N-1185-NSF

June 1979

COGNITIVE ECONOMY

Douglas B. Lenat, Frederick Hayes-Roth, Philip Klahr

A Rand Note
prepared for the
NATIONAL SCIENCE FOUNDATION
The materials incorporated in this work were developed with the financial support of the National Science Foundation under Grant MCS77-03273.

The Rand Publications Series: The Report is the principal publication documenting and transmitting Rand’s major research findings and final research results. The Rand Note reports other outputs of sponsored research for general distribution. Publications of The Rand Corporation do not necessarily reflect the opinions or policies of the sponsors of Rand research.
COGNITIVE ECONOMY

Douglas B. Lenat, Frederick Hayes-Roth, Philip Klahr

A Rand Note
prepared for the
NATIONAL SCIENCE FOUNDATION
Preface

This Rand Note describes work in progress, sponsored by the National Science Foundation under grants MCS77-04440 and MCS77-03273. It is addressed to a technical audience familiar with the concepts of artificial intelligence, knowledge representation, and knowledge engineering. Some knowledge of the LISP programming language is also assumed.

Although this Note does not contain final research results, it does present a rather complete description of the authors' paradigm: incorporating knowledge into programs so that they may make themselves run more efficiently. It is being disseminated at this time to stimulate discussion and interaction with colleagues on these issues.

Douglas Lenat is an assistant professor of computer science at Stanford University, and a consultant to The Rand Corporation.
Summary

Intelligent systems can explore only tiny subsets of their potential external and conceptual worlds. To increase their effective capacities, they must develop efficient forms of representation, access, and operation. In this Note we develop several techniques which do not sacrifice expressibility, yet enable programs to (semi-)automatically improve themselves and thus increase their productivity. The basic source of power is the ability to predict the way that the program will be used in the future, and to tailor the program to expedite such uses. Caching, abstraction, and expectation-simplified processing are principal examples of such techniques. We discuss the use of these and other economic principles for modern AI systems. Our analysis leads to some counterintuitive ideas (e.g., favoring redundancy over minimal storage in inheritance hierarchies).
Acknowledgments

We wish to thank Dave Barstow, John Seely Brown, Bruce Buchanan, Ed Feigenbaum, Cordell Green, Greg Harris, Barbara Hayes-Roth, Allen Newell, Herbert Simon, and Don Waterman for several discussions which have left their mark in this Note.
Contents

Preface ................................................................. iii
Summary .................................................................. v
Acknowledgments ................................................ v

1. INTRODUCTION ................................................... 1

2. THE ASSUMPTIONS .............................................. 6
   2.1 A Model of Intelligence .................................... 7
   2.2 A Model of Intelligent Program Organization ..... 8
   2.3 A Model of (Present) Computers .................... 9

3. SELF-MODIFICATION & -MONITORING: Intelligent Learning .... 11
   3.1 Dynamically Modifying Itself .......................... 11
   3.2 Dynamically Monitoring the Task Environment .... 19

4. CACHING: Intelligent Redundancy ......................... 24
   4.1 Types of Caches ........................................... 24
   4.2 An Example ................................................ 25
   4.3 Contrasts with Psychological Ideas of Economy .... 27
   4.4 Storing and Updating Cached Values ............... 28
   4.5 The Spectrum of Data Reductions .................. 33

5. EXPECTATION-SIMPLIFIED PROCESSING: Intelligent Focus
   of Attention ...................................................... 35

6. LEVELS OF ABSTRACTION: Intelligent
   Knowledge Structuring ......................................... 38

7. COGNITIVE ECONOMY REVISITED ......................... 41

References ................................................................ 42
1. INTRODUCTION

When we build an AI program, we often find ourselves caught in the following tradeoff: On the one hand, we want to develop our initial systems quickly and painlessly, since they are experimental and ephemeral. On the other hand, we want our systems to run as efficiently as possible, to minimize the temporal delays and to reduce the cpu time and space required. We are torn between expressibility and efficiency.

Many AI researchers cum language designers have focused on expressibility, the problem of rapidly constructing a working experimental vehicle.¹ They have produced some superlative software such as Interlisp's DWIM, File, and Break packages.² Four fundamental techniques utilized in highly expressive programs are: (i) reliance upon a very-high-level language, (ii) planning and reasoning at multiple levels of abstraction, (iii) inference by pattern-directed knowledge sources, and (iv) minimal, nonredundant representation, as in a canonical generalization/specialization hierarchy.

This paper addresses the second goal of AI programming, efficiency. We present techniques which do not sacrifice expressibility, yet enable programs to (semi-) automatically improve themselves and thus increase their productivity. The basic source of power is the ability to predict the way that the program will be used in the future, and to tailor the program to expedite such uses.³

¹This objective is apparent in the goals of SAFE [Balzer et al. 77], PSI [Green 77], EMYCIN [Feigenbaum 77], RITA [Anderson and Gillogly 76], ROSIE [Waterman et al. 79], and KRL [Bobrow and Winograd 77].

²[Bobrow and Raphael 74] is still an excellent discussion of the issues involved in such "AI language" efforts.

³This approach is sometimes used manually, as when a program is quickly coded in an expressive but inefficient form, used heavily, and then recoded in a much more optimised form. Thus, e.g., Ray Carhart spent 1977 rewriting DENDRAL (streamlining its CONGEN module) for a minicomputer.
The traditional approach to program optimization has assumed that the programmer characterizes the predicted program behavior (e.g., by explicitly providing assertions) or that static analysis can identify significant optimization opportunities. Three types of methods for analyzing program descriptions in this way include: (i) analyzing program flow and structure [Knuth 1968] [Dijkstra 76], (ii) designing data structures to be appropriate, and (iii) compiling (as in the FORTRAN compiler) and optimizing transformations of the program (as in [Darlington and Burstall 73 77] or [Balzer et al. 77].)

We advocate the use of methods more dynamic than these. Rather than improving the static description of a program, we propose to modify the program structure to adapt it to its operational environment.⁴

One valuable source of prediction comes from the program's experiences as it runs. If a program can "learn" from its experiences, it might try applying each piece of newly acquired knowledge to the task of specializing itself, modifying itself to run more efficiently in the current runtime environment. We believe that a program's "intelligence" can be increased in this way; that is, by increasing its ability to acquire appropriate knowledge, to organize that knowledge, and to refine the conditions under which that knowledge is recognized to be applicable. For any fixed quanta of manpower (sic) resources we expend, there is a limit to the size/complexity of programs which can be successfully implemented. This barrier might be overcome by programs which are self-extending, which actively do things to enlarge their input domain, their capabilities.

Three "dynamic" abilities which make a program efficient are:

- **Dynamic self-monitoring** (the ability to sense, record, and analyze dynamic usage) and self-modification (the ability to use that knowledge to redesign/recompile itself with more appropriate representations, algorithms, data structures; i.e., *intelligent learning*.)

⁴Some earlier automatic programming systems [Darlington and Burstall 73] [Lenat 75] [Low 74] [Kant 77] [Barstow 77] were designed to improve programs' efficiency, and many of their ideas have found their way into the techniques we now describe. Those systems had program construction, transformation, and optimization as their primary task; we are proposing general methods to provide self-improving abilities to other kinds of AI programs (understanders, theory formers, expert simulators, planners, etc.).
- Caching of computed results (storing the results of frequently-requested searches, so they needn't be repeated over and over again; i.e., intelligent redundancy.)

- Expectation-filtering (using predictions to filter away expected, unsurprising data, thereby freeing up processing time for more productive subtasks; i.e., intelligent focus of attention.)

A fourth ability, which we shall also discuss in this paper, is one of the more static techniques mentioned earlier, a kind of "designing appropriate data structures":

- Multiple levels of abstraction (redundant representation of knowledge at several levels of abstraction can be an economical way of structuring a knowledge base, especially if the program’s tasks are large and the resources available for different tasks vary widely in magnitude; i.e., this is a technique for intelligent knowledge organization.)

"Cognitive economy" is the term by which we describe such heightened productivity. Computer programs, no less than biological creatures, must perform in an environment: an externally imposed set of demands, pressures, opportunities, regularities. Extending this analogy, we find that cognitive economy is the degree to which a program is adapted to its environment, the extent to which its internal capabilities (structures and processes) accurately and efficiently reflect its environmental niche.

