DEPARTMENT OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

DAI Working Paper No. 210
Date: November 1987

Details of A Network Engine for Algebraic and Geometric Reasoning

Robert Fisher

Abstract:

This document gives additional details on the network engine for algebraic and geometric reasoning described in [Fisher 1987a, Fisher 1987b, Fisher and Orr 1987b]. The network is useful for simultaneously bounding possible values of variables linked by linear and non-linear equations and inequalities. This paper describes: the structure of the network elements, details of compilation including optimization and conditional techniques, how parallel activity is simulated, deadlock avoidance through default evaluations, module evaluation, operation definition, including over infinite values, heuristic bounds in the geometric modules, avoidance of timing problems on conditional constraints and some proposed extensions.

Acknowledgements

Funding for this paper was under Alvey grant GR/D/1740.3. The ideas in this paper benefited greatly from discussions with J. Aylett and M. Orr.

1.0 Introduction

We have been using a network implementation [Fisher 1987a, Fisher 1987b, Fisher and Orr 1987b] of the SUP-INF method [Bledsoe 1974, Shostak 1977, Brooks 1981, Davis 1987] for solving sets of inequalities that has advantages over previous implementations. First, the cost of symbolic manipulation is transferred to compile-time allowing speed up at run-time because of parallel evaluation (though we only simulate parallel evaluation here). Further, allowing iteration in the network improves the competence of the method when working with non-linear expressions.

The use of this engine has been motivated by some work on geometrical reasoning [Fisher and Orr 1987a, Orr 1987, Orr and Fisher 1987], which has concluded that algebraic inequalities are a preferential representation for expressing geometrical relationships. Hence, efficient solution of sets of algebraic constraints is desirable.

Further, analysis of typical geometrical relationships demonstrates repeated algebraic substructure, suggesting the creation of standard network modules computing the algebraic relationships in the substructures. We have implemented this notion, allowing modules to be connected together to solve larger algebraic problems.

This paper describes in greater detail some aspects of the network algebra engine and geometric reasoning functions.

2.0 Network Creation Details

2.1 Network Components

The major components of a value node are:

definition string (*) - relating the node to the original algebraic constraints
value type - whether a linear or a wrap-around (e.g. from 0 to 2*PI) value range
range limits - the minimum and maximum value the value can take, if known (used for both linear and wrap-around types)
value bounds - the current inf and sup of the value
various saved properties - for possible future state restorations
bounding operation nodes - links to the operation nodes that compute the inf and sup for this node
dependent operation nodes (**) - links to nodes that need recomputing if this node's inf or sup were to change.
update time - when the node was last updated
external binding - link to any externally bound variable
module copy link (**) - for evaluation

The properties marked with (*) are primarily for debugging use and those marked with (**) are used for evaluation or allocation efficiency (see section 5).

The major components of a operation node are:

operation type - the operator computed at this node. Those implemented are: {"+", "*", "/", "-", "sup_of_max", "sup_of_min", "inf_of_max", "inf_of_min", "extract_sup", "extract_inf", "constant", "cos", "sin", "sqrt", "<", "≤", ">", "≥", "and", "or", "not", "select", "enable", "delay" and "guard"}.
arguments - links to the nodes providing inputs for this operation. The links may specify the sup or inf of value nodes or other operation nodes.
evaluation - the operation's current value
initialized flag - signaling that this node now has a value usable by others
dependent nodes (**) - links to nodes that need recomputing if this node's value were to change.
various saved properties - for possible future state restorations
delay and current delay-in-progress for 'enable' type nodes

update time - when the node was last updated (a saved value is also kept)
module copy link (**) - for evaluation

The network interface forms the link between the external variables and the associated internal value nodes. The interface consists of a list of triples recording:

a) an external variable,
b) the associated internal variable inside the module and
c) other bindings on the same external variable from other modules.

2.2 Syntax Checking

The bulk of the syntax checker works by recursively unpacking the expressions. For example, the syntax of the 'plus' expression is checked using the clause (somewhat simplified):

```
checkconstraint([plus, A, B]) :- checkconstraint(A),checkconstraint(B).
```

Hence, it is easy to extend for checking new syntax.

