Reducing Computation by Unifying Inference with User Interface

Mark W. Perlin

June 1988
CMU-CS-88-150

Abstract

User interfaces are becoming an integral component of usable interactive artificial intelligence systems. A high-bandwidth graphical interface between the user and the computer's representation of knowledge greatly facilitates intelligent human/computer interaction. When user input or inference modifies knowledge base assertions, computationally expensive cascaded updates often propagate through the underlying knowledge representation. To communicate these changes to the user, the graphical display must also be updated.

The speed of knowledge base update is crucial in AI interfaces. Real-time global update is often infeasible due to the richness and complexity of knowledge bases. Interestingly, the user's display usually presents only a fraction of all the available internally represented information. To rapidly show the user the global effects of changes, knowledge base update can be restricted to just those computations required for maintaining an accurate display.

We have developed an integrated approach to inference and user interface which offers a new programming paradigm for the construction of AI interfaces. The methodology:

- greatly simplifies the design, construction and runtime execution of user interfaces to knowledge bases;
- reduces inference computation by user interface-constrained lazy evaluation: only what the user can *observe* is computed;
- integrates inference and user interface into one system specifically tailored to the AI *user*'s immediate needs.

This publication was supported in part by grant R29 LM 04707 from the National Library of Medicine.

Table of Contents

List of Figures

I. Background

In integrating user interfaces (UIs) with artificial intelligence systems, the primary focus moves away from the underlying inference, and toward the user's interactive needs. From the system builder's perspective, a natural display representation of inference would ease UI design, implementation and, ultimately, runtime operation. A functional interactive interface requires an additional feature: realtime speed. A serious problem, therefore, is that complete inferencing in AI systems is often too time consuming for interactive use.

Previous user interface work in AI has been from the system builder's perspective. The object-oriented approach [10] offered part-whole relationships and object inheritance to ease UI design. In adding constraints, systems like Sketchpad [17] and Thinglab [1] incorporated useful inference into the UI. These systems also addressed the larger issue of how a separate (domain-specific) inference system can drive the UI control structure.

To achieve the realtime performance crucial for usable interactive UIs in AI, we have developed a new stratagem: use the UI to reduce inference. In our approach the *user's perspective* provides a powerful constraint, unlike prior systems (e.g., [1]) which only use internal constraints. Of the dozens of windows depicting inference available to the user, ordinarily only several are presented at any one time. When a reasoner can prove to itself that an inference which will not affect the visible display can be deferred, we defer it. Essentially, lazy evaluation derived from the user's active display can effectively constrain inference.

II. Inference

Inference in symbolic AI is often the cascading of small, uniform computations. The cascading is apparent in frame system inheritance [2], in the chaining of rules [3], in propagating constraints [16], and other graph-based algorithms. With rule chaining, for example, the uniform cascade is independent of direction (forward or backward).

This local regularity often enables the recursive cascade to form the control structure of other computation. Most commonly, it is exploited by attaching "demon" procedures [21], to generate useful side effects. Such procedural attachment is seen, for example, in the internal implementation of object or frame inheritance relations. A common useful side effect of the inference system, but *external* to it, is generating user interface and graphical output. The effects of inference can be pushed even further into the external world to drive the effector actions of an intelligent sensor-based system.

Just as inference can initiate other computation, inference can also be *constrained* by outside computation. In data-driven reasoning [6], for example, forward chaining from data tightly constrains the topology of the search space. Similarly, target goals prune the search space in goal-directed inference. This reduction in computation is more explicit in the constraint-based reasoners found in parsing [4], matching [8], planning [14] and scheduling [9] problems. Inference can also be bounded by resource limitations (fixed time or search depth) and by knowledge [12].

III. User Interface

The user interface is the communication window between human user and machine. Via the UI display, the computer outputs a perspective on underlying computation, and also receives user input for initiating further actions. Symmetrically, the UI display is the *input* the user receives from the program; the user's output (i.e., actions) provides the interface with input for the computer. Graphical presentation of I/O is increasingly common due to the low expense (both computational and price) of bit-mapped display [7] technology.

The control structure of the UI display often parallels the application program. As the underlying program performs computation, it drives associated user interface computation for displaying its results. This approach is used, for example, in object-oriented user interface programming. Two problems that arise when display is procedurally attached to underlying computation are

1. display computations for undisplayed windows may be executed unnecessarily, and
2. display computation may be redundantly repeated for the same window.

