Parallelism in Interpreters for Knowledge Representation Languages

Henry Lieberman

Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA
(617) 253-0315

Electronic mail (Arpanet):
Henry@AI.AI.MIT.Edu

Abstract

While there has been considerable interest in applying parallelism to problems of search in knowledge representation languages, lingering assumptions of sequentiality in the interpreters for such languages still stand in the way of making effective use of parallelism. Most knowledge representation languages have a sequential `QUERY-SEARCH-ANSWER` loop, the analog of the `READ-EVAL-PRINT` loop of Lisp, and employ parallelism only in the `SEARCH` phase, if at all. I will discuss parallel alternatives to sequential interpreters for knowledge representation languages, the effect of these alternatives on choice of search strategies, and new approaches to constructing user interfaces for these languages. The approach is motivated by a desire to respond to Hewitt's "open systems" critique of logic-based systems, which strives for systems that can deal with inconsistent beliefs, dynamically revise beliefs, and are sensitive to allocation of resources.

The move to "open systems" requires re-thinking interaction with knowledge representation systems

If we are to make computers that imitate human problem-solving capabilities, they will also have to imitate some of the characteristics of everyday, common sense human reasoning. Most knowledge representation systems take their inspiration from the specialized, idealized kind of reasoning that occurs in mathematics, rather than the more mundane problem solving that people do every day.

Since people can still function effectively, even if not all their beliefs are consistent, a knowledge representation system must be prepared to represent inconsistent beliefs. Since people can revise their beliefs if they learn new things, our knowledge representation systems cannot insist on monotonicity. Since people's ideas are strongly affected by how much attention they pay to each topic, are systems must do likewise. But if we move to systems that are inconsistent, change their beliefs often, and constantly shift their attention, what prevents total chaos?

Open systems are such a drastic departure from conventional knowledge representation systems that we must re-think the entire process of interaction between the user and the system. Just as automobiles were first conceptualized as "horseless carriages", there is a danger of carrying over our ingrained assumptions from sequential computing to the new era of parallel machines. The purpose of this paper is to explore some of the consequences of the move to parallelism for knowledge representation systems, and in particular, to challenge some almost universally accepted principles for the process of interpretation used in such systems.

It's my belief that progress in knowledge representation systems has been slowed by lack of attention to interface issues. Perhaps this is due to a mistaken prejudice that interface issues are "merely cosmetic". Progress in interface techniques can dramatically improve the accessibility and usefulness of the intelligence built into knowledge representation systems. Through the use of parallelism and interactive graphics, we can at least improve the communication between the "open system" of the user's mind and that of the system so that they can cooperate effectively on intelligent problem solving. Insistence on building a knowledge representation system around a sequential, textual interface has put artificial constraints on the inference procedures and mechanisms used in these languages.

Parallelism in the interpreter changes the way we think about efficiency in knowledge representation systems

Common to most conventional knowledge representation systems is an interpreter that has roughly the following form:

```
* The user types an assertion [or a "rule"] or a query.
* If the user typed an assertion, the system adds it to the knowledge
base.
* If the user typed a query, the system searches for an answer to the
query.
```

```
* The system prints a YES or NO answer to the query,
  or prints values of variables in the query.
* The system loops back to read the next assertion or query.
```

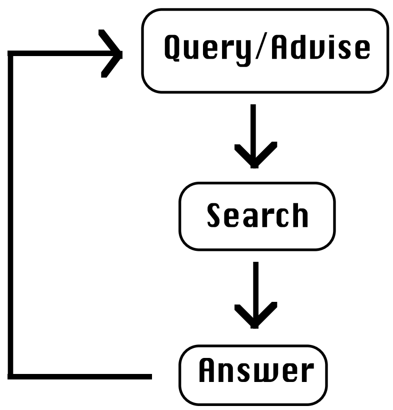
I will refer to this as the `QUERY-SEARCH-ANSWER` loop, analogous to the `READ-EVAL-PRINT` loop of Lisp. There are inherent assumptions of sequentiality in this interpreter. Although some modern interpreters may use parallelism during the search process, almost all rely on the `QUERY-SEARCH-ANSWER` loop to be executed serially.

This places a great burden on the inference procedure that answers the queries. First of all, it must expend all possible effort to answer the query when the user types in the query, keeping the user waiting during processing. Second, it must be very sure of the answer it returns, since there is no way to go back and "change its mind", since it can't "undo" printing the answer it gave to the query. Both these properties are inappropriate for open systems, which must be sensitive to resource allocations, and are not monotonic.