2.3 Network Compilation

The PROLOG procedure 'boundall' bounds the variables in a set of constraints. Each variable is bound over all the variables in the constraint set, that is, no terms are removed from the bounding expressions, unless the result is still guaranteed to be a correct bound. The bounding program used is the PROLOG re-implementation of ACRONYM's CMS [Brooks 1981]. This CMS is not capable of deducing strong bounds when not much is known about the variables in the bounding set. As the variables involved in bounding another variable are often unbounded themselves, the relationships between the bounds, such as alternative terms in a 'min' function, cannot be easily determined. Hence, the main effect of the bounding is to simplify arithmetic terms.

The 'boundall' procedure works by calling the CMS to find both the sup and inf for each variable in the constraint set. The variable is bound over all variables in the set. Bounds expressed in recursive constraints are then added and the whole expression simplified.

The complexity of the network creation lies in the compilation process itself. The compilation must account for all the algebraic structure, yet try to produce a space and time efficient network. At present, the compilation tries to: (a) not duplicate structure whether occurring in the same or other inequalities and (b) merge bounds on variables. These points are discussed below.

Compilation involves deducing the existence of value and operation nodes, and the connections between them. Hence, whenever the creation of a node or a connection is required, the asserted current network is examined for a previous instance of the structure, which is used if it exists. A new instance is created otherwise. Since the variables have global value, it is inefficient to recompute the same function several times and this optimization is thus useful as well as correct.

If expressions are compiled bottom-up, then previously compiled sub-expressions will be detected. (This detection knows that op(A,B) = op(B,A) for certain commutative binary operations). Using the previous sub-expression may cause the current expression to also be a duplicate of an existing expression. Hence, virtually all duplication of structure can be removed. This applies to value, operation and relation nodes.

The key post-compilation simplification is the recognition of instances of:

$$a \le b_1,\ a \le b_2,\ a \le b_3,\ \ldots$$

and reduction of these to:

$$a \le \min(b_1, b_2, b_3, \ldots)$$

A similar reduction applies for max.

When compiling the 'supmax' (or 'supmin', 'infmin', ...) operator over a set of expressions, the compiler tries to produce a balanced tree of binary 'supmax' nodes. This uses the same number of nodes, but reduces the average tree depth - to produce a 'faster' network.

The 'times' expression can be compiled more efficiently if it is known that one or more of its inputs is a constant. The compilation chosen depends on the parity of the constant. Thus, the product

sup(A*B)

where A is a negative constant can be reduced from

max(sup(A)*sup(B),sup(A)*inf(B),inf(A)*sup(B),inf(A)*inf(B))

to

inf(A)*inf(B)

The 'select' construction commonly occurs in the compiled network. This construction selects between alternative evaluations, according to the logical value of a test expression, and is used with the 'enable' function to create a case-statement-like selection function suitable for parallel evaluation. An example occurs in the compilation of the sup of the reciprocal function. Here, there are three cases:

If sup(x) < 0 then sup(1/x) = 1 / inf(x)
If inf(x) > 0 then sup(1/x) = 1 / inf(x)
Otherwise sup(1/x) = p_infinity

The network defined for this logic has the form:

```
                <select_op>
                  /       \
          <select_op>   <enable_op>
            /       \
     <enable_op> <enable_op>
```

Each 'enable' node connects to two subexpressions, one computing the status of the test and the other the value returned if the test is true. Then, the value of the 'select' operation is the value of the first enabled argument. Thus, the 'enable' functions pass their value if the test condition is true, and the 'select' function chooses one of them. If a test never becomes true, its associated value never gets passed. The 'select' function does not become blocked by undefined values.

Not all cases need appear; for example, if some function is undefined for values other than those tested for, it is not necessary to even put in a case for the 'undefined' cases - the operation will just not evaluate in this case.

The 'zero' (or 'notzero') construct results in network structure to see if any possible value of the expression (i.e. somewhere between the inf and sup) has the value zero (or no possible value is zero).

