MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 339 September 1975

VERY LARGE PLANNER-TYPE DATA BASES

Drew V. McDermott

This paper describes the implementation of a typical data-base manager for an A.I. language like Planner, Conniver, or QA4, and some proposed extensions for applications involving greater quantities of data than usual. The extensions are concerned with data bases involving several active and potentially active sub-data-bases, or "contexts". The major mechanisms discussed are the use of contexts as packets of data with free variables; and indexing data according to the contexts they appear in. The paper also defends the Planner approach to data representations against some more recent proposals.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-75-C-0643.

Very Large Planner-Type Data Bases

Drew McDermott

Contents

I *INTRODUCTION*

One of the durable ideas in AI is the pattern-oriented data base developed by Carl Hewitt for the original Planner programming language. <Hewitt, 1971> It has appealed to many people because it seems to be a good framework for implementing a "blackboard" communication channel between processes. <Newell, 1962>

Such a data base operates by indexing records so that they may be accessible with any of several different key patterns. This is the same as the more traditional "file inversion" <Rivest, 1974>, except that the patterns are more complex. The data base also provides primitives for copying and combining collections of data. These facilities can be used to implement controlled "deductive" procedures which limit their searches to formulas that have been included in the current local data base, and which are likely to match the current goals. When such data bases were implemented (by various people in different ways, e.g., <Rulifson *et. al.*, 1972>, <Sussman and McDermott, 1972>), they were found to help overcome the obstacles facing previous deductive programs.

However, recently such data bases have come in for some criticism. Some critics feel that sets of pattern-accessible records are wrong (or misguided) as a data structure for AI applications. They usually recommend some more strongly organized structure; they often appeal to the concept of "frame." <Minsky, 1974>

This criticism is buttressed by the feeling among many people that Planner-type data bases are intrinsically hard to organize except for small "toy" problems. Scott Fahlman <1975> has argued that the usual methods of organizing them will break down completely when we try to implement data bases of the size the human brain must contain.

Although it is possible, and done all the time, to misorganize a Planner data base, I feel that the critics are being unfair. In fact, it seems as though the frame idea flows naturally from the nature of Planner-type data base mechanism, including the "context" mechanism of QA4 and Conniver. In particular, the notions of hypothesis-driven recognition triggered by suggestion demons <Fahlman, 1973>, of default values in frame slots, explaining discrepancies, frame transition, etc., seem to be descended from ideas like consequent vs. antecedent reasoning <Hewitt, 1971>, making assumptions based on partial evidence <McDermott, 1974a>, and debugging slightly wrong data bases <Sussman, 1975>. As far as I can tell, a frame is just a particular way to use a Conniver context.

Most frame proposals, however, do not suggest further embellishments of an already sound framework, but instead tend to propose archaic data structures with lists of slots and values, or a small set of interfaces and possible messages. The only reason for this seems to be a widespread acceptance of the second criticism, that a large Planner-type data base could not be implemented efficiently. By "large" I mean large both physically, requiring secondary storage with today's technology; and organizationally, requiring care to avoid useless computation. It is the purpose of this paper to show that the case is unproven. Besides having this narrow technical intention, I intend to say a little about what kinds of operations ought to be efficient in an AI data base, and which proposed data structure might be the best.

II *A TYPICAL DATA BASE MANAGER*

In this section, I will describe the implementation of a typical Planner-like data base manager (TDBM). The description is drawn from my experience with several versions of Conniver <McDermott and Sussman, 1973>, but is somewhat idealized, so as to distract you from merely haphazard faults of existing systems and show you their deeper problems.

II.A Behavior

There are three important features of a typical data base management system: "partial-match" record retrieval, multiple data bases, and derived data.

II.A.1 "Partial-match" Record Retrieval

Any record may be accessed by specifying any subset of its components, with each specified component in its proper position. The unspecified, or "don't care," parts may be indicated by question marks. For example, the record (P (f a)) may be referred to by keys (P ?), (? (? a)), (P (? a)), etc. Records in the data base, called *items*, are *indexed* when they are put there (*added*); and *unindexed* when they are *removed*. The retrieval function is FETCH, which returns all the present items which *match* its pattern argument. For example, if (P (f a)), (P (f b)), and (P c) are present,

```
(FETCH '(P (f a))) returns {(P (f a))}
(FETCH '(P (f d))) returns {}
(FETCH '(P (f ?))) returns {(P (f a)), (P (f b))}
(FETCH '(P ?))     returns {(P (f a)), (P (f b)), (P c)}
```

(A note on notation: as in MACLISP <Moon,1974>, " '*S-expression*" is an abbreviation for "(QUOTE *S-expression*).")

Sometimes I mean by "match" that there is some mapping, from occurrences of "?" in the fetch pattern to S-expressions, which makes the fetch pattern equal to the item retrieved. In practical systems, the matcher is more complicated. It may be thought of as a unification algorithm <Robinson, 1965>, in which labelled "don't cares" play the role of variables. For example, (FETCH '(Q ?X ?X)) should find (Q a a) but not (Q a b). I will more or less ignore this kind of matching in this paper.

However, there is an associated peculiarity I must mention. Our TDBM must allow "don't cares" in the item records. This is because users will want to be able to model propositions with items, and need to have free (universally quantified) variables in items like ((IS MAN ?X) ⊃ (MORTAL ?X)). For our purposes, this may be treated as ((IS MAN ?) ⊃ (MORTAL ?)). We want to be able to find it with a fetch pattern like (? ⊃ (MORTAL FRED)).

II.A.2 Multiple Data Bases

In AI applications, there are several reasons to have many different data bases to select from at a given time. However, these cannot be represented by anything as clumsy as separate files or indexes. It has to be cheap to copy an entire data base in such a way that routine changes to the copy (additions and removals of data) do not affect the original

This is done by representing a data base as a list of *layers* which represent the history of changes to an original empty set. With each layer

are stored the differences between the data bases that contain it and those that do not. Such a list of layers is called a *cxt* (pronounced like "context," Conniver's misleading term). I will reserve the term "data base" from now on to mean the set of all data in all cxts.

For example, a problem solver may model time as a sequence of cxts, each with one more layer. In CXT1, it may have the items (ON A B), (ON B TABLE), (ON C TABLE), representing the scene of Fig. II.1(a).

Figure II.1

In contemplating an action, the problem solver may make a new layer (number 2), and "copy" CXT1 by pushing the new layer onto it, making CXT2. Now if it REMOVEs (ON A B) and ADDs (ON A C) with respect to CXT2, all other data are undisturbed, and CXT1's contents are unchanged. The two changes "remove (ON A B)" and "add (ON A C)" are associated with layer 2, and thus with CXT2 and all further cxts derived from it. (Another use is to model "hypothetical" worlds in which some assumption is assumed temporarily. For more detail on implementation, see <McDermott and Sussman, 1973>.) If the layers of two cxts C1 and C2 are such that all the layers of C1 are in C2, C1 is called a *super-cxt* of C2, which is a *sub-cxt* of C1. In my example, CXT1 is a super-cxt of CXT2.

Layers are well behaved enough so that new cxts may be formed by taking the union of sets of them. (This fact is often overlooked in discussions of cxts, so that a "cxt tree" created by adding new layers is taken for granted.) I will deal with this feature at length in Sect. IV.

II.A.3 Derived Data

The structure I have described is internally complete, but I must raise one more superstructure upon it to fit it into the usual tradition. Planner-type data bases almost always have built-in mechanisms for calling simple "deductive" procedures. One kind, the IF-NEEDED method or consequent theorem, is supposed to work with FETCH to generate "virtual items" which might not be there until the FETCH happens. I will not say much about this type, since the issues in implementing it are mostly centered on how to control a generator co-routine.

The other kinds are implemented as software "interrupts" of additions and removals of items matching a pattern. These are Conniver's IF-ADDED and IF-REMOVED method types. For example, in pidgin LISP for the TDBM, we might write

```
(if-added (is rose ?r)
    (add !"(color @r red))   )
```

(The !"-notation indicates "quasi-quotation"; the value of !"*S-expression* is *S-expression* with subparts marked by "@" replaced by their values.) A procedure like this is present in a cxt just like the items it interacts with. (ADD *item*) calls (FETCH !"(IF-ADDED @*item* ?)), and evaluates the third slot in the returned items.

These interrupt methods tend to be used in two ways: to implement useful forward deductions; and as genuine interrupts requiring non-trivial attention by the calling program. The former is illustrated by my "roses are red" example; the latter, by a method in a problem solver that notices when a "protected" goal is being clobbered. (Cf. <Sussman, 1975>) The latter, the "true interrupts," pose no problem from a data-base management point of view. I will ignore them.

The others, the "data derivers," fall into two classes: if-addeds that add consequences of their trigger items; and if-removeds that clean them up, like:

```
(if-removed (is rose ?r)
   (remove !"(color ⊛r red))   )
```

I will idealize this situation by assuming one kind of data deriver, an item of the form

(ANTEC-ITEM *input output*).

("Antec" is descended from Hewitt's <1971> "antecedent theorem.") For example, we can have (ANTEC-ITEM (IS ROSE ?R) (COLOR ?R RED)). When ADD adds an item matching *input*, the matching substitution is applied to *output*, and adds it, too. Removal works asymmetrically. In this scheme, ADD makes a note that the support for, say, (COLOR ROSE#71 RED) in this cxt is the presence of (IS ROSE ROSE#71) and (ANTEC-ITEM (IS ROSE ?R) (COLOR ?R RED)). This note is attached to all three items. When REMOVE flushes an item, it flushes as well all items supported by it that have no other support. This obviates clean-up methods.

The details of this somewhat arbitrary scheme are unimportant, but the notion of distinguishing true interrupts from data derivers has been shown to be useful in practice.

II.B Implementation

In the TDBM, there are three entities that stand between the caller of FETCH and the data: the index, the cxt filter, and the matcher. All previously-mentioned data are in the index (which behaves much like the LISP atomic symbol array), but a given call to FETCH should return only a handful. (See Fig. II.2.)

