Algebraic Constraints

James Gosling

May, 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Algebraic Constraints

James Gosling

Department of Computer Science
Carngeie-Mellon University
Pittsburgh, PA. 15213

May, 1983

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Algebraic Constraints

James Gosling

Abstract

Constraints are a way of expressing relationships among objects; satisfying a set of constraints involves finding an assignment of values to variables that is consistent with the constraints. In its full generality, constructing a constraint satisfaction algorithm is a hopeless task. This dissertation focuses on the problem of performing *constraint satisfaction* in an interactive graphical layout system. It takes a pragmatic approach and restricts itself to a narrow but very useful domain. The algorithms used by MAGRITTE, an editor for simple line drawings, are presented. A major portion of the work concerns the algebraic transformation of sets of constraints. It describes algorithms for identifying difficult subregions of a constraint graph and replacing them with a transformed and simplified new constraint.

Acknowledgements

My advisors Bob Sproull and Raj Reddy deserve many thanks for their guidance and support during my years as a graduate student and for their assistance in making this dissertation coherent. I would also like to thank the other members of my thesis committee, Guy Steele and Ivan Sutherland, for their insightful comments and patience. Marc Donner read the initial drafts of the thesis and provided valuable critical feedback.

I'll not name the people who were the most valuable to me. Writing their names out in a list, as is the custom, somehow trivialises their influence. They are all the incredible, crazy, and wonderful people of the Computer Science Department at Carnegie-Mellon University. My many close friends there kept me sane, kept me technically honest, and made the whole time very enjoyable. I cannot thank them enough.

Table of Contents

List of Figures

I Introduction

Constraints are a way of expressing relationships among objects; satisfying a set of constraints involves finding an assignment of values to variables that is consistent with the constraints. In its full generality, constructing a constraint satisfaction algorithm is a hopeless task. This dissertation focuses on the problem of performing *constraint satisfaction* in an interactive graphical layout system. It takes a pragmatic approach and restricts itself to a narrow but very useful domain. The algorithms used by MAGRITTE, an editor for simple line drawings, are presented. A major portion of the work concerns the algebraic transformation of sets of constraints.

One problem with many existing graphical layout systems is that they have no general notion of the semantics of objects. An object such as a rectangle has certain invariant properties. It has four sides, each being connected at its endpoints to the two adjacent sides of the rectangle, each pair of connected sides meets at a 90 degree angle, and the size of the rectangle may be fixed. If you grab a corner of the rectangle, the rest of the rectangle should follow along, to preserve the *rectangleness* of the object. If the size of the rectangle isn't fixed, perhaps it should stretch — the system has a certain amount of freedom in handling situations that are incompletely constrained. In typical layout systems, it is not possible to specify such things. When you grab a point or line, that point or line moves. You cannot connect things, you cannot define new types of objects with new behaviours. Instead, the person using the system must understand the semantics of the object and ensure manually that they are preserved. Several systems have extensive built-in semantic knowledge, but they lack general methods for extending it.

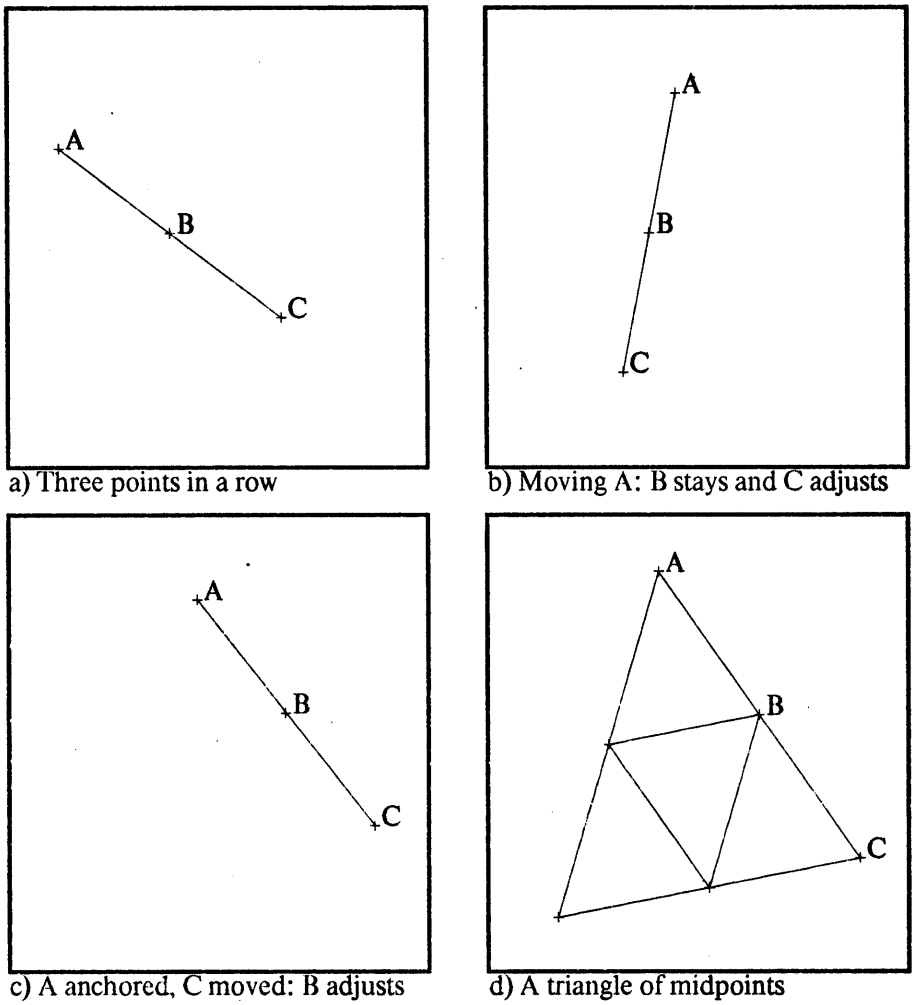
This lack of understanding provided the motivation for developing MAGRITTE. It is capable of accepting constraints like 'this line should be

horizontal' or 'this point should be midway between those two points'. It will then endeavour to insure that the truth of these constraints is maintained: when some change is made, like moving a point, MAGRITTE will move other points to satisfy the constraints that have been violated. The generality of the mechanism allows constraints to be arbitrary equations in conventional algebra.

Constraint systems are capable of dealing with more than just geometric properties. For example, the electrical properties of a circuit or the structural properties of a bridge can be described.

When MAGRITTE is started it presents the user with a blank canvas to sketch on. If the user lays down 3 points, A, B and C, connects them with lines and asserts that $A+T=B$ and $B+T=C$ then figure I-1a results. If A is moved, then it uses the two equations in an obvious way and produces figure I-1b. If, however, MAGRITTE were told that A could not be moved, and the user told it to move C, then the two equations can not be used as they are to deduce a new value of B. MAGRITTE employs some simple algebra and derives and uses $B=0.5*(A+C)$, resulting in figure I-1c. The construction can be continued to create a triangle, as in the figure I-1d. The inner triangle, whose vertices are the midpoints of the outer triangles sides, is similar to the outer triangle. Normally this fact does not concern MAGRITTE, since most changes made to this shape don't require knowing any more than what the midpoint constraints tell it directly. However, if two of the midpoints are fixed by the user and the other midpoint is moved, then MAGRITTE discovers that it needs to do some algebra, and does so.

In a similar fashion, drawings can be constructed that contain levers, pulleys, rocker arms, and cables. They can be strung together such that when a lever is moved, the mechanism that was drawn follows along. Figure I-2 contains the triangle of midpoints from the previous example along with two levers and rigid bars. One lever is moved and everything else follows along.



a) Three points in a row
b) Moving A: B stays and C adjusts
c) A anchored, C moved: B adjusts
d) A triangle of midpoints

Figure I-1: Starting a session with MAGRITTE

I.1. Constraints

A constraint is a relation among several objects stating what *should* be true. A set of constraints forms a network relating a larger set of objects. A constraint system has an associated *satisfaction* mechanism that attempts to adjust the related objects so that the relationships hold. For example, we may have two points that are not horizontally aligned and a constraint between them that insists they be horizontal. A constraint system should be capable of resolving this inconsistency by moving one of the points appropriately.

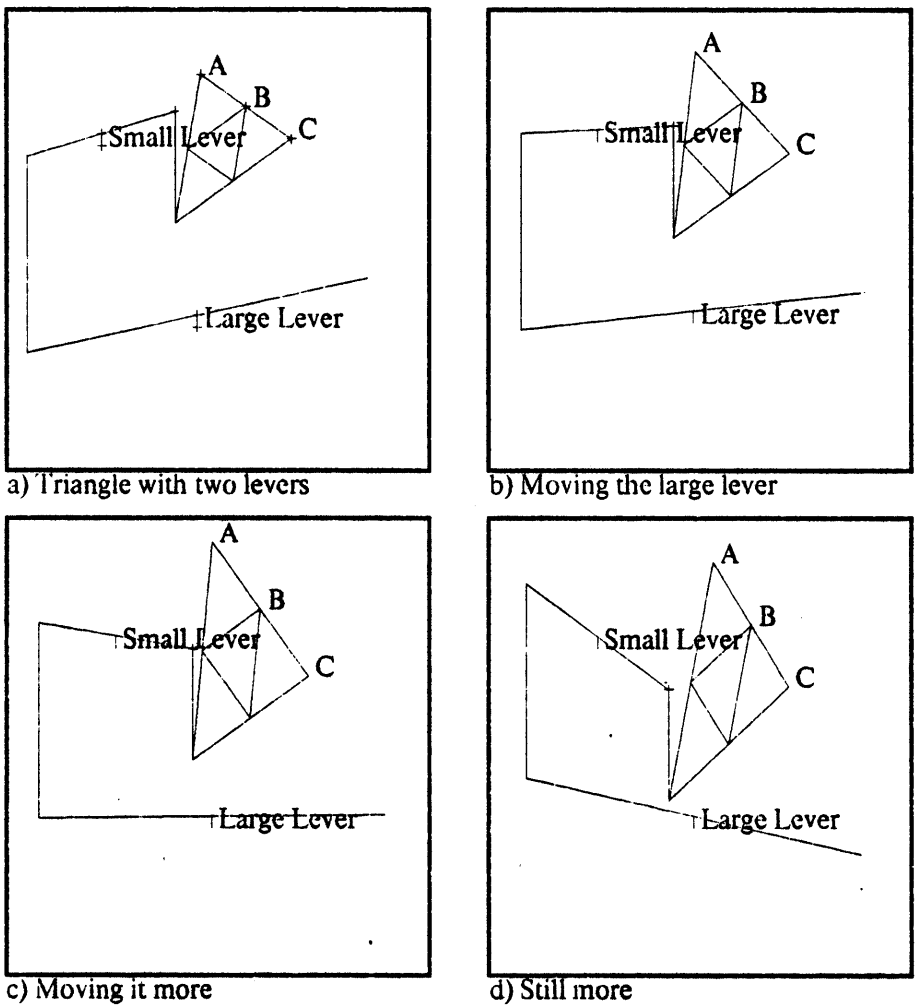


Figure I-2: Levers and cables.

The network formed by a set of constraints and their constrained objects may be visualised in several ways. One of these is as a set of devices that behave like components in an electrical circuit; the values of the system are the wires that connect them.

The boxes in figure I-3 are the constraints. They can be thought of as little devices that get wired together like transistors or resistors. The box labelled "+" constrains the value on the lead at the rounded end to be the sum of the values on the other leads. Similarly, the boxes labelled "*" are product constraints. This network constrains C to be the Celsius equivalent of the Fahrenheit temperature F.

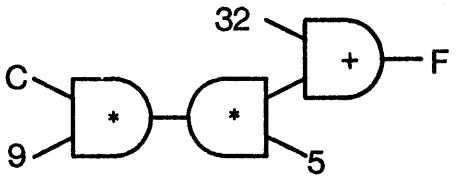


Figure I-3: Celsius-Fahrenheit temperature conversion

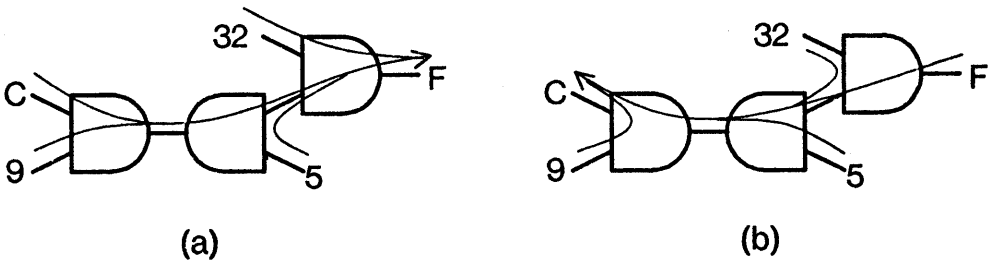


Figure I-4: Data flow in a constraint network

Unlike the circuit analogy, there is *no* directionality in the components of a constraint net, even though it is convenient to think of information as flowing through them. Each component is capable of deriving a new value in any direction. Figure I-4 shows the flow of information for calculating both Celsius from Fahrenheit and Fahrenheit from Celsius values.

A dual visualisation exists that is often more suggestive than the circuit visualisation. This dual view is as something akin to a network data base where the values of the system are the important things, mentally drawn as boxes, connected by lines that represent the constraints. The constraints can be thought of as statements of relationships between the nodes of the data base. Not facts, but compulsions.

Figure I-5 shows the representation of a rectangle as described in this way. It has four points that are related: two pairs are horizontal and two pairs are vertical. Two orthogonal sides have lengths specified; it is not necessary to specify the lengths of the other sides since they are easily derived from the other constraints. There are four line objects that have the con-

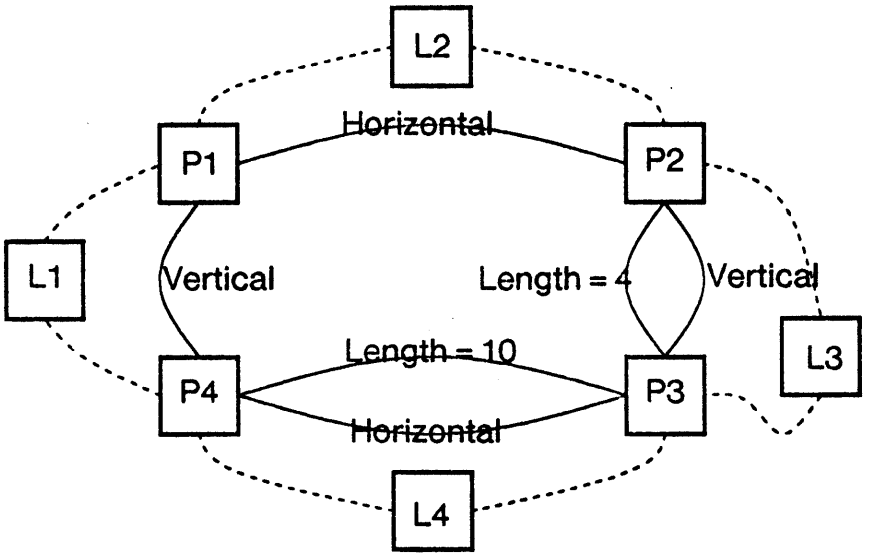


Figure I-5: A constraint network representing a rectangle

strained points as parts. Actually, the points could be broken down further into x and y coordinates, and the constraints could be broken down to constraints on these more primitive objects — this is what is actually done in MAGRITTE.

Yet another way to think of a constraint network is as a system of equations. This is a natural visualisation in MAGRITTE since it restricts constraints to be equations in conventional algebra. For example, constraining two points to be horizontally aligned is done by equating their y coordinates. This restricts the scope of what can be expressed, but there are tremendous advantages to having the deductive machinery that comes along with it.

I.2. Related Work

Constraint systems have a fairly long history within computer science. In this section I will briefly sketch the work of others. Several of these systems will be analysed in more detail later.

Ivan Sutherland's SKETCHPAD [18] was a novel drawing system that allowed the definition of arbitrary objects and constraints. It pioneered the use of interactive computer graphics and of constraint systems. Considering its age — it was written in 1962 — and the obvious benefits of some of its novelties, it is amazing that few general design systems have been built since then that have more powerful interaction mechanisms. It was capable of, for example, being told what a bridge looked like, along with all the structural properties of its components. A load could be placed on the bridge and the bridge would deform to satisfy all the constraints on it, such as the pull of gravity and the elasticity of the components. Its satisfaction algorithm is discussed in some detail in section III.2.2 on page 38.

Alan Borning's THINGLAB system [1] carried on where SKETCHPAD left off. It was a generalized simulation laboratory based on constraints. Users sketched a design and told THINGLAB what the parts were and how they behaved, and THINGLAB performed a simulation. It was capable of performing the bridge simulation as SKETCHPAD did, as well as circuit and physical simulations. It had a powerful class system, based in part on SIMULA and SMALLTALK, for representing hierarchically constructed designs and for composing new data types from old ones. New classes could be defined either graphically or, as in more conventional systems, from a piece of text. It was far more than a simulation system. More detail appears in section III.2.3 on page 40.

Steele and Sussman [17] present a language for the construction of hierarchical constraint networks and a simple solution technique that they call *local propagation*. They also briefly discuss algebraic manipulations of constraint networks.

Steele's thesis [16] is an examination of methods of implementing constraint systems — of representing the network, of finding values that satisfy it, and of asking questions about the state of the net. This system had a capacity for explaining itself to the user, of describing why it decided to do certain things.

Van Wyk built a system, IDEAL [20], for textually expressing the layout of line drawings. It used a constraint system that was limited to systems of linear equations. Objects in this system contained subparts that were in turn other objects. Simple equations could be written that related the coordinates of points in the objects; these were not assignment statements or procedural calculations, but equations that were combined into a constraint network. It was designed for a batch-like environment and had no capacity for interaction.

The VLSI layout system *i* [9], built by Steve Johnson at Bell Labs, is in many ways similar to Van Wyk's system. It allows the definition of *cells* that can be hierarchically composed. Van Wyk's system had a similar notion, called a *box*. Unlike Van Wyk's boxes and THINGLAB's classes, the constraints in a cell are satisfied once, independent of any invocation. All invocations use the results of the one satisfaction process, with some simple geometric transformation being applied. This has considerable performance advantages, but flexibility is lost. Neither Van Wyk's system nor *i* had a capacity for defining new data types with new operations and constraints.

EARL [11] is another VLSI design system that was built by Chris Kingsley. It bears a strong resemblance to *i*. An important difference between the two is that while *i* fixes the layout of subcells so that successive invocations of the same cell produce the same result, EARL does not. It satisfies the constraints for each invocation seperately. Its constraint satsifaction algorithm is described in section III.1.7 (page 34).

MUMBLE [8] is a compiler for microcoded machines that I wrote. It uses a simple constraint-based address assignment algorithm. Details appear in section III.1.6, page 29. The present thesis was in part motivated by this work.

I.3. Scope of the Dissertation

The bulk of the dissertation is concerned with constraint satisfaction algorithms. It is my intent to demonstrate that constraints can be used efficiently in practical graphical layout systems and to show how it can be done. Constraints will be discussed in more general terms but the special properties of such applications will be used extensively.

This dissertation follows in roughly the path of SKETCHPAD and THINGLAB. It presents, among others, the algorithms used in MAGRITTE, an editor for simple line drawings. Linguistic and user interface issues are explicitly ignored, although there is some cursory discussion. Other theses have dealt extensively with these. A progression of algorithms is presented and discussed, some original, some not. Of particular interest are the techniques for transforming networks of constraints into forms that are tractable using simple satisfaction algorithms. Constraints are generally restricted to equations in conventional algebra.