The 'guard' operation is designed to only allow the 'value' field to pass through if the 'test' field holds true. It acts just like the 'enable' operation, except in one respect. Because of timing variations due to argument path lengths, it may be possible for (e.g.) the test argument to become true before the associated value field is consistent. Hence, invalid values may be initially propagated through the node. It may be possible to determine when or if this might occur, and then a construct using 'enable' is compiled (see below). Otherwise, it is necessary to use a 'guard' operation.

The network can execute in two modes: 'guarded' or 'unguarded'. If it is unguarded, then 'guard' nodes execute identically to 'enable' nodes. If the network is guarded, then values are trapped at 'guard' nodes until the network converges. (This ensures that the test and value are applied over consistent values.) Then, all guard nodes pass their values through to continue evaluation. The trapping of new changes at guard nodes continues until no more updates are required.

Sometimes an 'enable' can be used instead of a 'guard'. For example, when the 'sqrt' function is compiled, only one value is needed for both the test and the value (i.e. the argument). Moreover, the depth of the argument in both the test and value tree is the same; hence, any changes to the argument will propagate up both trees to arrive simultaneously at the 'enable' node.

Most standard functions requiring an 'enable' have this property (sqrt, recip, srecip). One that does not is the inf(X^N) function, where N is even. (This function must decide if 0 is in the range of X.) This function is currently compiled as an unbalanced tree with maximum depth N -1, as seen in this prefix tree:

```
*( X, *( X, *( X, .... *( X, X)))))
```

On the other hand, the test has the variable X at depth 1 (for the cases where $\inf(X)^N$ or $\sup(X)^N$ is the result). Hence, we make the enable delay for N - 2 cycles until allowing the result through.

When the user adds a 'guard' function, it may be possible to deduce how to add delays so that the network can execute in 'unguarded' form. A problem exists because the network compiler cannot easily tell when a guarded expression is recursive, or involves other variables even, and hence when to allow an expression to enable a gate. For example, suppose we have the expression

```
[guard, [zero, [variable, c, 1]], [variable, b, 1]]
```

i.e. "if zero(c) then b". This suggests that the two variables are independent. However, suppose that c & b are external interface variables, and another module calculates $b = c^{10}$. Then, c will change value about 8 cycles before b changes value. As modules are compiled independently, it is currently impossible to detect this.

Therefore, if the networks are recursive (e.g. bounds on c are a function of b whose bounds are a function of c) and involve 'guard' operations, then the network should probably be run in guarded form. This may not be necessary as the structure of the problem (including the monotonic convergence of the variables) may make an unguarded network safe. We currently cannot determine when this is the case. If the network is non-recursive, the following theory explains how delays can be added to make the 'enable' safe.

The solution followed here has the gate change to true when all changes have propagated through the value expression and adds enough delays to the 'value' expression to ensure that when the gate changes to false, no values have slipped through. The theory behind this is dependent on the minimum and maximum expression depths of the variables. Let:

Ti be the maximum depth of variable i in the test expression
ti be the minimum depth of variable i in the test expression
Vi be the maximum depth of variable i in the value expression
vi be the minimum depth of variable i in the value expression

For disabling: do as soon as the test is false and add enough delays (D) to the value side to give $Ti \le vi + D$ for all i, to make sure no changes could get through the enable node before the test has been re-evaluated. The number necessary is:

$D = \max(0, Ti - vi)$ over all i

The path length condition causes the test to disable the node before any undesired values get through (but may also stop a few old valid ones still propagating as well). Because variables may appear at several depths in the test expression, the test expression may disable as a result of action at a shallow depth & then be re-enabled when all deeper values propagate through.

For enabling: enable after the new values propagated through the value computation for long enough to reach the enable node, provided the test is still true. The enable node delays switching to enabled for

$$E = \max(T_i - t_i, D + V_i - t_i, 0) \text{ over all } i$$

cycles. The first condition ensures the full test is valid, the second ensures the values used are those tested, and accounts for delays added previously. The node can be re-disabled during the time the node is waiting to enable.

2.4 Network Module Allocation

Since copying takes some time, a pool of previously allocated modules can be pre-allocated and connected as needed.

Since multiple copies of modules are allocated, the network allocation routines have to be careful with reference resolution when copying. That is, two modules may be isomorphic, but must refer to different instances of operator and value nodes. This problem is solved by using address mapping tables during copying.