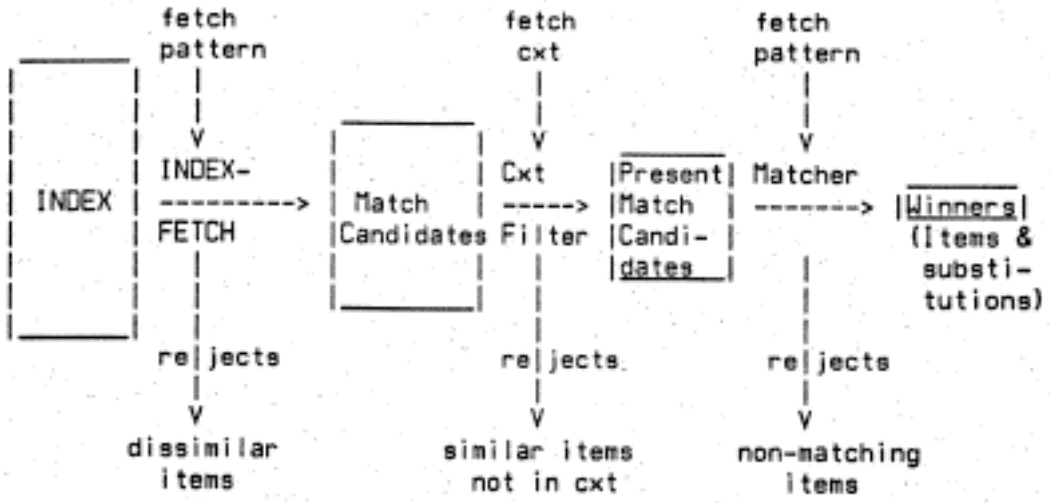


Figure II.2

The last step in Fig. II.2, which rejects (P a b) when the fetch pattern is (P ?x ?x), is not of interest to us. We focus on the others.

Notice first that cxts are implemented backwards. The system does not look in a cxt for a datum; it looks on a datum to see if it is in the current cxt. In the index, an item is stored as an *item datum*, of the form (*item -cmarkers-*). The cmarkers, one per relevant layer, say which cxts the item has been added to or removed from. <McDermott and Sussman, 1973> These markers are compared with the current cxt (the two lists being kept sorted by layer number) to see if the datum is actually there. (Datum "properties," such as what data support a datum, may also be stored in the cmarkers.)

The marker system treats datum presence asymmetrically from absence. A marker for a cxt layer is attached to a datum only if the datum was added with respect to the layer. If it is later removed from the layer, the marker is just thrown away. However, if it is removed in a cxt derived by pushing new layers onto the original one, the system must *cancel* the marker. For example, if (ON A B) is marked as being in layer 1, then REMOVing it in cxt (3 2 1) marks (ON A B) "present in 1 as cancelled in 3." Any cxt which includes layer 1 will contain (ON A B) unless it includes

layer 3 also.

This "backwards" cxt scheme is efficient enough for small data bases, but for large ones it could be embarrassing. This is the topic of Section IV.

Even for small data bases, it is important that the index-fetcher reject all but a small list of candidates. (Matching tends to be an expensive way to reject an item.) There have been many ways suggested to do this. The following is a composite.

Every item is indexed by its *features*. A feature is a pair <*atom, position*>. For example, (P (f ?)) has features <P, CAR>, <f, CAADR>, <?, CADADR>. (The positions are, of course, really represented as bit strings, but we will use LISP CAR-CDR compositions, with "CR" standing for the empty string that represents the top-level position.) For each feature, there is a *bucket* of items with that feature. (This may be implemented using a hash table.) For example, the <P, CAR> bucket will include items (P a), (P (f b)), (P m n o p), if these are present in any cxt in the data base. The <?, CADR> bucket will contain items like (P ?x), ((Q a) ?y), (P ?x (f c)), etc.

The function INDEX-FETCH takes a pattern and a position, and returns a list of candidates that contain, in that position, a structure which could match pattern. Thus (INDEX-FETCH '*pattern* 'CR) returns all items anywhere which could match *pattern*. If the pattern is "?," all items must be included; this is indicated by returning a symbol for the "universal bucket." Otherwise, the returned list must always include the members of the <?, *position*> bucket. If the pattern is an atom, the union of the <?, *position*> and <*atom, position*> buckets is returned. (Nothing else could match.)

When the pattern is non-atomic, INDEX-FETCH must examine the left and

right "extensions" of the current position; that is, the CAR and CDR relative to the current position. For example, (INDEX-FETCH '(P ?) 'CR) will call (INDEX-FETCH 'P 'CAR) and (INDEX-FETCH '(?) 'CDR). The pattern of calls may be represented by a tree isomorphic to the original fetch pattern:

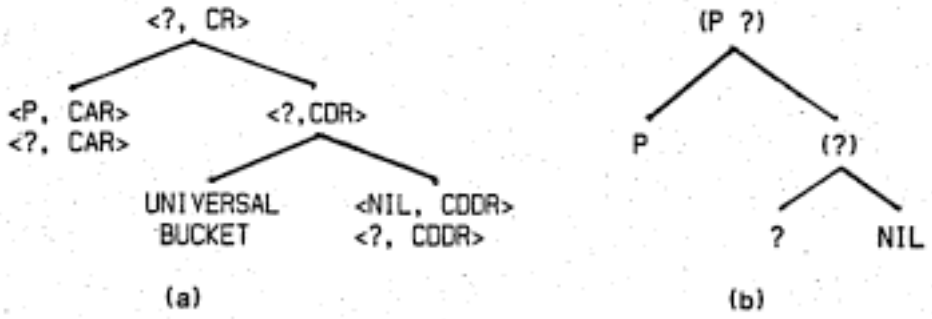


Figure II.3

Remember that at each "care" node the "don't care" bucket must be unioned in. But first, what does the system do with the results of the two subcalls to INDEX-FETCH at each level? Conceptually, the desired result is their intersection, the set of all items which share the feature combinations on each subtree. (This is actually done in the current Conniver, under some circumstances.) However, it is too painful in a large data base with many items. The output of the intersection may be quite small, and it will have to be redone every time there is a FETCH.

A cheaper procedure, most of the time, is just to take the shorter of the two buckets at each stage. This leaves in some losers, but it may not be worth it to get rid of them. For example, given the fetch pattern (P A), there may be many items starting with P, but only a few with an A in the CADR position. For now, let us assume it is cheaper to filter out the losers with the cxt and matching mechanisms than do intersections with arbitrarily long buckets.

The mechanism so far thus selects a branch of a tree like Fig. II.3(a),

and returns the union of the buckets on the branch. (There is no reason actually to do the union until it returns to the top level.) Notice that, since the tree is searched depth-first, the search down a branch can be cut off as soon as the sum of the sizes of the buckets encountered exceeds that of the best previous branch.

In a very large data base, taking the shorter of the two competing bucket unions may not be enough. If there are two competing features in a fetch pattern, and each has a large bucket, but the intersection of the two is small, it is costly to return either one or their (expensive to compute) intesection. The usual solution is to take one of the offending buckets and break it down by features again, to produce *sub-buckets* corresponding to pairs of features. This amounts to computing all non-empty intersections in advance. For example, if the indexed items are ((P a), (P b), (P c), (P d), (P e), (Q a), (R a), (S a), (T a)), then the <P, CAR> bucket is ((P a), (P b), (P c), (P d), (P e)), the <a, CADR> bucket is ((P a), (Q a), (R a), (S a), (T a)), and the <NIL, CDDR> bucket includes all the data. The best (INDEX-FETCH '(P A) 'CR) can do is find the two 5-item buckets. Assuming 5 exceeds the threshold of bad taste for bucket size, the indexer decides to break down the <P, CAR> bucket.

When it is done, the index contains the following:

```
Buckets
   <P, CAR>
      Sub-buckets
         <P, CAR> {(P a), (P b), (P c), (P d), (P e)}
         <a, CADR> {(P a)}
         <b, CADR> {(P b)}
         <c, CADR> {(P c)}
         <d, CADR> {(P d)}
         <e, CADR> {(P e)}
         <NIL, CDDR> {(P a), (P b), (P c), (P d), (P e)}
   <Q, CAR> {(Q a)}
   <R, CAR> {(R a)}
   <S, CAR> {(S a)}
   <T, CAR> {(T a)}
   <a, CADR> {(P a), (Q a), (R a), (S a), (T a)}
   <b, CADR> {(P b)}
   <c, CADR> {(P c)}
   <d, CADR> {(P d)}
   <e, CADR> {(P e)}
   <NIL, CDDR> {(P a), (P b), (P c), (P d), (P e), (Q a),
                (R a), (S a), (T a)}
```

Figure II.4

This procedure is called *rehashing* the <P, CAR> bucket. From now on, any reference to the <P, CAR> bucket is handled by finding the relevant sub-buckets given the rest of the pattern, and using those instead (and ignoring the corresponding buckets in the main index, which are always bigger). Consequently, bucket and sub-bucket keys must be generalized to sequences of features, like <<P, CAR>, <a, CADR>>. In this way, the index tends to organize itself into efficient sub-indexes. If necessary, the process may continue, generating sub-sub-indexes, etc. (with the sub-index coming more and more to resemble QA4's discrimination net index system <Rulifson *et. al.*, 1972>).

One drawback to my scheme is that some buckets created this way will never be used. For example, in the sample index shown, the only reason to have a bucket for the feature <a, CADR> in the main index as well as in various sub-indexes is the possibility of a future (FETCH '(? a)). If the user is sure is sure all future fetches will mention P, Q, R, etc. explicitly, the bucket is useless to him.

The solution is to make the searching, indexing, and unindexing routines *data-driven* in the sense that their default actions may be overridden by routines dependent on the data being handled. The simplest scheme is to associate special indexing routines with the CAR of an item. For example, the routines for atoms like P might specify skipping indexing and fetching on the CADR position at all, in the case where all FETCHes have a "?" in the CADR position.

Data-driven indexing is helpful in other ways. A form like QA4's (SET element1 ... element2) may be indexed in such a way that all the elements are associated with the same (CADR) position. Then a special set matcher can be used to process FETCHed sets. If the set (SET A B) is fished for with the pattern (SET B . ?), it will be overlooked unless something like this is done. A more prosaic example is avoiding looking for <?, *pos*> buckets when they are known not to exist. Ordinary Planner and Conniver programs usually have great quantities of dull data like (VERTEX 0.5 -4.7), where it is wasteful to have to look for the empty <?, CADR> and <?, CADDR> buckets.

III *THE SYMBOL-MAPPING PROBLEM*

III.A Introduction

Difficulties of scale with the typical data-base manager appear in connection with Fahlman's <1975> "symbol-mapping" problem.