---

For a developed treatment of the import of the "task environment", see Newell & Simon [72].
Notice that representing a corpus of knowledge as a minimal (canonical) generalization/specialization hierarchy, with interpretatively-computed inheritance, is not cognitively economical: this technique favors expressibility, compaction of representation, at the expense of performance. It is true that a dog is a mammal, and a mammal is an animal, and from those two we could compute that a dog is an animal (see Fig. 1), but it is more cognitively economical to store one redundant arc than to recompute it frequently. Psychological studies [Hayes-Roth and Hayes-Roth 75] [Rips et al. 73] indicate just such redundancies being created and strengthened by repetitions.

Obviously, the economy of specific representations and related inference methods depends heavily on underlying machine architectures and costs. We assume that intelligent systems aim chiefly to produce useful results (e.g., novel hypotheses) as rapidly as possible. In short, they should search cleverly and fast. Thus, a priori notions of aesthetics or design economy have little bearing on cognitive economy. Massive redundancies in storage, processors, and inference processes seem more directly relevant. Furthermore, we suppose that, like people, intelligent software should eliminate slow, interpretive searches for previously solved problems (especially if the same problems are repeated often). To this end, cognitive economy may be achieved through dynamic compilation of action sequences. Such compiling would reduce the time-cost to effect the corresponding behavior; on the other hand, this efficiency would be gained at the cost of interpretability, interruptibility, and modifiability. These are desirable attributes where a program potentially needs continual changing. In typical situations, both efficiency of compiled forms and accessibility to declarative forms are intermittently needed. These different needs point again to the economic benefits of maintaining simultaneously alternative and redundant representations.

Every scientific theory is constructed in a rich context of surrounding theories, methods, and standards determining which experiments are reasonable ones to perform and which point of view to take when interpreting the results — in short, a paradigm. We feel it useful to articulate the "core" of our paradigm (the assumptions our theory rests upon) before delving into more detail about cognitive economy. For that reason, the next section is devoted to a presentation of our assumptions, including: a model for intelligence (Section 2.1), a model for how intelligent com-
puter programs may be organized (Section 2.2), and a model of the characteristics of present computing engines (Section 2.3). Later sections assume these models, and contain detailed discussions and examples of the main components of cognitive economy: dynamic self-monitoring and self-modification (Section 3), caching of computed results (Section 4), and expectation-filtering (Section 5). One interesting result that emerges in Section 6 is the prediction that as task size grows large enough, the same techniques that currently support expressiveness (e.g., multiple levels of abstraction) will become important for efficiency — for cognitive economy — as well.
2. THE ASSUMPTIONS

Summary: The claims we make presuppose that "intelligence" can be adequately modelled as heuristic search, guided by pattern-directed rules, implementable on a uniprocessor.

Our theories are erected upon a foundation of assumptions, including:

(i) We continually face searches in more or less immense spaces; intelligence is the ability to bring appropriate knowledge to bear, to speed up such searching. Increasing intelligence then comprises increasing knowledge, improving its organization, and refining the conditions for its applicability. How are these new discoveries made? Empirical induction (generalizing from experimental and other observations), analogy, and the criticism of existing theory all lead to new conjectures, new possibilities.

(ii) Intelligent systems can make the applicability of their knowledge explicit by representing that knowledge as condition-action rules (IF situation THEN appropriate reaction). Such pattern-directed inference systems (PDIS) can benefit from a schema representation (frame, unit, being, script, etc.), because this adds structure which the rule descriptions can exploit.

(iii) Current computing technology presents us with limited cycles, cheap storage, and expensive knowledge acquisition.

While the reader can turn immediately to a discussion of our detailed ideas on cognitive economy (beginning in Section 3), it may be useful to examine the above assumptions in more detail first.
2.1. A Model of Intelligence

Summary: Since "intelligence" depends upon bringing relevant knowledge to bear, it can be increased not only by adding new knowledge, but also by better organizing the existing knowledge.

Many human cognitive activities, including most of those commonly referred to as "requiring intelligence," can be cast as searches, as explorations through a search space, meandering toward some goal. We call a problem-solver "more intelligent" if it can work efficiently toward a solution even in the face of an immense search space and an ill-defined goal. Somehow, it is imposing more structure on the problem, and using that to search more effectively. How might it do this? According to our model of human intelligence, it accomplishes the task by bringing knowledge to bear, knowledge not supplied explicitly in the problem statement. This knowledge can be about problem-solving in general (e.g., how to recognize and break cultural blocks [Adams 74]) or about the problem domain specifically (e.g., which groups of chemical atoms can usually be treated as aggregate superatoms [Feigenbaum 77]).

This implies that a problem-solver can become more effective by (i) increasing its knowledge, (ii) better organizing its knowledge, and (iii) refining its criteria for deciding when various pieces of knowledge are applicable. In terms of production (If-Then) rules, these correspond to adding new rules, modifying the data structure in which rules are held, and modifying the conditions (If parts) of existing rules.

One very basic assumption we are making is that of continuity: the problem-solving environment does not change its character too radically, too rapidly. If the program abstracts from its experiences a plausible heuristic H, it may be able to analyze past scenarios to verify that possessing and obeying H would definitely have improved its performance (e.g., led to the current state of knowledge quicker); but its only justification for believing that H will help it in the future is the empirical evidence for the continuity of the environment. Learning can be translated effectively into action only when the structure of the environment has not altered too greatly.
A related assumption is that of irreversible knowledge accretion. Our body of facts and observations continually grows and only occasionally shrinks. Certainly the abandonment of a whole system of thought is a rare event indeed (ask Kuhn!), whereas the adoption of new systems of thought is part of the standard educational process.

New knowledge is discovered by specializing some broadly applicable but weak principles, or by generalizing particular experiences. The latter includes abstraction (condense some experiences into a heuristic which would have greatly aided us in the past if only we'd had it) and analogy (draw parallels not merely to other existing facts and heuristics, but also to the paths which led to their discovery.) One sometimes “spirals in” [Lakatos 76] toward precision (see Fig. 2).

2.2. A Model of Intelligent Program Organization

Summary: Since we want our AI programs to access, reason about, and expand their own knowledge bases, it is useful to represent such knowledge in a clean, modular form (e.g., employing any one of the current schematized representations.)

The preceding remarks about intelligent problem-solving apply equally to “hardware” and “wetware” alike. To be intelligent, computer programs must ultimately grapple with the tasks of knowledge acquisition, representation, and refinement. We cannot provide an absolute answer to how they should be organized, but we can posit some design constraints which have proven useful so far.

A very general heuristic in AI programming is the following: If your program is going to modify its own X, then make X as separable, clean, and explicit as possible. In our case, X can be instantiated as “knowledge,” or as “applicability conditions for each piece of knowledge.” In this case, the heuristic advises us to represent our knowledge in a separate, clean, explicit form, say as knowledge modules having some fixed internal structure, and also advises us to keep the applicability conditions for each knowledge module separate from the rest of the knowledge it contains.
This naturally leads us to a pattern-directed inference system, in which knowledge is broken into separate modules, each with an explicit set of relevancy tests [Waterman and Hayes-Roth 78]. Such systems arising in Pittsburgh may champion syntactic purity, while those arising in California may tend toward the baroque, but such variations should not obscure their common source of power. The PDIS architect breaks his program’s knowledge into a set of condition-action production rules.

Having a clean representation for rules means having a clean, precise, elegant language in which to express them. By structuring the conceptual knowledge of the system, by partitioning each module’s knowledge into several categories, a rule can condense an entire cumbersome description into a simple reference (often a single pointer). The popular schematized representations of knowledge (scripts for episodes, frames for static situations, beings for specialists, units for everything) enable a rule like “If there are no known methods specific to finding new examples of prime numbers, then...” to have its condition coded as a simple null-test on the “To-get” subslot of the “Examples” slot of the schema called “Prime Numbers.” By a judicious choice of slot types and subslot types, the system builder can reduce most triggering conditions to such quick checks on the state of various (subslots of) slots of schemata.