A solution using topologically ordered update is proposed in Section V.2.

Just as the user interface can be driven by some computations, so too can it *constrain* other computations. Restricting user actions to a specific syntax (via keyboard or mouse) is a simple mechanism for limiting the initiation of computation. More generally, the user interface should assist an application program in avoiding computation not visible to the user. In Section IV, we shall introduce a new mechanism for filtering computation based on the current set of displayed windows.

IV. Unifying Inference and User Interface

Inference performed at execution time is the natural control structure for driving user interface display update. Typical updating requirements include graphical recomputation and display (e.g., window) organization. Following the object-oriented side effect approach, UI update can be readily implemented via procedural attachment of display demons to inference computations. This can establish an efficient display regimen, tailored to showing the occurence and effects of inference.

By embodying the problem-solving focus of the user, the user interface can constrain inference. This is our key new idea: *never compute anything the user cannot see.* When a reasoner can establish that deferring a computation will not affect the user's interactive display, lazy evaluation can be invoked. How does a system know what is displayable? By letting the user's perspective, defined by the current set of display windows, filter computation.[1]

In a sense, the current window set forms the "goal" of a user's interface display. The set implicitly defines

[1]Periodically, the user will change which windows appear on the display. To preserve the integrity of the inferencer, a more complete update must then be performed. In practice, the associated delay requires less time and is more acceptable to the user than continual update delays incurred with each user action.

for the computer the central concerns of the user.[2] Where possible, then, inference should be constrained by these implicit user goals. This presents the implementer with an opportunity for reducing time-intensive AI inference.

The ideas in this section unify the perspectives introduced in Sections II and III, and suggest a new paradigm: inference both driving and contrained by the user interface. An algorithm (Section V) and an architecture (Sections VI and VII) emerge from this paradigm, with an integrated approach to the design, implementation and execution of user interfaces to inference systems.

V. Acyclic Update

A substantial number of inference techniques in AI employ directed acyclic graphs (DAGs) in their data structures and algorithms. DAGs effectively combine the flexibility of graphs with the indexing performance of trees. When dynamic DAG data structures are created and modified, their individual components (both node connection topology and node data) require updating. *Acyclic update* refers to this DAG update process.

Acyclic update is ubiquitous in AI. Efficient matching in rule systems, as with the RETE discrimination network [8], relies on forward propagation of assertion (working memory) changes through a compiled rule-pattern DAG. Equational networks, such as Bayesian networks of probabilities [5], use DAGs to avoid redundant numerical recomputation. In natural language processing, advanced syntactic [20] and semantic [4] methods use acyclic update to constrain and reduce search. In planning [14] and plan recognition [15], DAGs cache partial updates to reduce computation.

We present here a new algorithm for restricting computation in DAGs to just those results which interest the user. The algorithm is generally applicable to all DAG-based inference; we will explore in Section VI.2 a specific example with equational networks. After reviewing the DAG data structure in Section V.1, we show in Section V.2 how topological sort can eliminate redundant updates. The view mechanism of Section V.3 applies the user's focus of attention in further restricting computation.

V.1. Dependency data structure

DAGs may be useful whenever there exists a partial ordering[3] relation $\leq$ between objects. Depending on the application, the partial ordering may assume various names, such as "depends-on" in planning and equational nets, "part-of" in window systems, and "is-a" in inheritance relations. Since our focus here is the dependency of one computation on another, we shall use "depends-on" to describe the relationship between different computational objects. If object A's computation propagates to B (i.e., B depends on A), we write $A \rightarrow B$.

[2]The inferencer, of course, may bring new information to the user's attention in system-created windows. This, however, represents the computer's (versus user's) agenda, and can be treated appropriately.

[3]"$\leq$" is a partial ordering on a set S if and only if for any two elements A, B in S, $A \leq B$, $B \leq A$, or A and B are unrelated.

Given a consistent collection of dependency relations, we obtain a corresponding directed acyclic graph, as in figure V-1. Objects independent of all others are called *leaves*. Changes made to a subset of leaves (independent premises) propagate up though the DAG, updating the nodes in the leaves' transitive closure. A way of uniformly propagating changes is to associate a list of candidate predicate-action pairs with a propagation. At each visited node, for every predicate-action pair on the list, when the node satisfies the predicate, the local action is performed.