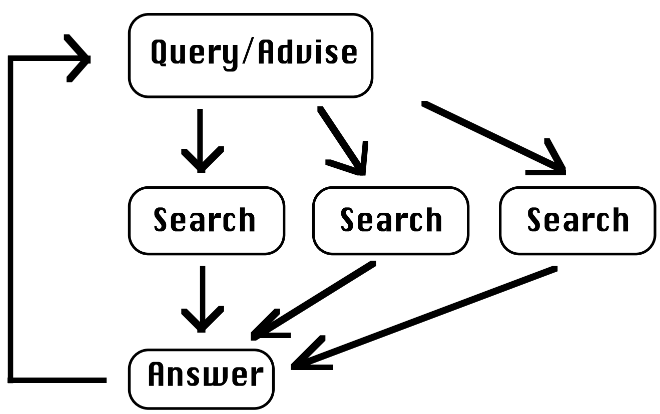
The sequential nature of the `QUERY-SEARCH-ANSWER` loop in conventional systems causes people to focus on *total computation time* to answer a query as the primary measure of efficiency in the system. In modern interactive personal computer systems, however, compute time is not nearly as important as *response time*. A system which is perceived by its user as responsive is considered efficient, even if it "wastes" computation time over the long run.

To keep the system responsive, we will employ parallelism, allowing search processes to overlap with further queries or assertions. When a user presents a query, the system expends only a modest amount of effort trying to figure out the answer to the query and displays its "best guess so far". The system then *continues* to work on the problem in parallel, while the user can perform further queries or assertions.

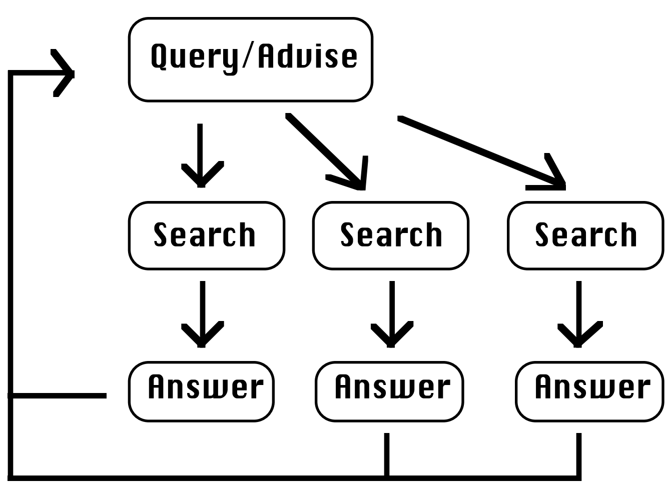
Three control structures for knowledge representation interpreters



Sequential



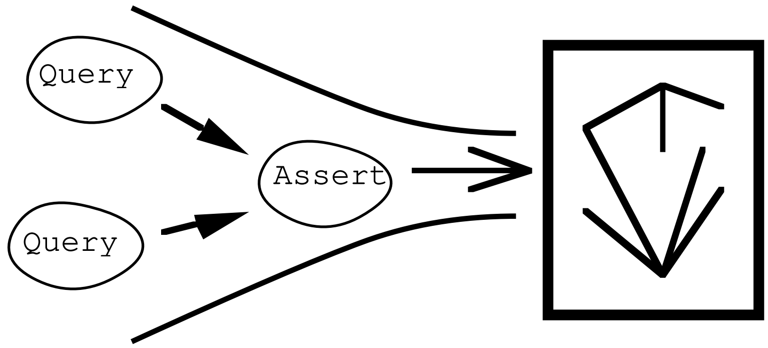
Parallel in search only



Parallel search and answers

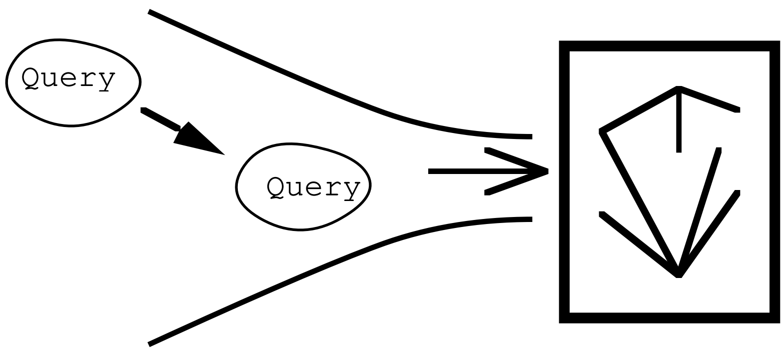
As the system discovers more information to answer the query, the system *updates* the displayed representation of the answer dynamically to reflect its new understanding, even after the query had ostensibly "returned". The control structure for this overlap is accomplished using the *future* mechanism, appearing in the actor programming languages [Baker and Hewitt 76] [Lieberman 87]. This works automatically even if a query appears to "need" the result of some previous query, delaying the computation until the required result is available.