> Suppose I tell you that a certain animal--let's call him Clyde--is an elephant. You accept this simple assertion and file it away with no apparent display of mental effort. And yet, as a result of this simple transaction, you suddenly appear to know a great deal about Clyde. If I say that Clyde climbs trees or plays the piano or lives in a teacup, you will immediately begin to doubt my credibility. Somehow,

"elephant" is serving as more than a mere label here; it is, in some sense, a whole package of properties and relationships, and that package can be delivered by means of a single IS-A statement.

In principle, such behavior can be achieved through the use of some form of deduction. Each fact is a separate entity, and new facts are produced by knocking together two old ones. Thus, if we have "All elephants have wrinkles" and "Clyde is an elephant", we have the right to deduce that Clyde has wrinkles. In one form or another, this has been the standard AI approach.

But having the right to deduce some fact is not the same as having the job done. Much ingenuity has been devoted to the search for fast deductive mechanisms, but the problem remains intractably combinatorial.

There are two problems described here: bringing in large amounts of typical-elephant knowledge, with, loosely speaking, typical elephant "bound to" Clyde; and detecting implausibilties ("Clyde climbs trees") when they come up.

The second is a serious problem, whose solution depends on the particular domain it is drawn from. We should not expect a raw data-base system to perform unassisted tasks like choosing between two interpretations of "he" in, "The monkey screamed at the elephant as he climbed the tree."

But I feel that the first problem should be made simple by such a system. Let us examine the three options for solving it that a Planner-type system offers us:

(1) We could treat

```
(is elephant ?x)
   ⊃ (color ?x gray) ∧ (size ?x big) ∧ (diet ?x plants)
      ∧ (is mammal ?x) ∧ ...
```

as a large ANTEC-ITEM (Sect. II.A.3). When (IS ELEPHANT CLYDE) is added to a cxt, the newly-consed facts (COLOR CLYDE GRAY), (SIZE CLYDE BIG), ... are added as well. Unfortunately, this is likely to waste a lot of time and space, since the number of facts known about elephants is large, but only a few of them are going to be used on any occasion. Further, facts like (IS MAMMAL CLYDE) are likely to cause many more facts to be deduced.

(2) We could wait until a fetch of (COLOR CLYDE ?C) was attempted, and propose (IS ELEPHANT CLYDE) as a subgoal (via an IF-NEEDED method or consequent theorem). Obviously, however, there are many equally good-looking subgoals, like (IS ROSE CLYDE) or (OCCUPATION CLYDE CHIMNEY-SWEEP), for the system to waste its time going through.

Another approach like this is to search through all things that Clyde IS when his color is wanted, looking for an assertion like (COLOR ELEPHANT GRAY). For this search to mean anything, the IS relation will have to carry the burden of representing most information in the system, indeed *any* piece of information from which inferences could have been made (in this case, about COLOR). For example, one would be forced to say (IS CHIMNEY-SWEEP CLYDE) instead of (OCCUPATION CLYDE CHIMNEY-SWEEP); or the item (COLOR CHIMNEY-SWEEP BLACK) would not be found. Although the approach can be useful when the use of IS is carefully tailored for a specific application, if used profligately it turns "(IS" into a mere syntactic symbol, a left bracket like "(," which indicates only that some inference might have been drawn from what it delimits. The method then becomes "search through (almost) everything known about Clyde to see if a COLOR turns up," which could not be efficient on a normal computer. (Of course, it is an open question whether all useful AI inference can be made to fit the "(IS...)" syntax. Cf. <Woods, 1975>) Another difference between "(IS" and "(" would be that IS presumably would signal that its second argument is what the assertion is "about." For example, if by accident we had (IS (OCCUPATION CLYDE) CHIMNEY-SWEEP), this fact would not be "about" Clyde, and would not be considered.

(3) If the right-hand side of the implication could be represented as a cxt layer or set of layers, we could just merge these layers into the current cxt. The problem with this is that "CLYDE" does not appear on the

right-hand side. We really want a "closure" of a cxt, analogous to closures in languages like POP-2 <Burstall *et. al.*, 1971> and LISP 1.5 <Levin *et. al.*, 1965>, in which a free variable like ?X is bound to CLYDE with minimal cost.

Approach (3) is the one we will explore, but it is worth examining in detail why a Conniver-style cxt will not work. The user can't replace ?X with something like THE-ELEPHANT, for two reasons: there might be more than one elephant; and he has to begin referring to Clyde as THE-ELEPHANT. This second reason is a problem because some of the facts involved might refer to THE-MAMMAL; and because two processes in the user's problem solving program might have trouble communicating with each other, each having focussed on a different aspect of this creature and chosen a different name.

These facts hurt, not only because they blunt our attack on the symbol-mapping problem, but because they appear to pull the rug out from under efforts to implement a Minskian frame system <Minsky, 1974> in a Planner-like way.

However, there is a way to debug approach (3), which makes a Planner-type data base a viable candidate for a substrate for frames.

III.B Potential Items

The idea is to index an item like (COLOR ?X GRAY) with ?X as a variable, but mark it as a mere *potential item*. When FETCH finds such a potential item, it is to "smash" it with values of ?X corresponding to creatures whose elephant-hood have been asserted. In addition, the item is to be present only in cxts including the "elephant layer," which is the

representation of the right-hand-side of our large implication. This layer is included only as long as there are elephants around. Such a layer (or, more generally, entire cxt), containing potential items referring to elephant, is called a *packet*, after Fahlman <1973> and Marcus <1974>.

In other words, I am trying to make the system "cheat" by substituting values for variables in the right-hand side of a large implication before the substitution to do it with is known. In lieu of that substitution, a set of temporary bindings to dummy quantities like (#ABSTRACT ELEPHANT) is used. Since the indexer is data-driven, it can be instructed to index (#ABSTRACT ...) as a variable. I will indicate this binding to a quasi-constant with the prefix "?##."

As an example, let us examine the elephant packet. It contains the potential items (COLOR ?##ELEPHANT GRAY)*, (SIZE ?##ELEPHANT LARGE)*, etc. (I will mark potential items with an "*.") The system gives the packet an atomic name, like ELEPHANT-PKT, and treats it as a predicate with the free variables as arguments, (ELEPHANT-PKT ?E). This allows it to process items like (ANTEC-ITEM (IS ELEPHANT ?X) (ELEPHANT-PKT ?X)). When (IS ELEPHANT CLYDE) is added to a cxt, it triggers the adding of (ELEPHANT-PKT CLYDE). The system indexes this item as usual, but notices that ELEPHANT-PKT is not an ordinary predicate, but a packet, so it *includes* the ELEPHANT-PKT cxt layers in the current cxt, and notes the substitution of CLYDE for ?X. The item (ELEPHANT-PKT CLYDE) is a *packet-closure*.

Now when a FETCH of any of

```
(color clyde gray)
(color fred gray)
(color ?z gray)
(color ?z ?c)
(color clyde ?c), etc.
```

is attempted, the potential item (COLOR ?##ELEPHANT GRAY)* will be found and *actualized* to (COLOR CLYDE GRAY). This new item will appear in the

local cxt, of course, not in the elephant packet.

If (IS ELEPHANT RALPH) is added, the system adds (ELEPHANT-PKT RALPH). On the next fetch of (COLOR ...), both (COLOR CLYDE GRAY) and (COLOR RALPH GRAY) will be generated. (The way the system finds all bindings, of course, is to FETCH (ELEPHANT-PKT ?X) and use the retrieved substitutions.)

Of course, this is wasteful as it stands, since the system doesn't get beyond the potential item. In fact, it must *replace* it with its actualizations. (It must be cautious enough to remember that it did that, so it can reactivate it in case another elephant appears.)

Finally, "data-dependency notes" like those of Sect. II.A.3 must be attached to actualizations like (COLOR CLYDE GRAY), so that if (IS ELEPHANT CLYDE) is removed from the cxt, the actualizations can be removed, too. And when the last (ELEPHANT-PKT ...) packet-closure is gone, the layers of the ELEPHANT-PKT must be *excluded* from the current cxt.

III.C Further Embellishments and Comments

When a packet is being built, it is desirable to be able to make deductions from its potential items as they are added. For example, it is important to be able to include a closure of the MAMMAL packet in ELEPHANT-PKT when it is built. So at this time the system must treat ?##ELEPHANT as an actual constant that can take part in matches, etc. When (IS MAMMAL ?##ELEPHANT) is added, it is not treated as a potential item, but causes (MAMMAL-PKT ?##ELEPHANT) to be included in the ELEPHANT-PKT cxt. This is not just applicable to transitive IS deduction; it works for ordinary antecedent reasoning, and for inclusion of other kinds of packets (such as the TUSK packet, two closures of which might be included in the elephant packet).

Thus there will be dependencies among the potential items. For example, (CAN ?##ELEPHANT (HIDE-IN FILING-CABINETS))* might be marked as a consequence of (COLOR ?##ELEPHANT GRAY)*. This will not matter much unless some user of the packet tries to remove the COLOR item for a particular elephant. This is intuitively a complex operation, since the reasons for and consequences of an alteration to a structured data base may be convoluted. However, some simple bookkeeping can be done by the data base manager. When (COLOR CLYDE GRAY) is removed, (CAN CLYDE (HIDE-IN FILING-CABINETS)) should go, too. The situation may be represented as follows:

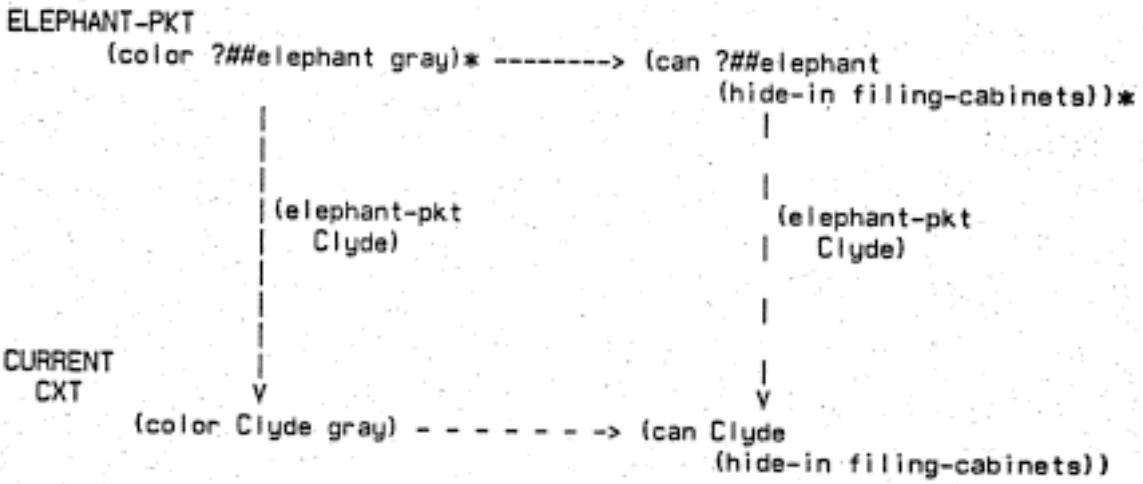
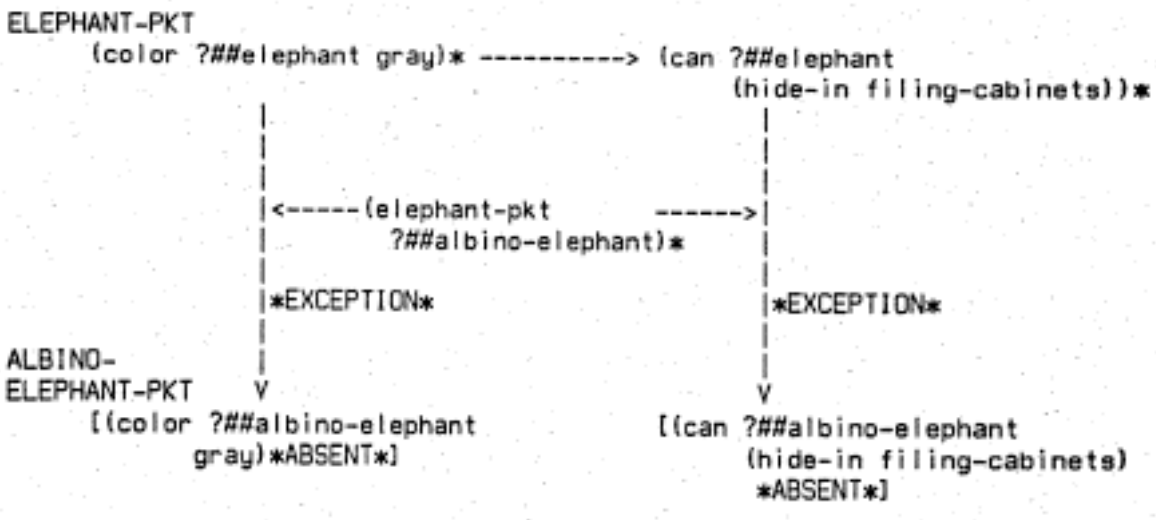


Figure III.1

Even though the broken lines are not yet present, they must be put in. Then (COLOR CLYDE GRAY) and its supportees can be removed as described in Sect. II.A.3. The actualization links must be marked so that future fetches will know not to add the actualizers again. This process is called ***detaching*** the item (COLOR CLYDE GRAY) from its packet.

This mechanism (which in its full generality is more complex) can be used for building packets defined in terms of other packets plus certain exceptions. For example, ALBINO-ELEPHANT-PKT may be defined as the ELEPHANT packet cxt with one layer added, in which (COLOR ?##ALBINO-ELEPHANT GRAY)* is removed and (COLOR ?##ALBINO-ELEPHANT WHITE)* is added.

If (IS ALBINO-ELEPHANT CLYDE) is added to a cxt, a FETCH of (COLOR CLYDE GRAY) finds the following situation:



CURRENT CXT
(albino-elephant-pkt Clyde)

Figure III.2

FETCH finds (COLOR ?##ELEPHANT GRAY)*, but in attempting to find all the elephants, it notices the *EXCEPTION* links through the ALBINO-ELEPHANT packet, and leaves Clyde out.

The "potential item" implementation of packet-closures has the feature that it is independent of a particular relation like set-inclusion ("IS"). For example, a large implication like

```
(loves ?x ?y)
  ⊃ (jealous-of ?x ?y) ∧ (admires ?x ?y)
     ∧ (habitually (together ?x ?y))
     ∧ (when (do ?x (for-some movie
                      (lambda (m) (attend ?x ?m))))
             (presumably (accompanies ?y ?x))
     ∧ ...
```

can be handled easily by creating a "love-affair" packet, containing items like (JEALOUS-OF ?##LOVER ?##LOVEE)*. In fact, implication itself is only incidentally connected with the use of packets. Any large conjunction of

items sharing free variables may be represented as a packet.

If the reader is feeling optimistic about packets, here are some cautions. First, although the "semantic baggage" problem doesn't look so bad any more, the other problems Fahlman is concerned with are not attacked directly by my solution to it. In particular, typical consequences of something's being an elephant are not automatically usable in recognition of elephants. Instead, elephant recognition knowledge will be contained in, say, the jungle packet, with the typical frame organization. (This is as good a place as any to point out that my absurd examples in this chapter are not representative of useful elephant knowledge, but only illustrate formal problems and solutions.)

The biggest drawback to my system is that it relies on a packet-closure, such as the one expressing Clyde's elephanthood, being a temporary structure, frequently added, removed, or abandoned. Otherwise, if all known information about all known elephants is always around, it will take too long to smash a potential item like (COLOR ?##ELEPHANT GRAY)*, and most of the products of the effort will be useless. Note that there is really no difference between the potential item idea and a particular kind of consequent (IF-NEEDED) reasoning. This reasoning is, "to find Clyde's color, find what types of colored entities are around, deduce what they are, and see if Clyde is one of them." This deduction is done efficiently by following the actualization pointers down through layers of packet (Cf. Fig. III.2), and we get the exception mechanism cheap by having the system do it this way, but the combinatorial limitations are the same. (I believe that even in bad cases this method is superior to the other approaches mentioned at the beginning of this section, which involve proceeding from Clyde to COLOR. My approach always sticks to pathways which are guaranteed to lead to present COLOR items (or exceptions). Furthermore, once they

have been found, the potential item that generated them is gone.)

In my own research on electronic design <McDermott, 1974b>, this assumption that only a few closures of a packet will be around at one time is justified. A closure of the "af-amplifier" packet will be made and included in the cxt for a radio receiver when an af amplifier is needed, but it will not be visible in cxts for later tasks. I think this sort of organization is typical of problem solving systems (and is crucial to the frame theory that most people accept), but there may be situations where it is inappropriate. (See Sect. V.)

III.0 Implementation

Although it is too early to tell how successful my packet scheme can be, I have implemented a preliminary version of it, the PANACEA data base manager. It is based on a modified Conniver data base with a data-driven indexer but no sub-indexing, which it uses as an associative memory and item uniquizer. It buffers the user from the actual data by smashing all potential item data as described above.

The PANACEA system itself is quite small, although my programming style has made the TDBM and others system programs it relies on too big and clumsy. Most of the complexity of PANACEA is in the routines that maintain data dependencies and exception links in a consistent state. The actual potential-item smasher occupies two or three pages of LISP code. It is hard to tell how fast it runs at this time, since it has not been fully compiled, and since there are known inefficiencies in the support routines for it. The best measure is the number of matches (unifications) and variant-tests done in the course of a fetch. This number depends on the complexity of the packets involved, but is much smaller than it would be in

an ordinary deductive scheme.

The item representation I am using is that of Boyer and Moore <1972>, in which an item is stored as a pattern and separate substitution. By analogy with function closures, the stored substitution is called the "environment" of the item. The original purpose was to save storage and enhance readability of formulas by representing a deduced formula as a pattern plus bindings acquired during its deductive history. However, it has the additional advantage for PANACEA that it enables potential items to be actualized by just switching their environments before adding them. Unfortunately, this is paid for in other ways, since special functions have to be written to manipulate items stored this way.

IV *INDEXING BY CONTEXT*

IV.A Structural Indexing

Except for cancellation, a cxt behaves like the union of the sets of data represented by its cxt layers. Therefore, it is a particularly ugly feature of the classical implementation that it finds all the data likely to match a pattern and throws away the ones not in the current cxt, instead of taking the union only of the relevant sets. Excluding a layer from the current cxt only means hiding it from yourself, not from the data base machinery. It has always seemed that these facts make cxts fit only for toy problems; that domain-dependent data structures or multiple CPU's would be needed for large data bases.

The problem is especially pressing in view of our willingness in Sect. III to have many items like (COLOR ?##ELEPHANT GRAY)* normally hidden from

view. Clearly, we cannot afford to have every FETCH of (COLOR x ?C) filter a bucket containing a formula for every type of object with a usual color.

The first alternative that comes to mind is to rehash any index buckets that have gotten too big by the layers their data are present in. On fetching, the system takes each layer in the fetch cxt and hashes its number to retrieve the relevant sub-buckets, if any. Then it unions the sub-buckets.

The problem with this is that most layers will not mention any data matching the current fetch pattern. Of 100 layers, most will mention no COLOR assertions, for example. This scheme is still probably an improvement over the original. (Its efficiency depends more on the length of the cxt than the number of matching data anywhere in the data base, which seems better.)

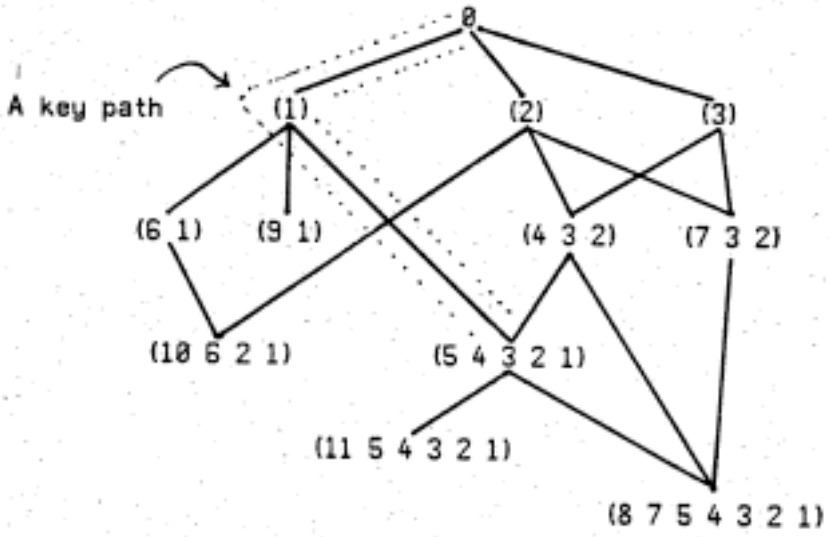
IV.A.1 Merge History Graphs

So far, I have been treating cxt layers as pearls of data which can be strung as we will. This notion is what leaves us with so little structure to use here, where we need it. In actual practice, it is possible to impose some useful discipline on the building of cxts without losing any real flexibility.

