Angle and Distance Geometry Problems

Andrew Kay

Wolfson College, Oxford

A thesis submitted for the degree of D. Phil.
Trinity Term 1990

Programming Research Group
Oxford University Computing Laboratory
11 Keble Road, Oxford, OX1 3QD

February 11, 1991

Abstract

Angle and Distance Geometry Problems

Andrew Kay
Wolfson College, Oxford

A thesis submitted for the degree of D. Phil.
Trinity Term 1990

Programming Research Group
Oxford University Computing Laboratory
11 Keble Road, Oxford, OX1 3QD

Distance geometry problems (DGPs) are concerned with the construction of structures given partial information about distances between vertices. I present a generalisation which I call the angle and distance geometry problem (ADGP), in which partial angle information may be given as well. The work is primarily concerned with the algebraic and theoretical aspects of this problem, although it contains some information on practical applications. The embedding space is typically real three dimensional space for applications such as computer aided design and molecular chemistry, although other embedding spaces are possible. I show that both DGP and ADGP are NP-hard, but that in some sense the ADGP is more expressive than the DGP. To combat the problems of NP-hardness I present some graph theoretic heuristics which may be applied to both DGP and ADGP, and so reduce the time required by general purpose algorithms for their solution. I discuss the general purpose algorithms Cylindrical Algebraic Decomposition and Gröbner bases and their application to this field.

In addition, I present an $O(n)$ parallel algorithm for computing convex hulls in three dimensions, using $O(n^2)$ processors connected in a mesh-like topology with no shared memory.

Acknowledgement

Hem-hem hav to sa that sort of thing you kno. *Molesworth*

O horrible, o horrible, o most horrible *Hamlet*

I would like to thank all my friends who tried to keep me at least half sane throughout this research, and especially some people with these names: Annabelle, Chris, Edwina, Geraint, Jane, Jeremy, Jim, Paul, Peter, Phil, Richard, Stefan and Tim, who tried the contrary, with some measure of success.

I wish to thank my supervisor, Bill McColl, for introducing me to computational geometry, and for delicate negotiations with SERC. Joy Reed for providing financial support through Esprit, an incentive to finish on time, and 'free' trips abroad. The SERC for giving me my grant. The PRG and its members for a pleasant and informal working environment.

Contents

List of Figures

Chapter 1

Introduction

In this thesis I examine algorithmic methods for solving distance geometry problems (DGPs) and angle and distance geometry problems (ADGPs). Distance geometry has been studied as an alternative representation system for Euclidean geometry[Blu70], and has been used in molecular chemistry to determine the three dimensional structure of large molecules[HKC83]. I consider an extension of distance geometry which includes angle information as well as distance information. I propose angle geometry as a "geometric programming language." Previously, geometric languages, such as `LineTool`[EY88][Eri89], have used real polynomials as their language. This language is so general that it requires very heavy (and slow) algebraic machinery to solve, and is hard to reason about geometrically. The language Juno[Nel85] allows general constraints, but requires the user to supply an initial configuration which is "almost" correct. It is also limited to two dimensions. Angle and distance geometry, on the other hand, *is* a dimension independent geometric representation, which lends itself to geometric reasoning.

In DGPs, a finite set of points in Euclidean space is represented by the set of distances between pairs of points, rather than by their Cartesian co-ordinates. The DGP is concerned with the transformation from distance representation into Cartesian co-ordinates (the reverse transformation being very straightforward). The easiest case is when all the distances are given, and given precisely. However, it is often necessary to consider problems in which only some of the distances are specified, and perhaps specified inexactly. In molecular chemistry for example, points represent positions of atoms and distances represent bond lengths which might be known only approximately. This sort of representation problem often has many different Cartesian "solutions." Even in the exact case, with all distances specified, a solution may be translated, rotated and reflected, and still be a solution.

In ADGPs angle information as well as distance information is provided, in the form of the specification of some or all of the angles of the triangle formed by three points. Again, a solution may be translated, rotated and reflected, and still be a solution. ADGPs may be used in molecular chemistry in the same way as DGPs, but have the advantage of allowing the specification of bond angles as well as bond lengths.

When using ADGP as a geometry language, the idea is that geometric objects should be specified using constraint based information. Then the tool is used to discover whether the object can exist. An example of this is the `LineTool` editor [EY88][Eri89], in which the constraints may be *any* set of algebraic equations and inequations. These problems are all algebraic, and so may be solved by any general purpose algebraic methods. The difficulty is that the best such algorithms take time $O(2^{2^n})$ in general, although if there is known to be a single solution, then $O(2^n)$ is sufficient. It is therefore necessary to limit the size of problems submitted to algorithms of this type. It is one of the aims of this thesis to show in part how this may be achieved.

In chapter 2 I give a formal definition of a DGP, and examine the work of Havel et al. [HKC83] on solving the DGP in the context of molecule configuration. I present complete polynomial time solutions for some of the easier cases. In chapter 3 I define the ADGP and show how some of the previous methods may be extended to handle it. Again I present complete methods for the easy cases. I show how some common geometric constructs can be modelled by the formalism. The general algebraic methods of CAD and Gröbner bases may be used to solve ADGP (and DGP) exactly, but they require a large amount of computing time. To assist the exposition, I postpone a description of these methods until chapters 6 and 7. However it is important to note that ADGP can be solved.

In chapter 4 I present a proof that DGP and ADGP are NP-complete, even when some apparently strong restrictions are imposed. Further I show that, theoretically at least, ADGP is no easier if the only angle allowed in constraints is π, and all distances are exact.

In chapter 5 I present a new type of approach to ADGP and DGP using graph theoretic methods. These methods allow me to consider only the topology of an ADGP, without having to concentrate on its metrical properties. To a large extent these methods do not depend on the dimension of the particular embedding space. There are two basic plans of attack. The first is to split the problem into independent subproblems. The second is to make local transformations to simplify the problem. Whilst neither method is complete, the objective is to simplify the problem so that the amount of effort used in general purpose methods is minimised.

In chapters 6 and 7 I discuss two general methods, Cylindrical Algebraic Decomposition (CAD) [ACM84a] and Gröbner bases[Buc85]. The Gröbner basis method is the one actually used in `LineTool`, but works only when there are only finitely many solutions, or when the user can identify in advance an independent set of parameters such that there are only finitely many solutions for each parameter setting. The CAD method does not have this requirement, but it is rather slower. However, in chapter 6 I present a method for using CAD to solve optimisation problems. In an optimisation problem, geometric constraints are given as before, but the requirement is to find the "best" solution, where "best" is defined as some polynomial function of the co-ordinates defining the geometric object. For example, in molecule configuration problems, the best solution is the one which minimises the total energy of the molecule. In robotics, the best new position for the robot arm may be the one which requires the least movement to get to from its present position.

In appendix C I present a new parallel algorithm for solving convex hull problems in three dimensions, using an array of communicating processors. The algorithm is based on one for convex hulls in two dimensions, given in [Cha84]. Problems of convex hull computation may be expected to arise when considering geometric objects constructed from ADGPs. The convex hull of an object is a guide to its general shape, and may be used as an approximation to it without having to consider internal details. This is useful in areas such as robotics to decide quickly whether or not (to a first approximation) a robot may move through a constriction. If the convex hull will fit through, then so will the robot. If not, then it might be necessary to do more precise calculation.

Chapter 2

Distance Geometry Problems

In this chapter I formalise the Distance Geometry Problem (DGP) and discuss some (non-algebraic) solutions which have been studied by several researchers, notably Havel et. al. [HKC83]. Their paper is directed towards reconstructing molecular structure, given partial experimental evidence. Although analysis of X-ray data can give position information directly, it is not always possible to get good X-ray data for a particular molecule, and so other methods must be used. For example, the technique of NMR can be used to give information about the linear separation of some pairs of atoms within the molecule. It is this kind of data which allows the use of distance geometry.

2.1 Definitions

Each distance is approximated by an upper and a lower bound, representing the best experimental evidence available. If the atoms are numbered from 1 to n, then ρ_{ij} is the actual (perhaps unknown) distance between atoms i and j. Suppose L_{ij}, U_{ij} are respectively the known lower and upper bounds, then conditions and constraints can be written down

$$\begin{aligned} \forall\, i,j \in 1 \ldots n \mid \quad & 0 \leq L_{ij} \leq \rho_{ij} \leq U_{ij} \\ \wedge \quad & i = j \Leftrightarrow U_{ij} = 0 \\ \wedge \quad & \rho_{ij} = \rho_{ji} \end{aligned}$$

By analogy with the terms of graph theory (see chapter 5), the atoms are referred to as nodes (or vertices), and pairs of atoms as edges (or sides). If no information is known about an upper bound for a particular edge, it is represented formally

as $+\infty$ with the rule that for any real number x, $x < +\infty$. If ij is a edge for which no information is available, I call it **undefined**, and represent it by $L_{ij} = 0$, $U_{ij} = +\infty$. If the edge ij is known precisely, then $L_{ij} = U_{ij}$ and I call it **exact**.

Havel et. al. look for an **embedding function** $\mathcal{E} \in 1..n \rightarrow \mathbb{R}^3$, although in general the dimension of the space can vary from problem to problem. Let d be an arbitrary but fixed integer greater than or equal to 1 which is the dimension of the embedding space. If $\mathbf{x} = \langle x_1, \ldots x_d \rangle$ and $\mathbf{y} = \langle y_1, \ldots y_d \rangle$ are vectors in $\mathbb{R}^d$ then define $\Delta(\mathbf{x}, \mathbf{y})$ to be $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$, that is, the usual Euclidean metric in $\mathbb{R}^d$. An embedding function $\mathcal{E}$ must satisfy

$$\forall\, i, j \in 1 \ldots n \mid L_{ij} \leq \Delta(\mathcal{E}(i), \mathcal{E}(j)) \leq U_{ij}$$

Although the actual configuration of the molecule satisfies this equation (for $d = 3$), it is probably not a unique solution. One problem with numerical rather than algebraic approaches to this kind of task is that a single solution might be found, but this provides no information about where (or whether) to look for other solutions.

For the problem to make sense computationally, the constraints and embeddings are required to be expressible in terms of integers, rationals or real algebraic numbers[1]. I assume in what follows that this is the case. In the context of numerical solution, rather than exact algebraic solution, it is reasonable to assume that a constraint is satisfied if it is satisfied to within the required accuracy. Further, if a pair of constraints L_{ij}, U_{ij} differ by less than this amount then they are deemed equal, and the edge is considered to be exact.

2.2 Very Easy Cases

A few moments' thought provides solutions for some simple degenerate cases of DGP. Although these are unlikely to arise at the outset, the methods of chapter 5 might produce them as subproblems requiring solution. It is therefore worth making a special check for their presence. The first is where there are no constraints at all, in which case *any* embedding function is allowable. The second occurs when all the lower bounds are 0, in which case all the nodes may be mapped to the same point in $\mathbb{R}^d$ (the first case is a special case of this). The third occurs when the problem has only two or three nodes. Two nodes may be embedded at the origin and any point at distance L_{12} from the origin. Three points may be embedded (or proved to be not-embeddable) by elementary trigonometry and case analysis, which I shall not discuss further.

[1] that is, solutions of univariate polynomials with integer coefficients.

2.3 The Complete DGP

When $L_{ij} = U_{ij}$ for all i and j, we have the *Complete DGP* which is fairly simple to solve by construction. In this case, a greedy algorithm simply takes the points one at a time and assigns co-ordinates consistent with a maximal linearly independent subset of the earlier points. For simplicity, restrict points to the x-axis at first, then to the xy-plane when this is no longer possible, and finally to the whole of $\mathbb{R}^3$. This makes it easy to see when the dimension has increased, and also provides a convenient maximal linearly independent set, points $\mathcal{E}(1)$, $\mathcal{E}(2)$, $\mathcal{E}(r)$ and $\mathcal{E}(s)$ in the proof.

Blumenthal [Blu70] presents a decision criterion for the Complete DGP, using Cayley-Menger determinants. The conditions on the determinants turn out to be similar to the equations which my algorithm solves. The advantage of my algorithm is that it is constructive, and takes no longer (asymptotically) than the problem of deciding whether an embedding is a correct embedding.

Lemma 2.1 *The Complete DGP for $\mathbb{R}^3$ has a constructive solution, that is, an algorithm exists which returns an embedding function if one is possible. Furthermore, the algorithm runs in optimal time $\Theta(n^2)$.*

Proof *Choose $\mathcal{E}(1)$ to be the origin $(0,0,0)$, and $\mathcal{E}(2)$ the point $(U_{1,2},0,0)$. Consider $E(3) = (x,y,0)$ for some x and y. These values must satisfy the constraint equations concerning $\Delta(\mathcal{E}(1),\mathcal{E}(3))$ and $\Delta(\mathcal{E}(2),\mathcal{E}(3))$, which are*

$$\begin{array}{lllll} A1 & x^2 & +y^2 & = & U_{1,3}^2 \\ A2 & (x-U_{1,2})^2 & +y^2 & = & U_{2,3}^2 \end{array}$$

Since $U_{1,2} > 0$ there are at most 2 real solutions, which may be seen by subtracting A1 from A2 and solving. If there are no real solutions, then the algorithm cannot proceed, and the embedding is impossible. If y is zero, accept $\mathcal{E}(3)$, and continue with $\mathcal{E}(4)$, $\mathcal{E}(5)$ etc., until the first one, $\mathcal{E}(r)$ say, for which y is non-zero is found (if one exists). In this case the two y values differ only in sign, and the positive one is chosen (arbitrarily). Let $(x_r, y_r, 0)$ be the co-ordinates of $E(r)$. At this stage the whole embedding becomes 2 dimensional, and it remains to embed points $r+1, \ldots n$. Consider the next point, $\mathcal{E}(r+1) = (x,y,z)$. There are now three equations to satisfy, concerning the distances to $\mathcal{E}(1)$, $\mathcal{E}(2)$ and $\mathcal{E}(r)$.

$$\begin{array}{lllll} B1 & x^2 & +y^2 & +z^2 & = & U_{1,r+1}^2 \\ B2 & (x-U_{1,2})^2 & +y^2 & +z^2 & = & U_{2,r+1}^2 \\ Br & (x-x_r)^2 & +(y-y_r)^2 & +z^2 & = & U_{r,r+1}^2 \end{array}$$

Again, since $U_{1,2} > 0$ and $y_r > 0$ there are at most 2 solutions, differing only in the sign of z. As before, if z is not real, then there is no embedding possible, otherwise search for the first point with $z \neq 0$. Take s to be the index of this point, choose the positive value for z, and let (x_s, y_s, z_s) be the co-ordinates of $E(s)$. The remaining points, $s+1, \ldots n$ are embedded by solving four equations for distances from points $\mathcal{E}(1)$, $\mathcal{E}(2)$, $\mathcal{E}(r)$ and $\mathcal{E}(s)$. Finally check all distance pairs $\Delta(\mathcal{E}(i), \mathcal{E}(j))$ to see if they are correct. If not, there is no embedding possible.

To show the algorithm is correct, suppose $\mathcal{E}'$ is an embedding. Translate and rotate using a mapping M, so that $M\mathcal{E}'(1) = (0,0,0)$ and $M\mathcal{E}'(2) = (U_{1,2}, 0, 0)$. This is possible since rotation and translation preserve distances. I claim that $M\mathcal{E}'(i) = \mathcal{E}(i)$ for each $i \in 1 \ldots r-1$. To see this, observe that adding a z^2 term to the equations $A1, A2$ does not add any more solutions, since we would then have to solve $y^2 + z^2 = 0$ each time, rather than $y^2 = 0$. Because both $\mathcal{E}$ and $M\mathcal{E}'$ must both satisfy these equations, they are the same for points 1 to $r-1$. Furthermore, $M\mathcal{E}'(r)$ cannot lie on the x-axis. Rotate about the x-axis, with a rotation R, so that $RM\mathcal{E}(r)$ lies in the positive-y half of the xy-plane. I claim that $RM\mathcal{E}'$ and $\mathcal{E}$ agree as far as point $s-1$, by similar reasoning. Now reflect the embedding, using reflection F in the xy-plane, if necessary, to ensure that $FRM\mathcal{E}'(s)$ has positive z-co-ordinate. I claim that $FRM\mathcal{E}'$ and $\mathcal{E}$ are identical, again by uniqueness of solutions of the defining equations. FRM is a distance preserving transformation, therefore I have shown that if an embedding $\mathcal{E}'$ exists, then the construction of $\mathcal{E}$ as given is correct. Furthermore, since the distances are checked at the end, if no embedding exists, the algorithm fails.

For each of the n atoms, the algorithm solves a fixed set of four or fewer equations, in time $O(1)$ (assuming constant time for all arithmetic). The consistency check at the end takes time $O(n^2)$, and so the algorithm is $O(n + n^2) = O(n^2)$. I claim that all the input, a set of $\frac{1}{2}n(n-1)$ distances, must be examined. To see this, suppose that an algorithm has produced an embedding but without examining the bounds on edge ij. Then the embedding is incorrect if $L_{ij} = \Delta(\mathcal{E}(i), \mathcal{E}(j)) + 1$, so the assumption that edge ij can be ignored is incorrect. Therefore the algorithm runs in optimal time $\Theta(n^2)$.

The algorithm as given may be extended to any number of dimensions, in an obvious fashion, and may be used to determine the smallest value of d for which the structure may be embedded in $\mathbb{R}^d$, and to indicate if all such embeddings are impossible. It is easy to see (by the proof of the lemma) that if such a d exists, then $d \leq n$, and in this case the algorithm solves at most d equations in d variables per atom. Also, the time taken to calculate a single Euclidean distance rises to $O(d)$, and so the running time is $O(nd^2 + n^2 d) = O(n^2 d)$. Also from the proof, if a structure can be embedded in $\mathbb{R}^d$, then any two embeddings are related by transla-

tion, rotation and reflection, and so the embedding is essentially unique. Distance geometry in general cannot distinguish between left and right handed instances of structures, since the two versions satisfy the same distance constraints.

2.4 Bound Smoothing Methods

One approach to solving the General DGP is examined in [HKC83]. In this context, we try to get as much information as possible from the constraints before attempting an embedding. There are several advantages to this. First, it may be possible to rule out some problems as insoluble without having to try an actual embedding. Secondly, it provides better bounds for edges which have not been specified, or which have a very weak specification. This is useful for constraint solvers which require an "initial guess" for all the lengths in the system. Thirdly, it is possible that some inexact constraints become exact, which is useful in the graph theoretic algorithms of chapter 5.

Bound smoothing is done by examining the consequences of properties of the embedding space on any embedded structure. The most useful of these is the triangle inequality, which states that the straight line distance between two points is always less than the distance between them via a third point.

$$\forall\, a, b, c \in \mathbb{R}^d \mid \Delta(a, c) \leq \Delta(a, b) + \Delta(b, c)$$

The triangle inequality bound smoothing algorithm consists of two stages, first upper bound smoothing and then lower bound smoothing. It is important to notice that this scheme of bound adjustment only works because the upper bounds do not depend on the lower bounds. If this were not the case then the scheme would have to be repeated, since the new lower bounds would affect the upper bounds. This could occur any number of times.

2.4.1 Upper Bound Smoothing

Suppose now that we are trying to embed a structure. If an embedding, $\mathcal{E}$, exists then

$$\begin{aligned} &\forall\, i, j, k \in 1 \ldots n \mid \\ &\quad L_{ik} \leq \Delta(\mathcal{E}(i), \mathcal{E}(k)) \\ &\qquad \leq \Delta(\mathcal{E}(i), \mathcal{E}(j)) + \Delta(\mathcal{E}(j), \mathcal{E}(k)) \\ &\qquad \leq U_{ij} + U_{jk} \end{aligned}$$

This gives a simple condition for ruling out certain structures. If $L_{ik} > U_{ij} + U_{jk}$ for any i, j and k, then no embedding is possible. A more important observation follows from the inequality $\Delta(\mathcal{E}(i), \mathcal{E}(k)) \leq U_{ij} + U_{jk}$, which is that if U_{ik} is greater than $U_{ij} + U_{jk}$ then it gives no extra information. The difference $U_{ik} - (U_{ij} + U_{jk})$ represents slack in the constraints, and if it is positive may be reduced to zero by replacing the upper bound U_{ik} with the value of $U_{ij} + U_{jk}$. Havel et. al. refer to the process of removing slack from the constraints as bound smoothing. Note that bound smoothing is guaranteed not to affect any embedding properties of the structure, but merely propagates constraint information. Constraint propagation is a useful heuristic in many constrained optimisation problems, such as VLSI layout[JS85].

The above process can be repeated for all possible triplets of points, until no further slack remains in the upper bounds. This method is inefficient, since it may take many iterations, and it is not obvious that it always terminates. A better scheme is obtained by the following observation. Consider starting at node i and travelling to node j along a route made out of edges of the DGP. For any such route, the length is defined to be the sum of the upper bounds, U_{kl}, of all the edges traversed. The final (smoothed) value for U_{ik} is the length of the shortest route from i to k. All these shortest route lengths can be found in time $O(n^3)$, using Floyd's algorithm [Sed88].

2.4.2 Lower Bound Smoothing

As well as being reduced by lowering upper bounds, slack can be reduced by raising lower bounds. To do this, start again from the triangle inequality, but now deduce its consequences for long chains of edges. Suppose s is any path from node $i = s_1$ to node $k = s_q$. By induction on q, and using the triangle inequality it can be deduced that for any embedding $\mathcal{E}$,

$$\Delta(\mathcal{E}(i), \mathcal{E}(k)) \leq \sum_{1 \leq x < q} \Delta(\mathcal{E}(s_x), \mathcal{E}(s_{x+1}))$$

Suppose further that t is any path from node $m = t_1$ to node $j = t_r$. Then the concatenation $t \frown \langle (j, i) \rangle \frown s$ is a path from k to m. So by the above formula,

$$\begin{aligned} \Delta(\mathcal{E}(k), \mathcal{E}(m)) \quad \leq \quad & \sum_{1 \leq y < r} \Delta(\mathcal{E}(t_y), \mathcal{E}(t_{y+1})) \\ & + \Delta(\mathcal{E}(i), \mathcal{E}(j)) \\ & + \sum_{1 \leq x < q} \Delta(\mathcal{E}(s_x), \mathcal{E}(s_{x+1})) \end{aligned}$$

Rearranging,

$$\begin{aligned}\Delta(\mathcal{E}(i),\mathcal{E}(j)) \quad \geq \quad & \Delta(\mathcal{E}(k),\mathcal{E}(m)) \\ & - \sum_{1\leq y<r} \Delta(\mathcal{E}(t_y),\mathcal{E}(t_{y+1})) \\ & - \sum_{1\leq x<q} \Delta(\mathcal{E}(s_x),\mathcal{E}(s_{x+1}))\end{aligned}$$

From the definition of embedding, the upper and lower bounds can be used to arrive at

$$\Delta(\mathcal{E}(i),\mathcal{E}(j)) \geq L_{km} - \sum_{1\leq y<r} U_{t_y,t_{y+1}} - \sum_{1\leq x<q} U_{s_x,s_{x+1}}$$

Providing the upper bounds have already been smoothed, they obey the triangle inequality, and so

$$\Delta(\mathcal{E}(i),\mathcal{E}(j)) \geq L_{km} - U_{mj} - U_{ik}$$

This is true for every embedding $\mathcal{E}$, and for all nodes i, j, k and m, and so the smoothed lower bound for the edge (i,j) is defined to be

$$(*) \quad L'_{ij} = \max\{L_{km} - U_{mj} - U_{ik} \mid 1 \leq k,m \leq n\}$$

If all these values are calculated, a new structure is obtained in which the lower bounds have no slack, with respect to the triangle inequality. It might appear that a repetition of this procedure leads to even tighter lower bounds, that is, to calculate

$$L''_{ij} = \max\{L'_{km} - U_{mj} - U_{ik} \mid 1 \leq k,m \leq n\}$$

using the lower bounds obtained from the first pass. I show that in fact this is not the case, and that a second pass does not change any of the bounds. This leads to a much simplified algorithm.

Lemma 2.2 *One pass of the above algorithm achieves the theoretical lower bounds of equation* $(*)$*, provided that the upper bounds have previously been smoothed.*

Proof *It is required to show for each* i *and* j *that* $L'_{ij} = L''_{ij}$*. It is certainly true, by properties of* max *that* $L'_{ij} \leq L''_{ij}$*, and so it remains to show* $L'_{ij} \geq L''_{ij}$*. By properties of* max*, and from the definition of* L''*, there exists a pair of values* k,m *such that* $L''_{ij} = L'_{km} - U_{mj} - U_{ik}$*. Furthermore, there exists a pair of values* x,y *such that* $L'_{km} = L_{xy} - U_{ym} - U_{kx}$*. Since the upper bounds are smoothed, it can be deduced that*

$U_{kx} - U_{ix} \geq -U_{ik}$ and $U_{ym} - U_{jy} \geq -U_{mj}$. Using these results, and the definitions of L and L''

$$\begin{aligned} L'_{ij} &\geq L_{xy} - U_{ix} - U_{jy} \\ &= L'_{km} - U_{ix} - U_{jy} + U_{kx} + U_{ym} \\ &\geq L'_{km} - U_{mj} - U_{ik} \\ &= L''_{ij} \end{aligned}$$

as required, and so the lemma is complete.

Although this seems to lead to an $O(n^4)$ algorithm to compute all the new upper bounds, an $O(n^3)$ algorithm is obtained easily by noting that

$$L'_{ij} = \max\{\max\{L_{km} - U_{ik} \mid 1 \leq k \leq n\} - U_{mj} \mid 1 \leq m \leq n\}$$

Then the inner maximisation can be computed for all possible i m in time $O(n^3)$ and the results stored. The outer maximisation then takes only $O(n^3)$ time, and so the whole algorithm is $O(n^3)$.

2.4.3 Higher Order Bound Smoothing

Havel et. al. also derive a higher order condition on the bounds, using an inequality which holds for any set of four points embedded in $\mathbb{R}^3$. Unfortunately, the upper and lower bounds are interdependent, and no terminating algorithm is known, although an approximation algorithm is given in their paper. Yet another problem arises, because removing higher order slack results in triangle inequality slack for the new bounds. This problem appears to have been solved, using an incremental version of the algorithms above. Despite these approximations, the algorithm has total complexity at least $\Omega(n^5)$, although no termination proof is known. They also discuss 5th, 6th, 7th ...order inequalities, but no algorithms are hinted at. It is possible that inequalities all the way up to $(n-1)$th order (where n is the number of points) would contribute to the removal of slack in some cases. Further, these methods are specific to $\mathbb{R}^3$ and require modification in other embedding spaces. I do not pursue these methods further.

2.5 Embedding the General DGP

Having obtained smoothed bounds for the edges, the next step is to perform the actual embedding. One technique involves choosing a random set of distances, ρ_{ij},

which satisfy the bounds. This corresponds to an instance of the complete exact DGP, with $L_{ij} = U_{ij} = \rho_{ij}$. From a statistical point of view, it might be thought better to choose the midpoint of each interval as ρ_{ij}. However, as we shall see, it is often necessary to be able to generate several different instances, in case the first instance turns out to be highly unembeddable. Havel has achieved good results choosing the distances independently and uniformly over the interval, and other distributions have been considered [Cri81].

It is unlikely that the distances, chosen at random, are embeddable, so the algorithm presented earlier is not applicable. It is, however, possible to find a set of points in $\mathbb{R}^3$ which most nearly match the chosen distances, with respect to the least squares differences. This is achieved by finding the largest three eigenvectors of a particular matrix, the metric matrix relative to the centroid of the points. If the node of index zero is defined to be the centroid, then it is simple to show, in any embedding in a Euclidean space, that

$$\forall\, i \mid \rho_{0i} = \frac{1}{n} \sum_{1 \le j \le n} \rho_{ij}^2 - \frac{1}{n^2} \sum_{1 \le j < k \le n} \rho_{jk}^2$$

Then the metric matrix $\mathbf{M}$ relative to the centroid is defined to have i,jth co-ordinate

$$M_{ij} = \frac{1}{2}(\rho_{0i}^2 + \rho_{0j}^2 - \rho_{ij}^2)$$

The metric matrix turns out to be very useful in the search for an embedding. The following lemma, first proved by Schoenberg[Sch35], shows that an embedding is possible if and only if the metric matrix has no negative eigenvalues. Furthermore, a possible embedding is computable from the diagonalising matrix. My proof is based on that given by Havel, in [HKC83], and is slightly simpler.

Lemma 2.3 *(Schoenberg) An instance of complete exact DGP is embeddable in* $\mathbb{R}^d$ *if and only if the corresponding metric matrix* $\mathbf{M}$ *is positive semidefinite of rank less than or equal to* d.

Proof *Suppose* $\mathbf{M}$ *is positive semidefinite, of rank* d. *Then there are* n *real eigenvalues (including multiplicities), at most* d *of which are strictly positive, and the rest zero. Let* $\mathbf{\Lambda}$ *be the diagonal matrix of eigenvalues, in non-increasing order. Using Gram-Schmidt orthonormalisation[Str80], a corresponding unitary matrix* $\mathbf{V}$ *can be found, whose* i*th row is the eigenvector of* $\mathbf{M}$ *corresponding to the* i*th eigenvalue. Then* $\mathbf{M} = \mathbf{V}^T\mathbf{\Lambda}\mathbf{V}$. *Now define* $\mathbf{\Lambda}^\diamond$ *to be the* $d \times n$ *matrix given by*

$$\forall\, i \in 1 \ldots d, j \in 1 \ldots n \mid {\mathbf{\Lambda}^\diamond}_{ij} = \sqrt{\mathbf{\Lambda}_{ij}}$$

so that $\Lambda^{\circ T}\Lambda^{\circ} = \Lambda$. *The chosen embedding is represented by a* $n \times d$ *matrix,* $\mathbf{X}$, *whose* i*th row,* $\mathbf{x_i}$, *forms the co-ordinates of* $\mathcal{E}(i)$ *in* $\mathbb{R}^d$. *I claim that* $\mathbf{X} = (\Lambda^{\circ}\mathbf{V})^T$ *represents a correct embedding. To prove this we must show for each* i *and* j, *that* $\Delta(\mathbf{x_i}, \mathbf{x_j})$ *is equal to* ρ_{ij}. *First note that* $\mathbf{X}\mathbf{X}^T$ *is an* $n \times n$ *matrix, whose* ij*th element is the inner product of* $\mathbf{x_i}$ *and* $\mathbf{x_i}$. *Furthermore,*

$$\begin{aligned}
\mathbf{X}\mathbf{X}^T &= (\Lambda^{\circ}\mathbf{V})^T(\Lambda^{\circ}\mathbf{V}) \\
&= \mathbf{V}^T\Lambda^{\circ T}\Lambda^{\circ}\mathbf{V} \\
&= \mathbf{V}^T\Lambda\mathbf{V} \\
&= \mathbf{M}
\end{aligned}$$

Comparing the elements on the diagonal of $\mathbf{X}\mathbf{X}^T$ *and* $\mathbf{M}$, *it can be deduced that*

$$\begin{aligned}
\mathbf{x_i}^2 &= \frac{1}{2}(\rho_{0i}^2 + \rho_{0i}^2 - \rho_{ii}^2) \\
&= \frac{1}{2}(2\rho_{0i}^2 - 0) \\
&= \rho_{0i}^2
\end{aligned}$$

Using these results and the definition of Δ *in* $\mathbb{R}^d$ *produces the required result as follows.*

$$\begin{aligned}
\Delta^2(\mathbf{x_i}, \mathbf{x_j}) &= (\mathbf{x_i} - \mathbf{x_j})^2 \\
&= \mathbf{x_i}^2 + \mathbf{x_j}^2 - 2\,\mathbf{x_i} \cdot \mathbf{x_j} \\
&= \mathbf{x_i}^2 + \mathbf{x_j}^2 - (\rho_{0i}^2 + \rho_{0j}^2 - \rho_{ij}^2) \\
&= \rho_{ij}^2
\end{aligned}$$

For the converse, suppose $\mathbf{X}$ *represents a correct embedding in* $\mathbb{R}^d$, *as above. By simple calculation,* $\mathbf{X}\mathbf{X}^T$ *is equal to* $\mathbf{M}$. *Since* $\mathbf{M}$ *is a real, symmetric matrix, it can be diagonalised, using a unitary matrix* $\mathbf{V}$. *Then the eigenvalues lie on the diagonal of the matrix* $\mathbf{V}\mathbf{M}\mathbf{V}^T = \mathbf{V}\mathbf{X}\mathbf{X}^T\mathbf{V}^T$. *If the matrix* $\mathbf{V}\mathbf{X}$*is designated* $\mathbf{A}$, *then each eigenvalue is the length of the vector represented by one of the rows of* $\mathbf{A}$, *and so must be greater than* 0. *Further,* $\mathbf{X}$ *can be considered to be a linear mapping from* $\mathbb{R}^d$ *to* $\mathbb{R}^n$, *and so the product* $\mathbf{X}\mathbf{X}^T = \mathbf{M}$, *which is a linear mapping from* $\mathbb{R}^n$ *to* $\mathbb{R}^n$ *must have rank less than or equal to* d.

Although this lemma is useful, it is only applicable when the eigenvalues of $\mathbf{M}$ are all positive. If a set $\{\rho_{ij}\}$ of distances are chosen randomly to satisfy the given bounds, then it is most unlikely that this is the case. [HKC83] provides a proof for the following lemma, which allows us under certain circumstances to compute an approximation to a non-embeddable metric matrix, which is embeddable in $\mathbb{R}^d$.

Lemma 2.4 *Suppose a set* $\{\rho_{ij}\}$ *of distances define an exact embedding problem, and that* $\mathbf{M}$ *is the corresponding metric matrix. If the* d *largest eigenvalues (ordered by absolute value) of* $\mathbf{M}$ *are positive, then the 'best' approximation to an embedding in* $\mathbf{R}^d$ *is obtained by setting the other eigenvalues to zero and proceeding as in lemma 2.3. In this context, the best approximation is the one which minimises the sum of squares of the differences between the* ρ_{ij} *and the actual distances.*

The eigenvalues of the metric matrix can also be used to determine the "natural" dimension of the DGP. Suppose λ is the most negative eigenvalue. Since only positive eigenvalues can arise from real configurations ($\mathbf{M}$ is positive semidefinite by lemma 2.3) λ must be entirely due to errors in the data. Therefore any eigenvalue of absolute value of around $-\lambda$ or less is likely to be due almost entirely to errors also, assuming that the data errors are evenly distributed amongst the constraints. Thus only (positive) eigenvalues significantly greater than $-\lambda$ should be regarded as non zero. It is the number of these which determine the dimension of the DGP.

Using these lemmas an embedding can be obtained which is 'almost correct' in many cases. The procedure is to choose a random set of distances which satisfy the bounds, and to calculate the first d eigenvalues of the metric matrix. This can be done in expected time $O(dn^2)$ using a probabilistic algorithm for computing eigenvalues, as described in [Str80]. If all of the d eigenvalues are positive then lemma 2.4 can be used to find an approximate embedding, that is, one in which most of the constraints are either satisfied, or almost satisfied. Otherwise a new set of random distances are chosen and the cycle repeated. Havel reports that usually only two or three attempts are required to find a suitable configuration.

The remaining problem is to find an embedding which does satisfy all the constraints, starting from the approximate embedding. Several general searching schemes are available to do this, such as steepest descent, conjugate gradient and simulated annealing. These methods require a target function which is minimised when the constraints are satisfied. A suitable target function for this problem might be

$$\begin{aligned} T(\mathbf{X}) &= \sum_{1 \le i < j \le n} (max\{0, \frac{\Delta^2(x_i, x_j)}{U_{ij}^2} - 1\})^2 \\ &+ \sum_{1 \le i < j \le n} (min\{0, \frac{\Delta^2(x_i, x_j)}{L_{ij}^2} - 1\})^2 \end{aligned}$$

This function is zero precisely when the bounds are satisfied, and is positive otherwise. It has the useful property that its gradient is easy to compute, and it increases fairly steeply when the bounds are violated. Therefore it makes good sense to use a technique such as steepest descent or conjugate gradient which can take advantage

of this information. Simulated annealing cannot use this information, and so is not a good choice for this problem.