Additional knowledge is required to locate efficiently all the rules which might have their conditions satisfied in a given situation, and also to decide which rules to execute (obey) among those whose IF parts have triggered (been satisfied).

2.3. A Model of (Present) Computers

Summary: Since we are considering the problem of building computer models of intelligent behavior, many of our assumptions deal with the characteristics of present-day computers. They are the symbol manipulators we use, but were not designed for that general purpose.

The foremost quality of computing machines, as we know them today, is their limited “inferential bandwidth.” Although they provide virtually (sic) limitless storage capacity, these machines can process only a small fraction of this information
in a reasonable amount of time. Two primary reasons for this disparity between storage and generation capacities include: (1) knowledge acquisition proceeds slowly and expensively; and (2) inference processes depend on operations that ordinarily require cycles of uniprocessor systems.

The first problem places a cultural limitation on the development of intelligent machines. This development is restrained by our inability to express what we would like our machines to do or to convert these expressions into effective machine interpretable codes.

The second problem reflects a technological constriction of our potential. The so-called von Neumann bottleneck, caused by machine architectures that sequentially interpret program steps through a single logical processor, strongly motivates concern for programs that can minimize unnecessary computations.

In some future environment, perhaps neither of these problems will persist. In the meantime, however, we assume that two of the best ways to increase AI system productivity, derived from these current limitations, will be to expedite knowledge acquisition and to avoid unnecessary computing (for instance, by opting to "store" rather than "compute").
3. SELF-MODIFICATION & MONITORING:

INTELLIGENT LEARNING

We would like an intelligent system to be able to improve its performance dynamically. To accomplish this, a program must first be able to monitor its processing, i.e., to sense, record, and analyze its representations, data structures, algorithms, inference procedures, and the way they are currently being relied upon. It can then use this gathered knowledge to modify or redesign itself to perform more efficiently.

3.1. DYNAMICALLY MODIFYING ITSELF

Summary: We illustrate various ways in which a program might use to advantage knowledge gleaned dynamically: selecting (or perhaps even discovering) new data structures, representations, algorithms, etc. which seem better suited to the current runtime environment, the current user, the current problem, etc.

In order for a program to dynamically modify its own processing, it must examine, integrate and apply knowledge of the task domain, knowledge about programming in general, knowledge about the actual runtime environment, and knowledge gathered from observing its own processing behavior. The next section sketches ways in which dynamic program performance could be self-monitored. This section draws attention to the possibility of applying such usage data to modifying — or in the extreme re-synthesizing — the program.

Tasks can be specified anywhere from a pure "What" (desired input/output behavior) to a pure "How" (precise algorithms, data structures, representations, formats). It has been observed [Feigenbaum 77] that one of the goals of AI research is to push our ability from the How end toward the What end of that spectrum.
This often takes the form of designing a new higher-level language than any that hitherto existed, one which takes over more of the program implementation. We can see this as far back as assemblers replacing machine language, and as recently as KRL [Bobrow and Winograd 77] and the Molgen Units Package [Stefik 78] extending Interlisp.

Assuming, then, that the user desires merely to specify What, an intelligent language (automatic code synthesizer) must decide How. Upon what can it base its design? Previous efforts have employed knowledge about program transformations [Burstable and Darlington 77] [Balzer et al. 77], programming techniques and task-specific knowledge [Green and Barstow 78], and — if the input language is at all complex — information to enable the successful translation of the input specification into a sufficiently precise specification of What [Lenat 75]. A couple recent efforts have begun to guide the decision-making during code synthesis by algorithmic analysis [Kant 77] or even by aesthetics [Lenat 76]. But all these methods are static: they feed off a fixed base of initially-supplied knowledge, heuristics, constraints, suggestions, facts. The user supplies a specification for the desired program, and the knowledge base is employed to produce that target.

What might it mean for a program to modify itself dynamically? Suppose a hospital simulation program were specified, and on the basis of the user specification an event-driven system appeared to be the best design. During subsequent usage, it’s observed that the majority of questions refer to precise times of event occurrences. If this had been part of the initial specification, the design decision would have been made differently, say in favor of a synchronous time-step simulation. It would be no less useful a change in the system even at this late date. Even if this redesign is performed, the environment might keep changing in various ways, slowly, perhaps even cyclically. Future redesigns might be mandatory ad infinitum. The attractiveness of having a system which can automatically adapt itself to its current runtime environment is apparent.

The simplest kind of adaptive abilities along this line would be to select a data structure, a representation, or an algorithm from a set of well-known choices — and to reserve the right and the ability to change that decision later. One could gather knowledge about the relative strengths and weaknesses of each choice [Kant 77].

---

1 Two entries know which happened before the other, but not absolute times of day at which they occurred.
This knowledge could be in the form of rules which ask not only about static data, but which sample the runtime environment of the program as well. For instance:

"If one conjunct appears to be false more often than any other, make it the first one tested in the conjunction"

"If each person will use the program only once, briefly, it is not worth building a model of each such user"

"A function that is used frequently and changed rarely should be compiled"

Such design questions could be answered initially, during program synthesis, but our suggestion of automating the gathering of such data is a further step toward the "What" of artificial intelligence.

There is another possibility, much more exotic than the previous one, which deserves attention: the discovery—in "real time"—of new types of data structures and representations which would be useful for the program in its current environment.

Defining a new data structure is not as difficult as it may first appear: a data structure can be thought of abstractly in terms of a set of operations one wants to be able to perform on it (e.g., First, Last, Next, and Previous, for lists). The set of desired operations need not be minimal nor concern itself with which operations are most important, etc. The run-time usage of such a data structure can be sampled, and that can guide the search for an ever more appropriate implementation (e.g., in one usage, it might be very useful for Last to be efficient, and Interlisp's TCONCing might be chosen as an implementation; in another environment, it might be useful for Next and Previous to be efficient, and a doubly-linked implementation would be best).²

The two points here are (i) the implementation of a data structure may depend upon how it is commonly used in a particular spot in a program, by a particular

²Dave Barstow is one of the few AI researchers investigating issues such as these.
user, and (ii) a new kind of data structure may be defined abstractly merely by selecting a set of operations; this choice, too, can be made by examining the runtime data as it accumulates.

Defining a new representation is not quite so neat. In fact, humans have only managed to find a handful of representations to date. In lieu of any constructive suggestions along that line, we shall focus on extending a given representation:

3.1.1. EXTENDING A SCHEMATIZED REPRESENTATION

Summary: The Eurisko program extends its schematized representation by defining new types of slots. It uses a very simple grammar for defining new slots from old ones (legal moves), and a corpus of heuristic rules constrain that process (plausible moves).

For a schematized representation,3 "extension" could occur by defining new types of slots. The Eurisko program (an extension of the AM program [Lenat 76]) has this capability, and we shall briefly describe how this mechanism was developed.

First, we isolated four ways in which new slots are defined from old ones:

**STAR** (e.g., Ancestor $=_{df}$ Parent$^*$ which means a Parent of a Parent of... of a Parent of me; the general case should be apparent);

**UNION** (e.g., Parent $=_{df}$ Father $\cup$ Mother);

**COMPOSITION** (e.g., First-Cousin $=_{df}$ Child$\circ$Sibling$\circ$Parent; i.e., any child of any sibling of either of my parents);

**DIFFERENCE** (e.g., Remote-ancestor $=_{df}$ Ancestor $-$ Parent).

---

3Knowledge broken in pieces called schemata, frames, concepts, units, scripts, which in turn are merely collections of smaller pieces called properties, slots, facets, aspects.
These four operators which define new types of slots (\(\cdot\), \(\cup\), \(\circ\), \(\rightarrow\)) are called slot-combiners.

Next, we added to AM’s knowledge base a concept for each type of slot, and a concept for each type of slot-combiner. Figure 3 shows some of the Generalizations and Star concepts. Note in particular the way in which Generalizations is defined as Genl* — i.e., immediate generalizations, their immediate generalizations, and so on.