Observe first that ninety-nine percent of all Conniver cxt manipulation is in terms of PUSH-CONTEXT and POP-CONTEXT. PUSH-CONTEXT adds a new layer on the front of a cxt, and POP just gets back what you started with. (It is equivalent to CDR.) Because we want to manipulate packets, let us generalize PUSH to *merge*: take *n* cxts, union them, and push a new layer onto the result. Merging may be used to include a packet-closure in a cxt. (We are also going to need an implementation of packet-closure exclusion,

which I will describe later.)

If we restrict ourselves to merging for the time being, every cxt has *n parents* of which it is the *child*. The process starts with the empty cxt, with no layers, called 0. All one-layer cxts are children of 0 (the result of a merge with *n*=0). Every layer is the first layer of exactly one cxt. Over time, the structure of cxts might evolve like this:



Merge History Graph
Figure IV.1

Notice that there may be more than one merge of the same two cxts. This fact is basically irrelevant, and I will ignore it, allowing myself to say "the" merge of two cxts. The new layer in each cxt is called its *primary layer*, of which it is the *primary cxt*.

The merge history graph gives us some structure to work with. Let us associate with every bucket in the index its *key paths*, each of which is a (partial) descending branch through a cxt graph like Fig. IV.1. (A "key path" is a "path used as a key," not a vitally important path.) A key path for a bucket must terminate on a cxt which is a super-cxt of the cxt of each datum in the bucket. For example, in Fig. IV.1, <0, (1), (5 4 3 2 1)> would be a key path for a bucket of data associated with cxts (5 4 3 2 1), (11 5 4 3 2 1), and (8 7 5 4 3 2 1). Since a layer unambiguously

identifies a its primary cxt, I will abbreviate this key path to <0, 1, 5>. Hitherto, all buckets have implicitly had key path <0>, since the cxt of every item is a descendant of 0. (An item may have more than one cxt, but that's not important.)

Hitherto, however, such a bucket has often been able to grow without bound. Now, when its size exceeds some threshold, let the system rehash it into sub-buckets, each of whose key paths is 1 longer than the old one it came from. For example, a <0> bucket may be broken down into <0, 1>, <0, 2>, and <0, 3> sub-buckets. (If any of these is empty, it should be omitted.)

Now say (P a) is in layer 1 and (P b) in 2. I will abbreviate this as $\{(P\ a)_1,\ (P\ b)_2\}$. The system can put $\{(P\ a)_1\}$ in bucket <0, 1>, put $(P\ b)_2$ in bucket <0, 2>, and omit the <0, 3> bucket. Now when a (FETCH '(P ?)) is done in cxt (6 1), the data-base manager can ignore some of the (P...) items altogether. It just "inverts" (6 1) (this may be done in advance) to give <0, 1, 6>, and takes the <0, 1> sub-bucket when it discovers the <0> bucket to have been rehashed.

Clearly, this process is extendable. Say that $\{(P\ c)_6,\ (P\ d)_6,\ (P\ e)_1,\ (P\ f)_9,\ (P\ g)_9\}$ are added to this bucket structure. All have inverted cxts starting with <0, 1,...>, so they are all placed in the <0, 1> sub-bucket. Now a fetch of (P ?) in cxt (1) will get $\{(P\ a)_1,\ (P\ c)_6,\ (P\ d)_6,\ (P\ e)_1,\ (P\ f)_9,\ (P\ g)_9\}$, all but two of which are then filtered out. This low efficiency indicates the list has gotten too big, so it is rehashed into three sub-buckets:

Key Path	*Contents*
<0, 1>	$\{(P\ a)_1,\ (P\ e)_1\}$
<0, 1, 6>	$\{(P\ c)_6,\ (P\ d)_6\}$
<0, 1, 9>	$\{(P\ f)_9,\ (P\ g)_9\}$

Now if the fetch is from cxt (1), the system gets $\{(P\ a)_1,\ (P\ e)_1\}$ without

any filtering. If it is from (6 1), it takes the union of the <0, 1> and <0, 1, 6> sub-buckets to get $\{(P\ a)_1, (P\ e)_1, (P\ c)_6, (P\ d)_6\}$.

So long as only pushes are used to make new cxts, the fetch algorithm (once you have a bucket) is

(a) invert the fetch cxt

(b) use successive prefix strings of the inversion as key paths to find your way down the sub-bucket tree

(c) take the union of the sub-buckets found, filtering out items not actually in the fetch cxt

Part (b) will proceed only so long as the as yet unseen sub-buckets generated by past ADDs are too big for part (c) to be efficient.

If merges are allowed, the algorithm is much the same, except inverting a cxt will not give a single key path. For example, in Fig. IV.1, cxt (5 4 3 2 1) looks like

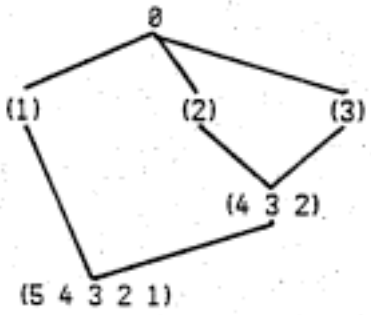


Figure IV.2

Therefore, we must take into account the likelihood of having more than one way to extend the key path to find sub-buckets. In this case, if the bucket is too big, the fetcher must look in the <0, 1>, <0, 2>, and <0, 3> sub-buckets, and take the union of the results. If these sub-buckets are themselves broken down, it must avoid finding and filtering the bucket for (4 3 2) twice, under key paths <0, 2, 4> and <0, 3, 4>. Similarly, the sub-bucket for (5 4 3 2 1) has three key paths. All the system has to do is compute the "inversion" of (5 4 3 2 1) (again, in advance, if desired) as

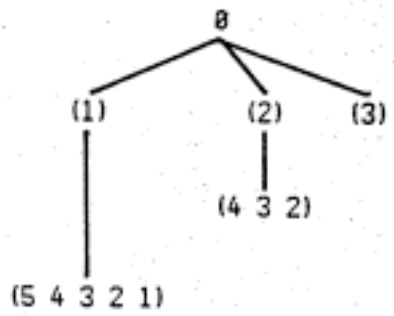


Figure IV.3

or one of the other arc-deletions that turns Fig. IV.2 into a tree. Of course, the routine that first creates an ambiguously-named sub-bucket must make sure that it is pointed to from all relevant places. Then it will not matter which is used to find it again.

Before extending this concept, let me explain what its costs and pitfalls are. First, like any table look-up scheme, it trades off the cost of adding an entry against the cost of retrieving it. We have to spend more time and space adding an item if we must index it by cxt. However, memory is getting cheaper, and this scheme will work with cheap mass storage (Sect. IV.C). Furthermore, if necessary we can arrange to expend the resources primarily on cxts (like packets) which are to be relatively permanent, fetched from more often than added to.

Second, it seems as though it could be a big nuisance to have to push a new layer onto a cxt every time you do a merge. Why not just use the old top layer, especially if you're doing only retrieval and no new storing? The answer is, you can; I just wanted the one-to-one correspondence between layers and cxts for purposes of clear exposition. Since it's the complete cxt inversion that guides the fetcher, it doesn't matter whether two cxts share the same top layer. (Of course, two cxts sharing a primary layer can have side effects on each other.)

Third, there are "pathological" cases where this scheme will not work any better than the primitive union case (and practically reduces to it).

If a cxt is shallow and wide, like this:

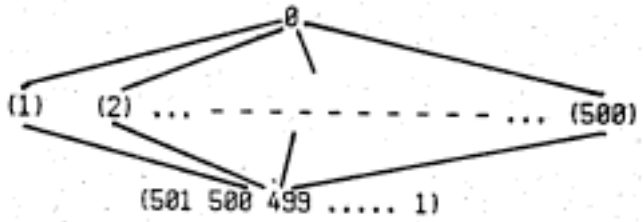


Figure IV.4

and its items are sparsely distributed among its layers, then we are back to unstructured unions. However, this is not a very common case, because packet-like cxts tend to have been built by merges. Thus the packet for "elephant" will include the mammal packet; as will the packet for "wombat." If both elephant and wombat packet-closures are added to the current cxt, they will not be children of 0:

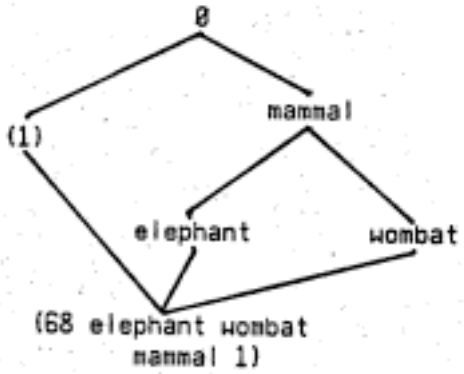


Figure IV.5

(Here and later, I use, e.g., "elephant" to mean the primary layer of the elephant packet cxt.) In general, I expect there to be a few "utility packets" for a given domain ("body part," for example, in one domain, and "electronic component" in another) that will serve as strong keys for pursuing sub-buckets. Even though most of them will draw a blank for a given pattern, they will then hide their sub-cxts (e.g., the cxts for animals or circuits whose parts they represent) from being used as key path elements at all.

It may be possible to avoid this problem by artificially ensuring that

no cxt has more than some small number N of children. Whenever a cxt acquires N+1 children, the system could insert dummy layers between it and its children to group them into fewer than N groups, each with fewer than N members. If the data are sparsely enough distributed, some of the intermediate cxts will draw blanks and save us some work. However, I predict this is more trouble than it is worth.

A good remedy for many problems of this kind is to store a complete list of the sub-buckets of a bucket and *throw out* the sub-buckets with bad key paths instead of *finding* those with good ones. If the data are distributed so that they all fall into 2 or 3 sub-buckets out of a possible 500 key path extensions, it is worth looking at it this way.