2.6 Multidimensional Scaling

Multidimensional scaling is a set of methods for solving "non-metric distance problems." That is, problems where the distance constraints between pairs of points are not necessarily specified by length, but by some other property. As a special case, when lengths are specified (or bounds on lengths) we have the DGP in one of its forms. Then the method reduces to what is called "classical scaling," essentially the complete exact DGP. This type of data is handled well by the matrix methods of section 2.5, even when there is random error in the input. If the error is an additive constant (perhaps caused by a zero error) then extensions to the method, such as in [Tor52] can be used to good effect. However, when the error is non-linear, or a general monotonic function, then the classical methods are not reliable [SBO81], and other algorithms must be used. Essentially, the data can no longer be regarded as distances, but as some "non-metric" measure of separation. A fairly exhaustive survey and taxonomy of multidimensional scaling methods can be found in [CA80].

The most common non-metric example is when an ordering is imposed on pairs of points so that a pair of points lower in the ordering must be closer together than a pair higher in the ordering. The ordering need not be total. This sort of ordering data arises naturally when information is extracted from non-numerical sources. For example, in psychology a subject may say that a stimulus C is more like stimulus A than B is, telling us that the difference AC is (in some sense) less than the difference AB. The difference can be expressed on a scale as a *dissimilarity* coefficient which increases with greater difference. Sometimes it is more natural to measure a *proximity* coefficient which decreases with difference. By repeating the experiment for many triples of stimuli and applying multidimensional scaling a map can be drawn which places similar stimuli close together, in some Euclidean embedding space. The data is then in a form suitable for the usual kinds of statistical analysis, or for the experimenter to detect patterns of similarity made clear by the picture (when the dimension of the embedding space is two or three). It can be seen that the most important property of the dissimilarity or proximity coefficients is that they should be monotonic with respect to the distances in the embedding, since the actual values are likely to be subjective [Coo58]. For this to be possible, the derived ordering relation must of necessity be transitive and reflexive. Sibson [Sib72] gives some more conditions on these coefficients which makes them easier to work with.

Surprisingly, it has been shown that ordering information is sufficient in many cases to provide an almost unique embedding (up to rigid transformations and scaling), when there are many points in few enough dimensions (see for example Shepard [She62a][She62b] or Kendall [Ken71]). This leads to a new method for solving the exact DGP, by throwing the distance data away, keeping a record of the ordering only, then using the methods of multidimensional scaling to find an embedding, and finally expand or contract the mapping to match the distance data. Of course this method can only work well when many of the distances are specified, and when there are many more points than embedding dimensions. Therefore the methods of Multidimensional Scaling are probably less suited to the general problem of exact DGP than to special cases (as in molecular geometry) where this condition is more likely to hold. In the following sections I shall discuss the two main methods used.

2.6.1 Shepard's Method

Shepard [She62a] explains his method for multidimensional scaling where the aim is to produce an embedding of minimal dimension which satisfies (or nearly satisfies) the order constraints of a set of proximity data. Given n points and proximity data it is always possible to satisfy the constraints in $n-1$ dimensions. To see this, consider a regular simplex (generalised tetrahedron) with n vertices in $\mathbb{R}^{n-1}$. Any pair of vertices has the same separation to begin with. Take each pair of vertices in turn, starting with the pair with greatest proximity, and draw them together by a small distance δ, which should decrease for each pair. This movement can be achieved without altering the distances between any of the other pairs [BH60], so it can be seen that after all the pairs have been adjusted the configuration satisfies the order constraints. However, since the dimension is high, there are many possible configurations which could achieve this.

Shepard's method is iterative, with two components of change at each iteration, one to achieve the ordering constraints and one to reduce the dimension. He argues that the dimension of a set of points tends to be least when their variance (sum of the squares of the distance from the centroid) is greatest. To see this intuitively, note that the simplex formation has low variance and high dimension, but that a "dumbell" formation with two clusters far apart has high variance and is essentially one dimensional.

The first step is to normalise the set of rankings so that they are real numbers between 0 and 1. In its simplest form, Shepard's algorithm takes a regular simplex in $n-1$ dimensions centred at the origin as a starting point, then applies transformations according to the following rules until there is convergence. In fact, to

improve the numerical behaviour the configuration is recentred at the origin and renormalised after each step.

The first rule attempts to satisfy and maintain the ordering constraints by moving pairs of points which are too close together further apart, and vice versa, by an amount proportional to the discrepancy in order ranking. The second rule is one which tends to lower the dimension by increasing the variance. This may be achieved by stretching those lengths which are greater than the mean, and shrinking those lengths less than the mean. [She62a] provides some experimental data to determine what the sizes of the two adjustments should be to achieve fast and accurate convergence.

The result of this stage is a (hopefully) low dimensional configuration in n-dimensional space, and requires a projection into a lower dimensional space. This can be done easily using the eigenvector methods of section 2.5.

2.6.2 Kruskal's Method

Kruskal[Kru64a] argues that Shepard's algorithm is rather ad hoc, because it is never made explicit what the final configuration should satisfy. Rather it is a recipe to be followed because it gives (psychologically) reasonable results. Shepard defines a quantity δ which must be small when the algorithm terminates, but this is a measure of convergence of the iterative step, not a measure of how good the embedding is.

Kruskal avoids this pitfall by defining the *stress* of a configuration, that is the degree to which the constraints are violated. It is claimed that the stress function is a natural quantitative measure of the non-monotonicity of the embedding. The object is then to minimise stress for the desired embedding space, thus multidimensional scaling becomes a problem of statistical fitting. The advantage of Shepard's algorithm is that the embedding dimension is determined automatically, whereas Kruskal's requires a separate calculation for each possible dimension if the dimension of the structure to be embedded is not known in advance. In general the minimum stress configuration for an embedding in a low dimensional space cannot be derived as the projection of a minimum stress embedding in a higher dimensional space.

Suppose the data is supplied as dissimilarity information where δ_{ij} is the dissimilarity of i and j, with $\delta_{ij} = \delta_{ji}$, $\delta_{ii} = 0$, and not all the δ_{ij} need be defined. If f is any monotonic function (i.e. $a > b \Rightarrow f(a) \geq f(b)$), and $\mathcal{E}$ is any embedding, then

the (normalised) stress of $\mathcal{E}$ is given by

$$S(\mathcal{E}) = min_f \sqrt{\frac{\sum_{i,j} (\Delta_{ij} - f(\delta_{ij}))^2}{\sum_{i,j} \Delta_{ij}^2}}$$

where $\Delta_{ij} = \Delta(\mathcal{E}(i), \mathcal{E}(j))$ and the summations are over pairs i, j such that $i \neq j$ and δ_{ij} is defined.

The stress, S, in d dimensions of the particular set of constraints is the minimum value of $S(\mathcal{E})$ as $\mathcal{E}$ ranges over all embeddings in $\mathbb{R}^d$. The desired embedding is taken to be this stress minimising embedding. The stress is regarded as a measure of how well the data fits in the embedding space. For psychological data, Kruskal recommends stress of less than $2\frac{1}{2}\%$ to be excellent. It is likely that much lower values would be required in molecular geometry, since more accuracy is required in the embedding. Kruskal notes in [Kru64b] that if there are insufficient constraints, then the stress is zero, and the method may break down.

There are two computational subproblems using Kruskal's method. The first is to calculate $S(\mathcal{E})$ for any given $\mathcal{E}$, and secondly to find the particular $\mathcal{E}$ which minimises this. The second subproblem is solved in [Kru64b] using the standard method of steepest descent. To keep arithmetical errors to a minimum the embedding is renormalised after each iterative step by scaling and translating the centroid to the origin (as in Shepard's method). The problem of avoiding local minima is achieved in the usual way by repeating the descent from several randomly chosen starting points.

The problem of calculating $S(\mathcal{E})$ requires that a minimising monotonic function f be found. In fact, since it is only the values of f evaluated at each of the δ_{ij} which are required, f may be regarded as a step function. Kruskal's algorithm for this is essentially the same as that described in [Mil59]. Suppose that $\delta_1 \ldots \delta_m$ is an ordering of the δ_{ij} in increasing order, and $\Delta_1 \ldots \Delta_m$ is the corresponding ordering of the Δ_{ij}. (Note that the Δ_i will only be in increasing order if the order constraints are satisfied for this particular embedding. In this case the embedding process is complete.) Since f is a step function, there is a partition of $\{1 \ldots m\}$ into consecutive blocks $b_1 \ldots b_k$ such that f is constant on each block. To minimise $S(\mathcal{E})$ it can be seen that the value of f in any block b must be chosen as the average value $\overline{\Delta}_b$ of the Δ_i such that i lies in b. It remains to calculate the correct partition. Kruskal begins with the partition such that only values of Δ_i corresponding to equal values of δ_i lie in the same block. Thus if the δ_i are distinct then each block has only one element. In general this initial partition does not give rise to a

monotonic sequence of $\overline{\Delta}_{b_1} \ldots \overline{\Delta}_{b_k}$, rather this is the goal of the following procedure. The blocks are successively merged with their neighbours until the larger blocks so formed do give rise to a monotonic sequence. The method is rather similar to bubblesort, except that blocks are merged rather than swapped when they do not form an increasing sequence. A very clear description of the algorithm is given in [Kru64b].

Two blocks can be merged in constant time if a linked list structure is used. Further, at most $k-1$ mergings can occur and $k \leq m$, Therefore the merging process is at most linear in m. The merging process must be repeated at each step in the steepest descent in order to calculate $S(\mathcal{E})$, and since m can be as large as $n^2/2$ (where n is the number of nodes), the method is $O(n\log(n) + sn^2)$, where $n\log(n)$ is the time to sort the δ_{ij} (which needs to be done only once), and s is the number of steps in the minimisation.

2.7 Another Approach

Instead of attacking the whole problem in one go, I advocate using a certain amount of effort simplifying the problem in advance. As I show in chapter 4, DGP in any fixed number of dimensions is NP-hard, so it is likely that any general algorithm will be slow. My method, which relies mainly on graph theoretic tools (chapter 5), attempts to reduce problem size or difficulty, locate easy subproblems, or split into independent parts for efficient parallel reduction. Of course there will usually still be difficult subproblems, and for these the brute force methods of minimisation are probably the only possible courses of action.

Chapter 3

Angles and Distances

In this chapter I introduce angles into the Distance Geometry Problem, and arrive at the Angle and Distance Geometry Problem (ADGP). There are now several possibilities for interesting subproblems, depending on how much information is given to start with for both angles and distances.

3.1 Algebraic Formulation

Consider three nodes to be embedded in $\mathbb{R}^d$ for some d. These nodes form the vertices of an imaginary triangle in $\mathbb{R}^d$. Any of the internal angles of this triangle can be specified with a single number in the range 0 to π. If the three nodes are labelled i, j and k, and the angle is the one included between the edges ij and jk, then this angle is denoted θ_{ijk} (see figure 3.1). To allow inexact angles, θ is specified by a closed subinterval of $[0, \pi]$ (which can be represented by its endpoints), so that when $0 \leq \alpha < \beta \leq \pi$ the interval $[\alpha, \beta]$ represents a range of possible angles, and when $\alpha = \beta$, the angle is exact. The interval $[0, \pi]$ represents a completely unconstrained angle, which I denote by the symbol ∞. I say that the angle is *undefined.*

In space of dimension greater than two, the concepts of clockwise and anticlockwise are not well defined, and so I make no distinction between positive and negative angles. I assume for convenience that θ_{ijk} is defined if and only if θ_{kji} is defined, and that in this case the two are equal. I also assume that $\theta_{iij} = \theta_{iji} = \{0\}$ for any nodes i and j.

It is convenient to represent angle constraints in terms of cosines, which are required

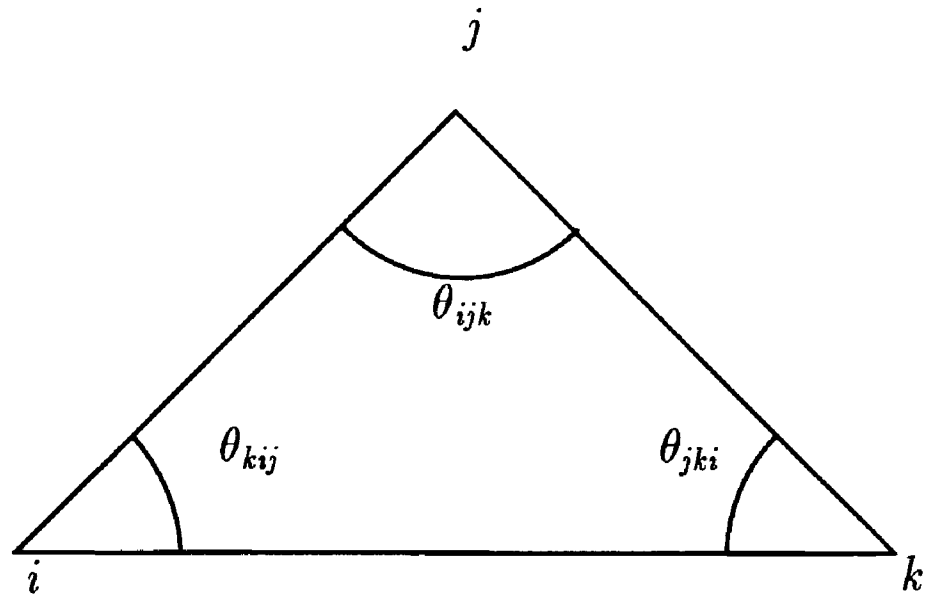


Figure 3.1: Labelling the angles of triangle ijk.

to be algebraic numbers. This makes it possible to represent angles such as $\frac{\pi}{2}$, even though it is not an algebraic number. The sufficiency of the representation is justifiable because of two observations. First, the fact that the angles must lie between 0 and π, in which range the cosine function is strictly monotonic, and second from the rule for the cosine of the angle θ between two vectors $\mathbf{p}$ and $\mathbf{q}$. This can be expressed by

$$\mathbf{p} \cdot \mathbf{q} = pq \cos\theta$$

where $\cdot$ denotes scalar product, and p and q are the lengths of $\mathbf{p}$ and $\mathbf{q}$ respectively ($p^2 = \mathbf{p} \cdot \mathbf{p}$, $q^2 = \mathbf{q} \cdot \mathbf{q}$). With this equality in mind, for points P, Q and R in $\mathbb{R}^d$ define the internal angle at Q subtended by P and R to be

$$\angle(PQR) = \cos^{-1}\left(\frac{\overrightarrow{QP} \cdot \overrightarrow{QR}}{\sqrt{\overrightarrow{QP}^2 \overrightarrow{QR}^2}}\right)$$

provided $P \neq Q$ and $R \neq Q$. Cosines of sums, differences and simple multiples of angles can be computed using standard trigonometric formulae and provided that the original cosines are algebraic, so are the results.

For numerical solutions, rather than algebraic ones, it is of course usual to represent the cosines as rational numbers to a certain maximum accuracy, rather than insist on exact algebraic numbers. In this context, a constraint may be regarded as satisfied if it is correct to the accuracy of solution required. This issue is discussed further in chapter 4.

I now define the Angle and Distance Geometry Problem. This is the problem of finding an embedding of a structure which satisfies both angle and distance constraints. Although angle constraints are in practice represented by a pair of cosines (for the upper and lower bounds), the exposition is clearer in terms of intervals of angles. In the remainder of this work when I talk about the manipulation of intervals of angles it should be understood that the calculations really concern the cosines of the endpoints.

Definition 3.1 *An instance of the* **Angle and Distance Geometry Problem,** *or* **ADGP** *for short, consists of a pair of integers n and d, and a triple of constraint functions θ, L and U such that*

$$
\begin{array}{l}
\theta \in (1\,..\,n) \times (1\,..\,n) \times (1\,..\,n) \rightarrow \mathbf{Int} \\
L \in (1\,..\,n) \times (1\,..\,n) \rightarrow \mathbb{R} \\
U \in (1\,..\,n) \times (1\,..\,n) \rightarrow \mathbb{R} \cup \{\infty\} \\
\forall\, i,j,k \in (1\,..\,n) \mid \\
\qquad (\theta_{ijk} = \theta_{kji} \land \theta_{iji} = \{0\} \land \theta_{iij} = \{0\}) \\
\qquad \land\, (L_{ij} = L_{ji} \land U_{ij} = U_{ji} \land L_{ii} = 0 \land U_{ii} = 0) \\
\qquad \land\, (L_{ij} \leq U_{ij} \lor U_{ij} = \infty)
\end{array}
$$

where **Int** *is the set of all closed subintervals of* $[0, \pi]$. *The problem is then to find an embedding function $\mathcal{E}$ satisfying*

$$
\begin{array}{l}
\mathcal{E} \in (1\,..\,n) \rightarrow \mathbb{R}^d \\
\forall\, i,j,k \in (1\,..\,n) \mid \\
\qquad (L_{ij} \leq \Delta(\mathcal{E}(i), \mathcal{E}(j)) \leq U_{ij} \\
\qquad \land\, ((\mathcal{E}(i) = \mathcal{E}(j) \lor \mathcal{E}(k) = \mathcal{E}(j)) \lor \angle(\mathcal{E}(i)\mathcal{E}(j)\mathcal{E}(k)) \in \theta_{ijk})
\end{array}
$$

As is the case for the DGP, the constraints and embeddings must be expressible in terms of integers, rationals or algebraic numbers for the problem to make computational sense. I assume this condition holds in what follows.

3.2 Algebraic Solution

As with the DGP, there are algebraic solutions to the ADGP which are guaranteed to produce an embedding, if one exists, within a bounded (although large) length of time. The constraints must be defined exactly (as algebraic numbers) to allow the methods to work successfully. When this conditions is met, exact embeddings may be computed using algebraic means, such as cylindrical algebraic decomposition

(chapter 6) or Gröbner bases (chapter 7), in at most double exponential time. As before, using the extension to cylindrical algebraic decomposition discussed in chapter 6, an embedding can be calculated which maximises an algebraic target function of the co-ordinates of the points.

3.3 Various SubProblems

Some instances of ADGP lend themselves to special methods of solution. This can be seen in the case of the complete DGP, discussed in chapter 2, which has its own specific algorithm. Any ADGP gives rise naturally to a DGP, simply by removing the angle constraints. If this DGP has a unique solution up to rigid motions of the embedding space (for example, it may be a complete DGP), then the ADGP can be attacked in the following way. First solve the corresponding DGP, then check that the angle constraints are satisfied. If so, then the ADGP is soluble, otherwise it isn't.

3.4 Constraint Propagation

Constraint propagation is a technique for getting the most out of a set of constraints for the least work. The constraints are analysed for immediate or obvious conclusions, perhaps leading to more constraints. These derived constraints may imply more constraints, and so on, with the result that the solution search space is narrowed down. This technique is used in automatic VLSI design problems, such as floor planning and wire routing [JS85]. Bound smoothing is discussed in chapter 2, and in fact this is a method of constraint propagation, where there are no angle constraints present. It remains to find ways of propagating angle information too, and this I call *angle propagation.*

A constraint propagation method of this type consists of alternations between bound smoothing and angle propagation (described below). A possibility arises here for an infinite regression, because bound smoothing and angle propagation are interrelated. That is, repeating the sequence "bound smoothing – angle propagation" may result in new changes to the constraints every cycle, which might never terminate. I have been unable to find a way of deciding when to abort this cycle and accept a set of constraints which are not fully propagated, nor to find a way of incorporating these two components into one (terminating) algorithm. I show below that some of the problems involved in bound smoothing are NP-hard (see

chapter 4), so it is not surprising that constraint propagation is difficult. I suggest that the sequence "bound smoothing – angle propagation – bound smoothing" be applied just once, unless there is evidence available that more or less propagation should be used in any particular case.

3.4.1 Angle Propagation

In this section I consider how to make angle data propagate around the constraint graph. Angle constraints are specific to a single triangle, and do not directly affect angles in adjacent triangles. This is in contrast to length constraints, each of which may be common to many triangles. I propose a two stage angle propagation process. First all the angles within a single triangle are adjusted so that there is no slack in the bounds (pure angle propagation). Secondly, if some angles and lengths are known within one triangle, then angle constraints may be used to improve the length constraints (angle-length propagation).

Pure Angle Propagation

Suppose that some of the angles, θ_{ijk}, and some exact length constraints, $L_{ij} = U_{ij}$, are given. Suppose further that some triangle ijk has precisely two of its angles specified exactly. Clearly the third angle can be calculated, using the fact that in any embedding in Euclidean space the angles in a triangle sum to π. Conversely, if a triangle has all three angles specified, then it should be checked that their sum is indeed π, or else no embedding is possible. From now on, I assume that whenever a new angle constraint is encountered then this calculation is done, when appropriate, either to calculate or check the value of the third angle.

Similarly, if any of the angles are inexact, and not all completely undefined, then it may be possible to refine them in the following way. Suppose $[\alpha_i, \beta_i]$ for $i = 1, 2, 3$ (where $0 \leq \alpha_i \leq \beta_i \leq \pi$) are the angle constraints for three angles in a triangle of nodes in some instance of the ADGP. Let θ_1, θ_2 and θ_3 be the actual angles of any embedding of this triangle. Then for each $i = 1, 2, 3$, $\alpha_i \leq \theta_i \leq \beta_i$. The angles of the triangle must sum to π, therefore adding, $\sum_i \alpha_i \leq \pi \leq \sum_i \beta_i$. It is easy to see that this inequality is both necessary and sufficient for an angle embedding of the triangle to be possible. However, it may be that some of the bounds can be tightened. Suppose that $\beta_1 + \beta_2$ is less than π. Then the difference must be made up by the third angle, $\theta_3 = \pi - (\theta_1 + \theta_2) \geq \pi - (\beta_1 + \beta_2)$. Since this is true in all embeddings, the lower bound α_3 is tightened, to the new value $\alpha_3' = \max\{\alpha_3, \pi - (\beta_1 + \beta_2)\}$. A similar tightening is possible for the upper bounds.

I claim that the following assignments tighten the pure angle constraints in any one triangle as far as possible. That is, the new constraints are implied by the old, and no tighter bounds are so implied. Repetition of the assignments will have no further effect.

$$\begin{aligned} \alpha_1' &= \max\{\alpha_1, \pi - (\beta_2 + \beta_3)\} & \beta_1' &= \min\{\beta_1, \pi - (\alpha_2 + \alpha_3)\} \\ \alpha_2' &= \max\{\alpha_2, \pi - (\beta_3 + \beta_1)\} & \beta_2' &= \min\{\beta_2, \pi - (\alpha_3 + \alpha_1)\} \\ \alpha_3' &= \max\{\alpha_3, \pi - (\beta_1 + \beta_2)\} & \beta_3' &= \min\{\beta_3, \pi - (\alpha_1 + \alpha_2)\} \end{aligned}$$

It can be seen that the treatment of exact angles described above is just a special case of this (where $\alpha_i = \beta_i$).

Angle-Length Propagation

Now consider the case when, for one particular triangle, all the angles are known exactly, and at least one side is known. Then the lengths of all the sides can calculated (or checked), using the sine rule

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

for any triangle with angles A, B and C, and sides of length a, b and c, where a is opposite A etc. Similarly, if two sides and an included angle are given, we can calculate the third side and the other angles, using the cosine rule,

$$2\,a\,b\cos C = a^2 + b^2 - c^2$$

The cosine rule may also be used to calculate the three angles when all three sides are given. However, if two sides and a non-included (exact) angle are given, then there may be two possible non-congruent triangles which satisfy the constraints, corresponding to different choices of square root when using the cosine rule. This information is summarised in figure 3.2.

When there is enough data for a particular triangle to determine it uniquely, I call such a triangle *completable*. When all three sides and all three angles are known (exactly), then I call the triangle *complete*. In this case the angle data is redundant and is removed to simplify the representation. This removal does not affect the set of possible embeddings in any way. However, from a statistical point of view, it may be that some information may be lost, concerning which embeddings are more likely than others. In general these sorts of transformation will not preserve statistical properties of the embeddings.

A similar process may be used in triangles with inexact angle and length data. There is a large amount of case analysis to do, so I have consigned it to appendix A. All the mathematics is based on the sine and cosine rules.

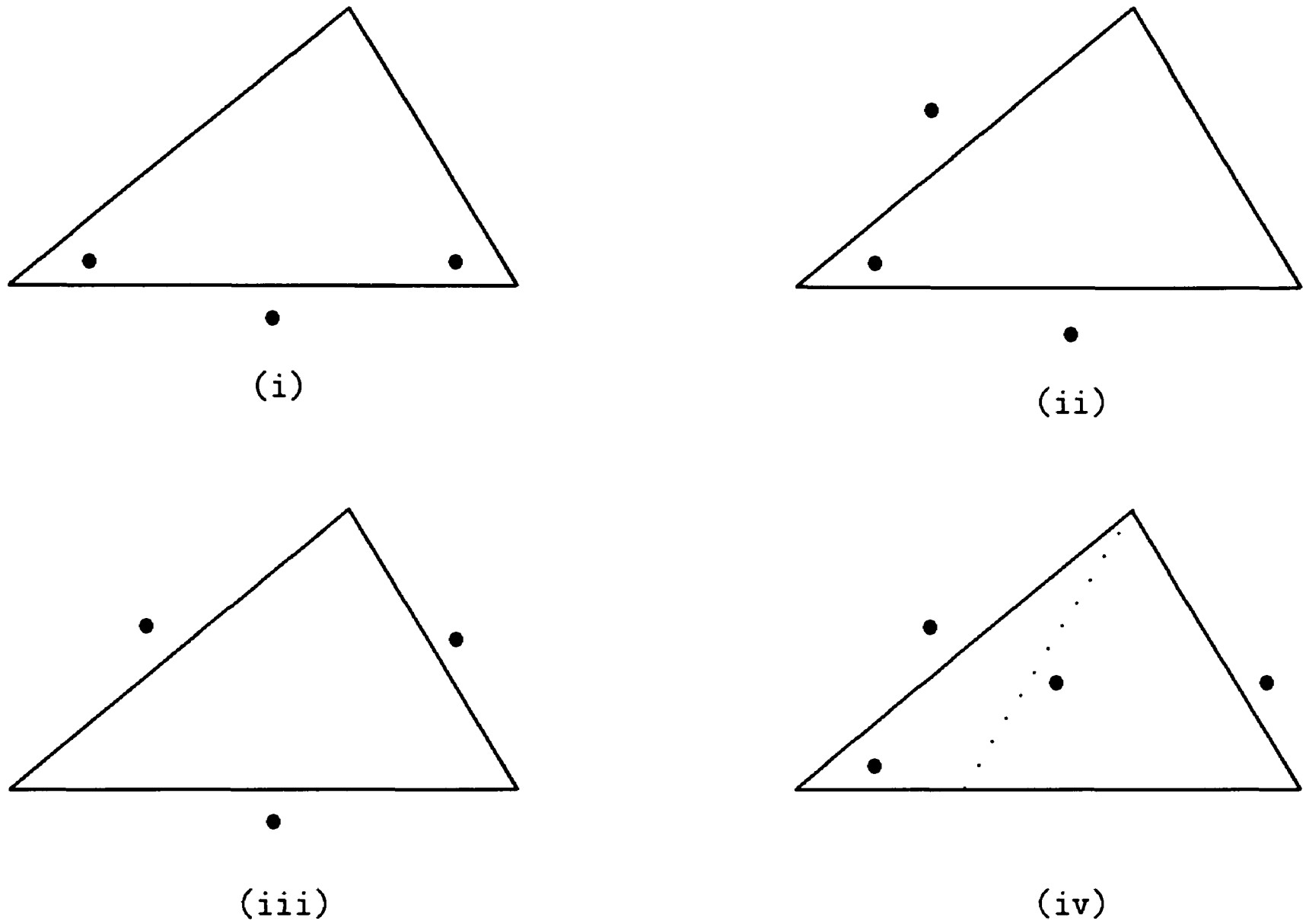


Figure 3.2: *Completion of triangles.* • *denotes a known (exact) angle or length. (i) Two angles and a side or (ii) two sides and included angle or (iii) three sides are sufficient to complete a triangle. For (iv) two sides and non-included angle, however, there are two possibilities.*

The next step is to propagate the new constraint information around the whole graph, and an algorithm to achieve this is as follows. First a pass is made through the data applying pure angle propagation and angle-length propagation to each incomplete triangle. If a triangle becomes complete, it may make one of its neighbours completable, in which case complete it, and recursively complete *its* neighbours which become completable. Assuming that completion takes constant time this algorithm is linear in the sum of the number of complete triangles produced and the number of original constraints, because each triangle can be completed at most once. It is however possible to construct artificial cases such that the order in which the search occurs affects the final set of completed triangles.

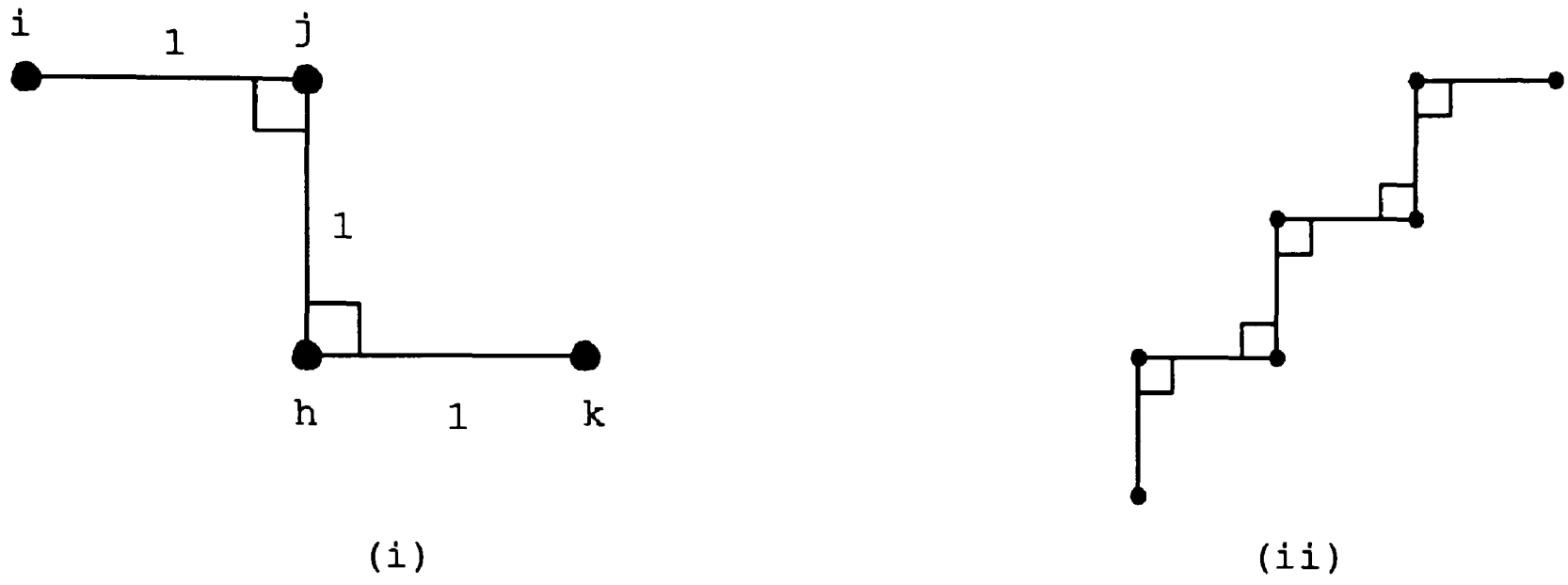


Figure 3.3: Difficulties in bound smoothing (see text).

3.4.2 Bound Smoothing in ADGPs

I now discuss the problem of bound smoothing in ADGPs. It is not possible to do as good a job with angle problems as it was with distance problems, as I show. At first glance the problem seems similar, and the following algorithm looks as though it works just as well for upper bound smoothing:

> Find the shortest path between each pair of points using Floyd's algorithm, except that when combining the lengths of two edges with an angle constraint between them use the cosine rule to calculate the longest possible length of the third side (instead of merely summing the upper bounds).

Unfortunately, although this algorithm produces bounds which are correct, they are not as tight as possible. In the case of DGP, the final upper bound between two points corresponds to the maximum length of the shortest path between them. That is, every path between the pair could be extracted from the rest of the DGP and embedded to at least reach that bound. However, in an ADGP, the algorithm can produce an upper bound which some paths cannot achieve.

For example, consider figure 3.3 (i) which depicts an instance of ADGP with three exact lengths and two exact angles. After this kind of upper bound smoothing the new upper bounds are $U_{ih} = U_{jk} = \sqrt{2}$ and $U_{ik} = 1 + \sqrt{2}$. However, a little calculation will show that in any dimension (> 1) of embedding space the upper bound for ik is $\sqrt{5}$ (which is slightly smaller that $1 + \sqrt{2}$). In other words, the

path cannot achieve the length $1 + \sqrt{2}$ which is its new upper bound. The extra slackness is because the shortest path algorithm does not take both angle constraints into account at the same time. I have been unable to discover a polynomial time algorithm which can successfully compute the maximum lengths of all the possible paths between two nodes. One reason for the difficulty of this problem is that there are $O(2^n)$ paths in a graph of n nodes.

Similar problems arise with lower bound smoothing. Figure 3.3 (ii) depicts an exact ADGP in two dimensions in which it is very difficult to determine if the two ends may meet. The condition for meeting is that there is a choice of signs for all the lengths such that those on the vertical edges sum to zero, and those on the horizontal edges sum to zero. This problem is the Partition problem, and is NP complete, so the lower bound problem is NP hard. For further discussion on this type of result, see chapter 4.

My suggestion is to use the two bound smoothing algorithms of chapter 2, with added information using the cosine rule whenever this is appropriate. The smoothed data is still better than if the angles were ignored completely, and is correct in the sense that it does not change the set of all possible embeddings.

3.5 Complete Exact Angle Problems

The Complete Exact Angle Problem occurs when all the angles are specified exactly, but none of the lengths are. It is easy to see that a degenerate embedding which maps all nodes to the origin is a solution. However adding just one (exact) length constraint allows all the rest of the lengths to be calculated, using the constraint propagation algorithm described above. This fact is proved in lemma 3.1.

Lemma 3.1 *If all the angles are given and exact, and at least one (exact) distance is known, then all the distances can be calculated using constraint propagation, and so all the distances are determined uniquely.*

Proof *Suppose, without loss of generality, that the distance given is* U_{12}. *For any pair* i, j *we wish to calculate* U_{ij}. *First it is possible to calculate* U_{1j}, *using* U_{12}, θ_{1j2} *and* θ_{j21}, *by the sine rule. Then* U_{ij} *can be calculated in a similar way from* U_{1j}, θ_{1ji} *and* θ_{ji1}.

Using this lemma allows all the possible embeddings of a complete exact angle problem to be related in a simple way.

Lemma 3.2 *Any two embeddings in $\mathbb{R}^d$ for a Complete Exact Angle Problem are related by scaling followed by a rigid transformation (including reflections) of $\mathbb{R}^d$.*

Proof *Suppose $\mathcal{E}$ and $\mathcal{E}'$ are the two embeddings, then one can be scaled so that the distances $\Delta(\mathcal{E}(1), \mathcal{E}(2))$ and $\Delta(\mathcal{E}'(1), \mathcal{E}'(2))$ are equal, and both are still embeddings of the original problem. Using constraint propagation, and the fact that all the angles are given, we conclude that corresponding edges in the two (scaled) embeddings must be equal in length. Therefore the two embeddings are solutions for the same Complete DGP, and by lemma 2.1 are related by a rigid motion of $\mathbb{R}^d$.*

3.6 The General ADGP

The general ADGP, as shown in chapter 4, is NP-hard, and so a polynomial time algorithm which works in all cases is not expected. The thrust of this work is to advocate careful preprocessing to a general ADGP before submitting it to a general purpose equation solving system.

One of the difficulties of using numerical minimisation methods for solving ADGPs is the choice of initial values. With DGPs, the metric matrix leads to a useful way of choosing initial values which are likely to lie close to the final positions, and correspondingly faster to solve. Unfortunately, no such neat method seems to exist for angle problems. Therefore the only strategy for selection of initial values appears to be to use as much distance data as possible, and hope that the angle data is almost right. For large problems this is a lot to hope for, therefore more effort should be expended on decomposition and transformation. More information about this type of approach is given in chapter 5.