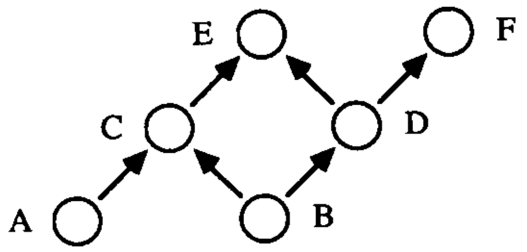

Figure V-1: A prototypical dependency network.

For example, suppose the list contains only the pair

```
(t
 (recompute-value *node*)).
```

That is, the predicate is always true, and the local action is to recompute[4] the node's value from the values of its predecessors. The simplest node traversal is depth-first search (DFS) from the leaves. In figure V-1, starting from the singleton leaf set {B}, we visit nodes in the order <B, C, E, D, E, F>. Clearly this DFS update schedule is inefficient, since some values (e.g., node E's value) are redundantly recomputed several times. Other traversal schedules, such as breadth-first, may be inefficient or yield incorrect results. In section V.2 we introduce a DAG traveral that is both optimally efficient and correct.

V.2. Topologically sorted update

Topological sort [11] is usually described as an algorithm for listing out the elements of a partially ordered set in an order satisfying: if $a \leq b$, then a is listed before b. If the partial ordering represents a dependency relation, then a topological sort will assure that no element appears until all of its predecessors have appeared. Since this DAG traversal visits dependent nodes only after all their predecessors have been visited, it assures update *correctness*. Further, since each required element in the traversal appears exactly once, a reasonable implementation would also be optimally *efficient*.

We have devised a topological sort algorithm for DAGs using a modified mark/sweep garbage collection scheme [11] with reference counts. The first pass is a *dependency* marking to determine which graph nodes are dependent on modified predecessor nodes. This is done in a depth-first manner, recording the

[4]To fix ideas, think of the DAG as an equational network, where each node has a formula for computing its value from its immediate predecessors.

number of immediate predecessors which have been modified. The second *revision* pass sweeps through the marked subgraph, updating a node only when all its marked immediate predecessors have been updated. Before formally presenting the algorithm, we give an example.

Consider the update following a change in leaf B.[5] This change propagates to all the descendants of B, as shown in the shaded area of figure V-2.

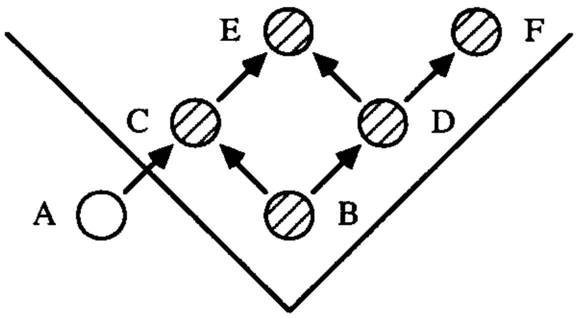

Figure V-2: The subgraph of nodes dependent on leaf B.

A depth-first traversal from B, propagating only on the first visit, marks the nodes in the order <C, E, D, F>. Using reference counts, node C has a count of 1 (only one immediate predecessor, B, was marked for update), node E a count of 2, node D 1, and F 1.

The second pass traverses the same subgraph, decrementing a node's count when each immediate predecessor is updated. A node is updated and propagates only when all of its predecessors are updated, i.e., when its reference count is decremented back to 0. For example, starting from node B, node C is visited and its reference count is decremented from 1 to 0. Node C is then updated (possibly using information from its immediate predecessors A and B), and locally propagates to its immediate successor E. Node E is not updated yet, because it depends on unupdated D; E knows this because its count is still >0. Control recursively returns to node B, which then propagates to its other immediate successor, D. D has no unupdated immediate predecessors, so it updates itself, and passes control to nodes E, then F. With all relevant predecessors updated, E and F can now be updated. Unlike DFS, the order of updates is topologically ordered as <C, D, E, F>. The order ensures that nodes (e.g., E) are updated only after all their updated predecessors.

The algorithm UPDATE is called on a subset of the leaf nodes.

```
UPDATE (L: a set of leaves)
        iterate DEPENDENCY over L
        iterate REVISION over L
```

[5]The update algorithm is equally efficient when changes are made to multiple leaves; for simplicity, we change only one leaf in our example.