The guiding principle used to judge the various algorithms is that they behave in a *local* fashion. The amount of work necessary to satisfy a net after some change has been made should be related to the size of the region affected by the change, not to the size of the net. The pursuit of locality has a pervasive influence on the construction of the algorithms and they should be viewed with this in mind.

The dissertation is broken into two parts:

Satisfaction: Given a constraint network, find values for the constrained variables that satisfy the constraints. A progression of algorithms is presented, each with its own faults. An important point is that there is no known, or even possible, universally good satisfaction algorithm. The special properties of the situation at hand must be used when constructing an algorithm.

Transformation: To deal with some of the problems encountered by local satisfaction algorithms, techniques for transforming parts of a net into more tractable forms are given. Interrelating transformation and satisfaction, detecting and identifying problems, is also discussed.

The net result is a set of algorithms and principles that can be used when constructing constraint-based graphical layout systems.

Several important advances are unique to this dissertation:

- The use of breadth-first search to plan propagation.
- The use of transformations to break loops, improve performance and provide a more elegant notation.
- The automatic invocation of transformation from satisfaction.
- And the use of a fast graph isomorphism algorithm to make frequent application of transformation feasible.

II Representation

The chapter covers the representation of constraint networks. It describes briefly the representations used by a few other systems and goes into the MAGRITTE system in some detail. The representation chosen does not affect the performance of the system nearly so much as it affects its elegance.

The usual representation chosen is that of a linked network of structures that directly parallels one of the visualisations of constraint networks presented in the introduction. In MAGRITTE, values and constraints are distinct structures with cross references to each other. Since it is built on top of Franz Lisp [6], it uses the facilities available there. These were insufficient so a structure package had to be constructed.

II.1. Type System

Simula's type system [2] was used as the model for MAGRITTE: a data type has named fields and may be a member of a single superclass. Both Steele's system, being written in Lisp Machine LISP [21], and THINGLAB, being written in SMALLTALK [7], are similar. THINGLAB added multiple superclasses to SMALLTALK.

One of the important distinguishing features of structure systems is the mechanism used to invoke operations on objects. In SIMULA, procedures behave as fields of structures. They are invoked by presenting an object and naming a procedure within it. In THINGLAB an object is an active entity that receives messages. Messages are roughly parallel to procedures. In both of these, selection of an operation is based on the type of a single operand. The major novelty in the type system used by MAGRITTE is that it allows the selection of an operator to be based on the types of all the operands, much like operator identification in ADA [5],

except that it is done dynamically, at runtime. For example, the following defines addition for all combinations of integer and real operands:

```
(defmethod plus ((a real) (b real))
                . . .)

(defmethod plus ((a integer) (b integer))
                . . .)

(defmethod plus ((a integer) (b real))
        (plus (float a) b))

(defmethod plus ((a real) (b integer))
        (plus a (float b)))
```

The flexibility and power of this type system made major contributions towards easing the implementation effort and simplifying the structure of MAGRITTE.

II.2. Objects

The constraint system in MAGRITTE, not just the type system, considers the basic object to be a simple numeric scalar. The constraint system, in fact, really understands nothing else. More complicated types are constructed from scalars by wrapping them in structures. Constraints are a separate type, distinct from constrainable objects. A simple constrainable object will often be referred to as a *cell.*

The relationship between constraints and objects varies from system to system. In THINGLAB constraints are actually parts of data objects. For example, if you wanted to constrain two points to be horizontally aligned, it would be necessary to have a data object for a horizontally-aligned-point-pair that would contain as components two points and the specification of the constraint between them. One would take this object and equate or merge its two component points with the two points that are to be horizontal. Steele's system is similar to this: one creates constraint instances and then binds external objects to 'pins' of the constraint. MAGRITTE takes the slightly more conventional approach of binding external objects to the 'pins' of a constraint at the same time that the constraint is instantiated. Data objects are completely external to the constraints acting on them. SKETCHPAD is a cross between these two: one creates constraints that have 'pins' and connects constraints by 'merging' their pins.

Constraints may be treated as full-fledged constrainable objects. They can be thought of as boolean predicates. Similarly, a constraint system can be thought of as a conjunction of these predicates. Allowing constraints to be combined in boolean equations would allow one to express such things as $a=b \vee a=c$. To limit the scope of this thesis, this issue has been avoided in MAGRITTE by not allowing constraints to be constrained.

II.3. Primitive Constraints

A *primitive* constraint is one that the satisfaction algorithm understands. Its definition contains all the necessary information for telling whether or not the constraint is satisfied and for doing the computations necessary to satisfy it if it isn't. The satisfaction process is discussed in depth in Chapter III.

A *constrainable object* contains a value and the set of *constraint instances* that apply to it. A constraint instance contains a reference to a *constraint definition* and a set of constrainable objects that are bound to the parameters of the constraint. A constraint definition contains a formal parameter list, a *rule* that evaluates to true or false depending on whether or not the constraint is satisfied, and a set of *clauses* that can be used to compute a new value for any of the parameters given the values of the others.

For example, the definition of the **sum** constraint that is used in MAGRITTE appears in figure II-1. It contains:

- A parameter list. In this case, they are *result*, *a*, and *b*, which are all scalars.
- A *rule* that is a boolean expression that evaluates to true or false, depending on whether or not the constraint is satisfied. For the sum constraint, the *rule* is $result=a+b$.
- A set of *clauses* that reevaluate the constrained values from the others. For example:

  ```
  ((result (a b)) (setf result (+ a b)))
  ```

  should be interpreted as saying that "in order to calculate *result* given *a* and *b*, assign it the value $a+b$. Similarly, each of the other clauses contain code to propagate a value in some direction, all possible propagation paths being enumerated.

The form taken by the clauses depends on the satisfaction mechanism used. The form used in this example follows THINGLAB and Steele's sys-

```
(defprimc sum ((result scalar) (a scalar) (b scalar))

    (equal result (+ a b))

    ((result (a b)) (setf result (+ a b)))
    ((a (result b)) (setf a (- result b)))
    ((b (result a)) (setf b (- result a)))
)
```

Figure II-1: The primitive **sum** constraint

tem. The clauses enumerate each subset of the constrained values from which other values can be deduced. It is the one used in an early version of MAGRITTE. More discussion of this appears in the section on transformations (Section IV, page 53).

SKETCHPAD defined constraints solely in terms of an error function. Given an assignment of values to variables, this function returned a measure of the violation of the constraint. To satisfy the constraint by changing one variable SKETCHPAD would numerically differentiate the error function and search for a zero crossing.

In MAGRITTE the only primitive constraints are sum, product, equals and less-than. All others are derived by composition and transformation. The transformation techniques given in chapter IV rely on there being only this small primitive set.

II.4. Compound, or Macro, Constraints

Constraints on non-scalar objects, as well as many constraints on scalars, can be expressed in terms of primitive constraints on scalars. This is implemented as something akin to macro expansion.

Consider the compound type *point* that contains two fields: *p-x* and *p-y*, the *x* and *y* coordinates of the point. To constrain two points *p1* and *p2* to be horizontal, the following suffices:

```
(defmacc horizontal ((p1 point) (p2 point))
    (constrain equal (p-y p1) (p-y p2))
```

The overloading of operator identification can be used to define the addition of two points:

```
(defmacc sum ((result point) (a point) (b point))
    (constrain sum (p-x result) (p-x a) (p-x b))
    (constrain sum (p-y result) (p-y a) (p-y b))
)
```

An even more interesting effect can be obtained by using the meta-type *arb* that represents an arbitrary type. A midpoint constraint that constrains one thing to be midway between two others can be defined like this:

```
(defmacc midpoint ((a arb) (b arb) (c arb))
    (local (temp (typeof a)))
    (constrain sum b a temp)
    (constrain sum c b temp)
)
```

This definition works on any type for which *sum* is defined.

III Satisfaction

At the heart of any constraint system, at least for the purposes of this thesis, is a *satisfaction* mechanism. Its task is to find a set of values that satisfy all of the specified constraints. The construction of a general satisfier is a very hard problem, even if the constraints are restricted to simple arithmetic operations. For example, boolean satisfiability can be cast in this framework and is NP-complete, so general satisfaction is at least as hard.

However, all is not lost. Satisfaction algorithms can be constructed that work often enough to be very useful. The structure and utility of these algorithms is highly dependent on the application context in which they exist. For example, several VLSI design aids, such as *i* [9] and EARL [11], have within them very simple constraint systems that achieve powerful results.

The algorithms are also influenced by the manner in which they will be used. Batch-style satisfiers can use more global and time-consuming techniques while interactive satisfiers have much tighter performance requirements. EARL [11] is a batch satisfier that performs an operation akin to a transitive closure on its constraint graph. THINGLAB, on the other hand, is interactive. It contains a complicated mechanism to improve performance by compiling satisfaction strategies.

In this chapter a progression of satisfaction algorithms is presented. Each is described along with an analysis of its effectiveness, efficiency and weaknesses. The progression is intended to justify and provide a basis for later algorithms. Some of the problems that they have are addressed in the following chapter on transformation.

III.1. One-shot Satisfaction

First consider the problem of solving a constraint system where the variables in the net are initially partitioned into a set of known and a set of unknown values. Such situations arise in batch-style satisfiers where the input consists of a set of values, constraints, and constants. Any assignment of values that satisfies the constraints is, by definition, acceptable.

III.1.1. Propagation

There is a very simple algorithm for solving such nets that has appeared in one form or another in many constraint systems that have been built. This technique is so simple and fast that it should always be tried before resorting to more general techniques. In some application areas it always works, but even small problems in the structure of the net can render it useless. Chapter IV of this thesis is devoted to a technique for expanding its usefulness.

In one sentence the algorithm is 'If you can deduce something immediately, then do so'. Viewing the constraint network as a system of equations, this corresponds to ordering the equations so as to solve them by back substitution. Algebraic manipulations may be used within individual equations, but never between equations, except to substitute actual numerical values.

In the circuit view, this amounts to firing any device that has enough known pins to allow the computation of the values on the other pins. Devices continue firing until either all values have been deduced, an inconsistency has been encountered, or no more deductions can be made. The clauses that appear in a primitive constraint definition contain instructions to perform the firing of a device in every possible direction.

When it works, propagation can be applied successfully in many domains. So long as the non-directional constraint definitions can be made, the satisfaction algorithm needs no knowledge of the nature of the constraints. Success or failure is determined by the structure of the graph. When propagation fails, more powerful methods must be applied. Unfortunately, these all tend to be very domain dependent and are often quite slow. Their power comes from their knowledge of the behaviour of the constraints. A common technique, *relaxation*, is a well known numerical approximation algorithm and is discussed in III.1.4 on page 25. Many other classical techiniques exist that can be used for constraint satisfaction.

Among them are Gaussian elimination and linear programming. Both of these, however, place even more strict restrictions on the nature of the constraints that can be handled.

III.1.2. A Simple Example

Consider the following example. It connects two sum constraints to correspond to the equations:

$C = B + T$
$B = A + T$

This simply constrains B to be the average of A and C. It is very simple, but it contains some twists that complicate the satisfaction process. I will use this example extensively in evaluating other techniques.

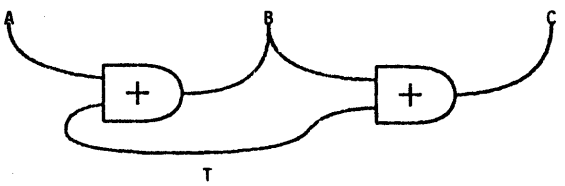


Figure III-1: Midpoint constraint

Suppose A and B are known. Then T can be deduced directly using the left constraint to evaluate $B - A$. From there, C is simply evaluated using the right constraint by adding T and B.

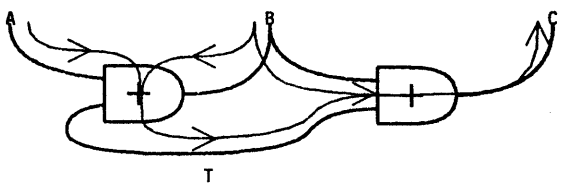


Figure III-2: Successful Propagation

Now suppose A and C are known but B and T are not. Nothing can be deduced from either constraint since neither has enough known values on

its pins. Given a sum constraint, the value on any pin can be deduced from the values on the other two.

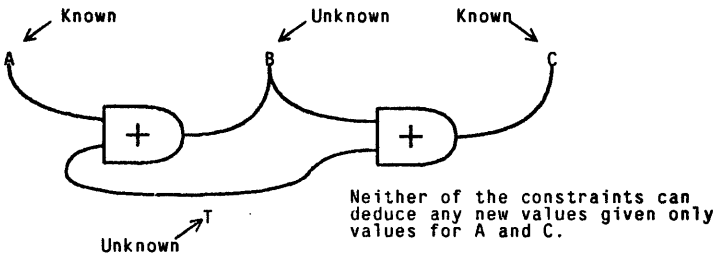


Figure III-3: Unsuccessful Propagation

Suppose that $A=1$ and $C=3$, then the equations for this net are:

$$3=B+T$$
$$B=1+T$$

It is clear that from neither of these equations can a numerical value for either B or T be deduced. However, if one is able to perfom simple algebraic manipulations the first equation can be solved for T, deriving $T=3-B$, which can be substituted in the second and solved for B, deriving $B=1+(3-B)$, $B=4/2$. The final evaluation of B as 2 requires performing a division, as would evaluating T. The propagation algorithm has no way to do this division. All that it has at its disposal are rules provided by the primitive constraints and it must find a sequence of rule applications that satisfy the system. The problem arises because of the circular nature of the graph. All values on the outside of the cycle are known, but an algebraic transformation is necessary to break the cycle. Almost all of the complexity of constraint satisfaction is a consequence of the presence of cycles.

Any satisfaction algorithm that is going to solve this net will have to look at it globally. SKETCHPAD and THINGLAB both use relaxation to cope with situations like this.

III.1.3. The Propagation Algorithm

Here is the propagation algorithm spelt out in detail. It is neither difficult nor new, but is presented here to provide a foundation for what follows.

- A global queue is maintained called *SuspectConstraints* that contains all constraints of interest to the satisfier. It is initially empty.
- Whenever a constrained cell is given a value all adjacent constraints are added to *SuspectConstraints* unless they are already there. This includes cells assigned by the satisfier.
- When a constraint is created it is added to *SuspectConstraints.*

The preceding points result in *SuspectConstraints* containing all those constraints that have a chance of firing. When the time comes to satisfy the system, iterate:

- Pick a constraint from *SuspectConstraints* without removing it.
- Apply the primitive constraint body. One of the following can happen:
    - The constraint is already satisfied.
    - The constraint fires, assigning values to constrained cells, and possibly enqueueing yet more constraints.
    - The constraint can't fire because not enough values were known: forget it. If it eventually becomes satisfiable it will be because an adjacent cell receives a value that will requeue the constraint.
    - The constraint can't fire because all values were known, but they disagree with the rule for the constraint. The network is inconsistent.
- Remove the constraint from *SuspectConstraints.* It is removed now, rather than when it was picked to avoid requeuing it when it fires: changing a value queues all adjacent constraints, but the one that changed it doesn't need to be requeued.
- Loop back to the beginning unless *SuspectConstraints* is empty.
- If, after *SuspectConstraints* becomes empty, cells still remain without values, then the system cannot be satisfied using local propagation.

The actual operation of the algorithm is fairly subtle. It bears a strong resemblance to Sutherland's ordering, see section III.2.2, page 38, and to the topological sort discussed by Knuth [12]. Except for newly created constraints, *SuspectConstraints* initially contains only those constraints connected to at least one known cell. Thereafter constraints will be enqueued only when an adjacent cell becomes known. The satisfier concentrates its attention where truths are known and extends them, completely ignoring other regions until there is some reason to believe that something might be deduced there. One might try to avoid firing constraint functions until enough values are known, but the bookkeeping involved can be more expensive than if the constraint functions are implemented well.

The definition of a primitive 'sum' constraint is given on page 16. A straightforward compiler for such constraints might generate the following Lisp code, as an early version of MAGRITTE did:

```
(defun sum-apply (result a b)
    (If (and (cell-boundp result)
             (cell-boundp a)
             (cell-boundp b)) then
        (If (equal result (+ a b)) then
            'AllIsWell
         else
            'Conflict)
     elseif (and (cell-boundp a)
                 (cell-boundp b)
                 (null (cell-boundp result))) then
        (setf result (+ a b))
        'OK
     elseif (and (cell-boundp result)
                 (cell-boundp b)
                 (null (cell-boundp a))) then
        (setf a (- result b))
        'OK
     elseif (and (cell-boundp result)
                 (cell-boundp a)
                 (null (cell-boundp b))) then
        (setf b (- result a))
        'OK))
```

There is a lot of redundancy in the testing of various booleans. The sequence of **if**'s can be reworked into a binary tree. The following compiled form might result:

```
(defun sum-apply (result a b)
    (If (cell-boundp result) then
        (If (cell-boundp a) then
            (If (cell-boundp b) then
                (If (equal result (+ a b)) then
                    'AllIsWell
                 else
                    'Conflict)
             else
                (setf b (- result a))
                'OK)
         else
            (If (cell-boundp b) then
                (setf a (- result b))
                'OK))
     else
        (If (and (cell-boundp a)
                 (cell-boundp b)) then
            (setf result (+ a b))
            'OK)))
```

This is exactly the sort of detailed fiddling that one can expect a compiler to do. Going even farther, if a compiler knew a little algebra it could generate the previous form given nothing more than $result = a + b$. This is precisely what chapter IV describes.

III.1.4. Relaxation

Relaxation is a classical numerical approximation technique for iteratively finding solutions to systems of equations. It is extremely general since all that it needs to do with each equation is compute an estimate of the error induced by some particular assignment of values to variables. The sum of these error values gives an error estimate for the entire system. Relaxation functions by minimizing this global error value. It does this by perturbing the values assigned to the variables and watching the change in the error.

One form of relaxation, as used in SKETCHPAD [18], presumes that errors in constrained values can be approximated by some linear function. To form a new estimate of the value of some variable, the derivative of each error function of the attached constraints is approximated and a linear function is obtained for each. Then a least squares fit is used to find a new value of the variable that minimizes the errors. This process is repeated for each variable until a solution is reached. The criteria for determining when a solution is reached are based on looking at the rate of change of the new values.

Whether or not relaxation converges, and the rate of convergence, are very sensitive to the choice of initial values, error functions and connectivity. It converges in a very wide range of circumstances, which is why it is so useful. But the convergence is usually slow. Even for well-behaved systems of linear equations the number of iterations is proportional to $m+(-\log_2\lambda)$ where m is the number of bits of accuracy and λ is the eigenvalue of the system with least absolute value [3].

A satisfaction algorithm is said to exhibit *locality* if when a value is perturbed in a satisfied system, the calculation of the effects of that perturbation do not go beyond the parts that are actually affected. Naive implementations of relaxation do not have this property. They always look at all values and all constraints. This can exact a severe performance penalty if the network is large. By trimming and partitioning the graph as in THINGLAB the extent of the application of relaxation can be restricted.

These problems are compounded when the constraint system is imbedded in an interactive program like a drawing editor. With each change of a value, such as the movement of a point, satisfaction must be invoked. There is no way that solutions from previous situations can aid the satisfaction process, except that the values from one situation provide good

guesses for the values in the next situation. They may, however, be very good guesses indeed.

The performance of relaxation can be improved by two techniques that restrict the size of the region examined. The first, used by THINGLAB[1], is to examine the network and remove values that may be calculated as the direct consequence of some unique other value. If you consider the graph of the system of equations, this amounts to trimming off the leaves that are connected to the inner cycles [see figure III-4].

