]. His method begins by looking for small subgraphs, like triangles, that are known to have unique conformations. These subgraphs are then merged together using various *welders*, heuristics to combine them into larger uniquely realizable structures.

This approach is similar in philosophy to the technique we are proposing in this dissertation. But whereas Yemini's algorithm works in a bottom-up fashion, ours proceeds top-down. For a robust bottom-up approach one would need a complete set of uniquely realizable subgraphs and welders. Yemini was unable to characterize such a set in two dimensions and the problem is considerably harder in three. There is one section of this dissertation in which small subgraphs are combined into larger uniquely realizable pieces using heuristics like Yemini's welders. This occurs in Section 10.1 in an attempt to reduce the size of the optimization problems.

A fundamental problem for most of these approaches is that the solution space that needs to be examined can be quite large. If each of the n atoms is allowed to move independently then the space of all possible, noncongruent conformations has dimension $3n - 6$. Molecules can have many thousands of atoms, making this solution space enormous. Using dihedral angles between amino acids reduces the number of variables, but greatly complicates the calculation of distances. The approach we are proposing in this dissertation avoids this problem in a different way. Instead of examining the entire parameter space at once we will only consider portions of it at a time. These subspaces will be collapsed, reducing the dimension of the full problem. Assume there is a subset of n' vertices whose relative locations can be determined. Treating these vertices individually they have $3n'$ degrees of freedom, but by solving for their relative locations and treating the solution as a rigid body we can reduce the number of variables to 6. Instead of searching for a solution in a large space we will perform a series of searches in smaller spaces.

Part II

Unique Realizability

Chapter 3

Unique Realizability

The following few chapters will address the question of when a graph has a unique satisfying realization. That is, when is there essentially only one way to position the vertices that satisfies all the edge length constraints? Any realization can be translated, rotated and reflected to yield a new one, but these alternatives are not very interesting. Two realizations will be considered *equivalent* if all pairwise distances between vertices are the same under the two realizations. We will only be interested in realizations modulo equivalences. This question of unique realizability can be posed in any dimension, and where possible we will develop the most general results. Of course, dimensions two and three are the most practically interesting.

To apply our recursive decomposition to an instance of the molecule problem we must be able to identify subproblems with unique solutions. This identification has two aspects. First, we must have an understanding of when a problem has a unique solution. Second, we need to be able to identify subproblems within our instance that satisfy our uniqueness criteria.

Given a graph and a realization, is there another, nonequivalent realization that induces the same edge lengths? Saxe investigated the computational complexity of this problem and concluded that it is as hard as the original molecule problem; namely strongly NP-complete in dimension one and strongly NP-hard for higher dimensions [Sax79]. This is a very discouraging result, but his proofs rely upon very special locations for the vertices. If we assume that the vertex locations are unrelated to each other then strong results about unique realizability can be proved, as the following chapters will indicate.

More formally, consider a set $\mathcal{S}$ with nonzero measure. A subset $\mathcal{T}$ of $\mathcal{S}$ is said to contain *almost all* of $\mathcal{S}$ if the complement of $\mathcal{T}$, $\{q \in \mathcal{S} | q \notin \mathcal{T}\}$, has measure zero. A realization is said to be *generic* if the vertex coordinates are algebraically independent over the rationals. This computationally unrealistic requirement is actually stronger than we truly need. We just have to avoid several specific algebraic dependencies. However, the set of generic realizations is dense in the space of all realizations, and almost all realizations are generic.

The results that will be developed in the following chapters will apply to almost all realizations of a graph. The obvious drawback of this approach is that the particular instance of the molecule problem that one wishes to solve may belong to the set of counterexamples. But since this set has measure zero, our hope is that for problems of interest the counterexamples are rare. Besides, Saxe's complexity results imply that we can't expect to do much better. It is highly unlikely that there is a good, general algorithm to determine whether every possible instance of the molecule problem has a unique solution.

By assuming that our problem is typical, and not in the small set of counterexamples, we tremendously simplify the analysis. As the following chapters will make clear, the values for the edge lengths can largely be ignored. We will develop an analysis of unique realizability that depends entirely on the underlying graph, without any reference to the given edge lengths. Unfortunately, we will not be able to offer a complete characterization, only necessary conditions and a sufficient condition. These conditions will be valid for almost all realizations of the graph. However, the fact that these conditions are generic does not guarantee that unique realizability is generic as well. In fact, this remains an open question. That is, if a single generic realization of a graph is unique, are almost all realizations unique?

To determine whether a graph has a unique realization, it is helpful to ask the converse; when does a graph have multiple realizations? There are several distinct manners in which nonuniqueness can appear. First, the framework built from the graph can be susceptible to continuous deformations like the one depicted in Figure 3.1. The rightmost vertex can pivot freely since it it underconstrained. A

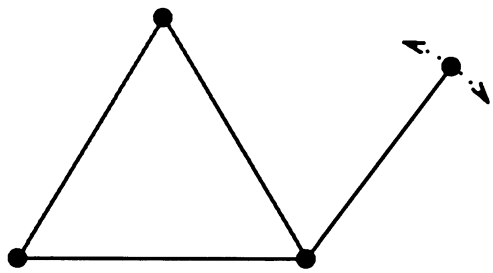


Figure 3.1: A flexible framework in two dimensions.

framework that can be continuously deformed while still satisfying the edge length constraints is said to be *flexible*; otherwise it is *rigid*. Graph rigidity is a well studied problem, although there remain many fundamental open questions. These issues will be explored in Chapter 4.

Even a rigid framework can suffer from nonuniqueness. Figure 3.2 provides a simple example. The rightmost vertex can reflect across the line formed by the central two vertices. These kinds of discontinuous transformations have not previously been well studied. We will investigate them in Chapters 5 and 6.

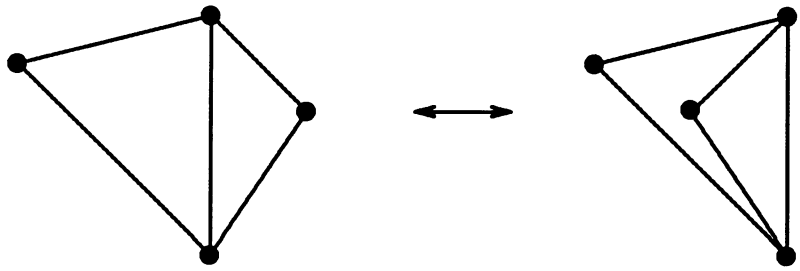


Figure 3.2: A graph with two realizations in the plane.

Three necessary conditions for almost all realizations of a graph to be unique will be developed in Chapters 4, 5 and 6, although only two of them are independent. Algorithms for detecting these properties will be presented that ignore the given edge lengths. These algorithms will have entirely different flavors in different dimensions. In addition, algorithms will be developed that identify subgraphs with these necessary properties, a critical capability for our recursive decomposition approach to the molecule problem.

Unfortunately, as we will see in Chapter 7, these conditions are not sufficient for unique realizability, at least not in dimensions larger than two. However, we will present a sufficient condition for uniqueness that again ignores edge lengths. Whereas the necessary conditions are largely combinatoric, the sufficient condition involves the nullity of a particular matrix. Unfortunately, we know of no way to use this sufficient condition to identify uniquely realizable subgraphs easily.

Once we have a handle on the question of unique graph realizations we can proceed to decompose an instance of the molecule problem into smaller subproblems. Instead of solving the full problem with a single optimization, we can identify uniquely realizable pieces and position them first. This reduces the number of parameters in the full problem, converting a large optimization into a sequence of smaller ones. Uniquely realizable subgraphs can be identified by combining the necessity and the sufficiency tests. Potentially unique subgraphs can be found using the necessary conditions, and their uniqueness can be confirmed using the sufficiency test.

Chapter 4

Graph Rigidity

A graph that has a unique realization cannot be susceptible to continuous flexings. It must be rigid. Questions about the rigidity of graphs have occupied mathematicians for centuries. More recently, structural engineers have been drawn to the problem because of novel building architectures like geodesic domes. The framework of a building can be thought of as a set of rigid rods, joined at their endpoints. One can consider the endpoints to be vertices of a graph and the rods to be edges of a fixed length. For the framework to bear weight, the corresponding graph must be rigid. For an old problem with an easy description, the characterization of rigid graphs has proved to be difficult, and many important questions remain unanswered.

Section 4.1 will develop the essentials of rigidity theory, stressing the importance of the rigidity matrix. A more complete discussion can be found in some of the references [AR78,AR79,Rot81,Cra79]. Section 4.2 will present sequential and parallel algorithms for rigidity testing.

4.1 Basic Concepts

A mathematical analysis of rigidity requires a formal definition of our intuitive notion of a flexible framework. Everything in this section occurs in an arbitrary Euclidean dimension d.

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 $(p_i(t) - p_j(t))^2 = \textit{constant}$ for every $(i,j) \in E$. Thinking of t as time, and differentiating the edge length constraints we find that

$$(v_i - v_j) \cdot (p_i - p_j) = 0 \quad \text{for every } (i,j) \in E, \tag{4.1}$$

where v_i is the instantaneous velocity of vertex i. An assignment of velocities that satisfies Equation 4.1 for a particular framework is an *infinitesimal motion* of that framework. Clearly the existence of a finite flexing implies an infinitesimal motion,

but the converse is not always true. However, for generic realizations infinitesimal motions always correspond to finite flexings [Rot81].

The infinitesimal motions of a framework constitute a vector space. Note that a motion of the Euclidean space itself, a rotation or translation, satisfies the definition of a finite flexing. Such finite flexings are said to be *trivial.* In d-space there are d independent translations and $d(d-1)/2$ rotations. If a framework has a nontrivial infinitesimal motion it is *infinitesimally flexible.* Otherwise it is *infinitesimally rigid.* As noted above, for generic realizations infinitesimal motions correspond to finite flexings. Since we are considering only generic realizations we will drop the prefix and refer to frameworks as either rigid or flexible.

We would like to be able to determine whether a particular framework is rigid or flexible. Conveniently, this is substantially a property of the underlying graph as the following theorem indicates [Glu75].

Theorem 4.1.1 (Gluck) *If a graph has a single rigid realization, then all its generic realizations are rigid.*

This theorem is critical for a graph theoretic approach to the realization problem. The frameworks built from a graph are either all infinitesimally flexible or almost all rigid. This allows for the characterization of graphs as either rigid or flexible according to the typical behavior of a framework, without reference to a specific realization. It 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.*

How can a rigid graph be recognized? Clearly, graphs with many edges are more likely to be rigid than those with only a few. In some sense the edges are constraining the possible movements of the vertices. In d-space a set of n vertices has nd possible independent motions. However a d-dimensional rigid body in d-space has d translations and $d(d-1)/2$ rotations. (If the body has dimension $d' < d$ then it has only $d'(2d-d'-1)/2$ rotations. This corresponds to a framework with only $d'+1$ vertices.) The total number of allowed motions is the number of total degrees of freedom, nd, minus the number of rigid body motions. For convenience we will call this quantity $S(n,d)$, where

$$S(n,d) = \begin{cases} nd - d(d+1)/2 & \text{if } n \geq d \\ n(n-1)/2 & \text{otherwise.} \end{cases}$$

If each edge adds an independent constraint then $S(n,d)$ edges should be required to eliminate all nonrigid motions of the graph. This intuition is sound as the theorems in this section will demonstrate.

Any realization of a flexible graph has a nontrivial infinitesimal motion. An infinitesimal motion is a solution for velocities in Equation 4.1. The matrix of this set of equations is the *rigidity matrix.* It has m rows and nd columns. Each row corresponds to an edge while each column corresponds to a coordinate of a vertex.

Each column has $2d$ nonzero elements, one for each coordinate of the vertices connected by the corresponding edge. The non-zero values are the differences in the coordinate values for the two vertices. For example, consider the graph K_3, the complete graph on three vertices, positioned in $\mathbb{R}^2$. If the realization maps the vertices to locations $(0,1)$, $(-1,0)$ and $(1,0)$, the rigidity matrix would be:

$$\begin{array}{c} \\ e_{1,2} \\ e_{1,3} \\ e_{2,3} \end{array} \begin{array}{c} \begin{array}{cccccc} v_1^x & v_1^y & v_2^x & v_2^y & v_3^x & v_3^y \end{array} \\ \begin{pmatrix} 1 & 1 & -1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 0 \end{pmatrix} \end{array}$$

The rank of the rigidity matrix is closely related to the rigidity behavior of the framework, as this section will elucidate.

Theorem 4.1.2 *A framework $p(G)$ is rigid if and only if its rigidity matrix has rank exactly equal to $S(n,d)$.*

Proof: All infinitesimal motions must be in the null space of M since the rigidity matrix expresses all constraints on the infinitesimal velocities. By construction, $S(n,d)$ is the size of the rigidity matrix minus the number of trivial infinitesimal motions. If the null space of M contains any nontrivial infinitesimal motions then the rank must be less than $S(n,d)$. ∎

So the question of whether a framework is flexible can be reduced to a question about the rank of the rigidity matrix. The framework is rigid if and only if the rank of the rigidity matrix is maximal, $S(n,d)$.

Theorem 4.1.3 *Every rigid framework $p(G)$ has a rigid subframework with exactly $S(n,d)$ edges.*

Proof: The rigidity matrix has rank $S(n,d)$ and each of its rows corresponds to an edge. Simply discard redundant rows and the corresponding edges until only S remain. ∎

Corollary 4.1.4 *For a framework $p(G)$, if $m > S(n,d)$ then there is linear dependence among the rows of the rigidity matrix.*

Proof: The maximum rank of the rigidity matrix is $S(n,d)$. ∎

Dependence among rows in the rigidity matrix can be expressed in terms of a matroid [LY82,GW88]. For our purposes it will be sufficient to say that a set of edges is *independent* if their rows in the rigidity matrix are linearly independent in a generic realization. A rigid graph has $S(n,d)$ independent edges.

Theorem 4.1.5 *If a framework $p(G)$ with exactly $S(n,d)$ edges is rigid, then there is no subgraph $G' = (V', E')$ with more than $S(n', d)$ edges, where $n' = |V'|$.*

Proof: Since there are only $S(n,d)$ edges, their rows in the rigidity matrix must all be independent by Theorem 4.1.3. But if G' has $|E'| > S(n', d)$, then by Corollary 4.1.4 there must be linear dependence among these edges which is a contradiction. ∎

Theorems 4.1.3 and 4.1.5 say that a rigid graph with n vertices must have a set of $S(n,d)$ *well distributed* edges, where well distributed means that no subgraph with n' vertices has more than $S(n', d)$ edges. This requirement is often referred to as *Laman's condition* after G. Laman [Lam70] who first articulated the two-dimensional version. This condition is necessary for a graph to be rigid in any dimension. It is sufficient in one dimension where $S = n - 1$. It is straightforward to show that this is equivalent to requiring the graph to be connected. Laman was able to show that it is also sufficient in two dimensions where $S = 2n - 3$.

Theorem 4.1.6 (Laman) *The edges of a graph $G = (V, E)$ are independent in two dimensions if and only if no subgraph $G' = (V', E')$ has more than $2n' - 3$ edges.*

Corollary 4.1.7 *A graph with $2n - 3$ edges is rigid in two dimensions if and only if no subgraph G' has more than $2n' - 3$ edges.*

This was the first graph theoretic characterization of rigid graphs in two-space. Several equivalent characterizations have since been discovered [Sug80, Ima85,LY82,TW85,Cra88].

Unfortunately, for all its intuitive appeal Laman's condition is not sufficient in higher dimensions. A three-dimensional counterexample is depicted in Figure 4.1. Although this graph has the required 18 well distributed edges it is still flexible. The top and bottom halves can pivot about the left and right-most vertices.

The problem with Figure 4.1 is that its edges are not independent in the sense of Theorem 4.1.2. The rows of the rigidity matrix are linearly dependent. Expressing this independence graph theoretically has proved to be a very difficult problem. No general characterization of rigid graphs in three dimensions is known, although the problem has been considered by many researchers, and several special cases have been solved. Cauchy proved that triangulated planar graphs (those with all $3n - 6$ edges) are generically rigid in three-space [Cau13]. Fogelsanger recently generalized this result to include complete triangulations of any two-manifold in three-space [Fog88]. Another class known to be rigid is complete bipartite graphs with at least five vertices in each vertex set [BR80,Whi84]. However, the characterization of general graphs remains open.

Recent work by Tay, Whiteley and Graver [TW85] has brought such a characterization almost within reach. However, it is difficult to see how this possible

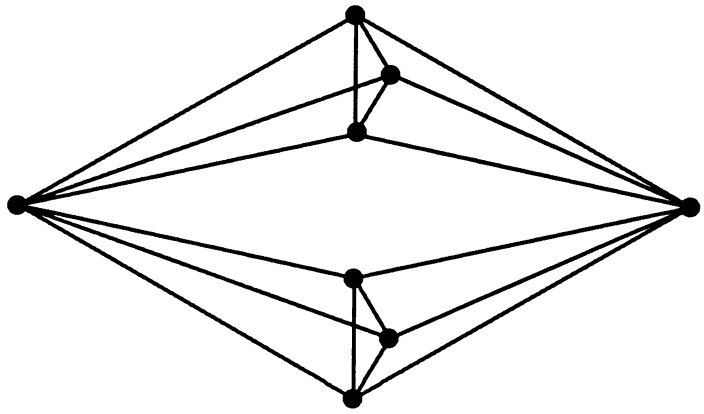


Figure 4.1: A flexible graph in three-space that satisfies Laman's condition.

solution could lead to an efficient algorithm. Any straightforward implementation of their approach would have a worst case exponential time behavior.

4.2 Algorithms for Rigidity Testing

In one-space, rigidity is equivalent to connectivity. There are simple connectivity algorithms that run in time proportional to the number of edges in the graph [AHU74].

4.2.1 Rigidity Algorithms in Two Dimensions

In two dimensions Laman's condition characterizes rigidity, but in its original form it gives a poor algorithm. It involves counting the edges in every subgraph, of which there are an exponential number. Sugihara discovered the first polynomial time algorithm for determining the independence of a set of edges in two dimensions [Sug80]. Imai presented an $O(n^2)$ algorithm for rigidity testing using a network flow approach [Ima85]. This time complexity was matched by Gabow and Westermann using matroid sums [GW88]. In this section we will develop a new $O(n^2)$ algorithm based on bipartite matching. Besides any intrinsic interest, this new algorithm will be needed in Chapter 6 when we need to test for a stronger graph condition.