The grand plan for solving ADGP is as follows:

- Test to see if the problem is an easy case, and if so solve it.
- Apply constraint propagation (section 3.4) and relocalisation (chapter 5)
- Test again to see if the problem is an easy case, and if so solve it.
- Use the graph theoretic methods of chapter 5 to try to simplify or decompose the problem.
- Test yet again to see if the problem is an easy case, and if so solve it.

- If all else fails, apply your favourite general problem solver, such as simulated annealing[KGV83], Cylindrical Algebraic Decomposition (chapter 6), Gröbner bases (chapter 7), etc.

3.7 Using ADGP

I conclude this chapter with some examples of the construction of some common geometric structures using the ADGP representation. The purpose of this is that it shows how a geometry tool using ADGP as its model can be used to express geometric constraints at a higher level of abstraction than just lengths and angles. In this way, ADGP may be used as a geometric constraint programming language, for creating objects in terms of constraints.

For example, the graphics system JUNO[Nel85] provides a macro facility for commonly used constraints, such as the requirement that three points form an equilateral triangle. The purpose of this is so that the user sees only the macro, and does not have to work with a large number of low level constraints.

In figure 3.4 are seven common constructions. For each one I give a description and a corresponding ADGP representation, in the form of a macro expansion. Each macro must achieve the required constraint precisely, and not imply a stronger condition that is not implied by its defining sentence. The special macro $new(i)$ defines i to be a node label which is otherwise unused in the ADGP.

(i) Constrain i and j so that their separation is exactly r.

Macro $exact(i,j,r) := L_{ij} = r \wedge U_{ij} = r.$

(ii) Constrain edge jk to be the tangent at j on the circle with centre at i and radius r.

Macro $tangent(i,j,k,r) := \Delta(i,j,r) \wedge \theta_{ijk} = \{\frac{\pi}{2}\}.$

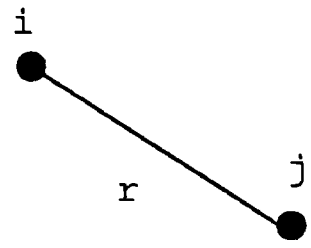
(iii) Constrain node k so that it lies on the line segment between i and j.

Macro $between(i,k,j) := \theta_{ikj} = \{\pi\}.$

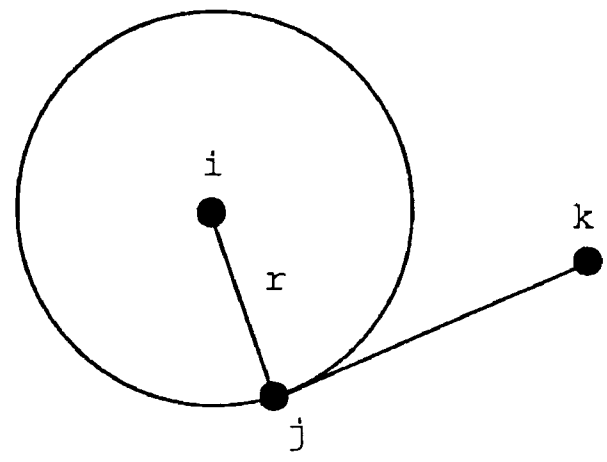
(iv) Constrain nodes i, j and k so that they are collinear.

Macro $collinear(i,j,k) := new(k') \wedge between(i,j,k') \wedge \theta_{ik'k} = \{0\}$

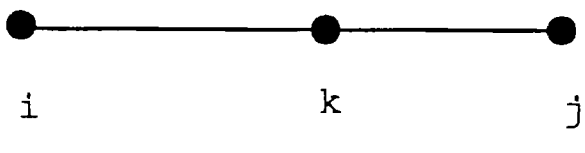
(v) Constrain nodes i, j and k so that they form an isosceles triangle with apex



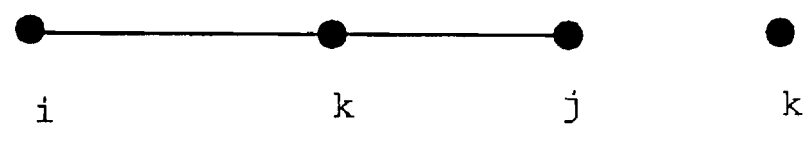
(i)



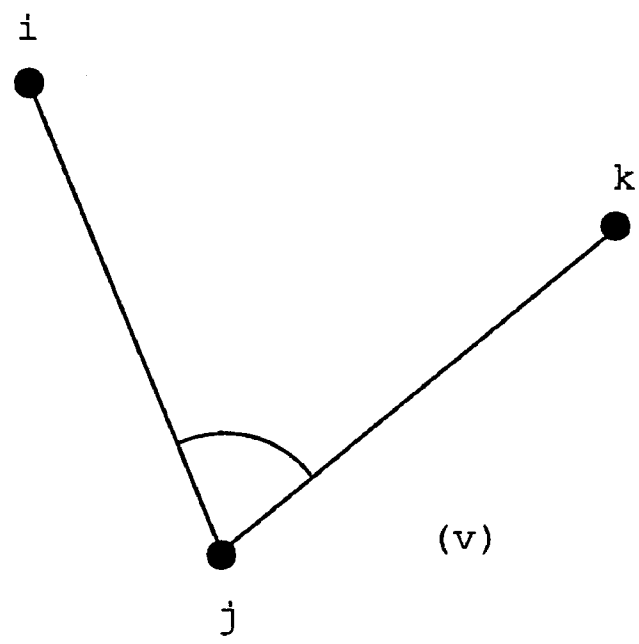
(ii)



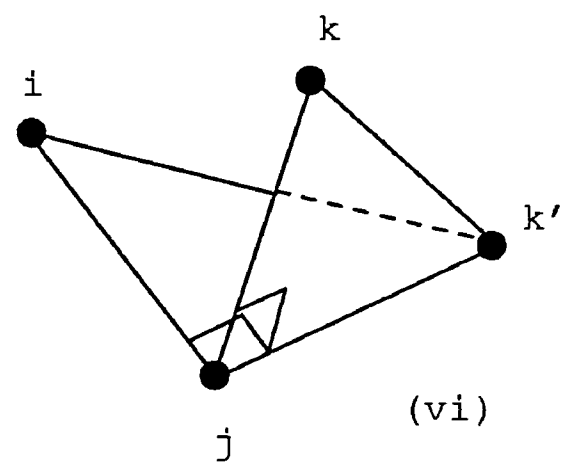
(iii)



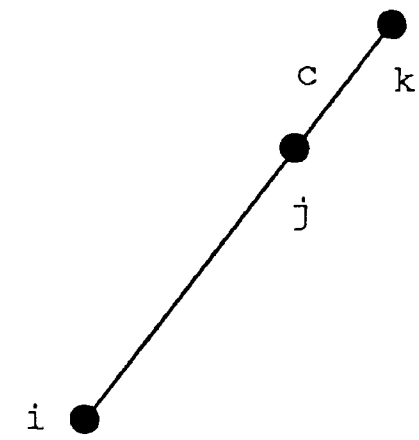
(iv)



(v)



(vi)



(vii)

Figure 3.4: Some examples of ADGP (see text)

angle of α at j.

Macro $isosceles(i,j,k,\alpha) \;:=\; \theta_{jik} = \{\frac{\pi - \alpha}{2}\} \wedge \theta_{jki} = \{\frac{\pi - \alpha}{2}\}$

(vi) In three or more dimensions, constrain nodes i, j and k so that ij has the same length as jk. The extra dimension is required to allow the construction to flex along the edge jk' and so permit the apex angle to vary.

Macro $3d\_isosceles(i,j,k) \;:=\; new(k') \wedge isosceles(i,j,k',\frac{\pi}{2}) \wedge isosceles(k',j,k,\frac{\pi}{2})$

(vii) Constrain the length of ik to be c longer than the length of ij, and j to lie between i and k.

Macro $add\_constant(i,j,k,c) \;:=\; between(i,j,k) \wedge exact(j,k,c)$

Chapter 4

NP and Problem Transformations

4.1 Introduction to NP

When solving a set of algorithmic problems, it is desirable to relate them in terms of complexity, to discover which of them are hardest, which are equivalent, etc. The basic technique for these sorts of comparisons is to attempt to transform one problem to another. To use this method, it must be shown that problem B can be solved quickly if a fast subroutine for solving problem A is available. This demonstrates that problem A is at least as hard as problem B, in some sense, provided that the transformation of B into a problem of type A can be done quickly enough. We say B reduces to A. One of the most important types of reducibility is polynomial time reducibility, which has been used to study NP problems, that is, problems which are solvable in polynomial time by a nondeterministic algorithm[GJ79]. This leads to the notion of NP-hardness, problems which are at least as hard as the hardest problems in NP. NP-complete problems are those problems which are both NP-hard and in NP. At present it is not known whether the NP-complete problems are in P, the set of problems solvable in polynomial time by an ordinary (deterministic) algorithm, although if any one NP-complete problem is in P then NP=P. The practical consequence of this is that until there is some major theoretical advance, solutions for NP-hard problems should be looked for in the following categories: solutions to special cases, heuristics for simplification and approximate solutions.

Strictly, NP-completeness relates only to decision problems, that is, problems which require the answer "yes" or "no." For example, the problem of deciding whether a positive integer is prime is a decision problem. Problems of the form "construct an X if one exists" may be converted into decision problems in an obvious way, to

become "does an X exist?" It is easy to see that such a construction problem is at least as hard as its corresponding decision problem - given a solution to the former it only remains to throw X away and answer "yes." To prove a problem of this type NP-complete it is sufficient to show both that it is NP-hard, and that the problem of checking a candidate solution X for correctness is in P. This second condition is equivalent to showing that the decision problem is in NP.

4.2 Application to DGP and ADGP

The DGP and ADGP are construction problems of the form "construct an embedding if one exists", and so correspond naturally to the decision problem "does an embedding exist?" It is this problem which I show to be NP-hard. Further I show that for numerical versions of DGP and ADGP, the problem of checking that a candidate embedding meets its constraints can be done in polynomial time. Therefore in this case ADGP and DGP decision problems are NP-complete.

The practical consequence of this is that the ADGP and DGP are unlikely (without major breakthrough in the theory) to have efficient algorithmic solutions which work in all cases. However, there may be efficient algorithms which work in some (or even most practical) cases, or which produce approximate answers which are good enough for most purposes. For example, the numerical minimisation techniques discussed in chapter 2 work well in practice to give approximations to embeddings, but cannot guarantee to converge to the correct solution in every case. This is the well known problem of avoiding local minima in the search for the true global maximum. It is important to bear this in mind when using such a method, although it will rarely pose a practical constraint provided some kind of sanity check is imposed on the answers.

In this chapter, I present proofs of the following:

- The algebraic DGP and ADGP are both computable, in the sense that if an embedding exists then an algebraic embedding exists.
- The ADGP and DGP are NP-hard even for some simple special cases.
- That there exist 'small' embeddings for exact DGPs, whereas the same is not true for exact ADGPs.
- ADGP can be restricted to exact angles of π only, without becoming easier.
- The inexact DGP (with some restrictions) reduces to the exact DGP

I also conjecture that in a fixed number of three or more dimensions the algebraic ADGP construction problem cannot be solved in polynomial time, because it may have exponentially large output.

4.3 Algebraic v. Numeric

One of the most awkward parts of the ADGP is doing arithmetic with exact algebraic numbers[Loo82]. For almost all the work in this thesis, the particular type of arithmetic is not important (as long as it can be done efficiently and sufficiently precisely). Here, however, I make a few observations, relevant to the topic of the chapter. I make the following definitions relating to the arithmetic used in an embedding problem. Recall that angle constraints are represented by cosines.

Definition 4.1 *An* **Algebraic** *ADGP (DGP) is a version of ADGP (DGP) which is posed exactly in terms of algebraic real numbers (and possibly the special symbol ∞), and which requires for solution an exact embedding expressed in similar terms. A* **Numeric** *ADGP (DGP) is a version of ADGP (DGP) posed in terms of approximate rationals (and possibly ∞), together with an accuracy parameter, and which requires for solution an embedding expressed in similar terms which satisfies the constraints to within the required accuracy.*

Definition 4.2 *An embedding function is said to be* **algebraic** *if it maps every node to a point in real space such that each co-ordinate is an algebraic number.*

Definition 4.3 *An* **Integer** *ADGP (DGP) is a version of ADGP (DGP) in which all the values in the constraints are integers (or ∞). An* **Integer Embedding** *is an embedding in which the co-ordinates of all the points are integers.*

If a non-algebraic method of solution is to be used at the heart of the algorithm (for example, simulated annealing), then it is sufficient to keep angles and lengths as floating point numbers to a certain accuracy. The numerical analysis required to determine this accuracy and prevent degeneracy is beyond the scope of this work. However it may be supposed that there is a certain number $1 \gg \epsilon > 0$ below which accuracy the constraints are considered to be satisfied, and also a maximum size m for any number in the constraints or in the solution. Then all the numbers used may be represented by no more than $M = \log_2(m) - \log_2(\epsilon)$ bits. I allow M to grow polynomially with n and d, so that larger problems may use more accurate

arithmetic to avoid rounding errors etc. Assuming these facts, the following lemma holds.

Lemma 4.1 *Suppose M as defined above determines the maximum size and accuracy of solution required in any ADGP. Then the time taken to check that the constraints are satisfied (to within this accuracy) by a candidate embedding function $\mathcal{E}$ (of this accuracy) is bounded by a polynomial in M, the size n of the ADGP, and the dimension d of the embedding space.*

Proof *The operations required to check that a distance constraint is satisfied are simply a vector subtraction, a scalar product, two squarings and two comparisons. (The calculation is to test whether $L_{ij}^2 \leq (\mathcal{E}(i)-\mathcal{E}(j))^2 \leq U_{ij}^2$, where the comparisons are to within ϵ.) The time to perform a single multiplication or addition to the required accuracy is at most the square of the number of bits in the representation, $O(M^2)$. This takes at most $O(dM^2 + 2dM^2 + (2+2)M^2)$ for the whole operation, which simplifies to $O(dM^2)$. For an angle constraint $\theta_{ijk} = [\alpha, \beta]$, the operations are two vector subtractions, three scalar products, seven multiplications and two comparisons. (This time the calculation is to test that*

$$\cos^2(\beta) \leq \frac{(\mathcal{E}(i) - \mathcal{E}(j)) \cdot (\mathcal{E}(k) - \mathcal{E}(j))}{(\mathcal{E}(i) - \mathcal{E}(j))^2 (\mathcal{E}(k) - \mathcal{E}(j))^2} \leq \cos^2(\alpha)$$

where the comparisons are again to within ϵ. Recall that the cosines are supplied as the input, so do not require calculation.) The time for the whole operation is therefore at most $O(2dM^2 + 6dM^2 + (7+2)M^2)$ which simplifies to $O(dM^2)$ again. Since there are at most n^2 length constraints and n^3 angle constraints, the total time is less than $O(dM^2n^2 + dM^2n^3) = O(dM^2n^3)$.

Therefore, a candidate embedding function may be checked in time polynomial in n and d, and so the numeric ADGP and numeric DGP are in NP, as discussed in section 4.1. Unfortunately, no such result holds for the algebraically exact problem. Even a simple algebraic problem such as $P(x, y) = x + y$ has arbitrarily complicated real roots, including non-algebraic roots. For *any* real number γ, the assignment $x = \gamma$ and $y = -\gamma$ is a solution. However, there are uncountably many real numbers, and so it is impossible to represent all of them in a finite way (on a computer). Therefore I restrict my attention to the real algebraic numbers, A_R, each of which can be represented finitely as a univariate polynomial with integer coefficients, and a pair of rational numbers to indicate which root of the polynomial is being represented[Loo82]. I show in theorem 4.1 that if an embedding function exists for an algebraic ADGP, then an embedding function exists which can be expressed entirely in terms of algebraic numbers. This shows that the real algebraic

numbers are sufficient for the purposes of solving the algebraic ADGP, however there is another problem. The polynomial $x + y$ mentioned above has algebraic solutions of arbitrarily high degree, and so the problem of checking candidate solutions for x and y to see if $x + y = 0$ takes arbitrarily long. This means that the standard method for showing a problem to be in NP does not work in this case.

4.4 Existence of Algebraic Solutions

I show that if an algebraic ADGP is posed which has any embedding, then it has an algebraic embedding. This shows that the real algebraic numbers are sufficient to represent solutions to the algebraic ADGP.

Lemma 4.2 *Suppose $P(\mathbf{x})$ is a multivariate polynomial over m variables, $\mathbf{x} = \langle x_1 \ldots x_m \rangle$. If there is a vector $\mathbf{v}$ in $\mathbb{R}^m$ such that $P(\mathbf{v}) = 0$, then there is another vector $\mathbf{v}'$ in $\mathbb{R}^m$ such that $P(\mathbf{v}') = 0$, and all of whose co-ordinates are algebraic.*

Proof *The proof proceeds by induction on the number of variables. Suppose there is just one variable, x, then by definition, any root of $P(x) = 0$ is algebraic. For the inductive case, suppose there are $m + 1$ variables, and that the lemma holds for m variables or fewer. Consider the subset V of $\mathbb{R}^{m+1}$ defined by $V = \{\mathbf{v} \in \mathbb{R}^{m+1} \mid p(\mathbf{v}) = 0\}$. I show that if there is a non-algebraic element of V then there is an algebraic one. Suppose $\mathbf{v} \in V$ is non-algebraic. If any of its co-ordinates are algebraic, then they can be substituted into P, resulting in an algebraic polynomial of lower dimension, which can be solved for the remaining variables by the induction hypothesis. Otherwise, consider a cylindrical algebraic decomposition (CAD) of P (see chapter 6). Let C be the cell containing P, then all points in C are roots of P. All the 0-dimensional cells in a CAD have algebraic co-ordinates, and so C must have dimension greater than 0. Therefore C is homeomorphic to some non zero power of the unit interval, and since the algebraic numbers are dense in $\mathbb{R}$, C must contain a point $\mathbf{v}'$ with at least one algebraic co-ordinate. Now the algebraic co-ordinates of $\mathbf{v}'$ may be substituted into P, resulting in an algebraic polynomial of lower dimension. Then as before, P may be solved for the remaining variables by the induction hypothesis.*

Theorem 4.1 *Fix $d \geq 1$, the dimension of an embedding space. Suppose an instance of the ADGP is given in which all the lengths and angle cosines are specified as algebraic numbers, and there is a corresponding embedding in $\mathbb{R}^d$, then there exists an algebraic embedding in $\mathbb{R}^d$.*

Proof *I show in chapters 6 and 7 that all the constraint functions are expressible in terms of polynomials of the co-ordinates of the nodes in the embedding space. Therefore any instance of the ADGP can be recast in the form* $P(\mathbf{x}) = 0$, *where* P *is a multivariate polynomial with real algebraic coefficients, and the elements of the vector* $\mathbf{x}$ *represent the co-ordinates of the nodes. If the problem has* n *nodes, and the embedding space is* $\mathbb{R}^d$, *then* $\mathbf{x}$ *is an* nd*-dimensional vector. The theorem is now an immediate consequence of lemma 4.2.*

It is interesting to see from the proof of lemma 4.2 that if there is an embedding for the particular DGP under consideration, then there must be an algebraic embedding in the same connected component of the abstract solution space, $\mathbb{R}^{nd}$. In geometric terms, this means that the first embedding may be continuously deformed into the other, at all times satisfying the constraint equations. Also, the proof holds for a solution of ADGP with optimisation. In this context the conclusion is that if the target function is maximal (with respect to the constraints) for some embedding function, then is is maximal (with respect to the constraints) for some algebraic embedding function.

4.5 How Hard is the Algebraic ADGP ?

In this section I conjecture that the Algebraic ADGP construction problem cannot be solved in polynomial time. The intuition is that some Algebraic ADGPs result in an exponential amount of output, and that this must therefore take an exponential amount of time. However, I cannot quite achieve this result. Instead I show (lemma 4.3) that there are some Algebraic ADGPs which result in embeddings some of whose co-ordinates must have algebraic degree[Loo82] exponential in the size of input, $O(n)$. This means that the output must contain polynomials (representing those numbers) of degree $\Omega(2^n)$. However, these polynomials may be sparse, and so representable in a very few symbols.

Conjecture 4.1 *The Exact Integer Algebraic ADGP in fixed dimension* $d \geq 3$ *cannot be solved in polynomial time, even if all the (input) constraints are exact integers.*

Lemma 4.3 *There are instances of the Exact Integer Algebraic ADGP of size* $O(n)$ *in fixed dimension* $d \geq 3$ *such that any embedding must contain amongst its co-ordinates an algebraic number of degree* $\Omega(2^n)$.

Proof *The proof revolves around the construction of a figure which requires a length which is an algebraic number of degree* $\Theta(2^n)$. *Consider a right angled triangle with a hypotenuse of length* $x+1$, *and one of its other sides of length* $x-1$. *Then by Pythagoras' theorem the third side is of length* $\sqrt{(x+1)^2-(x-1)^2} = 2\sqrt{x}$. *Suppose an ADGP has a side whose length is* x. *Using the constructions of chapter 3 it is possible to build two sides of a triangle whose lengths must be* $x+1$ *and* $x-1$, *and since this is in 3 or more dimensions, the angles of the triangle can be made arbitrary. Therefore, the appropriate angle can be constrained to be a right angle, and by the argument above, in any embedding the third side has length* $2\sqrt{x}$. *The construction uses only three new nodes, two exact length constraints and can be done with just three exact angle constraints. All the length constraints are* 1, *and all the angle constraints are* $\frac{\pi}{2}$ *(with cosines of* 0*). Starting with* $x = 2$ *(degree* 0*), and using induction, it is therefore possible to construct a side with length* $2\sqrt{2\sqrt{2\ldots}}$ *(with* $\Theta(n)$ *levels of* $\sqrt{}$*) of degree* $\Theta(2^n)$ *with just* n *nodes and* $O(n)$ *integer constraints. Suppose the embedding assigns the endpoints of this edge to be* $\mathbf{p}$ *and* $\mathbf{q}$, *all of whose co-ordinates are algebraic. Let* m *be the maximal degree of all these numbers. Then* $x = \sqrt{(\mathbf{p}-\mathbf{q})\cdot(\mathbf{p}-\mathbf{q})}$. *By standard results of Galois theory[Ste73], the degree of the right hand side is at most* $O(m^d)$, *whereas the left hand side is at least* $\Theta(2^n)$. *Therefore, since* d *is constant,* m *must be at least* $\Omega(2^n)$.

4.6 Integer Exact DGP is NP-hard

I present a reduction of the Integer Exact DGP to the **partition** problem [GJ79] (problem SP12), and thus show it is NP-hard, in any *fixed* number of dimensions $d \geq 1$.

Definition 4.4 *An instance of the* **partition** *problem consists of a finite set* A *and a size function* $s \in A \to \mathbb{N}$. *The question is whether there is a subset* $X \subseteq A$ *such that* $\sum_{a\in X} s(a) = \sum_{a\in(A-X)} s(a)$.

Theorem 4.2 *For any fixed* $d \geq 1$ *the Integer Exact DGP problem is NP-hard, even if it is restricted to integer lengths embedded at points of* $\mathbb{R}^d$ *with integer co-ordinates.*

The theorem is an immediate consequence of the following reductions of DGP to the partition problem.

I start with a proof of NP-hardness for $d = 1$ and indicate how the proof is extended to any $d \geq 1$. The technique is as follows. Given an instance of the partition problem, find an instance of DGP which is embeddable if and only if the partition problem is soluble. To do this, connect together in a chain a set of edges with lengths given by the size function s, and constrain them to be collinear (this is trivial in $d = 1$). Some of the edges will "point" to the left and others to the right. If it is possible for the two ends of the chain to meet then the sum of the left pointing edges is equal to the sum of the right pointing edges.

Lemma 4.4 *The Integer Exact DGP in* one *dimension reduces to the partition problem.*

Proof *Given an instance (A, s) of the partition problem, construct a DGP as follows. Suppose the elements of A are $1 \ldots n-1$, then the DGP has n nodes. For $i = 1 \ldots n-1$ set $L_{i,i+1} = U_{i,i+1} = s(i)$. Finally, set $L_{1,n} = U_{1,n} = 0$. Suppose there is an embedding function $\mathcal{E}$. Construct a solution for the partition problem by choosing X to be those points i for which $\mathcal{E}(i) \leq \mathcal{E}(i+1)$. It follows from the geometry that the sum of the edges in X must equal the sum of the edges not in X, otherwise $\mathcal{E}(1) \neq \mathcal{E}(n)$, in violation of the constraints. Conversely, if X is a solution to the partition problem construct an embedding by starting with $\mathcal{E}(1) = 0$. For each i, if $i \in X$ choose $\mathcal{E}(i+1) = \mathcal{E}(i) + s(i)$ otherwise choose $\mathcal{E}(i+1) = \mathcal{E}(i) - s(i)$. It follows easily that $\mathcal{E}(1) = \mathcal{E}(n)$ and that the constraints are satisfied.*

For higher dimensions the only problem is keeping the chain edges collinear. This is possible using $(3, 4, 5)$ right angled triangles to build rigid frames to which the chain edges are fixed, as shown below. The requirement is that when two frames are joined end to end the only non-rigid motion possible is a reflection perpendicular to the line in which the chain edges lie.

Consider one end of such a frame in $d > 1$ dimensions. I take d nodes, $a_1 \ldots a_d$, which are constrained to lie at distance 4 from the end node, c, of the chain edge cc' of length s. Another d nodes, $b_1 \ldots b_d$, are constrained to lie on the lines connecting c to each of the a_i and at distance 3 from c. This is achieved by constraining $\Delta(c, a_i) = 4$, $\Delta(c, b_i) = 3$ and $\Delta(a_i, b_i) = 1$ for each $i = 1 \ldots d$. The directions from c to each of the a_i are then constrained to be mutually perpendicular with the $O(d^2)$ constraints $\Delta(a_i, b_j) = 5$ for $i, j = 1 \ldots d$ and $i \neq j$. Finally the chain edge is bound to the frame, collinear with the edge ca_1, by constraining $\Delta(c', a_1) = \mid s - 4 \mid$. Two chain edges are joined by identifying one each of their endpoints, and the associated $a_2 \ldots a_d$ frame points. In $\mathbb{R}^d$ any pair of vectors perpendicular to the same set of $d-1$ mutually perpendicular vectors must be parallel, therefore two adjacent chain edges are parallel, and are free to point in the same or opposite

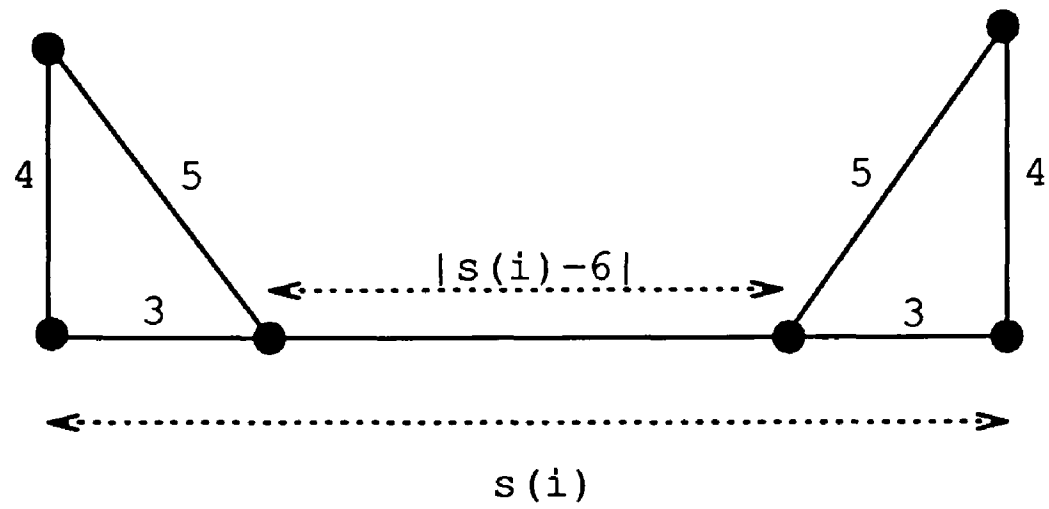


Figure 4.1: Special case construction of chains in two dimensions for theorem 4.2

directions. Further, the construction uses at most $O(nd^2)$ nodes and constraints. Therefore, as before, in fixed d dimensions the integer exact DGP reduces to the partition problem.

In figure 4.6 I illustrate a simpler construction for $d = 2$.

Corollary 4.1 *The Numeric Integer Exact DGP in any fixed number of dimensions greater than* 0 *is NP-complete.*

Proof *By theorem 4.2 the problem is NP-hard. By lemma 4.1 it is in NP. Therefore it is NP-complete.*

The proof of theorem 4.2 also shows that the Integer Exact DGP is NP-hard even if solution is restricted to an integer embedding.

4.7 ADGP is NP-hard

In this section, I show that the ADGP with exact lengths is NP-hard, using a reduction from the **subset sum** problem [GJ79](problem SP13). Further, I am able to show that the Integer Exact ADGP is NP-hard for any embedding space dimension strictly greater than 0. That is, the problem of deciding whether there are any embeddings of the ADGP in any number of dimensions is NP-hard. This is slightly more useful than it looks at first glance, for if there is an embedding function in a space of dimension $d \geq n$, then it can be projected into $\mathbb{R}^{n-1}$ (because n points cannot span more than $n-1$ dimensions in a vector space). Further, since any $\mathbb{R}^d$

can be considered to be a subspace of $\mathbb{R}^{n-1}$ whenever $d \leq n-1$, it is the case that if there is any embedding for an instance of ADGP, then there is an embedding in $\mathbb{R}^{n-1}$.

Definition 4.5 *An instance of the* **subset sum problem** *consists of a triple* $(k, n, \langle w_1, \ldots, w_n \rangle)$, *where* k, n *and all the* w_i *are non zero natural numbers. The problem is to decide if there is some subset* S *of* $\{1, \ldots, n\}$ *such that* $\sum_{i \in S} w_i = k$.

To show that ADGP is NP-hard, I show how any instance of the subset sum problem can be solved by converting it (in polynomial time) to an ADGP which is embeddable if and only if the subset sum problem is. Since the subset sum problem is itself NP-hard, it follows that ADGP is too. In fact, I show that the Integer ADGP is NP-hard, since the construction uses only integers.

Theorem 4.3 *The subset sum problem reduces to the Integer ADGP for any Euclidean embedding space of dimension at least one.*

Proof *Suppose* $(k, n, \langle w_1, \ldots, w_n \rangle)$ *is an instance of the subset sum problem. The essence of the construction is to use the bases of a chain of* n *(degenerate) triangles to represent either the inclusion or the exclusion of* i *from* S, *and to force them to lie in a straight line so that their combined length represents the sum of the included elements.*

Let the nodes in the construction be $b_1 \ldots b_{n+1}$ *to form the bases of the triangles, and* $a_1 \ldots a_n$ *to represent the apexes. The* i*th triangle has vertices* b_i, b_{i+1} *and* a_i, *and is constrained by setting the included angle at* b_i *to be* 0, *and setting the side lengths* $\Delta(b_i, a_i) = 1 + w_i$ *and* $\Delta(b_{i+1}, a_i) = w_i$. *Then in any embedding it is the case that* $\Delta(b_i, b_{i+1})$ *is either* 1 *or* $1 + 2w_i$ *(by the cosine rule). The chain is forced to lie in a straight line by specifying the angle* $b_{i-1} b_i b_{i+1}$ *to be* π *for* $i = 2 \ldots n-1$. *Finally the length of the chain is specified by setting* $\Delta(b_1, b_n) = n + 2k$. *Then in any embedding it is the case that* $\sum \Delta(b_i, b_{i+1}) = n + 2k$.

It remains to show that this is a correct reduction. Suppose the ADGP has an embedding. Take S *to be the subset of* $\{1 \ldots n\}$ *such that* $\Delta(b_i, b_{i+1}) = 1 + w_i$ *in this embedding. Then from the constraints of the embedding*

$$\sum_{i \in S} (1 + 2w_i) + \sum_{i \notin S} 1 = n + 2k$$

Therefore $\sum_{i \in S} w_i = k$ *and so the subset sum problem is soluble. Conversely, if the subset sum problem is soluble, then there is a set* S *such that* $\sum_{i \in S} w_i = k$. *There is an embedding for the corresponding ADGP, formed by placing all the nodes on the* x*-axis with* b_j *at* $x = j + 2\sum_{i \in S} w_i$ *and* a_j *at* $b_j + 1 + w_j$.

The proof of theorem 4.3 also shows that the Integer ADGP with exact lengths is NP-hard even if solution is restricted to an integer embedding.

Corollary 4.2 *The Numeric Integer ADGP with exact lengths in any number of dimensions greater than* 0 *is NP-complete.*

Proof *By theorem 4.3 the problem is NP-hard. By lemma 4.1 it is in NP. Therefore it is NP-complete.*

4.8 How Large must an Embedding be ?

In this section I show that DGP embeddings are essentially small, that is, can be restricted to a small region of $\mathbb{R}^d$. On the other hand, there are exact ADGPs all of whose embeddings are very large. In this sense, the ADGP is a more expressive language than is the DGP. The first step is to define some terms concerning the size of some objects.

Definition 4.6 *The* **diameter** *of a finite set S of points in $\mathbb{R}^d$ is defined by $diam(S) = \max\{\Delta(x, y) \mid x, y \in S\}$. The diameter of an embedding function $\mathcal{E}$ is the diameter of its range, $diam(\mathcal{E}) = diam(\operatorname{ran} \mathcal{E})$. The diameter of an instance of DGP is defined to be the sum*

$$\sum \{U_{ij} \mid U_{ij} \neq \infty\} + \sum \{L_{ij} \mid U_{ij} = \infty\}$$

The following lemma shows that the diameter of an exact DGP is related to the diameter of its possible embeddings. The graph theoretic terms used in the proof are explained in chapter 5. This result extends to the general DGP, as I prove in theorem 4.4. I present the special case first because the proof is easier to follow.

Lemma 4.5 *Suppose an exact DGP has diameter D and is embeddable. Then there exists an embedding which can be contained in the sphere of radius D centered at the origin.*

Proof *Let G be the constraint graph corresponding to this instance of the exact DGP, and suppose G is a connected graph. Suppose that $\mathcal{E}$ is an embedding function for the structure. Then I claim that $diam(\mathcal{E}) \leq D$. If not, then there exist nodes i and j such that $\Delta(\mathcal{E}(i), \mathcal{E}(j)) > D$. But since G is connected, there is a (simple)*

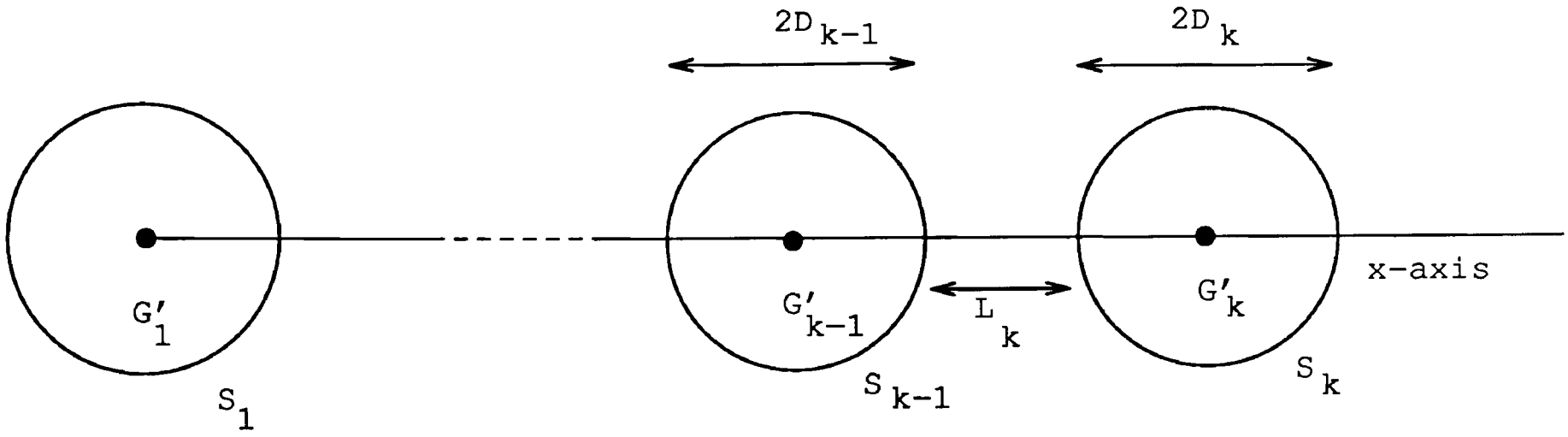


Figure 4.2: Proof of theorem 4.4

path from i to j in G, and the length of that path cannot be greater than D, the sum of all the edge lengths. By the triangle inequality $\Delta(\mathcal{E}(i), \mathcal{E}(j)) \leq D$ and this is a contradiction, so the claim is proved. Translate $\mathcal{E}$ so that one of the nodes is embedded at the origin. Then each point of the embedding lies within the sphere of radius D centred on the origin.