All data structures used in a module are linked together starting from the module header for fast allocation, deallocation, re-initializing, etc.

3.0 Network Evaluation Details

Ordinarily, an operator will not be evaluated until all arguments have values. This may causes deadlock problems, as discussed in [Fisher 1987a]. To solve this, the max and min operators are evaluated differently according to whether the sup or inf is desired. The sup_of_max (inf_of_min) operator does not evaluate until all arguments are ready, because increasing the upper (decreasing the lower) bound may be necessary as other arguments become ready, and the sup function is only allowed to decrease (increase). However, the inf_of_max (sup_of_min) operator can evaluate when one argument is ready, because later arguments either have no effect or improve the bound. This allows the results to propagate through sections of the network not yet evaluated or never able to be evaluated.

Each network structure keeps three lists of nodes that need recomputing (1) those to be recomputed in this cycle, (2) those to be recomputed in the next, resulting from changes occurring in this cycle and (3) guard nodes whose results get propagated when this round of evaluation has converged. At the end of a cycle, the future list is moved to the present list. The selection of which nodes to recompute is based on the dependency lists associated with each node. Future parallel operation will make some of this method obsolete.

The guard nodes have two different modes of operation, depending on whether the network is in guarded or unguarded mode. If in unguarded mode, it follows the 'enable' node below. If in guarded mode, it does not evaluate until all other network activity has ceased. Then, it evaluates, which may cause new network activity to occur.

The enable node only operates if the test and value arguments are initialized. Then, if the test is true, it may delay a given number of cycles until passing the value through. If the test is false, it disables the node.

Each network structure has its own list of modules allocated to it and its own clock and update queues. Hence, it is possible to create totally independent networks and evaluate them as desired.

If an external variable is interfaced to more than one module, updating the bounds on the variable in one module requires propagating changes into the other modules. This is done by setting up the following links:

a variable in one module links to the module interface

the variable record in the module interface links to the external variable

the external variable links to the variable records in all modules that
the variable is linked with

Then, when an internal variable changes, it:

updates the external variable

causes the equivalent internal variable to be put on the 'changed' list of all modules
linked to the external variable

Inconsistency arises whenever, for any value node, the sup is less than the inf. Unfortunately, because most networks are fully interconnected, there is usually no distinction between input and output variables in a module - the module just expresses the interrelationships between the variables. This means a variable that is set before evaluation, as an input, might also be changed. Hence, the crossing of the sup and inf values of a variable seldom means anything specific about that particular variable.

The effect of having the "input" values change might be desirable, if, for example, the input has a certain amount of uncertainity. Then, the changed input represents a better estimate, taking into account other relationships that must hold.

If the change of a value at a node is less than the convergence threshold assigned for the network, then the change is not made, thus preventing infinitely slow convergence of the network.

It is possible to save the current state of a network, then restore it later. This is useful for testing hypotheses that cause inconsistency, then backtracking.

The following defines the evaluation functions for the different operation types:

constant - returns a constant value

extract sup (inf) - returns the sup (inf) of the referenced value

plus - returns a result if both arguments are initialized:
If $+\infty + +\infty$, then return $+\infty$
If $-\infty + -\infty$, then return $-\infty$
If $+\infty + -\infty$, then indeterminate
If one argument is $+\infty$ or $-\infty$, then return that
Otherwise return sum of arguments

minus - returns a result if both arguments are initialized:
If $+\infty$ - X, then return $+\infty$
If X - $-\infty$, then return $+\infty$
If $-\infty$ - X, then return $-\infty$
If X - $+\infty$, then return $-\infty$
If $+\infty - +\infty$, then indeterminate
If $-\infty - -\infty$, then indeterminate
Otherwise return difference of arguments

times - returns a result if both arguments are initialized:
If one argument is $+\infty$ and the other is > 0, then return $+\infty$
If one argument is $+\infty$ and the other is $= 0$, then return 0
If one argument is $+\infty$ and the other is < 0, then return $-\infty$
If one argument is $-\infty$ and the other is > 0, then return $-\infty$
If one argument is $-\infty$ and the other is $= 0$, then return 0
If one argument is $-\infty$ and the other is < 0, then return $+\infty$
Otherwise return product of arguments