To deal with parallelism effectively, the knowledge base must be implemented using a *serializer*, an actor which can respond to requests from multiple parallel processes, and has changeable internal state. Assertions and queries are put on a *queue*, and accepted one-by-one so that they do not cause conflicts and timing errors. The knowledge base must time-share the processes of accepting queries from the user with performing search to process those queries and updating the display dynamically as a result. While the interpreter is "waiting for input" it can be processing any pending queries that have not been fully answered.

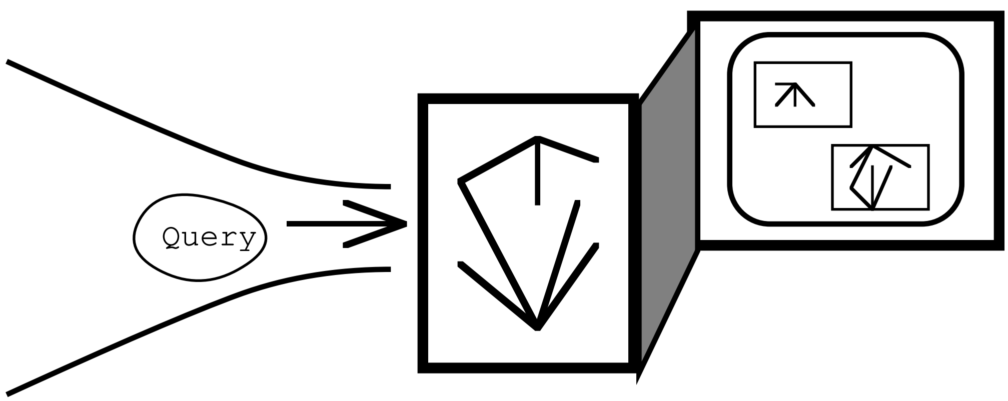


Input pipeline of queries, assertions

Stored description network



Modifications to the network are serialized



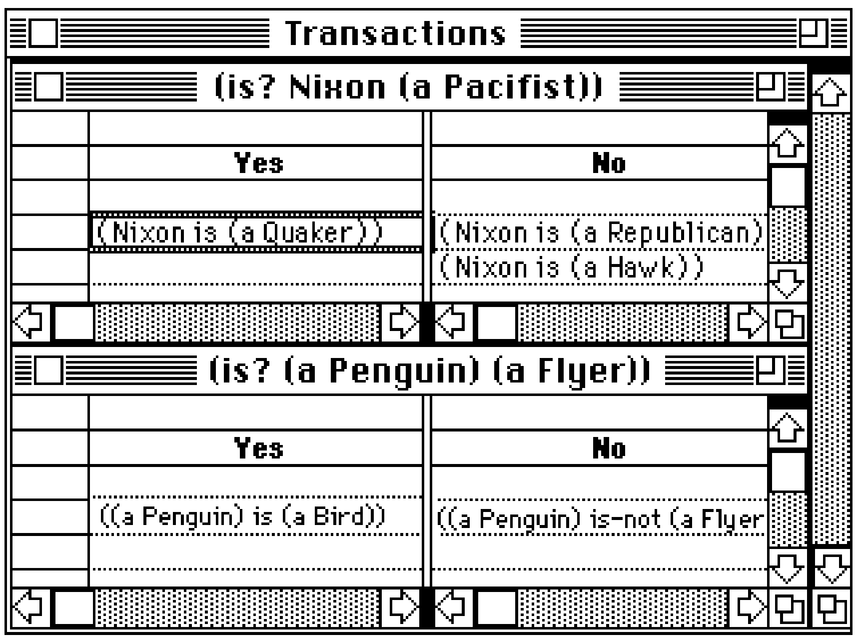
Queries generate searches in parallel

Relationship of queries, assertions and search processes

A multi-window graphical interface makes accessible parallelism the knowledge representation interpreter

Essential to making the parallelism in the interpreter effective is careful design of the user interface. If we rely solely on textual "read-and-print" interfaces, the user cannot effectively monitor and control the parallelism the system provides. Graphical, multi-window interfaces map the parallelism of human vision and hand-eye coordination onto the parallelism inherent in the system's computation.