If G is not connected, repeat the argument with each connected component, and take the union of the translated embedding functions for each component to be the new embedding function. This function maps all the nodes into a sphere of radius D.

As an added bonus, it can be seen from the proof of lemma 4.5 that the low diameter embedding can be computed easily from the original embedding.

Theorem 4.4 *If an instance of DGP has diameter D and is embeddable, then there exists an embedding of diameter at most $2D$.*

Proof *See figure 4.2. Suppose the instance of DGP is embeddable, and has embedding $\mathcal{E}$. Consider the constraint graph G corresponding to the problem. The edges of G fall into two sets, E_B and E_U, where the E_B are the edges with finite upper bounds, and the E_U are the edges with non-zero lower bounds and infinite upper bounds. Let G' be the subgraph of G with the same set of nodes, but formed by ignoring those edges which belong to E_U. Consider the connected components $G'_1 \ldots G'_c$ of G'. For each $k = 1 \ldots c$, let $\mathcal{E}_k$ be the restriction of $\mathcal{E}$ to the nodes of G'_k. Let D_k be the sum of the upper bounds of the edges in G'_k.*

Consider a pair of nodes i and j in G'_k. Since G'_k is connected, there is a (simple) path from i to j in G'_k made up of bounded edges. By the triangle inequality the distance between i and j in $\mathcal{E}_k$ cannot exceed the maximum length of that path, which in turn cannot exceed D_k. Therefore each $\mathcal{E}_k$ lies in a sphere S_k of radius $2D_k$.

The distance constraints within each of the G'_k are satisfied by the $\mathcal{E}_k$, so it remains only to satisfy the constraints between different G'_k. All these constraints are of the unbounded above type, in E_U. Therefore to satisfy the constraints the spheres must be moved far enough apart, and to prove the theorem they must be kept close enough together. The spheres are 'moved' by rigid transformations of the $\mathcal{E}_k$, which preserve internal distances.

For each k define L_k to be the length of the longest lower bound amongst those edges which have one endpoint in G'_k and the other in $G'_{k'}$ for some $k' < k$. The spheres are arranged in order along the x-axis so that the separation between two adjacent spheres S_{k-1} and S_k is L_k.

It is easy to see that this new embedding satisfies all the constraints. Furthermore, I claim that it has a diameter of less than $2D$. To see this, note that the sum of the L_k is no more than the sum of the lower bounds of edges in E_U. Further, the diameters of the spheres sum to $2\sum D_k$ which is no more than twice the sum of the upper bounds of edges in E_B. Adding these together and comparing with the definition of diameter of a DGP, the diameter of the whole construction is no more than $2D$.

I show that a similar theorem cannot hold for the case of exact ADGP. The method is to use $O(n)$ constraints to construct a very small angle, of size $\Theta(2^{-n})$ radians. Then a triangle with this angle at the apex, and a base of size 1 has a height of about $\Theta(2^n)$. The construction is valid in any Euclidean embedding space of dimension of at least one.

The nodes of the construction are labelled a, $b_1 \ldots b_n$, $c_1 \ldots c_n$ and $d_1 \ldots d_n$. Node a is the apex of the large triangle, and the pairs b_i, d_i form successively more distant bases as i increases up to n. Each point c_i lies half way between b_i and d_i and serves to roughly halve the apex angle for each new i. This is illustrated in figure 4.3. The constraints are as follows:

To start everything off neatly, constrain the length ab_1 to be 1.
For each $i \leq n$, constrain the angle ab_id_i to be $\frac{\pi}{2}$.
For each $i \leq n$, constrain the angle $b_ic_id_i$ to be π.
For each $i < n$, constrain the angle ab_ib_{i+1} to be π.

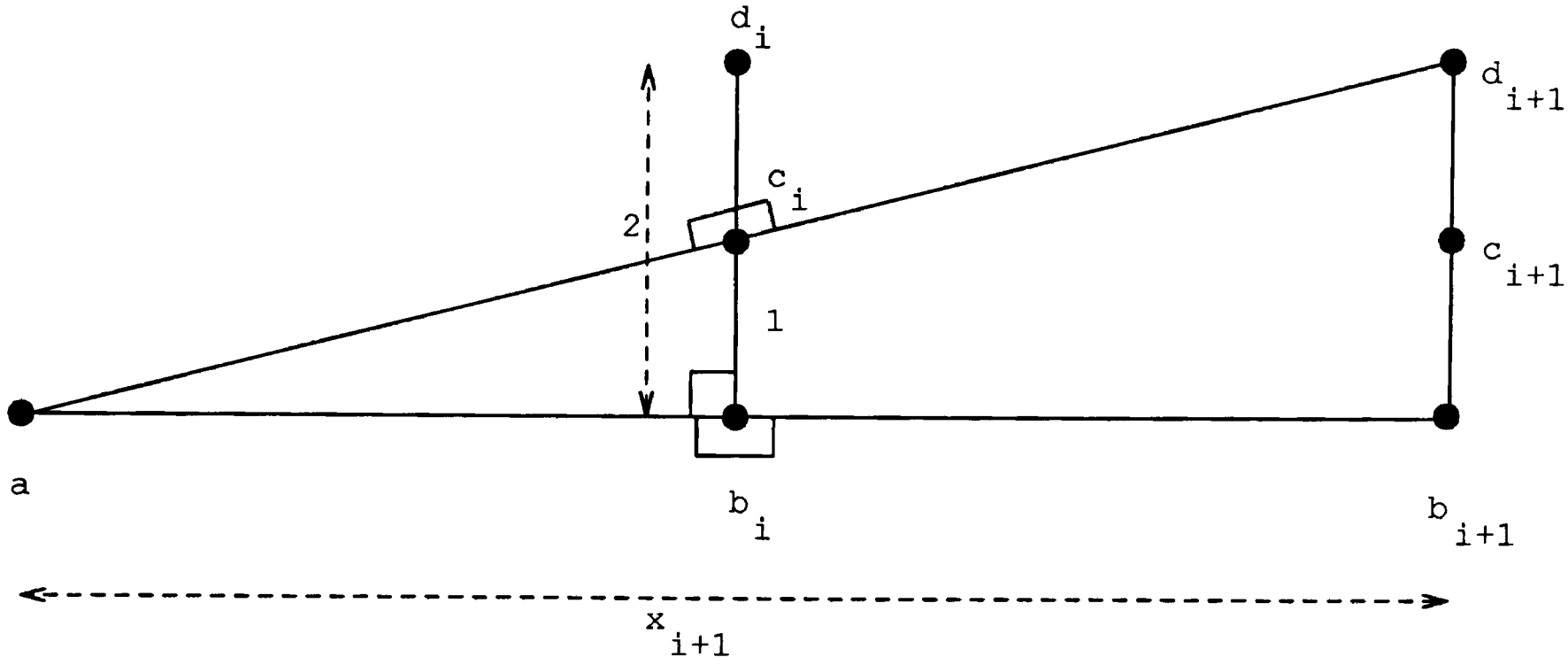


Figure 4.3: Constructing long edges with ADGP

For each $i < n$, constrain the angle $ac_i d_{i+1}$ to be π.
For each $i \leq n$, constrain the length $b_i d_i$ to be 2.
For each $i \leq n$, constrain the length $b_i c_i$ to be 1.

There are $O(n)$ constraints, and all the values in the constraints are independent of n, so the input is "small" in some sense. An exact DGP with this kind of input has diameter at most $O(n)$, and so there is an embedding of diameter at most $O(n)$. However I now show that any embedding of this ADGP has diameter at least 2^n.

To achieve this I show that the distance ab_{i+1} is exponential in n in any embedding. Define x_i to be the length ab_i in any embedding. Now x_1 must be 1, directly from the constraint on ab_1. The triangle $ab_{i+1}d_{i+1}$ is similar to the triangle $ab_i d_i$. This is because they both have a right angle, and the two pairs of sides ab_i, ab_{i+1} and ad_i, ad_{i+1} are collinear and share the node a. Therefore they share the same angle at a. However, the length $b_{i+1}d_{i+1}$ is twice that of $b_i d_i$, so by similar triangles x_{i+1} is twice x_i. Therefore $x_n = 2^n$, and so the diameter is at least 2^n.

4.9 Exact versus Inexact Lengths

In this section I demonstrate that problems with exact lengths are as hard to solve as graphs with inexact lengths (and finite bounds), even though at first they may seem easier. First I say what it means to transform an embedding problem, but without essentially altering it.

Definition 4.7 *Suppose N and N' are the sets of nodes of a pair of instances P and P' (respectively) of ADGP or DGP, and that $N \subseteq N'$. I say that P and P' are* **equivalently embeddable** *if and only if (1) every embedding $\mathcal{E}$' of P' when restricted to the nodes in N is an embedding for P and (2) for every embedding $\mathcal{E}$ of P there is an embedding $\mathcal{E}$' of P' such that $\mathcal{E}$' restricted to N is equal to $\mathcal{E}$.*

Suppose an instance of ADGP is given, and that ij is an edge which is not exact. Provided the upper limit U_{ij} is not infinite, it is possible to exchange this single inexact edge for two exact ones, and an extra node. Let k be a label for a node which has not previously been used. Consider the new problem obtained by setting $L_{ik} = U_{ik} = \frac{1}{2}(L_{ij} + U_{ij})$ and $L_{jk} = U_{jk} = \frac{1}{2}(U_{ij} - L_{ij})$, and removing the length constraints on $\{i, j\}$. Any embedding function $\mathcal{E}$ for the new graph must satisfy the triangle inequality, and so

$$\begin{aligned} \Delta(\mathcal{E}(i), \mathcal{E}(j)) &\leq \Delta(\mathcal{E}(i), \mathcal{E}(k)) + \Delta(\mathcal{E}(j), \mathcal{E}(k)) \\ &\leq U_{ik} + U_{jk} \\ &\leq \frac{1}{2}(L_{ij} + U_{ij}) + \frac{1}{2}(U_{ij} - L_{ij}) \\ &\leq U_{ij} \end{aligned}$$

and

$$\begin{aligned} \Delta(\mathcal{E}(i), \mathcal{E}(j)) &\geq \Delta(\mathcal{E}(i), \mathcal{E}(k)) - \Delta(\mathcal{E}(j), \mathcal{E}(k)) \\ &\geq U_{ik} - U_{jk} \\ &\geq \frac{1}{2}(L_{ij} + U_{ij}) - \frac{1}{2}(U_{ij} - L_{ij}) \\ &\geq L_{ij} \end{aligned}$$

Furthermore, it can be seen that given any embedding of the original problem, an embedding of the new node k can be found which satisfies the new constraints. Therefore the original and transformed problems are equivalently embeddable. This constitutes a proof of the following theorem.

Theorem 4.5 *If an instance of DGP has all its length constraints either completely undefined or finitely bounded above then it may be transformed in polynomial time to an equivalently embeddable Exact DGP.*

Thus the only sort of edge which is not suitable to be changed into an exact edge is one which has a non zero lower bound and no upper bound. This type of edge, however, will often be found to have a finite upper bound after constraint propagation (or bound smoothing) has been used (see chapters 2 and 3). Also, in the light of theorem 4.4, it can be seen that an a priori upper bound of twice the diameter of the DGP can be placed on all the edges. In the case of ADGPs, I show in the next section that angle constraints can be used to make even these constraints exact.

4.10 Exact versus Inexact Angles

In this section I demonstrate that it is possible to use exact angles to represent inexact angles. As a consequence of this, angle distance geometry with exact angles is shown to be as hard as with inexact angles. Furthermore, I show that the only angle required is π, and that all the other angles can be constructed using this. Also, I show that all the lengths in an ADGP can be made exact.

The construction used in this reduction is illustrated in figure 4.4. Suppose an instance of the ADGP with inexact angles is given. Such an instance can be transformed into an equivalently embeddable instance of ADGP with exact angles. Only a constant number of extra nodes and constraints are required to transform each angle constraint. Because there are at most $O(n^3)$ angle constraints (where n is the number of nodes in the original ADGP) the number of extra nodes required is $O(n^3)$ and the transformation can be performed in time $O(n^3)$.

The construction works like a pair of scissors. Suppose the angle between nodes i, j and k is constrained to lie between α and β (where $0 \leq \alpha \leq \beta \leq \pi$). Two new nodes i' and k' are created, and length constraints are chosen in the triangle $i'jk'$ to restrict the internal angle at j to precisely this range (the blunt end of the scissors). A suitable choice is $L_{i'j} = U_{i'j} = 1$, $L_{k'j} = U_{k'j} = 1$, and $L_{i'k'} = 2\sin(\frac{\alpha}{2})$, $U_{i'k'} = 2\sin(\frac{\beta}{2})$. Note that $\cos(\alpha)$ and $\cos(\beta)$ are given in the original constraints (angles are represented by their cosines), and so $\sin(\frac{\alpha}{2}) = \sqrt{\frac{1}{2}(1-\cos\alpha)}$ and $\sin(\frac{\beta}{2}) = \sqrt{\frac{1}{2}(1-\cos\beta)}$ are algebraic, and can be calculated in polynomial time[Loo82]. Finally, the edges ij and kj (the blades of the scissors) are constrained to lie parallel to $i'j$ and $k'j$ respectively, so that the angle in ijk is equal to the angle in $i'jk'$. This is achieved with $\theta_{i'ji} = \theta_{k'jk} = \{\pi\}$.

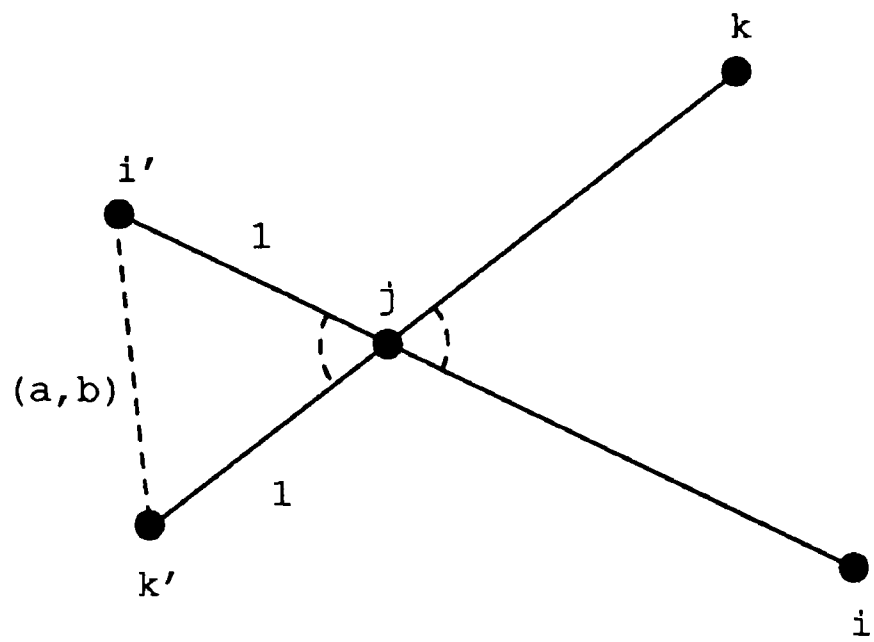


Figure 4.4: Making angles exact (see text).

The following points should be noted. (1) If both edges ij and kj are exact, then it is sufficient to compute constraints for the edge ik using the cosine rule, without having to use the extra nodes i' and k'. (2) The only angle constraints introduced are angles of π. Therefore the exact angle π is sufficient to represent all other angles. (3) The new and old problems are equivalently embeddable (by the cosine rule).

Finally, I show that in an ADGP all the length constraints can be made exact. First consider all the length constraints with finite upper bounds. These can be replaced by exact edges using the procedure in section 4.9. Secondly, suppose there is an edge ij with constraints $L_{ij} > 0$ and $U_{ij} = \infty$. To make this into exact constraints, choose a new node k, and define new constraints $L_{ik} = L_{ij}$, $U_{ik} = L_{ij}$ and $\theta_{ikj} = \{\pi\}$. Remove the old constraint on L_{ij}. Note that the only angle introduced is the exact angle π. Furthermore, since there are at most $O(n^2)$ length constraints in an ADGP with n nodes, the transformation takes time at most $O(n^2)$ and adds at most $O(n^2)$ new nodes.

These constructions form the proof of the following theorem.

Theorem 4.6 *Any instance of ADGP can be transformed in polynomial time to an equivalently embeddable instance of ADGP with the property that all its length constraints are either exact or completely unconstrained, and all its angle constraints are either the exact angle π or are completely unconstrained.*

Chapter 5

Graph Theoretic Algorithms

In this chapter I examine the use of graph theory to simplify some instances of ADGP. (In general, it should be assumed whenever I mention ADGPs that this includes DGPs too, which are a special case of the former.) Lovász and Yemini[LY82] and Laman[Lam70] show how graph methods can be used to derive properties of exact distance embeddings in the plane. Their work concerns graphs whose embeddings cannot be continuously deformed in the plane whilst maintaining the distance constraints. They produce characterisations of such 'rigid' structures in terms of the constraint graph only, without the distance information. Unfortunately they have to make assumptions about degeneracies which are hard to interpret in terms of DGPs, and further the methods work only in two dimensions.

An ADGP can be made to correspond to a graph which represents the topological properties of the problem, without special emphasis on the metrical properties. Although the techniques in this chapter assume the embedding space is Euclidean, its dimension is not important for the most part so long as it is at least two.

Once a graph has been obtained from an embedding problem it may be simplified using one of two strategies which I describe. The first, called **graph decomposition** involves breaking the graph into subgraphs each of which represents, in an obvious way, a subproblem of the original embedding problem. These parts are solved independently, perhaps in parallel, by whatever means are appropriate and their solutions combined. In some cases one or more of the subproblems may be of an easier type to solve. For example, a partial angle problem might split into a complete angle problem and a complete length problem. In all cases when I give a complexity measure for a graph decomposition method, the measure relates only to the time taken for the decomposition and recombination itself, and not to the time taken solving the subproblems which the decomposition produces.

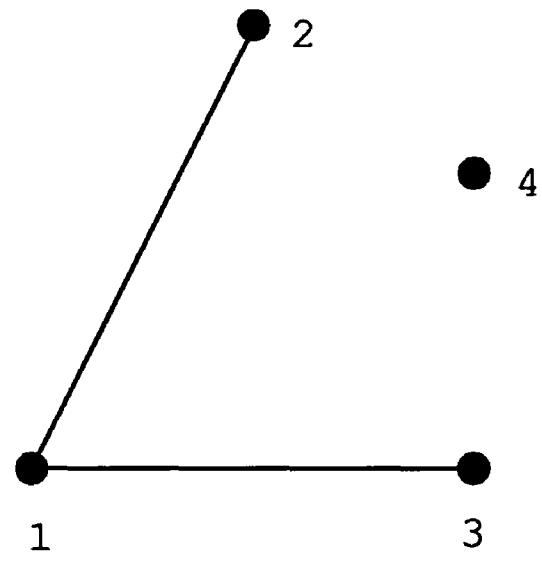


Figure 5.1: A representation of the graph $(\{1,2,3,4\},\{\{1,2\},\{1,3\}\})$

The second simplification strategy replaces sections of the problem by smaller or simpler pieces, with the property that an embedding for the altered problem can be transformed (easily) into an embedding for the original problem. This type of strategy is called **graph transformation**.

5.1 Turning ADGPs into Graphs

The basic mathematical structure underlying the work of this chapter is the undirected graph, defined below. (The symbol $\mathbb{P}$ denotes the power set, or set of all subsets of a set).

Definition 5.1 *An (undirected)* **graph** $G = (N, E)$, *consists of a set* N *of* **nodes** *and a set* $E \subseteq \mathbb{P}(N)$ *of (undirected)* **edges** *such that each element of* E *contains precisely two distinct elements of* N. *If* $\{i,j\} \in E$, *then there is said to be an edge between* i *and* j *in* G. *A graph* $G' = (N', E')$ *is a* **subgraph** *of* $G = (N, E)$ *if* $N' \subseteq N$ *and* $E' \subseteq E$. *If* $X \subseteq N$ *then a subgraph of* G, *denoted* $G - X$, *is obtained by removing the nodes of* X *from* N *and any adjacent edges from* E. *The node set of this graph is* $N - X$, *and the edge set is* $E \cap \mathbb{P}(N - X)$.

A graph can be visualised as a collection of points to represent the nodes, with lines connecting pairs of nodes to represent the edges (figure 5.1).

The first task is to relate each instance of ADGP to a particular graph, which I call the **constraint graph** of the problem. The node set, N, is taken to be

the set $\{1 \ldots n\}$ of vertex indices. The idea is to place an edge between nodes i and j if there is a direct constraint relating the positions of vertices i and j in the embedding. For length constraints this is quite natural: include $\{i, j\}$ in E if $L_{ij} > 0$ or $U_{ij} < +\infty$. For angle constraints there are several possibilities. One is to have a separate graph for the angles, whose nodes correspond to pairs of vertices. Another possibility is to include an edge in E whenever it lies opposite a defined angle. The one I have found most useful is: if $\theta_{ijk} \neq \infty$ then include $\{i, j\}$, $\{j, k\}$ and $\{i, k\}$ in E.

Definition 5.2 *For any ADGP with constraint graph $G = (N, E)$, there are two subsets of the edges which are of special importance. I define the* **exact edges** *$E_X \subseteq E$ to be those edges for which the constraints are exact, $\{i, j\} \in E_X \Leftrightarrow U_{ij} = L_{ij}$. In an Exact DGP E_X is equal to E. Secondly, I define the* **angle edges** *$E_A \subseteq E$ to be those edges which form part of a triangle any of whose angles are constrained. That is, $\{i, j\} \in E_A$ if and only if there exists k such that any one of θ_{ijk}, θ_{jki} or θ_{kij} is defined.*

A subgraph of a constraint graph can be seen to imply a subproblem of the original ADGP in an obvious way. There is one vertex for each node in the subgraph, and the constraints are those which apply only to that set of vertices. These can easily be extracted (and renumbered if necessary) from the original problem.

5.2 Constraint Propagation and Relocalisation

In chapters 2 and 3 I discussed the concept of constraint propagation, and an algorithm for it. Whilst this method is useful in many situations, some of its effects can be detrimental, and serve only to make a problem more complicated. I propose, therefore, a graph transformation method which I call **relocalisation**, that removes some of these side effects. The graph theoretic methods used in this chapter work best when most of the edges of E are in E_X. Unfortunately, the constraint propagation method often adds redundant edges to E which are not exact. For example, in figure 5.2 the graph on the left gives rise to the graph on the right under this method. The new edge created with partial information is of little use, since it is not exact. By definition of constraint propagation the new edge constraint is satisfied by *any* embedding of the other constraints. It can therefore be removed safely, without affecting the solution of the problem. This reduced graph is more suited to the graph theoretic approach of this chapter.

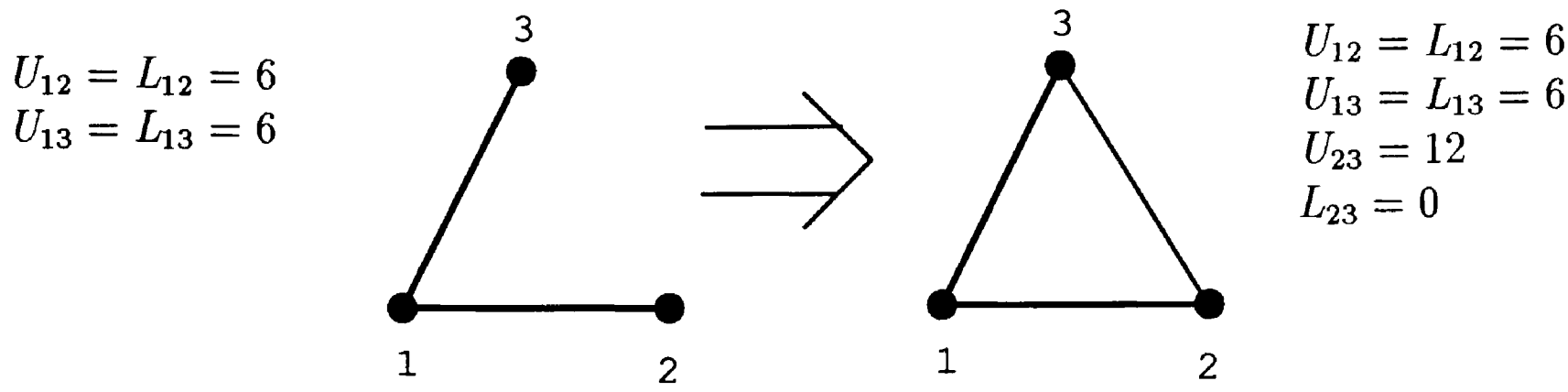


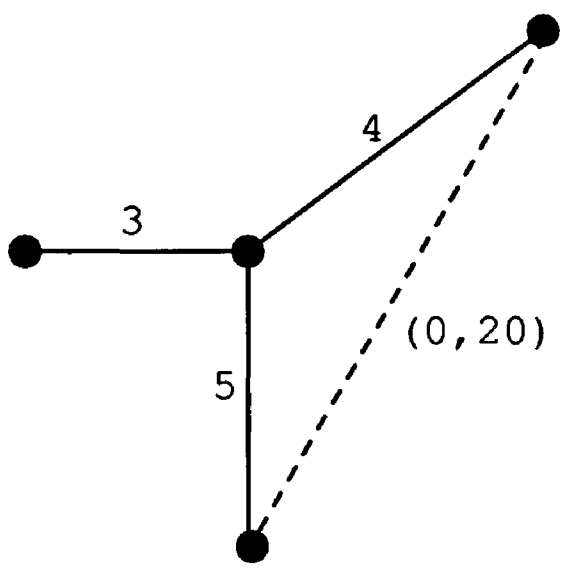
Figure 5.2: An example of constraint propagation

I propose the following two stage algorithm for relocalisation. Suppose $G = (N, E)$ is the constraint graph of an ADGP (with exact edge set E_X), and that these become $G' = (N, E')$ and E'_X after constraint propagation. The first stage is to remove those inexact edges which were added unnecessarily by constraint propagation, the result of this stage being a new graph with edges $E \cup E'_X$. The second stage is to remove those inexact edges which were unnecessary in the first place. This is done in the following way. For each inexact, non-angle edge e in the graph ($e \in E - (E_X \cup E_A)$), remove it, and then calculate new constraint values for it using constraint propagation. If the new value is equal to the old, then the old value is not necessary and e may be discarded. If it is unequal, then the old value is necessary, and e is reinstated before continuing. It is important never to remove an angle edge (E_A) because then there would be no record in the graph of the corresponding angle constraint.

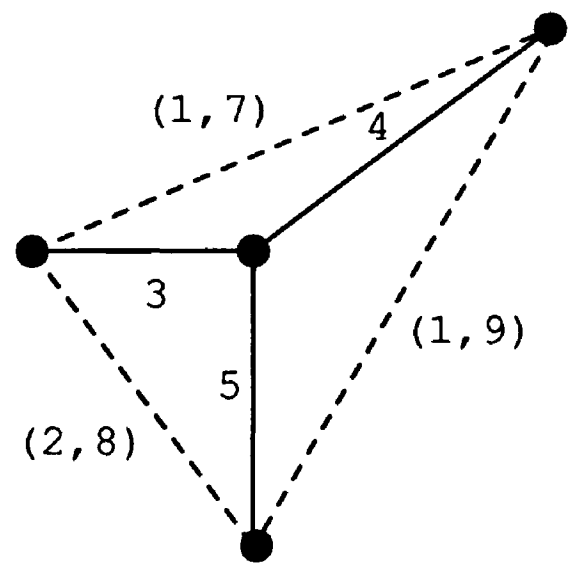
The following example should make this method clear. Consider the graph in figure 5.3. Here a solid line represents an edge in E_X, and a dashed line represents an edge in $E - E_X$. A number, x, by an edge represents an exact length ($U_{ij} = L_{ij} = x$), and a pair of numbers (x, y) represents a non-exact edge ($L_{ij} = x$, $U_{ij} = y$). The stages represented by the diagrams are: (a) the initial graph (b) after constraint propagation (c) relocalisation (stage 1) (d) relocalisation (stage 2).

5.3 Connected Components

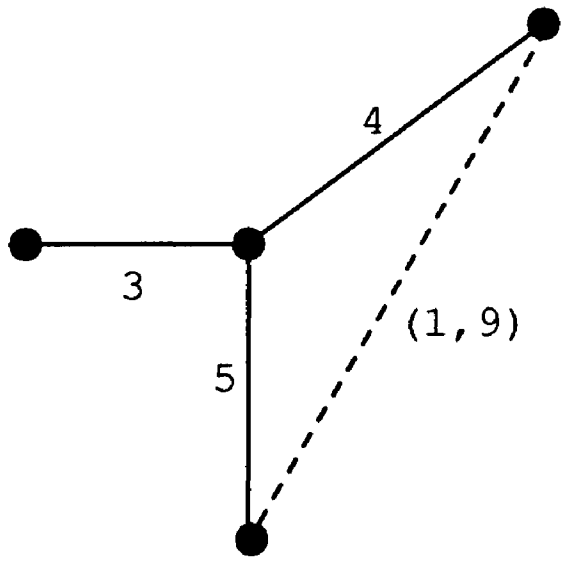
The most straightforward graph decomposition tactic for an ADGP involves the use of the connected components of a graph.



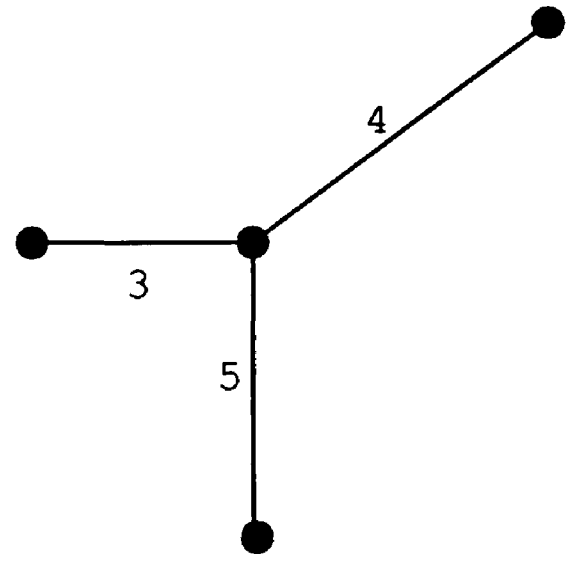
(a)



(b)



(c)



(d)

Figure 5.3: An example of relocalisation (see text)

Definition 5.3 *A* **path** *between two nodes i and j in a graph is an ordered list of vertices such that each vertex in the list is connected to its successor by an edge in the graph. A graph is* **connected** *if there is at least one path between any pair of nodes. If a graph is not connected, it may be (uniquely) partitioned into subgraphs, each of which is connected, and such that any pair taken together is not connected[Sed88]. These subgraphs are call the* **connected components** *of the graph.*

It is clear that if the graph corresponding to a DGP is not connected, then its connected components may be embedded independently. This is because nodes in different connected components can exert no constraints on each other. For a proof of this, consider the constraint equations (chapter 3). If a pair of nodes are constrained by any of these equations then from the definition of a constraint graph (section 5.1) there is an edge between them. A single edge constitutes a path between its endpoints, and so both nodes lie in the same connected component.

A possible first step in solving a DGP is to find the connected components of the corresponding graph, and solving each component separately. The connected components of a graph with n nodes and e edges can be found using depth first search in time $O(n + e)$ [Sed88][Tar72]. In fact, this type of graph decomposition can be merged with the method of biconnected components in the next section.

5.4 Biconnected Components

A better decomposition method requires the concept of biconnectivity.

Definition 5.4 (Biconnectivity) *Two paths are said to be* **non-intersecting** *if they have no nodes in common, apart from (possibly) one or both endpoints. A graph is said to be* **biconnected** *if and only if for any two distinct nodes a and b there are (at least) two non-intersecting paths from a to b. As a special case, any single node is biconnected, as are any pair of nodes connected by an edge (see figure 5.4). It turns out (see for example [Tar72]) that any graph can be split, in a unique way, into components, each of which is biconnected, and not contained within a larger biconnected subgraph. Furthermore, any pair of components have no edge in common, and at most one node in common. The components are called the* **biconnected components**, *and the common nodes are called* **articulation points**. *The union of all the components is the whole graph. The set of all articulation points of G is written $AP(G)$.*

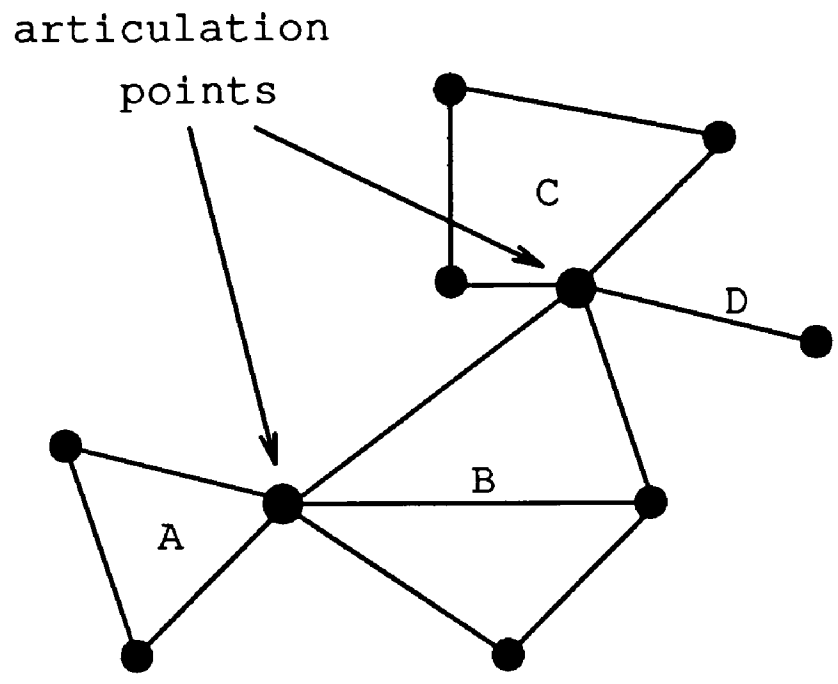


Figure 5.4: A graph with four biconnected components, A, B, C and D

By definition, it follows immediately that any biconnected graph is also connected. An articulation point is a node whose removal would cause the connected component in which it lies to become disconnected.

This method of decomposition has two stages. First the biconnected components are computed. This can be done at the same time as finding the connected components in time $O(n + e)$ for a graph with n nodes and e edges, using a variation on depth first search[Tar72]. The connected components are embedded independently as in the preceding section, but now biconnectivity information is available for each one. The biconnected components of a connected graph can be ordered such that each component intersects the union of all the preceding components at precisely one node (an articulation point) (theorem 5.1 below). In the second stage the components are embedded independently and then joined together in this order by translation. Since two embeddings can always be made to agree at any single node (by adding a constant vector to each point) this is always possible. Using the depth first ordering already calculated in stage one, the sorting and recombination can be achieved in time $O(n)$, since there at most n articulation points and n bicomponents.