The marking phase DEPENDENCY records the number of immediate predecessors of a node in the dependency subgraph which have been marked.

```
DEPENDENCY (node: a node in the DAG)
        increment node.counter
        if first-visit(node)
            then iterate DEPENDENCY
                        over node.successors
```

node.successors is the list of immediate successors of the node. *first-visit* is determined by simply looking at the counter (which is initialized to zero).

The sweep phase REVISION performs the local update computation when all predecessors are updated.

```
REVISION (node: a node in the DAG)
        decrement node.counter
        if last-visit(node)
            then{ perform local update(s)
                  iterate REVISION
                        over node.successors }
```

The updates may be a simple formula evaluation, or may be a complex series of computations (e.g., value update, diagnostic messages, output display demons) conditional on the local state. The predicate-action list mechanism described above in Section V.1 is especially well-suited to complex conditional updates.

Let L be the subset of modified leaves. Algorithm UPDATE satisfies:

1. The set of updated nodes is precisely the set of nodes dependent on L.
2. Every dependent node is updated exactly once.
3. Update at a node is performed only after all its predecessors have been updated.

Facts 1 and 2 assure the *efficiency* of UPDATE, while facts 1 and 3 guarantee *correctness*. These facts can all be easily proven from the monotonic increase, then decrease, of the local reference counts.

Let n be the size (number of nodes) of the DAG, and $m \leq n$ the size of the subgraph dependent on the leaf subset L. k is the branching factor of the DAG. The bookkeeping counter operations require O(km) operations, O(k) branches for each of O(m) node expansions.[6] Typically, though, the very expensive computation is all in the local update operations; UPDATE limits these to *m*. This is better than the $O(n^2)$ complexity of the totally naive method[7] and the $O(m^2)$ behavior of DFS update.

V.3. View mechanism

Suppose, out of the entire network of possible computation, the user is focused on the outcome of only several results. From the user's perspective, then, all computations not directed toward these results are wasted effort. This suggests an approach to speeding up user interaction time. We describe in this

[6]Equivalently, one bookkeeping operation is required for each *edge* in the dependent subgraph. The number of such edges is the product km.

[7]Ignore all leaf information, and repeatedly update *everything* until consistent.

section a DAG update algorithm which filters computation based on the user's perspectives, or *views*.

Our goal is to avoid updating any nodes which are not in a view. A node will be *active*, therefore, if it has been explicitly ***set active*** as a user view, or if it is required as a predecessor computation for a view node. The active nodes are established by a reverse propagation from a view node back toward its predecessors. When UPDATE is initiated from a leaf set L, it will only update a node which both descends from a leaf in L, and has been activated by (precedes) a view.

As an example, consider imposing views on the DAG of figure V-1. Suppose we are interested in node E. A reverse propagation from E activates all nodes but F. If an update from leaf B is done, we update the intersection of successors of B and predecessors of E (i.e., B, C, D, and E), as is shown in figure V-3. Changes to the view state can be done incrementally. Activating node D with node E already set active, necessitates only setting D active: the active state need not be propagated to D's (already active) predecessors.

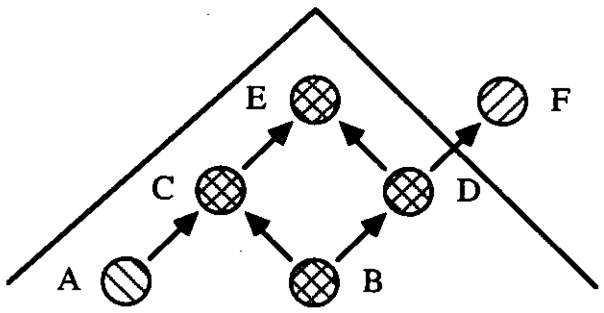

Figure V-3: The DAG subgraph dependent on leaf B needed for view E.

The active state can be defined locally with a bit and a counter. Each node records

1. whether it has explicitly been set active as a view, and
2. the number of immediate successors which are active.

The recursive ACTIVE predicate succeeds if a node is either explicitly set active, or has at least one active successor. To activate a view, ACTIVATE-VIEW is called on the view node.

```
ACTIVATE-VIEW (node: a node in a DAG)
        explicitly set the node active
        if the predicate ACTIVE(node) changed
            then iterate ACTIVATE
                        over node.predecessors

ACTIVATE (node: a node in a DAG)
        increment node.active-successor counter
        if the predicate ACTIVE(node) changed
            then iterate ACTIVATE
                        over node.predecessors
```

DEACTIVATE-VIEW and DEACTIVATE are similar incremental algorithms.