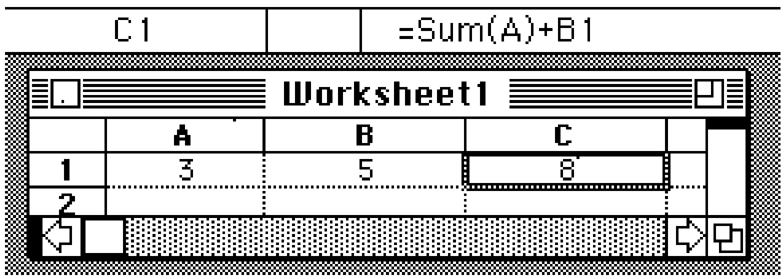
Each transaction, a query or assertion and its associated response, is displayed graphically on the screen in its own window. A scrollable *history* window displays a sequence of such transactions.



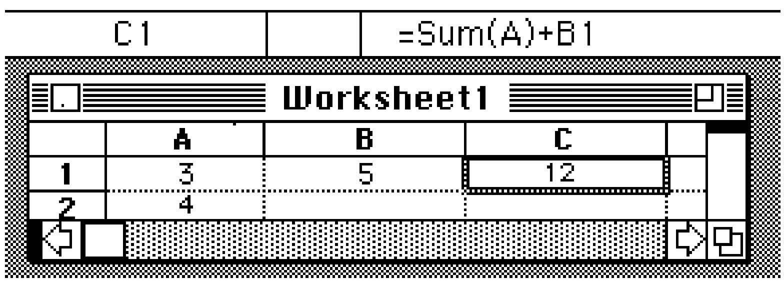
Window structure of a graphical interface to a knowledge representation system

The fact that many queries may be "running" at one time is natural in this system; users of multi-window graphical interfaces are comfortable with the concept of having more than one window active at once. The fact that the "answers" may change over time is also natural in this setting. The act of making an assertion might also cause previously displayed answers to change. We are assuming that being an open system means that when an assertion that affects outstanding queries is made, belief revision is performed that changes dependent assertions.

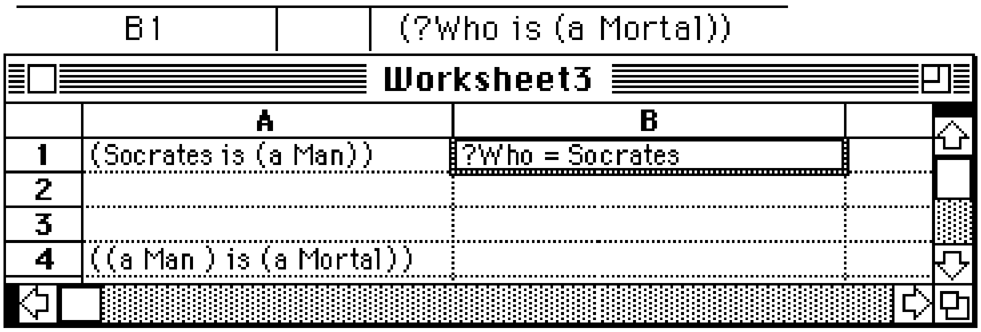
A transaction can be thought of as being like a *spreadsheet cell*, where the "formula" is the query or assertion, and the "value" is the displayed answer. Changing something in one cell may cause previously computed dependent values to change dynamically. As in a spreadsheet, one doesn't worry about "wasted computation" caused by recalculating cell values, unless it slows the response time of the system down to unacceptable levels.



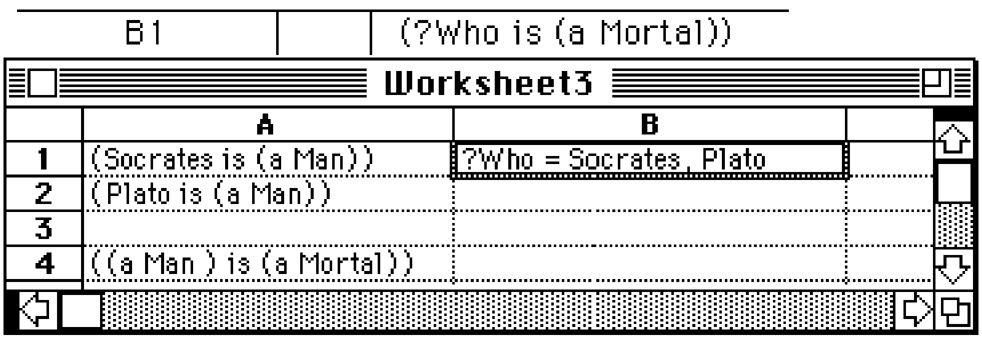
In a conventional spreadsheet, adding new information may cause the "answers" previously displayed to change.