We will first need to introduce a particular bipartite graph, $B(G)$, generated by our original graph $G = (V, E)$. The bipartite graph has the edges of G as one of its vertex sets, and two copies of the vertices of G for the other. Edges of $B(G)$ connect the edges of G with the two copies of their incident vertices. More formally, $B(G) = (V_1, V_2, \mathcal{E})$, where $V_1 = E$, $V_2 = \{q_1^1, q_1^2, \ldots, q_n^1, q_n^2\}$, and

$\mathcal{E} = \{(e, q_i^1), (e, q_i^2), (e, q_j^1), (e, q_j^2) : e = (v_i, v_j) \in E\}$. $B(G)$ has $2n + m$ vertices and $4m$ edges, where n and m are respectively the number of vertices and edges in G. A simple example of the correspondence between G and $B(G)$ is presented in Figure 4.2 for the graph K_3.

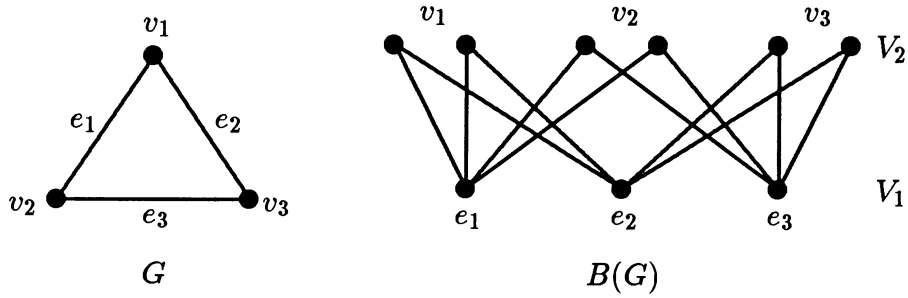


Figure 4.2: The correspondence between G and $B(G)$.

This bipartite graph leads to an alternate form of Laman's condition, expressed in the following theorem. As above, a set of edges is said to be *independent* if the corresponding rows in the rigidity matrix are linearly independent in a generic realization.

Theorem 4.2.1 *For a graph $G = (V, E)$ the following are equivalent:*

A. The edges of G are independent in two dimensions;

B. for each edge (a, b) in G, the graph $G_{a,b}$ formed by adding three additional edges between a and b has no subgraph G' in which $m' > 2n'$;

C. for each edge (a, b), the bipartite graph $B(G_{a,b})$ generated by $G_{a,b}$ has no subset of V_1 that is adjacent only to a smaller subset of V_2.

D. for each edge (a, b), the bipartite graph $B(G_{a,b})$ generated by $G_{a,b}$ has a complete matching from V_1 to V_2.

Proof: The equivalence of A and B is a restatement of Laman's condition. The equivalence of B and C is a straightforward consequence of the construction of $B(G_{a,b})$. Property D is equivalent to C by Hall's theorem from matching theory. Assertions C and D were first discovered in a slightly different form by Sugihara [Sug80]. ∎

Our algorithm will be based upon the characterization in Theorem 4.2.1D. The basic idea is to grow a maximal set of independent edges one at a time. Denote these *basis* edges by $\hat{E}$. A new edge is added to the basis if it is discovered to be independent of the existing set. If $2n - 3$ independent edges are found then the

graph is rigid. Determining whether a new edge is independent of the current basis can be done quickly using the bipartite matching characterization.

Assume we have a (possibly empty) set of independent edges $\hat{E}$. Combined with the vertices of G these form a graph $\hat{G}$, which generates a bipartite graph $B(\hat{G})$. Note that $|\hat{E}| = O(n)$ and so $B(\hat{G})$ will have $O(n)$ edges. We wish to determine if another edge, e, is independent of $\hat{E}$. Adding e to $\hat{G}$ produces $\bar{G}$ and $B(\bar{G})$. By characterization D, e is independent of $\hat{E}$ if and only if there is a complete bipartite matching in B after any edge in $\bar{G}$ is quadrupled. Actually, only e needs to be quadrupled as the following claim demonstrates.

Claim 4.2.2 *If a complete matching exists when e is quadrupled then e is independent of $\hat{E}$.*

Proof: Assume the matching succeeds but e isn't independent of $\hat{E}$. Then there must exist some edge in $\hat{E}$ whose quadrupling causes G', a subgraph of $\bar{G}$, to have $m' > 2n' - 3$. Since the edges of $\hat{E}$ are independent this bad subgraph must include e. But this bad subgraph has the same number of edges it had when e was quadrupled. Since the matching succeeded when e was quadrupled we have a contradiction. ∎

Determining whether a new edge can be added to the set of independent edges is now reduced to the problem of trying to enlarge a bipartite matching. This is a standard problem in matching and it is performed by growing Hungarian trees and looking for augmenting paths. The basic idea is to look for a path from an unmatched vertex in V_1 to an unmatched vertex in V_2 that alternates between edges that are not in the current matching and edges that are. When such an augmenting path is found the matching can be enlarged by changing the unmatched edges in the path to become matching edges, and vice versa. These paths can be found by growing Hungarian trees from the unmatched vertices in V_1. These trees grow along the unmatched edges from the starting vertex to its neighbor set in V_2. Matching edges are followed back to V_1 and unmatched edges back to V_2. If an unmatched vertex in V_2 is ever encountered an augmenting path has been identified. Growing a Hungarian tree takes time proportional to the number of edges in the bipartite graph.

Claim 4.2.3 *If $\hat{E}$ is independent and a corresponding matching in $B(\hat{G})$ is known, then determining whether a new edge is independent requires $O(n)$ time.*

Proof: By claim 4.2.2, testing for independence of e requires just enlarging the matching in $B(\hat{G})$ to include the four copies of e. This involves growing four Hungarian trees in a bipartite graph of size $O(n)$. ∎

This gives a two-dimensional rigidity testing algorithm that runs in time $O(nm)$. Build a maximal set of independent edges one at a time by testing each edge for

independence. Each test involves the enlargement of a bipartite matching requiring $O(n)$ time. If the matching succeeds the edge is independent and is added to the basis. Otherwise it is discarded.

To improve this to $O(n^2)$ we need to make use of failed matchings to eliminate some edges from consideration. Define a *Laman subgraph* as a subgraph with n' vertices and $2n' - 3$ independent edges. A matching will fail precisely when the new edge lies in a subgraph which already has $2n' - 3$ independent edges. No edge can be added between vertices in this subgraph, so it is a waste of time to even try. By avoiding these unnecessary attempts we can improve the performance of our algorithm. To accomplish this we will need some further insight into the bipartite matching.

Theorem 4.2.4 *In a bipartite graph $(V_1, V_2, \mathcal{E})$, if a Hungarian tree fails to find an alternating path then it spans a minimal subgraph which violates Hall's theorem. That is, it identifies a minimal set of k vertices in V_1 with fewer than k neighbors.*

Proof: This is a simple consequence of Hall's theorem. ∎

Claim 4.2.5 *If the new edge, e, is tripled instead of quadrupled, generating a graph $\underline{G}$ from $\hat{G}$, then $B(\underline{G})$ has a complete matching.*

Proof: Assume the contrary. Then there is some subgraph G' of $\bar{G}$ with $m' > 2n'$. Remove the three copies of e from this subgraph and quadruple one of the other edges. This altered subgraph still has $m' > 2n'$, but it is the graph generated by quadrupling an edge in $\hat{G}$. But since the edges of $\hat{G}$ are assumed to be independent this is a contradiction. ∎

Claim 4.2.6 *If edge e fails the matching test, then the failing Hungarian tree spans a set of edges of $\hat{E}$ that form a Laman subgraph.*

Proof: By Claim 4.2.5 when e is quadrupled the first 3 copies of it can be matched. By Theorem 4.2.4 when the fourth fails it spans a set of vertices of V_1 adjacent to a smaller set from V_2. Discarding the four copies of e leaves a set of k elements of V_1 adjacent to no more then $k+3$ vertices from V_2. By the construction of the bipartite graph this is a set $\hat{E}'$ of k edges of $\hat{E}$ incident upon no more than $(k+3)/2$ vertices. That is, $m' \geq 2n' - 3$. Since the edges of $\hat{E}$ are independent we must have equality. ∎

We will need the following result to analyze the running time of our algorithm.

Theorem 4.2.7 *Let $G = (V, \hat{E})$ be a graph whose edges are independent. If two Laman subgraphs of G share an edge then their union is a Laman subgraph.*

Proof: Let the subgraphs be (V', E') and (V'', E'') with union $(\bar{V}, \bar{E})$. Let $\bar{m} = m' + m'' - l$ and $\bar{n} = n' + n'' - k$. Since the subgraphs share at least one independent edge, $l \leq 2k - 3$. Hence,

$$\begin{aligned} \bar{m} &= 2n' + 2n'' - 6 - l \\ &\geq 2n' + 2n'' - 2k - 3 \\ &= 2(n' + n'' - k) - 3 \\ &= 2\bar{n} - 3. \end{aligned}$$

Since the edges are independent we must have equality. ∎

We are now ready to present our algorithm. We will maintain the appropriate bipartite graph with a matching of all the independent edges discovered so far. We will also keep a collection of all the Laman subgraphs that have been identified, represented as linked lists of independent edges. The algorithm is outlined in Figure 4.3.

```
basis ← ∅
for each vertex v
    Mark each vertex in a Laman subgraph with v, and unmark all others
    for each edge (u, v)
        if u is unmarked then
            if (u, v) is independent of basis then
                add (u, v) to basis
                create Laman subgraph consisting of (u, v)
            else a new Laman subgraph has been identified
                Merge all Laman subgraphs that share an edge
                Mark each vertex in a Laman subgraph with v
```

Figure 4.3: An $O(n^2)$ algorithm for two-dimensional graph rigidity.

By Theorem 4.2.7 we know that no edge need be in more than one subgraph. By merging whenever a new Laman subgraph is found, we guarantee that the total number of elements in all the subgraphs is kept to $O(n)$. This ensures that the marking and merging operations can be done in $O(n)$ time. As above, checking for independence of (u, v) requires $O(n)$ time. Each time an edge is checked it results in either a new basis edge or a merging of components, so this can only happen $O(n)$ times. Hence the total time for the algorithm is $O(n^2)$.

4.2.2 Rigidity Algorithms in Higher Dimensions

For dimensions greater than two there are no graph theoretic characterizations of rigidity, so there are no good combinatoric algorithms to test for it. One approach would involve a symbolic calculation of the rank of the rigidity matrix by symbolically constructing the determinant. However, the determinant can have an exponential number of terms, so this requires an exponential amount of time. A different approach is possible which relies instead upon Theorem 4.1.1. Since this theorem is valid in all dimensions, the following discussion is applicable to all spaces. If the graph is rigid then almost any realization will generate a rigid framework. Simply select a random realization for the graph. Once these vertex locations are selected it is a straightforward matter to determine the rigidity of the framework using Theorem 4.1.2. Just construct the rigidity matrix M and determine its rank. If the rank is $S(n,d)$ then the graph is rigid. A lower rank indicates that the framework is flexible. Unless the selection of vertex coordinates was extremely unlucky the underlying graph will be flexible as well. So even without a graph theoretic characterization an efficient practical randomized algorithm for rigidity exists.

To determine the rank of M we suggest using a QR decomposition with column pivoting, requiring $O(mn^2)$ time [GVL83]. This is more numerically stable than Gaussian elimination, but not as costly as a singular value decomposition. A QR factorization has several advantages over an SVD in this application. Performing a QR on M^T will identify a maximal independent set of rows of M one at a time, corresponding to a maximal set of independent edges in the graph. This ability to identify independent rows will be needed in Chapter 6. Also, the rigidity matrix is quite sparse, having only $2d$ nonzeros in each row. To save time and space, sparse techniques could be used for large problems. There are sparse QR algorithms, but none for SVD [CEG86,GH80,Gil86].

There are also efficient parallel algorithms for finding the rank of a matrix. Ibarra, Moran and Rosier [IMR80] discovered an algorithm that runs in $O(\log^2 m)$ time on $O(m^4)$ processors. This means that rigidity testing is in random NC for any dimension. The class NC is the set of problems that can be solved in polylogarithmic time using a polynomial number of processors. It is a standard measure of a *good* parallel algorithm, although its applicability is more theoretical than practical.

Chapter 5

Partial Reflections

Even rigid graphs can have multiple realizations as was shown in Figure 3.2. This discontinuous flavor of nonuniqueness has not been well studied, probably because it is not relevant to structural engineers. Buildings can only deform continuously. For the graph realization problem these discontinuous transformations must be considered. This chapter and the next will be concerned with multiple realizations that do not arise from flexibility. These are cases in which there are two or more noncongruent realizations that satisfy all the distance requirements, but there is no continuous flexing of the framework to transform one to another while maintaining the constraints. Whereas flexible graphs have an infinite number of potential configurations, the number of realizations of a rigid graph is finite, although possibly exponential in the size of the graph.

A two-dimensional example of the simplest type of discontinuous transformation is depicted in Figure 3.2. The right half of this graph is able to fold across the line formed by the two middle vertices. When can this type of nonuniqueness occur? As in Figure 3.2 there must be a few vertices about which a portion of the graph can be *reflected.* These vertices form a *mirror.* There must be no edges between the two halves of the graph separated by this mirror. For the general d-dimensional problem, the mirror vertices must lie in a $(d-1)$-dimensional subspace. We will say that a framework in d-space allows a *partial reflection* if a separating set of vertices lies in a $(d-1)$-dimensional subspace.

The realizations in which more than d vertices lie in a $(d-1)$-dimensional subspace are not generic. So for almost all frameworks, partial reflections only occur when there is a subset of d or fewer vertices whose removal separates the graph into two or more unconnected pieces, that is, when G is not vertex $(d+1)$-connected. This gives us the following well known result.

Theorem 5.0.1 *A rigid graph positioned generically in dimension d will have a partial reflection if and only if it is not vertex $(d+1)$-connected.*

The connectivity of a graph is an important property, and it has been well studied. There are well known $O(m)$ time algorithms for vertex two-connectivity, also

known as *biconnectivity* [AHU74]. Avoiding partial reflections in two dimensions requires a vertex three-connected (or *triconnected*) graph. Hopcroft and Tarjan [HT73] were the first to discover an $O(m)$ time algorithm to find triconnected components. Miller and Ramachandran [MR87] have recently proposed a parallel algorithm to identify triconnected components in $O(\log^2 n)$ time with $O(m)$ processors. This places triconnectivity in NC.

Four-connected components are more difficult, but Kanevsky and Ramachandran [KR87] have recently found an $O(n^2)$ time algorithm. They also discovered a parallel implementation of their algorithm that runs in $O(\log n)$ time using $O(n^2)$ processors. So the problem of partial reflections is in NC in both two and three dimensions.

For k greater than 4, the question of k-connectivity for a general k is less well understood. Consequently the partial reflection problem is more difficult in spaces of dimension greater than three. There are randomized algorithms for general k-connectivity that run in time proportional to $n^{5/2}$ [B+82,LLW86]. Recently, Cheriyan and Thurimella have described an algorithm with a time complexity of $O(k^3n^2)$, which reduces to $O(n^2)$ for a fixed k [CT90]. There are also NC algorithms that run in time $O(k^2 \log n)$ [KS90].

Chapter 6

Redundant Rigidity

Rigidity and $(d+1)$-connectivity are necessary but not sufficient to ensure that a graph has a unique realization. A two-dimensional example of a rigid, triconnected graph with two satisfying realizations is given in Figure 6.1.

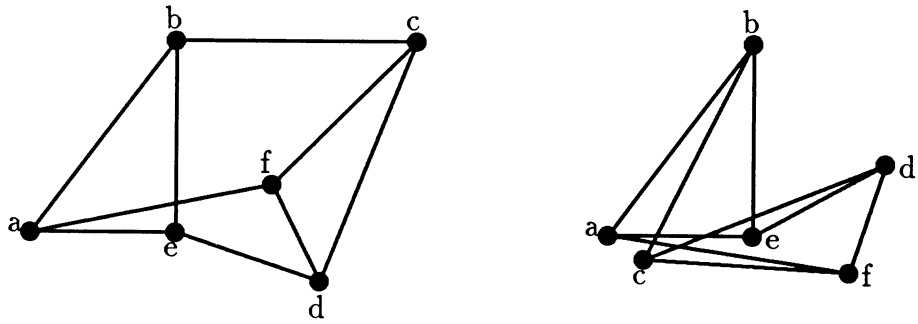


Figure 6.1: Two realizations of a rigid triconnected graph in the plane.

To understand this nonuniqueness, consider Figure 6.2. Edge (a, f) has been removed from the original graph. This resultant graph is now composed of a quadrilateral *bcde* with two attached triangles *abe* and *cdf*. The quadrilateral is not rigid, so this new graph can flex. The flexing will lift vertex d up until it crosses the line between c and e as depicted in the center picture of Figure 6.2. Eventually vertex c can swing all the way around to the right. As the graph moves, the distance between vertices a and f varies. When vertex c swings far enough around, this distance becomes the same as it was originally, as shown in the rightmost picture of Figure 6.2. Now the missing edge can be replaced to yield a new satisfying realization.

The fundamental problem with the graph in Figure 6.1 is that the removal of a single edge makes it flexible. We will define an edge of a framework to be

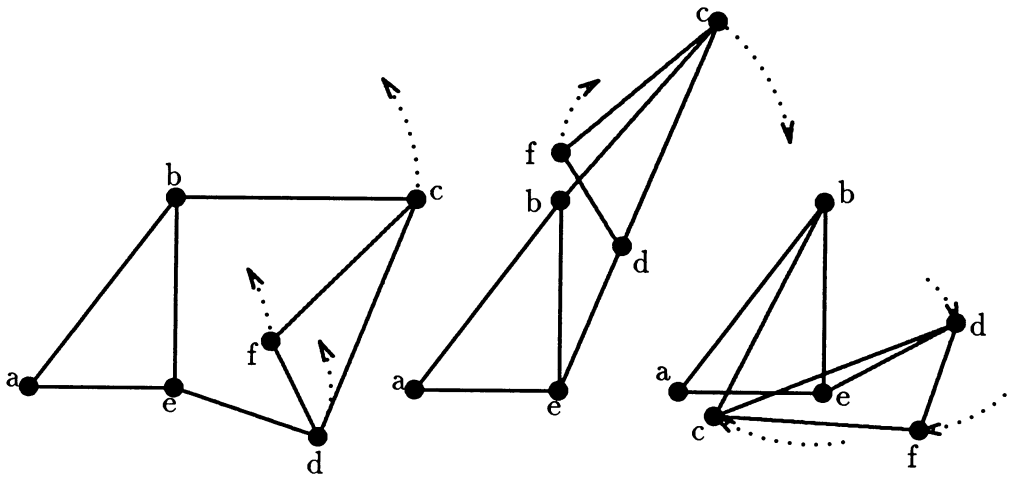


Figure 6.2: Intermediate stages in the construction of Figure 6.1.

redundant if the framework remaining after its removal is rigid. A framework is *redundantly rigid* if all its edges are redundant. Redundant bracing is a familiar concept to engineers who wish to build frameworks with additional strength and failure tolerance properties. But this precise formulation and its significance with regards to the unique realization problem are entirely new.