To see how this remedy might apply, consider the case of the creation of a huge set of data matching a pattern, in just one cxt that is the result of many pushes:

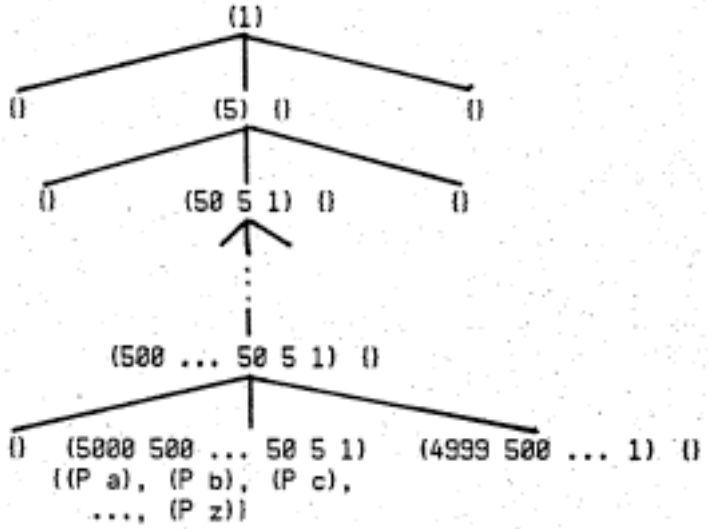


Figure IV.6

The problem here is that, when fetching from (4999 ... 1), the indexer does not know it is on a wild-goose chase until the last step. There are always lots of data still around down there, or there would be just one bucket to be filtered the old way.

This problem will be solved if we make a note in the bucket for cxt (1) that there is only one non-empty sub-bucket (or two or three) anywhere below it in the graph. If so, it is better just to keep that bucket around and test its key paths against the fetch cxt. In fact, there is no reason to build the actual graph until the number of non-empty buckets exceeds some threshold, and then only the first couple of forks need to be built immediately. Consequently, we can modify the previous algorithm to include the possibility that a bucket has, say, four or fewer sub-buckets, which are just filtered a bucket at a time, not by individual item. The graph structure exists only to guide the indexer to this situation or the old case of one small bucket as yet unrehashed. (There is no difficulty in interleaving layers of graph and amorphous bucket list.) For example:

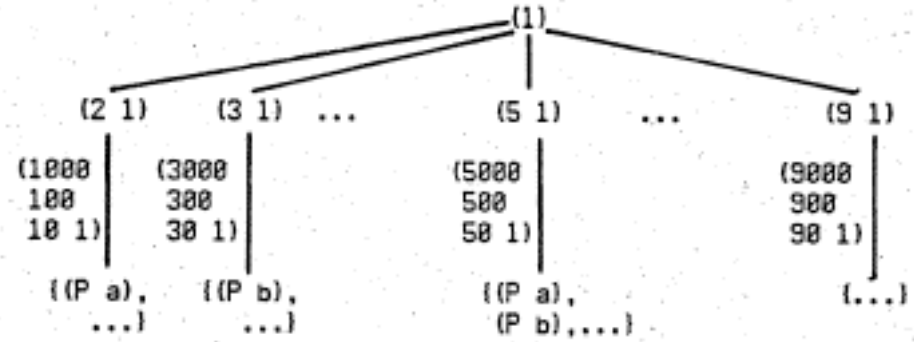


Figure IV.7

Here, from (1) there are eight big buckets, too many to examine one by one (let us assume). However, if the fetch cxt is (4999 ... 1), after one step down the tree there is only one big bucket left to worry about, the one for (5000 ... 1), which can be compared with (4999 ... 1) and ignored.

IV.A.2 Excision of Cxt Layers

To implement packet-closure exclusion, we have to have a way to remove cxts from a merge. The way I have developed, although somewhat unintuitive, serves this purpose well enough. (It is so unintuitive that you may want to skip this section, if you wish to take my word for it that cxt layer excision is compatible with cxt indexing.)

A first guess at how such alterations ought to work is to generalize the usual POP-CONTEXT operation, and allow any layer to be removed from a cxt. However, this turns out to be not as useful as the seemingly more indirect method I will describe, for two reasons. First, we want the operation to implement packet exclusion. Since there may be more than one closure of a packet in a given cxt (and a given layer may be in more than one packet) we must solve the problem of how to "undo" just one inclusion of a packet, removing only those layers with no other attachment to the cxt. Second, the alteration method must preserve the usefulness of any index structure that may have been previously built on a cxt.

Therefore, it is desirable to represent an excision in terms of alterations to the merge history graph. Let us augment the usual merge operation with the *link cut*: a notation to the effect that one edge of the graph is to be invisible in the result of the merge and cxts later derived from it:

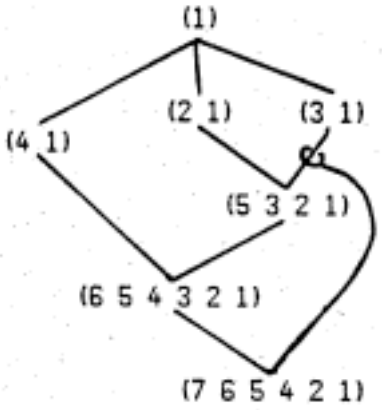


Figure IV.8

In Fig. IV.8, layer 7 has been pushed onto (6 5 4 3 2 1), and layer 3 has been deleted. However, deleting a link doesn't always have any immediate effect.

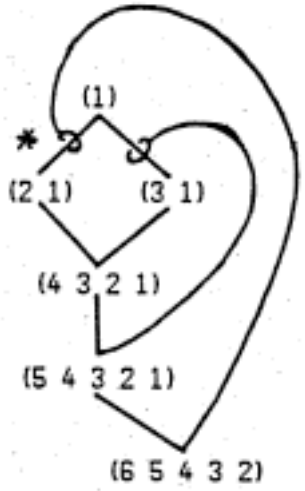


Figure IV.9

In Fig. IV.9, layer 1 cannot be flushed from cxt (5 4 3 2 1), because there is another link "*" to it from (4 3 2 1). But (6 5 4 3 2), generated by excision of "*," has lost layer 1.

Sometimes deleting a link removes more than one layer:

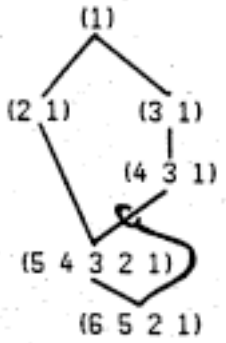


Figure IV.10

Here, snapping the link removes the last tie to layer 3 as well as layer 4.

Unfortunately, there may be more than one path to a link (which corresponds to its being part of a twice-included packet). In this case we must specify which one we mean:

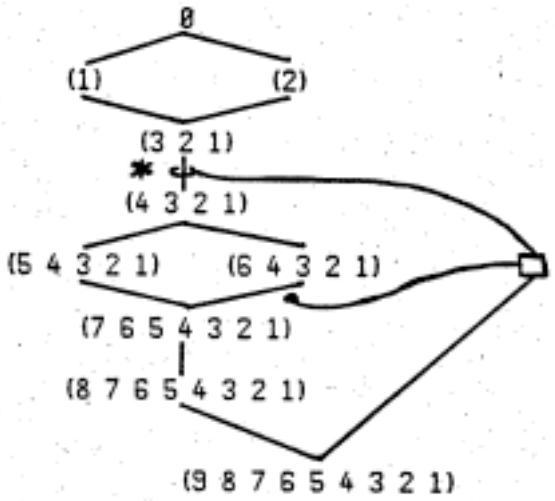


Figure IV.11

Here, we mean to delete one "occurrence" of the link "*," as determined by its subsequent history. The cut is specified by its target link "*" and the path taken to it. Since there remains another path, layers 3, 2, and 1 remain.

In general, the rule is that cutting a link during a merge causes the excision of all layers whose primary cxts can no longer be reached from the result of the merge. A cxt cannot be reached only if every path to every link immediately below it is cut by some node on the path.

In Fig. IV.12, there are four paths to link "*" from (8 7...1).

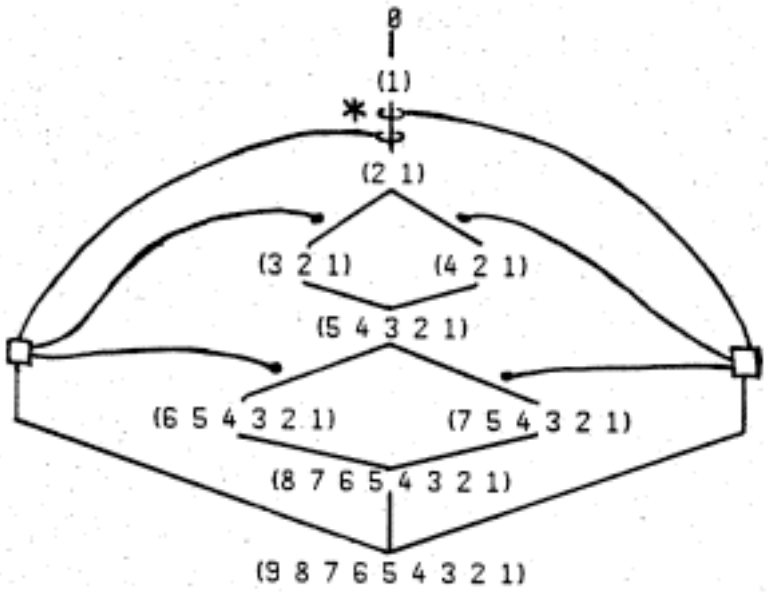


Figure IV.12

Only two of the paths are cut in producing (9 8 7 6 5 4 3 2 1), so layer 1 remains in the merge.

If a cut diagram leaves a path ambiguity, my convention is that all paths to the cut link are simultaneously flushed.

Here is another example of a merged and cut graph:

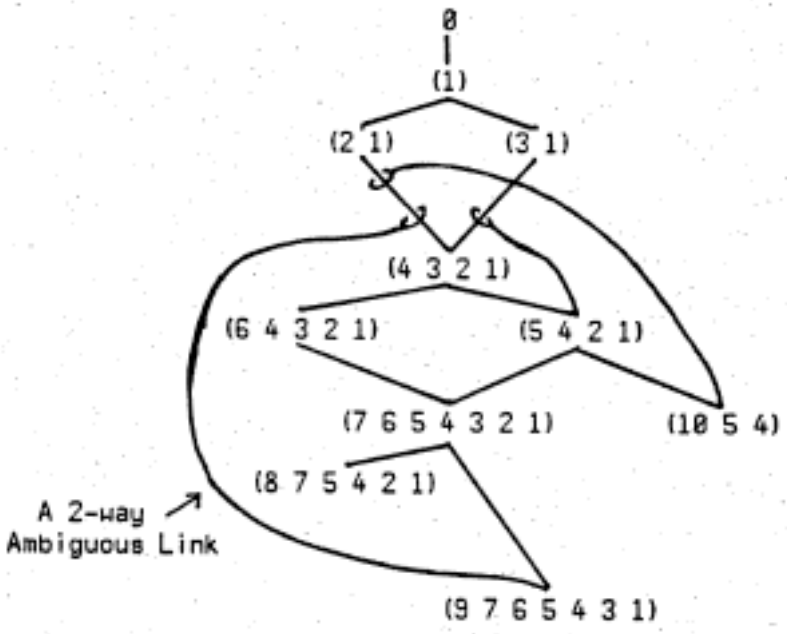


Figure IV.13

The reader should convince himself of the correctness, given the merge history graph, of each of the cxts in Fig. IV.13.