A spreadsheet analogy for an interpreter interface



Adding new information to a knowledge base may cause answers to previous queries to change.



The window-oriented history also provides a convenient source of heuristics for controlling processing power in the system. The system could be biased toward alloting more processing power to the most recent queries at the expense of older ones.

A scrolling window displays only a subset of the extant queries. The system need only work hard on the ones currently visible. Scrolling the window can alter the allocation of resources. Each transaction window can have a pop-up menu operation for manually altering the amount of resources allocated to the query in that window. Discarded or inaccessible windows can have their query processes garbage collected to conserve processing resources.

Parallel systems may prefer breadth-first to depth-first search

An example of how the sequential bias affects decisions regarding inference procedures, consider the preference of traditional systems for *depth-first* versus *breadth-first* search. Breadth-first search is actually preferable for many reasons. Breadth-first search is guaranteed to find an answer if one exists, whereas depth-first search carries with it the danger of getting stuck in an time-consuming loop down one branch while a good solution lies unnoticed on another. This actually turns out to be a common practical problem in systems like Prolog. Breadth-first search solves problems with "shallow" solutions quickly, leading to good response time in the "easy" cases, while guaranteeing eventual success in harder ones. Uniform [Kahn 82] was a innovative "logic programming" system that accrued many advantages from replacing depth-first search with breadth-first, but the author later felt that it had to be abandoned on efficiency grounds, in favor of depth-first search.

The reason for preferring depth-first search on sequential computers is that it allows intermediate states to be held on a stack. In parallel systems, however, the efficiency advantage of stacks will all but disappear. Techniques for rapidly garbage collecting objects with short lifetimes, as outlined in [Lieberman and Hewitt 83] will make heap-allocated storage for search asymptotically as efficient as stacks.

A similar kind of prejudice surrounds the issue of whether to do *forward chaining* search or *backward chaining* search. Logic-based systems traditionally employ backward chaining, based on an efficiency argument appropriate to the case of sequential machines. In general, however, either forward or backward chaining may be more efficient, depending on the characteristics of the individual situation. Since in general it is difficult to forsee which one will be better, in a parallel environment probably the best thing to do is explore in *both* directions simultaneously, cooperating so that either direction may yield a solution.

Open systems should use both positive and negative information

A weakness of many representation systems, including logic-based approaches, is that they only represent *positive* information. Assertions about what is not true are often important in control of search. While undirected search through positive assertions such as `IS` links can often be exponential, negative assertions can serve to "kill off" unwanted search paths so as to keep the entire computation manageable.

[Hewitt and Kornfeld 82] call this effect *combinatorial implosion*. It involves having *proponent* processes that serve to try to establish goals, and *opponent* processes that work to defeat them. People generally become better at problem solving as they learn more, while adding more information to present AI systems generally results in poorer performance, since it increases the search burden. The judicious use of negative information points a way out of this paradox. The conjecture is that as people become more expert in a domain, much of what they learn tells them what *not* to think about, thereby reducing search. Minsky [Minsky 87] calls the mental components that do this *censors*.

We propose having two parallel hierarchies of descriptions, one related by Is links, the other by `Is-Not` links. The search for satisfying a query `(Is? Description-1 Description2)` should proceed by searching *both* hierarchies simultaneously.

The open systems critique also faults the classical logic-based approach to knowledge representation on its inability to deal with inconsistent information, as people can demonstrably do. Where logic dictates that a system holding inconsistent beliefs "spouts smoke" and falls apart, people seem to be able handle contradictions by localizing responsibility for them.

To allow the ability to represent *inconsistent* information, we propose that the answer to *every* query posed to the system consist of two parts: A set of paths the system discovers to establish the goal, and a set of paths the system discovers defeating the goal. If one of these sets is empty, that corresponds to the usual "consistent" answer. The system tries to optimize the special case of a consistent network for efficiency, on the assumption that most networks will be "mostly consistent". As long as the system propagates both proponent and opponent answers throughout, predicates can test for consistency, or procedures written to resolve conflicts.