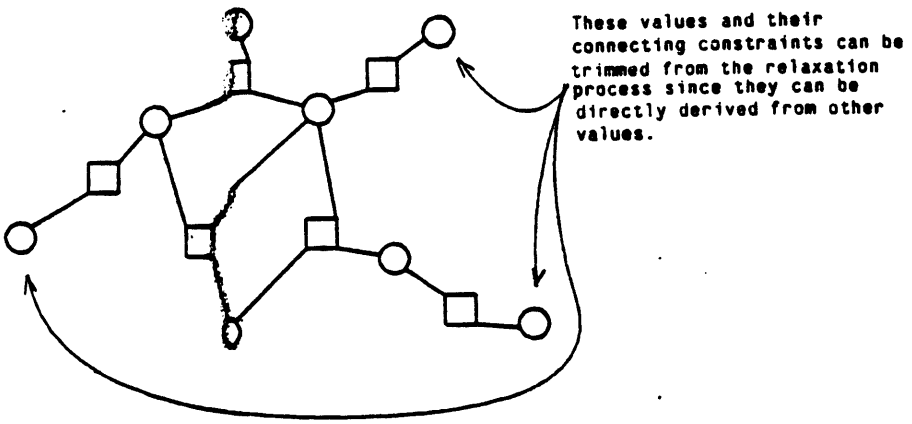


Figure III-4: Trimming leaves from a net to be relaxed

The second technique, which hasn't been used in other systems, involves maintaining a queue of values whose perturbation is likely to reduce the overall error. If you start with a satisfied system, then when the user changes some value, that new value is frozen and the values connected to it by constraints are queued. The 'freezing' is done to avoid having the satisfier change the value back to what it was, undoing the user's change. Remember that the overall error is the sum of the squares of the errors from all of the constraints. The error from a constraint will change only if one of the constrained values changes, hence improvements can be attained by relaxing only values directly related to changed values. This notion of change implying relaxation of neighbours is recursively applied, spreading the relaxed region outward from the original change. Depend-

[1] SKETCHPAD uses it too, but it isn't in the thesis.

ing on the connectivity of the graph and the nature of the induced errors this will limit relaxation to only those values that change and the values adjacent to them. A common example of the necessary use of relaxation in constraint systems is the structural description of a bridge under load, calculating the sag of the bridge and the tension and compression of the members. If the bridge is divided into two spans by a rigid pylon, then a change in the loading on one span will not affect the other span, and with localised relaxation the values there will not even be examined.

While relaxation did work in SKETCHPAD and THINGLAB, using it on anything but trivial examples was not practical. Admittedly, there was a fair penalty in the way that the two systems were implemented, but an improvement in the implementation would yield only a constant factor. The slow rate of convergence remains a large problem.

III.1.5. Alternate Views

In their PhD theses [16, 1] Steele and Borning independently observe that one way to handle the problems introduced by circularities is to add redundant constraints to the network that are alternate ways to view the constraints that are already there. Using our midpoint example, two constraints might be added as follows:

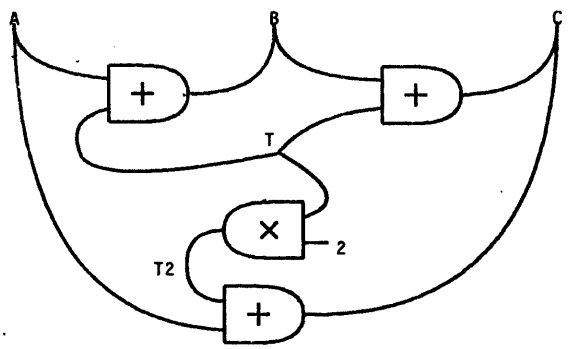


Figure III-5: The midpoint constraint with an alternate view

Expressed algebraically:

$B = A + T$
$C = B + T$
$C = A + T2$
$T2 = T * 2$

Now, no matter which pair of values from the set $\{A, B, C\}$ is defined the other can be determined by propagation. Given A and C, the third constraint can be used to derive $T2$, using the forth constraint T can be derived, then B follows from either the first or the second.

In previous systems where the use of alternate views was advocated as a method for extending the applicability of propagation, they have been constructed manually. This introduces substantial complexity to the process of specifying constraint nets since the user must both recognize the need for an alternate view and construct the alternate view. Redundant specifications like this introduce opportunities for errors that are very hard to detect. In many programming languages redundancy like that found in variable declarations is used in part for cross-checking the user's program; the compiler is able to compare the two differently expressed statements of the same thing and check their consistency. This is not possible in a constraint system since comparing systems of equations for equivalence is generally an unsolvable problem, although it is often solvable in practice.

Inconsistency problems can be avoided by a slightly different use of alternate views. If the user detects a situation where an alternate view would be required, a single primitive constraint can be constructed by hand that captures the semantics of the original network but avoids its problems. The part of the network that was causing the problem is replaced by the newly constructed constraint.

Returning to the midpoint example, one would construct a single constraint on three values that constrains one to be at the midpoint of the other two. Semantically this is equivalent to the other network, but propagation works properly on it. See figure III-6.

It should be clear from the preceding discussion that a propagation technique is very desirable. It combines simplicity and good performance very nicely. Its major drawback is that there are many situations where it fails. However, by using alternate views it is possible to transform a network where propagation doesn't work to one where it does. The construc-

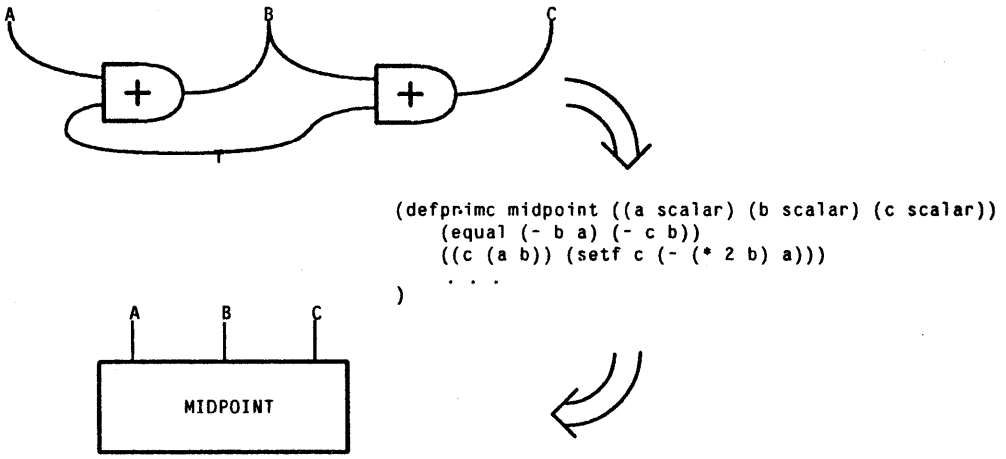


Figure III-6: Replacing a net by an alternate view of it.

tion of these alternate views when the basic constraints are limited to algebraic relations is a mechanical task, given the ability to perform simple algebra. Recognizing situations where alternate views are productive is also fairly mechanical.

This suggests that what one really wants is a system that automatically recognizes situations where alternate views are necessary and constructs and applies them. This automatic recognition and construction is the subject of chapter IV.

III.1.6. The MUMBLE Microcode Compiler

This section describes the code generator in the MUMBLE [8][2] compiler. A simpleminded constraint satisfier is used that achieves surprisingly good results. It is the effectiveness of such a simpleminded approach that is of interest.

MUMBLE generates horizontal microcode from programs in a simple high level language. Statements are translated into micro-operations which are the simple conceptual atomic instructions out of which programs are built. These correspond closely to the instructions in a conventional machine. The interesting aspects of a horizontal micromachine are that several of these micro-operations can be fit into one instruction word and that there

[2] A *M*ost *U*nlikely *M*icroassem*bler*

are often critical timing dependencies between operations. For example, consider a hypothetical micromachine that has a general register set and an ALU. Within a microword are fields specifying the source registers, the ALU operation, and the destination register. The catch is that the ALU operation in one microword uses the source operands specified in the previous microword and its result is stored according to the destination specified in the following microword - this is just an explicit specification of the stages of a pipeline. Thus a simple statement like 'A:=B+C' must be compiled into three microwords, but there is much 'free space' in them that can be used by merging adjacent statements.

A linear block of code is represented as a dependency graph, each micro-operation having a link to all those micro-operations on which it depends and each link having information about the type of dependency. The implementation restricts the dependencies to be of one of the following five types:

- The two microinstructions must not reside in the same microword.
- They must be executed in the same microcycle.
- One must be executed precisely one microcycle before the other.
- One must be executed in the same or earlier microcycle as the other.
- One must be executed in some microcycle preceding the execution of the other.

Figure III-7 contains a sample of a dependency graph.

Once the dependency graph is constructed, the address assignment algorithm is trivial:

1. Assign all micro-operations to relative address 0 within the linear block.
2. Scan all the micro-operations and examine their dependency arcs. For each micro-operation:
   - If some dependency upon another micro-operation is violated
   - then push one of the micro-operations to a higher address until the dependency is satisfied.
   - This may cause other dependencies to be violated.
3. If no micro-operation had to be pushed forward, then quit. It is guaranteed that this address assignment is admissible.
4. Otherwise go back to step 2.

Figure III-8 demonstrates one pass of the address assignment algorithm over the example of figure III-7, again many dependencies have been ig-

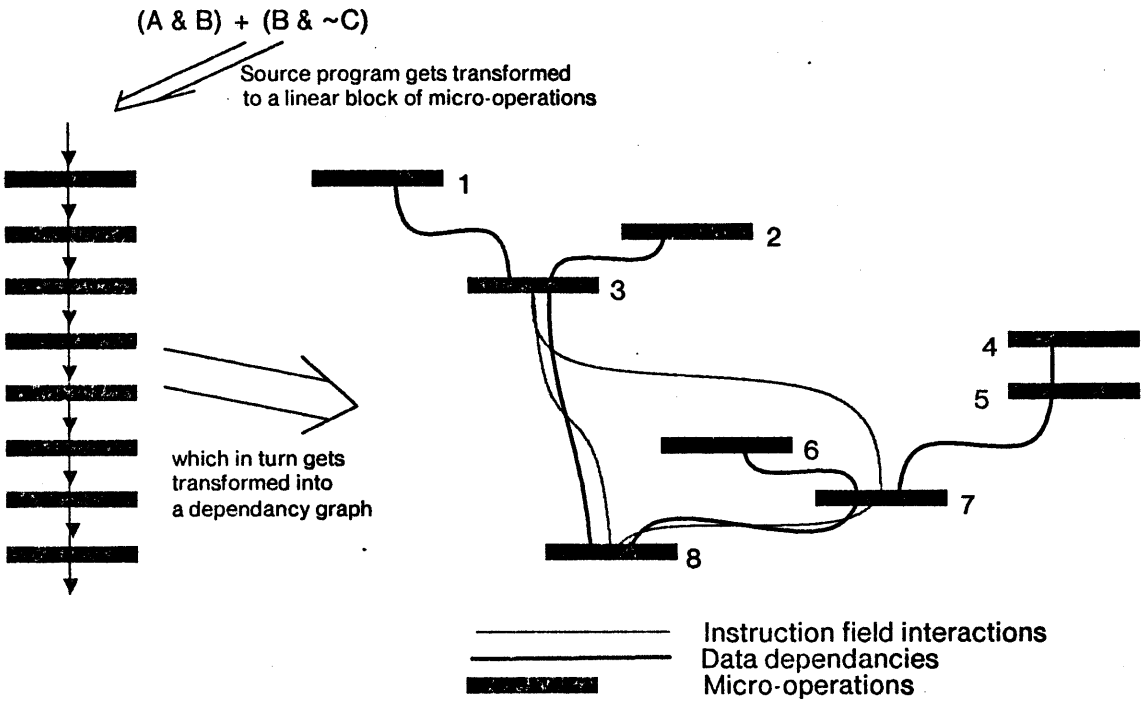


Figure III-7: Translating a program into a dependency graph

nored. Note that after only one pass, the address assignment is already correct.

It should be clear from the construction of the algorithm that the address assignment produced will be admissible; that is, it will be functionally correct with respect to the semantics of the original program and the characteristics of the micromachine.

What is not clear is the minimality of the assignment: that the length of the linear block is as short as possible. Experiments have shown that human microprogrammers can rarely do better than the compiler for a single linear block. On average the two perform identically, with a lack of vigilance on the part of human microprogrammers often causing them to do worse than the MUMBLE compiler.

Termination is also an issue: if no address assignment is possible because of conflicting circular dependencies the algorithm will never terminate, even if an address assignment is possible. By using an adversary-style

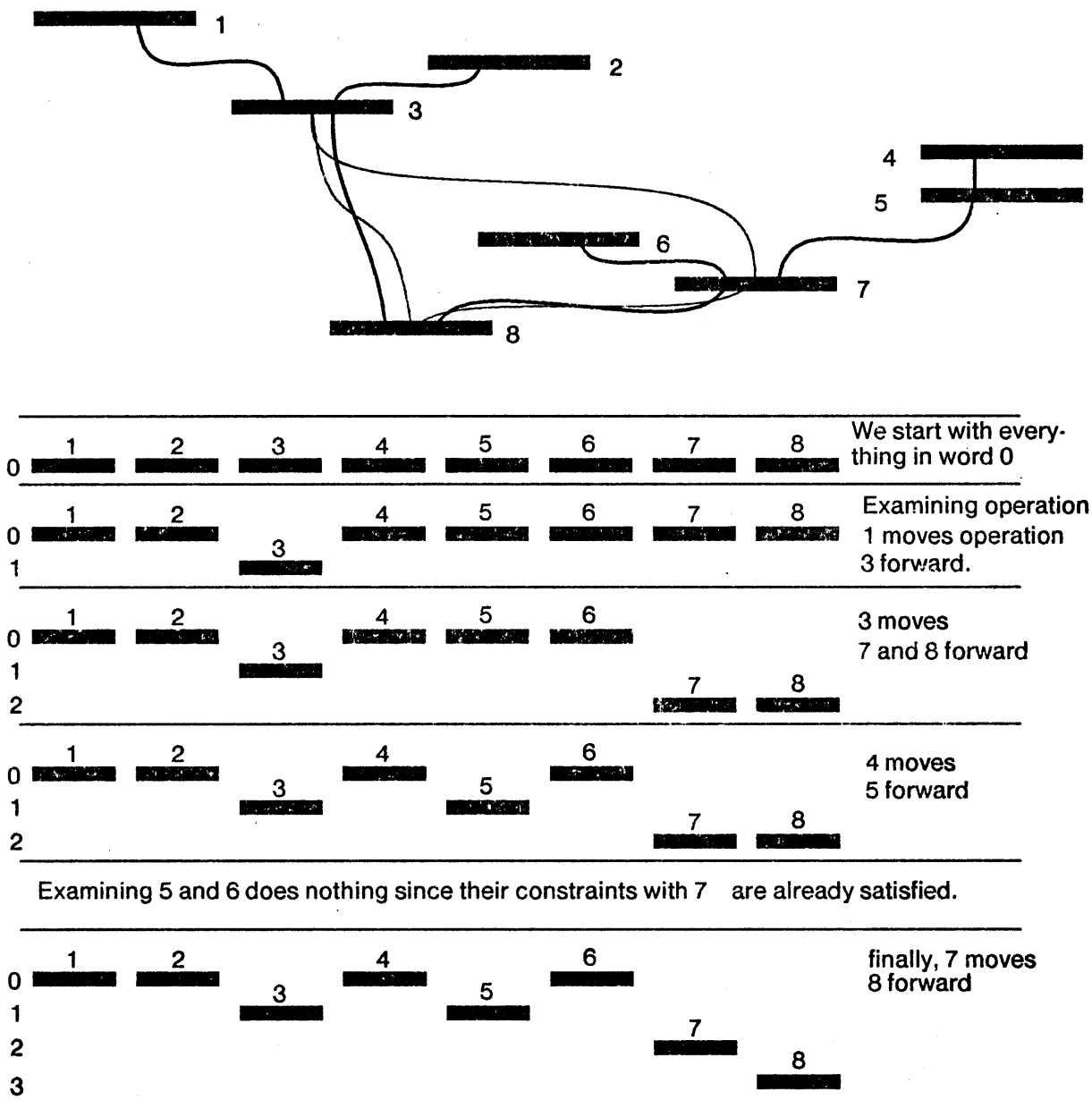


Figure III-8: The address assignment algorithm

argument it is possible to cause a choice of which micro-operation to push that will keep the algorithm running indefinitely. This situation is avoided by heuristics[3] that have proven to be very effective. In the im-

[3] a.k.a. *Hacks*

plementation it is assumed that non-termination, detected using a counter, is a result of a fundamental contradiction in the constraint graph. MUMBLE does no sophisticated graph analysis.

The number of iterations of the algorithm when applied to real programs is usually in the range from 2 to 4. Figure III-9 contains statistics collected from the microcode support for the ECHOES [10] operating system. It relates the number of micro-operations in a block and the number of iterations required to pack the block to the number of blocks with those properties. As you can see, nearly half of all linear blocks were packed in one iteration and contained fewer than 10 micro-operations; since they were packed in one iteration, they had to generate one microword. Over 85% of all the linear blocks were packed in 3 or fewer iterations.

	Number of Micro-operations										total	% of overall total
	0	10	20	30	40	50	60	70	80	90		
1	126										126	40.91%
2	33	18	14	5	1						71	23.05%
Number 3		24	11	24	5		1	1			66	21.43%
of 4			2	4	5						11	3.57%
Iterations 5		11	3	3	6	2			1		26	8.44%
6			2	3						2	7	2.27%
7				1							1	.32%

Figure III-9: Packing algorithm performance

Most of the blocks appeared, on inspection, to be packed as tightly as possible. The skilled human microprogrammers who examined the code found almost none that they could improve. While examples can be constructed that MUMBLE will do poorly on, these examples occur infrequently in practice. MUMBLE did poorly in interblock code motion, but was good at packing within one block.

The moral to this story is that there is often no need for theoretically complete but computationally expensive solutions when the hard cases never appear. MUMBLE's very simple satisfaction algorithm works perfectly well, but only in a very limited and special domain.

III.1.7. EARL, a VLSI Layout System

Chris Kingsley in [11] describes EARL, an integrated circuit design language. At its heart is a constraint system that computes the layout of a chip based on user-specified constraints. The only constraints allowed are that a point be either to the left of or above another point by a distance of either at least or exactly k. The x and y constraint graphs are thus separable.

This amounts to a constraint network based on scalars where the only constraints allowed are either $a = b + k$ or $a \geq b + k$. Constraints of the form $a = b + k$ are removed from the graph by substituting $b + k$ for a everywhere that it occurs. The satisfaction algorithm minimizes the values of each variable, relative to the least value.

Consider constructing the transitive closure of the relation represented by this graph. That is, if $a \geq b + k_1$ and $b \geq c + k_2$ then $a \geq c + k_1 + k_2$ can be deduced. Also, if $a \geq b + k_1$ and $a \geq b + k_2$ are known, and $k_1 \geq k_2$, then the second constraint may be ignored since it is redundant. Thus a matrix can be constructed of size n^2, where n is the number of cells. Each element of the matrix contains the minimum spacing between two cells.

If there is a contradiction in the network then $a \geq a + k$ will be deduced, where $k > 0$.