Lemma 5.1 *If p, q and r are distinct nodes in a biconnected graph, then there exist non-intersecting paths from p to q and from p to r.*

Proof *(See figure 5.5) Since the graph is biconnected there are two non-intersecting paths L_1, L_2 from p to q. Similarly, there are two non-intersecting paths L_3, L_4 from p to r . Therefore, it is impossible that both L_3 and L_4 pass through q, so*

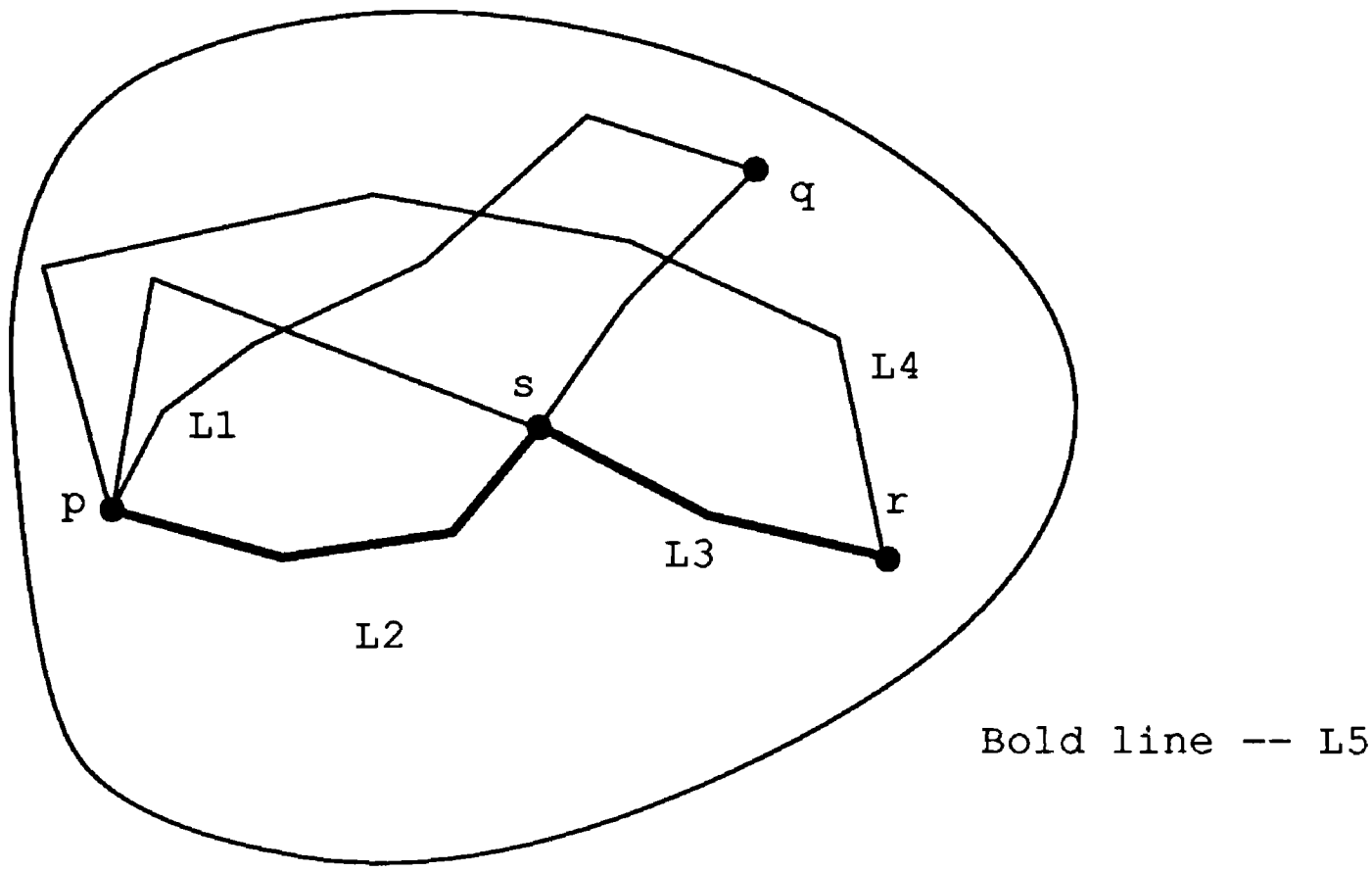


Figure 5.5: Proof of Lemma 5.1

without loss of generality, assume L_3 does not pass through q. Follow L_3 from r towards p. Let s be the first node at which this route meets one of L_1 or L_2 (note that s may be equal to p or r). Without loss of generality, suppose s is in L_2. Construct a new path L_5 which follows L_3 from r to s and then L_2 from s to p. It follows that L_1 and L_5 are the required paths.

Lemma 5.2 *If two graphs A and B are biconnected, and have at least two nodes in common, then their union is biconnected.*

Proof *(See figure 5.6) Suppose nodes q and r are common to A and B. Suppose further that a and b are nodes in $A \cup B$. We must show that there are two non-intersecting paths from a to b. If a and b are both in A (or both in B), then the paths exist by biconnectivity of A (or B). Therefore, assume that a is in A and that b is in B. By lemma 5.1, take non-intersecting paths L_1 and L_2 in A from a to q and a to r respectively. Let s be the first node in L_1 (starting from a) which lies in B, and let t be the first node in L_2 which lies in B. By lemma 5.1, take non-intersecting paths L_3 and L_4 in B from s to b and t to b respectively. Then the path from a along L_1 to s and along L_3 to b and the path from a along L_2 to t and along L_4 to b are non-intersecting.*

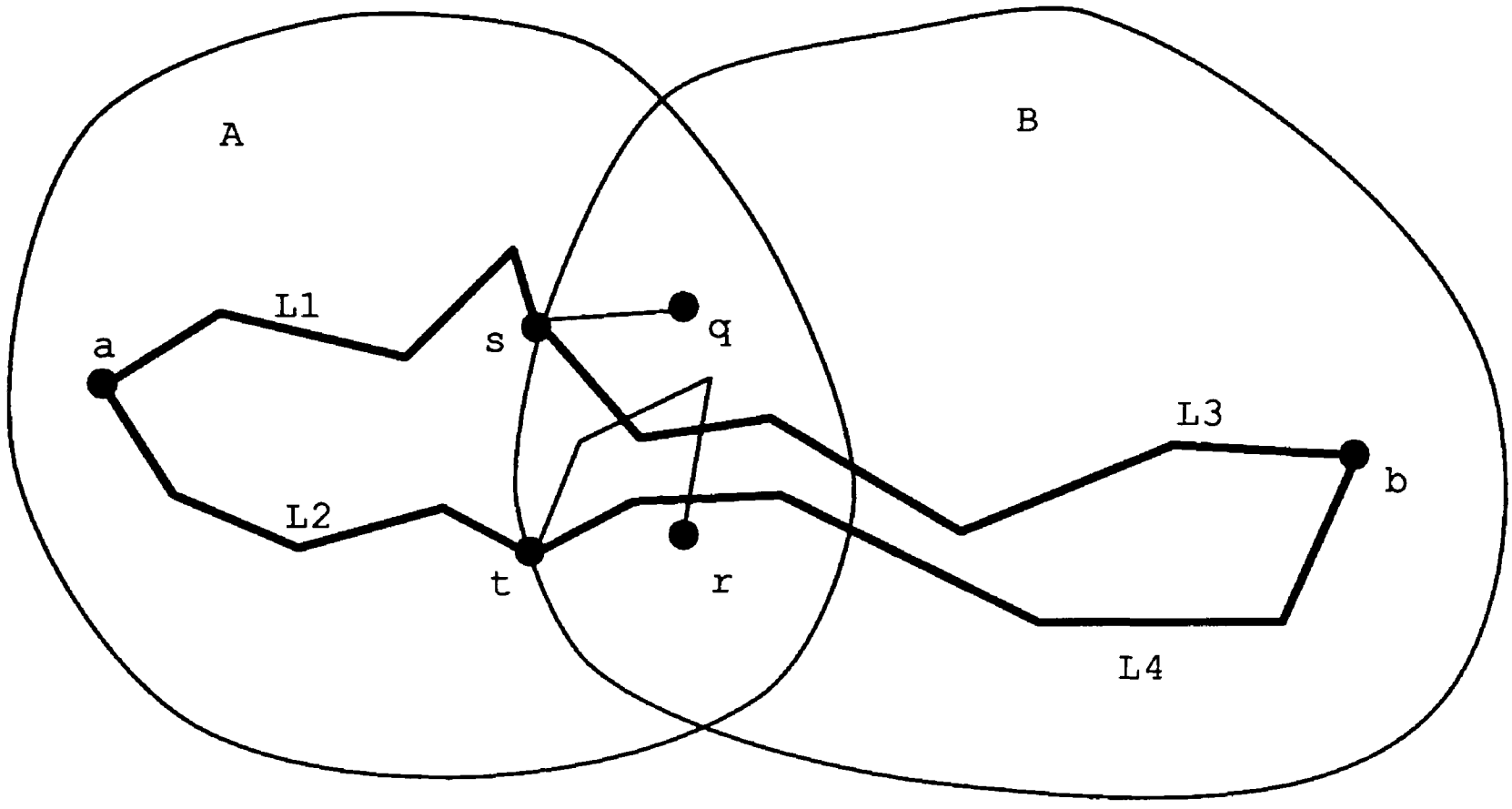


Figure 5.6: Proof of Lemma 5.2

Theorem 5.1 *The biconnected components of an embeddable graph (corresponding to an ADGP) may be embedded independently, and translated into place to provide an embedding for the whole structure.*

Proof *Suppose $B_1, \ldots, B_k$ are the bicomponents of the original graph. The connected components can be embedded independently, so assume the graph is connected. Since each B_i is connected, assume that the ordering of the bicomponents is such that the union of each initial segment, $B_1 \cup \ldots \cup B_i$ is connected, for $1 \leq i \leq k$. Let $C_i = B_1 \cup \ldots \cup B_i$.*

I claim that each B_{i+1} has precisely one node (and no edges) in common with C_i, and that that node is an articulation point. If B_{i+1} had no node in common with C_i, then their union would not be connected, contrary to supposition, since each bicomponent contains only internal edges. Now suppose for contradiction that there are two common nodes, a and b. Since C_i is connected, there is a (simple) path L from a to b in C_i. Similarly, there is a (simple) path L' from a to b in B_{i+1}. So $L \cup L'$ is a cycle, and therefore is biconnected. Now $L \cup L'$ and B_{i+1} have a and b in common, therefore by lemma 5.2 their union is biconnected. This contradicts the maximality of biconnected components, and so there can be only one common point. It follows immediately from the definitions that the common point is an articulation point. Since biconnected components are edge disjoint, there can be no common edges, and the claim is proved.

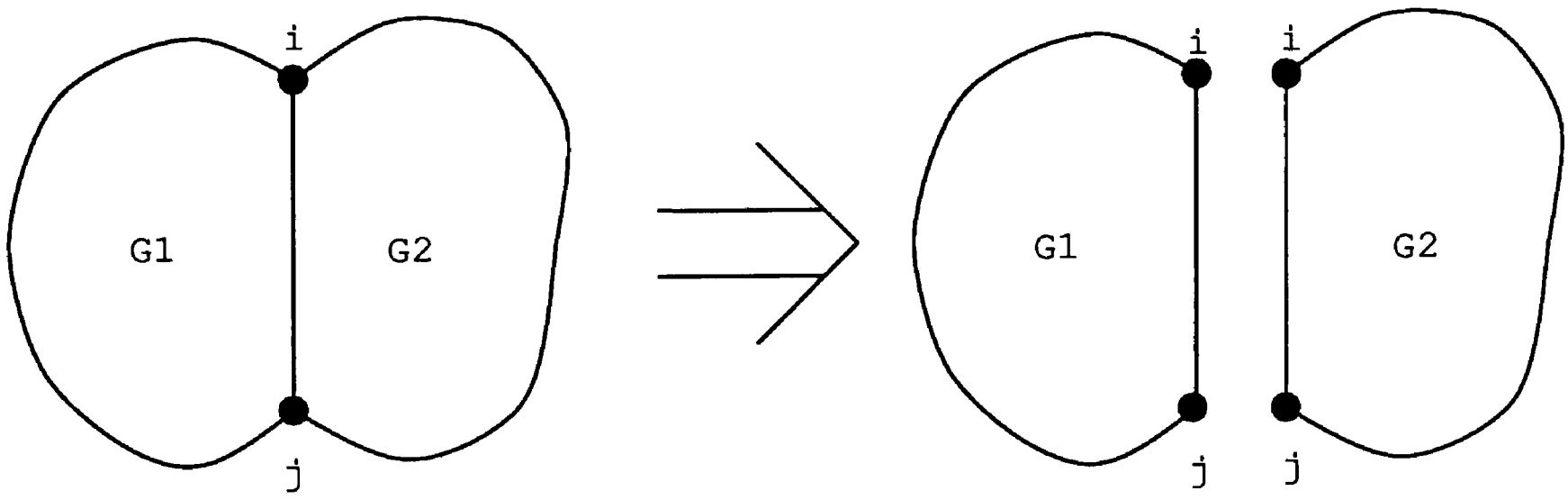


Figure 5.7: A case for the application of the articulation edge method.

The lemma now follows simply. Embed B_1 by any means available. Now suppose that C_i has been embedded ($i < n$), with an embedding function $\mathcal{E}$, and that B_{i+1} has been embedded with embedding function $\mathcal{E}'$. If u is the unique node common to these two, then translate each point in $\mathcal{E}'$ by a vector equal to $\mathcal{E}(u) - \mathcal{E}'(u)$ and adjoin the new mapping to $\mathcal{E}$ to find an embedding for C_{i+1}. By induction, the whole graph can be embedded in this way.

5.5 Articulation Edge Method

In this section I introduce a variation on the articulation point method (see above), which I call the articulation edge method. As before, the object is to split the graph into smaller parts, solve and recombine. Whereas the articulation point method produces subgraphs which share only single points, this method produces subgraphs which share a single (exact) edge.

Suppose an embedding problem is given, with corresponding constraint graph $G = (N, E)$ such that G is biconnected. Search for an edge $\{i, j\} \in E_X$ such that the removal of the nodes i and j and associated edges disconnects the graph (see figure 5.7). If such a pair can be found then an application of the articulation edge method is possible.

To apply this method, each of the connected components is recombined with nodes i and j and embedded separately. These parts can then be translated and rotated to form an embedding for the whole graph, by finding any rigid motion which causes the i and j vertices of the components to coincide. This can be done, because the

edge $\{i,j\}$ is exact, and so their separation in any embedding will be constant. The intuition is that the graph is free to flex about the edge $\{i,j\}$ in the same way as biconnected components are free to flex about an articulation point.

Definition 5.5 *Suppose $G = (N,E)$ is a biconnected graph. The* **articulation edges**, *$AE(G)$ are those elements $e \in E_X$ such that $G - e$ is not connected. If $AE(G) = \emptyset$ then I say G is* **AE-connected**.

Lemmas 5.3 and 5.4 provide the motivation for the effective method for finding articulation edges, using the biconnectivity algorithms mentioned in the previous section[Tar72].

Lemma 5.3 *If G is biconnected and $\{i,j\} \in E_X$ then $\{i,j\}$ is in $AE(G)$ if and only if j is in $AP(G - \{i\})$.*

Proof *G is biconnected, so for any i, $G - \{i\}$ is connected. Therefore $\{i,j\} \in AE(G)$ if and only if $G - \{i,j\}$ is not connected if and only if $(G - \{i\}) - \{j\}$ is not connected if and only if $j \in AP(G - \{i\})$.*

Lemma 5.4 *If $G = (N,E)$ is biconnected, and $\{i,j\} \in AE(G)$, and $G' = (N',E')$ is a connected component of $G - \{i,j\}$, then $G'' = G \cap (N' \cup \{i,j\})$ is biconnected.*

Proof *Suppose nodes a and b are in G''. G is biconnected, so there exist paths L_1 and L_2 in G from a to b which intersect only at a and b. I claim that if L_1 does not lie wholly inside G'' then it contains both nodes i and j.*

To prove the claim, suppose c is a node in L_1 but not in G'' (see figure 5.8). Suppose $i \notin L_1$. Since a and c are in different components of $G - \{i,j\}$, and since part of L_1 is a path from a to c in $G - \{i\}$, then this part of the path must pass through j. But since b and c are in different components of $G - \{i,j\}$ as well, the part of L_1 from c to b must also pass through j. This is a contradiction, since a path cannot contain the same node twice. Therefore $i \in L_1$ and similarly $j \in L_1$.

Therefore, if L_1 does not lie wholly inside G'' then the loop of L_1 lying outside can be replaced by the edge $\{i,j\}$. Similarly for L_2. Therefore, unless both L_1 and L_2 contain i and j, we have 2 non-intersecting paths from a to b. If they do both contain i and j, then by disjointness, these must be the endpoints. Without loss of generality, $a = i$ and $b = j$. It remains to find a path from i to j within G'' that doesn't require the edge $\{i,j\}$. Choose any node d in G'. Using lemma 5.1 there are non-intersecting paths from d to i and d to j. These paths cannot pass outside G'', because they would have to pass through i or j (arguing as above), and these are the endpoints. The concatenation of these two paths is the required path.

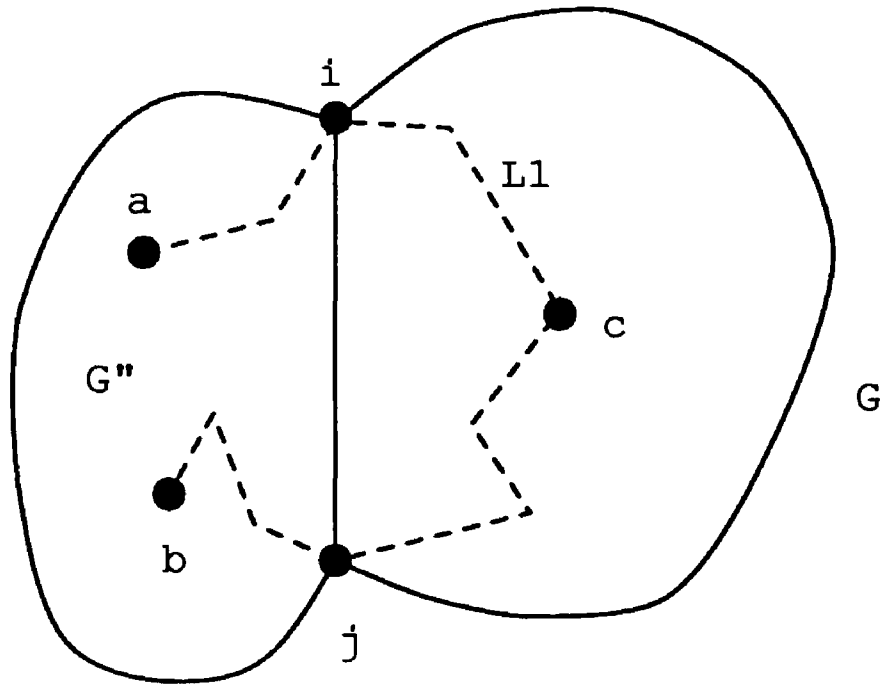


Figure 5.8: Proof of lemma 5.4.

My first reaction to the problem of splitting a biconnected graph using articulation edges resulted in an $O(e(n+e))$ method. This algorithm is as follows. Choose any edge $\{i,j\}$ in E_X, remove it, and compute the connected components. If there is more than one, replace the edge $\{i,j\}$ in each component and recursively split each component. If not, try a new edge until all exact edges have been tried. There are $O(e)$ exact edges to try, and each one requires time $O(n+e)$ for the connected components test. Lemma 5.3 suggests a way of finding all the useful j corresponding to a particular i in one go.

I propose the following algorithm $SPLIT(G,i)$ for splitting a biconnected graph G with respect to a node i. First, remove i from the graph. Then use a variant of the biconnected components algorithm to find those articulation points j of $G-\{i\}$ such that $\{i,j\} \in E_X$. These points determine a decomposition $G_1, \ldots G_p$ of $G-\{i\}$. Finally, the node i is added, along with its corresponding edges, back into each of the G_k for $1 \leq k \leq p$, and these subgraphs are returned as the result of the algorithm. Since this variant of the biconnected components algorithm is actually doing less work than the standard algorithm (because there will be fewer components, and each will be a union of bicomponents), I claim that it can be accomplished in time $O(n+e)$ as before. Furthermore, no extra time is required to provide the ordering required in the theorem, since the depth first search order produced by the biconnected components algorithm is sufficient. Theorem 5.2 shows how *SPLIT* can be used as a method of graph decomposition.

Theorem 5.2 *Suppose G is a biconnected graph, and i is a node in G. Then the subgraphs $G_1 \ldots G_p$ which are returned by the algorithm $SPLIT(G,i)$ may be*

embedded independently.

Proof *This theorem is very similar to theorem 5.1 above, so only an outline proof is given. Any pair of the subgraphs $G_1 \ldots G_p$ have node i in common, and at most one other node, which is an articulation point of $G - \{i\}$. Ignoring node i, the G_k can be reordered so that for any k, $\bigcup\{G_1 - \{i\}, \ldots, G_{k-1} - \{i\}\}$ is connected, and such that $G_k - \{i\}$ intersects this union at precisely one node, say a_k. With this new ordering, $\bigcup\{G_1 \ldots G_{k-1}\}$ is connected for any k, and G_k intersects this union at precisely i and a_k. Recall that $\{i, a_k\} \in E_X$. The embedding is now easy. Each component G_k is embedded by any means available, with embedding function $\mathcal{E}_k$. Suppose there is an embedding $\mathcal{E}$ for $\bigcup\{G_1 \ldots G_{k-1}\}$. Rigidly transform $\mathcal{E}_k$ so that nodes i and a_k coincide. This can always be done, because $\{i, a_k\} \in E_X$ and so the length of this edge is the same in $\mathcal{E}$ and $\mathcal{E}_k$ (this is the crux of the method). The combination of these two functions is an embedding function for $\bigcup\{G_1 \ldots G_k\}$.*

Finally I present the complete articulation edge algorithm, $AE(G)$, which applies to a biconnected graph G. Mark each node as unused. For each unused node i in G, Mark i as used and compute $SPLIT(i, G)$ in time $O(n + e)$, until split returns more than 1 component. Solve these components recursively, combining the returned embedding functions as in theorem 5.2. If all nodes are marked used, embed this graph by any means, and return the embedding function. This description is made more precise in figure 5.9.

This takes time $O(n(n + e))$, and may be faster by a factor of n than the naive algorithm when there are many more edges than nodes.

5.6 Generalisations

The biconnectivity and articulation edge methods work because they use subgraphs with only one possible embedding, up to rigid transformations of the embedding space. I shall call such a subgraph **fixed**. It is safe to embed two graphs with a common fixed subgraph independently, because it is certain that the two copies of the subgraph can be made to coincide afterwards. Generalisations of these methods may be found by looking for other types of fixed subgraphs. In general, the fixed subgraphs are **cliques** of exact edges, where A clique is a complete subgraph, that is, one in which there is an edge between each pair of nodes. Using lemma 2.1 it may be seen that a clique of exact edges is an instance of the complete exact DGP, and therefore is fixed. The following algorithm arises.

Suppose A is a subgraph of a connected graph G, and A is a clique of exact edges.

Algorithm AE

```
global variable UNUSED;

function AE(G)  (* returns an embedding for G *)
    (N, E) := G;
    UNUSED := N;
    return AE_1(G);
end

function AE_1(G)  (* returns an embedding for G *)
    (N, E) := G;
    foreach i ∈ UNUSED ∩ N do
        S := split(i, G);
        UNUSED := UNUSED − {i};
        if #S ≥ 2 then
            foreach G_k ∈ S do
                E_k := AE_1(G_k);
                if E_k = FAIL then return FAIL fi
                od
            E := E_1;
            for k := 2 upto #S do
                {i, j} := "edge connecting G_1 ∪ ... ∪ G_{k−1} to G_k";
                "translate E_k so that E_k(i) = E(i)"
                "reflect E_k so that E_k(i) = E(i) and E_k(j) = E(j)"
                E := E ⊕ E_k;
                od
            return E
            fi
        od
    (* no AE decomposition possible, so use some other method *)
    return embed(G)
end
```

Figure 5.9: The function AE for articulation edge decomposition. The notation $\#S$ represents the number of elements in a set S.

Attempt to embed A using the algorithm of lemma 2.1. If A also contains angle constraints check that the embedding satisfies them also. If not then no embedding is possible. Otherwise, remove A from the graph, and collect the connected components of the remainder. Independently recombine each connected component with A, and embed by any other method. Since A is fixed, the components may be recombined to form an embedding for G.

Unfortunately, finding large cliques in a graph is difficult. The problem of deciding whether there is a clique of size k or more in a graph is NP complete[GJ79]. However, some profit may me made by considering smaller cliques. One obvious approach is to consider all the exact triangles. Another is to choose an exact edge, and add more nodes as long as a clique is formed each time. This solution finds a maximal clique, that is one which is not contained in a larger one. Using this algorithm, a maximal clique can be found in time $O(n^3)$.

A further use for fixed subgraphs occurs during the process of general constraint solving. For example, suppose the general constraint solving method is simulated annealing[KGV83]. Then in $\mathbb{R}^3$ with n nodes in the constraint graph, there are $3n$ variables. Now suppose that a fixed subgraph with i nodes has been isolated, with $i \geq 3$. Then it is possible to reduce the number of variables in the following way. First find an embedding $\mathcal{E}$ for the fixed subgraph. From the definition of fixed, any possible embedding of this subgraph is obtainable from $\mathcal{E}$ by translation rotation and reflection. These transformations may be represented by six real variables (three for the translation and three for the rotation) and a single toggle indicating 'normal' or 'mirror image.' The transformed version of $\mathcal{E}$ can be computed in time $O(i)$, therefore the minimisation may be carried out using just $3(n-i)+7$ variables rather than the full $3n$. Furthermore, if two fixed subgraphs share a node then by choosing that node as the origin of both, only one of the translation vectors is necessary, plus four for each for rotation and reflection. This method can be extended to take advantage of larger overlaps between fixed subgraphs.

5.7 Graph Transformations

The most important graph transformations, namely constraint propagation and relocalisation, have already been discussed in this chapter. I define a graph transformation to be an operation which maps constraint graphs to new constraint graphs, such that there is a simple correspondence between embeddings of the old and new graphs. In the case of constraint propagation and relocalisation the possible embeddings of old and new are identical.

5.7.1 Removing Chains

Chain removal is an operation of the second type, that is a graph transformation rather than a graph decomposition. Suppose the graph contains the following structure: Two edges $\{i,j\}$ and $\{j,k\}$ such there are no angle constraints on either edge and there are no other edges containing j. Then the two edges and node j may be removed, and replaced by the single edge $\{i,k\}$, with new upper and lower bounds U' and L' given by

$$\begin{aligned} U'_{ik} &= \min(U_{ik}, U_{ij} + U_{jk}) \\ L'_{ik} &= \max(L_{ik}, L_{jk} - U_{ij}, L_{ij} - U_{jk}) \end{aligned}$$

The reasoning is that any embedding of the new graph can be extended (in constant time) to an embedding including j of the old graph. Chain removal can sometimes be used to remove a whole sequence of edges, by successively folding them into a single edge.

The basic algorithm can be extended to the case when there is a value for θ_{ijk}. Then the cosine rule and a messy case analysis (see appendix A) can be used to find the new edge bounds. Also the angle data can be removed. Again, the logic of the operation is that whatever value the length of the edge $\{i,k\}$ takes in a particular embedding (within its constraints), the triangle ijk can be constructed so that it satisfies the old constraints.

5.8 Plan of Algorithm

It remains to collect together the different threads of the method in one place, to act as an implementation guide. I assume that the algorithm is provided with a valid version of ADGP (as defined in section 3.1) expressed as a graph. The object is to produce any correct embedding (if one exists). The choice of exact (algebraic) or approximate arithmetic is not important for most of the algorithm, except in the final embedding stages.

The first step is to check whether the problem lies in one of the easier subcases mentioned in chapters 2 and 3. In fact, any time the graph is changed or decomposed this check is performed on the new graph or the subgraphs. The special cases (and what to do about them) are as follows:

1. Graph with no constraints – In this case, any embedding function is acceptable.

2. Graphs with no angle constraints.

   (a) All length lower bounds 0 – again, any embedding function is acceptable.

   (b) Complete DGP – solve using lemma 2.1 in time $\Theta(n^2)$.

3. Graphs with three or fewer nodes – solve directly using elementary trigonometry to find any correct embedding.

4. Complete Exact Angle Problems as explained in section 3.5.

   (a) If there are no length constraints, choose an exact length arbitrarily and convert into a complete DGP using constraint propagation and lemma 3.1. Solve as special case 5.

   (b) If there are length constraints, perform constraint propagation first, then choose a single length which satisfies its bounds. Perform constraint propagation again (see lemma 3.1), and proceed as in case 5.

5. Complete DGP with some angles – solve the complete DGP part first (using case 2b), then check that the resulting embedding satisfies the angle constraints.

The main algorithm is broken up into the following stages. I do not explicitly mention the checks for special cases as described above, but these should be applied before each step in the list below. Each item in the list constitutes a method which either succeeds completely (producing an embedding) or succeeds partially (producing a series of calls to the next item on the list) or fails (passing the problem on unchanged to the next item in the list).

1. Perform Constraint Propagation (section 3.4) to distribute constraint information around the graph.

2. Perform Relocalisation (section 5.2) – to remove unnecessary inexact constraints introduced by step 1.

3. Apply graph decomposition strategies as described in this chapter. Each method is capable (in favourable cases) of producing a collection of independent subproblems. The subproblems may be solved in parallel and recombined to form an embedding for the whole. as follows:

   (a) Apply the method of connected components (section 5.3). Perform the following step in this list on each component. Recombine the embeddings by simply taking their union.

(b) Apply the method of biconnected components (section 5.4). Perform the following step in this list on each biconnected component. Recombine the embeddings using theorem 5.1.

(c) Apply the method of articulation edges (section 5.5), using the algorithm *AE* (figure 5.9). Calls to the function *embed* are to be implemented by the next item.

(d) Apply the methods of section 5.6 (optional step), again passing unsolved (sub)problems to the next step.

4. Perform Chain Removal (section 5.7.1)

If any of the decompostion steps have made progress, it might be possible to make even more progress by repeating the algorithm up to this point. This option should be under user control. When any subproblem cannot be further simplified using the above methods, solve it using one of the general purpose algorithms as follows.

For algebraic problems where exact answers are required, consider CAD and Gröbner basis methods (chapters 6 and 7), with the (optional) heuristic described in section 5.6 for reducing the number of variables.

For numerical problems the best approach is to use a numerical minimisation technique such as steepest descent or conjugate gradient, again using the heuristic described in section 5.6 for reducing the number of variables. If there are many more length constraints than angle constraints, the method of section 2.5 can be used to generate a starting configuration by ignoring the angle constraints. If not, it may be better to choose the starting state at random and repeat the minimisation a few times to ensure that a global (rather than local) minimum is reached.

5.9 Worked Example

Figure 5.10 shows an example of an ADGP in two dimensions. The robot in (i) is required to grasp the apple, without moving from its current position. It has an elbow joint which can assume any angle between $\frac{\pi}{2}$ and π, and an extensible forearm with a reach of between 5 and 7 units. All the other lengths are shown, and all other joints are assumed to be completely unconstrained. The corresponding ADGP is shown in (ii), where a solid line represents an exact length, and a dashed line represents a partially constrained length. The nodes are labelled $a \ldots e$ so as not to confuse them with lengths. The correspondence is such that the robot can fulfill its task if and only if there is an embedding of the ADGP, and further, such an

embedding allows us to compute the appropriate robot parameters. I have drawn the graph in its final configuration, with the gripper coinciding with the apple (at a), so providing the lengths are correct this is indeed an embedding in two dimensions. Of course in reality, an embedding like this is precisely what is required from the algorithm. The working is done by hand, as I have not implemented the algorithm.

The graph does not fall into any of the special cases mentioned in section 5.8, so the first step is to perform constraint propagation and relocalisation (section 3.4), the result of which is shown in (iii). After constraint propagation all possible edges have upper and lower bound information, but most of these are inexact bounds, and so are removed by relocalisation. In this case the triangle abc is completed, to give the exact length 10 for ac. Note that the redundant right angle at b has been removed. The triangle rule applied to the triangle ace produces constraints $U_{ec} = 12$ and $L_{ec} = 8$. Considering triangle cde using the cosine rule allows the lower bound on ed to be raised from 5 to $\sqrt{39}$. Thus after renormalisation all that remains of the added information is the exact length ac and the improved bound of ed.

In (iv) articulation edge decomposition has resulted in two simpler subgraphs (the method of biconnected components does not apply here) with common articulation edge ae. The triangle on nodes a, b and c may be embedded immediately, as a special case. In (v) chain removal has turned the inexact angle θ_{abc} and inexact edge de into a new inexact edge ce, with bounds 8 and 12. The resulting triangle ace may be embedded as a special case. An embedding for the whole may be derived from this information.

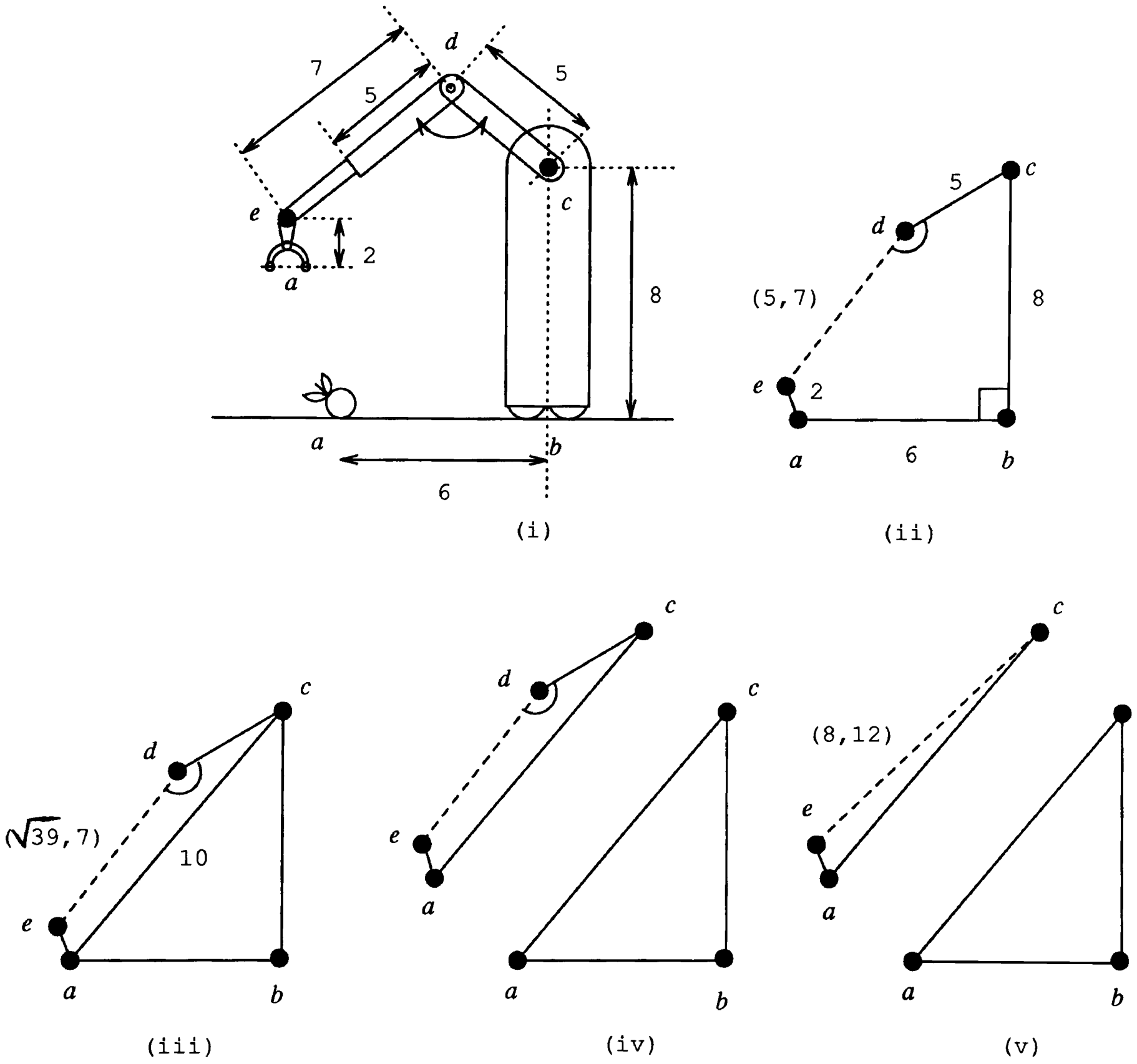


Figure 5.10: An example of ADGP in two dimensions

Chapter 6

Cylindrical Algebraic Decomposition

In this chapter I present a method for an algebraic solution to the ADGP with or without optimisation. The method, although complete, is unfortunately doubly exponential in time and space, but nevertheless is useful to show that an exact solution is indeed computable (although not necessarily feasible). This method actually applies to a wider class of optimisation problems, namely those where the constraint and target functions are polynomials over an algebraic subfield of $\mathbb{R}$. The definition in the next section captures the class of problems solvable by this method.