To work with views, UPDATE is modified to filter out inactive nodes. For example, DEPENDENCY is changed to propagate only to active successors.

```
DEPENDENCY (node: a node in the DAG)
        increment node.counter
        if first-visit(node)
            then iterate DEPENDENCY
                        over the ACTIVE node.successors
```

The modification to REVISION is analogous.

The correctness of (DE)ACTIVATE follows from a simple induction on the DAG using the ACTIVE predicate. The computational cost is linear in the number of nodes activated, which is $\leq O(n)$. The incremental approach improves average behavior.

VI. Architecture

We introduce a DAG-based control structure for user displays, and then describe its associated programming paradigm. We conclude with an example implementation.

VI.1. Control structure and paradigm

We have been using the "depends-on" relation in the sense of "depends-on-value". A more primitive "depends-on" relation is simply "depends-on-existence". (Note that if B "depends-on-value" A, then also B "depends-on-existence" A.) In object-oriented window systems, the primary relations are "part-of" and "is-a", though more advanced systems may have other relations, such as "constrains" [1]. Interestingly, if A is "part-of" B, then B "depends-on-existence" A; with this observation, we unify inference and user interface.

For each node in the inference DAG with an associated window display object, we interface inference and display with an input and/or output node. Figure VI-1 shows the interface relations between the DAG and the window system. UPDATE will follow "depends-on-existence" links, hence all "depends-on-value" links in the DAG and all "part-of" links in the display. Therefore, as UPDATE is used in inference, the display will correctly and efficiently reflect the DAG state. Conversely, the view mechanism propagates from the user's selected windows, back through the DAG. These views selectively activate just those DAG nodes which affect the user's display, and thereby constrain inference.

This architecture offers a paradigm for building DAG-based inference systems with displays.

Design. The display can be independently designed, specifically for the (graphical) representation of domain and inference concepts.

Implementation. Inference and display can be built and prototyped separately, and later merged using the input/output graphics nodes.

Execution. Updates on the inference DAG are the primary control structure for the output display. Symmetrically, the current window set propagates views onto the underlying inference DAG, filtering out computation (both inference and display) not needed for the user's interaction.

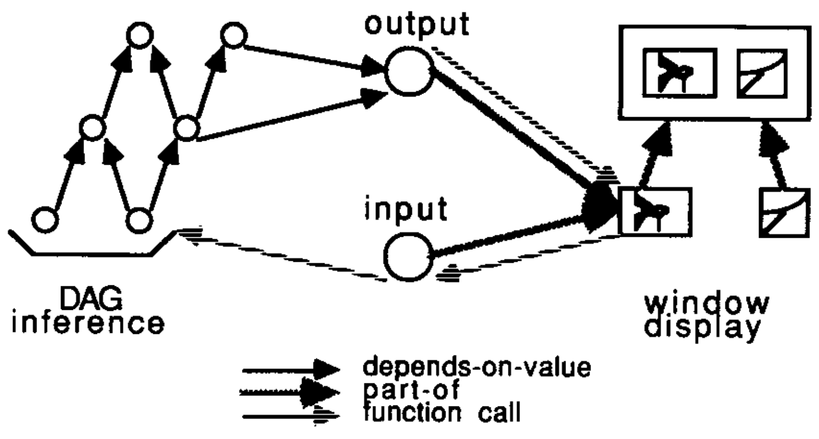

Figure VI-1: A mutual control structure for inference and display.

VI.2. An Implementation

In medical magnetic resonance imaging (MRI), an important task of the radiologist is to select MR scanner parameters [19] in accordance with his/her imaging goals. This is the domain task of the CONTRAST expert system [13]. Much of the imaging goal interactions can be approximated using equations from the underlying MR physics. For example, contrast-to-noise between tissues A and B is roughly

$$C/N = S/N \; |I_A - I_B|,$$

where I_A, I_B and S/N recursively depend on other formulas. These relationships form a DAG equational network, which is used by CONTRAST in the inner loop of its problem solver. The network is also used to control the display graphics in the user interface. Windows associated with DAG nodes represent domain-dependent concepts (e.g., contrast, voxel, geometry) as user-manipulable graphics objects.