Redundant rigidity is clearly a more restrictive property than rigidity, but the two properties have many similarities as the following theorem indicates.

Theorem 6.0.1 *If a graph has a single redundantly rigid realization, then all its generic realizations are redundantly rigid.*

Proof: This is a straightforward consequence of Theorem 4.1.1. ∎

As with Theorem 4.1.1 for rigid graphs, Theorem 6.0.1 says that either none of a graph's realizations are redundantly rigid, or almost all of them are. *Almost all* again means that the set of counter examples has measure zero. This blurs the distinction between a redundantly rigid framework and its underlying graph. Graphs with redundantly rigid realizations will be referred to as *redundantly rigid graphs.*

In Figure 6.1 the lack of redundant rigidity led to multiple realizations. This is usually true, and the proof will be the main result of this chapter. Intuitively, a flexible framework can move around, but it must always end up back where it started. That is, the path it traces in nd-space will be a loop. If the removal of an edge allows the graph to flex, then the distance corresponding to that edge must be a multivalued function as the flexing completes its loop. However, there are some graphs for which this argument fails. Consider the triangle graph, K_3. It has

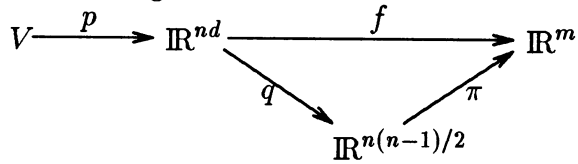
only one realization, but if an edge is removed it becomes flexible. To understand which graphs need to be redundantly rigid to have unique realizations we will need to carefully investigate the space of satisfying realizations for flexible graphs. This will require an incursion into differential topology, and is the subject of the next section.

6.1 The Necessity of Redundant Rigidity

The proof that flexible graphs typically move in closed loops will rely upon some special properties of the graph realization problem. Given a framework $p(G)$ there is a *pairwise distance function* $q : \mathbb{R}^{nd} \rightarrow \mathbb{R}^{n(n-1)/2}$ that maps vertex locations to squares of all the pairwise vertex distances. That is,

$$q(p(v_1), \ldots, p(v_n)) = (\ldots, |p(v_i) - p(v_j)|^2, \ldots).$$

For the molecule problem we are only interested in a specific set of pairwise distances; namely, those corresponding to the edges of G. These can be obtained from $R^{n(n-1)/2}$ by a simple projection, π. We will define an *edge function*, f, to be the composition of these two operations, $f = \pi \circ q$. These functions are described by the following commutative diagram.



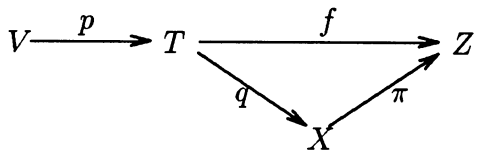
The functions f and q have many nice properties. We will say a function is *smooth* at a point x if it has continuous partial derivatives of all orders at x. The functions f and q are everywhere smooth. Also, the Jacobian of f is twice the rigidity matrix introduced in Section 4.1.

The realization problem is really that of finding the inverse of the edge function. Of course, this inverse is multivalued because edge lengths are invariant under translations, rotations and reflections of the entire space. Two realizations will be considered *equivalent* if all pairwise distances between vertices are the same under the two realizations. That is, two realizations are equivalent if they map to the same point under q. We will be interested only in the inverse of f modulo equivalences. More formally, define the *realization set* of $p(G)$ to be $\pi^{-1}f(p(G))$, the set of nonequivalent, satisfying realizations for the graph realization problem generated by $p(G)$. For $p(G)$ to be a unique solution to the realization problem it is necessary and sufficient that this realization set consist of a single point. Our goal in this section is to investigate the structure of the realization set. Our first result is the following theorem.

Theorem 6.1.1 *If a graph, G, is connected, then the realization set of $p(G)$ is compact.*

Proof: The realization set is a subset of $\mathbb{R}^{n(n-1)/2}$. It is bounded since the graph is connected, and it is trivially closed. ∎

Although every point in $\mathbb{R}^{nd}$ corresponds to a realization, the image of $\mathbb{R}^{nd}$ under q does not cover $\mathbb{R}^{n(n-1)/2}$. Define this image to be a space $W \subset \mathbb{R}^{n(n-1)/2}$. The space W has a natural topology and measure inherited from the larger Euclidean space. For technical reasons we will restrict our consideration of realizations to those in which not all the vertices lie in a hyperplane. Call this subset of realizations T. The space T is a dense, open subset of $\mathbb{R}^{nd}$. Define X to be the subset of points in W that are images of points in T under q. If the graph has d or fewer vertices then X is empty. Otherwise, X is a dense, open subset of W, with a nice structure, as we will see shortly. Define Z to be the image of X under π. This gives us the following structure.



We will need the following notation from differential topology. Say the largest rank the Jacobian of a function $g : A \to B$ attains in its domain is k. A point $x \in A$ is called a *regular point* if the Jacobian of g at x has rank k. A point $y \in B$ is a *regular value* if every point in the preimage of y under g is a regular point. If a point or value is not regular, it is *singular*. Note that for the edge function singular points are not generic. A *j-dimensional manifold* is a subset of some large Euclidean space that is everywhere locally diffeomorphic to $\mathbb{R}^j$.

Consider the following procedure for identifying equivalent realizations, which is defined for any realization in T. Select a set of $d+1$ vertices from $p(G)$ whose affine span is all of $\mathbb{R}^d$. Translate the realization so that the first of these vertices is at the origin. Next rotate about the origin to move the second of these vertices onto the positive x_1 axis. Now rotate, keeping the first two vertices fixed, to move the third to the (x_1, x_2) plane so that the x_2 coordinate is positive. Continuing this process in the obvious way gives a smooth mapping that makes $d(d+1)/2$ of the vertex coordinates zero. Finally, if the $d+1^{st}$ vertex has its $d+1^{st}$ coordinate less than zero, reflect the vertices through the hyperplane defined by the $x_1, \ldots, x_{d-1}$ axes.

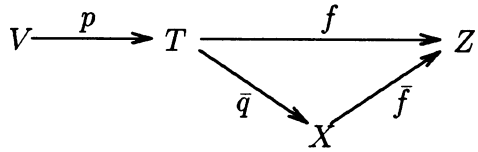
This procedure maps all equivalent realizations to a single one. This single realization can be described by its remaining variable coordinates, of which there are $nd-d(d+1)/2$. Since each of these remaining coordinates can vary continuously, the realization can be considered to be a point in $\mathbb{R}^{nd-d(d+1)/2}$. This defines a coordinate chart for X. Note that the sequence of operations performed on the original realization is smooth and invertible. If a different set of $d+1$ initial vertices was selected a different coordinate chart would have been generated. Since these coordinate transformations are smooth and invertible, on regions of intersection the

two charts are diffeomorphic. The union of all such charts gives a differentiable structure to our space X. This construction provides a diffeomorphism between each open set of a collection that covers X and $\mathbb{R}^{nd-d(d+1)/2}$, giving us the following theorem.

Theorem 6.1.2 *If the graph has at least $d+1$ vertices, then X is a smooth manifold of dimension $nd - d(d+1)/2$.*

The dimension of this manifold is a quantity that will come up frequently so it will be convenient to reintroduce the following notation: $S(n,d) = nd - d(d+1)/2$. This function was first defined in Section 4.1 as the maximal rank of the rigidity matrix of a graph with n vertices positioned in d-space.

The procedure described above gives us an alternate way in which to view the space X. The sequence of translations, rotations and reflections constitute a function $\bar{q}$ that maps an entire set of equivalent realizations to a single one. The remaining variable coordinates uniquely define a point in X. Considering these to be the independent variables, the mapping from X to Z becomes more complicated than a simple projection. We will define this function to be $\bar{f}$, giving us the following commutative diagram.



This function $\bar{f}$ is closely related to the edge function f. In fact, $\bar{f}$ is everywhere smooth in X, and the rank of the Jacobian of $f(x)$ is the same as that of $\bar{f}(\bar{q}(x))$. So the singular values of f are the same as the singular values of $\bar{f}(\bar{q})$. If we designate the number of independent edges of G by k then the rank of these Jacobians is almost always k. The following is a special case of a well known theorem due to Sard [Sar42].

Theorem 6.1.3 (Sard) *The set of singular values of f has k-measure zero.*

Lemma 6.1.4 *If Z' is a subset of Z with k-measure zero, then for almost all realizations p, $f(p) \notin Z'$.*

Proof: The singular points of f constitute an algebraic variety in $\mathbb{R}^{nd}$ with dimension less than nd. Hence, the regular points of f can be covered by a countable number of open neighborhoods in such a way that the rank of the Jacobian of f is maximal within each neighborhood. Consider one of these neighborhoods $\mathcal{R}$, and let its image under f be $\mathcal{Z}$. By the implicit function theorem from analysis there is a submersion from $\mathcal{R}$ to $\mathcal{Z}$. That is, on this neighborhood f is diffeomorphic to a projection from $\mathbb{R}^{nd}$ to $\mathbb{R}^k$. Since Z' has k-measure zero its inverse image under this submersion must have (nd)-measure zero in $\mathcal{R}$. The countable union of these

sets of (nd)-measure zero yields a preimage for Z' with (nd)-measure zero. ∎

These last two results imply the following theorem.

Theorem 6.1.5 *For almost every realization p, $f(p)$ is a regular value.*

All this has been leading up to the following crucial result.

Theorem 6.1.6 *For almost every realization p, the realization set of $p(G)$ restricted to X is a manifold.*

Proof: Almost all realizations map to regular values of f and hence of $\bar{f}(\bar{q})$. The preimage of a regular value is a submanifold of X by the implicit function theorem from differential topology [GP74]. ∎

If the graph is flexible then this manifold describes the allowed flexings. At any point in the manifold, the tangent space is exactly the null space of the Jacobian of $\bar{f}$. To show that flexings typically move in closed loops (actually, one-manifolds diffeomorphic to the circle), we will need the flexings to remain entirely in our manifold X. This can be ensured if the graph has *enough* independent edges. Enough means more than can be independent in a lower dimensional space as the following theorem demonstrates.

Theorem 6.1.7 *If G has more than $S(n, d-1)$ independent edges, then for almost all realizations $p(G)$, the realization set of $p(G)$ stays within X.*

Proof: Assume the theorem is false. By the definition of X this means that the realization set must include a point at which all the vertices lie in a $(d-1)$-dimensional hyperplane. When this happens the edges can only constrain infinitesimal motions within the hyperplane. The rows of the rigidity matrix describe these infinitesimal constraints, so when the vertices lie in a hyperplane the rank of the rigidity matrix can be no larger than $S(n, d-1)$. Since there are more than $S(n, d-1)$ independent edges, this implies that $f(p(G))$ is a singular value, but by Theorem 6.1.5 this cannot be the case for almost all realizations. ∎

We can finally prove that flexings typically move in closed loops.

Theorem 6.1.8 *If a graph, G, is connected, flexible, and has more than $d+1$ vertices, then for almost all realizations $p(G)$ the realization set of $p(G)$ contains a submanifold that is diffeomorphic to the circle.*

Proof: Generate a new graph G' from G by arbitrarily adding additional edges until G' has $S(n, d) - 1$ independent edges. The realization set of $p(G')$ must be a subset of the realization set of $p(G)$. Since $n > d+1$ it is easy to show that the number of independent edges is now greater than $S(n, d-1)$. By Theorems 6.1.6

and 6.1.1 we know that for almost all realizations the realization set of $p(G')$ is a compact manifold of dimension one. It is a well known result from differential topology that such manifolds are diffeomorphic to the circle. ∎

This finally leads us to the main result of this section.

Theorem 6.1.9 *If G is not redundantly rigid and G has more than $d+1$ vertices, then almost all realizations of G are not unique.*

Proof: Assume the only interesting case, that G is rigid. Then the graph G must have $S(n,d)$ independent edges, and there is some edge e_{ij} of G whose removal generates a flexible graph G'. By Theorem 6.1.8, for almost all realizations p the realization set of $p(G')$ contains a submanifold diffeomorphic to the circle. The distance between vertices i and j will be a multivalued function for almost every point on this circle. The only distances that might not be multivalued are the extremal ones. When a flexing reaches a realization that induces an extremal value between i and j the derivative of $d^2_{i,j}$ is zero in the direction of the flex. In this case the realization is not generic [Rot80]. So almost all realizations do not induce extremal edge lengths. ∎

Theorem 6.1.9 means that the example in Figure 6.1 was not a fluke. Redundant rigidity is a necessary condition for unique realizability.

6.2 Algorithms for Redundant Rigidity

How difficult is it to test for redundant rigidity? A simplistic approach would use the algorithm for rigidity repeatedly, removing one edge at a time. This approach parallelizes easily by simply running the m different problems on independent sets of processors. Since rigidity testing was shown to be in deterministic or random NC for all dimensions, redundant rigidity is as well.

In one dimension redundant rigidity is equivalent to edge two-connectivity. This property can be determined by looking for cut points of the graph, requiring $O(m)$ time [AHU74].

For the two-dimensional case a simple modification of the rigidity testing algorithm described in Section 4.2 can be employed. The rigidity algorithm grows a basis set of independent edges one at a time by checking them against the existing independent set. If a new edge is found to be independent of the existing set, then it is added. Independence is determined by the success of a particular bipartite matching. If the matching fails then there must be some dependence among the edges. Identifying and utilizing these dependencies will lead to an efficient redundant rigidity algorithm.

As in Section 4.2 we will denote by $B(G)$ the bipartite graph constructed from $G = (V, E)$. The current set of independent, *basis* edges is $\hat{E}$, generating a subgraph

$\hat{G} = (V, \hat{E})$. When a new edge, e, is to be tested for independence, four copies of it are added to $\hat{G}$ generating $\bar{G}$ with its corresponding bipartite graph $B(\bar{G})$. As we saw in Section 4.2, if a complete bipartite matching exists in $B(\bar{G})$ then e is independent of $\hat{E}$. For our current purposes we are interested in dependent edges and how they contribute to redundant rigidity. Dependent edges fail to have complete matchings in $B(\bar{G})$. However, if we triple e instead of quadrupling it, generating $\underline{G}$ and $B(\underline{G})$, then Theorem 4.2.5 guarantees that $B(\underline{G})$ always has a complete matching. So only a single vertex in $B(\bar{G})$ can go unmatched. This is important because of the following general property of bipartite matching.

Theorem 6.2.1 *Let $B = (V_1, V_2, \mathcal{E})$ be a bipartite graph with a matching from V_1 to V_2 involving all but one vertex from V_1, denoted by v. Also let $\mathcal{V}_1$ be the subset of V_1 that is in the Hungarian tree built from v. Then if any vertex from $\mathcal{V}_1$ is deleted from B, the resulting graph will have a complete matching.*

Proof: The removal of a vertex w from $\mathcal{V}_1$ creates an unmatched vertex in V_2 that is reachable from v along an alternating path. ∎

Theorem 6.2.1 identifies which vertices of a bipartite graph can be removed to result in a perfect matching. For our purposes, these are vertices in $B(\bar{G})$, which correspond to edges of $\bar{G}$. If any of these edges of $\bar{G}$ is removed then the new edge e will be independent of the remaining basis edges. That is, e can replace any of these edges identified by the Hungarian tree, leaving the number of basis edges unchanged. More formally, we have the following theorem.

Theorem 6.2.2 *In the rigidity algorithm, assume a new edge, e, is found to be not independent of the current set of k independent edges. Let $\mathcal{V}_1$ be the subset of vertices of V_1 that are in the Hungarian tree of the failed matching. Then if e replaces any of the edges in $\mathcal{V}_1$ the resulting set of k edges is still independent.*

Theorem 6.2.2 gives an efficient algorithm for redundant rigidity testing. An edge is not independent of the current basis set if the bipartite matching fails. When this happens the Hungarian tree identifies precisely which edges are dependent. All these edges are redundant because any of them could be replaced by the new edge. In the $O(n^2)$ algorithm from Section 4.2 a Laman subgraph is identified by this Hungarian tree. Hence, any edge in the Laman subgraph is redundant. When the algorithm is finished, if there is a basis edge that has not been merged into a larger Laman subgraph then it is not redundant and the graph is not redundantly rigid. Note that if the full graph is not redundantly rigid then the Laman subgraphs identified by this procedure are redundantly rigid components. This takes essentially no more effort than testing for rigidity, so two-dimensional redundant rigidity can be decided in $O(n^2)$ time.

In dimensions greater than two there is no graph theoretic characterization of redundant rigidity. As in Section 4.2 an algorithm will have to randomly position

the vertices and then examine the rigidity matrix. Like the two-dimensional case, the basic idea will be to build a set of independent edges one at a time, and then determine which of them are redundant. Every time a new edge fails to be independent it supplies information about the redundancy of some of the independent edges. If a full set of redundant, independent edges are found then the graph is redundantly rigid.

Begin by positioning the vertices randomly and constructing the rigidity matrix M. The rigidity of the framework can be determined by performing a QR factorization on M^T to find its rank. This procedure will form an independent set of edges one at a time. A new column is added if it is linearly independent of the current set of k columns; otherwise it is discarded. A discarded column, corresponding to an edge e, 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.

How difficult is it to determine which of the current columns contribute to the linear combination? Assume the algorithm has identified k independent columns of M^T. Place these columns 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 a $nd \times k$ 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.

How much work does this take? There are $O(m)$ triangular systems to solve, each of which requires $O(k^2)$ operations, where k is always $O(n)$. So the total additional time is of the same order as the QR factorization itself, $O(mn^2)$. As in the two-dimensional case, the redundant rigidity of a graph can be determined by modifying the rigidity algorithm without incurring substantial increased cost.