Conventional systems, which throw away possibly conflicting information, don't permit sufficient mechanism to debug the appearance of contradictions. Even so-called "truth maintenance" [Doyle 83] systems, require each proposition to be marked with a unique truth value at any given moment. The assumption-based truth maintenance of deKleer is better in allowing inconsistent sets of assumptions. Touretsky [Touretsky 85] proposes "conditioning" a semantic network so that it will avoid storing inconsistent information due to representation of exceptional information or redundant links. But this discards information which may possibly be of use when the network is later modified. We recommend keeping the extra information in the semantic network, and applying transformations such as conditioning to the answers to queries instead.

A four-part inference procedure: positive and negative, forward and backward

Inference procedures are, of course, the heart of a knowledge representation system. In this section, we present a sketch of some characteristics that we advocate for inference procedures for the next generation of parallel systems. Few current systems embody all of these characteristics. We have implemented some of these as experimental interpreters for existing representing schemes, but since the design choices are made for future parallel systems, we as yet do not have a practical efficient implementation on contemporary serial machines.

A key idea is to reify the dynamic behavior of the search process by creating *search objects*, objects which represent partial or completed search processes. Since we expect that the usual situation will be running many searches in parallel, the user can observe and control search through interaction with interfaces attached to individual search objects. If we were considering a sequential search, the search object would contain the "stack" of goals searched. For a parallel system, the search object can be thought of as incrementally "copying" the part of the description network searched through to answer a query.

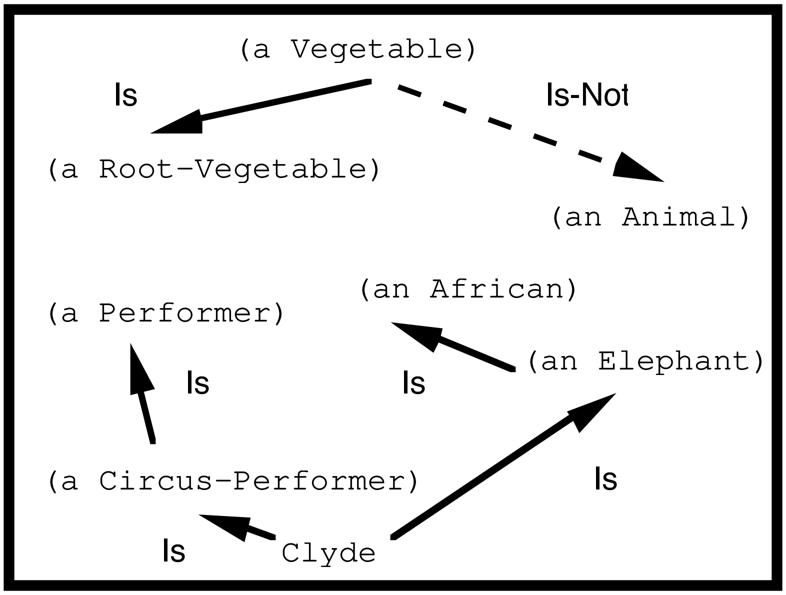
The issue of control of the inference process in knowledge representation systems, and the issue of human interface to the system are more closely coupled than people tend to realize. If the knowledge base is small, and inference is simple, the precise nature of the inference strategy will not be critical, as long as it does not introduce any inconsistencies. Similarly, an expert user is likely to be able to cope with any one of a number of presentation strategies for the information involved. The issues become more critical when knowledge bases become large and inference more complex.

In both areas, the major problem is one of *resource management*. For the interface problem, the issue is that the user does not have the time and energy to pay attention to all the information which the system could possibly show him or her. For the inference problem, the issue is that the machine does not have the time to compute all the inferences that might possibly be relevant. In both cases, control mechanisms are concerned with limiting the system to the activities deemed most important, at the cost of accepting the consequences of incompleteness.

