

# COGNITIVE ECONOMY

## in Artificial Intelligence Systems

Douglas B. Lenat, Computer Science Dept., Stanford University, Stanford, Ca.

Frederick Hayes-Roth, Information Sciences Dept., The Rand Corporation, Santa Monica, Ca.

Philip Klahr, Information Sciences Dept., The Rand Corporation, Santa Monica, Ca.

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 paper 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).

## 1. INTRODUCTION

When we build an AI program, we often find ourselves caught in a tradeoff between expressibility and efficiency. Many AI researchers cum language designers have focused on *expressibility*, the problem of rapidly constructing a working experimental vehicle. 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*: getting programs to use as few resources (time, space) as possible. 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.

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 [*à la* Knuth and Dijkstra], (ii) designing data structures to be appropriate, and (iii) compiling (as in the FORTRAN H compiler) and optimizing transformations of the program (as in [1,2]).

We propose the use of methods more *dynamic* than these. Rather than improving the static description of a program, we advocate a *plastic* program structure which adapts to its operational 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 quantum of manpower 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 reason — and reprogram themselves — to enlarge their input domain, their capabilities.

Abilities which make a program *efficient* include:

- **Dynamic self-monitoring** (the ability to sense, record,

<sup>1</sup>This research was supported in part by the National Science Foundation under grants MCS77-04440 and MCS77-03273.

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*.)

- **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*.)

"*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.

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, but it is more cognitively economical to store one redundant arc than to recompute it frequently. Psychological studies [3] indicate just such redundancies being created in human memory.

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. But we are advocating efficiency in *addition* to, not in *lieu* of, expressibility (interpretability, interruptibility, and modifiability). In typical situations, both efficiency of compiled forms and accessibility to declarative forms are intermittently needed. These different needs point

to the economic benefits of simultaneously maintaining alternative and redundant representations.

## 2. THE ASSUMPTIONS

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:

- (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.

- (ii) Since we want intelligent systems 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.) Its *applicability* can be made explicit by encoding it as condition-action (if/then) rules.

- (iii) Current computing technology presents us with limited cycles, cheap storage, and expensive knowledge acquisition. They are the symbol manipulators we use, but were not designed for that purpose.

## 3. SELF-IMPROVEMENT

NOTE: Due to IJCAI length limitations, the authors have been forced to eliminate many sections of this paper; usually, at least their summaries remain. The complete version can be obtained as [5].

### 3.1. DYNAMICALLY MODIFYING ITSELF

**Summary:** We illustrate various ways in which a program might use to advantage knowledge gleaned dynamically: selecting [6] (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.

### 3.1.1. Extending a Schematized Representation.

For a schematized representation,<sup>1</sup> "extension" could occur by defining new types of slots. The Eurisko program (succesor to the AM program [4]) has this capability, and we 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 =<sub>d</sub> Parent\* which means a Parent of a Parent of... of a Parent of me; the general case should be aparent);  
**UNION** (e.g., Parent =<sub>d</sub> Father  $\cup$  Mother);  
**COMPOSITION** (e.g., Cousin =<sub>d</sub> ChildoSiblingoParent);  
**DIFFERENCE** (e.g., RemoteAncestor =<sub>d</sub> Ancestor - Parent).

These four operators which define new types of slots (\*,  $\cup$ , o, -) 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 1 shows part 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.

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. Assume that there is no such property recorded for PRIMES. If we looked, we would find a property called GENL (*immediate* generalization), whose value would be NUMBERS. Similarly, the NUMBERS concept has only a GENL slot, containing the entry OBJECTS. Anyway, since there is no GENERALIZATIONS attribute, the call upon GET will return NIL (the empty list).

In Eurisko, we modified 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 DEFN), and if that succeeds we apply it to C. Let's continue the example of (GET PRIMES GENERALIZATIONS). As we stated in the last paragraph, there are none recorded. So GET now calls itself recursively; our original call is replaced by ((GET GENERALIZATIONS DEFN) PRIMES). But as Fig. 1 shows, there is no slot labelled DEFN on the concept for GENERALIZATIONS. So we recur one more time. By now our call has become (((GET DEFN DEFN) GENERALIZATIONS) PRIMES).