sup_of_max - returns the larger of the arguments if both initialized

sup_of_min - returns the largest of any initialized arguments

inf_of_max - returns the smallest of any initialized arguments

inf_of_min - returns the smaller of the arguments if both initialized

recip - returns a result if the argument is initialized:
If argument is $+\infty$ or $-\infty$, then return 0
If $0 \leq$ argument $\leq \varepsilon$, then $+\infty$
If $-\varepsilon \leq$ argument < 0, then $-\infty$
Otherwise return 1 / argument

sqrt - returns a result if the argument is initialized and ≥ 0:
If argument is $+\infty$, then return $+\infty$
Otherwise return sqrt(argument)

cos (sin) - returns a result if the argument is initialized:
If argument is $+\infty$ or $-\infty$, then indeterminate
Otherwise return cos(argument) (or sin(argument))

greater (or greatereq) - returns a result if both arguments are initialized:
If first argument is $-\infty$, then return false
If second argument is $-\infty$, then return true
If first argument is $+\infty$, then return true
If second argument is $+\infty$, then return false
If $+\infty$ > (or)>=) $+\infty$, then indeterminate
If $-\infty$ > (or)>=) $-\infty$, then indeterminate
If first argument > (or $\geq$) second argument, then return true
Otherwise return false

less (or lesseq) - returns a result if both arguments are initialized:
If first argument is $+\infty$, then return false
If second argument is $+\infty$, then return true
If first argument is $-\infty$, then return true
If second argument is $-\infty$, then return false
If $+\infty$ < (or)<=) $+\infty$, then indeterminate
If $-\infty$ < (or)<=) $-\infty$, then indeterminate
If first argument < (or $\leq$) second argument, then return true
Otherwise return false

and - returns a result if both arguments are initialized:
If both arguments are true, then return true
Otherwise return false

or - returns a result if at least one argument is initialized:
If the first argument is not initialized, then return the second
If the second argument is not initialized, then return the first
If either argument is true, then return true
Otherwise return false

not - returns a result if its argument is initialized:
If its argument is true, then return false

Otherwise return true

delay - returns value (has effect of delaying value 1 cycle)

enable - returns the value argument if both arguments are initialized and the test argument is true and value is not being delayed. If it is being delayed, reduce the delay by 1 count and re-queue the node.

select - returns the value of any initialized argument (arbitrary if more than one).

guard - if network is not in guarded form, then use 'enable' logic. Otherwise, enqueue the changes until the guarded period is temporarily disabled to allow the changes to occur.

Given that the network iterates until convergence, one might ask if the network CMS is now a complete decision procedure (the original ACRONYM CMS was only a partial decision procedure). The answer, unfortunately, is no. While it now detects the system:

$y = x + 1$
$y = x + 2$

as inconsistent, it still cannot detect all cases. In particular, the case

$1 \le x,y,z \le 4$
$x*y \le 4$
$x*z \le 4$
$y*z \le 4$

$x*y*z \ge 9$ (maximum possible is 8)

is not detected as inconsistent. Moreover, even if

$x*y*z \ge 8$

was given, it would not improve the bounds on the variables x, y or z.

4.0 Geometrical Reasoning Modules Details

While the modules are designed for algebraic bounding based on equalities, coming from analytic solution, there are also some heuristic bounds that are applied to speed up convergence or improve partial bounds when complete information is not available. An example of the first case occurs in the "two points are close" module, where the constraint on the euclidean distance between the two points also applies a constraint on the individual coordinates of the points. An example of second case occurs in the "a position maps two vectors rigidly to another two vectors" module, where partial bounds on the position are generated even if only one input and output vector are known.