As was noted in Section 4.2, the rigidity matrix consists mostly of zeros. For large problems this property should be exploited by using sparse matrix techniques. The only real modification to the rigidity algorithm required to verify redundant rigidity is a sequence of triangular solves. These can be done sparsely, so the entire algorithm can be implemented in a sparse setting. An algorithm very similar to this has been described by Coleman and Pothen [CP87].

Chapter 7

A Counterexample and a Sufficient Condition

In the preceding three chapters we have developed necessary graph theoretic conditions for a graph to have a unique realization. These conditions have several nice properties. First, being generic properties they are valid for almost all realizations of the graph. This permits a greatly simplified analysis that ignores the actual edge lengths. Second, efficient algorithms exist for checking these conditions. Third, and critically important for our purposes, if the entire graph fails one of the tests, subgraphs that pass it can be identified. This combination of properties is ideal for our recursive decomposition approach to the molecule problem.

Unfortunately, these three properties are not sufficient for unique realizability. A counterexample, the graph $K_{5,5}$, will be described in Section 7.1. This graph satisfies our necessary conditions while still having generic realizations that are not unique.

This counterexample places our recursive decomposition technique in jeopardy. The necessary conditions might pass a subgraph that is not truly unique, allowing it to be positioned incorrectly. The subgraph will then be unable to fit properly with the remainder of the graph. Fortunately for our purposes, there is a sufficient condition for uniqueness. It involves the nullity of the *stress matrix*, and will be described in detail in Section 7.2. Like the necessary conditions, this sufficiency test is generic. If a generically realized framework passes the test then all generic frameworks will pass and they will all be unique. However, it is not clear how the sufficiency property can be used to identify uniquely realizable subgraphs. Identifying subgraphs is a crucial aspect of our approach to the molecule problem, so this shortcoming is a serious problem.

The weaknesses of the necessary and the sufficiency tests can be substantially overcome by combining them. The necessary conditions can be used to identify subgraphs that satisfy all the necessity tests. These subgraphs are good candidates for unique realizability, which can be confirmed by the sufficiency test. In this way

subgraphs can be efficiently identified that have unique realizations. An outline of this approach is presented in Figure 7.1.

```
if Graph is K5,5 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
```

Figure 7.1: An algorithm for finding uniquely realizable subgraphs.

The only case that is not handled with this approach occurs when a graph passes the necessary conditions and fails the sufficiency test. We have yet to stumble across such a graph, although we would be very interested in finding one. In practice this approach seems to work very well in practice, at least on the problems that will be described in Chapter 12. Incidently, our failure to find any graphs that pass the necessary conditions while failing the sufficiency test is strong evidence that such graphs are uncommon if they exist at all.

7.1 A Counterexample, $K_{5,5}$

Consider the complete bipartite graph $K_{5,5}$, depicted in Figure 7.2. It has 10 vertices and 25 edges. It can be shown to be redundantly rigid in three-space, and it is four-connected (and in fact, five-connected). Thus it satisfies the necessary conditions developed in the preceding chapters for a graph to be generically uniquely realizable.

However, Connelly has shown that there are generic realizations of $K_{5,5}$ in three-space that are not unique [Con89a]. (There is evidence that all generic realizations are not unique, but this has not been proven.) This demonstrates that the necessary conditions are not sufficient.

Furthermore, Connelly has identified an entire class of bipartite graphs in high dimensional spaces that satisfy our necessary conditions while having nonunique generic realizations [Con89b]. In this class of graphs there are no examples in two dimensions and $K_{5,5}$ is the only one in three-space.

The nonuniqueness of $K_{5,5}$ is difficult to understand from a purely combinatoric viewpoint like the one taken in the preceding three chapters. An alternate way to

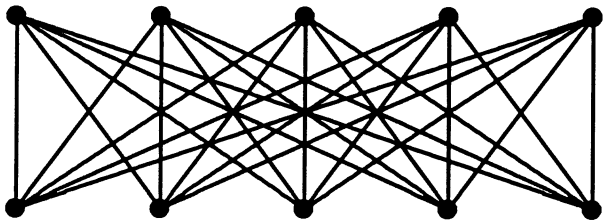


Figure 7.2: The graph $K_{5,5}$.

view the uniqueness of graph realizations will be developed in the next section. It was from this analysis that the class of bipartite graphs was first identified.

7.2 The Stress Matrix and a Sufficiency Test

Consider a framework $p(G)$ consisting of a graph G and a generic satisfying realization p. We wish to determine whether there is another satisfying realization that is not equivalent to p. A *stress* for $p(G)$ is an assignment of scalars $\omega_{ij} = \omega_{ji}$ to every edge of G satisfying for each vertex i

$$\sum_j \omega_{ij}(p_i - p_j) = 0, \tag{7.1}$$

where p_k is the location in $\mathbb{R}^d$ of vertex k, and the sum is over all vertices j adjacent to i. Note that these are vector equations since each p_k has d coordinates. Each of the d dimensions must satisfy an identical set of equations. We will discuss how to compute a stress in the next section, but any redundantly rigid framework has a least one stress.

For notational convenience we will assign a stress of zero to every pair of vertices in the graph which do not share an edge. This allows the summation in Equation 7.1 to extend over all vertices in the graph. 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 Equation 7.1.

Equation 7.1 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 [Con89c].

Theorem 7.2.1 *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 Equation 7.1. It will be convenient to rewrite the stress equations. Let q_i^r denote the r^{th} coordinate 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:

$$(\sum_{j=1}^{n} \omega_{ij})\, q_i^r - \sum_{j=1}^{n} \omega_{ij} q_j^r = 0, \tag{7.2}$$

where the summation now extends over all j since we assume unconnected vertices have a zero stress. 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 the r^th coordinate of each vertex by q^r, then Equation 7.2 can be succinctly expressed as:

$$\Omega q^r = 0, \tag{7.3}$$

for each dimension r. Any satisfying realization must satisfy these equations. 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 $d+1$. Any linear combination of these null vectors satisfies 7.3. These linear combinations 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. This gives us the following theorem.

Theorem 7.2.2 *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 $p(G)$ 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 [Con89c].

This gives us the following sufficient condition for a graph to have a unique realization.

Theorem 7.2.3 *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, 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.

The graph $K_{5,5}$, realized in three-space, generates a 10×10 stress matrix Ω with nullity 8. It was this fact that first identified it as a potentially nonunique graph.

For a given realization, the stresses defined by Equation 7.1 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. This gives us the following results.

Theorem 7.2.4 *The nullity of the stress matrix is a generic property; that is, it has the same value for all generic realizations.*

Corollary 7.2.5 *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 is itself a generic property. Whether or not this is the case is an open problem.

Since the stress test is generic, we are justified in 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.

7.2.1 Finding Stresses

The sufficient condition for unique realizability expressed by Theorem 7.2.3 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 factorization of the rigidity matrix that was required in Section 6.2 to determine redundancies.

Recall the rigidity matrix, M, introduced in Chapter 4 and used again in Chapter 6. It is an $m \times nd$ matrix constructed from a random realization p in $\mathbb{R}^d$.

Element $[e(i,j), di + r]$ is $p_i^r - p_j^r$ if the edge numbered $e(i,j)$ connects vertices i and j, and zero otherwise. Imagine there is a linear combination of the rows of M that sums to zero. If the multipliers in this linear combination are denoted by $\alpha_{e(i,j)}$ for edge $e(i,j)$, then for each $1 \leq i \leq n$ and $1 \leq r \leq d$

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

If we equate $\alpha_{e(i,j)}$ with ω_{ij} in Equation 7.1 we see that the multipliers in the linear combination constitute a stress. So to find a stress we need only find a linear dependency among rows of the rigidity matrix.

Linear dependencies among rows of the rigidity matrix were already computed in Section 6.2. There we looked for columns of M^T that could be removed without changing the rank of M. These columns were identified by performing a QR decomposition of M^T. When a column was found to be dependent we determined which of the basis columns it could replace by solving a triangular system of equations. The solution vector to that triangular system identifies a set of multipliers of the rows of M that sum to zero. In our current language, it gives us 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. Any linear combination of stresses is also a stress. 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. We had excellent success with 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. We recommend a singular value decomposition for its excellent numerical properties. Unfortunately, there is no known sparse implementation of the SVD, so $O(n^2)$ storage will be required. An alternate factorization could save space for large problems, but it is with these large problems that numerical issues become most severe and the stability of the SVD becomes most essential.

Part III

The ABBIE Program

Chapter 8

An Overview of ABBIE

The ideas presented in the preceding chapters have been implemented in a program to solve the three-dimensional molecule problem. It decomposes a large global optimization into a sequence of smaller, more localized problems. The program is named ABBIE in honor of Abbie Hoffman for his admonishment to "think globally, act locally," although it is doubtful he had nonlinear optimization in mind! ABBIE is primarily written in C, with a few FORTRAN subroutines. It is an experimental piece of software developed to test the feasibility of the recursive decomposition ideas. As such, ease of implementation was as important a factor as asymptotic complexity in the selection of algorithms. Our expectations were that the cost of the decomposition would be dwarfed by the cost of the optimization. This topic will be discussed more fully in Chapter 12.

The high level logical structure of ABBIE is outlined in Figure 8.1. ABBIE solves an instance of the molecule problem by a recursive decomposition. It first determines whether or not the input graph has a unique realization. This analysis uses the tools developed in Part II. If there is not a unique answer then maximal subgraphs are identified which have unique realizations.

```
Find maximal uniquely realizable subgraphs
for each such subgraph
    if subgraph is small enough then
        Position graph with global optimization
    else Break into smaller pieces
        for each piece call ABBIE
        Combine pieces with global optimization
return (positioned subgraphs)
```

Figure 8.1: The logical structure of the ABBIE program.

ABBIE positions the vertices in a uniquely realizable subgraph by doing one of two things. If the number of vertices is less than an input tolerance then a global optimization is performed to directly determine the realization. However, if there are too many vertices for a direct optimization to be practical then the graph is further decomposed. This decomposition involves finding a small vertex separator and will be described in Section 11.1. The resulting subgraphs are passed recursively to ABBIE. Although these may not be uniquely realizable, ABBIE will identify and position any of their uniquely realizable components. Once these components are positioned, they are fit together by an additional call to the global optimizer. In this way large optimization problems are avoided.

There are two distinct aspects to this calculation. First is the identification of uniquely realizable subgraphs. This problem was discussed in depth in the preceding chapters. The second element of ABBIE is a technique to actually determine coordinates. Most previous approaches to the molecule problem focused exclusively on this problem. It is our hope that by preprocessing the full problem into a sequence of smaller ones, the burden on the routines that compute coordinates will be substantially reduced. Partly because of this expectation, ABBIE employs a relatively unsophisticated approach to this coordinatization problem. It encapsulates all the distance constraints into a penalty function, which it then tries to minimize. The global minimizer of this function will correspond to a set of vertex locations that satisfies all the constraints.

The various components of which ABBIE is comprised are sketched in Figure 8.2. Each of the boxes in this figure is composed of multiple subroutines, with several hundred in all. As the figure implies, the redundant rigidity and optimization routines are the most complicated. Each of these boxes will be described in detail in the chapters that follow. Chapter 9 will discuss the implementation of the algorithms to identify uniquely realizable subgraphs, which were described in Part II. The global optimization routines that actually compute the coordinates will be covered in Chapter 10. Additional, miscellaneous aspects of the program will be described in Chapter 11, including the technique used to decompose uniquely realizable subgraphs into smaller pieces. Our computational experiences using ABBIE to analyze simulated molecular data will be presented in Chapter 12. Conclusions and open problems will be discussed in the final chapter.

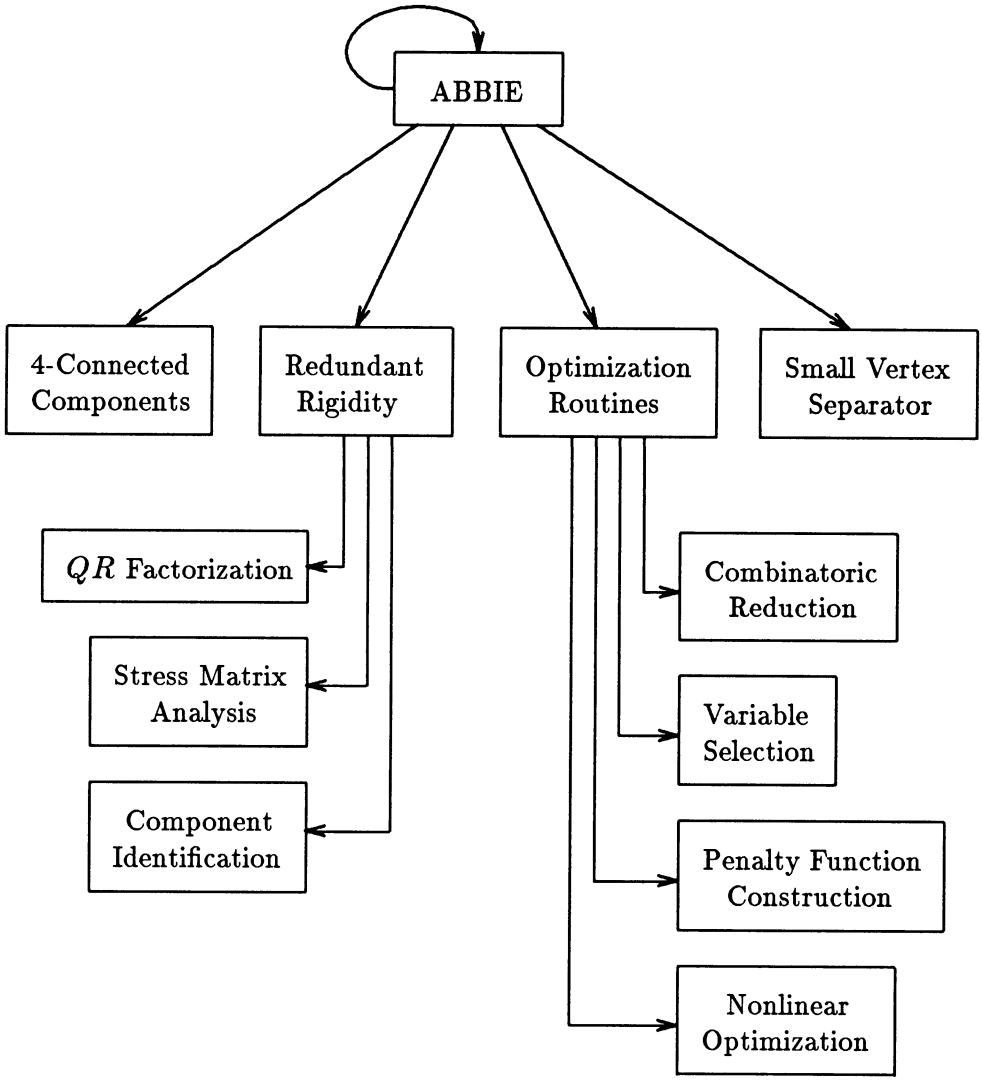


Figure 8.2: A high level breakdown of the routines in ABBIE.

Chapter 9

The Unique Realizability Algorithms in ABBIE

This chapter provides implementation details of the unique realizability algorithms in ABBIE. In Part II, three independent necessary conditions were developed for a graph to have a unique realization in three-space. They were four-connectivity, redundant rigidity, and that the graph not be $K_{5,5}$. A sufficient condition was also developed concerning the nullity of the stress matrix. Algorithms for detecting these properties were described in Part II, and this chapter will provide further details. The first section will describe the four-connectivity routines in ABBIE. Section 9.2 will discuss the implementation of the redundant rigidity algorithm. Finally, the stress matrix analysis will be covered in Section 9.3.

9.1 The Four-Connectivity Algorithm in ABBIE

The four-connectivity algorithm in ABBIE works by looking for sets of three vertices whose removal separates the graph into unconnected pieces. It does this by removing a pair of vertices and running the well known linear time biconnected components algorithm [AHU74]. This is done for each pair of vertices in the graph, giving an overall time complexity of $O(n^2m)$, where n is the number of vertices in the graph and m the number of edges. If a separator is identified, the graph is decomposed into pieces and each piece is recursively tested for four-connectivity. Eventually, maximal four-connected components are returned. Only components with at least four vertices are included. Two maximal four-connected components can only share a maximum of three vertices, so any component with at least four vertices can be identified only once. Also, as will be discussed in Chapter 10, components with fewer than four vertices are inconvenient to piece together.

The results described in Chapter 12 indicate that this four-connectivity calcu-

lation is generally slightly faster than that for redundant rigidity. For this reason the four-connectivity algorithm is always run first, potentially reducing the size of the graphs checked for redundant rigidity. An additional benefit of this order of computations is that the redundant rigidity routines are only invoked with four-connected graphs. As was discussed in Section 7.2, the redundant rigidity calculations generate stresses that can be used in the stress matrix analysis. By performing the four-connectivity calculations first we ensure that the stress analysis will be performed only on graphs that satisfy all of our necessary conditions for unique realizability.

9.2 ABBIE's Redundant Rigidity Algorithm

Given a graph, ABBIE determines if it is redundantly rigid, and if not the program finds maximal redundantly rigid subgraphs. It does this using the randomized matrix approach described in Chapter 6. The fundamental calculation is a QR factorization of the transpose of the rigidity matrix for a random realization of the graph. This factorization allows the determination of redundancy among the rows of the matrix, which identifies redundancy among the corresponding edges. In this way a maximal set of redundant, independent edges can be identified.

As noted in Chapter 4 the rigidity matrix is quite sparse, with only 6 nonzeroes in each row. However, there are several aspects to our application that make this sparsity difficult to exploit. First, most sparse QR approaches discard the orthogonal matrix Q [GN84]. As we shall see in Section 9.2.4 we need to retain Q, as it contains a basis for the infinitesimal flexes. Second, the rows of the rigidity matrix are not linearly independent, and in fact the dependencies reveal which edges are redundant. This implies that we cannot determine the nonzero structure of Q and R beforehand since we will need to do some form of pivoting. The standard approaches that divide the computation into a symbolic phase followed by a numeric phase will not work in this setting.

Third, for problems of interest the graph can have $m \gg 3n-6$, so the transpose of the rigidity matrix can have many more columns than rows. We are interested only in a maximal set of redundant, independent edges, corresponding to a subset of at most $3n-6$ columns. Most sparse QR techniques generate Q as a sequence of Householder or Givens transformations and act the transformations on the entire matrix. In our case we would like to avoid forming the entire matrix, even sparsely. Instead we would like to form it one column at a time, discarding columns that are not independent.