6.1 POP and PSP

Definition 6.1 *Given a collection $g, f_1, \ldots, f_n$ of polynomials with algebraic coefficients and indeterminates $X_1, \ldots, X_m$, and a collection $\delta_1, \ldots, \delta_n$ of non-empty subsets of $\{-1, 0, +1\}$, the* **Polynomial Optimisation Problem, POP**, *is to find a point $\mathbf{x} = \langle x_1, \ldots, x_m \rangle$ in $\mathbb{R}^m$, when one exists, such that*

$$\forall\, i \in 1 \ldots n \mid sign(f_i(\mathbf{x})) \in \delta_i$$

and so that $g(\mathbf{x})$ is maximal amongst all such $\mathbf{x}$. (The function $sign(r)$ returns -1 if $r < 0$, 0 if $r = 0$ and $+1$ if $r > 0$)

When the target function, g, is constant, the POP reduces to the problem of solving a system of polynomial equalities and inequalities, and returning any element of the solution set. This is the **Polynomial Solution Problem, PSP**.

In fact, the method of Cylindrical Algebraic Decomposition (CAD) [ACM84a] discussed in this chapter allows access to the distinct connected regions of the solution space (regarded as a subset of $\mathbf{R}^m$), and can be used to find at least one sample solution in each region.

I show how the ADGP (without optimisation) may be cast in the form of a PSP, and then how to solve this using the standard method of CAD. Finally I present the extension to the CAD method which allows solution of the full POP and thus the full ADGP (with optimisation) when the target function is polynomial.

6.2 Turning the ADGP into a PSP

Many geometric problems fit into the PSP category, in particular ADGP does. The two types of constraint, angle and length, may be expressed as polynomials in terms of the co-ordinates of the positions of the nodes in a successful embedding.

Consider the d dimensional ADGP with constraint functions L_{ij}, U_{ij} and θ_{ijk}. Suppose $\mathcal{E}$ is a candidate for a suitable embedding function, mapping each of the n nodes to $\mathbf{R}^d$. For each node i $(1 \leq i \leq n)$ I assign d variables $x_{i1}, x_{i2}, \ldots x_{i_d}$ corresponding to the co-ordinates of the position $\mathcal{E}(i)$ of the vertex in the embedding space. The vector $\overrightarrow{\mathcal{E}(i)\mathcal{E}(j)}$ has lth co-ordinate equal to $x_{jl} - x_{il}$, so the scalar product $\overrightarrow{\mathcal{E}(i)\mathcal{E}(j)} \cdot \overrightarrow{\mathcal{E}(i')\mathcal{E}(j')}$ is equal to $\Sigma_{l=1}^{d}(x_{jl} - x_{il})(x_{j'l} - x_{i'l})$, which is a polynomial in the x variables. Since $\Delta(\mathcal{E}(i), \mathcal{E}(j))^2 = \overrightarrow{\mathcal{E}(i)\mathcal{E}(j)} \cdot \overrightarrow{\mathcal{E}(i)\mathcal{E}(j)}$, the constraint $L_{ij} \leq \Delta(\mathcal{E}(i), \mathcal{E}(j)) \leq U_{ij}$ may be expressed as two polynomial equations

$$sign((\overrightarrow{\mathcal{E}(i)\mathcal{E}(j)} \cdot \overrightarrow{\mathcal{E}(i)\mathcal{E}(j)}) - L_{ij}^2) \in \{0, +1\}$$

$$sign((\overrightarrow{\mathcal{E}(i)\mathcal{E}(j)} \cdot \overrightarrow{\mathcal{E}(i)\mathcal{E}(j)}) - U_{ij}^2) \in \{-1, 0\}$$

When $L_{ij} = U_{ij}$, the obvious simplification to a single equation is used.

For an angle constraint $\angle(\mathcal{E}(i)\mathcal{E}(j)\mathcal{E}(k)) \in \theta_{ijk}$, when $\theta_{ijk} = [\alpha, \beta]$ is defined, I make use of the identity

$$\overrightarrow{\mathcal{E}(j)\mathcal{E}(i)} \cdot \overrightarrow{\mathcal{E}(j)\mathcal{E}(k)} = \Delta(\mathcal{E}(j), \mathcal{E}(i)).\Delta(\mathcal{E}(j), \mathcal{E}(k)).\cos(\angle(\mathcal{E}(i)\mathcal{E}(j)\mathcal{E}(k)))$$

to arrive at the polynomial equations

$$sign((\overrightarrow{\mathcal{E}(j)\mathcal{E}(i)} \cdot \overrightarrow{\mathcal{E}(j)\mathcal{E}(k)})^2 - \Delta(\mathcal{E}(i), \mathcal{E}(j))^2.\Delta(\mathcal{E}(j), \mathcal{E}(k))^2.\cos^2(\alpha)) \in \{-1, 0\}$$

$$sign((\overrightarrow{\mathcal{E}(j)\mathcal{E}(i)} \cdot \overrightarrow{\mathcal{E}(j)\mathcal{E}(k)})^2 - \Delta(\mathcal{E}(i),\mathcal{E}(j))^2.\Delta(\mathcal{E}(j),\mathcal{E}(k))^2.\cos^2(\beta)) \in \{0,+1\}$$

Again there is an obvious simplification when $\alpha = \beta$. Note that the angle constraints are given in terms of their cosines, and therefore these equations really are algebraic. Also, because the cosine function is a decreasing function in the range $[0,\pi]$ the sense of these inequalities appears at first sight to be the wrong way round.

6.3 Definition of a CAD

Let $\mathbf{A}_R$ be the field of real algebraic numbers over the rationals. By this, I mean the set of real numbers which are roots of univariate polynomials with rational coefficients. Such a number may be represented by an irreducible univariate polynomial of which it is a root, and an isolating interval chosen so that the particular root is the only one in the interval. These objects can be manipulated with reasonable efficiency, allowing all the usual rational numerical operations, including exponentiation and testing whether an algebraic number is positive, negative or zero[CL82][Loo82], to be done in low order polynomial time.

Suppose R is a simply connected region in $\mathbf{R}^d$ (i.e. one without any 'holes'). We define the **cylinder over** R, $Z(R)$ to be the subset $R \times \mathbb{R}$ of $\mathbf{R}^{d+1}$. Now suppose $f_1,\ldots,f_n$ is a sequence of continuous functions from R to $\mathbb{R}$, such that for all $1 \leq i < j \leq n$ and $x \in R$, it is the case that $f_i(x) < f_j(x)$. Then the graphs of the roots of f_i partition $Z(R)$ into $2n+1$ simply connected regions, i.e. the graphs of the f_i, the spaces between f_i and f_{i+1} for each i, and the space below f_1 and the space above f^n. A partition of $Z(R)$ which can be described by a sequence of functions in this way is called a **stack over** R.

Now suppose D is a partition (i.e. a decomposition) of $\mathbf{R}^d$. Then D is a **Cylindrical Decomposition**, CD, of $\mathbf{R}^d$ if and only if either

1. $d = 1$ and D is a stack over $\mathbf{R}^1$, or
2. $d > 1$ and there is a CD, D', of $\mathbf{R}^{d-1}$ such that for each R' in D' there is a subset R of D which is a stack over R'.

In other words, the regions of a CD of $\mathbf{R}^{d+1}$ form vertical towers over $\mathbf{R}^d$ such that no two towers are interlinked (they have smooth sides), and so that the 'ground-plan' of the towers form a CD of $\mathbf{R}^d$.

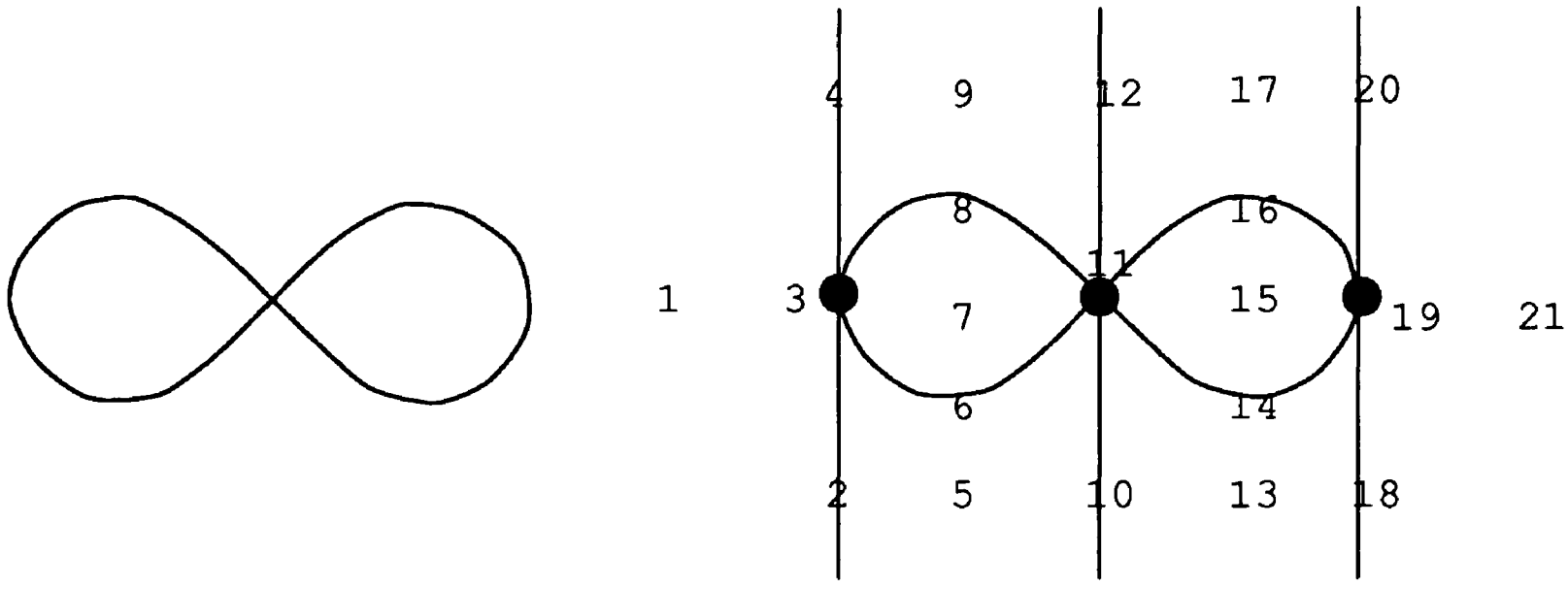


Figure 6.1: The graph of the roots of a polynomial (left), and a possible CAD (right)

A **Cylindrical Algebraic Decomposition**, CAD, of $\mathbf{R}^d$, is a CD of $\mathbf{R}^d$ in which the functions defining all the stacks (and their sub-stacks) are algebraic. Collins [ACM84a] refers to the elements of the decomposition as **cells**.

For any function f we define $V(f)$ to be the set of roots of f in its domain. Further, if F is a set of functions, we define $V(F)$ to be the union of the sets $V(f)$ for each f in F.

Now suppose $F = \{f_1, \ldots, f_n\}$ is a set of multivariate polynomials in variables $X_1, \ldots, X_m$ with coefficients in $\mathbf{A}_R$. A CAD for F is a CAD of $\mathbf{R}^m$ such that $sign(f_i(\mathbf{x}))$ is constant throughout each cell for each i. In other words, within any particular cell, none of the polynomials will pass through zero (although they may be zero throughout). Such a decomposition is called **compatible with** F. For example, consider the single polynomial f, whose set of roots, $V(f)$, is shown in figure 6.1. In this case I have drawn the smallest possible CAD compatible with $\{f\}$, although in general the Collins algorithm may subdivide the cells further. The CAD consists of 21 cells, which are the white areas (1,5,7,9,13,15,17,21), arcs of $V(f)$ (6,8,14,16), straight construction line segments (2,4,10,12,18,20) and points of intersection (3,11,19).

6.4 The CAD Algorithm

In this section I explain Collins' algorithm for Cylindrical Algebraic Decomposition (CAD) [ACM84a] [ACM84b] for solving PSP. The CAD algorithm finds one possible

CAD for an instance of PSP. It proceeds by induction on the dimension of the problem, that is, the number of independent variables. To find a possible CAD for some set F of multivariate polynomials (denoted $CAD(F)$), we eliminate a variable to arrive at F', then solve $CAD(F')$ recursively, and finally build $CAD(F)$ from this. Thus there are three stages in the algorithm: Projection (to lower dimension), Base case for the induction, and finally Construction of the result. I start with the base case, since this is the simplest to understand.

6.4.1 Base Case

Suppose F is a set $\{f_1, \ldots, f_m\}$ of polynomials in one variable X. A CAD for F will be a partition, D, of the real line $\mathbb{R}$ into points and intervals. The points of D will be the isolated values at which at least one of the f_i vanishes. Therefore to calculate D we find the set of roots of all the f_i, call them $s_1 < s_2 < \ldots < s_r$. This set is indeed finite, since by the fundamental theorem of algebra, no polynomial of degree $n \geq 0$ can have more than n roots. The only awkward case is the zero polynomial, but since *any* CAD is a CAD for the zero polynomial (or indeed any constant function), it may be discarded from F without affecting the result. Consider the partition D of $\mathbb{R}^1$ defined by

$$D = \{\{s_i\} \mid 1 \leq i \leq r\} \cup \{(s_i, s_{i+1}) \mid 1 \leq i < r\} \cup \{(-\infty, s_1), (s_r, \infty)\}$$

By the intermediate value theorem, no polynomial can change sign without passing a root, hence D is compatible with F. Furthermore D is a stack over $\mathbb{R}^0 = \{0\}$, defined by the constant functions $c_i : \mathbb{R}^0 \to \mathbb{R}$ where $c_i(0) = s_i$. These are algebraic (since the s_i, being roots of algebraic polynomials are algebraic), hence D is a CAD for F. The output from the base case is a list of sample points for the cells of D, in increasing order.

6.4.2 Projection

In the projection phase, we are given a set $F = \{f_1, \ldots, f_n\}$ of polynomials in variables $X_1, \ldots, X_m$, and want to reduce this to a set F' in variables $X_1, \ldots, X_{m-1}$ in such a way that $CAD(F)$ may be deduced sensibly from $CAD(F')$ (and F). $CAD(F')$ represents a partition of $\mathbb{R}^{m-1}$ into regions compatible with the elements of F', so it is sensible to aim to construct $CAD(F)$ out of stacks based on elements of $CAD(F')$. This is possible if and only if F is **delineable** on each element R of $CAD(F')$. Essentially this says that the various connected regions of $V(F)$ should form the outline for a stack on $Z(R)$, ie there should be no crossovers or

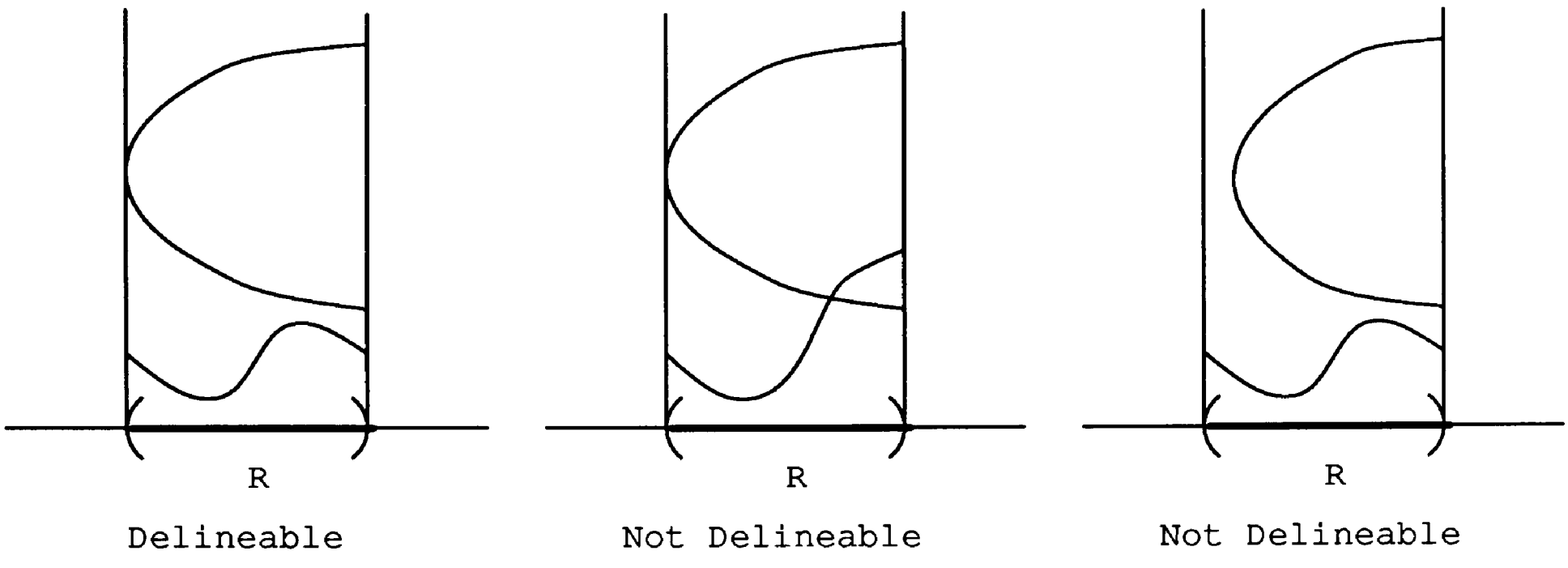


Figure 6.2: Illustration of Delineability

overlappings. More formally, F is **delineable** on R if and only if $Z(R) \cap V(F)$ consists of k connected regions (for some $k \geq 0$), and for each region C, the vertical projection $\pi \in C \rightarrow R$ is a bijection.

It can be seen from figure 6.2 that there are at least two ways in which F can fail to be delineable on R. The first type is when two leaves of $V(F)$ cross, the second when one leaf becomes vertical. To explain how these cases are dealt with, it is necessary to define some terms.

Suppose f is a multivariate polynomial. The **leading coefficient** of f with respect to a variable X is the coefficient of the highest power of X when f is expanded as a polynomial in X, and is written $lc(f, X)$. The highest power of X is the **degree** of f with respect to X, and is written $\delta(f, X)$. The **leading term** of f in X is defined to be $lc(f, X).X^{\delta(f,X)}$, and is written $lt(f, X)$. The **reductum** of f, written $red(f, X)$, is the polynomial left after subtracting $lt(f, X)$ from f. The **reductum set** of f, written $RED(f, X)$, is the set of iterated non-zero reducta of f.

$$RED(f, X) = \{red^i(f, X) \mid 0 \leq i \leq \delta(f, X) \wedge red^i(f, X) \neq 0\}$$

The reductum set of f gives the behaviour of f at those points for which the higher order terms of f vanish, and is required for technical reasons in what follows.

The Sturm sequence (with respect to X) of a pair of polynomials f and g, is the sequence obtained by applying Euclid's algorithm for highest common factor. The kth principal subresultant coefficient, $psc_k(f, g, X)$ is the leading coefficient of the degree k element (if one exists), in the Sturm sequence of f and g, up to multiplication by a constant. An efficient algorithm and formal definition of subresultants

is given in [BT71]. The **psc set** of a pair of polynomials, $PSC(f,g,X)$, is defined to be the empty set if either f or g is the zero polynomial, and otherwise is the collection of non-zero $psc_k(f,g,X)$ for $0 \leq k \leq min(\delta(f,X),\delta(g,X))$.

We require then that the set F' is such that F is delineable above each region of $CAD(F')$. We have remarked that the awkward parts of $V(F)$ are where leaves cross or have a vertical tangent. Therefore we should aim to make $V(F')$ lie below these areas, that is, find a set of polynomials in one fewer variables whose roots demarcate the singular behaviour of $V(F)$. The set F' used in [ACM84a] to achieve this is given by

$$F = proj(F, X_m)$$

where

$$\begin{aligned}
proj(F,X) &= proj_1(F,X) \cup proj_2(F,X) \\
proj_1(F,X) &= \bigcup_{f \in F} \bigcup_{f' \in RED(f,X)} (\{lc(f',X)\} \cup PSC(f', \frac{df'}{dX})) \\
proj_2(F,X) &= \bigcup_{f,g \in F, f \neq g} \bigcup_{f' \in RED(f,X), g \in RED(g,X)} PSC(f',g')
\end{aligned}$$

The essential idea is that $proj_1$ corrects for places where a polynomial has a repeated root, and $proj_2$ corrects for places where two polynomials share a root. The proof that F is delineable over each region of $CAD(proj(F))$ is well beyond the scope of this thesis, but may be found in [ACM82](theorems 3.6 and 3.7). The main ideas are in [Col75](theorems 4 and 5).

We can now see why the complexity of this algorithm is so high. Let n be the number of polynomials in F, let δ be the maximum degree (over all variables), and let m be the number of variables, $X_1, \ldots, X_m$. Let f, g be polynomials, then

$$\#RED(f,X) = O(\delta(f,X))$$

$$\#PSC(f,g,X) = O(min(\delta(f,X),\delta(g,X))$$

Hence

$$\#proj_1(F,X_m) = O(n.\delta.\delta)$$

And

$$\#proj_2(F,X_m) = O(n^2.\delta^2.\delta)$$

Therefore

$$\#proj(F, X) = O(n^2.\delta^3)$$

In fact this may be an overestimate, since the projection set can be simplified by removing polynomials which are constant, or which are constant multiples of others, or are products of others.

Since there are m variables to be eliminated during the projection phase, we arrive at a sequence of polynomial sets, F_i, each containing polynomials in one fewer variables than the last.

$$\begin{aligned} F_1 &= F \\ F_2 &= proj(F_1, X_m) \\ &\vdots \\ F_m &= proj(F_{m-1}, X_2) \end{aligned}$$

F_m is the set which is processed as the base case, with a single variable X_1. Assuming that δ (i.e. the largest degree of any of the polynomials) remains constant, we obtain a recurrence relation for the size s_i of F_i.

$$\begin{aligned} s_1 &= n \\ s_{i+1} &= s_i^2.\delta^3 \end{aligned}$$

having the solution

$$s_m = n^{2^m}.\delta^{3(2^m)}$$

which is doubly exponential in the number of variables. In fact this is not quite true since δ may double with each iteration, from the PSC calculation, which leads to an even larger result, with larger polynomials in the set. This explains why the algorithm is not suitable for practical applications involving more than a few variables. For low dimensional problems, better projection functions are known, see for example [McC88].

6.4.3 Construction

In the construction phase the algorithm has calculated $CAD(proj(F, X_m))$ using recursion, and must calculate $CAD(F)$. The form of $CAD(proj(F, X_m))$ will be a list of sample points, one from each cell in the CAD, arranged in lexicographic order. The advantage of this ordering is that the stack structure is not lost, because

all the elements of the 'leftmost' stack (with respect to variable X_{m-1}) will come before all the elements of any other stack with respect to this variable, and so on. Since the projection of these stacks consist of alternations of open intervals and single points, we can deduce the dimension of the space spanned by each cell. To construct the new CAD, we take each sample point, $\mathbf{t}$ say, in $CAD(proj(F, X_m))$, and extend it to sample points in one more dimension ($\mathbb{R}^m$). (We know this is possible since F is delineable over each cell of the CAD). Suppose f is an element of F. Then we can substitute the co-ordinates $\langle t_1, \ldots, t_{m-1} \rangle$ of $\mathbf{t}$ for the variables $X_1, \ldots, X_{m-1}$ of f, arriving at a polynomial in only X_m, denoted $f_t(X_m)$. We do the same for all the elements of F, and generate a set of polynomials in X_m only, throwing out any which are constant functions. Using the same procedure as the base case, we solve these and order the roots $s_1, \ldots, s_r$. It can be seen that these points represent the heights of the leaves of $V(f)$ lying vertically above the point $\mathbf{t}$. These points generate a partition of $\mathbb{R}$ consisting of alternations of open intervals and single points as before ($2r+1$ altogether), so we choose one algebraic number, a_i, from each partitioning set. The output then consists of $\langle t_1, \ldots, t_{m-1}, a_i \rangle$ for each i. The procedure is repeated for each of the sample points in $CAD(proj(F, X_m))$.

6.5 Solving PSP using CAD

The CAD algorithm as outlined above is a useful tool for directly solving PSP, providing of course that the systems are small enough for the answers to be computed in a feasible length of time (see the example in appendix B).

Suppose $F = \{f_1, \ldots, f_n\}$ is a set of polynomials, with algebraic coefficients over variables $X_1, \ldots, X_m$, and $\delta_1, \ldots, \delta_n$ are non-empty subsets of $\{-1, 0, +1\}$, then to solve the system of inequalities

$$(*) \quad \forall\, i \in 1..n \mid sign(f_i(\mathbf{x})) \in \delta_i$$

we first compute $CAD(F)$. This constructs a set of sample points representing a possible CAD of the set of polynomials. From the definition of CAD we know that if (*) holds at one point in a cell, then it holds over the whole cell. Therefore we can find those cells in which (*) holds by substituting the co-ordinates of the sample points into the polynomials, and checking whether or not the sign conditions hold. Furthermore, by an analysis of the precise arrangement of the sample points we can determine the dimensions and extent of the cells, should this be necessary.

6.6 Solving POP using CAD

The mechanism for CAD also provides a way of solving POP. POP requires us to maximise a target polynomial $g(X_1, \ldots, X_m)$ subject to (*). My approach is to add a new variable X_0, and a new equation such that the new equation is satisfied when X_0 is equal in value to the target polynomial. Then I compute a CAD, and find the cell which maximises X_0, satisfies the new equation, and satisfies all the constraint equations. This can be done with a simple scan.

I define a new variable X_0, and define $\hat{f}_i(X_0, \ldots, X_m)$ to be $f_i(X_1, \ldots, X_m)$, that is, the same but with a dummy variable added. Finally, define $\hat{f}_0(X_0, \ldots, X_m)$ to be $X_0 - g(X_1, \ldots, X_m)$, with corresponding sign condition $\delta_0 = \{0\}$. Consider the new system of equations:

$$(**) \quad \forall\, i \in 0..n \mid sign(\hat{f}_i(\mathbf{x})) \in \delta_i$$

Suppose $\mathbf{x} = \langle x_0, \ldots, x_m \rangle$ is a solution of (**), then $\langle x_1, \ldots, x_m \rangle$ is a solution of (**), and furthermore, $x_0 = g(\langle x_1, \ldots, x_m \rangle)$. Therefore, to solve the optimisation problem, we require to find a solution to (**) which maximises x_0.

Consider a CAD for the polynomials $\hat{f}_0, \ldots, \hat{f}_m$. Since X_0 is the first variable in the list, it will be the one remaining at the base phase, and so we will form a partition of $\mathbb{R}$ described by a set $s_1, \ldots, s_r$ of values for X_0 (values which are the roots of the polynomials at this point). This leads to a set of sample points $t_1, \ldots, t_{2r+1}$ for the partitioning sets, satisfying the conditions that t_i is algebraic and

$$\begin{aligned} t_1 &< s_1 \\ t_{2i} &= s_i \\ s_i &< t_{2i+1} < s_{i+1} \\ t_{2r+1} &> s_r \end{aligned}$$

From the way the CAD is built during the construction phase, we can see that X_0 will be the primary sorting index for the samples. This means that if two sample points have the same X_0 co-ordinate, then the corresponding cells will have the same shadow when projected onto the X_0 axis. The converse is also true, and both facts are a consequence of the notion of delineability. Therefore a cell which satisfies (**) and maximises X_0 is one whose representative sample point satisfies (**) and maximises X_0. We can find such a cell by scanning the sample points in order of decreasing X_0 value.

Suppose now that such a maximising sample point $\mathbf{x}$ has x_0 equal to t_j as defined above. Then a maximising solution to the problem must lie within the corresponding cell. There are two cases.

- If j is even then $t_j = s_{j/2}$, and the only possible value for X_0 in this cell is t_j itself. The solution to the optimisation problem is therefore the point $\langle X_1, \ldots, X_m \rangle$, achieving the maximal value t_j for g.

- If j is odd, then t_j lies between adjacent values of s_i and s_{i+1} (or possibly between $-\infty$ and s_1 or s_r and $+\infty$. In this case there is no well-defined maximal value for g, but the value s_{i+1} (or $+\infty$) may be approached as closely as desired, within the cell represented by **x**.

In appendix B I present an encoding of the CAD algorithm in the MAPLE language [CFG+][CFG+86], and give an example of solving a POP in a small number of variables. The example shows that CAD is far too inefficient to be of practical use for anything but the smallest of problems.

Chapter 7

Gröbner Bases

In this chapter I examine the concept of Gröbner basis, as a possible means of solving the ADGP. Gröbner bases have been used in the `LineTool` geometric editor program by Lars Ericson and Chee-Keng Yap [EY88][Eri89], to solve geometric constraint problems in the plane.

Their program takes as input a set of polynomial equations representing constraints, and a dependency graph on (sets of) the input variables. It then computes the Gröbner basis of the input set, with respect to the dependent variables, and allows the user to ask questions about the construction, and assign values to the independent variables.

A good introduction to the uses of Gröbner bases, and a useful bibliography may be found in [Buc85], in conjunction with [Buc87]. [Eri89] contains a section on the historical development of the subject.

7.1 Definition of Gröbner basis

Suppose F is a non-empty finite set of non-zero polynomials with rational coefficients over a set $\mathbf{X} = \langle X_1, \ldots, X_n \rangle$.

$$F \subset \mathbb{Q}[X_1, \ldots, X_n] - \{0\}$$

The polynomials correspond to constraints between the X_i, and their solutions correspond to solutions of the constraint problem. Unlike the method of cylindrical algebraic decomposition, we restrict our constraints to be equalities only, that is,

polynomial equations of the form

$$f(\mathbf{X}) = 0$$

rather than inequations > 0, ≥ 0, $\neq 0$. However, these types of constraint can be constructed (at the expense of adding extra variables) using the observations

$$f(\mathbf{X}) \neq 0 \equiv \exists\, y \in \mathbf{R} \mid y \cdot f(\mathbf{X}) - 1 = 0$$

$$f(\mathbf{X}) \geq 0 \equiv \exists\, y \in \mathbf{R} \mid y^2 - f(\mathbf{X}) = 0$$

$$f(\mathbf{X}) > 0 \equiv \exists\, y \in \mathbf{R} \mid y^2 \cdot f(\mathbf{X}) - 1 = 0$$

where y is some new variable not occurring in F.

The objective of this technique is to triangularise the set F so that the variables can be eliminated one at a time, in a similar fashion to solving linear simultaneous equations.

To start with, we fix on an ordering $\prec$ on the collection of power products in our polynomial ring. A power product is a polynomial of the form

$$X_1^{i_1} X_2^{i_2} \ldots X_m^{i_m}$$

where i_j is a non-negative integer. That is, a pure product of variables, with only one term and no coefficient. Note that $1 = X_1^0 \ldots X_m^0$ is a power product, and that if u and v are, then so is their product $u \cdot v$.

The ordering that we choose is called pure lexicographic, and is defined (recursively) by

$$X_1^{i_1} X_2^{i_2} \ldots X_m^{i_m} \prec X_1^{j_1} X_2^{j_2} \ldots X_m^{j_m}$$

$$\equiv$$

$$(i_1 < j_1) \vee ((i_1 = j_1) \wedge (m \geq 2) \wedge (X_2^{i_2} \ldots X_m^{i_m} \prec X_2^{j_2} \ldots X_m^{j_m}))$$

In fact, the particular ordering chosen is not essential to the notion of a Gröbner basis, provided that it has the property that for all power products s, t, u,

$$(1 \neq u) \;\Rightarrow\; (1 \prec u) \tag{7.1}$$

$$(s \prec t) \;\Rightarrow\; (s \cdot u \prec t \cdot u) \tag{7.2}$$

However, each different ordering will lead to a different Gröbner basis, and this will affect how useful the basis is, and also how hard it is to compute.

Definition 7.1 *If f is a non-zero (multivariate) polynomial, then the* leading power product *of f, denoted $lpp(f)$, is the $\prec$-maximal power product which occurs in f with a non-zero coefficient. The coefficient of any power product u in f is written $coeff(f, u)$.*

Definition 7.2 *If f, g are polynomials, and F is a set of non-zero polynomials, we say f* reduces to g modulo F, *written $f \rightarrow_F g$, if and only if there exist a constant b, a power product u and an element h of F such that*

$$\begin{aligned} &g = f - b \cdot u \cdot h \\ &coeff(f, u \cdot lpp(h)) \neq 0 \\ &coeff(g, u \cdot lpp(h)) = 0 \end{aligned}$$

In other words, we subtract a multiple of h from f such that the leading coefficient of that multiple exactly cancels one of the (non zero) terms in f. This is similar to one step in the process of long division of polynomials, except that we do not insist on removing the largest possible term from f. A special case occurs when there is no reduction possible using the elements of F, and in such a situation we say that f is in normal form modulo F.

Definition 7.3 *A polynomial f is in* normal form modulo F *if and only if there is no g such that $f \rightarrow_F g$. Further, if f is in normal form modulo F, and there is a (finite) sequence f_0, f_1, f_k of polynomials such that for each i in the range $0 \leq i < k$ we have $f_i \rightarrow_F f_{i+1}$, and $f_k = f$, then we say f* is a normal form of f_0 modulo F.

Some important questions arise at this point. Are normal forms unique ? Can normal forms be computed ? Are normal forms useful? The answers to these questions are 'No', 'Yes' and 'Yes' respectively. The following example demonstrates a simple case where normal forms are not unique.

Example 7.1 Consider the set of polynomials $F = f_1, f_2$ where

$$\begin{aligned} f_1 &= x^2 - 1 \\ f_2 &= x^3 - 1 \end{aligned}$$

Then the two polynomials x and 1 can be reached in a single reduction step from x^3 using f_1 and f_2 respectively. Furthermore, both are normal forms modulo F, since they cannot be further reduced. Hence x and 1 are two different normal forms of x^3 modulo F.

The next lemma shows that normal forms always exist, and can be computed.

Lemma 7.1 *For any polynomial f_0 and set of non-zero polynomials f, there exists a polynomial which is a normal form of f_0 modulo F. Furthermore, such a normal form can be constructed in a finite time.*

Proof *[Buc76b] We construct the sequence $f_0, f_1, \ldots$ starting with f_0 and for each i obtaining f_{i+1} satisfying $f_i \rightarrow_F f_{i+1}$, provided that f_i is not already in normal form, in which case we are finished. Note that the algorithm is non-deterministic, since several reductions may be possible at each stage. The heart of the lemma is the proof that reduction cannot continue indefinitely, and may be shown to hold for any ordering $\prec$ of the terms, providing the basic properties of equations 7.1 and 7.2 hold.*

We have seen that normal forms always exist, but that they may be non-unique. A set of polynomials which satisfies this uniqueness property is of particular interest in solving equations, and in other applications.

Definition 7.4 *A set F of polynomials is called a* Gröbner basis *if and only if for all polynomials f, h_1 and h_2, if h_1 and h_2 are both normal forms of f modulo F, then $h_1 = h_2$. Furthermore, a Gröbner basis is called a* reduced Gröbner basis *if and only if each of its elements is in normal form modulo the other elements, and has a leading coefficient of* 1.

In such a case, normal forms are unique, and it makes sense to talk of *the* normal form of f modulo F, and so when F is a Gröbner basis, we define $S(F,f)$ to be that normal form.

Next we will see that given a set of polynomials, a Gröbner basis can be computed that has the same set of solutions as the original. We do this by use of a characterisation lemma.