GENERALIZATIONS
ISA: Slot
SPEC: Extreme-Generalizations, Genl, Proper-Generalizations
GENL: Higher-nodes
WORTH: 875 (out of 1000)
AVG-SIZE: 7 entries
INVERSE: Specializations
ANALOGS: Supersets, Relaxations,
SLOT-COMBINER: Star
BUILT-FROM: Genl

STAR
ISA: Slot-combiner
WORTH: 488
ANALOGS: Repeat, Closure
SPEC: Nonzero-star
GENL: Countable-star
COMBINER-FOR: Generalizations, Specializations, Ancestor, ...
DEFN: \(\lambda (s) \ (\text{SUBST } s \text{ for SLOT in: } \lambda (c) (\text{AND } c \ (\text{CONS } c \ (\text{CONS } (\text{CONS } (\text{CONS } \text{SELF} (\text{GET } c \ \text{SLOT})))))))\)

[FIGURE 3: Generalizations and Star concepts]

Finally, we modified the way in which slot entries are accessed. To illustrate this, we choose a simple task in mathematics, whose paraphrase is, “What are all the generalizations of the concept ‘primes’?” The traditional way in which (GET PRIMES GENERALIZATIONS) would work is to examine the property list of PRIMES, looking for the attribute GENERALIZATIONS; if found, the entries listed there would be returned. If, as in Fig. 4, there is no such attribute, the call upon GET will return NIL (the empty list).
Anything
\text{genl}
\rightarrow
\text{Objects}
\text{genl}
\rightarrow
\text{Numbers}
\text{genl}
\rightarrow
\text{Primes}

\textbf{[FIGURE 4: network of concepts from Primes up]}

What we modified was the way in which any retrieval request of the form (GET C F) operates. In case the F attribute of C has no entries (or doesn't exist), we examine the definition of F and — if one exists — try to use it to compute the entries that could legally be stored on the F attribute of C. More precisely, before quitting we try to (GET F DEFIN), and if that succeeds we apply it to C. Let's continue the example of (GET PRIMES GENERALIZATIONS). As we see in Fig. 4 above, there are none recorded. So GET now calls itself recursively; our original call is replaced by ((GET GENERALIZATIONS DEFIN) PRIMES). But as Fig. 3 shows, there is no slot labelled DEFIN on the concept for GENERALIZATIONS. So we recurse one more time. By now our call has become

(((GET DEFIN DEFIN) GENERALIZATIONS) PRIMES).

Here is part of the DEFIN concept:
DEFN
ISA: Slot
SPEC: Necessary-Defn, Sufficient-Defn, Operational-Defn, Recursive-Defn
WORTH: 975
AVG-SIZE: 1.3 entries
ANALOGS: Statement, Algorithm
DEFN: λ(x) (APPLY* [GET (GET x Slot-Combiner) Defn] (GET x Built-From))

[FIGURE 5: DEFN concept]

Luckily, it does have a DEFN slot, so we end the recursion. Applying the entry stored there to the argument "GENERALIZATIONS," we see our original call becoming transformed into

[((GET (GET GENERALIZATIONS SLOT-COMBINER) DEFN)
  (GET GENERALIZATIONS BUILT-FROM))
PRIMES]

We see from Fig. 3 that the slot-combiner of Generalizations is "Star," and the argument (old slot) which it is built from is "Genl." So the entire call shrinks into (((GET STAR DEFN) GENL) PRIMES). The Star concept has an entry for its Defn slot; it turns the preceding call into ((λ(c) (CONS c (self (GET c GENL)))) PRIMES). This function first calls for (GET PRIMES GENL), which is NUMBERS, then calls itself on NUMBERS; that in turn calls for (GET NUMBERS GENL), which is OBJECTS, and calls itself recursively on OBJECTS; that calls for (GET OBJECTS GENL), which is ANYTHING, and the next recursive call terminates when it is discovered that ANYTHING has no GENL (no immediate generalization.) The list CONstructed and returned is thus (PRIMES NUMBERS OBJECTS ANYTHING). These four items are the legal entries for the GENERALIZATIONS slot of PRIMES, according to the definition of GENERALIZATIONS.
Notationally there is no distinction between slots which are "primitive" (values actually stored as attributes on a property list) and slots which are "virtual" (values must be computed using the slot's definition). A heuristic might refer to the Generalizations of Primes without knowing, or caring, whether that initiated a single access or a dizzy chase.

To define a new kind of slot, then, one need merely specify one of the slot-combiners and list the old pre-existing slots from which it is built. Thus we might define a new slot, by creating a new concept (calling it, say, "DG"), filling its Slot-combiner slot with the entry "Difference", filling its "Built-from" slot with the arguments "Generalizations Genl." This would be a new kind of slot, one which returned all generalizations of a concept except its immediate generalizations; the call (GET PRIMES DG) would return (PRIMES OBJECTS ANYTHING).

It is only a small extension to see how new kinds of slot-combiners can be defined. For instance, one which took two old slotnames as arguments, f and g, and defined a new slot which was \( f \circ g \circ f \), would be extremely useful (e.g., in database searches: see Fiskel and Bower [76]). In particular the crucial slot "Examples" is defined as \( \text{Spec} \circ \text{Immed-Exs} \circ \text{Spec} \).

Here is another example of how the expanded GET works: we consider an access on the PARENTS slot, when that slot is not primitive, but rather is defined as the union of slots for father and mother.

\[
\begin{align*}
&\text{(GET MERLE PARENTS)} \\
&\text{(((GET PARENTS DEFN) MERLE)} \\
&\text{(((GET DEFN DEFN) PARENTS) MERLE)} \\
&\text{(((\lambda (x) ((GET (GET * SLOT-DEFINER) DEFN) (GET * BUILT-FROM))) PARENTS) MERLE)} \\
&\text{(((GET (GET PARENTS SLOT-DEFINER) DEFN) (GET PARENTS BUILT-FROM)) MERLE)} \\
&\text{(((GET UNION DEFN) FATHER MOTHER) MERLE)} \\
&\text{(((\lambda (s1 s2) (\lambda (c) (LIST (GET c s1) (GET c s2))) FATHER MOTHER) MERLE)} \\
&\text{(((\lambda (c) (LIST (GET c FATHER) (GET c MOTHER))) MERLE)} \\
&\text{(LIST (GET MERLE FATHER) (GET MERLE MOTHER))} \\
&\text{(LIST SID BETTY)} \\
&\text{(SID BETTY)}
\end{align*}
\]
We have discussed how a new slot can be defined; consider now how a program is to know when/how to define a new one. The new type of slot might be defined for purely exploratory reasons (e.g., it's aesthetic to define "first cousins": the specializations of the generalizations of a concept). The slot's definition might be based soundly upon need — or the absence of need. For example, by monitoring usage, we might notice that many concepts have a large number of entries for their F slot, and infer that the F slot should be specialized. This would lead us to create several new slots, each having fewer entries on the average than F had, to cover the original F slot. E.g., if we noted that the Examples slot was heavily used, we might consider creating new slots like Boundary-examples and Typical-examples. The new kind of slot-combiner defined in the previous paragraph \((\lambda (f, g) f^* o g o f^*)\), could be derived from the fact that those slots which are most useful in the system (e.g., Isas, Examples) share that common form; or it could be motivated purely from symmetry, and the existing good uses of it would later be noticed in retrospect.

We close this section with a further exotic possibility. If the task domain of the system is the exploration of (a domain isomorphic to) programming, some new "theorem" might occasionally be discovered which can be incorporated into a data structure or algorithm (or even representation) to speed the whole system up or supplant it.

3.2. DYNAMICALLY MONITORING THE TASK ENVIRONMENT

Summary: The previous section illustrated how a program might profit from knowledge it gathered dynamically. This suggests that rather than working on the given problem exclusively, a program may be better off to expend some time learning (about the specific problem, its broader domain, or problem-solving in general). Note this suggestion would encompass traditional education (being told), empirical research (observing), and theoretical exploration (predicting). While such "very high-level" problem-solving strategies typify human problem-solvers (especially those of us who have rationalized spending twenty or more years "in school"), very few programs to date have employed any of these forms of learning to improve their operation.

Some programs (and wish-lists for program capabilities) have included learning
more about the problem being worked on, in this Note, however, we are stressing learning about the ways the program is being currently employed.