One approach that would avoid these problems would be to perform the symbolic and numeric aspects of the factorization together. At each step the nonzero structure of a particular column could be determined symbolically, and the values computed numerically. However, this determination of nonzero structure on-the-fly would require additional storage, partially mitigating the advantage of exploiting

the sparsity in the rigidity matrix. It would also be time consuming since the ordering of columns would need to be recomputed each time a dependent column is identified and discarded.

For these reasons, as well as for ease of implementation, ABBIE is endowed with a hybrid scheme that makes partial, but not optimal, use of sparsity. ABBIE's QR factorization approach is divided into two phases. In the first phase the rows and columns of M^T are permuted in such a way that much of the subdiagonal is set to zero. More specifically, the leading columns of the matrix are selected to have their nonzero values as near to the top of the matrix as possible. This makes the columns effectively shorter, since no space below their lowest nonzero location need ever be accessed. This saves space by avoiding the allocation of the unneeded region of the matrix, and it also saves time since the Householder vectors become correspondingly shortened and only act on the top rows of the matrix. The heuristic that ABBIE uses to achieve this ordering will be described in Section 9.2.1.

The second phase of ABBIE's QR factorization involves computing and using these shortened Householder vectors in a dense fashion. Except for the initial ordering of columns the sparsity structure of M is ignored. This wastes some amount of space, but leads to a straightforward QR implementation. This computation will be described in detail in Section 9.2.2.

9.2.1 Ordering Heuristic for the QR Factorization

To save space and time in the QR factorization of the transpose of the stress matrix, ABBIE permutes rows and columns to place nonzero values of the leading columns as high as possible, exploiting the special structure of the rigidity matrix. The matrix M^T has m columns, one for each edge in the graph, and nd rows, one for each coordinate of each vertex. Each column, corresponding to an edge (i, j), has 6 nonzero values at the locations corresponding to the coordinates of vertices i and j.

We would like to order the rows and columns of M^T so that there are many short columns and few long ones. Long columns correspond to edges that are incident to the highest numbered vertex. Hence, we want high numbered vertices to be adjacent to as few edges as possible. ABBIE uses a greedy heuristic to achieve this. It assigns the highest vertex number to the vertex incident to the fewest edges. These edges are then assigned the highest edge numbers. The vertex and edges are removed from the graph and the process is repeated.

The degree of each vertex can be determined in $O(m)$ time using a bucket sort, and these values are easily updated as the assignment process proceeds. Hence, the cost of this algorithm is linear in the problem size.

As will be discussed below the full rigidity matrix is never generated in ABBIE. Instead it is formed one column at a time. If a column is found to be linearly independent of its predecessors it is added to the matrix; otherwise it is discarded.

Using this approach the total space needed for the QR factorization is the total space for this maximal set of independent columns. The column ordering heuristic is ABBIE is very effective in reducing this storage space. For the problems described in Chapter 12 the heuristic reduces the space required for the final set of columns by between 28 and 35 percent over a dense approach. The heuristic performed better as the problem size got larger. Note that the best possible improvement would be slightly less than 50% since this space is required to hold the upper triangular matrix R.

9.2.2 Details of the QR Factorization

After reordering rows and columns to partially exploit sparsity, the QR factorization in ABBIE proceeds in a dense fashion on the shortened columns. The algorithm allocates space for one column at a time, multiplies it by Q^T and then decides to discard or retain the column. The orthogonal matrix Q^T is generated and stored as a sequence of Householder vectors. When a new column is allocated it is first multiplied by the existing Q^T. If the resulting subdiagonal portion of the column is nonzero (within a numerical tolerance to be described in Section 9.2.3), the column is accepted and added to the matrix of independent columns. Then the Householder reflection zeroing the subdiagonal of the new column is computed, and stored in the subdiagonal of the column.

However, if the subdiagonal is found to be zero after multiplying by Q^T then this new column is not independent, and it can be expressed as a linear combination of the existing independent columns. As described in Chapter 6, the independent columns that contribute to this linear combination are redundant. Determining which columns contribute involves solving an upper triangular system of equations. The multipliers identified by this solution vector are used in two ways. First, those that are nonzero identify redundant edges. In practice a numerical tolerance is used which will be discussed in Section 9.2.3. Second, as was described in Section 7.2, these multipliers constitute a stress on the framework. Section 9.3 will explain how ABBIE uses these values in its stress matrix analysis.

As was discussed in Chapter 6, in two dimensions detecting a redundant edge identifies a redundantly rigid subgraph. This observation allowed us to skip over some of the edges in the two-dimensional redundant rigidity algorithm. Unfortunately, the same is not generally true in three dimensions as the graph in Figure 4.1 exemplified. However, there are situations in three-space in which redundant edges do identify redundantly rigid subgraphs. Whenever a dependent edge is encountered there is a set of independent edges that contribute to the dependency. In our QR algorithm these edges are identified by their nonzero contribution in the solution of the upper triangular system of equations. If perchance this set of m' redundant edges is incident to n' vertices so that $m' = 3n' - 6$, then a redundantly rigid subgraph has been identified. No additional independent edge can be added

to this subgraph, so it is a waste of time even to try.

ABBIE exploits this fact to speed its redundant rigidity algorithm. Whenever it finds a redundant edge it determines whether the independent edges contributing to the redundancy span a redundantly rigid subgraph. This just involves counting redundant edges, E', and their incident vertices, V', to see if $|E'| = 3|V'| - 6$. If so, two things happen. First, this subgraph is added to the list of redundantly rigid subgraphs. Second, an attempt is made to merge it with other subgraphs in the list to ensure that each subgraph is maximal. This second operation makes use of the following observation.

Theorem 9.2.1 *Let G be a graph whose edges are independent. If two rigid subgraphs of G share at least 3 vertices then their union is rigid.*

Proof: The proof is a simple counting argument implying that if the union graph has n' vertices then it has exactly $3n' - 6$ edges. As this is exactly analogous to the proof of Theorem 4.2.7, the proof is omitted. ∎

The edges counted as contributing to the redundancy are all independent and redundant. Using this theorem we can conclude that any two redundantly rigid subgraphs sharing at least 3 vertices can be merged. So our new redundantly rigid subgraph is compared to the others in the list to check for three common vertices. If they are found, the two subgraphs are merged.

This list of redundantly rigid subgraphs is used to eliminate some new edges from consideration. Before attempting the QR operations on a new column, the corresponding edge is checked against this list. If the edge is contained in one of the subgraphs then it can not possibly be independent and its dependence can not involve any edges that are not already known to be redundant. If this is the case the edge is skipped, and the numerical operations are avoided. For the largest problem described in Chapter 12 this technique avoided 410 of the initial 3292 edges.

There is one serious drawback to this technique. When a redundant edge is found numerically it does more than just identify some redundant, independent edges. It also generates a stress to be used in the stress matrix sufficiency test. When edges are skipped they no longer contribute their stress. This may cause the stress matrix to lose rank, which destroys the utility of the sufficiency test. For this reason ABBIE includes an input option which determines whether or not the edge skipping scheme is activated. If it is, the stress analysis is automatically turned off.

One practical aspect of the QR factorization should be mentioned. The pattern of memory accesses in multiplying new columns by Q^T is highly structured. When Q^T is stored in factored form the Householder vectors are required in sequence. Multiplying a set of vectors by Q^T results in a cycle of memory accesses, looping repeatedly through the Householder vectors. If this algorithm is run on a virtual

memory machine and the set of Householder vectors is too large to fit in core, then this pattern of accesses is very undesirable. The most common heuristic to decide which parts of core should be replaced is the *least recently used* strategy. That is, whichever portions of memory have been untouched for the longest time get swapped out to make room for new values. However, when memory is accessed cyclically the least recently used information is precisely what will be needed next! Under these circumstances the computer will spend much of its time moving information between core and disk, greatly reducing the time it has to do useful work. We observed this in practice for early versions of ABBIE's QR factorization of large problems. For this reason, we opted to run large QR factorizations on a machine with a sufficiently large memory to keep the entire set of independent columns in core.

9.2.3 Numerical Tolerances

There are two different situations in the QR factorization in which ABBIE needs to decide whether or not a numerical value is zero. In exact arithmetic this is easy, but in the presence of finite precision it can be problematic. An understanding of the propagation of roundoff error is necessary. The initial values in the rigidity matrix are the differences between coordinates of a random realization. The coordinates have values randomly selected between zero and one, so both absolute and relative errors in the matrix have a magnitude roughly equal to the machine precision ϵ.

The first situation requiring a test for zero values occurs when a new column is checked for independence. After multiplying the new column by Q^T the 2-norm of the subdiagonal is computed, and checked for a value of zero. This is essentially a value in the upper triangular matrix R. The analysis of Stewart, [Ste77], indicates that since the error in the initial rigidity matrix, M, is small, in the Frobenius norm the error in the computation of R relative to M is bounded by $3\epsilon\kappa_F(M)$. In this expression $\kappa_F(M)$ is the Frobenius condition number of M defined by $||M||_F||M^\dagger||_F$, where $\|\cdot\|_F$ denotes the Frobenius norm and $M^\dagger = (M^TM)^{-1}M^T$. is the pseudo-inverse of M. In our case we are interested in the error in a single term in R not in its norm, but the functional dependence should be roughly the same.

For our calculation we are discarding columns that are not linearly independent. When working on a particular column the only relevant portion of M is the set of independent columns that has been identified so far. Since orthogonal transformations do not alter the Frobenius norm of a matrix, the Frobenius condition number of the set of independent columns can be expressed as $||R'||_F||R'^{-1}||_F$, where R' is the portion of the upper triangular matrix generated by the current set of independent columns. This gives us a bound on the roundoff error of $3\epsilon||R'||_F||R'^{-1}||_F$.

Since a dependent column can show up at any time we always need to keep a value for the condition number of the portion of R that has been generated so far.

Since it is generated one column at a time we would like an incremental approach that allows us to update our value as each new column is added. A condition number estimator that accomplishes this in the 2-norm has been described by Bischof and implemented in ABBIE [Bis88]. To update the condition estimation it requires a linear amount of time, so all the estimates combined require $O(n^2)$ time. The 2-norm can be used to bound the Frobenius norm since for any $k \times k$ matrix A, $||A||_2 \leq ||A||_F \leq \sqrt{k}||A||_2$. This implies that $\kappa_2(A) \leq \kappa_F(A) \leq k\kappa_2(A)$, where $\kappa_2(A)$ denotes the 2-norm condition number of A, $||A||_2||A^{-1}||_2$. As a compromise between these two bounds ABBIE estimates the Frobenius condition number of R' by $\sqrt{k'}\bar{\kappa}_2(R')$, where $\bar{\kappa}_2$ is the estimate of the 2-norm condition number given by Bischof's technique, and k' is the current number of independent columns.

Bringing this all together, for determining whether a column is accepted or rejected ABBIE uses a tolerance value of $\alpha\epsilon\sqrt{k'}\bar{\kappa}_2(R')$, where α is a small constant. A value for α of 30 was used in the calculations described in Chapter 12.

The second situation in the QR factorization where ABBIE needs a tolerance value occurs when redundant columns are being identified. A nonzero contribution to a linear combination indicates redundancy, and this potentially nonzero value occurs in the solution of a triangular system. The triangular matrix, R', is a leading set of columns of R. The roundoff error in finding the solution to this linear system can be estimated as $\epsilon\kappa_F(R')$. This allows us to use the same approximation as above, namely that the roundoff error can be expressed as $\beta\epsilon\sqrt{k'}\bar{\kappa}_2(R')$, where β is a small constant. For the calculations in Chapter 12 a value for β of 6 proved acceptable.

9.2.4 Finding Redundantly Rigid Components

In Section 9.2.2 we described how the QR factorization can be used to identify a maximal set of independent, redundant edges of the graph. If there are $3n - 6$ such edges then the graph is redundantly rigid. Otherwise we wish to identify subgraphs that could be redundantly rigid. To accomplish this we need to make several observations. First, any edge that is not redundant in the original graph will not be redundant in any subgraph. Discarding these edges will increase the space of flexes of the graph, but it will not alter the redundantly rigid subgraphs. After discarding these unhelpful edges the remaining independent edges are all redundant. The flexes that remain will not affect the redundantly rigid subgraphs. To identify these subgraphs we first need to generate a basis for these flexes.

This basis can be generated by a QR factorization of the columns of the transpose of the rigidity matrix that correspond to the redundant, independent edges. ABBIE performs this calculation in the same way the full rigidity matrix was factored. By keeping these columns in their original order the existing data structure for the factorization can be used without requiring any additional space. To remove the trivial motions from the basis ABBIE explicitly adds 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 the final $3n-6-k$ columns of Q are a basis for the flexes. The matrix Q is stored in factored form, but these columns can be explicitly formed by multiplying the final $3n-6-k$ columns of the identity matrix by Q.

These flexes are the infinitesimal motions of the framework that are not constrained by redundant edges. Sets of vertices whose relative positions remain unchanged under these flexes are potentially redundantly rigid subgraphs. Identifying these sets 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 dot 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 the location of i. If this quantity is zero then the distance between i and j remains unchanged.

In practice this quantity is rarely zero due to numerical roundoff in the QR factorization, so some tolerance is required. Note that the the error in the vertex locations is the machine precision, ϵ, but the flexes are columns of Q with larger errors. The analysis of Stewart [Ste77] provides a bound on the Frobenius norm of the error in the calculation of Q roughly as $3\epsilon\kappa_F(M)$, where $\kappa_F(M)$ is as described in Section 9.2.3. This is an absolute, not a relative error. As in Section 9.2.3 ABBIE approximates $\kappa_F(M)$ by $\sqrt{k}\bar{\kappa}_2(R)$. The error in the calculation of the flex can then be bounded by $\gamma\sqrt{k}\bar{\kappa}_2(R)|p_i-p_j|_2$, where γ is a small constant. This is the expression ABBIE uses to identify zero values. For the calculations described in Chapter 12 a value for γ of 15 was used.

Pairs of vertices whose distance doesn't change under any of the allowed flexes could just as well be connected by an edge. We will consider such vertices to be joined by an *induced edge*. Finding sets of relatively fixed vertices corresponds to finding cliques, subgraphs with all edges, 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.

This greatly simplifies the identification of maximal cliques in the graph of induced edges. Begin 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.

The algorithm in ABBIE is somewhat better than this. The point of this decomposition is to find subgraphs that have unique realizations. Being a clique in the graph of induced edges is a necessary, but not sufficient condition. In particular, if there are many induced but few real edges in the subgraph it may not be redundantly rigid when isolated from the rest of the full graph. ABBIE

only tries to identify uniquely realizable subgraphs with more than four vertices. The identification of smaller subgraphs will be described below. Any uniquely realizable graph with more than four vertices has no vertex of degree less than four. The algorithm in ABBIE to find potentially redundantly rigid subgraphs takes advantage of this fact. Instead of looking only at the graph of induced edges it works with the original graph. It looks for a vertex with four real neighbors all of whom have induced edges between them. This set of five vertices is enlarged to a maximal set by looping through the remaining vertices and checking them for three induced edges into the starting set. If the maximal degree of a vertex in the original graph is δ then this algorithm requires $O(mn\delta^4)$ time. For the problems described in Chapter 12, δ is always between 13 and 17. For molecular conformation problems in general δ will always be limited by the number of atoms that can be located in a small region of space. This limit will be independent of the size of the problem, so the algorithm in ABBIE really requires $O(mn)$ time.

One more aspect of the redundantly rigid component calculation needs to be mentioned. The theorems in Chapter 6 identify redundant rigidity as a necessary condition for unique realizability when $n > d + 1$. For our present purposes this implies that subgraphs that are k-cliques for $k \leq 4$ may not be identified as unique because their edges may not be redundant. In the design of ABBIE it was deemed unnecessary to worry about subgraphs of size three or smaller. However, cliques of size four are desired. For this reason, after completing the calculation of redundantly rigid components described above ABBIE looks through the graph for four-cliques that are not already in a component.

As will be discussed in Chapter 12, the cost of the entire component finding process, including the search for four-cliques, is negligible compared to the cost of the QR factorizations.

9.3 The Stress Matrix Analysis

As discussed in Section 7.2 the rank of the stress matrix can provide a sufficient condition for unique realizability in three dimensions. If for a generic realization there is a stress in which the nullity of the stress matrix is 4 then all generic realizations are unique. When the QR factorization approach to redundant rigidity is used, a stress is generated each time a redundant edge is identified. The solution vector to the triangular system of equations constitutes a stress. When the stress option is selected by the user, ABBIE sums all of these stresses, scaling each of them by a random multiplier between one half and one. Note that as mentioned in Section 9.2.2, ABBIE is capable of skipping columns in the QR factorization that can not possibly be independent. However, their absence can affect the nullity of the stress matrix, So whenever the column skipping option is invoked ABBIE automatically disables the stress analysis.

The stress matrix is a sparse, symmetric matrix of size $n \times n$. Each row and

column corresponds to a vertex in the original graph. Nonzero values in the matrix occur along the diagonal and at locations $[i,j]$ for each edge (i,j) of the graph. The total number of nonzeros in the stress matrix is $2m+n$.

When the stress option is invoked, ABBIE constructs and determines the rank of the stress matrix. To make this determination ABBIE computes a singular value decomposition of the stress matrix using the appropriate FORTRAN routines from LINPACK [DMBS79]. The shortcomings of the SVD that made it inappropriate for the analysis of the rigidity matrix are less of a problem in this setting. The primary difference is that the stress matrix is only about one ninth as large as the independent columns of the rigidity matrix so sparsity considerations are less paramount. In fact, a dense factorization of the stress matrix will always use less space than the semi-sparse QR factorization of the rigidity matrix described in Section 9.2. In addition, the numerical stability of the SVD is essential in the stress analysis. The elements in the stress matrix are the result of the solution of a linear system of equations, and so they already have some modest roundoff error. The stability of the SVD limits the further growth of this error.

The rank of the stress matrix is determined by the number of nonzero singular values. Because of roundoff errors a small tolerance is required in determining which values should be considered zero. The LINPACK users guide [DMBS79] recommends using a tolerance equal to the errors in the initial elements of the matrix. For the stress matrix these values are determined by solving linear systems of equations. As discussed in Section 9.2.3, the errors in these values can be approximated by $\beta\epsilon\sqrt{3n-6}\bar{\kappa}_2(R)$, where β is a small constant. A value of 10 for β seemed to work well in practice.