Definition 7.5 *If f_1 and f_2 are polynomials, with leading coefficients c_1 and c_2, and leading power products u_1 and u_2 respectively, then if v is the least common multiple of u_1 and u_2, we define the* S-polynomial *of f_1 and f_2 to be*

$$SP(f_1, f_2) = \frac{v}{u_1} \cdot f_1 - \frac{c_1}{c_2} \cdot \frac{v}{u_2} \cdot f_2$$

Lemma 7.2 *[Buc70] F is a Gröbner basis if and only if for every pair $f_1, f_2 \in F$, $SP(f_1, f_2)$ reduces to 0 modulo F.*

The forward direction of the proof of lemma 7.2 is simple, since $SP(f_1, f_2)$ may be obtained from $\frac{v}{u_1} \cdot f_1$ by a single reduction (using f_2), and $\frac{v}{u_1} \cdot f_1$ also reduces to 0 (using f_1), which is a normal form. Hence, by the defining property of Gröbner bases, $SP(f_1, f_2)$ must also reduce to 0 modulo F. The reverse direction is more complicated, and can be found (in German) in [Buc70]

Using lemma 7.2 as a guide, we can construct an algorithm for finding Gröbner bases:

Algorithm $GB_1(F)$

```
local G, P, f1, f2, h;
G := F;
P := {{f1, f2} | f1, f2 ∈ F};
while (P ≠ ∅) do
    {f1, f2} := choose (P);
    P := P − {{f1, f2}};
    h := S(G, SP(f1, f2));
    if (h ≠ 0) then
        P := P ∪ {{g, h} | g ∈ G};
        G := G ∪ {h}
    fi
od
return G.
```

The termination proof for this algorithm may be derived from Hilbert's lemma on ascending chains of ideals [VdW53], whilst the partial correctness proof follows easily from lemma 7.2. The claim is that the solution set of $GB_1(F)$ (in $\mathbb{C}^m$) is the same as that of F. We let $V(F)$ denote the set $\{\mathbf{x} \in \mathbb{C}^m \mid \forall f \in F\}$, and let G denote $GB_1(F)$. Then the correctness proof is required to show that $V(G) = V(F)$, and that G is a Gröbner basis. Clearly $G \supseteq F$, and therefore $V(G) \subseteq V(F)$. If $\mathbf{x} \in V(F)$, then it is easy to prove by the method of invariants that all the polynomials appearing in P and G at any step in the algorithm have $\mathbf{x}$ as a solution. This is because of the easy facts (i) if $f(\mathbf{x}) = g(\mathbf{x}) = 0$ then $SP(f, g)(\mathbf{x}) = 0$, and (ii) if $\mathbf{x} \in V(F)$ and $f(\mathbf{x}) = 0$ then $S(F, f)(\mathbf{x}) = 0$. Hence $V(G) \supseteq V(F)$, and so $V(G) = V(F)$. Further, by lemma 7.2, we see that G is a Gröbner basis.

As we shall see, a reduced Gröbner basis is more useful than a general Gröbner basis. A reduced Gröbner basis can be obtained from a Gröbner basis by reducing

each element to normal form with respect to the other elements, and normalising the leading coefficients to 1 by scaling. Furthermore, the reduced Gröbner basis corresponding to a set of polynomials is unique.

Lemma 7.3 *[Buc76a] If G_1 and G_2 are reduced Gröbner bases, such that $V(G_1) = V(G_2)$ then $G_1 = G_2$. Therefore we may use the notation $GB(F)$ to denote the unique reduced Gröbner basis such that $V(F) = V(GB(F))$.*

The following lemma, in conjunction with lemma 7.1 shows that reduced Gröbner bases may be found from ordinary Gröbner bases, by repeated normalisation steps.

Lemma 7.4 *[Buc70]* Construction of reduced Gröbner bases: *Suppose G is a Gröbner basis, $g \in G$ and $h = S(G - \{g\}, g)$, then $G' = ((G - \{g\}) \cup (\{h\} - \{0\}))$ is also a Gröbner basis, and $V(G') = V(G)$.*

Buchberger[Buc85] presents several ways of optimising the Gröbner basis algorithm which incorporate the normalisation steps for very little extra cost. The normal form function, S, can be computed more efficiently by choosing at each iteration a reduction which removes the $\prec$-largest possible power product. The complexity of GB also depends on the particular power product used. Buchberger's results show that with lexicographic ordering, computation time can depend crucially on the ordering of the variables.

7.2 Method of Application

When a set F of polynomials has a finite number of solutions, we can use the method of Gröbner bases to substitute a set G of polynomials with the same solutions, but such that G is in a 'triangular' form. This means that there is one polynomial in the first variable only, which we can solve for that variable, and substitute the various solutions into all the other equations. To solve each of these subproblems we would find another polynomial in just one unknown, and recursively solve them all.

Definition 7.6 *A set G of multivariate polynomials in variables $x_1, \ldots, x_n$ over a field $\mathbf{A}$ is said to be in* triangular form *if and only if for each $1 \leq i \leq n$,*

$$\#(G \cap \mathbf{A}[x_{n-i}, \ldots, x_n]) = i$$

Lemma 7.5 *If G is a reduced Gröbner basis with respect to the lexicographic ordering on power products of $x_1, \ldots, x_n$, and $V(G)$ is finite, then G is in triangular form.*

Example 7.2 Consider for example the set F of polynomials

$$\begin{aligned} f_1 &= x^2 + y^2 - 1 \\ f_2 &= x - y^2 \\ f_3 &= x^2 - z \end{aligned}$$

These equations correspond geometrically to a circular cylinder and two parabolic cylinders, and we want to find a (real) point common to all three. Each polynomial is listed in decreasing order of power products, (assuming $z \prec y \prec x$). It would be easy to solve these with pen and paper, but let us follow the method through. Start with $G = F$. We calculate $SP(f_1, f_2) = xy^2 + y^2 - 1$, and then reduce to normal form modulo G. Using f_2 we arrive at $f_4 = y^4 + y^2 - 1$, and include it in G. Next, $SP(f_2, f_3) = xy^2 - z$ which reduces to $f_5 = -y^2 - z^2 + 1$ modulo G, using f_2 followed by f_4. We include f_5 in G. $S(G, SP(f_1, f_3))$ is zero, using f_5 to perform the reduction step, and so we may ignore it. In fact there is only one more polynomial to include in G, $f_6 = S(G, SP(f_3, f_4)) = z^2 - 3z + 1$, since all the other S-polynomials reduce to zero. Therefore one possibility for a Gröbner basis corresponding to F is

$$\begin{aligned} f_1 &= x^2 + y^2 - 1 & \qquad f_4 &\quad y^4 + y^2 - 1 \\ f_2 &= x - y^2 & \qquad f_5 &\quad -y^2 - z^2 + 1 \\ f_3 &= x^2 - z & \qquad f_6 &\quad z^2 - 3z + 1 \end{aligned}$$

To arrive at the reduced Gröbner basis, we take an element $f \in G$ and replace it by its normal form modulo $G - \{f\}$. Repeating this procedure until no more reductions are possible, and scaling all leading coefficients to 1, we arrive at the unique reduced Gröbner basis,

$$\begin{aligned} g_1 &= x + z - 1 \\ g_2 &= y^2 + z - 1 \\ g_3 &= z^2 - 3 * z + 1 \end{aligned}$$

Note that this is in triangular form (even though there are no y terms in g_1). We solve g_3 for z, getting $z = \frac{3 \pm \sqrt{5}}{2}$. Substituting these values into g_2, and solving for a (real) value of y we get

$$\begin{aligned} z &= \frac{3 - \sqrt{5}}{2} \\ y &= \pm\sqrt{\frac{-1 + \sqrt{5}}{2}} \end{aligned}$$

In general we would need values of y and z to solve g_1 for x, but in this case we only need z. Substituting this value of z into g_3 and solving we get

$$\begin{aligned} z &= \frac{3-\sqrt{5}}{2} \\ y &= \pm\sqrt{\frac{-1+\sqrt{5}}{2}} \\ x &= 1-\frac{3-\sqrt{5}}{2} \end{aligned}$$

7.3 More about LineTool

In `LineTool`, the user is required to define the dependency relationship between variables, nominating certain variables as independent parameters. It is important for this to be done correctly, as `LineTool` cannot function properly when there is an infinite or empty set of solutions for any assignment of the parameters. `LineTool` uses the method of Gröbner bases to solve the constraints, and discover the number of degrees of freedom of the solution space, which may be -1 (insoluble), 0 (finite number of solutions), 1 or more (under specified, infinitely many solutions). `LineTool` concentrates on the finite solution case, for algorithmic reasons. It is known that in the worst case, the Gröbner basis technique is doubly exponential in the size of input. However, when the solution is finite, this can be reduced to single exponential time [Bro87][CGH88]. The user can then examine the solution space by making assignments to the independent variables.

`LineTool` uses several devices to speed calculation time, in an attempt to make it an interactive tool. Firstly, it contains a speeded up version of the Gröbner basis algorithm [Buc85], and uses some clever data structures to implement head reduction (that is, if the leading term of a polynomial is not reducible, the algorithm should not try further reduction on that polynomial).

Ericson and Yap use a technique which they call the 'hierarchical method' for computing Gröbner bases. This is done using the dependency graph specified by the user. Each node in the graph may be identified with a set of variables, and hence with the set of polynomials which depend on this set and its dependents. Starting with the 'most independent' variables, Gröbner bases are computed for for the polynomial set at each node in the graph, at each level combining Gröbner bases for dependents with the Gröbner basis for input polynomials which are new at this node. This combination is easier than calculating a Gröbner basis for a random set of polynomials, since we know in advance many of the S-polynomials which will be zero. Ericson and Yap appear to give no indication of how effective this method is

in reducing the complexity or increasing the practicality of the algorithm.

In addition to the subtle refinements of the algebraic algorithms above, some more obvious optimisations are considered in the implementation of `LineTool`. One such technique is to check explicitly for variables which are identical. For example, if at any stage one of the constraining polynomials is $x - y = 0$ (or equivalently $x = y$), then we may reduce the number of free variables by 1, by substituting y for x and removing the equation $x - y = 0$. Another technique is to search for linear subproblems, and solve these using standard linear techniques, such as Gaussian elimination.

Chapter 8

Conclusion

In this thesis I have examined the introduction of angle constraints into the problem of distance geometry. I have described a previous numerical approach to the DGP, and shown how it may be extended to handle the ADGP. I have also discussed algebraic methods of solution of the ADGP, and shown how they lead to exact algebraically correct embeddings.

The ADGP appears to be strictly more expressive than the DGP. For example, I have found no way to constrain three nodes to be collinear (but otherwise unconstrained) using just distance information, but this is fairly simple using the construction in section 3.7. I have shown (section 4.8) that exact DGPs of low diameter have embeddings of low diameter, but that the same is not true of exact ADGPs. However, in a fixed number of dimensions, the numerical decision problem versions of both ADGP and DGP are NP-complete, and therefore of equivalent difficulty in a formal theoretical sense.

The graph theoretic methods of chapter 5 show that progress can be made in some cases of ADGP, and that it is not always necessary to solve a whole problem in one go with a general purpose minimising or equation solving algorithm.

8.1 Further Work

There are some useful geometric concepts which are awkward or impossible to model with the ADGP as defined, especially dimension dependent constructions. For example, in two dimensions it would be desirable to have a way of expressing the orientation of angle constraints, clockwise or anticlockwise. The embedding

of such a structure might be achievable using similar techniques to the standard ADGP. Special care is required when recombining substructures with such oriented angles, since the operation of reflection reverses orientation. Hartley[Har88] gives a linear time algorithm for embedding in $\mathbb{R}^2$ a polygon whose external angles are specified by signed values in the range $[-\pi, \pi]$ (and no edge lengths given).

The JUNO constraint based graphics system[Nel85] permits constraints of the type that two edges have the same length. This could be included into the ADGP framework, using an algorithm which updated all the length constraints of both equal edges whenever one changed. Extra constraint edges are required in the graph representation to show that the two edges are interdependent. However, problems arise for relocalisation and chain removal in producing transformation rules which are guaranteed to preserve embedding properties. Extending the concept of length equality to include angle equality constraints is an obvious step, allowing the specification of isosceles triangles without requiring any further information. Another of JUNO's standard features is the parallel line constraint. For any fixed dimension d this constraint can be encoded into an ADGP, however, the construction requires $O(d)$ extra nodes. It might be possible to include parallel edges as a primitive in an ADGP. Similar problems might be expected as with equal length edges. It would be interesting to specify not just equal lengths or angles, but to express other relations between them, such as "greater than," "double" and so on, and to see how far these kinds of equational relationships could be extended without having to resort to general purpose methods. With general purpose algebraic algorithms these problems are all equally soluble.

Other generalisations of ADGP are possible to include higher dimensional concepts than length and planar angle. For example, the angles between adjacent faces may be specified by supplying angle limits for quadruples of nodes. This would be of use for constraining 3 dimensional objects constructed out of flat sheets. Also the possibility exists for constraining areas of triangles, volumes and hypervolumes and so on. All these values may be expressed as (algebraic) functions of the co-ordinates of the nodes. It is interesting to speculate whether decomposition rules exist for these properties.

Lovász and Yemini[LY82] and Laman[Lam70] present characterisations of rigid graphs in the plane. These graphs correspond to exact DGPs, with some lengths known to be exact, but not actually given. That is, the constraint graph is known, but the values of the constraints are not. The notion of rigidity used is that the structure cannot be moved (continuously) to another position without violating the constraints (ignoring rigid motions of the whole embedding space). Note that this is not the same as my concept of a fixed graph (chapter 5), since there may be several isolated solutions. It would be interesting to try to extend this work to

arbitrary dimensions and angle constraints, and to use the results to improve on the methods of solving DGP and ADGP using fixed subgraphs.

Related to the ADGP and the rigidity problems are the path problems. Here the question is whether a structure may be continuously deformed to another structure whilst maintaining the constraints. This could be of use in robot arm path planning, to decide if the arm can manœuvre so that an object can be grasped. In molecular geometry it is interesting to know whether a molecule is free to move between two different states without requiring extra energy to overcome its constraints. A possible approach is given by the CAD algorithm (chapter 6), which provides information about connected regions of embedding space in which the constraints are satisfied. As has been discussed, however, the CAD algorithm is very slow, and it would be desirable to discover new graph theoretic ways of simplifying and solving the problem.

The general CAD and Gröbner basis solutions to algebraic problems are too inefficient in their original form to be considered as serious methods for solving ADGP. These problems may be alleviated by the use of heuristic methods, perhaps to identify subcalculations which cannot lead to solutions. Since the polynomials defining ADGPs are all of low degree there is some hope that such methods may exist.

Appendix A

Triangles

In this appendix I show how upper and lower bounds for one side of a triangle can be computed from the constraints in the rest of the triangle in two particular cases. The first case is relevant to chain removal (section 5.7.1) and angle-length propagation (section 3.4.1), whilst the second is relevant only to angle-length propagation.

A.1 Using the Cosine Rule

The cosine rule is suitable for propagating information to a side of a triangle in which there is information about the other two sides and the opposite angle, as in figure A.1. Suppose that the sides are of lengths a, b and c and that the angle opposite c is θ, with constraints $L_a \leq a \leq U_a$, $L_b \leq b \leq U_b$ and $\alpha \leq \theta \leq \beta$. Suppose further that we are trying to find (or tighten) the constraints L_c and U_c on c. I assume in what follows that when an infinite upper bound appears in an expression the following formal rules hold, that for any positive real number r, $r + \infty = r.\infty = \infty.\infty = \infty$, $\max\{r, \infty\} = \infty$ and $\min\{r, \infty\} = r$.

The cosine rule states that $c = \sqrt{a^2 + b^2 - 2ab\cos\theta}$, and the object is to find the maximum and minimum (U_c and L_c) possible values of c subject to the constraints on a, b and θ. First, since c is positive it is maximised or minimised when c^2 is. For convenience define $f(a, b, \theta)$ to be $a^2 + b^2 - 2ab\cos\theta$. Then $c^2 = f(a, b, \theta)$, which is an increasing function of θ in the range $[0, \pi]$, so c is maximised when $\theta = \beta$, and minimised when $\theta = \alpha$. Note that since $f(a, b, \theta) \geq (a - b)^2$ it is always the case that f is positive, and so $\sqrt{f}$ is well defined.

Consider the upper bound U_c. If $\beta \geq \pi$ then $f(a, b, \beta)$ is an increasing function of

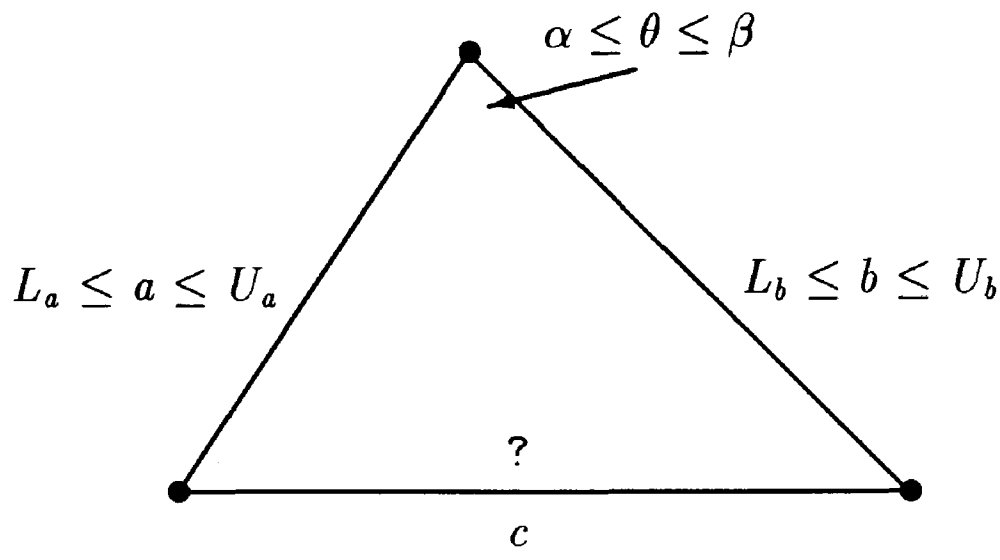


Figure A.1: A case for the cosine rule

a and b, so the maximum is achieved for $a = U_a$ and $b = U_b$, leading to a value $U_c = \sqrt{f(U_a, U_b, \beta)}$. Otherwise, if $\beta \leq \pi$, then $f(a, b, \beta)$ is convex and so must be maximised at one of the extreme points of $[L_a, U_a] \times [L_b, U_b]$, so

$$U_c = \sqrt{\max\{f(U_a, U_b, \beta), f(L_a, U_b, \beta), f(U_a, L_b, \beta), f(L_a, L_b, \beta)\}}$$

Now consider the lower bound L_c. If $\alpha \geq \pi$ then $f(a, b, \alpha)$ is an increasing function of a and b, so the minimum is achieved for $a = L_a$ and $b = L_b$, leading to a value $L_c = \sqrt{f(L_a, L_b, \alpha)}$. Otherwise, if $\alpha \leq \pi$, the situation is more complicated. Suppose a is fixed, then $f(a, b, \alpha)$ achieves a minimum of $a^2 \sin^2 \alpha$ when $b = a \cos \alpha$. Similarly, if b is fixed, then $f(a, b, \alpha)$ achieves a minimum of $b^2 \sin^2 \alpha$ when $a = b \cos \alpha$. These minima are increasing functions of the 'fixed' variable in each case. With these facts in mind, define d_a to be $L_b \sin \alpha$ if $L_b \cos \alpha \in [L_a, U_a]$, and ∞ otherwise, and define d_b to be $L_a \sin \alpha$ if $L_a \cos \alpha \in [L_b, U_b]$, and ∞ otherwise. Then the geometry leads to the conclusion

$$L_c = \sqrt{\min\{f(L_a, L_b, \beta), f(U_a, L_b, \beta), f(L_a, U_b, \beta), f(U_a, U_b, \beta), d_a^2, d_b^2\}}$$

A.2 Using the Sine Rule

The sine rule is suitable for propagating information to a side of a triangle in which there is information about one other side and two angles, as in figure A.2. Suppose that the side has length $a \in [L_a, U_a]$, and the two angles are $\theta_a \in [\alpha_a, \beta_a]$ and

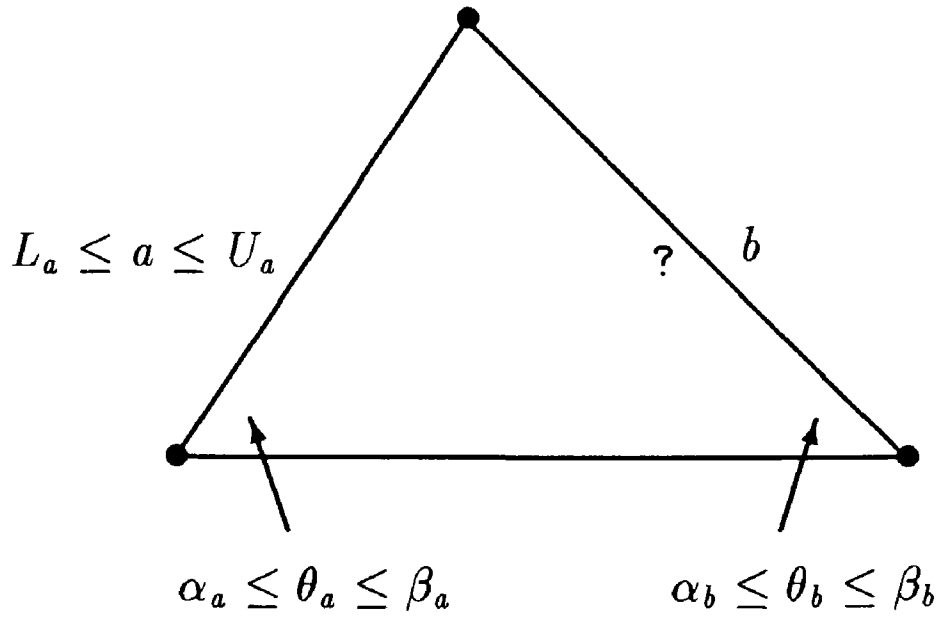


Figure A.2: A case for the sine rule

$\theta_b \in [\alpha_b, \beta_b]$. Suppose further that we are trying to find (or tighten) the constraints L_b and U_b on b. It is assumed that the rules for manipulating ∞ are the same as in section A.1.

The sine rule states that $b \sin\theta_a = a \sin\theta_b$, and the object is to find the maximum and minimum (U_b and L_b) possible values of b subject to the constraints on a, θ_a and θ_b. For convenience, define the following two functions for angles within the range $[0, \pi]$, and $\alpha \leq \beta$.

$$minsin(\alpha, \beta) = min\{sin\theta \mid \theta \in [\alpha, \beta]\}$$

$$maxsin(\alpha, \beta) = max\{sin\theta \mid \theta \in [\alpha, \beta]\}$$

If $minsin(\alpha_a, \beta_a) \neq 0$, then it is easy to see that U_b should be $\frac{a\ maxsin(\alpha_b, \beta_b)}{minsin(\alpha_a, \beta_a)}$. Otherwise, $U_b = \infty$.

If $maxsin(\alpha_a, \beta_a) \neq 0$, then it is easy to see that L_b should be $\frac{a\ minsin(\alpha_b, \beta_b)}{maxsin(\alpha_a, \beta_a)}$. Otherwise, θ_a is a fixed angle, either 0 or π. In the former case, $L_b = L_c - U_a$ and $U_b = U_c - L_a$. In the latter case, $L_b = L_a - U_c$ and $U_b = U_a - L_c$. The new bounds must be checked for consistency (ie $0 \leq L_b \leq U_b$), and if they are not consistent then the triangle is not embeddable.

It only remains to calculate the *maxsin* and *minsin* functions. From the shape of the sine graph between $[0, \pi]$ we see that $minsin(\alpha, \beta) = \min\{\sin\alpha, \sin\beta\}$. Further, if $\frac{\pi}{2} \in [\alpha, \beta]$ then $maxsin(\alpha, \beta) = 1$, and otherwise $maxsin(\alpha, \beta) = \max\{\sin\alpha, \sin\beta\}$.

Appendix B

CAD Implementation

In this appendix I present a program to perform CAD and optimisation using the algebraic manipulation language MAPLE 4.2 [CFG+][CFG+86] (see figures B.3 et. seq.). The algorithm is simplified slightly, as I have not used any special routines for manipulating algebraic numbers. Instead, I have just used the built in limited precision floating point capability of MAPLE to approximate the solutions. This seems to work reasonably well for low dimensional problems, although it is possible that some cells of the CAD may vanish because their size is too small to be distinguishable in the approximation, and some may be represented twice if repeated roots are perturbed so that they become two single roots. Several languages (for example MACSIMA, Mathematica etc.) exist which can deal with arithmetic of algebraic numbers, however none of these were available on the machines to which I had access. It would be straightforward to substitute algebraic arithmetic procedures for the various arithmetic operations used in the construction phase, but in general these will be very much slower than the standard functions. Unfortunately the rate of growth of the algorithm in time and space means that on our Sun workstation algebraic problems with four or more variables cannot terminate successfully. Problems in three variables may take several minutes.

The program produces a list of sample points for the CAD of its input. It also produces an index for each one, as described in [ACM84a]. Briefly, the index of a cell is the information required to find out which stack, sub-stack, sub-sub-stack, etc. it lies in. By counting the number of even and odd co-ordinates in an index, we can determine the dimension of a cell. In fact this information is all available in the sample point structure, although it is slightly less convenient to compute.

I provide an example CAD produced by the program (figure B.1), an example optimisation (figure B.2), and a listing in MAPLE (figures B.3 et. seq.).

```
[x = 0, y = -2.000000000],
[x = -1, y = -1.000000000],
[x = 0, y = -1.000000000],
[x = 1, y = -1.000000000],
[x = -2.000000000, y = 0],
[x = -1.000000000, y = 0],
[x = 0, y = 0],
[x = 1.000000000, y = 0],
[x = 2.000000000, y = 0],
[x = -1, y = 1.000000000],
[x = 0, y = 1.000000000],
[x = 1, y = 1.000000000],
[x = 0, y = 2.000000000]
```

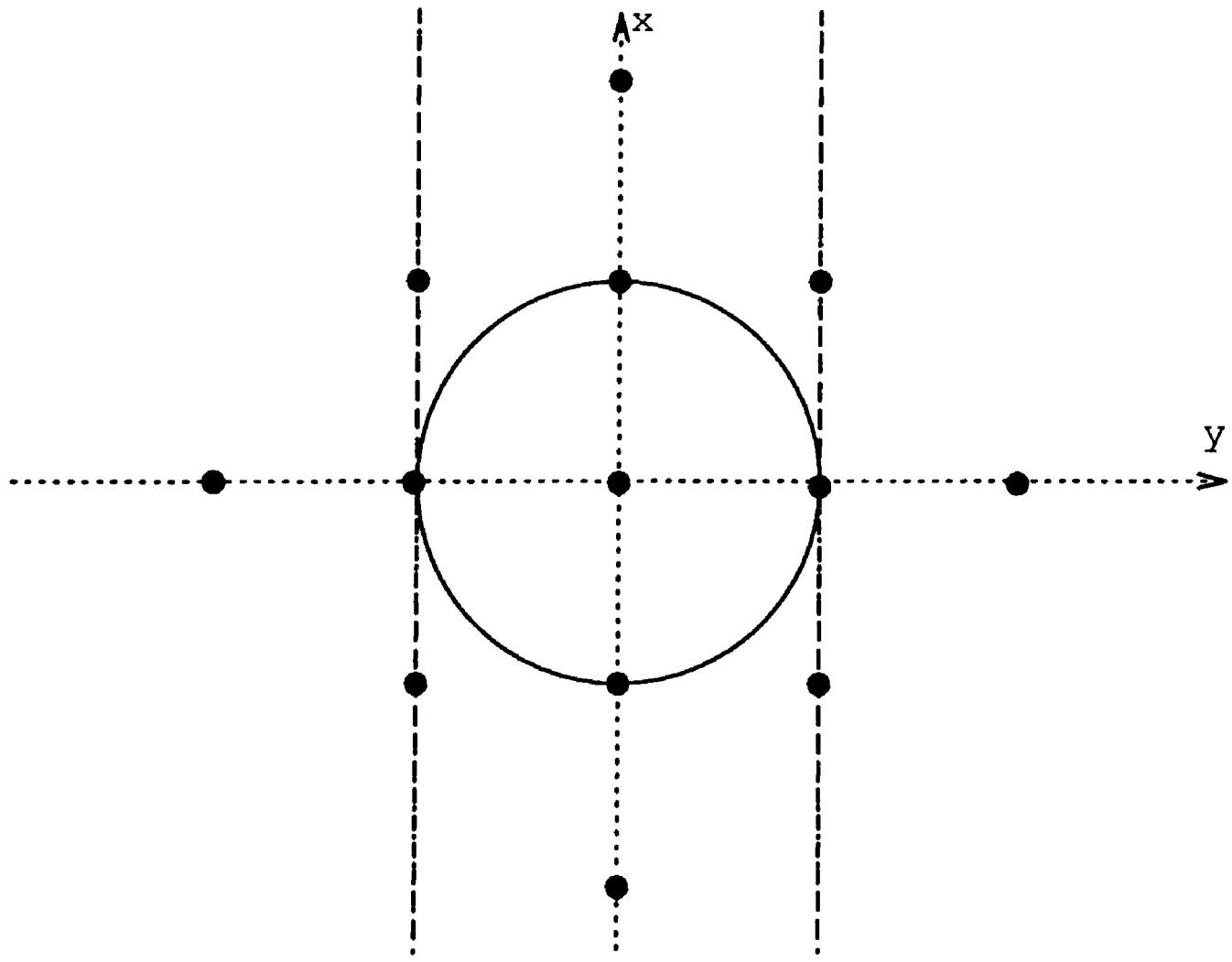


Figure B.1: Set of sample points produced by the program for a CAD of $x^2 + y^2 - 1$, projecting first with respect to x.

```
A := [x^2+y^2-1,z-x-y]:
vars := [x,y,z]:

c:=cad(A, vars):
si := c[1]:
ss := c[2]:
for i from nops(ss) by -1 to 1 do
         v1 := subs(ss[i], A[1]);
         v2 := subs(ss[i], A[2]);
         if (abs(v1)<=0.000001 and abs(v2)<=0.000001)
            then
                lprint(si[i],ss[i]);
                break;
         fi:
od:

[5, 5, 16]    [x = .7071067804, y = .7071067823, z = 1.414213563]
```

Figure B.2: An example of optimisation.

Figure B.1 provides an example of an optimisation performed by the CAD program. The problem under consideration is to maximise $x + y$ subject to $x^2 + y^2 = 1$. The new variable z is added to the list of variables, and the equation $z - x - y$ is added to the list of polynomials. The for-loop checks for the cell with highest z-value which satisfies the constraints (to within a certain degree of accuracy). The answer given is shown below the input. Since the z co-ordinate of the cell index (16) is even, this result is an attainable maximal value for z. The results are correct to 8 figures in this instance, the exact values being $x = y = \frac{1}{\sqrt{2}}$, $z = \sqrt{2}$. The complete CAD has over 900 cells!

```
# cad (F, vars) produces a cad of the list of
# polynomials F wrt the list vars of variables.
# The output consists of an indexing scheme and
# a list of sample points.

cad := proc(F, vars)
    local i, nvars, root, roots, nroots, sample, mid, j,
          mids, newI, newS, IS, P, PF, ii, si, Ac, v;

    mid := proc(a,b) (a+b)/2 end;
    nvars := nops(vars); v:=vars[1];
    if nvars=1 then

###                 Base Case

      roots := usort(map(fsolve, F, v));
      nroots := nops(roots);
      if (nroots > 1) then
          mids :=[ [[v=roots['i']],
                    [v=mid(roots['i'],roots['i'+1])]]$i=1..nroots-1];
          sample := [[v=roots[1]-1],
                     op(map(op,mids)),
                     [v=roots[nroots]],
                     [v=roots[nroots]+1]];
         [ [ ['i'] $ i=1..nops(sample)], sample ]
      elif (nroots=1) then
          root := roots[1];
          [ [[1],[2],[3]], [[v=root-1],[v=root],[v=root+1]] ];
      elif (nroots=0) then
          [ [[1]], [[v=0]] ];
      fi

    else

###                 Projection

      P := proj(F,vars[1]);
      PF := product(F['i'], 'i'=1..nops(F));
      IS := cad(P, [vars['i'] $ 'i'=2..nvars] );
```

Figure B.3: Main CAD program (page 1)

```
###                Construction

      newI := []; newS := [];
      for i from 1 to nops(IS[1]) do
          ii := op(i,IS[1]);
          si := op(i,IS[2]);
          Ac := subs(si, PF);
          roots := usort([fsolve(Ac, v)]);
          nroots := nops(roots);
          if (nroots>1) then
              mids :=[ [v=roots['j'],
                        v=mid(roots['j'],roots['j'+1])]
                                    $ 'j'=1..nroots-1 ];
              mids := map(op,mids);
              sample := [v=roots[1]-1,
                         op(mids),
                         v=roots[nroots],
                         v=roots[nroots]+1];
              newI := [op(newI), ['j', op(ii)] $ j=1..nops(sample)];
                  newS := [ op(newS),
                            [sample['j'],op(si)] $ j=1..nops(sample) ];
          elif (nroots=1) then
              root := roots[1];
              newI := [op(newI),[1,op(ii)],[2,op(ii)],[3,op(ii)]];
              newS := [op(newS), [v=root-1, op(si)],
                         [v=root, op(si)], [v=root+1, op(si)]  ];
          elif (nroots=0) then
              newI := [op(newI), [1, op(ii)] ];
              newS := [op(newS), [v=0, op(si)] ];
          fi
      od;
      [newI, newS];

    fi
end:
```

Figure B.3: Main CAD program (page 2)

```
# proj(F, x) returns the projection set of F wrt the variable x
# as defined in [ACM84a]
#
proj := proc(F, x)
    local F1, PROJ1, PROJ2, R, i, j, k, l, Ri, Rj, Rik;
    F1 := [op({op(map(sqfactor, F, x))} minus {0})];
    if F1=[] then RETURN ({}) fi;
    PROJ1 := {};
    R := map(REDSET,F1,x);
    for i from 1 to nops(F1) do
        Ri := [op(R[i])];
        for k from 1 to nops(Ri) do
            Rik := Ri[k];
            PROJ1 := PROJ1 union
                     {lcoeff(Rik,x)} union
                     PSC(Rik, diff(Rik, x), x);
            od;
        od;
    PROJ2 := {};
    for i from 1 to nops(F1)-1 do
        Ri := [op(R[i])];
        for j from i+1 to nops(F1) do
            Rj := [op(R[j])];
            for k from 1 to nops(Ri) do
                for l from 1 to nops(Rj) do
                    PROJ2 := PROJ2 union PSC(Ri[k], Rj[l], x)
                    od
                od
            od
        od;
    (PROJ1 union PROJ2) minus {0}
end:
```

Figure B.4: The projection operation

```
# PSC(f,g,x) computes the principal subresultant coefficient
# set for the polynomials f and g w.r.t. variable x. See
# [ACM84] and [BT71].
    PSC := proc(f,g,x) local j1, j2, R, fi, fpi, fni, PSC, fx, gx;
        fx := expand(f); gx := expand(g);
        if fx=0 or gx=0 then RETURN({}) fi;
        if degree(fx,x) < degree(gx,x) then
            fpi:=gx; fi:=fx;
        else
            fpi:=fx; fi:=gx;
        fi;
        R := {1};
        while degree(fi,x)>0 do
#           one step in the Euclid algorithm,
#           using pseudo division for polynomials
            fni := expand(primpart(prem(fpi,fi,x)));
            fpi := fi; fi := fni;
            j1 := degree(fi,x); j2 := degree(fpi,x)-1;
            R := R union
                {primpart(coeff(fi,x,j1)),
                 primpart(coeff(fi,x,j2))};
            od;
        R minus {0};
        end:
```

Figure B.5: Principal subresultant coefficient computation

```
# REDSET(f, x) computes the reducta set of f with respect to x.
    REDSET := proc(f,x) local R, redj, df, lc, t;
        df := degree(f,x);
        R := {expand(f)}; redj := expand(f);
        from 0 to df do
            lc := lcoeff(redj,x,'t');
            redj := expand(redj - lc * t);
            R := R union {redj}
            od;
        R;
    end:
```

Figure B.6: Reducta set computation

```
# sqfactor(f) performs factorisation of a multivariate poly, and
# returns the primitive squarefree factors
sqfactor := proc(f,x) local pos, ss;
        pos := proc(f) if lcoeff(f)<0 then -f else f fi end;
        pos(primpart(quo(f, primpart(gcd(f, diff(f,x)), x), x))) ;
        end:

# usort(L) takes a list, set or expression sequence of
# numbers, and sorts them in increasing order, removing
# duplicates, returning a list.
usort := proc(L) sort([op({op(L)})], `<`) end:
```

Figure B.7: Auxilliary definitions for the CAD program

Appendix C

A Parallel Algorithm for Three Dimensional Convex Hulls

The main result of this appendix is an $O(n)$ algorithm for calculating the faces of the convex hull of n points in three dimensional space. The algorithm requires $O(n^2)$ constant size processors connected in an n by n mesh-like configuration. The best sequential algorithms are $\Theta(n \log n)$ [PS85], but they are complicated, and have large constants of proportionality. In contrast, algorithm presented here is simple, and has a small constant of proportionality.