3.2.1. Learning by Being Told

There are many situations in which a user could provide advice to a program:

(a). The user might want to convey to the program some task-specific item: "Integration by parts won't work on this problem;" "This is a very hard problem." This can easily be input as part of the user's statement of the problem for the system to work on; e.g., the user might designate some methods as being preferred or constrained, he might estimate a resource allotment, etc.

(b). Another type of advice will apply permanently to the nature of the program's functioning (e.g., "Answers, even if uncertain, must be produced in real time;" "Hard problems will be starred.") Such knowledge can be reflected permanently in the design of the program, in its code.

(c). But there is an intermediate level of advice, some information the user knows will concern the next several tasks to be submitted — not just the present one and not all future ones: "The next four problems will be supplied in order of expected difficulty;" "Several of these problems share common assumptions and raise related issues;" "The user for the next entire session will be a physician;" "Until I find out why F34 returned NIL, I'll be in debugging mode, not production mode." Such advice is rarely communicable even to AI programs.

We believe this type of intermediate-level advice would improve programs if they had the opportunity to accept and use it. In general, we would like to advise the program which capabilities to consider and which to ignore during a prolonged period of solving closely related problems. A fixed (static) approach, for example, would be to define a few modes (e.g., Debug, Demo, Production, Input,...) and

4 McCarthy's advice-taker and Belzer's dialogue system (to get airline reservation programs written) would be told, Gelernter's theorem-prover (employing diagrams as analogic modes) could observe, Lenat's AM could observe and predict.
permanently build into the code all changes required to specialize the system for each mode. A flexible (dynamic) approach would provide a larger language in which the user could describe his current mode of usage of the program (e.g., by typing "USAGE of KNOWLEDGE-BASE will-be INCREASED VERY-MUCH"). The program would translate such statements into changes in its data structures, algorithms, explanations, core management strategies, etc. In the current example, heavy usage of the knowledge base might cause the program to swap out its compiler, optimizers, and other code, enabling it to hold many more knowledge chunks at a time in core. The ability to hold more knowledge in memory would directly affect the rate and probability of productive inferences.5

3.2.2. Learning by Observing

Spending a little time/space reconfiguring to reflect the current state of the world (the nature of the recent and succeeding inputs) seems worthwhile; the problem is how to know what that abnormal nature of the environment is. In the last subsection, we discussed the case where the user explicitly tells the program. But this is often impossible (as when the user is unaware of an impending irregularity, or when the "user" is Nature), and almost always a nuisance (learning the language in which to express such advice, remembering to do it, taking the time to do it).

The next step seems to be to allow the program to infer such facts about its runtime environment from observations. What kinds of self-monitoring can be usefully employed?

The method which comes to mind first, perhaps, is to save a complete record of everything that ever happened to this program: all the inputs it received (when, from whom), function calls and accesses it made internally, paths it chose/rejected (and why), and outputs it generated. The program could in principle deduce, from such an exhaustive history-list, patterns such as "very light usage of compiler;" "SQRT is always called on the result of SUM-OF-SQUARES."

The space costs of such an archive, and the temporal cost of the procedures to infer useful patterns, are surely prohibitive. The more economical approach is to

5Similar effects have been shown for humans: [Hayes-Roth and Walker 1979].
analyze in advance certain features we would wish to look for in a complete history record, and then to have the program record just enough dynamically to gather and maintain those features. Some of these will be domain-independent and commonly useful: Frequency of calls on various functions, requests for subtasks, etc.; Patterns in the arguments to calls on certain functions and inputs to the program; Patterns in the sequence of calls on functions; Patterns in functions' results (e.g., F34 always seems to return a User-Name). Some of the features will be domain-dependent specializations of the preceding features ('Average degree of input equation' is a special "pattern in the input.")

3.2.3. Learning by Predicting

The theoretician and the engineer share a powerful technique: manipulating a model of a system to generate predictions and identify those possible outcomes that would be desirable. This is a third route to learning. How might programs produce it? What sorts of theories, or models, can be employed to generate new information?

Models of the program's users can be valuable. This includes maintaining and extending profiles of individual users and whole groups. When a user opens a dialogue with Eurisko with the word "Let", it is a safe bet that he's a mathematician. Later, when he uses the word "function," his membership in that group makes it easy to determine which meaning he has in mind (instead of, say, what a physiologist or a computer scientist or sex therapist might mean by "function.") When the program wishes to mention \sqrt{-1}, it uses its MATHEMATICIAN model to choose an appropriate notation ("i" for mathematicians instead of, say, "j" for engineers). Models can be arranged in a hierarchy, with inheritance of features from more general groups to their subsets, a lattice ultimately terminating in models of individuals. User inputs would be recognized by various models, which would then hypothesize membership in that group (or explicate some future conditions which would confirm or deny such inclusion). Once triggered in this way, the user's apparent groups (and its generalizations) would actively filter his inputs and outputs, could modulate the program's actions (e.g., by changing the worth of the "Proof" concept, which in turn would change the priority rating of tasks on an AM-like agenda and thus alter the program's choice of what to work on next.) New user models can be created either top-down (e.g., when a model has many diverse instances or when it's proven historically to be very valuable, then try to split it into several specializations, each having a share of the general model's instances) or bottom-up (i.e., notice that a
group of individuals share a set of common characteristics, and create a new user group out of them).

The preceding kind of prediction is based on a theory of user types, embodied in a set of "typical representative" models. The same source of power can be used in many ways: stereotyping of events (as in scripts), stereotyping of static complexes (as in frames), stereotyping of communities (as in actors, beings, and contract nets). In all these cases, the source of power lies in quickly recognizing probable membership in (applicability of) some schema, after which all the rest of the schema (plus knowledge about all its generalizations) is presumed to be relevant. Programs might build up and utilize scenarios of how problems are solved typically in their domain. The building-up could be top-down via specialization or bottom-up via generalization, or even "sideways" via analogy. The models would be used for prediction (recognize relevant models and activate them, have them hypothesize probable future situations). Together, the models would form a theory of how problems are solved in that domain.
4. CACHING: INTELLIGENT REDUNDANCY

Summary: "Caching" the results of computations can dramatically improve the performance of many programs. Reasoning can be brought to bear to decide whether to cache, if so what to remember, and (later) whether or not to ignore the cached value and recompute it. We often refer differently to "caching" depending upon what it is that's being retained: open-ended, inductive searches can be condensed in hindsight (i.e. cached) into heuristics, deductive searches can be cached into much less branchy algorithms, subroutines can be cached into tables of frequently called arguments and resultant values, and variable quantities have their value cached simply by the process of binding.

After computing some result, it is typically put to some use, and the result itself is typically forgotten. If needed again, it is recomputed. Hardware designers have long recognized (both theoretically and empirically) the tremendous gain in efficiency afforded by caching: recording the result, at least temporarily, in case it is called for again soon. If that occurs, then no time need be spent recomputing it. Transferring that notion from hardware to software, and generalizing it slightly, we have the philosophy we shall call software caching, modifying memory to save computed results to speed subsequent accesses. In this section, we discuss this principle, giving examples of its use as well as identifying heuristics for determining when, why, and how to cache.

4.1. Types of Caches

The simplest case of caching occurs just after computing F(x), for some function called F with argument x, by remembering the value returned. Whenever F is called, we first check to see if a cached value is stored (e.g., associative retrieval on function/argument/value triples; e.g., scanning a list of function/argument pairs for which values are recorded, or hashing F and x, etc.) If no cached value is found, F(x) is computed and cached for future usage. If a cached value is found, then we must decide whether to accept it, or else to ignore it, recompute it, and store the
new value there. In chess, this might correspond to storing the common openings — a practice which uses a great deal of storage, yet is cost-effective because the book openings frequently recur.

But even this extravagant record-keeping is not complete: we have not saved information about the process of computing $F(x)$, the path that was followed, the traps, the blind alleys, the overall type of successful approach, etc. While much more costly to record than simply the value returned by $F$, this type of information can often be of extreme usefulness, especially for the purposes of generalizing, of monitoring the runtime environment of the program, of leading to new and better algorithms and data structures (see Section 3). The analogue in chess, e.g., was first hinted at by Newell, Shaw, and Simon in their early chess paper [63], then elaborated by Berliner in his thesis [74]: the idea that you should abstract out a description of a path through the search tree, so that the remainder of the search could partake of it. If one path leads to a discovered check against you, so will many others. You would be wiser to keep that threat in mind, instead of discarding all your state information and blindly backtracking. In other words, subtrees should communicate.