To assign values to the variables, pick some variable a. For all b_i such that $b_i \geq a + k_i$ is known, set b_i to $a + k_i$ if it's greater than b_i's old value, if it had one. We don't need to account for constraints of the form $b_j \geq b_i + k$ being violated since they would be accounted for in the transitive construction of $b_j \geq a + k_j$. Minor complexities are introduced when one such pass does not assign values for all the variables; another must be picked to deal with related unaccounted-for relations.

The full transitive closure is not actually computed. EARL computes what Kingsley calls the transitive almost-closure. It depends on the observation that the address assignment phase ignores constraints of the form $a \geq b + k$, where b is not one of the picked variables. For each node in the constraint graph that has both incoming and outgoing arcs, each incoming arc is combined with each outgoing arc to yield a new constraint and the old outgoing arc is removed. Multiple constraints connecting the same pair of nodes are resolved in favor of the widest of them. The result is a net where no node has both incoming and outgoing arcs. See figure III-10.

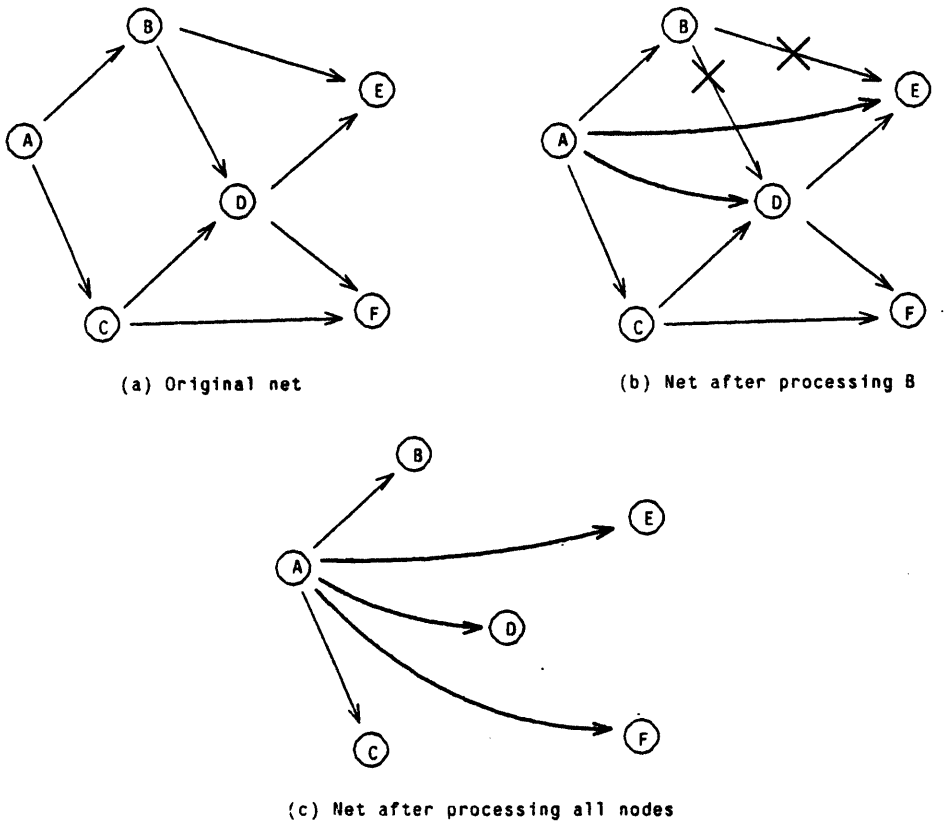


Figure III-10: Construction of the transitive almost-closure

Looked at in another way, what this algorithm does is compute the longest path between each point and each leftmost point.

The point of this detailed explanation of Kingsley's algorithm is that special properties of the constraints involved should be exploited to the fullest. He obtains a lot of leverage, and a fairly fast algorithm that performs a minimal packing, by being able to do deductions based on a restricted class of constraints. Compromises must always be made; this algorithm does not solve the real problem, that of laying out an IC design in the minimum area. It solves a different problem, but one that is close enough to be useful.

III.2. Incremental Satisfaction

The applications of interest in this thesis maintain a network of constraints and values that is incrementally modified. Constraints and cells will be created and deleted and cells will have their values changed. In the midst of these changes, and in step with them, the assertions represented by the interconnected values and constraints must be maintained. Values related to the changes must be changed to account for the constraints. Hence the satisfaction process is given a set of values that is not neatly partitioned into known and unknown categories. Not only must the process calculate new values, but it must also choose which values to change. Often the choice is not obvious or may be done in one of many ways. Subsequent parts of this section present a progression of incremental satisfaction techniques.

III.2.1. Unplanned Firing

The work on MUMBLE led me to try a naive extension of its satisfaction algorithm to handle arbitrary constraints. Each value in a constraint network is connected to some constraints, which are in turn connected to other values. When a value changes, the simple algorithm examines each attached constraint and uses it to change some other value attached to it. The choice of which value to change can be difficult. This is exactly like propagation, without the partitioning between known and unknown values. One can think of the partitioning as providing guidance in the selection of which rule to apply.

This algorithm is very similar to the way that dataflow machines operate [4, 19]. Each constraint acts as a processing element in the dataflow machine. The crucial difference is that in a dataflow machine the devices are directional, but in a constraint network they are not; they can compute results in any direction. Both Petri nets and Markov algorithms are similar to dataflow machines.

If unplanned firing is applied to the Fahrenheit-Celsius network that appears in figure I-3, page 7, and the only guidance used is not to fire rules that would overwrite constants or values that have just been generated, then changes to any variable node will be correctly propagated.

Unfortunately, there are many situations where problems arise. A simple case is shown in figure III-11a. Here the value of cell *A* is uniquely determined by a constraint relationship with a constant. If the sum is satisfied

by changing *A*, then the equality constraint cannot be satisfied without either changing the constant or *A*. A simple heuristic for handling such cases is to attach to each value the time that it was last changed. When constraints fire, the rule chosen is the one that alters the oldest non-constant values. With this modification, the equality constraint in figure III-11b will change *A* back to its old value, then the sum constraint will change *B*. This amounts to a crude form of backtracking.

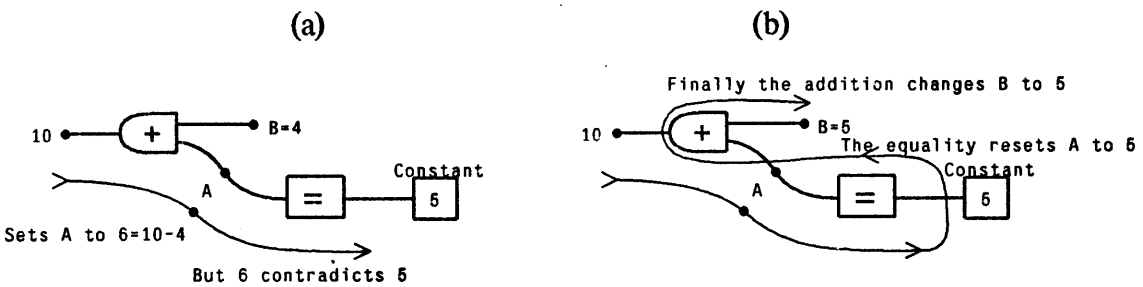


Figure III-11: Failures in unplanned firing.

There are still problems. Networks like III-12 can be constructed that are satisfiable, but that will cause an unplanned firing satisfier to loop. One might be tempted to try detecting these loops, but it is easy to get confused between loops and backtracking.

There is an interesting special case involving constraints on two cells where a known value for one implies a unique value for the other. In such a system all deductions are forced. If a sequence of firings exists that will satisfy the system, then unplanned firing will find it. The only situation where a satisfactory set of values exist but will not be found occurs when a global technique is necessary. For example, the constraints $a=b$ and $a=-b$ are only satisfied when a and b are both 0. This cannot be deduced given an arbitrary initial value for one and a sequence of applications of the rules $a \leftarrow b$, $b \leftarrow a$, $a \leftarrow -b$ or $b \leftarrow -a$. This special case is actually more general: it holds whenever the constrained cells can be partitioned into two sets where knowing values for all in one set implies unique values for all in the other. When designing a constraint-based system, decreeing that this special case will hold and making appropriate restrictions on the constraints that it will deal with can result in substantial simplifications and speedups.

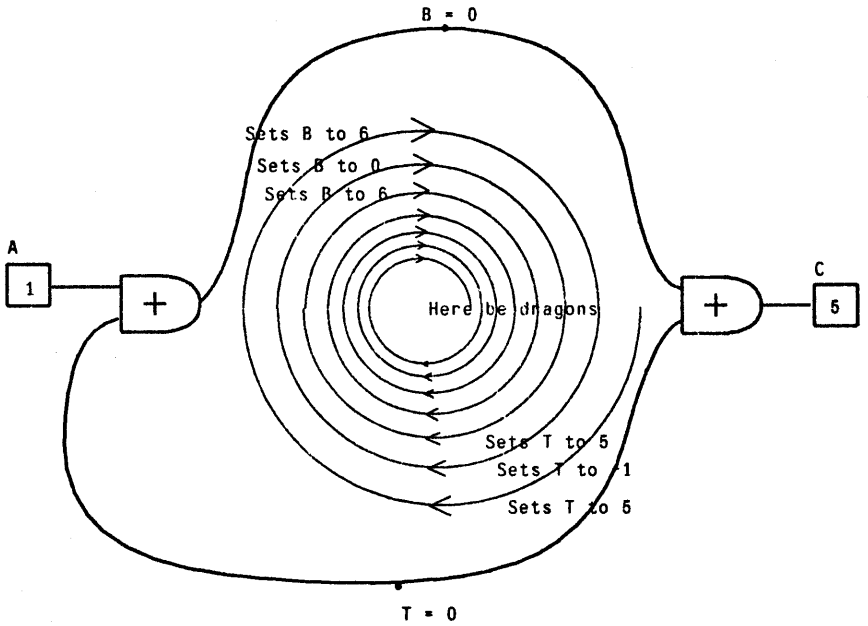


Figure III-12: An infinite propagation loop

While this unplanned firing method is simple, local, and potentially fast, it depends too much on making the right guesses. It would be wonderful on a nondeterministic computer. In the sections that follow, satisfaction methods that perform sophisticated planning are presented along with a discussion of their shortcomings. Finally, unplanned firing will be revisited and improved.

III.2.2. SKETCHPAD

Sutherland's SKETCHPAD system [18] contained a satisfaction mechanism that used propagation. The propagation phase was preceded by a planning phase that used a topological sort to order the constraints. If no ordering could be found, relaxation was used.

A *free variable* in SKETCHPAD is one 'which has so few constraints applying to it that it can be re-evaluated to completely satisfy them '[4]. When the constrained value is a scalar, it can have at most one constraint applying to it. The significance of such a free variable is that if the one constraint that it is attached to is not satisfied, then the variable can be changed without fear of invalidating some other constraint.

[4] Tut, tut Ivan. A split infinitive!

The ordering algorithm iteratively identifies all free variables, eliminates them and the constraints they are attached to. This may free up variables for the next iteration. Iteration stops when there are no more free variables: if there are no variables left at all in the network then an ordering has been found, otherwise relaxation must be applied. When free variables are eliminated they are placed in a list in the order of elimination. This order is the ordering used in reverse for the propagation stage.

The propagation phase runs through the list of variables in order from the most recently eliminated to the least recently eliminated. For each variable it uses the constraint attached to it that has all other parameters specified to calculate a value.

Consider figure III-13. The squares represent constraints and the small circles represent constrained values. The large concentric circles divide the network into layers. The outer layer contains only variables that are attached to one constraint. SKETCHPAD's algorithm works by stripping off this outer layer, exposing the next layer, that is stripped off in turn. Then, working from the inside out, values are assigned to variables.

This algorithm has several good points:

1. Its runtime is linear in the size of the network.
2. Checking for failure is simple, it falls out naturally from the order calculations.
3. The ordering is not affected by changes in the values of the constants in the network, so the same ordering may be reused.
4. It calculates an exact answer, within the usual numerical limitations of the representation of real numbers.

It does, however, have some problems:

1. Its runtime is linear in the size of the network.
2. The ordering is affected by changes in the structure of the network.

The linear runtime of the algorithm may appear attractive, and it certainly is by comparison with relaxation. But when you consider the task of interactively editing large graphical structures, like the layout of an integrated circuit, a runtime that is even linear in the size of the graph is unacceptable. Many changes to the structure of the net have only a local effect. However, the planning phase starts at the farthest reaches of the graph and works inwards to the center, where the change is happening. It would

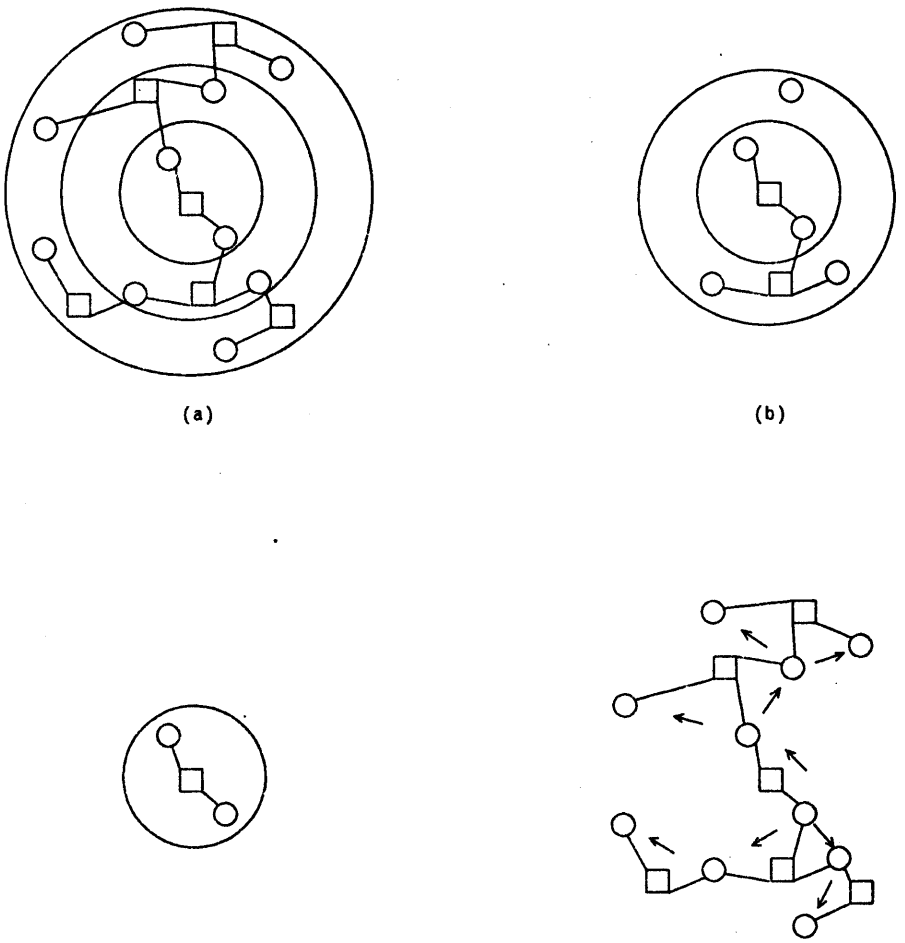


Figure III-13: Propagation and Ordering in SKETCHPAD

be better to restrict the range of the algorithm to just those areas of the graph that are affected by a change.

Many of the changes made when editing a drawing affect the structure of the net, hence the evaluation of this ordering may have to be made quite frequently. Since this is a global process, it is quite expensive.

III.2.3. THINGLAB

Alan Borning's THINGLAB system has some stringent performance requirements that have a substantial impact on the satisfaction mechanisms

it uses. These requirements arise from the way that users are expected to interact with THINGLAB. Users interactively sketch constraint systems and can grab points and drag them, with constraint satisfaction happening continuously as the points are being dragged. For example, suppose two lines are defined such that the length of one interpreted as a Celsius temperature matches the length of the other as a Fahrenheit temperature. If the end of one line is dragged, then the end of the other will also drag along. Think of a drawing of two thermometers side by side where the mercury in one can be grabbed and dragged using a pointing device. Then the mercury in the other will move correspondingly. This corresponding motion happens continuously in step with the dragging of the other line.

To achieve this, for each editable thing, such as a point or a line, a SMALLTALK routine is constructed that performs the necessary assignments to propagate the changes indicated by the constraints. These routines are constructed on the fly as they are needed. Like SKETCHPAD, THINGLAB performs satisfaction in two phases: planning and execution. The planning phase is the construction of these routines, and the execution phase merely invokes the constructed routines. Dragging an object simply involves repeated invocation of the routines with only an initial execution of the planning algorithm.

The planning phase of THINGLAB provides an interesting contrast to SKETCHPAD. While SKETCHPAD's planning is global and independent of the change being made, THINGLAB's is local and driven by the change. It starts by looking at the part to be changed. Then it looks at the constraints connected to it and the parts they connect to. This planning spreads out radially along possibly many paths. If two paths intersect, the method fails and relaxation is used. This carries on outwards, progressively getting farther and farther from the original points, until the edges of the connected region are encountered. SMALLTALK code is generated by traversing this set of constraints in the order that it was built.

It is important to note that this planning is dependent *only* on the structure of the net and not on any values, just as in SKETCHPAD. In situations where the change of a value on a pin of a constraint may leave the constraint satisfied, this sort of planning can involve examining more of the net than is necessary. A good example of such a constraint is "less than or equal to". Such constraints are common in VLSI layouts. Every point is

likely to be reachable by every other point through a chain of them, yet moving a point in a layout will affect only the parts nearby.

In the actual operation of THINGLAB this problem with planning does not inflict a runtime penalty. Rather, the performance problems stem from two sources. First, THINGLAB resorts to relaxation when propagation doesn't work. This happens when changes are made near any circularity in the net. The use of relaxation is made less costly by some tree-trimming operations. By providing many powerful primitive operations and advocating the use of alternate views, the use of relaxation is often avoided.

Second, THINGLAB performs an expensive compilation whenever a point is moved that hasn't been moved since the last structural change. Any time that a structural change is made to the net, all of the remembered and compiled plans are thrown away. Some of this could have been avoided by clever bookkeeping.

III.2.4. Retraction

The constraint systems built by Stallman and Sussman, and Sussman and Steele [15, 16] keep track of the flow of information during propagation. Variables are either bound to a value or they aren't. Each time an unbound variable is bound to a value the *premises*, those variables on which the new value is based, are recorded. If the value of a bound value is to be changed, then it must be unbound first. This 'unbinding' is called *retraction.* If a variable to be retracted is a premise of some other variable's value then that variable is retracted too, so that its value can be recomputed. If a variable to be retracted is premised on some other values, then these must be accounted for as well.

Values come either from outside the system, set by the user, or from inside as the result of some constraint firing. Each time a rule is fired and a new value for a cell is computed, the cells that the computation used as inputs are recorded.

Each cell has a set of deductions associated with it. This is the set of cells whose values were based on its value, through however long a chain of reasoning. Looking at the history information in the other direction, each cell has a set of premises that contains those cells that implied it. The *ultimate premises* of a cell are those premises that do not themseleves have premises.

Steele [16] achieves the effect of incremental satisfaction by retracting a cell's value, giving the cell a new value, then performing propagation as in section III.1.1, page 20.

Consider the case where a cell has no premises; that is, it wasn't deduced from anything by the firing of a contstraint. This corresponds to a value set by the user. In order to retract the value from its cell, the effects of the old cell must be undone. This is simply a matter of recursively removing the effects of each of the deduced values.