For the calculations described in Chapter 12 the stress matrix analysis was not used. This sped up the calculations in two ways. First, the cost of the SVD was avoided, and second the edge skipping technique described in Section 9.2.2 could be used. The obvious danger of not invoking the sufficiency test is that a subgraph that passes the necessary tests might actually have multiple realizations. When its coordinates are determined the wrong realization might be generated, which would be unable to fit with the remainder of the graph. There are two reasons why this is not a severe problem. First, if a subgraph is positioned incorrectly the problem will be detected by the later inability to fit subgraphs together. In this way we detect nonuniqueness without having to perform the stress test. The second, and more substantial reason why skipping the stress test is acceptable is that the necessary conditions seem to be quite restrictive. For all the problems described in Chapter 12 and all the subgraphs that were generated, none encountered this problem. It seems that in practice the necessary conditions are usually sufficient. Graphs other than $K_{5,5}$ that pass the necessary tests while still not being uniquely realizable are rare, if they exist at all.

Chapter 10

Optimization Routines in ABBIE

Any program that solves the molecule problem eventually has to assign coordinates to vertices. The combinatorial analysis described in Part II 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. The two objects are vertices, and subsets of vertices whose relative positions have already been determined. Each of these subsets, or *chunks*, consists of a subgraph along with a set of coordinates that describes the relative locations of all the vertices in the subgraph. These chunks can be treated as rigid bodies with at most six degrees of freedom in three-space.

ABBIE solves these problems using a two-phase approach. In the first phase the program uses a combinatoric 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. This position can be determined and the vertex added to the chunk. This combinatoric phase will be described in greater detail in Section 10.1.

After exhausting its bag of combinatoric 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. The location of any of the vertices can be determined as function of these parameters. ABBIE then constructs a cost function that penalizes a realization for not satisfying an edge length constraint. This penalty is expressed as a function of the rotational and translational variables. The sum of all these 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. The locations of all the vertices now constitute a satisfying realization. Details of this nonlinear optimization will be given in Section 10.2.

More sophisticated techniques to solve this positioning problem are possible. In fact, most previous approaches to the molecule problem focus on this aspect, as was described in Chapter 2. ABBIE was developed to test the feasibility of our divide-and-conquer approach to the molecule problem. Our expectations were that the overall approach will be successful, even though the optimization techniques are rather simplistic.

10.1 Combinatoric Positioning Techniques

To reduce the number of variables in the global optimization ABBIE first tries to combine vertices and chunks combinatorically. That is, it looks for small numbers of chunks and vertices that can be easily combined into a larger chunk. For this portion of code, asymptotic complexity of the algorithms was not considered to be important for two reasons. First, the number of objects being pieced together is typically quite small. Second, the computational payoff of a reduction in the size of the optimization problem was expected to more than compensate for the cost of performing this analysis. The results in Chapter 12 will bear out this expectation.

To reduce the degrees of freedom left in the problem ABBIE tries to enlarge chunks by either adding single vertices to them or by combining them with other chunks. If presented a problem with no chunks (which will happen at the lowest depth of the recursion) ABBIE tries to create chunks by looking for cliques of size four. If it finds any then it forms them into chunks and proceeds as normal. Otherwise it looks for cliques of size three and forms these into chunks. However, chunks of size three can not be enlarged by the techniques encoded in ABBIE.

ABBIE has five different heuristics for enlarging chunks. The success of these techniques depends upon specific sets of vertices not being coplanar. This is guaranteed by the assumption that the final solution is generic. For the first technique, if two chunks have at least 4 vertices in common then they can only be combined in one way, and ABBIE can merge 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 it can use 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.* 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.* The last three heuristics for enlarging chunks make use of this observation. Technique number three 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.

ABBIE tries to apply these techniques to all combinations of chunks and vertices until no more merging is possible. Additional, more complicated heuristics are possible. As the numerical results in Chapter 12 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 by the reduction in size, and consequently cost, of the optimization problems.

10.2 ABBIE's Nonlinear, Global Optimizer

If the combinatoric approaches fail to position the entire graph, ABBIE resorts to a nonlinear, global optimization. There are two distinct aspects to this calculation. First, the program must select an appropriate set of variables to describe the allowed locations of all the chunks and vertices left in the problem. This part of the calculation is purely combinatoric. Second, the program devises a continuous, nonlinear function that penalizes a realization for not satisfying all the constraints, and tries to minimize it. The process of variable selection will be described in Section 10.2.1 and the global optimization in Section 10.2.2.

10.2.1 Selecting Optimization Variables

The optimization problems ABBIE must solve involve fitting together sets of chunks and vertices. Vertices have three translational degrees of freedom. The location and orientation of a chunk can be described by three translational and three rotational variables. For angular parameters ABBIE uses a system similar to quaternions [Sal79]. That is, a rotation is defined by a direction about which to rotate, and an amount of rotation. However, instead of using three parameters to define the direction, ABBIE uses a standard (θ, ϕ) system. This representation has fewer singularities than the more familiar Euler angles. A solution to an optimization problem is a set of translational and rotational parameters that move the

vertices and chunks into positions in which all the distance constraints are satisfied. This calculation is performed by minimizing a penalty function that expresses all of the constraints in terms of the translational and rotational variables.

There are many possible ways to parameterize the motions of the vertices and chunks. The selection of optimization variables in ABBIE was designed to minimize the computational effort required by the global optimization. This global optimization technique will be described in detail in Section 10.2.2, but it is an iterative procedure, selecting random starting points and finding nearby local minimizers until a global optimum is found. There are two factors which determine the total cost of this process, 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 an iterative approach, computing a local second-order-accurate approximation to the penalty function at each iteration. This approach requires the repeated formation and factorization of the Hessian, the matrix of second derivatives. ABBIE's penalty function will be described below, but its formation is relatively inexpensive, roughly proportional to the number of distance constraints, which is typically linear in the size of the optimization problem. The factorization of the Hessian tends to dominate the cost of each iteration, requiring $O(q^3)$ 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. But since we know each iteration costs $O(q^3)$ it is reasonable to assume that the cost of each local minimization is roughly 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. We saw in Chapter 2 that the number of local minimizers can grow exponentially with the number of vertices. Thus, 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, but an appropriate value for this parameter depends on the class of problems under consideration.

There is one more important 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. There is no continuous rigid body motion to transform between these two realizations. We can not know in advance which of these two *parities* is the correct one, so we have to try them both. 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. It is somewhat disconcerting that an exponential term appears in our estimated cost in such an unavoidable way. But since the molecule problem is NP-hard, there will always be problems that require an exponential amount of computation.

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

It is not always helpful to include a chunk in the optimization. For instance, if all the vertices in our chunk are already contained in other chunks, then including our chunk will add unnecessary variables to the optimization. Instead, it is better to discard our chunk and add its constituent edges to those in the penalty function. More generally, discarding a chunk may free some vertices, each of which adds three variables to the optimization. ABBIE decides whether to accept or discard a chunk depending on which option generates a smaller estimated optimization cost.

One chunk, called the *base chunk*, is held fixed at the origin to remove translational and rotational ambiguities. All the chunks are considered as candidate base chunks, and the one which minimizes the cost function is selected. If a second chunk shares three vertices with the base chunk then it has no continuous degrees of freedom. Its parity contributes a factor of two to the optimization cost, but it adds no new variables. 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. Hence, the selection of a base chunk that shares vertices with other chunks leads to a reduction in the number of optimization variables, and a corresponding decrease in the cost of the global optimization.

In evaluating the candidate base chunks ABBIE adds the remaining chunks in a greedy manner, trying to minimize the estimated optimization cost. At each step in this procedure ABBIE selects the remaining chunk with the largest number of vertices that are not yet contained in an accepted chunk. The relative costs of accepting or rejecting this chunk are computed, using the estimation of optimization cost developed above. If the chunk is accepted it increases the number of chunks by one, and it can increase the number of variables depending on how many vertices it shares with the base chunk. 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. The chunk is accepted if this leads to a lower approximate estimation cost. 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. Each of the chunks in turn is analyzed as a candidate base chunk, and ABBIE selects the one which minimizes the estimated cost function. The corresponding variables are used in the global optimization.

There is one additional special case that needs to be described. If presented an optimization problem with no chunks, and no cliques of size four or three ABBIE has no chunks from which to select a base chunk. For example, a bipartite graph contains no cliques of size 3 or larger. Without a base chunk it is more difficult to get rid of translational and rotational ambiguities. In this case ABBIE selects an edge to place on the x-axis, but this leaves a rotational ambiguity. So ABBIE selects a third vertex to be constrained to lie in the x-y plane. This affects the calculation of derivatives in the optimization procedure.

10.2.2 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 an especially simple one. For each edge (i, j) that needs to be satisfied the program computes a value $P_{ij}^1 = (|p_i - p_j|^2 - d_{ij}^2)^2$, 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 $F_1 = \sum_{(i,j)} P_{ij}^1$, where the sum is taken over all edges in the graph which aren't contained in any of the accepted chunks.

Positioning problems in ABBIE can involve multiple chunks other than the base chunk which share one or more vertices. These vertices must be made to coincide in a satisfying realization. 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 $P_{ij}^2 = |p_i - p_j|^2$. Summing all of these types of constraints together gives F_2, a second component to the cost function. The full penalty function is then $F = F_1 + F_2$.

One word of caution is in order. The full penalty function is composed of 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. It then runs a local minimization routine to find a local minimizer. 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. The probability that this process will find the global minimizer goes to 1 as the number of trials goes to infinity.

A random starting point is composed of random values for the angle and translation parameters that comprise the vector of variables, as well as a random parity for each chunk. As discussed in Section 10.2.1, the variables describe locations relative to a fixed base chunk. Any vertices that have edges to this chunk are initially

placed at a random point that satisfies these edge constraints.

For local optimization ABBIE uses a modified version of the trust region code, NTRUST, developed by Danny Sorensen. A trust region technique is an iterative procedure that tries to converge to a local minimizer. At each point it evaluates the function, its gradient and its Hessian. It uses these to construct a local quadratic model to the function. It then tries to move to a new point that approximately minimizes this model, while still staying within an elliptical region in which the model is deemed trustworthy. It is this latter constraint that gives the technique its name. The size of the trust region can shrink or grow depending on how well the estimated function values agree with the true values. There are many possible variants of trust region techniques with respect to stopping criteria or adjusting the size of the ellipse or the calculation of the Hessian. For more details on trust region techniques the reader is referred to any good text on optimization, including [DS83,Fle87].

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. One aspect of NTRUST is clearly not optimal for ABBIE. NTRUST computes and factors the Hessian matrix densely, but the Hessians generated in ABBIE are typically sparse. This is mitigated by the decomposition approach in ABBIE. Large problems are avoided, along with their large Hessians, which reduces the inefficiency of treating the matrix densely. Except for the singular value decomposition routines described in Section 9.3, NTRUST and its subroutines are the only part of ABBIE that is written in FORTRAN.

The main modification that needed to be made to NTRUST was to add the capability to scale variables. Without variable scaling the trust region is simply a sphere. Our optimization problems include both translation and rotation variables. The angular parameters have a range of values of π or 2π, but the translation variables can have a much different range of values, depending on the unit of length in the problem. For this reason it was inappropriate treat the angular variables as equivalent to the translational ones. For the problems to be described in Chapter 12 the range of translational values is about 70. After some experimentation, we decided to scale angular variables by a factor of 10 relative to translations.

Chapter 11

Additional Features of ABBIE

This chapter describes features of ABBIE that don't logically belong in either of the previous two chapters. Some of these capabilities are central to the algorithm, while others simply provide greater flexibility for the user.

11.1 ABBIE's Decomposition Heuristic

The high level description of ABBIE presented in figure 8.1 contained a feature that hasn't yet been discussed. If the program identifies a uniquely realizable graph it can position the graph immediately. But if the graph is large, this direct approach may be infeasible. Instead, if the graph has more than a user specified number of vertices it is broken into smaller pieces which are passed recursively to ABBIE. Once these pieces have been processed, they are combined together by an additional call to the global optimization routines. This section will describe how the graph is broken into pieces.

In selecting a value for how large a graph 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 Chapter 12 a cutoff of 15 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. As the decomposition proceeds it is important to discard as few edges as possible. When the 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 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. This procedure tries to identify a small set of vertices whose removal separates the graph into components in such a way that no edge connects the different components. When the separator set is added to each component no edges are lost.

Small vertex separators are also needed in nested dissection approaches to sparse matrix factorization, so heuristics to find them have been developed. ABBIE contains an algorithm described by Liu [Liu88]. This algorithm has two phases, the first to obtain an initial separator, and the second to improve it. The initial phase uses a minimum degree ordering, a technique originally developed to reduce fill in sparse matrix computations. The vertex with the fewest neighbors is selected first. It is removed from the graph and edges are added between every pair of its neighbors. Then the process is repeated on the resulting smaller graph. At any stage in this algorithm, the set of neighbors of a vertex being removed constitute a separator of the initial graph. We followed Liu's suggestion and selected as an initial separator a neighbor set whose removal generated a connected subgraph with just under half of the total number of vertices.

This initial separator is then improved using a bipartite matching approach. If some subset of the separating vertices is adjacent to a strictly smaller subset of vertices in the largest component, then the subset of separating vertices can be replaced by their neighbor set to yield a smaller separator. This process is repeated until no more local improvements can be found.

As we will discuss in the next chapter, this approach proved to be quite successful at finding small vertex separators for problems of interest.

A reasonable alternative approach, not used in ABBIE, would be to divide the graph by identifying a small number of edges whose removal disconnects the graph. Finding small edge separators is an important problem in designing circuit layouts, so efficient heuristics have been developed [KL70]. The advantage of this approach over vertex separators is that the sum of the vertices in the subgraphs is exactly the total number of vertices. This leads to modestly smaller subproblems than the technique in ABBIE, since with vertex separators the separating set of vertices is added to both subgraphs. The disadvantage of using edge separators is that the separating edges are lost to the subproblems.

11.2 The Restart Capability

Solving large molecule problems can be computationally intensive. The sequence of graph algorithms and global optimizations can be time consuming, and the rigidity matrix calculations can require large amounts of space. For these reasons a restart

capability was added to ABBIE. As it proceeds with its calculations the program writes its intermediate results to an output file. By reading this file, a second run of the program can pick up where the first left off without duplicating all the computational effort.

This restart capability provides three main benefits. First, if the optimization routine fails to find the global minimizer within its allowed number of attempts the program halts. By adjusting the random number seed at the appropriate point in the restart file a second run can be started that continues looking for the global minimizer without repeating all the preceding calculations.

The second main benefit has to do with space. As discussed in Chapter 9, if the graph is large, the rigidity matrix calculations can require large amounts of space. This large matrix is only needed in the early phases of a calculation, and no other routines in ABBIE require large amounts of memory. So once the molecule has been decomposed into smaller pieces the space requirements of the program are modest. Although the initial phase of a problem may run most efficiently on a large computer with a large core, later phases will fit easily on smaller, less expensive machines. The restart capability allows the user to take advantage of this fact. The problem can be run on a large machine, generating a restart file, until the molecule has been broken into suitably small subproblems. Then the calculation can be resumed on a smaller machine.

In our case the large machine was a Sun4, with 32 megabytes of memory. The smaller machines were Sparcstations with 16, 12 or 8 megabytes. More details will be provided in Chapter 12.

The third use of the restart capability occurs when the molecule does not have a unique solution, but the user wishes to examine the set of conformations consistent with the constraints. In this case ABBIE can first compute the relative coordinates of all the uniquely realizable subgraphs. This information can be written to the restart tape and used as input to a series of runs that combine these pieces while using different random number seeds. In this way, the full conformation space can be explored while the unique pieces only need to be generated once.

11.3 Exploiting Protein Structure

ABBIE was developed to help solve problems of molecular conformation. A very important class of biological molecules amenable to this type of analysis is proteins. Proteins are composed of a linear sequence of relatively simple amino acids, and the conformation of each amino acid is well known. ABBIE can use this fact to preprocess the input graph, simplifying its calculations.

Since the conformation of an amino acid is known, all pairwise distances between atoms are determined. As a graph, each amino acid is a clique, a graph with maximal edges. So the graph of a protein consists of a set of cliques which share no vertices. If we know the locations of four noncollinear vertices in a clique it is

trivial to position the remainder. ABBIE exploits this observation to reduce the size of the graph it analyzes. If a vertex in a clique of size five or greater has no edges to vertices not in the clique, then the vertex is discarded. After the resulting, smaller graph is positioned the discarded vertices are added back in. In this way the size of the graphs that ABBIE works with can be substantially reduced. For the largest problem described in Chapter 12, the number of vertices can be reduced in this way from 1849 to 777.

ABBIE takes advantage of the structure of proteins in an additional way. Each amino acid can be considered as a rigid body. To combine with the remainder of the protein in a unique way an amino acid must satisfy several conditions. First, it must have more than 3 vertices with edges to the remainder of the protein. Otherwise the full protein is not four-connected. Similarly, an amino acid must have at least 7 total edges to the rest of the molecule. If not, the graph cannot be redundantly rigid. If an amino acid fails either of these simple tests it is identified as a separate component which cannot be uniquely combined with the remainder of the protein. The uniquely realizable components routines would eventually perform this decomposition, but by this simple screening procedure they are saved the effort. For the largest problem discussed in Chapter 12 this reduced the size of the graph from 777 vertices to 698.

The recursive decomposition ideas enshrined in ABBIE were developed with protein structure in mind. As the chain of amino acids folds back upon itself we expect portions of the molecule to have many edge constraints. ABBIE was designed to identify and take advantage of this occurrence. The validity of these expectations will be confirmed in Chapter 12.

11.4 Space Saving Techniques

ABBIE doesn't know in advance how large a problem it will be asked to run, so it needs to allocate space as it proceeds. The program uses several techniques to conserve space and to reduce the number of number of times it needs to ask the operating system for more memory. Two of these techniques are described here.

One of ABBIE's most important data structures is a graph. This is stored as a linked list of edges for each vertex. Each edge contains information about which vertices it connects and what distance it is supposed to attain. As a calculation proceeds an initial graph is recursively broken into a sequence of subgraphs. This multiplicity of subgraphs has the potential to require large amounts of memory. To mitigate this problem ABBIE makes use of a simple observation about a subgraph. Every edge in a subgraph was initially in the original graph. Hence, instead of creating new space for all the edges in the subgraph, ABBIE uses the space already allocated for the old graph. When it finishes working with the subgraph the program reconstructs the original graph, returning its borrowed edges.