Similarly, in theorem proving, one could store (and perhaps even generalize) partial proof trees — and proof-finding experiences — for use in subsequent proof attempts. In STRIPS [Fikes et al. 72] we see a program caching partial plans so it can later reuse them.

4.2. An Example

To illustrate the process of caching values, consider again the way in which (GET PRIMES GENERALIZATIONS) was computed by Eurisko, as explained in section 3.1.1. Several function calls were required, and a fair amount of cpu time was expended. It would be quite economical to store the value once computed, to save time in the future. Eurisko simply stores (PRIMES NUMBERS OBJECTS ANYTHING) as the value of the GENERALIZATIONS attribute of PRIMES. When the call on GET is reissued, it tries to access this very spot. Though it failed previously, it now would succeed, and it would return the cached list almost instantly. No computing would be done, the concepts of STAR, DEFN, and GENERALIZATIONS needn't be in core now, and no space (even temporary space) would be required.
We saw earlier that notationally (e.g., to a rule) there was no need to distinguish an access of a primitively-stored slot from an access of a virtual, computed slot. Now we see that once the value is computed and cached away, there may be no telling it from a primitive one.\footnote{We might prefer to enclose cached values within brackets, to indicate their nonprimitive status.} Thus, with but a small one-time cost, our program runs as quickly as if all slots were primitive.

Note that in originally computing Generalizations of Primes, it was necessary to call for (GET GENERALIZATIONS DEFN), and the value of this virtual slot was also computed. The Eurisko policy is to cache this value also. It is useful because, when a request such as (GET DUCK GENERALIZATIONS) is received, it would otherwise have to be recomputed all over again. The definitions of slots are very slowly — if ever — changing, hence the recomputation of Defn of Generalizations is quite a rare event. Caching that value must be cost-effective in the long run.

In general, we see that caching a value for slot F of concept C is most applicable when the value can be expected to vary quite infrequently. In Eurisko, this gives us the flexibility to redefine a slot's definition if we wish, but (since this change of representation will be rare) the program will run just about as efficiently as if that capability were absent. This is analogous to compiling: so long as the definition of a function doesn't change too often, it's undisputedly worthwhile to compile it. The caching of DEFN and GENERALIZATION slots is not in any way special; the value of any virtual slot can be cached.

The careful reader may have spotted a difficulty in our caching policy: what happens if the value does change? One way to check for this would be to recompute the value, but of course if this is done too often it negates the benefits of caching. Eurisko faces this problem by adding some extra arguments to GET, parameters which describe a cost/benefit curve: for each amount of resources expended and each degree of precision of the result, this curve specifies precisely how acceptable that resource consumption / accuracy of solution pair is. One extreme of this is to provide an ironclad limit on resources and say "Do the best you can within these time/space/... limits." Another extreme (most common in present-day programs) is to specify an ironclad goal criterion ("Find an answer which is at least this good, no matter how long it takes to get.") We are calling attention to the space in between the two extremes.
To bring us back from the lofty heights of generality, let's see in more detail how we got Eurisko to do this. First, we linearized this space: we picked a few significant parameters of it, and defined a description to be merely a list of values for these parameters. When a call on GET was executed, and a cached value was encountered, a formula relating these extra arguments to GET would then determine whether to accept that cache, or to ignore it and recompute the value. Eurisko's parameters (extra arguments to GET) are: cpu time to expend, space to expend, whether the user can be asked about this, how fresh the cache must be, minimum amount of resources which must have been expended at the time the cache was written.

So (GET PRIMES GENERALIZATIONS 200 40 NIL 3 0) would allow any cached value to be accepted if it appeared that recomputing would take more than 200 milliseconds, or would use up more than 40 list cells, or if the value had been put there less than three Cycles ago. Otherwise, the cache would be ignored, a newer value would be computed, and it would be stored away. With it would also be recorded the following information: (i) the fact that it was a cached value, not a primitive one, (ii) how long it took to compute, (iii) how many list cells it used up in computing this value, (iv) the current Cycle (number of tasks worked on so far). The above call on GET might result in the following value being stored on the GENERALIZATIONS slot of PRIMES:

(*cache* (PRIMES NUMBERS OBJECTS ANYTHING) 54 8 9).

4.3. Contrasts with Psychological Ideas of Economy

When semantic networks first appeared in cognitive psychology, they led to the idea that function should follow form. In particular, since taxonomic knowledge structures were hierarchical, inference processes would presumably follow the same inter-category network paths each time the corresponding relation needed testing. It was for this reason that Collins & Quillian [69, 72] conjectured that the time required to verify semantic relations would correlate with distance in a corresponding "economical" semantic network. Thus, they showed that people ordinarily verified

---

2In Eurisko, a "Cycle" is the time it takes to work on the top-rated task on the top-rated agenda.
"a canary is a bird" faster than they could verify either of the two more remote relationships, "a canary is an animal" or "a canary has wings."

Subsequent research however showed that the needs of particular retrieval experiences, not a priori categorizations, determined which inference paths humans would follow and the associated response times. A simple counterexample to the presumed hierarchical retrieval schemes was found by Rips, Shoben and Smith [73]. They showed that people access a familiar but so-called "indirect" relation such as "a dog is an animal" much faster than the supposedly immediate but less frequent relation "a dog is a mammal." Using experimental materials and methods, Hayes-Roth & Hayes-Roth [75] found that human memory seemed plastic and adaptive in many of the same ways we have proposed for intelligent machine programs. Their results led them to "an adaptive model of memory, in which all learned relations are represented directly, with strength proportional to experience. There is also good evidence for redundancy in the network, with multiple routes connecting nodes representing particular pairs of concepts" [75, p. 519].

In another study, Hayes-Roth and Hayes-Roth [77] found evidence that people understand and remember English texts directly in terms of high-level lexical concepts, rather than low-level or primitive concepts. They argued that cognitive processes would presumably benefit if they could build directly upon the words or other meaningful units of familiar language. Any system that needed to interpret and store data in terms of primitive units, as is often assumed in language understanding theories, would encounter many slower and costlier searches. Both the empirical evidence and theoretical principles support the idea that human language understanding exploits many kinds of caching.

In short, we have chosen the term caching to refer to these types of redundant storage. Caching, to capture and exploit repetition and regularity, will benefit both humans and machines of limited processing capabilities.

4.4. Storing and Updating Cached Values

The caching process involves storage and updating. For both of these aspects, we can discuss details of when, why, how, and what. The decisions that arise are
made with the guidance of heuristic rules, many of which are illustrated below in
Figs. 6 and 7. We have omitted many of the very general "common-sense" heuristics,
and those which deal with details of Lisp programming. We also have specified the
rules in English, rather than in Lisp; we have freely used expressions like "recently",
although in any encoding that would have to be made much more precise.

Finally, note that these heuristics are not always relevant; they are applicable
for many AI programs, running on uniprocessing PDP-10-sized computers, running
Interlisp, with a human user watching and interacting. There would be other rules
for other machine architectures and languages. In other words, one could say that we
have only partially specified the IF-part (left hand sides) of the following heuristic
rules. An alternative would be to build up an entire knowledge base of heuristics
just to determine the applicability, the domains of relevance, of the following (as a
function of machine and language.)
When (and when not to)

Every time you have to call a lower order function.
You called a function and it took a long time to return.
The value returned has not changed since the last time you called it.
If F is called with the same argument x very often.
The amount of time to recompute the value would be very high.
   E.g., after a lucky accident, where a repeat might be very costly or even fail entirely to
duplicate your recent success at getting a solution.
If (frequency of usage)*(avg cost to recompute)/(freq of change) is high, then it pays off.
If the value is so critical that the cost-benefit criteria (e.g., the resource limits are enormous)
   always favor recomputation rather than chancing a blunder, Then don't cache.
If the function evaluates as fast as the caching mechanism itself, or if space is too tight,
   Then don't cache.