In the case where a cell has premises, more needs to be done than simply retracting the deductions. This value was deduced from other values; if it were changed without accounting for this then the constraints involved in those deductions would be contradicted. However, if any one of the ultimate premises were retracted, then the original cell would also be retracted since it must be in the set of deductions from the ultimate premise. The satisfaction process is then expected to propagate a new value from the changed cell to the cell originally retracted. The changed cell will now be an ultimate premise of its old ultimate premise.

For example, we start with the constraint graph in figure III-14.

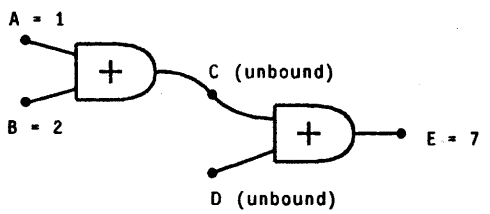


Figure III-14: Initial graph.

Running the propagation algorithm, A and B are used to deduce C, then C and E are used to deduce D. The final net with the deduction information is shown in figure III-15. A, B and E have no premises. C has A and B as premises and ultimate premises. D has A, B, C and E as premises but only A, B and E as ultimate premises.

In order to change A we first retract it. This involves following the depen-

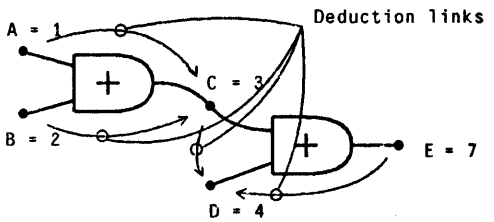


Figure III-15: Deduction information after propagation.

dency chains and also retracting **C** and **D**. The result is shown in figure III-16.

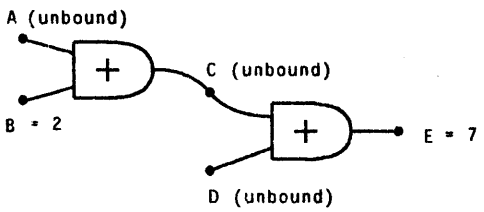


Figure III-16: Retracting a cell with no premises.

Assigning new values to **A** and performing propagation, the net in figure III-17 results.

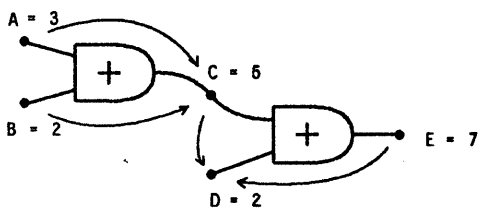


Figure III-17: Propagation after retraction.

A similar, but less extensive retraction occurs when **E** is changed.

Continuing with this example, we attempt to change **C**. Its set of ultimate premises, **A** and **B**, is not empty. One of these is picked, say **B**, and is retracted. This retracts **C** and **D**, leaving the net of figure III-18.

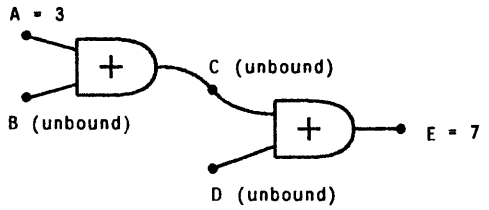


Figure III-18: Retracting a cell with premises.

C is now given its new value and is propagated. This and the new deduction links are shown in figure III-19. Note the differences between this and figure III-17.

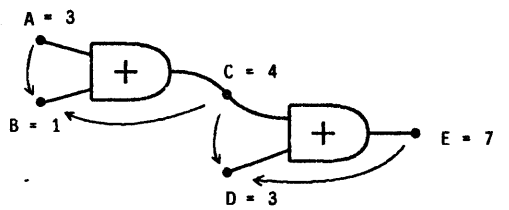


Figure III-19: Propagation after retraction with premises.

Those nodes that were retracted and had been deduced from **C** before the retraction are again deduced from it by propagation. But those that were retracted and were premises are now deduced. The deduction chains have been reversed.

A set of deductions forms a tree that is rooted at some ultimate premise. For each ultimate premise there is a tree of deductions. These trees may overlap. Retraction and propagation together have the effect of taking one of these trees, picking some node in it as the new root, and reshaping the tree around it.

Retraction has several problems: it is not data directed, it presumes that all deductions are reversible, it still has to decide which ultimate premise to retract, it deals with dynamic values but not with dynamic nets, and it overly restricts the system's choice of how to propagate the effects of a change. By a deduction being reversible, I mean that the deduction could have been performed in the reverse direction — if A can be used to deduce B via some constraint's rule, then B can be used to deduce A via some other rule of that constraint. This is equivalent to saying that the deduction function is invertable: $B=f(A)$ implies that f^{-1} exists.

Retraction is based entirely on the flow of previous computations. It takes no direction from the actual data in the net. When the value of a cell is changed it might be possible to restrict the scope of the retraction by taking the new value and the semantics of the constraint into account; changing a constrained value might not contradict the constraint.

For example, if **A** is constrained to be less than **B**, and **B** was deduced from **A**, then if the value of **A** is decreased there is no need to retract the value of **B**, which flow-directed retraction would. Planning techniques which take into account the values that will be deduced, like the one described in section III.2.5, page 49, don't have this problem.

There is a strong presumption in retraction that all computations are reversible. This is used in the revocation of the ultimate premise of a cell and the subsequent deduction of a new value for it. Problems can occur when this presumption is invalid and reverse flow cannot happen.

When the cell to be retracted has more than one ultimate premise, the choice of which to retract must be made. Usually, any of them may be retracted and the system will still be satisfiable. The choice is often an aesthetic one of 'doing the right thing'. This requires an omniscience that is beyond the power of mere mortals. Any of a number of heuristics may be used:

- Ask the user: This gets verbose and annoying. It also makes the foolhardy assumption that the user knows what is going on, what is right, and cares about the differences between the solutions.
- Pick the nearest ultimate premise: the rationale being that this cell is the one 'most responsible'.
- Pick the ultimate premise with the smallest set of repercussions: this minimizes the extent of a change, but can be tricked by adversary-argument style problems and can be computationally expensive.

- Random: the rationale being that no premise is worse than any other. The user should probably be allowed to complain and force the system to try something else.

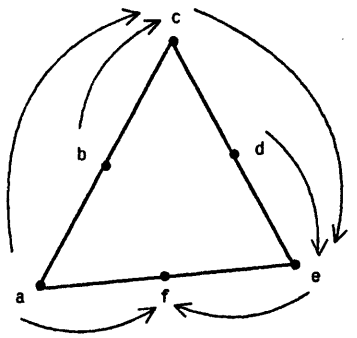


Figure III-20: A triangle of midpoint constraints.

Unfortunately, it is not always the case that the retraction of any ultimate premise will leave the system satisfiable by propagation. In figure III-20, **B** is constrained to be the midpoint of **A** and **C**; **D** is the midpoint of **C** and **E**; and **F** is the midpoint of **E** and **A**. **A** and **B** derived **C**; **C** and **D** derived **E**; and **E** and **A** derived **F**. If **F** is to be moved we must retract one of its ultimate premises, **A**, **B** or **D**. If **A** is retracted then both other vertices, **C** and **E**, will also be retracted since they were deduced from it. The system that results, even with **F** bound, is not satisfiable by propagation. All of the vertices have been retracted, which is too much. One could, in this case, pick some arbitrary value for one of the newly unbound cells, but the result would certainly not be the 'right thing'. If either of the other two ultimate premises, **B** or **D**, is chosen for retraction, then the problem will not occur.

If we look again at the triangle example, another problem can be seen by trying to change the value of **A**. Retraction has no choice: it removes the values of **C**, **E** and **F**. Propagation can then re-evaluate them without any problems. If this triangle was drawn on a screen and was being manipulated by the user, what he would see is shown in figure III-21. A user of such a drawing system would probably be surprised; such a twisting of the structure is probably not what he wanted. Figure III-22 shows what was

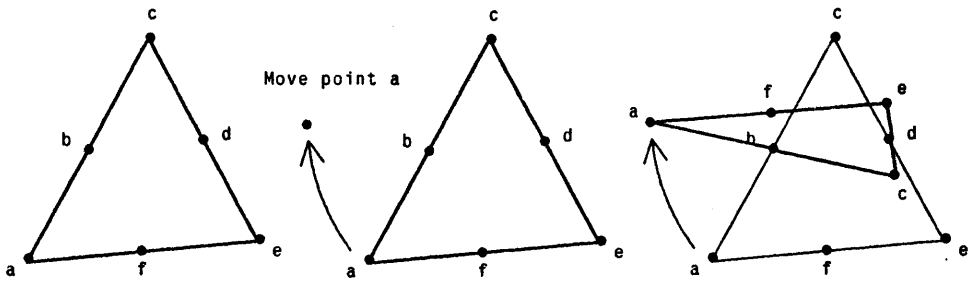


Figure III-21: Moving a vertex using retraction.

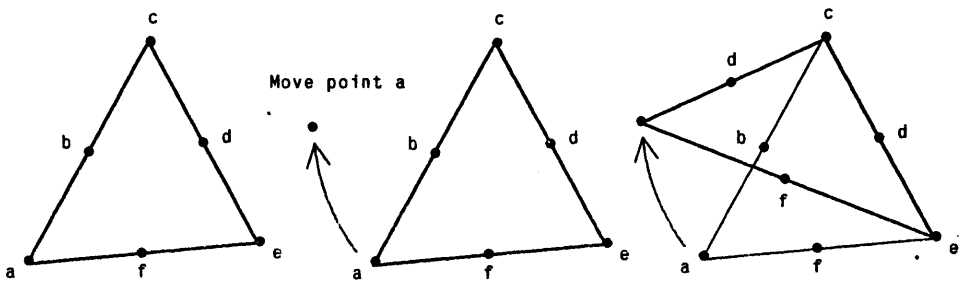


Figure III-22: Moving a vertex, doing the 'right thing'

more likely to have been on his mind. Of course, the twisting solution may be the one that the user has it mind, but it isn't the only possible solution. The point here is that by restricting itself to the data flows in the dependency graph the algorithm has blinded itself from considering in this case that **B** could have been moved in response to the movement of **A**. It is based on the naive belief that if something worked once, you should stick to it.

Retraction uses previous flow patterns to guide subsequent flows. It provides no guidance if there were no previous flows. This problem occurs when a new constraint is added to an existing network. If it is unsatisfied when it is instantiated then some of the constrained values must change. Often, as in most of the examples presented so far, a constraint can be satisfied by changing just one of the constrained values — in that case the satisfaction algorithm can pick one of the ultimate premises of one of the constrained values and retract it. In more complicated cases, more than one value needs to be retracted. Information is needed from

the constraint of the form 'in order to satisfy this constraint it is necessary to retract either **A** and **B** or **C** and **D**.'

Note that there are few problems when a constraint is removed from the system. If the system was originally satisfied, then the system will surely remain satisfied. If it was unsatisfied before, then it may be satisfiable after.

Despite these flaws retraction is still a useful and elegant technique. It plans the propagation process with low overhead using only local information. A small bookkeeping expense in the propagation phase has paid off with performance in planning.

III.2.5. Breadth-First Planning

Preceding sections of this thesis have discussed several simple and elegant techniques for performing incremental satisfaction. Unfortunately they all have either the problem of not finding a solution or finding a 'bad' one in many cases. In this section the brute application of force is discussed. This results in guarantees of success and 'goodness' and, it will be argued, without much performance penalty.

What is a 'good' solution? To a first approximation, it is any one that is correct. When there are many correct solutions one must be chosen somehow. This choice is necessarily domain dependent. For the purposes of this thesis I have chosen to prefer solutions that change a minimum number of values. For the applications that I am concerned with, graphical editors, this seems reasonable since it maximises the coherence of the satisfaction process; as many things as possible are left alone on the screen.

The unplanned firing algorithm of section III.2.1, page 36, was the first one tried in MAGRITTE. It worked most of the time but sometimes would not find a solution, or the solution found would be 'surprising'. Next, backtracking was added. This turned it into a depth-first search. It always found a solution if one existed. If the search was stopped when the first answer was found, it tended to give 'surprising' answers like the one seen in the retraction example, figure III-21, page 48. But exhaustively searching for a *good* solution consumed too much time. Other slight variations and algorithms from the literature were tried without much success. It became clear that there was no good alternative but to bite the bullet and

use breadth-first search. In this case, the criteria for a good solution are used to limit the search.

Breadth-first search works by first constructing all sequences of constraint firings of length one. Then it extends these to all sequences of length two; then length three, four, five... until finally one is reached were all constraints are satisfied. This is then taken as the solution. In the general case, the performance of breadth-first search is most affected by high branching factors. If there are many different deductions that can be made, then each sequence of length n leads to many sequences of length $n+1$. After very few layers this exponential blowup will produce an overwhelming number of possibilities.

Fortunately, in the case of constraint systems used in drawing editors the network structure provides a lot of guidance. The most common branching factors seen are one and two. Constants have the effect of cutting down the branching factors of the constraints to which they are attached. It is crucial that every opportunity to reduce the branching factor be exploited.

One can think of a single sequence of constraint firings as a tendril that spreads out through the network. At each cell it branches to all the attached constraints and at each constraint it picks some adjacent cell, although it might have to choose several, and spreads through them. Finally, the tendril stops when it reaches a cell that has no attached constraints other than the one it just came through, or where none of the attached constraints is violated by the cell's new value. The tendril satisfies the system when all of its branches stop.

Different tendrils are created each time there is a choice of which cell to alter at a constraint. The breadth-first algorithm maintains a set of tendrils. It iteratively extends each one, creating new tendrils as choices are encountered, rejecting them when they reach a contradiction, and finally succeeding when any of the tendrils succeeds.

The basic data structure used by the algorithm is a *tendril.* A tendril is composed of a set of bindings and two sets of constraints. The bindings are those changes to cells values that are represented by this tendril. One set of constraints contains those that have fired to create the tendril. The other contains those constraints that are suspected to be unsatisfied. This

is the *SuspectConstraints* queue from the simple propagation algorithm of section III.1.3, page 22. All constraints that need to be satisfied must be in this queue, but not all of the constraints in the queue definitely need to have some cell's value changed in order to be satisfied.

Two sets of these structures are maintained. One contains sequences of length n and the other of length $n+1$: one for this iteration and one for the next. Initially the set for the first iteration contains one tendril that contains those cells that were changed by the user in the last editing step and the adjacent constraints. The algorithm terminates when a tendril is constructed that has an empty set of suspect constraints.

The breadth-first algorithm:

1. Empty the set of tendrils for the next iteration.
2. If the set of tendrils for this iteration is empty, terminate. The system is unsatisfiable.
3. While the set of tendrils for this iteration isn't empty:

    a. Take a tendril from the set of tendrils.
    b. Pick a constraint from its set of suspect constraints.
    c. Apply it, taking into account the bindings for this tendril. The constraint is allowed to fire in any direction that doesn't change a constant or a cell bound by this tendril.
    d. If the constraint was already satisfied, go back to **b** and pick another constraint.
    e. If the constraint wasn't satisfied and it's a member of the set of constraints that were fired to create this tendril, then a loop has been encountered: abandon this tendril. ‡
    f. If the constraint couldn't fire, then this tendril is a dead end. Go back to **2** and pick another one.
    g. If the constraint fired then create a tendril for the next iteration that is almost the same as the current tendril. The exceptions are that it contains the bindings from the firing and has the constraint moved from the set of suspect constraints to the set of constraints that have fired. Add to the set of suspect constraints all constraints adjacent to the values that changed. Go back to **c** to apply the constraint again, leaving around hints so that it won't fire in the same direction again.

4. If a tendril has been constructed that has an empty set of suspect constraints, terminate with a cheer. The system has been satisfied.
5. Move the set of tendrils for the next iteration to the set for this iteration and start a new iteration.

For an example of the running of this algorithm, see section IV.5.2, page 80. It shows the integration of this algorithm and the network transformations of chapter IV, page 53.

It is difficult to provide a detailed analysis of the performance of breadth-first satisfaction. So much depends on the general nature of the networks encountered. To say that it is exponential in time and space is to ignore the special properties of the tasks at hand. In networks where there are many 'loose ends' — objects that have at most one constraint — the satisfaction process is very quick since the loose ends terminate the search soon. If there are many direct deductions — constraints with only two non-constants attached to them — then not many tendrils will be created and the satisfaction process will go quickly. In MAGRITTE's domain of graphical editing, both of these conditions tend to hold.

The configurations that cause the most difficulty involve loops, particularly small tight ones. The search will spiral around the loop, branching out at its contacts with the rest of the world, creating a new tendril at each such branch. As it spirals, many tendrils are generated and the search effectively grinds to a halt. There are many heuristic tricks that can be used to make these situations more bearable, but they still remain a problem. Fortunately it is exactly these situations, small tight loops in the graph, where the algebraic transformation techniques of the next chapter really pay off. They can be used to analyse such hot spots and reduce them to much simpler configurations.

IV Transformation

The efficiency and tractability of constraint network satisfaction algorithms can be enhanced by transforming parts of the network of constraints into single constraints. Preceding sections of this thesis have discussed the need for and application of such transformations. This chapter discusses how to perform these transformations, and how to detect the situations where they are needed and where they fail.

These transformation techniques only apply to networks where the primitive constraints are restricted to the operations of conventional algebra: sum, product, and equality. The transformations are based on the ability to perform algebraic transformations on equations derived from the constraints.

Two transformation algorithms are presented in this chapter. The first generates constraints whose primitive definitions require the enumeration of all useful cases. This is the form used in preceding sections of this thesis and discussed in section II.3 on page 15. The second algorithm is based loosely on the first but uses a different target primitive constraint definition. It usually avoids the exponential complexity blowup of the first.

The chapter ends by discussing when to use transformations, how to avoid them when possible, and how to link them to satisfaction algorithms.

IV.1. Transforming Entire Given Nets

Assume that we have a network that is to be converted entirely to a single new constraint. We will be abstracting its properties to maintain its external behaviour while suppressing inner details. Such networks are usually derived either by detecting a problem subnet of a larger network or by using the hierarchy of object construction.

Given such a network, the transformation algorithm seeks to construct a single primitive constraint that captures the external semantics of the entire net. Remember that a primitive constraint specification contains three parts:

1. A set of *parameters* to which the constrained values are bound.
2. A *rule* that is a boolean expression that evaluates to true or false depending on whether or not the constraint is satisfied.
3. A set of *clauses* that are used to evaluate propagated values. One exists for each possible evaluation direction.

The exhaustive set of clauses in this form of primitive constraint definition is used by SKETCHPAD, THINGLAB, and Steele's system. Transformation techniques for this form are given in this section. Following sections present a different form and a corresponding transformation technique.

Figure II-1, page 16, contains the actual definition of the primitive *sum* constraint in the MAGRITTE system. It constrains three scalars called *result*, *a*, and *b* such that $result = a + b$. The three clauses at the end specify how to propagate values in all possible directions. For example, the last of them states that to evaluate *b* given *result* and *a* the difference of *result* and *a* should be computed and assigned to *b*.

The last and most complicated stage of the transformation process is the construction of the propagation clauses. One wants to derive expressions for each subset of the set of parameters that evaluates each of them using only the values of the other parameters. This will usually be possible for only a few of the subsets. Brute force enumeration works, but it is very expensive computationally.

As an example of what is to be constructed, consider once again the midpoint network. Here is an enumeration of all subsets of the parameter list and expressions evaluating to them, the output set, in terms of the other parameters, the input set.