A ubiquitous data structure in ABBIE is the linked list of integers. This structure is used to keep track of the subgraphs that each vertex is in, and the set of vertices in each subgraph, as well as many other properties. The basic element of a linked list is a list element that contains a value and a pointer to the next list element. Instead of allocating and then freeing a list element each time one is used, ABBIE recycles elements. The program keeps a list of allocated, but temporarily unneeded list elements and if a routine needs a new one it is taken from this list if possible. In this way the number of expensive allocation calls is substantially reduced. If, however, the program ever needs more space than the operating system is able to provide, it frees all the space used by this list of unneeded elements and attempts the allocation again.

11.5 Output Options

ABBIE can be run in several different modes. In the first mode it merely determines if the graph is uniquely realizable, and if not it identifies maximal uniquely realizable subgraphs. In this mode it never attempts to compute coordinates, so it runs relatively quickly. This allows the user to see how a full run would proceed, as well as directing further distance measurements if the problem does not have a unique solution.

In its second mode ABBIE will position whatever portions of the graph have unique solutions. This may be the entire graph, or it may be some set of subgraphs. But if the entire problem does not have a unique solution the program doesn't attempt to fit the subgraphs together. This option is provided for two reasons. First, this partial solution may be sufficient. If a chemist is interested in a particular region of the molecule, a full realization may not be necessary. Second, when combined with the restart option this mode prepares the user to explore the solution space of the full problem. The restart file of this run can be used with different random number seeds as inputs to ABBIE runs in the third mode. In this third mode the program will find a single solution to the problem, even if there is not a unique answer.

Chapter 12

Results

ABBIE has been used to analyze simulated molecular data provided by Kate Palmer, from the chemistry department at Cornell University [Pal90]. The data set 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 consequences of this distinction will be discussed more fully in the next chapter. The molecule that generated the distances was ribonuclease, a protein consisting of 124 amino acids and, after discarding end chains, 1849 atoms. The three-dimensional conformation of ribonuclease is known, so all the 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 later could be measured by two-dimensional NMR spectroscopy experiments. All together this made for a total of 15,046 edges, implying an average degree of just over 16.

A single problem would give only limited insight into the strengths and weakness of ABBIE, so we desired a set of related test problems of varying sizes. To generate these we simply extracted leading subchains of amino acids from the ribonuclease. The six different problem sizes we used are presented in Table 12.1. The second column presents the number of vertices and edges in the initial, unadulterated graphs. The other columns will be explained shortly.

In Section 11.3 we described how ABBIE can exploit protein structure to preprocess the input, reducing the size of the graph to be analyzed. First, ABBIE attempts to set aside superfluous atoms from each amino acid. The calculation proceeds without these atoms, and they are trivially added back in at the very end of the computation. Using this technique, ABBIE reduced the size of the full ribonuclease graph from one with 1849 vertices to one with 777, held together with 3504 edges. The corresponding values for the other problems are presented in the

Reduced Graph column of Table 12.1. The quick check described in Section 11.3 for identifying amino acids without enough edges to be attached uniquely further reduced the largest graph to 698 vertices and 3292 edges. This is the graph that is passed to the unique realizability algorithms. The sizes of these starting graphs are given in the third column of the table for the different test problems. The final column presents the size of the largest uniquely realizable subgraph that was found within each of the starting graphs.

Table 12.1: Sizes of the test problems; vertices (edges).

Amino Acids	Initial Graph	Reduced Graph	Starting Graph	Largest Unique Subgraph
20	292 (2263)	102 (336)	63 (236)	57 (218)
40	604 (4902)	236 (957)	186 (828)	174 (786)
60	902 (7264)	362 (1526)	310 (1392)	287 (1319)
80	1193 (9556)	480 (2006)	405 (1804)	377 (1719)
100	1491 (12038)	599 (2532)	504 (2272)	472 (2169)
124	1849 (15046)	777 (3504)	698 (3292)	695 (3283)

The problems were run on several different Sun Sparcstations and a Sun4. Runs that were begun on one machine were sometimes continued on another. In particular, the early phases of large problems required large amounts of memory for the QR factorizations, and they were able to use the 32 megabytes of memory possessed by the Sun4. Later phases were then run on smaller, less busy, and faster Sparcs. This multiplicity of machines makes the issue of comparing running times somewhat problematic. To deal with this problem, identical small problems were run on each of the different machines to get rough comparisons of the different running times of the various portions of ABBIE on the different computers. These comparisons indicated that the speeds of the different machines were within 30% of each other, so the raw CPU timings give reasonable order of magnitude numbers. To improve on this, all the timings presented in this chapter have been normalized to adjust for the differences in machine speeds. All the values are for approximate CPU time on a Sparcstation 1^+.

The second output mode described in Section 11.5 was used in all the runs. In this mode, ABBIE identifies and determines coordinates for all the uniquely realizable portions of the graph, but does not try to join them since they can not be combined uniquely. The runs used the edge skipping technique in the redundant rigidity algorithm as described in Section 9.2.2. This led to faster execution times, but it automatically disabled the stress matrix analysis. Hence, the subgraphs identified by the necessary conditions could not be confirmed as being uniquely realizable. This could lead to incorrect coordinates being computed for the subgraph. However, this would be detected when the subgraph would be used in later

optimizations. It would not be able to fit properly with the remainder of the full graph, so the optimization would fail to converge. The fact that this never happened in our test problems is encouraging evidence that there are not many graphs that satisfy our necessary conditions while still having multiple realizations.

12.1 The Unique Realizability Algorithms

ABBIE's algorithm for finding four-connected components was described in Section 9.1. When invoked with a graph with n vertices and m edges it requires $O(n^2m)$ time. Our problems are sparse, with approximately $O(n)$ edges, so the algorithm should run in time approximately proportional to the cube of the number of vertices. In the course of an ABBIE run, the four-connectivity algorithm is invoked multiple times as new, smaller subgraphs are generated. The first invocation should require the most time, as it is the largest. A plot of total time spent computing four-connected components as a function of the starting graph sizes is presented in Figure 12.1 for the six different problems, and the values are given in Table 12.2. The near linear results on a log-log plot indicates that the time is growing roughly as the number of vertices to some power, and a quick calculation reveals this power to be slightly larger than three.

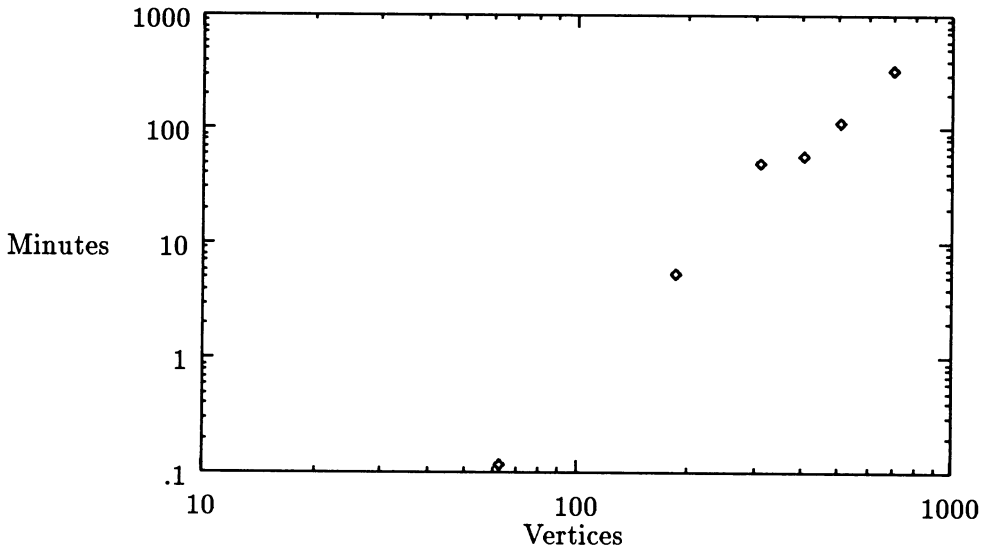


Figure 12.1: Four-connectivity time versus starting graph size.

Similar results apply to ABBIE's redundant rigidity algorithm. The dominant step in this calculation is a QR factorization that requires $O(n^2m)$ time. As discussed in Section 9.1, this calculation is always performed after the decomposition into four-connected subgraphs, so it involves slightly smaller graphs. This

Table 12.2: 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

algorithm is invoked repeatedly with smaller and smaller subgraphs, but as with four-connectivity, the first and largest invocation takes the bulk of the effort. The total time spent in the redundant rigidity routines as a function of starting graph size is shown in Figure 12.2, and in Table 12.2. As above, the results indicate that the cost of the computation grows approximately as a constant power of the problem size, with the exponent slightly larger than 3.

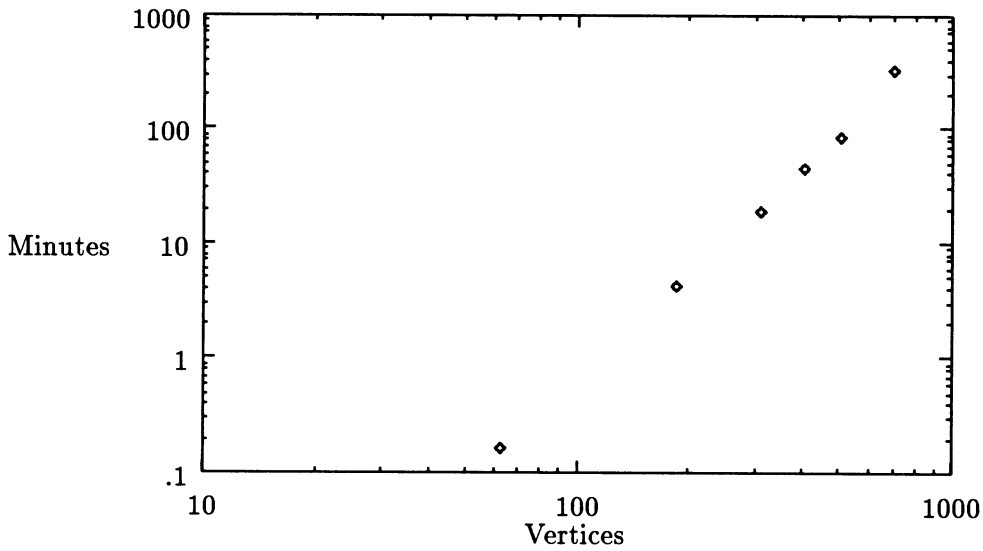


Figure 12.2: Redundant rigidity time versus starting graph size.

As discussed in Chapter 9, the total time spent on redundant rigidity involves two distinct phases; the QR factorization and the identification of subgraphs. However, the subgraph identification took a miniscule portion of the time for all the problems. For the largest problem the total time identifying redundantly rigid subgraphs was less than 2 seconds.

Whereas the four-connectivity routines are entirely deterministic, there is a degree of randomness involved in the redundant rigidity calculations (although not as much as in the optimization phase to be discussed in the next section.) In particular, 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 could have 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 determination of independence and redundancy was reproducible for most random realizations. For any of these numerically stable runs the redundant rigidity calculations were essentially deterministic.

12.2 The Small Vertex Separator

The algorithm for identifying small separators proved to be highly successful on our problems. It ran very rapidly and produced good separators. For the largest problem the total time spent in the separator routines was only 1.55 minutes, a miniscule fraction of the total running time. A plot of the size of the separator set versus the size of the graph is shown in Figure 12.3 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 are eventually combined with an invocation of the global optimizer. However, 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. In fact, 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 find an alternate technique for dividing large problems into smaller ones that is more successful at generating a small number of uniquely

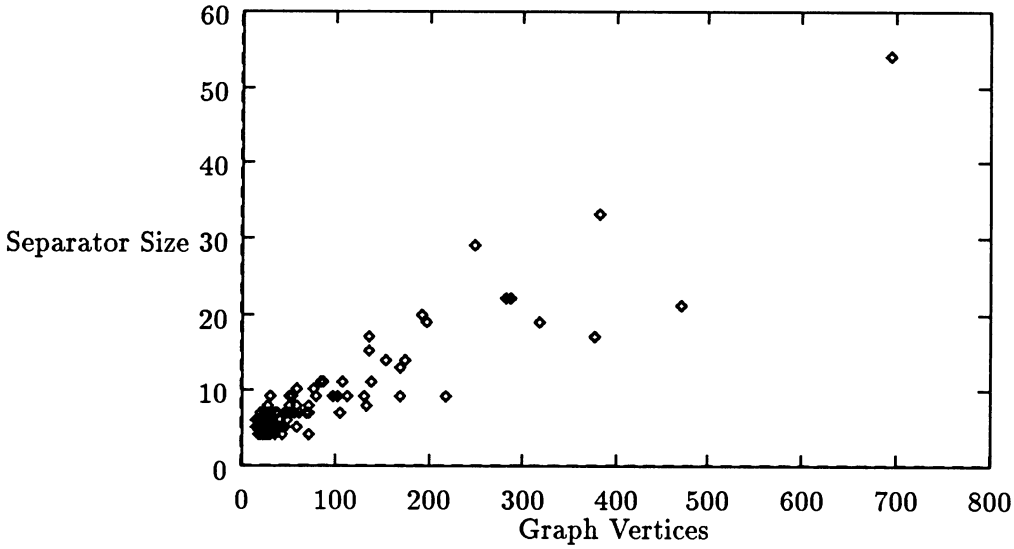


Figure 12.3: Separator size as a function of graph size.

realizable subproblems.

12.3 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 optimization routines encoded in ABBIE. A sophisticated optimizer should be able reduce the running time substantially. Hence, the running times to be 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 employs a two phase approach. First, as described in Chapter 10, several combinatoric heuristics are used to try to combine vertices whose relative locations can be determined. Most of the optimization problems encountered in the set of test problems were completely solved this way. Those that weren't had to be solved by a nonlinear global optimization. For all our problems the combinatoric operations were extremely efficient relative to the optimizations. For the largest problem all the combinatoric 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. Consequently, the cost of an optimization problem depends critically on the random seed. The program may happen to guess a good starting vector after only a few attempts and find the solution quickly. Al-

ternately, with bad luck it may require many attempts. Because of this randomness, the time required for a single optimization run is a rather uninformative number. This is partially mitigated by the fact that in the course of a run ABBIE solves many optimizations, presumably with a mixture of good and bad luck. However, as we will discuss below, the cost of a full run is often dominated by the cost of only a few of the optimizations, so the luck is amortized over only a small number of problems. For this reason we decided to run each of the six problems three times with different random seeds. 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 12.3.

Table 12.3: 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

The total cost of an optimization problem depends on three factors as discussed in Chapter 10. These factors are the number of positioned subgraphs (or *chunks*) to merge, the number of optimization variables and the topography of the penalty function. The dependence on the number of chunks is particularly simple since they must each have the same parity for the global minimizer to be found, and each parity is selected randomly. If there are k chunks, then on average only one out of every 2^{k-1} attempts will have the correct set of parities. However, for our test problems we already knew the answer, and hence the correct 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 12.3 were generated this way.

The total optimization time presented in Table 12.3 shows an unusual dependence on the size of the graph being realized. In fact, 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. More precisely, if the separator heuristic always divides the graph in half, and the combinatoric pass never eliminates an optimization problem, then the number of optimizations should grow linearly with the number of vertices. This expectation is born out by the results presented in Table 12.4, and Figure 12.4. Clearly, the optimization problems for the 40 and 60 amino acid problems must be

more difficult than those for the larger 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 also increases with problem size, as indicated by the last column of Table 12.4.

Table 12.4: 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

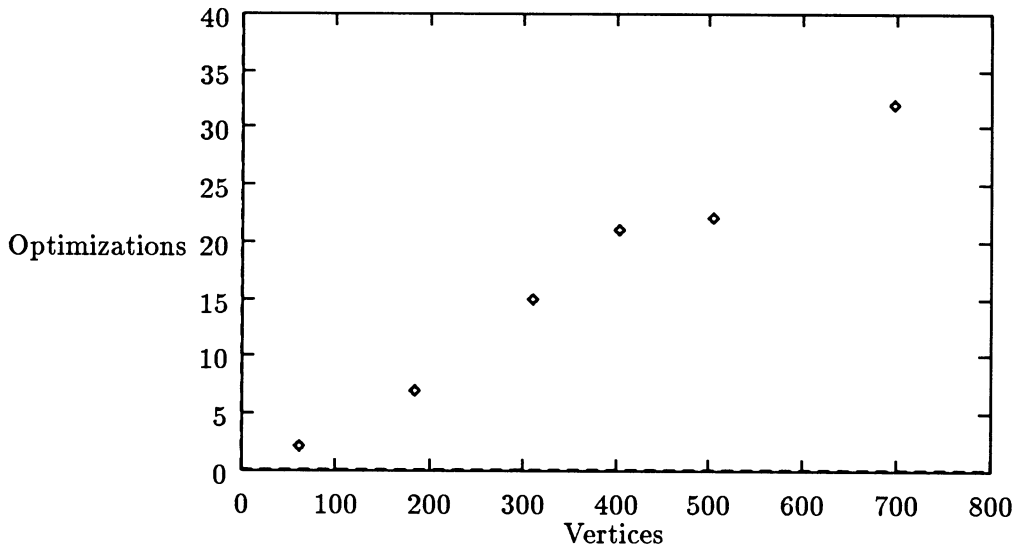


Figure 12.4: Number of optimizations versus starting graph size.

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 12.5. The number of trials necessary to find the global optimum for the three trials is included. The trials always had the parities of the chunks correct, so for a more correct measure of the difficulty of the problems these values should be multiplied

by 2^{k-1}, where k is the number of chunks. 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 increased. 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.

Table 12.5: 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

Unlike the unique realization algorithms, the running time of the optimization routines is not dominated by the initial problem. In fact, the initial optimization problem will generally occur at the lowest level of recursion and involve only vertices. These problems were all solved rapidly, which is partly a consequence of having a relatively small value for the cutoff parameter *toobig*. Without exception, the large, expensive problems all occurred while trying to combine chunks that were created by the small vertex separator. If the division heuristic were more successful at creating uniquely realizable subgraphs, these difficult problems would be avoided with a corresponding dramatic improvement in running time.

The values in Table 12.5 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 matrix factorizations. 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. As was mentioned in Section 2.1, the cost of an optimization problem can grow exponentially with the number of variables. In addition, the cost of local minimization grows approximately with the cube of the number of variables. But this doesn't mean large problems must always be expensive, and several counterexamples are presented.