Why

Temporal cost of recomputing is higher than spatial cost of storage.
Related to "When" rules above; namely: in those situations, it’s cost-effective to store a
   cache.
Analogy to utility of hardware caching.
Ability to utilize cached values as usage data to induct upon.

Where

The value to be cached should have a precise storage place.
One extreme is an assertional database, with no structure.
We prefer to add slot and subslot structuring, with some domain-specific tinkering of those
two sets (especially the former, the set of slots.)
Thus an extreme example of prime numbers should be stored on a slot called extreme-
examples, on a schema called primes.
At access time, we look in the precise spot(s) where X would be, and if it isn't there, we
compute and cache it right there.
At worst, there should be a precise set of places where X might be, a well-defined path to
search down for X (e.g., a conjecture about prime numbers would either be an
Example of Conjectures, or a Conjecture of Primes.)

How

Before (re)computing F(x), the program presumably searched in the "right place" (see
"Where" above) to find a cache of F(x) if one existed. At that time, set a pointer to
that place, and when the value is found, place it there.
Called functions might suggest how to cache their value in higher calling caches. E.g., "my

[FIGURE 6: STORAGE PRINCIPLES]
value changes often so compile my definition and cache that, but don’t waste resources caching this value.”
Cache should be transparent and discardable (should be able to throw them all away if space is needed).
If there are multiple places to store it (e.g., you learn that X is a PARENT of Y, but PARENT is a virtual slot defined as MOTHER OR FATHER; if the cache is to be discardable, then we must store X under either Y’s MOTHER or FATHER slots.)
Then here are some strategies one might use:
Store it redundantly in several (all possible) places
Pick one of the places at random and store it there
Look up information about each place, to decide
Pick one of the places accurately by reasoning, testing, etc.

What

Store the returned value in any case.
Good policy to store a description of the call which led to this value being compute
E.g., make sure you store the amount of resources allotted, the time of day and the relative place in the computation (e.g., task number), whether the user was allowed to be queried, etc.
Along with the call, store data about the actual path to a solution. This includes algorithms used, time it took to compute, which other caches were used, etc.
It may be useful to store other intermediate results of the process that led to the value’s being computed successfully (e.g., a trap that was found and may crop up again.)
This also includes storing some abstract, partially-evaluated symbolic expressions which generalize the value and its computation.
If possible, predict (criteria, time-limit, etc.) conditions under which this will no longer be current. E.g., ”Depends on the definition of Y being known, and on there being no examples of Z known, and...”
Include a discussion of the certainty of the result. It may be worthwhile to stack the previous cached values, enabling the program to determine if (when, how, how often) they’re changing.
If you choose not to cache, you may wish to store a marker recording — and hopefully explaining — that decision.

How & When to eliminate caches

Just reverse the criteria in “When” (above), to see if it is no longer cost-effective to keep the cache.
Remember that F(x) will occupy varying amounts of space, varying with F, x, and time.
AM, e.g., eliminated largest cached values first.
Eliminate least frequently (or least recently) used caches first.
In general: eliminate the caches that are least cost-effective.

[FIGURE 6 (continued): STORAGE PRINCIPLES]
When

If the current request would have produced a different answer.
Usually, only do this if the current request will produce a distinctly better answer. E.g.:
If the current request specifies more precision in the answer.
If the resource allocation is high enough to get a better or more current answer than the
one cached already.
Trivially: If there is no cached answer available.
Typically: If x or the definition of F has changed since F(x) was last cached.

Why

Frame problem: if the value is old, it's hard to know for sure that the world hasn't changed
since it was computed.
If a new value is computed (either intentionally or accidentally), and its freshness (or the
other conditions of its call) makes it "better" than the existing cached value.

How

Discard the old value, or remember it elsewhere (maybe on disk).
Can use demon traps that flag the cache as out of date, without bothering to recompute the
value if not required to.
If the new value is found, simply store it.
The user may want to know about the updating.
Many other cached values may depend upon this one, and they may have to be ferreted out
and updated also, or at least some information should be left posted so they can later
determine that they may be out of date.
Typically: discard the whole old value, store just the new one.
If the program is very smart: When the world changes, try to propagate that change
through the network of caches, changing them. This is much better than erasing them!

[FIGURE 7: UPDATING PRINCIPLES]
4.5. The Spectrum of Data Reductions

Caching, as we have described it, is merely the lowest-level of a tier of data reduction processes, techniques for compiling hindsight into usable, more efficient, more procedural forms. To see this, consider the progression of ever more sophisticated tasks which programs perform [see Fig. 8]: accessing a datum, calculating a value from a fast algorithm, deducing a result by searching for a solution or proof in a well-defined space, inducing a result by a very open-ended search process. Often there is the opportunity to take experiences at one of these levels of sophistication and abstract them into an entity which is one level lower, yet which appears to contain almost the same power.

\[
\text{Access} \Rightarrow \text{Calculate} \Rightarrow \text{Deduce} \Rightarrow \text{Induce}
\]

[FIGURE 8: Progression of caching]

Consider first the way in which we reduce induction. After much experience with inducing, we produce a heuristic rule which enables the same inferencing to be done in a much more controlled space, in a much more deductive way. Heuristics reduce discovery tasks to mere (!) symbolic manipulation within a well-defined system (as in AM [Lenat 78]).

Deduction is reduced similarly. By many painful deductions, we gain knowledge of the space we are searching, e.g. by deriving many theorems about it. Those
Theorems can then be used to vastly simplify our search. Ultimately, the process becomes efficient calculation. A short, optimal algorithm may be found, e.g., which replaces a search for a solution (this has recently occurred with the "n queens" chess problem.)

Now we see that our caching process is the next step in this progression. It reduces a calculation (e.g., to traverse a particular access path to gather all Ancestors of a node) by a lookup, by a single access of a datum in memory.
5. EXPECTATION-SIMPLIFIED PROCESSING: INTELLIGENT FOCUS OF ATTENTION

Summary: For efficiency's sake, an intelligent system should be willing and able to add new facts, but should be eager to add surprising new facts. Surprises can only be noticed by contrast with expectations, so an intelligent system should maintain a context of expectations and filter incoming observations against that. Furthermore, expectations and surprises can aid an intelligent system in comparing its model and processing of the domain to the real world. Through such monitoring, discrepancies may be found and diagnosed, leading to changes in the model making it more consistent with observed behavior. Our discussion here centers on the importance of using such expectations to focus and filter intelligent processing.

The world bombards our senses with data, much more than we can process in detail in real time; yet we can't live in a hundred times real time. We survive by ignoring most of that data, or, more precisely, by knowing (almost immediately deciding) what can be ignored. We maintain a set of expectations, against which the incoming torrent is matched. Almost all of it will match those expectations, and we then need merely process the unexpected inputs, the surprising observations. We reserve our computing for those opportunities which promise us genuine new facts, rather than reconfirmation of known ones.

Such concentration on the abnormal is a natural side-effect of being guided by expectations — albeit usually stereotyped ones — about situations, events, and people. One is able to zip through most of what is encountered, match it immediately to the active schema, and move on. The only processing which is slow and tricky is that dealing with the remaining anomalies, exceptions, puzzles.

The more general position is that each slot indicates not merely a default, but also an indication of how important it is to confirm that value. Some slots will have meta-comments like "I'm worth computing or verifying", or just the opposite. For
When the ceiling is falling, it's easier to see; when all the lights go out except for the EXIT signs, it's easier to see them. Generalizing, we may state that it's worth a little computing to reason out whether (and maybe even how long) a slot-filling can be ignored.

(d). Predict and Prepare: This includes the whole continuum we discussed in the Caching section: bind a variable, store the triple \((F, x, F(x))\), store enough information so that \(F(x)\) could be recomputed very quickly, store information which would make \(F'(x')\) more easily computable (for \(x'\) similar to \(x\), and \(F'\) similar to \(F\)), store away a partial model of the world with good and bad features tagged. Notice that moving along this continuum, programs grow more flexible, more descriptive (less scalar), more dynamic. In all cases, the motivation for doing this storage is to (i) speed up the recognition of (the need to do) similar computations in the future, and (ii) speed up or even eliminate those computations when you decide they are relevant and you want the values they would return. These are the two uses of scripts, but also of cached constants (e.g., caching the value returned by OWNED-BY, a function from CARS to PEOPLE). A better example is user models: they range from a single bit (e.g., a NOVICE/PRO toggle), to a few numbers (e.g., in PARRY), to a dynamic concept-modelling scheme (e.g., in Eurisko).