| GENERALIZATIONS                                             | STAR                                                                                                                               | DEFN                                                                                                 |
|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| ISA: Slot                                                   | ISA: Slot-combiner                                                                                                                 | ISA: Slot                                                                                            |
| SPEC: Extreme-Generalizations, Genl, Proper-Generalizations | WORTH: 488                                                                                                                         | SPEC: Necessary-Defn, Sufficient-Defn, Operational-Defn, Recursive-Defn                              |
| WORTH: 875 (out of 1000)                                    | ANALOGS: Repeat, Closure                                                                                                           | WORTH: 975                                                                                           |
| AVG-SIZE: 7 entries                                         | COMBINER-FOR: Generalizations, Specializations, Ancestor, ...                                                                      | AVG-SIZE: 1.3 entries                                                                                |
| INVERSE: Specializations                                    | DEFN: $\lambda (s) \text{ (SUBST } s \text{ for SLOT in: } [\lambda (c) \text{ (CONS } c \text{ (self (GET } c \text{ SLOT))})])}$ | ANALOGS: Statement, Algorithm                                                                        |
| ANALOGS: Supersets, Relaxations,                            |                                                                                                                                    | DEFN: $\lambda (x) \text{ ( [GET (GET } x \text{ Slot-Combiner) Defn] (GET } x \text{ Built-From))}$ |
| SLOT-COMBINER: Star                                         |                                                                                                                                    |                                                                                                      |
| BUILT-FROM: Genl                                            |                                                                                                                                    |                                                                                                      |

<sup>1</sup>Knowledge broken in pieces called schemata, frames, concepts, beings, association lists, units, scripts,... which in turn are merely collections of smaller pieces called properties, slots, facets, aspects.

FIGURE 1: Generalisations & Star & Defn concepts

Luckily, the DEFN concept does have a DEFN slot (see Fig. 1), 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. 1 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 (( $\lambda$  (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). In particular the crucial slot

"Examples" is defined as  $\text{Spec}^* \circ \text{Immed-Exs} \circ \text{Spec}^*$ .

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 into several new slots.

### 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.

## 4. CACHING

**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.

Hardware designers have long recognized the tremendous gain in efficiency affordable by *caching*: recording a result, at least temporarily, in case it is called for again soon. Here we consider an analogous technique: *software caching*. The simplest case occurs just after computing  $F(x)$ , for some function  $F$  called with argument  $x$ : simply store the value returned. Whenever  $F$  is later called, check first to see if a cached value is stored.

A more sophisticated kind of caching would involve saving 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. The analogue in Search is to have the subtrees intercommunicate.

To illustrate the process of caching values, consider again our old access, (GET PRIMES GENERALIZATIONS). After computing (PRIMES NUMBERS OBJECTS ANYTHING) as the answer, Eurisko simply stores that list as the value of the GENERALIZATIONS attribute of PRIMES. If the same call to GET is reissued, it tries to access this very spot. Though it had failed previously, it now would succeed, and it would return the cached list almost instantly. Thus, with but a small one-time cost, our program might run 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.

What happens if the value *does* change? Eurisko handles

this in a passive manner: it doesn't attempt to maintain a completely consistent, up-to-date knowledge base. Rather, when a call on GET is executed, and a cached value is encountered, a formula then determines whether to accept that cache, or to ignore it and recompute a fresher value. Each call on GET is supplied with a few extra parameters which specify how much cpu time and space are budgeted for this particular function call, whether the user may be queried about this, how recent a cache must be to be acceptable, and a 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 Tasks 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 Task number. 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 6 9).

[At this point in [5] comes a discussion contrasting our ideas with psychological ideas of economy. Similarly omitted here is a lengthy section containing heuristics for Storing and Updating cached values: when (not) to, how to, and what to.]

## 5. Expectation-Filtering

**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. Such expectations can be used to focus and filter intelligent processing.

## 6. CONCLUSIONS

**Summary:** We identified three characteristics 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.

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.

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 three 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?"

## Acknowledgements

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 paper.

## References

- [1] Balzer, R., Goldman, N., and Wile, D. "Informality in program specifications," 5IJCAI, Cambridge, 1977, 389-397.
- [2] Darlington, J. and Burstall, R. M. "A system which automatically improves programs," 3IJCAI, Stanford, 1973, 479-485.
- [3] Hayes-Roth, B. and Hayes-Roth, F. "Plasticity in memorial networks," Journal of Verbal Learning and Verbal Behavior, 14, 1975, 506-522.
- [4] Lenat, D. B. "AM: an artificial intelligence approach to discovery in mathematics as heuristic search," Memo AIM-286, Stanford AI Lab, 1976.
- [5] Lenat, D.B., Hayes-Roth, F., and Klahr, P., "Cognitive Economy," Memo HPP-79-15, Heuristic Programming Project, Computer Science Dept., Stanford University, 1979; also issued as Rand Report N-1185, Rand Corporation, Santa Monica, Ca., 1979.
- [6] Low, J. "Automatic coding: choice of data structures," Memo AIM-242, Stanford AI Lab, 1974.