Inputs	Outputs	Expressions
{}	{A,B,C}	*Nothing can be derived*
{C}	{A,B}	*Nothing can be derived*
{B}	{A,C}	*Nothing can be derived*
{B,C}	{A}	$A=2B-C$
{A}	{B,C}	*Nothing can be derived*
{A,C}	{B}	$B=(A+C)+2$
{A,B}	{C}	$C=2B-A$
{A,B,C}	{}	*Nothing can be derived; everything is known*

If the constraint to be transformed consists of a single equation of k variables, as this example is, then only k of the 2^k sets are interesting. Restricting the enumeration to the useful cases when more complicated constraints are transformed can be difficult. This is addressed in section IV.2 on page 68.

The algorithm goes through several major phases. First, the internal and external variables are identified. Then the network is converted into a system of equations. These equations are manipulated to remove all internal variable references. The equations that remain are used to generate all of the useful assignment statements, then these are merged into the actual propogation clauses. Finally, the parts are put together into the primitive constraint.

IV.1.1. Identify Internal and External Variables

Parameters of a constraint to be constructed are identified based on the context from which the net was derived. If it comes from a cell definition, then the parameters to the cell have been explicitly specified by the user and these become the parameters to the constraint. When a net is clipped out of a larger net, the parameters are those values that are visible to the outside world.

Values are visible in two ways: they can be changed and examined by either the user or the constraint system. Those that are visible to the user are determined by the nature of the application. Those visible to the constraint system are at the periphery of the subnet, where the cut was made. Figure IV-1 should make this clear. Here the cycle involving the two lower sum constraints has been clipped out and the three values A, B and C become the parameters. T is not a parameter since it is not referenced from outside the subnet.

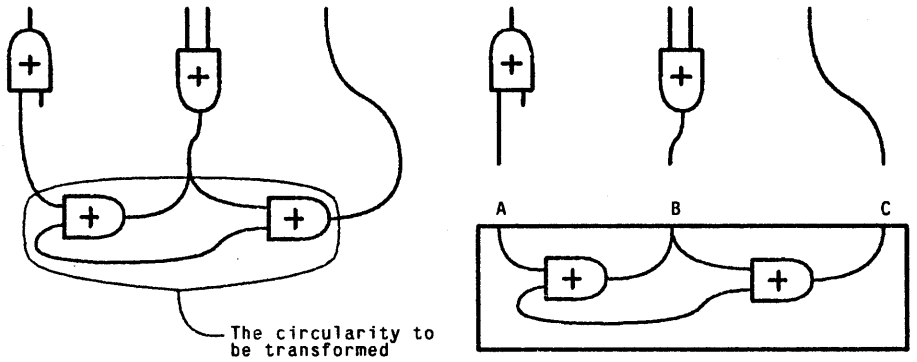


Figure IV-1: Deriving parameters when clipping out a subnet.

IV.1.2. Convert the Network to a System of Equations

The system of equations represented by the network is derived by traversing the network and extracting the rules from each constraint. Since the network is satisfied only when all of the component constraints are, the rule for the new derived constraint is simply the conjunction of the collected rules.

Using the the network clipped out in figure IV-1 and the definition of *sum* from figure II-1 we get the following two equations:

$$B = A + T$$
$$C = B + T$$

Notice that the propagation clauses of the constraints within the subnet are completely ignored. All that is being used is the satisfaction test rule. This suggests another method for defining primitive constraints: as a system of equations that will be transformed to something suitable for use by the propagation algorithm. This technique is discussed again in section IV.3.3 on page 76.

IV.1.3. An aside: solving a single equation

An important operation performed in several situations in the rest of the transformation algorithm is that of solving some equation for the value of some variable. For example, solving $B = A + T$ for T should return $T = B - A$. This can be performed by invoking a system such as

MACSYMA [13]. As far as possible in this thesis, I want to treat these systems as black boxes.

It is not necessary to have the full algebraic manipulation power of MACSYMA. Many applications need no more than the ability to deal with simple linear equations and some restricted quadratics. An example of such a system is Chris Van Wyk's IDEAL [20]. It is similar to the one used in MAGRITTE, but not as general. The batch processing orientation of IDEAL lets a numerically-oriented manipulation system work.

There is a strong assumption made in several places that it is always possible to solve any equation for the value of any variable. While this is false in general, for many domains it is quite reasonable. The existence of an extensive algebra, with its accompanying deductive machinery, is the key to the success of these algorithms.

In subsequent sections this manipulation function will be referred to as:

expression = **SolveFor**[*var*, *equation*]

which solves for *var* in the *equation*, returning the resulting expression in *expression*. Appendix I on page 89 discusses the algebraic manipulations used by MAGRITTE.

IV.1.4. Eliminate Redundant Internal Variables

Since the values of internal variables are not accessible from outside the subnet, they can safely be eliminated.

In simple cases it suffices to pick an equation that contains an internal variable, solve for it, and substitute it into all other equations that contain that internal variable. Returning to the previous midpoint example, we eliminate the internal variable T by solving for it in the first equation, which yields $T = B - A$. Then by substituting that in the second equation we obtain $C = B + B - A$.

This process is equivalent to the tree-tracing extraction of equations that was done in chapter 3 of [16]. The only real difference is that here the operations are explicitly performed in terms of symbolic algebra.

Elimination in this manner fails only if, contrary to our assumption, an equation can't be solved for an internal variable, or if some internal vari-

able occurs non-trivially in only one equation[5]. What does it mean for an internal variable to occur in only one equation? Since these equations are constraints, they must be true for the constraint represented by the whole net to be satisfied. Assume that the equation can be solved for the internal variable and that the equation is continuous in all the other variables. Then it can be transformed to $T = e$, where T is the internal variable and e is the expression that defines it. Since T is unbound outside the constraint and appears nowhere else inside the new constraint, the satisfaction algorithm is free to choose any value at all for it. The only constraint imposed by the equation is that e evaluate to something. Effectively, this means that the equation can simply be ignored and discarded.

Recapping our example so far, we have the set of parameters $\{A, B, C\}$ and the rule $2B = A + C$. The primitive constraint so far is shown in figure IV-2.

```
(defprimc midpoint ((A scalar) (B scalar) (C scalar))
    (equal (* 2 B) (+ A C))
    The propagation clauses will go here
)
```

Figure IV-2: The partially constructed 'midpoint' constraint

IV.1.5. Deriving Propagation Assignments

The propagation clauses consist of assignment statements that evaluate parameter values from other parameter values. Before constructing the clauses we need to derive for each parameter all possible ways of evaluating it in terms of the other parameters.

Consider an equation where every variable appearing in it is a parameter to the constraint. Each such equation is a relation among the variables that appear in it and may be solved for each, yielding some assignment statements:

[5] A trivial occurance is something like $A - A$.

```
OutputAssignments [e] =
  For each unique variable v in e
    Output "v := SolveFor [v, e]"
```

The set of equations that results from the elimination of internal variables, as described in the preceding section, can have *OutputAssignments* applied to its members to generate some of the assignments. All of the other assignments can be generated from equations that are compositions of the original equations.

Two equations are composed by solving one for a variable that appears in the other and substituting it. This can be done for each variable that appears in both expressions. It doesn't matter which is substituted in the other; the two resulting equations would be semantically the same.

Consider a set s of equations. Initially it is the equations from the preceding section. S is enlarged as follows:

$\forall e_1 \in s,\ e_2 \in s,\ V(e_1) \cap V(e_2) \neq \varnothing$

$\quad \forall v \in V(e_1) \cap V(e_2)$ let $s = s \cup (e_1$ with SolveFor(v, e_2) substituted for $v)$

until no new equations are being added to s. $V(e)$ is the set of variables appearing in e.

This procedure results in s being the set of all possible equations that relate a distinct subset of the input paramters. Some of these equations may be redundant, if there were redundancies in the initial set.

As described, this is a computationally expensive procedure. Here is the real implementation, as it appeared in an early version of MAGRITTE, with some notation changed to make it more legible:

ExtractedSystem is the system of equations to be transformed.
SolutionSets and *ExamineSets* are lists of pairs { *e*, *used* }:
e is an expression,
used is a set of expressions.
TotalSolution is the final set of assignment statements.

```
AddAssignment (e, used) =
  let vars be
    if ¬∃s∈SolutionSets such that V(e)= V(s.e) then
       push {e, used} onto ExamineSets
       for v in V(e) do
         push "v := SolveFor (v, e)" onto TotalSolution
    endif

FindAllAssignments =
  TotalSolution := {}
  SolutionSets := {}
  ExamineSets := {}
  for e in ExtractedSystem do
    AddAssignment (e, {e})
  for s1 = pop(ExamineSets) while s1≠∅ do
    begin
      for s2 in SolutionSets do
        for v in s1.v∩s2.v unless s1.used⊆s2.used∨s1.used⊇s2.used do
            let e be s2 with Solvefor(v, s2.e) substituted for v.
            AddAssignment (e, s1.used∪s2.used)
       push s1 onto SolutionSets
    end
```

The only tricky part of this algorithm concerns the use of the *used* sets associated with *SolutionSets* and *ExamineSets*. *Used* is the set of equations from the original set that have been composed to form the new equation. The algorithm does not compose two equations if either *used* set is a subset of the others — the composition must yield an equation whose *used* set will differ from those of the original two — each must bring in some new information.

The output of FindAllAssignments comes from the value of *TotalSolution* whose elements are generated in the deepest part of the loop. Hence its runtime will be linear in the output size. This is not quite true since the various conditionals may cut off some inner loops. It also isn't a very strong statement: the output size can be exponential in the input size.

IV.1.6. Propagation Clause Construction

Each propagation clause consists of three parts: an input set, an output set and an evaluation set. The input set contains the variables input to the clause, the output set contains those that are output by it, and the evaluation set contains the code necessary to evaluate the members of the output set from the input set. The input and output sets are used by the propagation mechanism to determine which clause to apply, and the evaluation set is used when it is decided that that clause applies.

Given the total solution from the previous phase, prototype clauses are constructed by examining the input and output variables. For the midpoint example, this results in:

Input	Output	Evaluation set
{ B, C }	{ A }	$\{ A := 2B-C \}$
{ A, C }	{ B }	$\{ B := (A+C)+2 \}$
{ B, A }	{ C }	$\{ C := 2B-A \}$

Some solutions returned from FindAllAssignments may be redundant and should be eliminated. The following procedure eliminates an assignment if there is another that evaluates to the same variable and uses a set of variables that is a subset of the variables used by the first.

For each ProtoClause *i*
  For each ProtoClause *j*
    Such that $i \neq j$, $output_i = output_j \wedge input_j \subseteq input_i$
      eliminate ProtoClause *i*

The last step remaining before the final construction of the constraint is the merging of prototype clauses with equal input sets resulting in a single clause with a larger output set. At the same time we take any prototype clause whose input set is a subset of another's and merge it into the other, but leave it as a distinct clause as well.

The midpoint example that we've been working through in this description is so simple that nothing is done by this step, but the example worked through in IV.1.7 shows it in action.

We're finally ready to put everything together. The constraint name, parameter list, satisfaction test rule, and propagation clauses are simply concatenated together and wrapped inside a *defprimc*. *Defprimc* is a LISP function used to define a primitive constraint. It takes the constituent

parts and builds the structures necessary for use by the satisfier. The final constructed midpoint constraint is:

```
(defprimc midpoint ((A scalar) (B scalar) (C scalar))

    (equal (* 2 B) (+ A C))

    ((B (A C)) (setf B (+ (* 0.5 C) (* 0.5 A))))
    ((A (B C)) (setf A (- (* 2 B) C)))
    ((C (B A)) (setf C (- (* 2 B) A)))
)
```

IV.1.7. A Large Example

The following example is similar to the midpoint example that has been used time and again in this thesis. Instead of constraining three points to be equally spaced, it defines four to be equally spaced. It is no more complicated algebraically, but it does show a few more points of the construction of the constraint.

Defalgcon is a LISP function to define an algebraic constraint. In this example, a constraint called *fourspaced* is being defined with four scalar paramaters. The parameters are related by three boolean equalities.

First, the following definition is presented to MAGRITTE:

```
(defalgcon fourspaced (a b c d)
    (equal b (+ a t))
    (equal c (+ b t))
    (equal d (+ c t))
)
```

It corresponds to the following equations:

$$t+a=b$$
$$t+b=c$$
$$t+c=d$$

Eliminating the internal variable t, MAGRITTE simplifies this to:

$$c+b=d+a$$
$$2b=c+a$$

Then FindAllAssignments is invoked on this set of equations. It generates the following equations and passes them to AddAssignment, which outputs the associated assignment statements.

$c+b=d+a$ was given.
Yielding the following assignments:

```
(setf d (- (+ b c) a))
(setf c (- (+ a d) b))
(setf a (- (+ b c) d))
(setf b (- (+ a d) c))
```

$2b=c+a$ was given.
Yielding the following assignments:

```
(setf c (- (* 2 b) a))
(setf a (- (* 2 b) c))
(setf b (+ (* 0.5 a) (* 0.5 c)))
```

$-2a+3b=d$ was derived from:
$2b=c+a$
$c+b=d+a$
Yielding the following assignments:

```
(setf d (+ (* 3 b) (* -2 a)))
(setf a (+ (* 1.5 b) (* -0.5 d)))
(setf b (+ (* 2/3 a) (* 1/3 d)))
```

$d-2c+b=0$ was derived from:
$2b=c+a$
$c+b=d+a$
Yielding the following assignments:

```
(setf d (- (* 2 c) b))
(setf c (+ (* 0.5 b) (* 0.5 d)))
(setf b (- (* 2 c) d))
```

‡

$2d-3c+a=0$ was derived from:
$2b=c+a$
$c+b=d+a$
Yielding the following assignments:

```
(setf d (+ (* -0.5 a) (* 1.5 c)))
(setf c (+ (* 1/3 a) (* 2/3 d)))
(setf a (+ (* 3 c) (* -2 d)))
```

The following solutions will be eliminated as being redundant. They were all derived from $c+b=d+a$. The rationale is simply that for the last assignment, for example, what use is being able to derive d from b, c and a when it can be derived from b and c alone? See ‡ above.

```
(setf a (- (+ b c) d))
(setf b (- (+ a d) c))
(setf c (- (+ a d) b))
(setf d (- (+ b c) a))
```

Given this set of assignment statements, MAGRITTE merges those with identical input sets into single clauses, wraps them up and produces the final primitive constraint definition:

```
(defprimc fourspaced ((a scalar) (b scalar) (c scalar) (d scalar))
    (and (equal (+ c b) (+ d a))
         (equal (* 2 b) (+ c a)))
    (((a b) (d c))
     (progn (setf b (- (* 2 c) d))
            (setf a (+ (* 3 c) (* -2 d)))))
    (((c a) (d b))
     (progn (setf c (+ (* 0.5 b) (* 0.5 d)))
            (setf a (+ (* 1.5 b) (* -0.5 d)))))
    (((d a) (c b))
     (progn (setf d (- (* 2 c) b))
            (setf a (- (* 2 b) c))))
    (((c b) (d a))
     (progn (setf c (+ (* 1/3 a) (* 2/3 d)))
            (setf b (+ (* 2/3 a) (* 1/3 d)))))
    (((d b) (c a))
     (progn (setf d (+ (* -0.5 a) (* 1.5 c)))
            (setf b (+ (* 0.5 a) (* 0.5 c)))))
    (((d c) (a b))
     (progn (setf d (+ (* 3 b) (* -2 a)))
            (setf c (- (* 2 b) a)))))
```

If, instead of the definition presented at the beginning of this example, the following had been used, the final constraint would have been exactly the same:

```
(defalgcon fourspaced (a b c d)
    (equal (* 2.0 b) (+ c a))
    (equal (* 2.0 c) (+ d b))
)
```

This next example shows the complexity that arises when there are many parameters to a constraint and small subsets of the parameters can derive other small subsets of the parameters:

```
(defalgcon redun (a b c d t1 t2)
    (equal t1 t2)
    (equal a (+ b t1))
    (equal c (+ d t2))
)
```

It corresponds to the following equations:

$$t2 = t1$$
$$b + t1 = a$$
$$d + t2 = c$$

Since there are no internal variables, simplification changes nothing.

The process of finding assignments generates:

$d+t2=c$ was given.
Yielding the following assignments:

```
(setf d (- c t2))
(setf c (+ t2 d))
(setf t2 (- c d))
```

$b+t1=a$ was given.
Yielding the following assignments:

```
(setf a (+ t1 b))
(setf b (- a t1))
(setf t1 (- a b))
```

$t2=t1$ was given.
Yielding the following assignments:

```
(setf t1 t2)
(setf t2 t1)
```

$b+t2=a$ was derived from:
$b+t1=a$
$t2=t1$
Yielding the following assignments:

```
(setf a (+ t2 b))
(setf b (- a t2))
(setf t2 (- a b))
```

$c=d+t1$ was derived from:
$d+t2=c$
$t2=t1$
Yielding the following assignments:

```
(setf d (- c t1))
(setf c (+ t1 d))
(setf t1 (- c d))
```

$c+b=d+a$ was derived from:
$d+t2=c$
$b+t1=a$
$t2=t1$
Yielding the following assignments:

```
(setf d (- (+ b c) a))
(setf c (- (+ a d) b))
(setf a (- (+ b c) d))
(setf b (- (+ a d) c))
```

None of these can be eliminated as redundant.

The final constraint is:

```
(defprimc redun
          ((a scalar) (b scalar) (c scalar) (d scalar)
                                          (t1 scalar) (t2 scalar))
          (and (equal (+ d t2) c)
               (equal (+ b t1) a)
               (equal t2 t1))
          (((a t1 t2) (d c b))
           (progn (setf t2 (- c d))
                  (setf t1 (- c d))
                  (setf a (- (+ b c) d))))
          (((a t1) (b t2))
           (progn (setf t1 t2)
                  (setf a (+ t2 b))))
          (((a t2) (b t1))
           (progn (setf t2 t1)
                  (setf a (+ t1 b))))
          (((b t1 t2) (d c a))
           (progn (setf t2 (- c d))
                  (setf t1 (- c d))
                  (setf b (- (+ a d) c))))
          (((b t1) (a t2))
           (progn (setf t1 t2)
                  (setf b (- a t2))))
          (((b t2) (a t1))
           (progn (setf t2 t1)
                  (setf b (- a t1))))
          (((c t1 t2) (d a b))
           (progn (setf t2 (- a b))
                  (setf t1 (- a b))
                  (setf c (- (+ a d) b))))
          (((c t2) (d t1))
           (progn (setf t2 t1)
                  (setf c (+ t1 d))))
          (((c t1) (d t2))
           (progn (setf t1 t2)
                  (setf c (+ t2 d))))
          (((d t1 t2) (c a b))
           (progn (setf t2 (- a b))
                  (setf t1 (- a b))
                  (setf d (- (+ b c) a))))
          (((d t2) (c t1))
           (progn (setf t2 t1)
                  (setf d (- c t1))))
          (((d t1) (c t2))
           (progn (setf t1 t2)
                  (setf d (- c t2))))
          (((t1 t2) (d c))
           (progn (setf t2 (- c d))
                  (setf t1 (- c d))))
          (((t1) (t2)) (setf t1 t2))
          (((t1 t2) (a b))
           (progn (setf t2 (- a b))
                  (setf t1 (- a b))))
          (((t2) (t1)) (setf t2 t1)))
```

In this case the transformation process has taken a trivial set of equations and blown them up into a very complicated routine. If one were to have taken the original three equations and created three separate constraints from them, the resulting net would have been simpler and still been solvable by propagation since there are no circularities. In section IV.2.1 a

different form of primitive constraint is given along with a corresponding transformation algorithm that generates much simpler constraints.