Based on experience with the geometric reasoning, some new modules have been added. Also, changes to the compilation has changed the module sizes somewhat. The current modules and their sizes are listed here:

Module	Size
two scalers are close: ss	15
two points are close: pp	188
two vectors are close: vv	727
transform point: tp	1076

transform a vector: tv	1704
transform two vectors: tv2	1937
(plus special cases: tv2s)	1244
transform position: tt	2088
transform vector and point: tvp	3011
turn 2 points into a vector: p2v	173

5.0 Possible Extensions

This section proposes several extensions to the compilation and evaluation of the networks:

(1) Currently, the compilations for the variable^n constructions produce a linear tree of binary function nodes. Since the evaluations must propagate up these trees, constructing a balanced binary tree would produce faster computations. This does not affect the geometric reasoning modules since the maximum exponent is two.

(2) All function nodes are at most binary. N-ary nodes could be introduced, at the expense of more complicated logic for dealing with infinity, etc.

(3) If some bounds were known on A or B in A*B, then the compiler could remove some of the cases. Most of the network code involves multiplication, so recognizing simplifications might produce dramatic savings. Possibilities include multiplying out complex terms and exploiting associativity.

(4) It might be possible to extend the analysis of when guarding a network is necessary, particularly when all variables are in a single module. While we use some guarded clauses in a heavily recursive network, with externally connected variables, guarding does not seem to be necessary. This may be due to the particular conditions we are testing for, or may result from deeper results related to the fact that the bounds on values can only tighten. Not using the guarding reduces the number of cycles before convergence by about 60% in our test cases, so there is clearly an advantage to not using guarding unless necessary.

(5) Allocation of network modules is currently done by special purpose subroutines. Given that one often allocates a 'standard' group of modules with set connections, it may be desirable to introduce a 'macro' module type, whose definition makes explicit the modules and connections implicit in the subroutine-allocated form.

6.0 Conclusions

While this paper covers many short topics, there are several that are worth highlighting:

(1) the network optimization methods for reducing the size of networks and improving execution times through path shortening,

(2) how a "case" statement can be built into the network without preventing parallel execution,

(3) how judicious choice of default behaviour for functions can help prevent deadlock when not all nodes have usable values (such as at startup or when from disabled values),

(4) how the network can compute with infinite bound values, and

(5) how to safely implement and evaluate parallel conditionals in these networks.

7.0 References

[Bledsoe 1974] Bledsoe, W. W., "The sup-inf method in Presburger arithmetic", Memo ATP 18, Dept. of Math. and Comp. Sci., Univ. of Texas at Austin, 1974.

[Brooks 1981] Brooks, R. A., "Symbolic Reasoning among 3D Models and 2D Images", Stanford AIM-343, STAN-CS-81-861, 1981.

[Davis 1987] Davis, E., "Constraint Propagation with Interval Labels", Artificial Intelligence, Vol 32, No 3, July 1987, pp281-331.

[Fisher 1987a] Fisher, R. B., "Solving Algebraic Constraints In A Parallel Network, As Applied To Geometric Reasoning", Dept. of Artificial Intelligence Working Paper 205, Univ. of Edinburgh, 1987.

[Fisher 1987b] Fisher, R. B., "Users Guide for a Network Engine for Algebraic and Geometric Reasoning", Dept. of Artificial Intelligence, forthcoming Software Paper , Univ. of Edinburgh, 1987.

[Fisher and Orr 1987a] Fisher, R. B., Orr, M., J., L., "Geometric Constraints from 2 1/2D Sketch Data and Object Models", Dept. of Artificial Intelligence Research report 318, University of Edinburgh, 1987.

[Fisher and Orr 1987b] Fisher, R. B., Orr, M., J., L., "Solving Geometric Constraints In A Parallel Network", Dept. of Artificial Intelligence Research report 334, University of Edinburgh, 1987.

[Orr 1987] Orr, M.J.L., "Geometric constraints in 3D computer vision", Dept. of Artif. Intel., working paper 203, Edinburgh University, 1987b.

[Orr and Fisher 1987] Orr, M. J. L., Fisher, R. B., "Geometric Reasoning for Computer Vision", *Image and Vision Computing,* Vol 5, No 3, pp233-238, 1987.

also Univ. of Edinburgh, Dept. of Artificial Intelligence Research Paper 311, 1986.

[Shostak 1977] Shostak, R. E., "On the sup-inf method for proving Presburger formulas", J. Assoc. Comput. Mach., **24**, pp529-543, 1977.