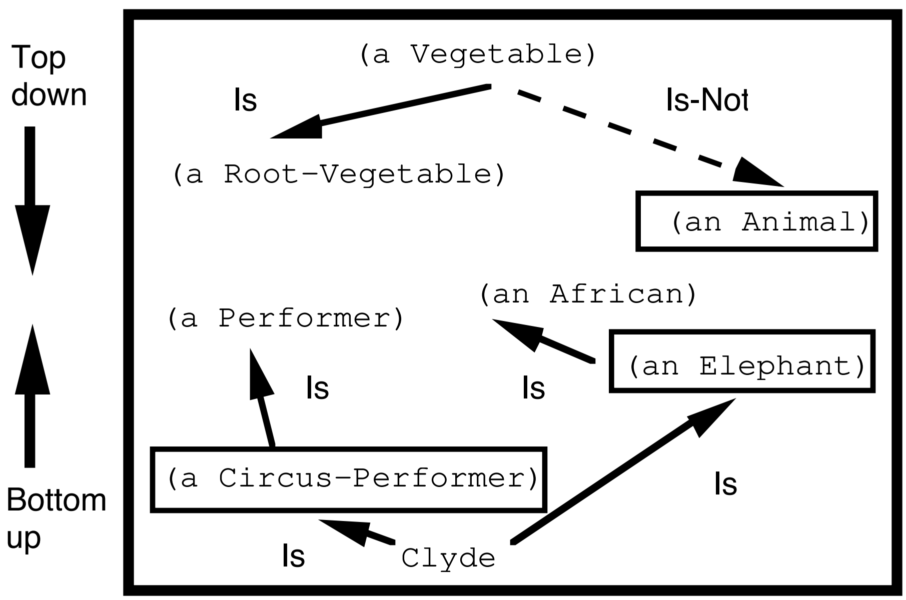
For both interface and inference management, the control heuristic is to afford priority to that information deemed most important, and elide that of priority below the point where the system can afford to expend resources [either compute power or screen space] on it. Where the system cannot decide a priori which of a number of competing strategies is best, the prudent thing to do is pursue and display both equally, falling back on the user to accept responsibility for the division of resources between them.

Such is the case with the choice of inference strategies. Should the system do forward or backward chaining? Should it search through positive or negative paths? Since there is no way to, in general, determine which is best for a given problem, the system pursues all four, in a "breadth-first" parallel manner. The user is given interactive controls to allocate the balance of effort, and can view their progress through interactive windows.

Visually, the system displays search objects as two graphs. One represents the progress of the search starting from the first argument of the `Is?` query, up the generalization hierarchy, the other the search from the second argument down the specialization hierarchy. For graphs larger than can be conveniently displayed on the screen, all the traditional techniques for level-of-detail control are applicable: scrolling, zooming, panning, elision, abbreviation. There are two kinds of links which may be asserted: `Is` links and `Is-Not` links. The following illustration shows the progress of a sample search.



A network is to be searched for (IS? Clyde (a Vegetable))



The progress of the search after a few steps

The system searches for connections between boxed descriptions

At each "tick" of the inference procedure, the system checks to see whether there is explicitly stored in the network an Is link asserting that the more specific description is described by the more general description. At the same time, it also checks to see if an explicit `Is-Not` link denies the connection between the descriptions. If either is found, they are added to the proponent or opponent parts of the solution, respectively.

If not, the search finds an intermediate description to which

```
(IS =Specific-Description =Intermediate-Description)
```

is directly connected in the network, and sets up the goal

```
(IS? =Intermediate-Description =General-Description)
```

to implement the forward-chaining part. Symmetrically, an intermediate description

```
(IS =Intermediate-Description =General-Description)
```

generates

```
(IS? =Specific-Description =Intermediate-Description)
```

for backward chaining. Matching is performed by unification. These recursive goals are not explored all at once, but incrementally, in a breadth-first manner, expanding the search graph a little each time processing resources are allocated to that particular search.