This last example is much simpler than most of the preceding ones. It demonstrates some slightly more complicated algebra by constraining the point (x_1, y_1) to be r units from (x_2, y_2). It could have come from the constraint graph of figure IV-3:

```
(defalgcon dist (x1 y1 x2 y2 r)
    (equal dx (- x1 x2))
    (equal dy (- y1 y2))
    (equal (* r r)
           (+ (* dx dx) (* dy dy)))
)
```

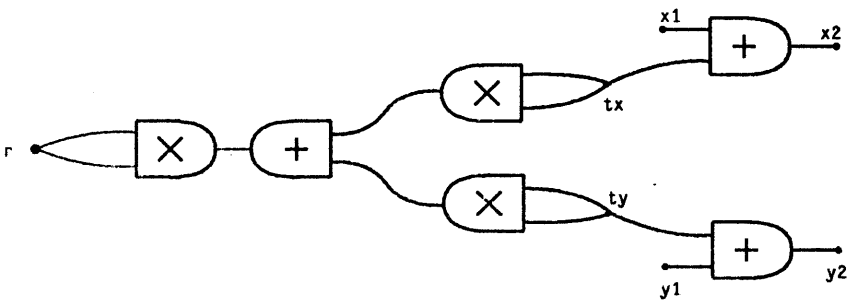


Figure IV-3: Constraining the distance between two points.

This is equivalent to:

$$x_1 = x_2 + dx$$

$$y_1 = dy + y_2$$

$$dy^2 + dx^2 = r^2$$

MAGRITTE eliminates the internal variables dx and dy, resulting in:

$$x_1^2 + y_1^2 - 2x_1x_2 + x_2^2 - 2y_1y_2 + y_2^2 = r^2$$

The deductions from FindAllAssignments are straightforward, and the following definition is produced:[6]

[6]This is the actual output of MAGRITTE, reformatted to be more legible.

(defprimc dist-cons ((x1 scalar) (y1 scalar) (x2 scalar) (y2 scalar)
(r scalar))

$$x_1^2+y_1^2-2x_1x_2+x_2^2-2y_1y_2+y_2^2=r^2$$

(((r) (x_1 y_1 x_2 y_2))
($r := (y_2^2-2y_1y_2+x_2^2-2x_1x_2+y_1^2+x_1^2)^{1/2}$)
((x_1) (r y_1 x_2 y_2))
($x_1 := 0.5(-4y_2^2+8y_1y_2-4y_1^2+4r^2)^{1/2}+x_2$)
((x_2) (r x_1 y_1 y_2))
($x_2 := 0.5(-4y_2^2+8y_1y_2-4y_1^2+4r^2)^{1/2}+x_1$)
((y_1) (r x_1 x_2 y_2))
($y_1 := 0.5(-4x_2^2+8x_1x_2-4x_1^2+4r^2)^{1/2}+y_2$)
((y_2) (r x_1 y_1 x_2))
($y_2 := 0.5(-4x_2^2+8x_1x_2-4x_1^2+4r^2)^{1/2}+y_1$))

IV.2. Reducing the Complexity of Transformed Constraints

The example of the transformation of a loosely coupled network that appears on page 64 shows the exponential blowup that can result. In other cases substantial simplification can result, as is seen in the distance constraint example on page 67. The simplifications result from the elimination of internal variables and from the tightly coupled relationships among the parameters. In the first example, transformation was unnecessary. Normal propagation could have dealt with it easily if each equation were implemented as a separate constraint. The only improvement came from the speedup of applying fewer constraints. In the second, transformation was essential in order to deal with the circularities in the multiplications used for exponentiation. It also helped by hiding the internal variables. Transformation does best where propagation does worst, and transformation does worst where propagation does best. In this section techniques will be shown that blend the advantages of both.

IV.2.1. Primitive Constraints as Subnets

To exploit these differences and strengths, MAGRITTE uses a primitive constraint definition that is more like a constraint network. A primitive constraint contains not a set of clauses, but a set of sets of clauses. Each set of clauses is, in a sense, a primitive constraint on its own.

For example, the constraint on page 64 could be expressed as follows:

```
(defprimc redun
    ((a scalar) (b scalar) (c scalar) (d scalar)
                                        (t1 scalar) (t2 scalar))

    (and (equal (+ b t1) a)
         (equal (+ d t2) c)
         (equal t2 t1))

    part (equal t2 t1)
    ((t1 (t2)) (setf t1 t2))
    ((t2 (t1)) (setf t2 t1))

    part (equal (+ d t2) c)
    ((d (c t2)) (setf d (- c t2)))
    ((c (d t2)) (setf c (+ t2 d)))
    ((t2 (d c)) (setf t2 (- c d)))

    part (equal (+ b t1) a)
    ((a (b t1)) (setf a (+ t1 b)))
    ((b (a t1)) (setf b (- a t1)))
    ((t1 (a b)) (setf t1 (- a b))))
```

Each set of rules between **part** keywords acts as a constraint on its own. The expression at the head of each **part** defines the part of the constraint dealt with by the following clauses. Firing a constraint is no longer a matter of picking the *single* clause that applies and executing its body. The firing process is now a loop: pick a clause, execute it, then pick a clause again. This continues until no more clauses can be applied. The clauses must be chosen carefully so that a sequence of clause applications exists that will satisfy the entire constraint.

Multipart primitive constraints aren't the only alternate form that could be used. The original primitive form along with the macro constraints of section II.4 on page 16. If this were done each **part** would become a primitive constraint and the entire constraint would be a macro constraint that invoked the primitive parts. This exposes an opportunity to use common subparts. However, in an attempt to keep the data structures more compact, this form was not used.

IV.2.2. Transforming to this Form

We can generate primitive constraints in this new form by using a slight modification of the previous transformation algorithm. Refer back to the calculation of TotalSolution from section IV.1.5, page 58. Take each set of clauses generated by a call to AddAssignment, separate them with **part** keywords and we have a primitive constraint of the new form.

This new constraint will be applicable in exactly the same situations as the old. Iterative firing replaces the merging of section IV.1.6, page 61. We're trading compile time work for run time work and a complexity reduction.

A brief recap: *Clauses* are generated from equations. An equation relates the set of variables that appear in it. A clause is generated for every distinct variable that appears in an equation. Two equations are composed by picking a variable that they have in common, solving for it in one and substituting the result in the other.

Now, let's try to reduce the number of clauses in the primitive constraint. Suppose two equations are composed that have only one variable in common. The clauses that result from the new equation will be able to compute no more than the clauses from the original two. Hence, they are redundant and may be eliminated. Why? The new equation can deduce something only when all but one of its variables are known: it will deduce a value for the one unknown variable. This one unknown variable will come from one of the two original equations. The other original equation is thereby able to deduce the variable that they have in common, assuming that it is unknown. After this, the one equation is able to deduce the unknown value. One can think of the iteration of clause application as being equivalent to algebraic composition in some cases.

By a similar chain of reasoning, if the set of variables appearing in one equation is a superset of those appearing in another, then it and its derived clauses may be omitted. It and the smaller equation must have been algebraically composed; the result of the composition and the smaller one supplant the larger one. One must also be careful to avoid circular elimination relationships: *a* can be eliminated because of *b* and *b* can be eliminated because of *a*.

Here is the derivation of the redundant constraint example of section IV.1.7, page 64, using the new form. Given this definition:

```
(defalgcon redun (a b c d t1 t2)
    (equal t1 t2)
    (equal a (+ b t1))
    (equal c (+ d t2))
)
```

It corresponds to these equations:

$t2 = t1$
$d + t2 = c$
$b + t1 = a$

MAGRITTE performs the following derivations:

$b + t1 = a$ was given.
Yielding the following assignments to a, b and $t1$
$a \leftarrow t1 + b$
$b \leftarrow a - t1$
$t1 \leftarrow a \neg b$

$d + t2 = c$ was given.
Yielding the following assignments to d, c and $t2$
$d \leftarrow c - t2$
$c \leftarrow t2 + d$
$t2 \leftarrow c - d$

$t2 = t1$ was given.
Yielding the following assignments to $t1$ and $t2$
$t1 \leftarrow t2$
$t2 \leftarrow t1$

$d + t1 = c$ derived from $d + t2 = c$ using:
$t2 \leftarrow t1$
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

$c + b = d + a$ derived from $b + t1 = a$ using:
$t1 \leftarrow c - d$
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

$b + t2 = a$ derived from $b + t1 = a$ using:
$t1 \leftarrow t2$
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

In this case, none of the derivations generates anything useful. The con-

straint that is finally built contains no more than the assignments that are generated from the initial equations.

The following example is slightly more complicated. It generates both useful and useless clauses. Think of it as constraining the *y* coordinates of the five points on a figure shaped like an A. *Vertex* is the vertex of the A, *LeftMid* and *RightMid* are the endpoints of the crossbar, and *LeftBot* and *RightBot* are the feet. The crossbar is horizontal and halfway along the length of the A. Here is the definition:

```
(defalgcon A-shape (LeftBot LeftMid vertex RightMid RightBot)
    (= LeftMid (+ LeftBot T1))
    (= vertex (+ LeftMid T1))
    (= RightMid (+ RightBot T2))
    (= vertex (+ RightMid T2))
    (= LeftMid RightMid))
```

MAGRITTE simplifies this to:

vertex + *RightBot* = 2**RightMid*
2**LeftMid* = *LeftBot* + *vertex*
RightMid = *LeftMid*

Then the following deductions are done:

vertex + *RightBot* = 2**RightMid* was given.
Yielding the following assignments to *vertex*, *RightMid* and *RightBot*
vertex ← 2**RightMid* − *RightBot*
RightMid ← 0.5**RightBot* + 0.5**vertex*
RightBot ← 2**RightMid* − *vertex*

2**LeftMid* = *LeftBot* + *vertex* was given.
Yielding the following assignments to *LeftBot*, *LeftMid* and *vertex*
LeftBot ← 2**LeftMid* − *vertex*
LeftMid ← 0.5**vertex* + 0.5**LeftBot*
vertex ← 2**LeftMid* − *LeftBot*

RightMid = *LeftMid* was given.
Yielding the following assignments to *LeftMid* and *RightMid*
LeftMid ← *RightMid*
RightMid ← *LeftMid*

2**RightMid* = *LeftBot* + *vertex* derived from 2**LeftMid* = *LeftBot* + *vertex* using:
LeftMid ← *RightMid*
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

RightBot = *LeftBot* derived from
vertex + *RightBot* = 2**RightMid* using:
vertex ← 2**RightMid* − *LeftBot*
Yielding the following assignments to *LeftBot* and *RightBot*
LeftBot ← *RightBot*
RightBot ← *LeftBot*

2**LeftMid* + *RightBot* = *LeftBot* + 2**RightMid* derived from
vertex + *RightBot* = 2**RightMid* using:
vertex ← 2**LeftMid* − *LeftBot*
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

vertex + *RightBot* = 2**LeftMid* derived from
vertex + *RightBot* = 2**RightMid* using:
RightMid ← *LeftMid*
Since it was derived by a one-variable deduction, its assignments won't be added to the final constraint.

The primitive constraint that results is:

```
(defprimc A-shape ((LeftBot scalar)(LeftMid scalar)(vertex scalar)
                   (RightMid scalar) (RightBot scalar))
    (and (equal (+ vertex RightBot)
                (* 2 RightMid))
         (equal (* 2 LeftMid)
                (+ LeftBot vertex))
         (equal RightMid LeftMid))

    part (equal RightMid LeftMid)
    ((LeftMid (RightMid)) (setf LeftMid RightMid))
    ((RightMid (LeftMid)) (setf RightMid LeftMid))

    part (equal (* 2 LeftMid)
                (+ LeftBot vertex))
    ((LeftBot (LeftMid vertex))
     (setf LeftBot (- (* 2 LeftMid) vertex)))
    ((LeftMid (LeftBot vertex))
     (setf LeftMid
           (+ (* 0.5 vertex) (* 0.5 LeftBot))))
    ((vertex (LeftBot LeftMid))
     (setf vertex (- (* 2 LeftMid) LeftBot)))

    part (equal (+ vertex RightBot)
                (* 2 RightMid))
    ((vertex (RightMid RightBot))
     (setf vertex
           (- (* 2 RightMid) RightBot)))
    ((RightMid (vertex RightBot))
     (setf RightMid
           (+ (* 0.5 RightBot) (* 0.5 vertex))))
    ((RightBot (vertex RightMid))
     (setf RightBot
           (- (* 2 RightMid) vertex)))

    part (equal RightBot LeftBot)
    ((LeftBot (RightBot)) (setf LeftBot RightBot))
    ((RightBot (LeftBot)) (setf RightBot LeftBot)))
```

Here, all of the deductions except *RightBot*= *LeftBot* were ignored. It is the only one that cannot be duplicated by the iterative application of the clauses generated from other rules. The algebraic magic occurs when *vertex* is eliminated from *vertex*+ *RightBot*=2**RightMid* by substituting *vertex* ← 2**RightMid*− *LeftBot.* Not only is *vertex* eliminated, but *RightMid* cancels and disappears too. This has happened because of the circular relationship among the expressions as linked by their common variables.

IV.3. When to use Transformation

Constraint network transformation may be applied for a variety of reasons in a variety of situations. It may used to increase performance, make a network tractable, and it may be used to provide a better notation for defining primitive constraints.

IV.3.1. Performance

Transformed constraint networks provide a performance advantage because they replace a set of constraints with a single constraint whose execution time is almost always substantially less. In a sense, what transformation does is to abstract from a network its external properties as visible to the outside world. The internal details of the operation of the constraint are completely hidden. This abstraction becomes particularly powerful when applied hierarchically.

A powerful circumstance for the use of transformations occurs with user-defined cells. In a design system, users will typically be able to define cells that they can invoke repeatedly. These cells usually have very few parameters in relation to the size of their internal state. For example, in the specification of a VLSI shift register a single cell will have many internal variables representing the coordinates of the various parts but all that matters to the outside world is the relationships among the externally visible points: the input, output, clock, power and ground. If a row of such cells is laid down then the constraint system composed of the transformed nets is much simpler than the one obtained without performing the transformation. When it is necessary to finally expand the design and lay out the internals of the entire shift register the solution obtained from the transformed instances can provide guidance for laying out each subcell as an independent constraint net.

IV.3.2. Tractability

The original motivation for applying transformations was to improve the tractability of the satisfaction process when a propagation algorithm is used. Chapter III discusses satisfaction and situations where satisfaction can fail. An example is shown in section III.1.2 on page 21.

The key observation is that propagation is a local technique, but cyclic dependencies may require that the network be examined at a more global level. Transformation locally alters the topology of the network and this

altered topology is amenable to a propagation algorithm. A global technique like relaxation will work, but it is applied each time a value changes. The operation of the algorithm depends both on the constraint and value instances. Transformation, on the other hand, is only sensitive to constraint instances and not to the values. Transformation may have to be reapplied when the shape of the network changes, but not when values change. It incurs a cost only when something new is seen, but the cost of relaxation must be paid over and over. Transformations can be invoked automatically by satisfaction algorithms when they encounter problems. This is discussed in section IV.5 on page 79.

IV.3.3. Notation

An almost serendipitous offshoot of the ability to perform transformations is the ability to define constraints in terms of conventional algebra. In systems like THINGLAB the definition of primitive constraints is complicated and error-prone. They require the redundant specification of a rule and all of the propagation clauses. MAGRITTE allows just the rule to be specified and can derive the propagation clauses automatically. For example, some systems might require a specification of the *sum* constraint that looks like what appears in Figure II-1 on page 16. In MAGRITTE, one need only write:

```
(defalgcon sum (result a b)
    (= result (+ a b)))
```

IV.4. Avoiding Transformations

While considerable care has been taken to make the constraint net transformation algorithm as inexpensive as possible, it is still not cheap. One of the easiest ways of making an algorithm run faster is not to run it at all. Transformation algorithm should be used only when really necessary. There are two ways of doing this that are employed in the MAGRITTE system: detecting situations that have been seen before and detecting situations where propagation can work without the aid of a transformation.

Situations that have been seen before are detected by the use of a high performance graph isomorphism algorithm. Each time a net is transformed the net and the resultant constraint are saved in a definition library. Then when a net is found that needs to be transformed, a quick

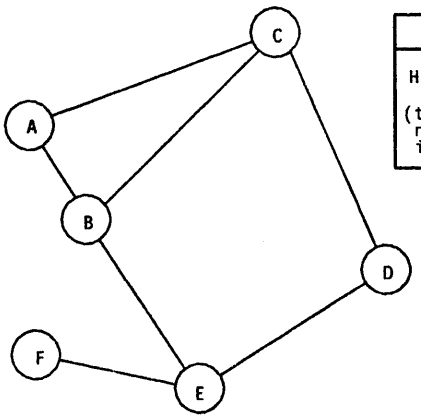
check is made to see if a network that is isomorphic to this one has already been seen. If it has, then it is used. Otherwise the transformation is carried out and the new definition is added to the library. Doing graph isomorphism checking quickly is not an obvious process. The following section is a pragmatic presentation of the algorithm; theoretical details can be found in Read [14].

Transformations can be triggered in a number of ways: the instant a circularity is created in the network, whenever a network is unsatisfiable without transformation, or at some intermediate point. It is important to use the simplest and fastest algorithm whenever possible and it is equally important to notice when it fails and then, and only then, apply a more powerful method. All of the satisfaction algorithms discussed in chapter III can easily be adapted to invoke transformation automatically if they fail. The breadth-first algorithm can be adapted to invoke transformation if it appears that its simple propagation solution will be too complicated. Later sections discuss the links between satisfaction and transformation in greater detail.

IV.4.1. Graph Isomorphism

The key part of this graph isomorphism algorithm is the construction of isomorphism-independent hashes for the graph and for each node. Given these hash values very high speed searches and validations can be done.

Consider the construction of an isomorphism-independent hash value for each node of a graph. Given an isomorphic pair of graphs, nodes that map together under an isomorphism must have the same hash value, although having the same hash value does not necessarily mean that two nodes map together, unless the hash values of all nodes in both graphs are all distinct. Examples of such hash functions are the constant 1 or the degree of the node. Neither of these is particularly good, but they may be refined iteratively. This iterative refinement must be isomorphism-independent, just like the initial hash. A workable, but not particularly good, refinement function is to replace the hash value of a node with the sum of its hash value and the hash values of its neighbours. A much better refinement function would be to use some good conventional hashing function on the *sorted* set of adjacent hash values. Sorting isn't necessary, but the commutativity of the hashing function is. The sorting stage converts a non-commutative function into a commutative one.



Nodes	A	B	C	D	E	F
Hash values	2	3	3	2	3	1
(two refinement iterations)	8	11	10	8	9	4
	29	38	36	27	32	13

Figure IV-4: Evaluating an isomorphism-independent hash.

In figure IV-4 the first row shows the initial hash for each node. It is just the degree of the node. The two subsequent rows show refined hash values derived by summing adjacent elements from the previous row. Note that in the second row the hashes for nodes A and D are the same. If you look at the graph up to a distance of two arcs from A and two arcs from D, the two are isomorphic to each other. By the end of the second refinement iteration, the hash values for the nodes are all distinct. The advantage of distinct hash values is that when the time comes to validate the match of a hash, no searching is required: just take the nodes of the two graphs that have equal hashes and presume that they match in the isomorphism. There are complications when the graphs are symmetric.

The complete graph isomorphism algorithm is:

1. Hash each graph independently:
    a. Pick an initial hash for each node.
    b. Refine their hash values:
        i. For each node replace each hash value by some commutative hash involving it and the hash value of neighboring nodes, always using the values from the previous iteration.
        ii. Continue the refinement until either all values are distinct or some fixed number of iterations have passed.
    c. Compute a hash value for each graph by hashing together the hashes of their nodes using a commutative hash.