In the interests of cognitive economy, we conclude that programs should build up expectations, monitor them, and change their models as needed. A natural mechanism for doing this is the use of pattern directed knowledge modules (PDMs), which can react to changing situations, to sudden disconfirmations of predictions. These can be used as triggers or demons; e.g., if you hear someone shout "Fire!" in a theater, that takes precedence over Jane Fonda. The second use of PDMs arises when an expectation is not met: the program then also has some chance of modifying the culprit rule(s) that made that false prediction (and/or adding new rule(s) which would make the right one.)
6. LEVELS OF ABSTRACTION: INTELLIGENT KNOWLEDGE STRUCTURING

Summary: An intelligent system's domain model is typically an approximation of reality. It is often the case that knowledge exists at several levels of detail and abstraction. The amount of detail required in intelligent processing depends on the problem-solving context. By appropriately structuring knowledge, a problem-solver can take advantage of abstraction by reasoning at the level appropriate to the given situation. We argue here that abstraction is crucial for intelligent systems which must process over large knowledge bases or within widely-varying time constraints.

In many real-life problem solving situations, the “goal” is not a single, precise answer. Rather, there is a cost-benefit tradeoff between the accuracy of the solution and the amount of resources consumed in producing it. Computer programs which are designed only to return exact answers lie at one extreme end of this curve, and rarely near the optimal point (of diminishing returns).

This presupposes a kind of “continuity” in the environment, as we discussed in Section 2.1. Namely, truncated search is assumed to be better than random selection of a legal move. Almost paradoxically, if the environment is mercurial, then the optimal point on the tradeoff will cause our program to be very expressive.

Let’s look at an example of the omnipresent need to approximate reality — a need which is well met by maintaining multiple representations of the world at various levels of abstraction; i.e., an entry in the knowledge base has all its generalizations also stored explicitly, redundantly.

A train accident occurs, and over two hundred people are injured. The only large nearby hospital is called, and asked if they can handle the load. The administrator queries his shiny new hospital operation simulator, and it begins to chug away. It
looks over the state of each current patient, each physician, each hospital resource. It begins simulating the arrival of the wounded, triage, their movement and temporary storage in hallways, etc. After quite a while of processing time, it shows that the hospital will be saturated and unable to care for any more incoming patients. The maximum they can accept is 157 wounded passengers. If “quite a while” is several hours (which it very likely would be), this type of analysis is simply inappropriate. There is a pressing need for a rough answer quickly; a human hospital administrator would answer right away, or after a very brief survey of a few parameters, and so should an intelligent program.

In addition to aiding us in quickly approximating reality, multiple levels of abstraction (“MLA”) are useful for analogizing; they can make that process easier, faster, and occasionally even more valid. Analogies can be recognized when two distinct concepts coincide at some abstract level. Conversely, if (corresponding slots of) two apparently related concepts (such as two homonyms) do not share related structure at higher levels of abstraction, that suggests that an analogy between them would be superficial and lacking in power.

Multiple levels of abstraction cost a certain amount to implement, both initially (man-hours of programming) and as the program runs (cpu time spent maintaining the redundancy, choosing which level to work on next, etc.) For problems of all sizes, this feature aids in the expression of new problems to the program, since they may be input at whatever level(s) seem appropriate to the user. Thus multiple levels of abstraction will always add to expressiveness, and will either detract or add to efficiency depending upon how small or large the task is. MLA is very useful when the program’s resources are curtailed or, even better, varied dramatically in magnitude from run to run (a program which was always limited to a 15-second execution time would be designed specifically for that task, but a program which might get anywhere from 5 seconds to 5 hours for the same simulation question could better benefit from multiple levels of abstraction.) MLA contributes more to efficiency as the quotient (task size)/(resource allotment) increases in size.

There is a more general result here: the methods which humans find useful for easy expression of problems may be useful to programs attacking very large problems. For instance, consider the decision about whether to do inference by a system of pattern-invoked experts, or rather to preprocess the triggering conditions into a big discrimination network. The former approach is more expressive, the latter more efficient — at least for small problems. For very large tasks, the need
to remain flexible grows very large. It is then more economical to retain pattern-directed control (the compilation of the patterns is so complex the whole procedure would have to be redone frequently during the course of execution, if we opted for that alternative instead.) A case in point is the HARPY speech understanding program [Lowerre & Reddy 79]: whenever the grammar changes — at any level — the entire speech recognition network must be recompiled. HEARSAY II [Lesser and Erman 77], while slower in performance, has sufficient flexibility (from several independent pattern-directed knowledge sources) to obviate such monolithic integrative reccompilation delays.
7. COGNITIVE ECONOMY REVISITED

Summary: We identified three dynamic characteristics and one static characteristic that serve the needs of intelligent adaptation. First, we showed that most inferential systems can benefit from learning about their task environment and their own behavior. For example, they can exploit new schemata or different slot-names to simplify and restructure their knowledge. Second, the need to explore large search spaces with some repetitive regularity motivates the use of caching to save partial results. We described a variety of techniques to implement caching, and explained how caching specific results is one of a spectrum of methods that can make trade-offs among precision, speed, certainty, and generality. The third dynamic capability we identified was expectation-filtering. In general, intelligent systems need to exploit their knowledge about typicality to reduce their cognitive load and increase their attention to important data. In many situations, we believe that expectations can both reduce processing requirements for handling ordinary events as well as simplify the identification of surprising events. Finally, we argued that multiple levels of abstraction (MLA) provide economical representations for complex tasks. In many simple environments, the advantages of MLA are minimal and may not even justify their costs. Thus, routine tasks may warrant the kind of knowledge compilation that would convert an initial, expressive MLA knowledge-base into a fast, integrated, and largely uninterpretable code. Conversely, as task complexity and variability increase, MLA increasingly provides a basis for intelligent and rapid restructuring.

In the years to come, AI programs will employ greatly expanded knowledge bases and, as a consequence, they will explore increasingly open-ended problem spaces. Already, a few existing systems show signs of having more potentially interesting things to do than they have resources to pursue (e.g., AM, Eurisko). In the past decades of intelligent systems R&D, several design concepts have emerged in response to contemporary needs for creating ever larger knowledge bases. For example, many researchers proposed multiple levels of abstraction and automatic property inheritance as keystones of efficiency or "cognitive economy." We believe that the value of such mechanisms derives largely from their usefulness in describing initial knowledge bases. Once an intelligent system begins to explore the consequences of its knowledge and to solve novel problems in a dynamic environment, it
needs to adapt its knowledge to achieve faster and more profitable retrievals.

A theory of cognitive economy should explain why knowledge needs to be adapted and should prescribe how to do it. In this paper, we have tried to lay the groundwork for such a theory. To date, cognitive science has developed few systems and generated few analytical arguments on which to base such a theory. In contrast, we have approached this problem by defining the issues, illustrating situations that motivate various types of economies, and proposing design and re-design heuristics that support intelligent adaptations. These heuristics embody the gross features of an eventual theory and suggest numerous paths for future research.

In conclusion, we have tried to show what cognitive economy is and is not. It does not consist of a set of static knowledge-base design principles, such as those proposing taxonomic concept structures with automatic property inheritance. Rather, cognitive economy is a feature of those intelligent systems that learn to solve problems efficiently and consequently realize more of their lifetime potential. Toward that end, we have proposed an initial set of four basic design characteristics. We anticipate these characteristics will find widespread application in many future AI systems. As knowledge bases expand and basic software obstacles are overcome, AI systems will increasingly address the same question facing intelligent humans: "What would I most like to accomplish next, and how can I do that economically?"

References


Barstow, D., “A knowledge-based system for automatic program construction,”


Kant, E., "The selection of efficient implementations for a high level language," Proc. ACM SIGART-SIGPLAN Symp. on Artificial Intelligence and Programming Languages, SIGART Newsletter 64, 1977, 140-146.