Now that link cutting is understood, I am in a position to describe its advantages in more detail. First, it is complete, in the sense that a cxt with any set of layers can be generated by a merge with appropriate cuts. Namely, merge the primary cxts of each layer, with a cut of every link between cxts:

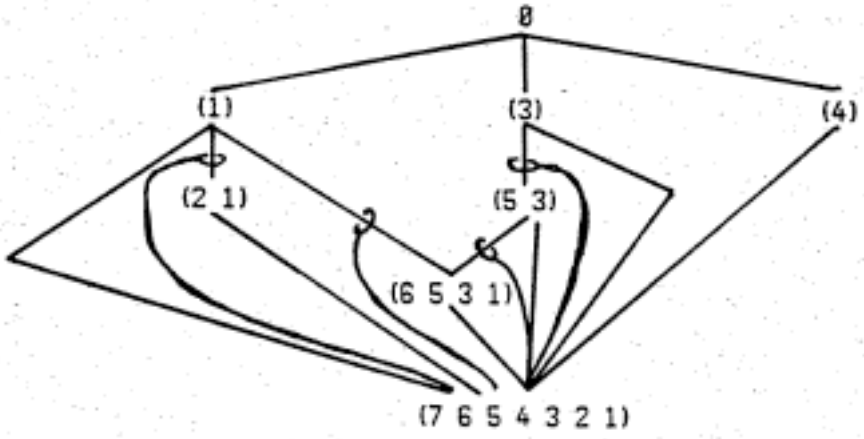


Figure IV.14

Second, as promised, link-cutting is a natural implementation of packet

exclusion. For example, say Clyde and Bonny are elephants, and every elephant has an LTUSK and an RTUSK.

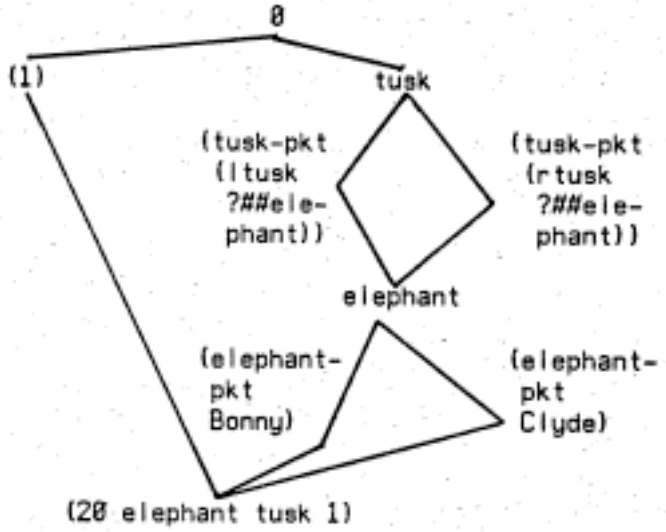


Figure IV.15

The packet-closure (TUSK-PKT (RTUSK CLYDE)) can be removed by cutting the graph this way:

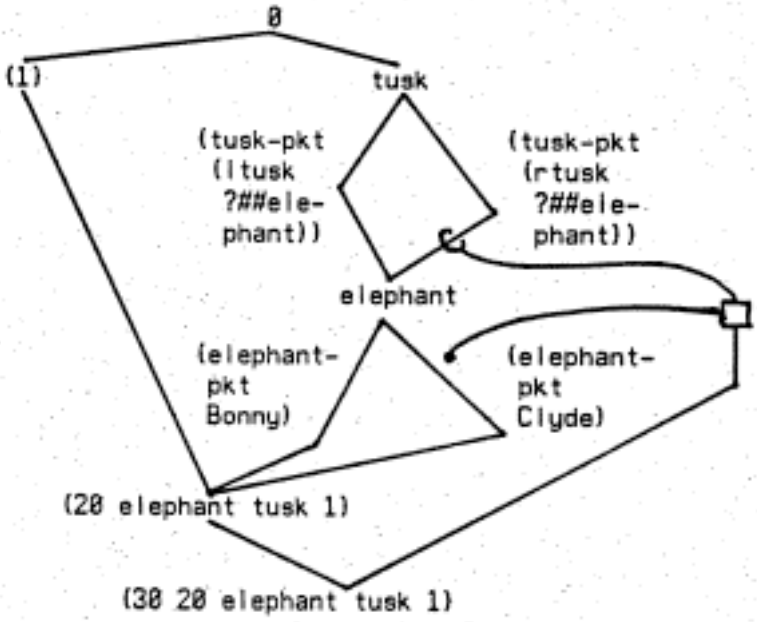


Figure IV.16

The third and main advantage is that we can use the same indexing strategy as for uncut graphs. All we have to be able to do is invert a cxt with some cut branches. In many case such as Fig. IV.16, this is trivial,

since all we have done is prune some redundant branches.

Problems arise with cxts like (10 5 4) of Fig. IV.13. Inverted, its graph is

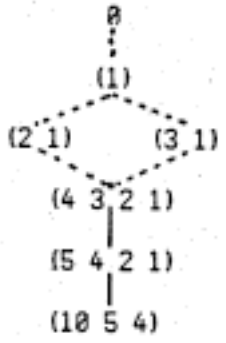


Figure IV.17

where the dotted areas are not there any more. On fetching from this cxt, the system needs a new way to extend the key path <0>.

The simplest approach is do retrieval the same as before, but to ignore the sub-buckets with key paths <0, 1>, <0, 1, 2>, and <0, 1, 3> (if they have been rehashed), and just use them to get to the <0, 1, 2, 4> sub-bucket. A solution like this may be painful if we have to follow a long "phantom" trail like this, only to find an empty bucket at the end. But there is no reason not to hang the <0, 1, 2, 4> sub-bucket directly from the <0> bucket for future reference, and give it the new key path <0, 4>. Cxt (4 3 2 1) becomes a "pseudo-child" of 0. The link from the <0> bucket to the <0, 4> sub-bucket is only used for sub-cxts of (10 5 4) and its ilk. (Notice that the system must do this for every bucket in the index rehashed on these cxts; having organized the (COLOR...) bucket this way will not help us on a fetch of (SIZE...).)

The implementation of cxt exclusion is compatible with the trick of associating with every bucket a list of its sub-buckets, to be used to speed up searches for isolated sub-buckets. (Sect. IV.A.1) Pseudo-child links do not alter the sub-bucket list, but merely shorten the paths to

some sub-buckets in some circumstances.

IV.A.3 Cancellation

So far, I have treated the cancellation of a cmarker (Sect. II.A.2) as an infrequent occurrence. It is handled by the still-necessary cxt filtering step that is done on the final output from the index. This is proper if only a few data are cancelled in a particular cxt.

However, there may be cases in which cancellation is so frequent that the system I have outlined works less well. (It can't do worse than the current algorithm.) For example, assume again that cxts are being used to model successive situations, whose approximate time is represented as items of form (TIME t). A sequence might look like this:

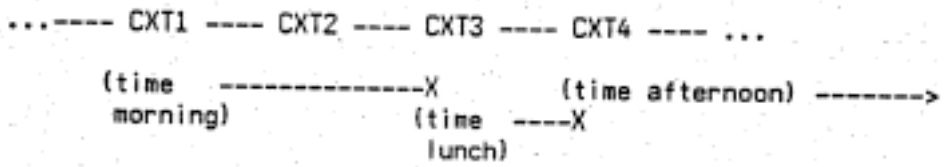


Figure IV.18

The X's mark the cancellation of an item. Each successive cxt is formed by pushing the previous one. Consequently, a FETCH of (TIME ?) in CXT4 will union the sub-buckets of the <<TIME, CAR>> bucket for CXT's 1, 2, 3, and 4. Only one of the three items found will survive the test for cancellation. By making the string of cxts and items longer, the efficiency can be made arbitrarily low.

My feeling is that in practice this problem will not be serious, because the only cxts that get fat enough to require sub-indexing are the more or less permanent ones that are planned too carefuly to be the result of a long string of cancellations. As discussed in <McDermott, 1974a>, there is a need to reorganize and epitomize cxt sequences periodically if

they are to represent long stretches of time in a useful way. During such a summary, the cancellations tend to be left out.

However, for curious purists who think it important that an algorithm always work, there are ways of avoiding some of the trouble. If you think of cancellation as a complement operation superimposed on the union of sub-buckets for different layers, the problem is that taking one big set from another can given an irritatingly small return from a large investment (while an expensive union is always worth it). Once the system has done such a complementation, however, it might as well try to save the result for future reference. For example, in the <<TIME, CAR>> bucket for CXT4, we can (after one reference) mark the sub-bucket ((TIME AFTERNOON)) as "complete," meaning that the buckets above it should be ignored, not unioned and filtered. The problem with this approach is that some bookkeeping is required to unmark it if an item is added or removed from a super-bucket. (One scheme: every bucket could be marked with its "time of completion"; a complete bucket would be accepted only if its time of completion was after the completion times of all its superiors.)

IV.B Interaction with the TDBM Indexer

I have referred vaguely to the rehashing of certain "buckets" into "sub-buckets" by cxt structure. In Sect. II, I described an independent bucket-sub-bucket system based on pattern features. It has to be decided how these systems should interact.

Logically, there is no particular problem. At each step in a FETCH, the indexer has a bucket and a remaining set of features and cxts that it hasn't used yet. If the bucket is a raw list, it takes that. If it is broken down by features, it uses the features it has; if by cxt, it goes

another step down the inverted fetch cxt. After each step, it has a new bucket and less remaining key material.

The problem is purely strategic. When an oversized bucket needs to be rehashed, there is a choice as to which of the remaining keys to rehash on. For example, if (FETCH '(P A ?)) finds too many items, should it rehash the <P, CAR> bucket by atoms or by cxts?