2. Check that the hash values of the two graphs match.
3. If they do, validate the comparison:
   a. Use a conventional exponential time brute force graph comparison, except that matches of node hash values are used for guidance.
   b. The guidance from the distinctness of the hash values usually causes the brute force match algorithm to run in linear time.

One of the outputs of this algorithm is a single number that is an isomorphism-independent hash of the graph. MAGRITTE uses this to construct a library of graphs that have been transformed. Each time a graph is transformed, it is added to the library. Then whenever another transformation is needed a very fast lookup can be done.

IV.5. Linking Satisfaction and Transformation

The transformation algorithms of this chapter become particularly useful if they can be linked to the satisfaction algorithms of the previous chapter. Satisfaction algorithms may fail in the face of a circularity in the network and transformation may provide the bridges necessary to get over them.

IV.5.1. Propagation

The basic propagation algorithm of section III.1.3 (page 22) can easily be modified to invoke transformation when it fails. If it fails after propagating as much as possible, it then picks some likely-looking cycle, transforms it, and tries again. This modification considers a cycle to look likely if it contains at least one constraint that is linked to one known and one unknown value. In detail:

1. Change the part of the algorithm that throws away constraints that couldn't fire because they didn't have enough adjacent values. Have it instead remember the constraint as unfireable and change that state if it ever does fire.
2. If the algorithm terminates and the list of unfireable constraints is non-empty then it means that there are cells that have not been assigned values and the system has not been satisfied. Each one of these constraints is attached to at least one known cell and one unknown cell.
3. Find the smallest cycle in the constraint graph that contains at least one of these unfired constraints. The cycle can be chosen in other ways, but this has proven to be effective.
4. Transform the constraints in it, replacing them by the result.
5. Resume propagation.

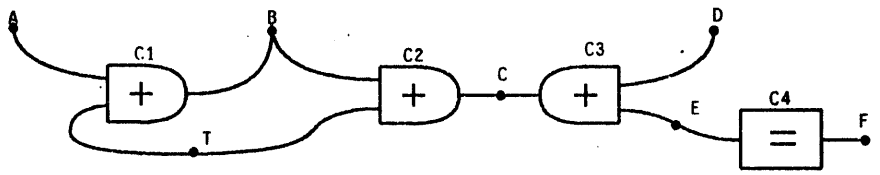


Figure IV-5: Linking transformation to propagation

Figure IV-5 revisits the midpoint constraint with some small frills added on. Here are the steps that the satisfaction algorithm will go through to satisfy the network, including the detection of a cycle that needs to be transformed:

1. Initially, **A**, **D** and **F** are known and *SuspectConstraints* contains **C1**, **C3** and **C4**.
2. **C3** is examined. It can deduce nothing, so it is remembered.
3. **C4** is examined, **E** can be deduced and is given the value 5. **C3** is put back into the *SuspectConstraints* set.
4. **C1** is examined and nothing can be deduced, so it is remembered.
5. **C3** is examined and the value 9 is deduced for **C**. **C2** is put into the *SuspectConstraints* set.
6. **C2** is examined but cannot fire and so is remembered.
7. The *SuspectConstraints* set is now empty, so no more propagation can be attempted. The unfireable set contains **C1** and **C2**.
8. The loop containing both of them is transformed and they are replaced by C12.
9. C12 is examined and deduces **B**.

IV.5.2. Breadth-First Planning

The breadth-first algorithm of section III.2.5 (page 49) can be modified similarly. The algorithm is modified so that loops detected in the section on page 51, marked with ‡, are remembered. Then, if the algorithm fails, a loop can be picked, transformed, and the algorithm tried again. One need not wait until the algorithm fails. Loops can be transformed as they are encountered, or they can be transformed if the prospects for success otherwise look bleak. For example, if the number of tendrils becomes large or the sequences become long, then a loop could be transformed and the algorithm retried.

This can best be understood by looking at an example. Consider a graphi-

cal editor where the user has drawn three points and constrained one to be the midpoint of the others. The constraint is expressed using the tried, true, and by now quite boring, pair of sum constraints.

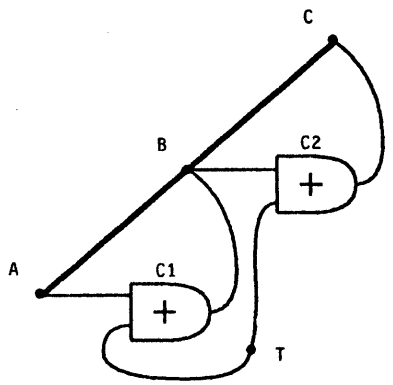
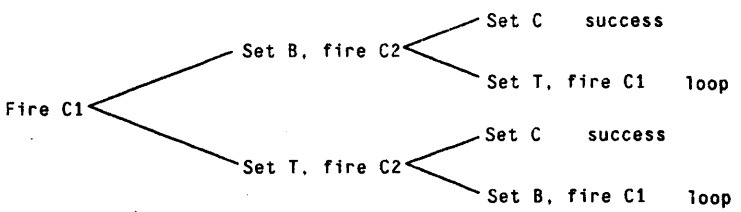


Figure IV-6: Three points in a row.

In figure IV-6, if the user moves A then four tendrils will be built in attempting to satisfy the system:



Two tendrils succeed and alter the same number of cells: one maintains the center point and moves the other end, the other maintains the relative displacements and moves both the center and the other end. If **C** had been a constant then neither of the successful tendrils would have been created, instead the algorithm would have terminated unsuccessfully having found a loop. It would be transformed and satisfaction tried again.

Now, suppose that **C** were rigidly connected to many other cells, as in figure IV-7. Moving **C** requires moving many other cells. If **A** is moved, breadth-first search will not consider moving **B** and leaving **C** alone. This would require transforming the loop, which can be avoided by moving **C** and its attached baggage. Moving all of the baggage, and the consequent loss of visual coherence since much will change, is probably less reasonable than attempting the transformation. The problem can be solved by

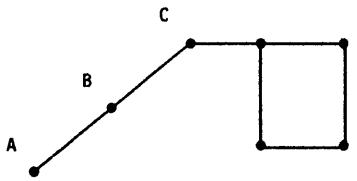


Figure IV-7: Midpoint with a hanging weight.

modifying breadth-first search to give up early if a loop has been seen. Having it try a little more after seeing the first loop cuts down on the number of transformations.

In one of MAGRITTE's many versions the breadth-first search maintained a list of the loops that it had encountered. Each time a loop was encountered the tendril that encountered it was thrown away but the loop was remembered. After the first loop was found and then some number of subsequent tendril extensions was performed, MAGRITTE would throw away all of the tendrils, pick one of the loops, transform it, and restart the breadth-first satisfaction.

This 'trying a little more' was done because transformation was believed to be expensive, search to be cheap, and solutions that avoided the loop to be common. All of these proved false. Transformation of small loops is very cheap, substantially cheaper than a flailing search. Early resolution of loops pays off substantially. The picture is not quite the same for larger loops, here there tend to be enough degrees of freedom that search is soon satisfied.

V Conclusion

The thesis of this dissertation is that constraints can be used effectively in practical interactive graphical layout systems. They have been discussed in more general terms, but emphasis has been placed on some of the special properties of such systems. The primary effect has been the pervasive pursuit of *locality* on the satisfaction process. The need to be *incremental* and do planning has also been important.

The dissertation contributes a broad analysis of several satisfaction algorithms and some novel algebraic transformation techniques.

Several satisfaction algorithms were presented. They ranged from simple to complex:

- Local propagation is simple and fast, but it often fails and needs global planning direction.
- Relaxation, and other classical methods, are general but slow and global. They provide a recourse of last resort.
- The techniques given in MUMBLE and EARL are simple, but they clearly show the necessity and advantage of exploiting the special properties of the domain.
- Unplanned firing is simple and fast, being essentially the same as local propagation. It substitutes dice rolling for sophisticated planning. It often fails or produces a 'surprising' answer. The special case of networks consisting only of forced deductions is powerful.
- Retraction is an elegant planning technique for local propagation. Unfortunately, it is prone to failure and blind to many possible solutions.
- Breadth-first search is reasonably simple, at least conceptually, and general. Even though it is local, its performance can be poor unless the branching factor is bounded. It also tends to eliminate 'surprising' results.

To cope with some of the problems encountered in satisfaction, algorithms for transforming networks of constraints into primitive constraints were developed:

- Networks of algebraic constraints can easily be transformed to primitive constraints where constraint definitions require a complete enumeration of all the possible propagation cases.
- An exponential size blowup can result. To counter this, constraints can be defined as networks themselves.
- Networks can also be transformed easily to this type of constraint definition.
- Transformation can be triggered automatically by satisfaction failure. The subnets to be transformed are chosen from among the cycles adjacent to the failure.
- Repeated transformation of the same subnet can be avoided efficiently by employing a graph isomorphism algorithm.
- Besides making networks tractable with simpler satisfaction algorithms, transformation yields performance improvements.
- They also yield a more appealing and robust notation for defining primitive constraints.

These have proven to be efficient and powerful. Stopping to transform a small subnet does not slow down the satisfaction process much, but speeds up subsequent satisfactions tremendously. Since these are local transformations that are independent of the values of variables, they can usually be retained for a long time.

The next step after this dissertation is the engineering of a useful layout system that incorporates these principles. MAGRITTE is really only a testbed; it is not directly useful itself. Most of its performance problems stem from being based on a slow LISP system and on some expediencies in the implementation (for example, instead of doing sophisticated table lookups it often does linear searches). The penalty for good performance and generality is substantial complexity.

MAGRITTE does not use relaxation. The techniques developed here do not supplant relaxation, they merely reduce the number of situations where it is needed. MAGRITTE avoids it since it is a well-understood technique. A real satisfaction system would have the following structure:

- It would first use breadth-first propagation.
- If a small loop is found, it would be removed immediately by using previous solutions to the same problem or by transformation if it hasn't been seen before.
- If a large loop is found, remember the fact but try to ignore it.
- If problems persist use incremental relaxation in a clipped subregion. Such regions should be marked so that if propagation encounters them again it uses relaxation more readily.

As a simplification, a system that employs only propagation could be

built. If it can't deal with a situation, it could just throw up its hands in despair. This may seem like a brutal thing to do, but it's likely to be surprisingly powerful. The many successful spreadsheet calculators available are nothing more than constraint systems with very simple satisfaction algorithms. A similar satisfaction algorithm in a graphical domain would be easy to implement.

Many questions are left unanswered or are suggested by this thesis. It is up to future works to deal with them. Among them are:

Interfacing relaxation to propagation and transformation: Knowing when and where to relax rather than transform or propagate — can one simply use relaxation on cyclic regions beyond a certain size? Can hints be maintained that make subsequent decisions to relax easier?

Automatic construction of error equations: Relaxation algorithms need to be able to evaluate the error introduced into the system by each constraint and the values of the objects it constrains. Can these error functions be constructed automatically given the rule to test for satisfaction that MAGRITTE uses? This is a more subtle problem than it at first appears because of the numerical properties of relaxation. For example, given the definition of the **sum** constraint that appears in section IV.3.3 on page 76 can the error function be constructed automatically? An obvious candidate is `(- (+ a b) result)`, is this good enough?

An incremental version of Sutherland's ordering: The ordering algorithm that Sutherland used in SKETCHPAD is quite attractive, it provides a static plan that can be reused for several satisfaction attempts. It is invalidated whenever a new constraint is added to the system. Is there a way to incrementally adjust the ordering when a new constraint is added to the net in order to take the new constraint into account? An important point to consider is the changing of the value of some object by the user, for example, moving the endpoint of a line. Is this equivalent to constraining the object to be equal to the new value? How does it affect the ordering?

Transformation with inequalities: MAGRITTE deals with inequalities ($\leq$, $\neq$, ...) during propagation but can not deal with them when transforming. Are the extensions to the transformation process merely a matter of augmenting the algebraic engine? In some sense, deductions based on inequalities are less powerful than those based on equality: they don't provide a tight a restriction on the set of acceptable values.

Meta-constraints: one can think of a constraint network as a large boolean expression which is the *and* of the test rules from all of the contained constraints. What are the implications of letting a constraint network be an arbitrary boolean expression that includes *or* and *complement*? This is necessary if, for example, you want to constrain two rectangles to *not* overlap. At first glance this appears to be simple when breadth-first search is used, although it does provide yet more branches in the search for a solution. Can the transformation techniques be extended to cover it?

Lazy evaluation: When MAGRITTE transforms a subnet it finds solutions for propagation in all possible directions. For example, if you have a geometric figure, the constraint needs to be able to cope with the user moving any point on that figure. This means that for each point it needs to know how changes to the value of that point affect the values of all the other points. When transforming such a net MAGRITTE can generate huge constraints, although it will do a fair amount of simplification. One form of simplification that appears attractive that wasn't tried is lazy evaluation. That is, lazy evaluation of the *construction* of the primitive constraint.

Domain-specific planning: The planning done by MAGRITTE does not exploit the fact that all of the networks it deals with concern graphical objects. This is good in terms of the generality that it provides, but is there some domain specific information that could be used to aid planning? For example, one of the most common changes that is made in a drawing is the moving of a point — a translation of its coordinates. It is fairly easy to detect constraints that are independent of translation, that will continue to hold so long as all of their constrained points are similarly translated. Networks of these translation independent constraints, and their constrained points, form rigid compound objects that can be moved as a group to compensate for the translation of one of the contained points.

Implicit constraints: There are many implicit constraints that one might want a constraint system to handle. When using MAGRITTE one often wishes it had the implicit constraint that "points that are not explicitly constrained to be equal are implicitly not equal". This would avoid the black-hole syndrome — many of the networks that have been worked with have had a degenerate solution where all the points get superimposed, MAGRITTE often finds these. What is the best way to deal with these? Should the satisfaction algorithms be altered or should explicit

constraints automatically be constructed? If there are n points in a drawing, automatic construction of these constraints would produce n^2 new constraints.

I MAGRITTE 's Algebraic Engine

This appendix describes the algebraic manipulation engine upon which MAGRITTE is based. It is very simple, certainly much simpler than MACSYMA. But that is precisely the point: powerful results can be had by applying even a very simple amount of algebraic knowledge.

At its heart there is a mechanism for maintaining a canonical representation. Expressions are always reduced to a sum of products of factors. Factors can be variables, constants, or exponentiated expressions. Expressions are kept as hashed, sorted lists so that comparisons are fast and so that like terms and factors are adjacent. All exponents are restricted to being constants. The following transformations are used:

$$
\begin{aligned}
-e &\Rightarrow -1\cdot e \\
e_1-e_2 &\Rightarrow e_1+(-1\cdot e_2) \\
e_1+e_2 &\Rightarrow e_1\cdot e_1^{-1} \\
e_1\cdot(e_2+e_3) &\Rightarrow e_1\cdot e_2+e_1\cdot e_3 \\
e\cdot 1 &\Rightarrow e \\
e+0 &\Rightarrow e \\
e\cdot 0 &\Rightarrow 0 \\
k_1\cdot e+k_2\cdot e &\Rightarrow (k_1+k_2)\cdot e \\
e^1 &\Rightarrow e \\
e^0 &\Rightarrow 1 \\
(e_1\cdot e_2)^{e_3} &\Rightarrow e_1^{e_3}\cdot e_2^{e_3} \\
e_1^{e_2}\cdot e_1^{e_3} &\Rightarrow e_1^{e_2+e_3} \\
e_1^2 &\Rightarrow e_1\cdot e_1 \\
e_1^3 &\Rightarrow e_1\cdot e_1\cdot e_1 \\
e_1=e_2 &\Rightarrow e_1-e_2=0 \\
e_1\wedge e_1 &\Rightarrow e_1 \\
e_1\vee e_1 &\Rightarrow e_1
\end{aligned}
$$

The **SolveFor** routine used in transformations is quite simpleminded. Given an equation in canonical form, $e=0$, all terms containing the target variable are collected on one side of the equality, and the rest on the other side. The terms are then grouped according to the exponent of the target variable, those with like exponents being together. If there is only one group (ie... all occurences of the target variable have the same exponent) then the obvious factoring out, division and exponentiation are performed. If there are two groups and the exponent of the variable in one is twice the exponent in the other, then it's a simple quadratic polynomial and is solved as such. If neither of these cases holds, MAGRITTE gives up. More complicated forms could be dealt with, but they were never needed.

Along with this algebraic manipulation facility is one for converting between LISP expressions and MAGRITTE's canonical form. The transformation process picks apart the LISP functions that define each primitive constraint, extracts the test expressions, and joins them together to form the system of equations represented by the constraints as a group. Once the transformation has been completed, the various derived expressions are converted back to LISP notation and pieced together as a new function. Thus the new function performs as well as any hand-written LISP function.

References

1. Borning, Alan, THINGLAB -- *A Constraint Oriented Simulation Laboratory*, PhD dissertation, Stanford, July 1979.

2. Dahl, O-J., Myhrhaug, B., and Nygaard, K., Norwegian Computing Center. *SIMULA Common Base Language*, 1970.

3. Dahlquist, Bjorck and Anderson, *Numerical Methods*, Prentice-Hall, 1969.

4. Dennis, J.B., Leung, C.K.C., and Misunas, D.P., 'A highly parallel processor using a data-flow machine language', Tech. Rep.CSG 134-1, MIT, Lab. for Computer Science, June 1979.

5. U. S. Department of Defense, *Reference Manual for the ADA Programming Language*, July, 1980.

6. Foderaro, J. K., Berkeley, *The Franz Lisp Manual*, 1980.

7. Goldberg, A., and Kay, A., 'Smalltalk-72 Instruction Manual', Tech. Rep., Learning Research Group, Xerox Palo Alto Research Center, Palo Alto, California, March 1976.

8. Gosling, J., *The Cm* Multiprocessor Project: A Research Review*, Carnegie-Mellon University Computer Science Department, 1980, The Mumble Microcode Compiler.

9. Johnson, S., 'The *i* VLSI Design system', personal communication.

10. Kazar, M., *The Cm* Multiprocessor Project: A Research Review*, Carnegie-Mellon University Computer Science Department, 1980, The Echoes Operating System.

11. Kingsley, C., *Earl: a language for integrated circut design*, PhD dissertation, California Institute of Technology, 1981.

12. Knuth, D. K., *Fundamental Algorithms*, Addison-Wesley, The Art of Computer Programming, Vol. 1, 1969.

13. The Mathlab Group, Laboratory for Computer Science, MIT, *Macsyma Reference Manual*, 9 ed., 1978.

14. Read, R. C. and Corneil, D. G., 'The Graph Isomorphism Disease', *Journal of Graph Theory*, 1(1977), 339-363.

15. Stallman, R. M., Sussman, G. J., 'Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis', *Artificial Intelligence*, **9**(1977), 135-196.

16. Steele, G. L. Jr., *The Definition and Implementation of a Computer Programming Language Based on Constraints*, PhD dissertation, MIT, August 1980.

17. Steele, G. L. Jr., and Sussman, G. J., 'Constraints', Tech. Rep., MIT AI memo 502, November 1978.

18. Sutherland, I. E., *SKETCHPAD: A Man-Machine Graphical communication system*, PhD dissertation, MIT, January 1963.

19. Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P., 'Data-Driven and Demand-Driven Computer Architecture', *Computing Surveys*, **14**, 1 (March 1982), 93-144.

20. Van Wyk, C. J., *A Language for Typesetting Graphics*, PhD dissertation, Stanford, June 1980.

21. Weinreb, D., and Moon, D., *LISP Machine Manual, Fourth Edition*, Massachusetts Institute of Technology Artificial Intelligence Laboratory, July 1981.