In two dimensions, the best sequential algorithms are $\Theta(n \log n)$. Bernard Chazelle has a $O(n)$ parallel implementation using $O(n)$ processors connected in series, with constant sized stores [Cha84]. My algorithm is a generalisation of this to three dimensions, and is also suitable for implementation on a systolic array.

Definition C.1 *Let S be a subset of $\mathbb{R}^n$. We say S is* **convex** *if and only if for every pair of points $p, q \in S$ the whole line segment connecting p and q is a subset of S. More formally,*

$$\forall\, S \in \mathbb{P}\,\mathbb{R}^n \mid \quad convex(S) \Leftrightarrow \forall\, u, v \in S;\ \forall\, t \in [0,1] \mid t\ u + (1-t)\ v \in S$$

Definition C.2 *If S is a subset of $\mathbb{R}^n$, define the* **convex hull** *$CH(S)$ of S to be the smallest convex set containing S. Note that since $\mathbb{R}^n$ is itself convex, CH is well defined. Formally,*

$$\forall\, S \in \mathbb{P}\,\mathbb{R}^n \mid CH(S) = \bigcap\{C \in \mathbb{P}\,\mathbb{R}^n \mid S \subseteq C \wedge convex(C)\}$$

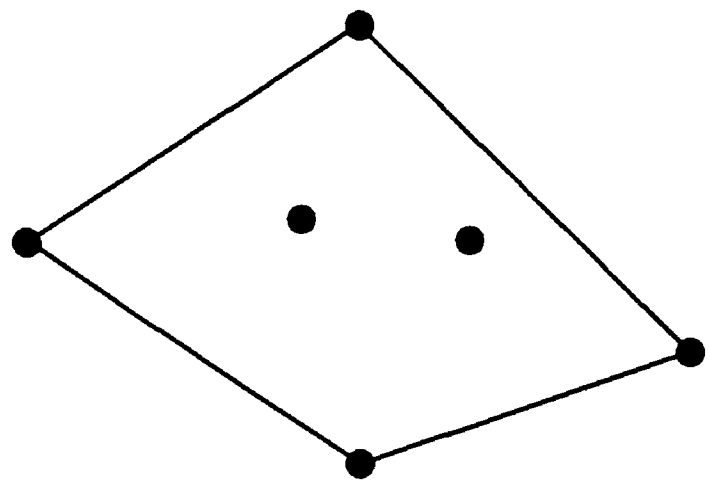


Figure C.1: The convex hull of a set of points in $\mathbb{R}^2$

Calculating the convex hull of a set of points has uses in several fields. In robotics, an obstacle may be represented by its convex hull as a first approximation. It may be possible then to compute the path of the robot as though the whole hull were present, and the full detailed structure of the original obstacle may be ignored. For complicated objects with many cavities and internal detail, this could vastly speed up the calculation. In statistics, a sample of d variables can be regarded as a point in d dimensional space. When many samples are taken, it may be necessary to discard outlying points, as rogue values. One way of doing this is to construct the convex hull of the points, and remove those which lie on the boundary. Successive convex layers may be stripped off until a certain number of points have been removed. Further applications may be found in [PS85, chapter 4.2].

We expect a convex hull algorithm to return a representation for the convex hull of a finite set of m input points. In two dimensions, it is a (filled) polygon, whose boundary can be represented by at most m oriented line segments. In three dimensions, the convex hull is a solid polyhedron and may be represented by a set of at most $2m - 4$ oriented triangles which form its boundary, as shown in section C.2. It can be shown that in n dimensions the convex hull is an n dimensional polytope[1], i.e. the boundary consists of sections of hyperplanes.

C.1 The Two Dimensional Algorithm

I begin by explaining Chazelle's algorithm, which I refer to as BCH. His algorithm maintains a set of points P in $\mathbb{R}^2$, and supports the following operations.

[1] the dimension may be lower in degenerate cases, but it is still be a polytope

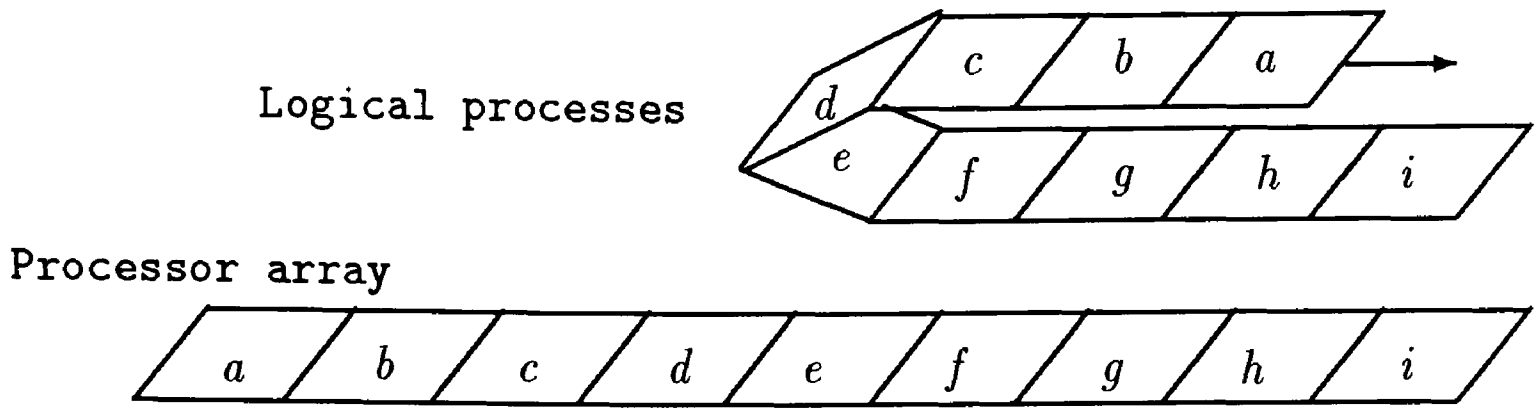


Figure C.2: The fold-over operation.

1. Insert point M into P. (time $O(1)$)
2. Delete point M from P.(time $O(1)$)
3. Report the vertices of $CH(P)$ in order.(time $O(n)$)
4. Query whether point M lies inside or outside $CH(P)$.(time $O(n)$)

The query operation, although it takes time $O(n)$ to produce the answer (latency), can be followed by another operation after a $O(1)$ delay, resulting in high throughput. BCH consists of two parts, BCH1 and BCH2, where the output from BCH1 is provided as input to BCH2, forming a pipeline of double length. BCH1 implements a data structure which can perform insertions and deletions on a set, by storing one element at each processor. No geometry is performed until a report or query instruction is received. When a report is called for, all the stored elements are processed in pairs in time $O(n)$ by sliding the data in a fold-over operation (figure C.2). Each data point corresponds to a single data structure called a **wedge**, and it is this structure which changes as it passes other elements. The wedge $W = (M, A, B)$ at point M represents the infinite sector of the plane contained between the extensions of the lines MA and MB (figure C.3), subtending a non-reflex angle (i.e. one less than π radians) at M. At any time, this sector must contain all of the points so far "seen" by W, so initially it is empty. When a new point X is encountered, there are four possibilities.

1. If X is contained in the wedge represented by W, then W is not changed.
2. If B is contained in the wedge (M, A, X), then W becomes (M, A, X)
3. If A is contained in the wedge (M, X, B), then W becomes (M, X, B)

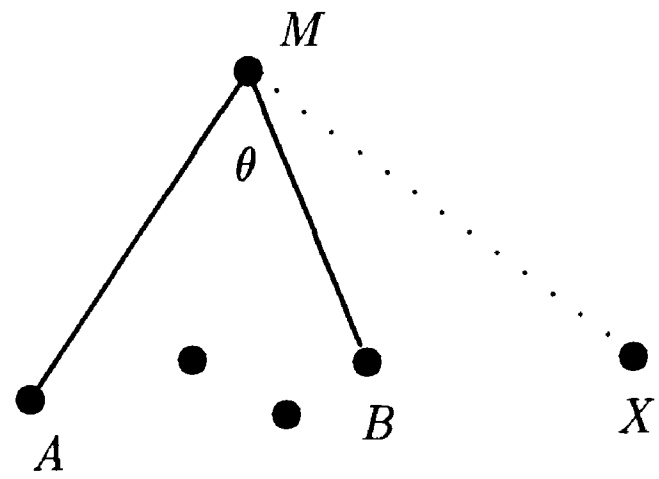


Figure C.3: Algorithm BCH1, Extending the wedge (M, A, B) to include a new point X

4. Otherwise there exists no wedge with a non-reflex angle containing all the points A, B and X, so a flag is set indicating that M is an interior point of the convex hull.

If, after all the points X have passed by, the flag is not set, then M is one of the vertices of the hull, and can be output. In fact, in this case, if (M, A, B) is the final wedge at point M then MA and MB are edges of the hull, although Chazelle does not use this fact.

The function **side** is used to determine the new value for the wedge. For any points A, B and C in $\mathbb{R}^2$, $side(A, B, C)$ returns 1 if, when viewed from A, C is to the left of AB, -1 if it is to the right of AB, and 0 if A, B and C are collinear. Formally,

$$side((a_x, a_y), (b_x, b_y), (c_x, c_y)) = sign(c_x.(a_y - b_y) + c_y.(b_x - a_x) + (a_x b_y - a_y b_x))$$

$$\begin{aligned} where\ sign(x) &= -1 \quad if \quad x < 0 \\ &= 0 \quad if \quad x = 0 \\ &= 1 \quad if \quad x > 0 \end{aligned}$$

Some pseudo code for updating a wedge is given in figure C.4, under the simplifying assumption that no three points are collinear. I shall have more to say about this assumption later when discussing degeneracies in three dimensions.

BCH2 sorts the output from BCH1 into cyclic order, and outputs the results. The advantage of this is that successive points in the output correspond to the endpoints

```
make_new_wedge(centre, W)
    {W = (centre, centre, centre); return;}

boolean add_to_wedge((M, A, B), X)
\* updates (M,A,B) and returns TRUE iff would
   make the included angle exceed pi              *\
    {
    if (A=B=M) \* wedge is new *\
        {A:=X; B:=X; return FALSE;} \* wedge now nearly new *\
    else if (A=B) \* wedge is nearly new *\
        {B:=X; return FALSE}
    else
        {
        ss1 := (side(M,B,A) = side(M,B,X))
               \* are A and X on the same side of MB ? *\
        ss2 := (side(M,A,B) = side(M,A,X))
               \* are B and X on the same side of MA ? *\
        if (ss1 AND ss2) return FALSE;
        if (ss1 AND (NOT ss2)) {A:=X; return FALSE};
        if ((NOT ss1) AND ss2) {B:=X; return FALSE};
        return TRUE;
        }
}
```

Figure C.4: Updating the wedge structure

of the boundary line segments. In three dimensions however, this concept is not so useful, since there is no linear ordering in which adjacent triplets represent boundary (triangular) faces. Therefore the three dimensional algorithm actually outputs a set of faces, rather than a set of vertices. For this reason, a study of BCH2 is mostly irrelevant to the understanding of the three dimensional case, and so I do not discuss it here.

C.2 The Three Dimensional Algorithm

The three dimensional convex hull algorithm presented here, 3CH, supports Insertion, Deletion and Report hull faces operations. It is assumed that all faces of the hull are triangular, and if not, a triangulation of each face is obtained, without adding additional vertices to the plane of the face.

I suppose that $\prec$ is some total ordering defined on points of $\mathbb{R}^3$, having the property that for all distinct points A, B, C in $\mathbb{R}^3$, if point C lies on the line segment AB then either $A \prec C \prec B$ or $B \prec C \prec A$. One possibility is to use lexicographic ordering on the co-ordinates of the points.

The algorithm proceeds by calculating in parallel a three-dimensional wedge for each possible edge of the convex hull. Each wedge is delimited by the two triangles formed by the edge in question and the two points representing the widest angular spread so far seen (figure C.5). As before, if the included angle ever exceeds π then the current edge is not an exterior edge. Otherwise, if all the points have been 'seen' and the angle is still less than π then the bounding triangles of the wedge are in fact bounding triangles of the convex hull, and may be added to the output list. One possible problem here is that each exterior edge, PQ, is found in the representation of bounding triangles three times (ie in $\triangle PQR$, $\triangle RPQ$ and $\triangle QRP$), and so unless care is exercised there is three times more output than required. One way of avoiding this is to allow the process examining PQ to output $\triangle PQR$ if and only if $(P \prec R)$ and $(Q \prec R)$.

Since I claim that the algorithm has total parallel complexity $O(n)$, where n is the number of input points, it is important to check that the size of the output is no more than $O(n)$. Each output consists of three points, describing one of the faces of a triangulation of the convex hull. I am able to ensure that no two triangles overlap, so they form all the faces of a polyhedron. Since some points may not lie at the vertices of the polyhedron, there are m vertices, for some number m less than or equal to n. Suppose there are T triangles. Each triangle has three edges, and each edge is common to precisely two triangles, hence there are $3T/2$ edges.

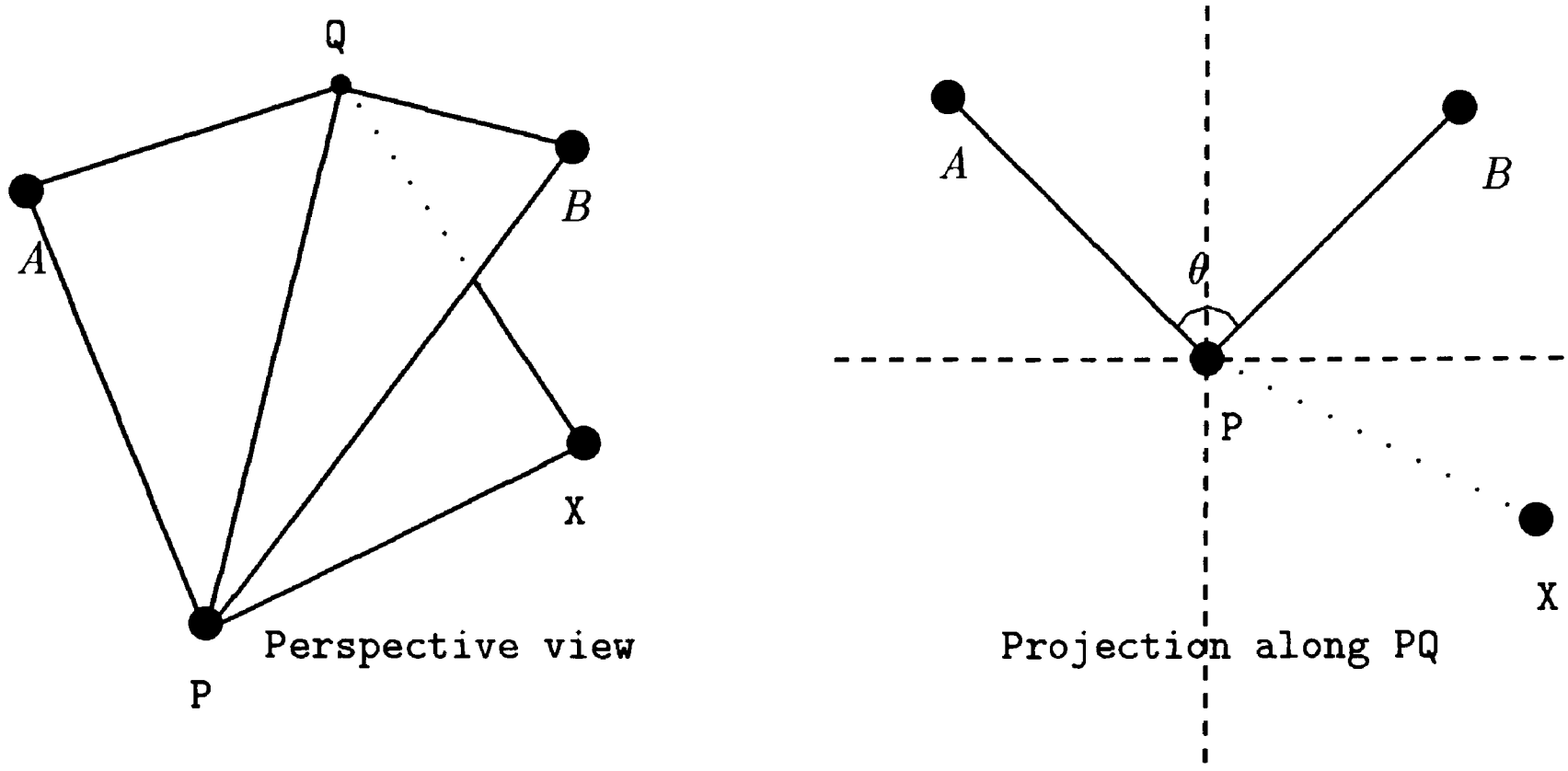


Figure C.5: The concept of wedge in three dimensions

By Euler's theorem [Lak76],

$$m + T - 3T/2 = 2$$

and rearranging

$$T = 2m - 4 \leq 2n - 4 \leq O(n)$$

so the number of triangles is linear in the size of the input.

The wedge corresponding to PQ is computed by projecting $\mathbb{R}^3$ into $\mathbb{R}^2$ along the line PQ, so that P becomes the new origin, and the centroid G of the input set of vertices lies along the x-axis. We then form the wedge based on the projected points, centred at the origin. It is easy and useful to calculate G, since it is guaranteed to be an interior point of the hull (except in some degenerate cases below). This allows us to determine which side of each face is the 'inside' and provides a simple method for producing all the output with a particular orientation. For example, we can insist that all triangles should be listed in a clockwise order when seen from outside the hull.

Suppose X is a point in $\mathbb{R}^3$, then the projection $\pi(X)$ into $\mathbb{R}^2$ is defined as follows,

$$\pi(X) = \begin{bmatrix} \mathbf{x} \cdot (\mathbf{p} \wedge \mathbf{g}) \\ \mathbf{x} \cdot ((\mathbf{p} \wedge \mathbf{g}) \wedge \mathbf{p}) \end{bmatrix}$$
$$\textit{where } \mathbf{x} = X - P$$
$$\mathbf{p} = Q - P$$
$$\mathbf{g} = G - P$$

where $\cdot$ denotes scalar product, and $\wedge$ denotes vector (cross) product in $\mathbb{R}^3$. This requires only a constant number of arithmetic operations. We can now construct a program which handles the non-degenerate cases, so assuming that no three points are collinear and no four points are coplanar, the output corresponding to edge PQ may be calculated as in figure C.6.

C.3 Coping with Degeneracies

As with many geometric algorithms, a large amount of time is spent checking for and dealing with instances which contain degeneracies. Several general methods exist for solving these problems, for example by perturbing the points by small amounts [Yap88]. In this case however, the effects of degeneracy can be dealt with directly. I treat the following cases:

1. some group of three or more points is collinear
2. some group of four or more points is coplanar

case 1: Suppose edge PQ is being processed, and point X arrives which is collinear with P and Q. If X lies between P and Q it may be ignored completely. Otherwise one of P or Q must be an internal point, and so PQ is not an external edge.

In two dimensions, collinearity requires special special treatment when updating the current wedge (M, A, B). Suppose X is the new point, and M, A and X are collinear. If M lies between A and X then this wedge is not part of the output. If A lies between M and X then the wedge becomes (M, X, B), otherwise no action is taken. A similar analysis holds for B.

case 2: Suppose edge PQ is being processed, and the current wedge projects onto $(\mathbf{0}, \pi A, \pi B)$. Suppose further that the new point X lies in the plane of P, Q and A (but that case 1 does not apply), then $\mathbf{0}$, πA and πX are collinear. If $\mathbf{0}$ lies between πA and πX then the wedge on PQ is not part of the output. Otherwise, replace A by the larger of A and X with respect to the $\prec$-ordering.

```
process_PQ(P, Q, V, G)
vertex P, Q, G;
vertex_set V;
{
    wedge W;
    boolean flag;
    point A, B, X;

    \* initialise the wedge -- O(1) *\
    make_new_wedge((0,0), W);

    \* flag denotes "do not output" *\
    flag := FALSE;

    \* for each vertex in V -- O(n) iterations *\
    while ((NOT flag) AND nonempty(V))
        {
        X := select(V);          \* choose next element  -- O(1) *\
        V := set_remove(V, X);   \* and remove it from V -- O(1) *\
        if ((X <> P) AND (X <> Q)) \* do nothing if X=P or X=Q *\

            \* project X, and update wedge -- O(1) *\
            flag := add_to_wedge(W, project(P, Q, G, X));
        }

    \* decide on output *\
    if (NOT flag)
        {
        \* select components of wedge W = ((0,0), A, B) *\
        A := W.A; B := W.B;

        \* ">" denotes lexicographic ordering on points *\
        if (A>P AND A>Q) output(P,Q,A); \* ensure no duplicates *\
        if (B>P AND B>Q) output(P,Q,B); \* ensure no duplicates *\
        }
}
```

Figure C.6: Non-degenerate case (see text)

Using these strategies, consider what happens when a face of $CH(V)$ is an n-gon. Suppose R is the $\prec$-largest vertex of the face, and P,Q are two adjacent vertices. If $R \neq P$ and $R \neq Q$ then by selecting always the largest point to update the wedge, and since one of the final wedges must lie in the plane of this face, the triangle PQR is produced. This triangle is output because $P, Q \prec R$ (see above). If $P = R$ then the final triangle for this face is PQS for some S in this face. But then $S \prec P$ since ($P = R$ and R is largest on the face), and so PQS is not output. Similarly if $Q = R$. Hence the triangulation obtained for the face is the one given by connecting R to each of the other face vertices.

C.4 Parallel Implementation of 3CH

It is clear that the edge calculations are independent, and can be performed simultaneously. However, for a practical implementation on a distributed memory architecture we must arrange for the correct data to be present in the right places at the right time. The model of parallelism I use is that of communicating processes, with no shared memory. All communication is synchronised, that is, cannot proceed until both parties are ready. This greatly simplifies problems of timing and clocking, which can now happen automatically, although the programmer must take responsibility for avoiding deadlock, and for spotting bottlenecks. The underlying algebra (CSP) is explained in [Hoa85], but the specific language used in the figures is occam [Jon87], whose semantics are based on CSP

3CH consists of two main processing sections with some extra code for transferring data (figure C.7). The first processing section is a parallel implementation of a data structure storing a set of up to n vertices. This structure uses n processes arranged in a pipeline, supporting Insertion ($O(1)$), Deletion ($O(1)$) and duplicate removal (free). Furthermore, a copy of the entire structure can be sent to output at any time ($O(n)$). This can be modelled in occam as a priority queue (figures C.8 and C.9).

The second main part of the algorithm uses $n^2/2$ processes arranged in an $O(n)$ by $O(n)$ rectangular mesh (although most of the horizontal connections are not required). This structure performs the actual convex hull calculation, inputting the set of n points (twice) and outputting the triangular faces of the hull after further time $O(n)$. Each process is allocated one edge calculation, which occurs the first time the data passes. This can be done in linear time. Also during this pass the centroid (G) may be calculated, just by averaging the points which pass, in total time $O(n)$. On the second pass, all the wedges are computed as the data flows through, and at the end of the pass, all the triangles still valid for output

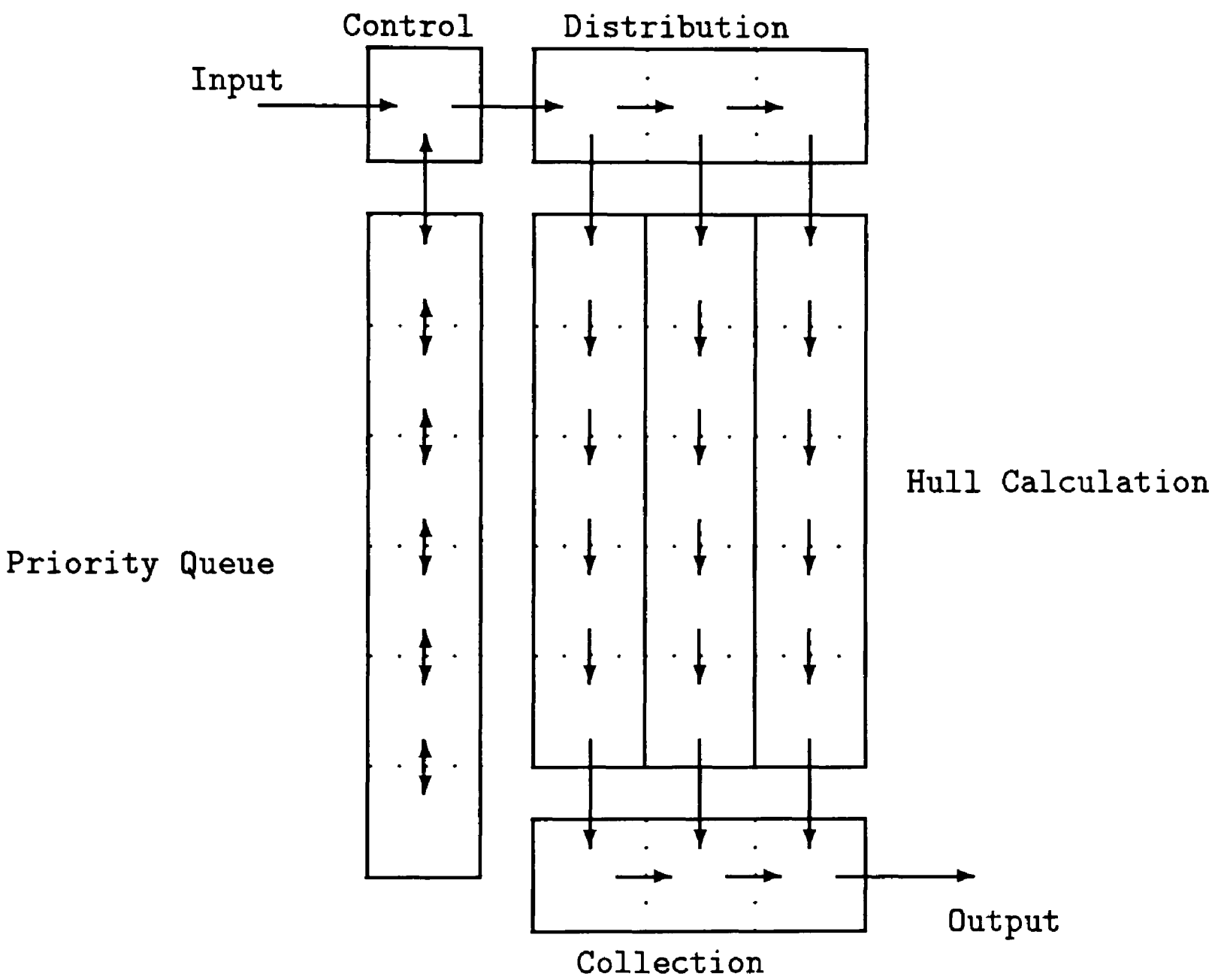


Figure C.7: The architecture of 3CH

```
PROC pq_element(CHAN left.in, left.out, right.in, right.out) =
  INSTRUCTION instr:
  POINT store, x:
  SEQ
    store := NIL -- NIL is defined to be MAXINT
    WHILE TRUE
      SEQ
        left.in ? instr; x
        IF
          (instr = INSERT)
            SEQ
              right.out ! INSERT; larger_point(x, store)
              store := smaller_point(x, store)
          (instr = DELETE) AND (x = store)
            SEQ
              right.out ! SHIFTOUT; ANY
              right.in ? store
          (instr = DELETE) AND (x <> store)
            right.out ! DELETE; x
          (instr = CLEAR)
            SEQ
              store := NIL
              right.out ! CLEAR; ANY
          (instr = SHIFTOUT)
            SEQ
              left.out ! store
              right.out ! SHIFTOUT; ANY
              right.in ? store :
          (instr = COPYOUT) AND (store = NIL)
            left.out ! NIL
          (instr = COPYOUT) AND (store <> NIL)
            SEQ
              x := store
              right.out ! COPYOUT; ANY
              WHILE (x <> NIL)
                SEQ
                  left.out ! x
                  right.in ? x
              left.out ! NIL
:
```

Figure C.8: An interior process in the priority queue implementation of a set. The special value NIL denotes the absence of a real value.

```
PROC pq_terminator(CHAN left.in, left.out) =
  PAR
    WHILE TRUE
      INSTRUCTION instr:
      POINT x:
      left.in ? instr; x
    WHILE TRUE
      left.out ! NIL :
```

Figure C.9: The last process in the priority queue implementation of a set. This process behaves like a permanently empty slot in the structure, and makes coding much simpler for the other processes. For clarity, I have pretended that operations on points in $\mathbb{R}^3$ are atomic in this example.

are passed out to a collection point. Since no stage of the computation takes more than linear time, the overall time is also linear.

The communication behaviour of 3CH is simple to describe using occam (or occam2). The only complicated part is the priority queue of points for processing, since this involves two way communication, and so it is essential to think carefully in order to both prevent deadlock, and keep as much parallelism as possible.

Bibliography

[ACM82] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. Technical Report Technical Report CSD–427, Purdue University, Computer Science Dept., 1982.

[ACM84a] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM Journal of Computing*, 13(4):865–877, November 1984.

[ACM84b] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition II: An adjacency algorithm for the plane. *SIAM Journal of Computing*, 13(4):878–889, November 1984. contains references for part I.

[BH60] J. F. Bennett and W. L. Hays. Multidimensional unfolding: Determining the dimensionality of ranked preference data. *Psychometrika*, 25(1):27–43, 1960.

[Blu70] L. M. Blumenthal. *Theory and Applications of Distance Geometry.* Chelsea, New York, 1970.

[Bro87] D. Brownawell. Bounds for the degrees in the nullstellensatz. *Annals of Mathematics, second series*, 26(3):577–591, 1987.

[BT71] W. S. Brown and J. F. Traub. On euclid's algorithm and the theory of subresultants. *Journal of the A.C.M.*, 18(4):505–514, October 1971.

[Buc70] B. Buchberger. An algorithmical criterion for the solvability of algebraic systems of equations. *Aequationes Mathematicae*, 4(3):374–383, 1970. (in German).

[Buc76a] B. Buchberger. Some properties of Gröbner bases for polynomial ideals. *ACM SIGSAM Bull.*, 10(4):19–24, 1976.

[Buc76b] B. Buchberger. A theoretical basis for the reduction of polynomials to normal form. *ACM SIGSAM Bull.*, 10(3):19–29, 1976.

[Buc85] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Co., 1985.

[Buc87] B. Buchberger. Applications of Gröbner bases in non-linear computational geometry. Technical Report 87-15.0, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1987.

[CA80] J. D. Carroll and P. Arabie. Multidimensional scaling. *Annual Review of Psychology*, 31:607–649, 1980.

[CFG+] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, and M. B. Monagan. *MAPLE Reference Manual.* Symbolic Computation Group, Dept. of computer science, University of Waterloo, Ontario, Canada.

[CFG+86] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, and M. B. Monagan. A tutorial introduction to MAPLE. *Journal of Symbolic Computation*, 2(2):197–200, 1986.

[CGH88] L. Caniglia, A. Galligo, and J. Heintz. Some new effectivity bounds in computational geometry. Technical report, Institutio Argentino de Mathematica, 1988. preliminary report.

[Cha84] B. Chazelle. Computational geometry on a systolic chip. *IEEE Trans. on Computers*, c-33(9):774–785, September 1984.

[CL82] G. E. Collins and R. Loos. Real zeros of polynomials. *Computing*, Supplement 4:83–94, 1982.

[Col75] G. E. Collins. *Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975.

[Coo58] C. H. Coombs. An application of a nonmetric model for multidimensional analysis of similarities. *Psychological Reports*, 4:511–518, 1958.

[Cri81] G. Crippen. Distance geometry and conformational calculations. In D. Bawden, editor, *Chemometrics Research Studies Series, Volume 1.* Research Studies Press, Wiley, 1981.

[Eri89] L. W. Ericson. *Planar Geometric Editing.* PhD thesis, New York University, 1989.

[EY88] L. W. Ericson and C. K. Yap. The design of LineTool, a geometric editor. *4th Computational Geometry Conference*, 1988.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability a guide to the theory of NP-completeness.* W.H. Freeman, San Francisco, 1979.

[Har88] R. I. Hartley. Drawing polygons given angle sequences. *Information Processing Letters*, 31:31–33, 1988.

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

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[Jon87] G. Jones. *Programming in occam.* Prentice-Hall International, London, 1987.

[JS85] R. Joobbani and D. P. Siewiorek. WEAVER: A knowledge-based routing expert. *proc. IEEE 22nd Design Automation Conference*, 1985.

[Ken71] D. G. Kendall. Construction of maps from 'odd bits of information'. *Nature*, 231:158–159, 1971.

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Kru64a] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit. *Psychometrika*, 29:1–27, 1964.

[Kru64b] J. B. Kruskal. Nonmetric multidimensional scaling: A numerical method. *Psychometrika*, 29:115–129, 1964.

[Lak76] I. Lakatos. *Proofs and Refutations: The logic of mathematical discovery.* C.U.P., Cambridge, 1976.

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

[Loo82] R. Loos. Computing in algebraic extensions. *Computing*, Supplement 4:173–187, 1982.

[LY82] L. Lovász and Y. Yemini. On generic rigidity in the plane. *SIAM Journal on Algebra and Discrete Methods*, 3(1):91–98, 1982.

[McC88] S. McCallum. An improved projection operation for cylindrical algebraic decomposition of three-dimensional space. *Journal of Symbolic Computation*, 5:141–161, 1988.

[Mil59] R. E. Miles. The complete amalgamation into blocks, by weighted means, of a finite set of real numbers. *Biometrika*, 46:317–327, 1959.

[Nel85] G. Nelson. Juno, a constraint-based graphics system. *ACM SIGGRAPH*, pages 22–26, 1985.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[SBO81] R. Sibson, A. Bowyer, and C. Osmond. Studies in the robustness of multidimensional scaling: Euclidean models and simulation studies. *Journal of Statist. Comput. Simul.*, 13:273–296, 1981.

[Sch35] I. J. Schoenberg. Remarks to Maurice Fréchet's article "sur la définition axiomatique d'une classe d'espaces vectoriels distanciés applicables vectoriellement sur l'espace de Hilbert". *Annals of Mathematics*, 36:724–32, 1935.

[Sed88] R. Sedgewick. *Algorithms.* Addison Wesley, 1988. Second edition.

[She62a] R. N. Shepard. The analysis of proximities: Multidimensional scaling with an unknown distance function I. *Psychometrika*, 27(2):125–140, 1962.

[She62b] R. N. Shepard. The analysis of proximities: Multidimensional scaling with an unknown distance function II. *Psychometrika*, 27(3):219–246, 1962.

[Sib72] R. Sibson. Order invariant methods for data analysis. *Journal of the Royal Statistical Society, series B*, 34:311–349, 1972.

[Ste73] I. Stewart. *Galois Theory.* Chapman and Hall, New York, 1973.

[Str80] G. Strang. *Linear Algebra and its Applications.* Academic Press, New York, 1980. Second edition.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tor52] W. S. Torgerson. Multidimensional scaling: I theory and method. *Psychometrika*, 17(4):401–419, 1952.

[VdW53] B. L. Van der Waerden. *Modern Algebra.* Frederick Ungar, New York, 1953.

[Yap88] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Proc A.C.M. symp on Computational Geometry*, June 1988.