Using the above architecture, many versions of the system have been built. The different environments include

Languages:	LISP, OPS5, FRAMEKIT, C, PASCAL;
Machines:	VAX, PC, PERQ, SUN;
Window systems:	Metawindow, SunView, X.

The current version runs in Common LISP on a color SUN3/260 in SunView. System development time depends only on working within the particular graphics window system, since the control structure is handled by UPDATE's DAG traversal. The view mechanism tremendously reduces the computer's interaction time with the user, particularly when only several windows are displayed. This is demonstrated in Figure VI-2, which plots the time required to update the system as a function of the number of active windows.

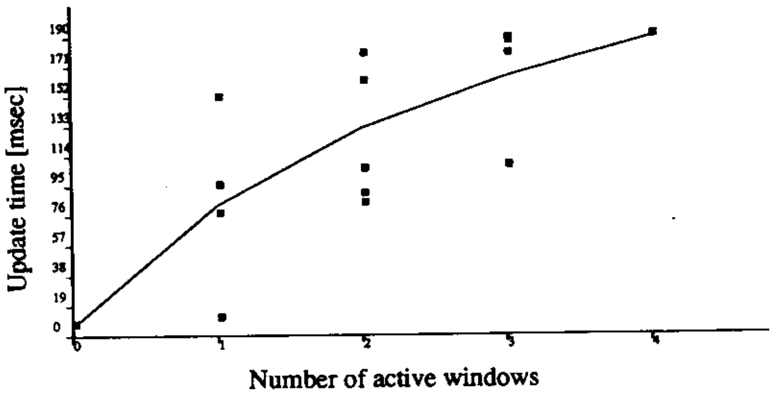

Figure VI-2: Update time as a function of the number of active windows.

VII. Graph Update

Inference traversing graphs with cycles is more general than DAG-based inference. Algorithms based on *graph update* include frame system link traversals, truth maintenance system inference and constraint propagation methods. There are two ways to use the architecture of Section VI on any given graph update algorithm:

1. explicitly break the cycles, thereby essentially transforming the graph into a DAG, and then directly use the methods of Section VI, or
2. adapt the notion of views to the particular update algorithm, diverging as necessary from the strictly DAG-based architecture.

We give an example of the second approach, and how its view mechanism can reduce inference.

Waltz scene labeling [18] is a classic constraint propagation algorithm. A two-dimensional projection of a three-dimensional scene is considered as a graph of lines and connecting junctions. The output of Waltz' procedure is a consistent labeling (if one exists) of the lines as convex, concave or boundary; figure VII-1 shows a sample labeled output from our interface. Three-dimensional geometry constrains the junctions to admit only a small number of the theoretically possible combinations of line labels. These constraints are readily classified at each junction by the two-dimensional geometry of the lines they connect, and then propagate throughout the scene graph. Propagations with cycles are inherent to Waltz' formulation.

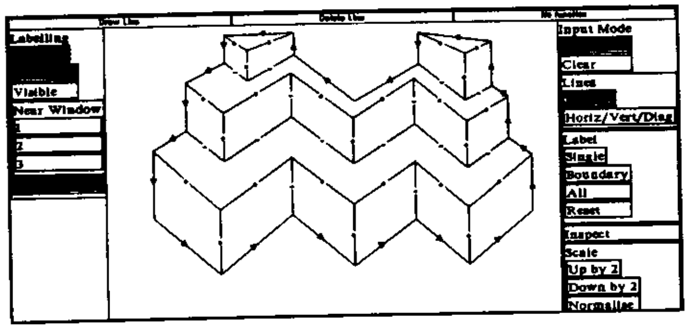

Figure VII-1: A labeled scene from the constraint propagation program.

We adapted the acyclic update architecture to Waltz' procedure. Junction and line label update in the problem solver form the primary control structure of the output display. A modified view mechanism eliminates much inference by restricting the constraint propagation to "active junctions". The user can choose from a variety of "active junction" predicates, such as activating only those junctions that are either on the boundary, or in (or near) the display window. With this tailored view mechanism, inferencing is substantially reduced. For example, following an initial (one second) boundary computation, the times needed to label the lines of an interior junction of the scene in figure VII-1 (including graph update and display) were[8]

single (1) node	**several (5) nodes**	**all (40) nodes**
40 msec	580 msec	1780 msec

These timings demonstrate how views can accelerate user response time.