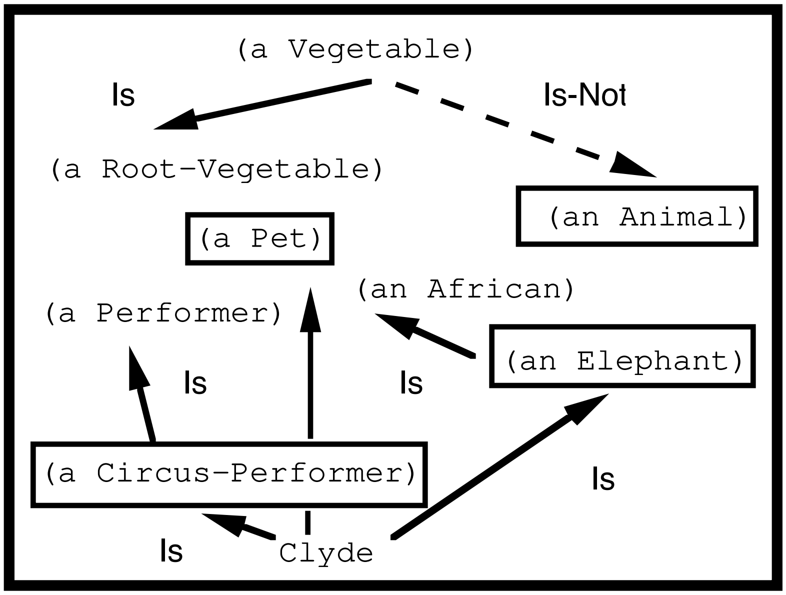
Combination of partial results of conjunctive searches is done by lazy and eager evaluation strategies. Exploration of conjuncts is driven by demand, determined by the attention of the user to presentations in the interface. Higher priority is given to searches under current observation on the screen. If the window is observed in greater detail [for example, sized larger on the screen], it is given higher compute priority. "Leftover" compute cycles are devoted to the presentation of less visible, or invisible information. Based on the progress of the search and interests of the user, the priorities of subsearches are dynamically adjusted through user feedback. Control is mediated through *sponsor* mechanism used in previous actor systems [Hewitt and Kornfeld].

The progress of the search is continuously displayed on the screen. The search object accepts a message at any time asking it for its current set of paths establishing and refuting the current query. This is the way of asking the current search to "return the best answer so far". Any answer returned, of course, can only be considered tenative, and may change as a result of further search or modifications to the network. The answers can be cached if necessary for efficiency.

An incremental redisplay algorithm inspired by that of the Emacs text editor manages display updates. Each displayed link is tagged with a *time stamp*. Whenever a modification to the structure is made, the time stamp is incremented. When the redisplay commences, a trace through the displayed structure is made, comparing the time stamps to that of the last redisplay. If the comparison shows that the structure hasn't been touched since the last redisplay, there is no need to redisplay it.

Changes to the network are reflected in changes in searches

A truly open system must leave provision for "changing its mind". The use of the search object to represent answers to queries records the dependencies of each reasoning path, and so can be used to revise "previously returned" answers to reflect revised beliefs without excessive recomputation. Each description network keeps a list of currently active search objects that are searching through it. When a message is sent to the description network to assert or retract an `Is` or `Is-Not` relation, it informs all its outstanding search objects of the occurrence. If an added or deleted link is not contained in the search object, then the addition cannot affect that search, and nothing further need be done. When adding a link to or from some description in the search, the search object must include the additional description on the list of those to be explored. When deleting a link, any answers that lie along that link become orphaned.



If (Is Clyde (a Pet)) is added, then (Is? (a Pet) (a Vegetable)) becomes relevant

Making an assertion affects searches already underway

We hope that the ideas presented in this paper point the way toward the kinds of implementation issues that have to be considered in building the next generation of knowledge representation systems for aritifical intelligence.

Bibliography

[Attardi and Simi 81]
Giuseppe Attardi and Maria Simi.
Semantics of Inheritance and Attributions in the Description System Omega.
In Proceedings of IJCAI 81. Vancouver, B. C., Canada, August, 1981.

[Baker, Hewitt 77]
Henry Baker and Carl Hewitt.
The Incremental Garbage Collection of Processes.
In Conference on Artificial Intelligence and

Programming Languages. ACM, Rochester, New York, August, 1977.

[Doyle 79] Jon Doyle.
Truth Maintenance Systems for Problem Solving.
Artificial Intelligence 12:231-272, 1979.

[Hewitt 86] Carl Hewitt.
The Challenge of Open Systems.
Byte , October, 1986.

[Kornfeld and Hewitt 81]
Kornfeld, W. A. and Hewitt, C.
The Scientific Community Metaphor.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1), January, 1981.

[Lieberman 86] Henry Lieberman.
Concurrent Object Oriented Programming in Act 1.
In Yonezawa and Tokoro (editor), Concurrent Object Oriented Programming. MIT Press, Cambridge, Mass., 1986.

[Lieberman and Hewitt 83]
Lieberman, H. and Hewitt, C.
A Real Time Garbage Collector Based on the Lifetimes of Objects.
CACM 26(6), June, 1983.

[Minsky 87] Marvin Minsky.
The Society of Mind.
Simon & Schuster, New York, 1987.

[Stirling and Shapiro 86]
Leon Stirling and Ehud Shapiro.
The Art of Prolog.

MIT Press, Cambridge, Mass., 1986.

[Touretsky 86] David Touretsky.
The Mathematics of Inheritance Systems.
Pitman, 1986.