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. In addition, as the results in Tables 12.3 and 12.5 indicate, the complexity of the optimization problems seems to be independent of the size of the initial molecule problem. For this reason it is reasonable to expect the total optimization cost to grow roughly proportionally to the number of optimizations, that is, linearly in the problem size.

Chapter 13

Conclusions

The recursive decomposition approach to the molecule problem that was implemented in ABBIE showed great 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. Chemists are often interested in this information for its own sake. Investigating this solution space is now reduced to a much smaller problem. The range of solutions may be related to the actual flexibility of the molecule. In this case, the motions identified by ABBIE may have important physical significance. An understanding of this *protein folding problem* would be a great boon to chemists.

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 holds the promise of greatly simplifying 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.

This approach to the molecule problem is most applicable to problems in which distances are known with a high degree of accuracy. This assumption is valid for many applications, like surveying. However, many problems, including those in chemistry, do not allow distances to be measured with great accuracy. Instead, only upper and lower bounds on the distances can be determined, and the uncertainty may be a significant fraction of the actual distance. In this case the decomposition approach becomes more problematic. In particular, our approach requires that subgraphs can be identified with unique realizations. When distances aren't known precisely, subproblems will not generally have unique answers. Instead, there will be a range of conformations that satisfies the constraints.

As long as this range of satisfying conformations remains relatively small the decomposition approach is applicable, with some modification. Instead of treating a solution to a subproblem as a rigid body in a larger problem, simply use it as a starting configuration for a subset of vertices. Then allow their relative locations to change slightly 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. If the set of solutions to the subproblem is bounded then, the particular identified solution should be near all the other solutions. By treating this specific solution as an intelligent starting point, the overall difficulty of the optimization should be greatly reduced.

This idea will not work when the range of satisfying solutions to the subproblem includes points that are far apart. This can happen when the tolerance in the distance bounds grows large enough to allow solutions that are clearly distinct. For instance, if a set of separating vertices can possibly lie in a plane then there are two distinct solution sets distinguished by a reflection of a portion of the framework through the separating plane. In general, our approach may have problems if the range of satisfying solutions contains some points in the set of measure zero that the analysis from Part II excluded. More insidiously, the solution to a particular problem may actually lie in this bad set. Molecules tend to exhibit a great deal of symmetry which violates our assumption of the algebraic independence of coordinates. It is quite possible that there are practically interesting problems for which

our analysis is inapplicable.

Even in the presense of measurement uncertainty the graph theoretic necessary conditions for unique realizability remain necessary. A generic instance of the problem that fails these tests will always allow multiple realizations. Instead of trying to solve a full, poorly posed problem, one can concentrate on the subgraphs that satisfy the necessary conditions. Any larger sets of vertices can never be unambiguously positioned. Once again a large problem can be replaced by a set of smaller ones, saving computational effort.

13.1 Future Work

Areas for future work are many and diverse. There are several open graph theoretic questions. A combinatorial characterization of rigidity in three dimensions is the Holy Grail of rigidity theorists, and is clearly a very difficult problem. A possibly easier problem would be to fully characterize which graphs have unique realizations in two and higher dimensions, or to demonstrate that this is not a graph property after all. This dissertation presented necessary conditions as well as a sufficient one, but there is a gap between that remains unfilled.

A better technique for dividing a uniquely realizable graph into pieces would also be highly desirable. As was discussed in Chapter 12, the vertex separator heuristic occasionally broke the graph into many uniquely realizable pieces. To recombine them required a large, expensive optimization. An alternate technique that was more successful at generating a small number of uniquely realizable subgraphs would have an enormous effect on ABBIE's overall running time.

On a somewhat more practical level, there are many algorithmic questions that could be addressed further. Foremost among these would be a redundant rigidity routine that takes full advantage of the sparsity of the rigidity matrix. It is disappointing that our current implementation requires $O(n^2)$ storage while the data typically consist of only $O(n)$ constraints. The sparsity structure of the rigidity matrix can be succinctly described by the original distance graph, which may allow for a nice characterization of the generation of nonzero values. This routine would have to perform several tasks on a large, very sparse matrix with many more columns than rows. First, it would need to find a basis set of linearly independent columns. Second, it would need to determine which of these are redundant; that is, which could be discarded without changing the rank of the full matrix. Third, a basis, preferably sparse, for the null space of the transpose of this matrix would need to be calculated.

We have used a QR factorization to perform these calculations, which inspires several more specific sparse matrix problems. First, is the problem of performing a sparse QR factorization in which the linear dependencies among the columns cannot be known in advance. Second, we would like to compute and store the

orthogonal matrix Q as sparsely as possible. These algorithmic questions are likely to have applications beyond the boundaries of this dissertation.

An algorithmic question that is certainly of interest in additional settings is that of computing four-connected components quickly. There are theoretical $O(n^2)$ algorithms, but to our knowledge none has ever been implemented.

The optimization phase, the most time consuming portion of the code, could be improved in numerous ways. The number of optimization variables could be reduced using more sophisticated combinatoric tricks than those described in Section 10.1. The parameter space could also be sampled in a more clever fashion. There are various optimization tools that could improve the performance of the algorithm. 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, the sparse structure of the Hessian could be exploited. Alternately, a more sophisticated global optimization routine could be employed. The approach implemented in ABBIE is rather elementary. One good candidate would be the stochastic technique of Rinnooy Kan and Timmer [KT84]. Like the current optimization routines in ABBIE this involves performing local minimizations from random starting points. However, with this approach the domain is sampled in a more sophisticated way. Additional possibilities would be the tunneling algorithm of Levy and Montalvo [LM85], or a simulated annealing approach [vA87].

A still more practical problem would be to extend the approach described here to deal with the more 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. For the approach described in this dissertation to be particularly useful to chemists, this problem must be addressed.

Another important area for future work is to extend the ideas in this dissertation to other optimization problems. The geometric nature of the molecule problem helped lead to the idea of a decomposition approach, and it is likely that there are other problems that could be similarly attacked. The optimization formulation of the molecule problem exhibits several properties that makes a decomposition conceivable. The penalty function consists of a sum of subfunctions expressing equality constraints, and each subfunction only relates a small number of variables. There are many optimization problems that share this general structure. The specific structure of the molecule problem is very special, but the idea of reducing a large problem to a sequence of smaller ones may generalize.

Parallel algorithms provide one more direction in which this work can be extended. Multiprocessor machines are becoming increasingly prevalent and impor-

tant in the area of scientific computation. The ideas described in this dissertation are extremely amenable to parallel implementation. At a coarse level, the recursive decomposition divides a large optimization problem into a set of smaller, independent subproblems. These subproblems can be run independently on separate processors, greatly reducing the overall time of a computation.

There is a finer grain parallelism that can also be exploited. We mentioned parallel algorithms for four-connected components in Chapter 5, and for rank determination in Chapter 4. The global optimization technique in ABBIE is simple to parallelize. As written it involves repeatedly generating random starting vectors and finding their nearby local minimizer. Many processors could be performing this independently in parallel until the global optimum is found. A more sophisticated version of this idea has been investigated as a general parallel approach to global optimization [BDKS86], but mostly for small dimensional problems.

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AR78] L. Asimow and B. Roth. The rigidity of graphs. *Transactions of the American Mathematical Society*, 245:279–289, November 1978.

[AR79] L. Asimow and B. Roth. The rigidity of graphs, II. *Journal of Mathematical Analysis and Applications*, 68:171–190, 1979.

[B+82] M. Becker et al. A probabilistic algorithm for vertex connectivity of graphs. *Information Processing Letters*, 15(3):135–136, 1982.

[BDKS86] R. H. Byrd, C. L Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel. Concurrent stochastic methods for global optimization. Technical Report CU-CS-338-86, Department of Computer Science, University of Colorado at Boulder, June 1986. To appear in *Mathematical Programming.*

[BG85] W. Braun and N. Go. Calculation of protein conformations by proton-proton distance constraints: A new efficient algorithm. *Journal of Molecular Biology*, 186:611–626, 1985.

[BGT81] Robert G. Bland, Donald Goldfarb, and Michael J. Todd. The ellipsoid method: a survey. *Operations Research*, 29(6):1039–1091, 1981.

[BHW86] M. Billeter, T. F. Havel, and K. Wüthrich. The ellipsoid algorithm as a method for the determination of polypeptide conformations from experimental distance constraints and energy minimization. *Journal of Computational Chemistry*, 8:132–141, 1986.

[Bis88] Christian H. Bischof. Incremental condition estimation. Technical Report ANL/MCS–P15–1088, Argonne National Laboratory, Mathematics and Computer Science Division, 1988.

[BR80] E. D. Bolker and B. Roth. When is a bipartite graph a rigid framework? *Pacific Journal of Mathematics*, 90(1):27–44, 1980.

[Bra87] W. Braun. Distance geometry and related methods for protein structure determination from NMR data. *Quarterly Reviews of Biophysics*, 19:115–157, 1987.

[Can86] J. Canny. Some algebraic and geometric computations in PSPACE. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pages 460–467, 1986.

[Cau13] A. L. Cauchy. Deuxième memoire sur les polygones et les polyèdres. *Journal de l'Ecole Polytechnique*, 19:87–98, 1813.

[CBKG86] G. M. Clore, A. T. Brunger, M. Karplus, and A. M. Gronenborn. Application of molecular dynamics with interproton distance restraints to three-dimensional protein structure determination: A model study of crambin. *Journal of Molecular Biology*, 191:523–551, 1986.

[CEG86] Thomas F. Coleman, Anders Edenbrandt, and John R. Gilbert. Predicting fill for sparse orthogonal factorization. *Journal of the ACM*, 33(3):517–532, July 1986.

[CH88] G. M. Crippen and T. F. Havel. *Distance Geometry and Molecular Conformation.* Research Studies Press Ltd., Taunton, Somerset, England, 1988.

[Con89a] Robert Connelly. On generic global rigidity. In P. Gritzmann and B. Sturmfels, editors, *Applied Geometry and Discrete Mathematics (The Victor Klee Festschrift)*, to appear, 1989.

[Con89b] Robert Connelly. Personal communication, April 1989.

[Con89c] Robert Connelly. Personal communication, September 1989.

[CP87] Thomas F. Coleman and Alex Pothen. The null space problem II. Algorithms. *SIAM Journal on Algebraic and Discrete Methods*, 8(4):544–563, October 1987.

[Cra79] Henry Crapo. Structural rigidity. *Topologie Structurale*, 1:26–45, 1979.

[Cra88] Henry Crapo. On the generic rigidity of plane frameworks. Technical report, Bat 10, INRIA, Cedex, France, 1988.

[CS80] C. R. Cantor and P. R. Schimmel. *Biophysical Chemistry.* W. H. Freeman and Company, NY, 1980.

[CT90] Joseph Cheriyan and Ramakrishna Thurimella. On determining vertex connectivity. Technical Report UMIACS-TR-90-79, CS-TR-2485, Department of Computer Science, University of Maryland at College Park, 1990.

[DH87] Andreas W. M. Dress and Timothy F. Havel. Shortest-path problems and molecular conformation. *Discrete Applied Mathematics*, 19, 1987.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[DS83] J. E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[dVBS+86] J. de Vlieg, R. Boelens, R. M. Scheek, R. Kapstein, and W. F. van Gunsteren. Restrained molecular dynamics procedure for protein tertiary structure determination from NMR data: a *lac* represser headpiece structure based on information on J-coupling and from the presence and absence of NOE's. *Israel J. Chem*, 27:181–188, 1986.

[EK85] J. G. Ecker and M. Kupferschmidt. A computational comparison of the ellipsoid algorithm with several nonlinear programming algorithms. *SIAM Journal on Control and Optimization*, 23:657–764, 1985.

[Fle87] R. Fletcher. *Practical Methods of Optimization, Second Edition.* John Wiley and Sons, 1987.

[Fog88] A. Fogelsanger. *The generic rigidity of minimal cycles.* Ph.D. dissertation, Cornell University, Department of Mathematics, May 1988.

[GH80] A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Alg. Appl.*, 34:69–83, 1980.

[Gil86] John Gilbert. Predicting structure in sparse matrix computations. Technical Report TR 86-750, Department of Computer Science, Cornell University, 1986.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[Glu75] Herman Gluck. Almost all simply connected closed surfaces are rigid. In *Geometric Topology*, pages 225–239. Lecture Notes in Mathematics No. 438, Springer-Verlag, Berlin, 1975.

[GN84] Alan George and Esmond Ng. SPARSPAK: Waterloo sparse matrix package; user's guide for Sparspak-B. Technical Report CS-84-37, Department of Computer Science, University of Waterloo, 1984.

[GP74] Victor Guillemin and Alan Pollack. *Differential Topology.* Prentice-Hall, Inc., 1974.

[GVL83] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 1983.

[GW88] Harold N. Gabow and Herbert H. Westermann. Forests, frames and games: Algorithms for matroid sums and applications. In *Proceedings of the 20th Annual Symposium on the Theory of Computing*, pages 407–421, Chicago, 1988.

[HKC83] Timothy F. Havel, Irwin D. Kuntz, and Gordon M. Crippen. The theory and practice of distance geometry. *Bulletin of Mathematical Biology*, 45(5):665–720, 1983.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[HW84] Timothy Havel and Kurt Wütrich. A distance geometry program for determining the structures of small proteins and other macromolecules from nuclear magnetic resonance measurements of intramolecular ^{1}H-^{1}H proximities in solution. *Bulletin of Mathematical Biology*, 46(4):673–698, 1984.

[Ima85] Hiroshi Imai. On combinatorial structures of line drawings of polyhedra. *Discrete Applied Mathematics*, 10:79–92, 1985.

[IMR80] O. H. Ibarra, S. Moran, and L. E. Rosen. A note on the parallel complexity of computing the rank of order n matrices. *Information Processing Letters*, 11(4,5):162, 1980.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, February 1970.

[KM69] K. Killian and P. Meissl. Einige grundaufgaben der räumlichen trilateration und ihre gefährlichen örter. *Deutsche Geodätische Komm. Bayer. Akad. Wiss.*, A61:65–72, 1969.

[Kol78] Gina Bari Kolata. Geodesy: dealing with an enormous computer task. *Science*, 200:421–422, 466, April 1978.

[KR87] Arkady Kanevsky and Vijaya Ramachandran. Improved algorithms for graph four-connectivity. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 252–259, October 1987.

[KS90] Samir Khuller and Baruch Schieber. Efficient parallel algorithms for testing k-connectivity and finding disjoint s-t paths in graphs. Technical Report TR 90-1135, Department of Computer Science, Cornell University, June 1990.

[KT84] A. H. G. Rinnooy Kan and G. T. Timmer. A stochastic approach to global optimization. In P. Boggs, R. Byrd, and R. B. Schnabel, editors, *Numerical Optimization*, pages 245–262. SIAM, Philadelphia, 1984.

[Lam70] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4:331–340, 1970.

[LCBJ87] O. Lichtarge, C. Cornelius, B. Buchanan, and O. Jardetzky. Validation of the first step of the hueristic refinement method for the derivation of solution structures of proteins from NMR data. *Proteins*, 2:340–358, 1987.

[Liu88] Joseph W. H. Liu. A graph partitioning algorithm by node separators. Technical Report CS-88-01, Department of Computer Science, York University, 1988.

[LLW86] N. Linial, L. Lovász, and A. Wigderson. A physical interpretation of graph connectivity, and its algorithmic applications. In *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science*, 1986.

[LM85] A. V. Levy and A. Montalvo. The tunneling algorithms for the global minimizer of functions. *SIAM Journal on Scientific and Statistical Computing*, 6:15–29, 1985.

[LY82] L. Lovasz and Y. Yemini. On generic rigidity in the plane. *SIAM Journal on Algebraic and Discrete Methods*, 3:91–98, 1982.

[MR87] G. L. Miller and Vijaya Ramachandran. A new graph triconnectivity algorithm and its parallelization. In *Proceedings of the Nineteenth ACM Annual Symposium on Theory of Computing*, pages 335–344, New York, NY, 1987.

[Pal90] Kate Palmer. Personal communication, March 1990.

[Rot80] Ben Roth. Questions on the rigidity of structures. *Topologie Structurale*, 4:67–71, 1980.

[Rot81] Ben Roth. Rigid and flexible frameworks. *American Mathematical Monthly*, 88:6–21, 1981.

[Sal79] Eugene Salamin. Application of quaternions to computation with rotations. Internal working paper, Stanford AI Lab, 1979.

[Sar42] Arthur Sard. The measure of the critical values of differentiable maps. *Bulletin of American Mathematical Society*, 48:883–890, 1942.

[Sax79] James B. Saxe. Embeddability of weighted graphs in k-space is strongly NP-hard. Technical report, Computer Science Department, Carnegie-Mellon University, 1979.

[Ste77] G. W. Stewart. Perturbation bounds for the *qr* factorization of a matrix. *SIAM Journal on Numerical Analysis*, 14(3):509–518, June 1977.

[Sug80] Kōkichi Sugihara. On redundant bracing in plane skeletal structures. *Bulletin of the Electrotechnical Laboratory*, 44:376–386, 1980.

[TW85] Tiong-Seng Tay and Walter Whiteley. Generating isostatic frameworks. *Topologie Structurale*, 11:21–69, 1985.

[vA87] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications.* D. Reidel Publishing Company, Boston, MA, 1987.

[Whi84] Walter Whiteley. Infinitesimal motions of a bipartite framework. *Pacific Journal of Mathematics*, 110(1):233–255, 1984.

[Wun77a] W. Wunderlich. Gefährliche annahemen der trilateration und bewegliche fachwereke I. *Angew. Math. Mech.*, 57:297–304, 1977.

[Wun77b] W. Wunderlich. Gefährliche annahemen der trilateration und bewegliche fachwereke II. *Angew. Math. Mech.*, 57:363–368, 1977.

[Wun77c] W. Wunderlich. Untersuchungen zu einem trilaterations problem mit komplaneren standpunkten. *Sitz. Osten. Akad. Wiss.*, 186:263–280, 1977.

[Wun79] W. Wunderlich. Eine merkwürdige familie von beweglichen stabwerken. *Elem. Math.*, 34/6:132–137, 1979.

[Wüt89a] Kurt Wütrich. The development of nuclear magnetic resonance spectroscopy as a technique for protein structure determination. *Accounts of Chemical Research*, 22:36–44, 1989.

[Wüt89b] Kurt Wütrich. Protein structure determination in solution by nuclear magnetic resonance spectroscopy. *Science*, 243:45–50, January 1989.

[Yem79] Yechiam Yemini. Some theoretical aspects of position-location problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 1–8. IEEE, October 1979.