VIII. Conclusion

We have developed a new paradigm for unifying AI inference with user interfaces. By using the interface to discern the user's perspective, inference is constrained by lazy evaluation: only what the user can *observe* at any time is computed. This reduces both inference and display computation. Symmetrically, we have also integrated the user interface control structure with inference. This eases the design, construction and execution of the user interface. Successful application of the paradigm produces integrated inference/user interface systems tailored to the AI *user's* needs.

ACKNOWLEDGEMENTS

Jaime Carbonell offered numerous insights. The many conversations with Greg Hood and Bruce Krulwich were invaluable. Sean P. Engelson and C. Charles Law provided superb programming support for the implementations. The clarity of the paper was improved by the careful readings of Yolanda Gil, Dawn Haberlach, Todd Kaufmann and Ira Monarch.

REFERENCES

1. Borning, A. ThingLab - an object oriented system for building simulations using constraints. Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI-5, MIT, Cambridge, August, 1977, pp. 497-498.

2. Brachman, R. J. On the Epistemological Status of Semantic Networks. In *Associative Networks,* N. V. Findler, Ed., New York: Academic Press, 1979.

3. Buchanan, B.G., and Shortliffe, E.H. (Ed.). *Rule-Based Expert Systems.* Addison-Wesley, Reading, MA, 1984.

4. Carbonell, J. G. and Hayes, P. J. Natural Language Understanding. In *Encyclopedia of Artificial Intelligence*, Shapiro, S. C., Ed., Wiley & Sons, New York, NY, 1987, pp. 660-677.

[8]The program was written in Common LISP, integrated with the X window system, and run on a SUN3/260. Each time shown is an average over five different problem instances.

5. Charniak, E., and McDermott, D.. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1986.

6. Charniak, E., Riesbeck, C.R., McDermott, D.V., and Meehan, J.R.. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987. second edition.

7. Foley, J.D., and VanDam, A.. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Massachusetts, 1982.

8. Forgy, C.L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem". *Artificial Intelligence 19*, 1 (September 1982), 17-37.

9. Fox, M.S., and Smith, S. "ISIS: A Knowledge-Based System for Factory Scheduling". *International Journal of Expert Systems 1*, 1 (1984).

10. Ingalls, D.H.H. The Smalltalk-76 programming system: Design and implementation. Principles of Programming Languages, 5th Annual ACM Symposium, Tucson, Arizona, January, 1978, pp. 9-16.

11. Knuth, D.E.. *The Art of Computer Programming*. Volume 1:*Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1973. 2nd Edition.

12. Newell, A. The Knowledge Level. Carnegie-Mellon University Computer Science Department, 1981. CMU-CS-81-131.

13. Perlin, M.W., Kanal, E., Carbonell, J.G., and Wolf, G.M. CONTRAST: An Expert System for Optimal Selection of Magnetic Resonance Scanner Parameters. Book of Abstracts, Society of Magnetic Resonance in Medicine, New York City, August, 1987, pp. 770.

14. Sacerdoti, E. D. "Planning in a Hierarchy of Abstraction Spaces". *AI 5*, 2 (1974), 115-135.

15. Schmidt, C., Sridharan, N. and Goodson, J. "The Plan Recognition Problem". *Artificial Intelligence 11*, 2 (1978), 45-83.

16. Steele, G.L. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. Ph.D. Th., Massachusetts Institute of Technology, August 1980.

17. Sutherland, I.E. *Sketchpad: A Man-Machine Graphical Communication System*. Ph.D. Th., Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1963.

18. Waltz, D. Understanding Line Drawings of Scenes with Shadows. In Winston, P., Ed., *The Psychology of Computer Vision*, McGraw Hill, New York, 1975, Chap. 2.

19. Wehrli, F. W., MacFall, J. R., Glover, G. H., Grigsby, N., Haughton, V., and Johanson, J. "The Dependence of Nuclear Magnetic Resonance (NMR) Image Contrast on Intrinsic and Pulse Sequence Timing Parameters". *Magnetic Resonance Imaging 2*, 1 (1984), 3-16.

20. Winograd, T.. *Language as a Cognitive Process, Volume I: Syntax*. Addison-Wesley, 1983.

21. Winston, P.. *Artificial Intelligence*. Reading, MA: Addison Wesley, 1977.