Notice that no matter how long a bucket is, it is "too long" under just two kinds of circumstance: either the cxt filtering step or the match filtering step rejected too large a fraction of its input. The proper rehashing strategy, therefore, is to rehash on the key corresponding to which of the two problems actually occurred. Thus, if there are twenty items in the <<P, CAR>> bucket, but only one is in the fetch cxt (4 1), the system should rehash by cxt and take the <<P, CAR>, 1> sub-bucket. If 19 of the items are in the cxt, but only one or two match (P A ?), it should rehash by features, and take the <<P, CAR>, <A, CADR>> sub-bucket.

Of course, there are exceptions. If the cxt filtering step's inefficiency is due to a lot of cancelled items, different remedies are called for. (See Sect. IV.A.3.) If a bucket is already rehashed, as is the <<P, CAR>, <P, CAR>> sub-bucket of Fig. II.4, there is no point in rehashing by features. The inefficiency can happen in this case only if the fetch pattern is something like (P ?X ?X) and there are too many items of the form (P *s1 s2*) with *s1≠s2*. (I do not know if there is a way to handle this case efficiently, or how important it is.)

With this scheme, the system can start out with one bucket for all data, with key <>. It treats this top-level bucket the way it would any other, rehashing it by relevant key when it gets too big.

IV.C Secondary Storage

An advantage of the cxt indexing scheme over the pure cxt filtering of the TDBM is that it appears to be compatible with efficient use of slow, non-random-access storage. Because we do not actually have to see items not in the current cxt, they can be purged out onto a disk or drum.

However, use of secondary storage will require development of a record I/O system for handling item data. It will be impossible to use LISP addresses, since LISP's address space is so small (and since most LISPs make a mess of secondary storage use). In particular, I doubt that symbols can be stored as LISP-like atoms, unless there is a large obarray-like hash table on the disk for simulating LISP's obarray. (Presumably, the analogous "uniquizing" function for items, called DATUM in Conniver, will have to be foregone, since the system can't afford to search the disk for a variant of a new item.)

So there are two approaches one can take: either wait for a LISP with a large, efficiently organized address space, or debug the following hack:

Assume symbols are represented by character strings. The system will have to use LISP's READ (or a streamlined version) to read items from the disk. Of course, we do not have to specify all of the characters in an item's printed representation, but only those left unspecified by the keys to the bucket the characters were found in. For example, if the <<P, CAR>> bucket of Fig. II.4 were written to the disk, it could be stored as ((a), (b), (c), (d), (e)) (plus cxt markers). Furthermore, the system can save time by postponing the actual READing of characters when they are brought in, until they are needed to reconstitute a bucket. That is, there is a buffer step between asking for a bucket and reading from the disk:

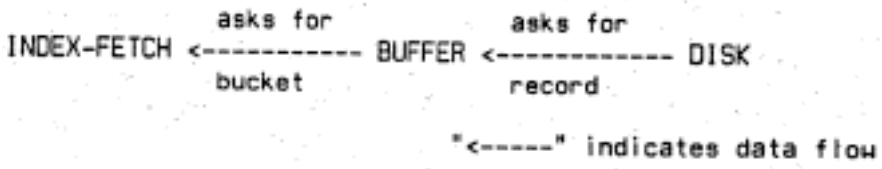


Figure IV.19

The most crucial requirement for such a system is that it respect locality of reference with respect to cxt. That is, when one bucket for a cxt is brought in, other buckets for different features in the same cxt should be stored in the same records. This can be achieved by keeping a list of all buckets and sub-indexes directly associated with a cxt layer, and writing them out together. (This will be cheaper if buckets are broken down by cxt before features; cf. Sect. IV.B.) Since the indexer needs the sub-buckets for all super-cxts when FETCHing from a cxt, a FETCH from a long-neglected cxt might cause several whole branches of a sub-index to brought in from disk.

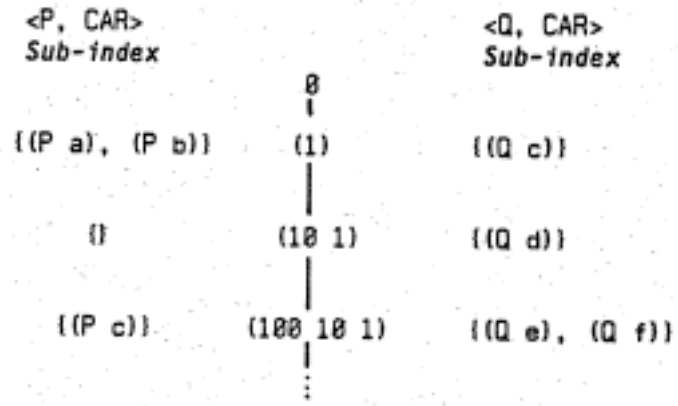


Figure IV.20

In Fig. IV.20, the items shown are written out in order (P a), (P b), (Q c), (Q d), (P c), (Q e), (Q f),.... A reference to (P ?) in cxt (1) will anticipate a reference to (Q ?) by bringing (Q c) in.

The complexity of the issues regarding secondary-storage management is the main obstacle to implementing the cxt indexing system I have described. Without taking secondary storage into account, an implementation would begin to thrash from external reasons (limitations of PDP-10's and LISPs)

before its advantages began to tell.

V *SO WHAT?*

The reader should by now be convinced that it is possible to implement large Planner-type data bases efficiently, if they can be well organized. The time required to fetch a pattern from a cxt depends on the size of the pattern and the length of the cxt, and to a small degree on the number of closely related data that might seriously confuse a more standard system.

It would be nice if I could prove a theorem regarding the average cost in time and space of using a system like the one I have described, but I lack the expertise to do it. I find it hard to imagine cases in which its behavior would be unacceptable, but I am probably overlooking some peculiar interactions. Most of the interactions I see are actually beneficial. For example, the more features in a pattern, the more sub-indexes to look through, if there are a lot of data. On the other hand, when a pattern has a lot of features, there are unlikely to be near duplicates of it in other cxts, so the bucket that is ultimately found will probably not be rehashed by cxt at all. So performance is probably seldom degraded on both types of key. I can't prove to what degree this is true, however.

The case is even more confused with my packet implementation, which relies on the organizing power of the frame concept. Here what is needed is experience, which I hope we will all soon have more than enough of. The most pressing question we should be asking is, how well does a program have to organize itself to avoid choking on irrelevant packets or frames?

Brute-force efficiency is thus a subordinate issue. A more interesting

cause I would like to rally to is the defense of Scott Fahlman's early paper "A Hypothesis-Frame System for Recognition Problems" <1973> from the critique in his recent "A System for Representing and Using Real-World Knowledge." <1975> The first paper was an elegant exposition of the developing frame theory. It is consistent with the work of Minsky <1974>, Kuipers <1975>, Winograd <1974>, Rubin <1975>, Marcus <1975>, and Moore and Newell <1973>, most of whom claim to be working on frames. People who admire all this research can see how the mechanisms I have described could be helpful.

Or maybe they can't. One problem with my exposition that makes it look different from previous frame theories is that I haven't mentioned recognition at all. Most of the theories are about nothing but recognition. The computational problems all revolve around guessing a frame to account for data, filling in that frame with data, proposing subframes, building super-frames, transforming to new frames, and so on. The only apparent purpose of a frame is to embed it in a more inclusive frame.

Probably this is a minor oversight. We are expected to see by ourselves the many advantages of finding an instance of a large "almost right" chunk of data like a frame. For example, after a medical diagnosis <Rubin, 1975>, a doctor program has a structure of frames representing a hypothesized constellation of diseases. How does it know what treatment to prescribe? Presumably, there is a frame slot TREATMENT = AMPUTATION, or something, which can just be read off.

Indeed, "just reading it off" seems to be the retrieval mechanism frame theorists long for. A frame represents a body of knowledge that is well understood. "A frame is a specialist in a small domain." <Kuipers, 1975, p. 182> This is why typical frame contents tend to be slots and values,

with all other information represented as auxiliary procedures. Indeed, the slot-value system reminds me of nothing so much as a called function with all its variables bound (with some slots being modelled as optional arguments with default values). Kuipers' frame implementation is just like this. His frames are processes which communicate by sending each other requests for slot values and the like.

All of this concentration on slots, values, and message passing makes me unhappy. Why did all these people abandon the Planner-type data base? There is no real difference conceptually in how a Planner-type index stores slot-value pairs. Clearly, if all assertions are of the form (COLOR RED) or (TREATMENT AMPUTATION), it doesn't matter whether they are stored in an a-list or a cxt. The problem lies in representing more complex pieces of information, like "the way to spot a gnurd is that his socks don't match." Well, this is a problem; it might be *the* AI problem.

Let me suggest that a most important feature of a representation of facts like this is that it be obvious what the fact is and why it is believed and what might have been believed in its place and how it *might* have been used, as well as how it *is* being used. Unless a program is such an expert that it knows, by reading off a slot, what to do in any conceivable situation, someday it is going to fail to solve a problem and have to debug its world model. (Cf. <Sussman, 1975>) It will have to start from the debris of its previous efforts. If this debris consists only of program counters of processes confused by conflicting slots, it will be useless. A program that relies completely on passing messages of anticipated form can have only unpleasant surprises. Intelligent people are pleasantly surprised by problems all the time.

Anyway, packets and cxts can support all the usual ways of doing recognition: let the data suggest a frame; focus attention by activating

only a few frames at a time; rely on "demons" (if-added interrupts, for example) to model "noticing"; let the frames guide the search; allow default facts which are easily replaced. Hopefully they will support whatever other functions frames are going to have.

Fahlman's recent paper, "A System for Representing and Using Real-World Knowledge" <1975> repudiates much of this frame tradition, and advocates the use of large parallel networks of "concept nodes" for storing information and doing recognition of unfamiliar objects. Fahlman claims in this paper that relevant subsets of a large set of data cannot be efficiently retrieved from a typical AI data structure; if "relevance" can be modelled using cxts (or frames), I am confident this claim is false. His other claim is that the usual frame-theoretic approaches to recognition I mentioned in the preceding paragraph are not as powerful as doing parallel intersections of large sets of nodes, each representing the concepts with a given property, in order to suggest a familiar concept that might share all of a set of properties. These intersections, which require special-purpose parallel hardware, can be made to solve other well-known problems. For example, disambiguation of co-occurrent terms like "pitcher" and "diamond" can be accomplished by intersecting their sets of possible contexts.

This proposal stands in direct opposition to the usual mechanisms proposed by frame theorists to do these tasks. Each theory makes different assumptions about where the complexity lies. The fundamental assumption of frame theory is that a problem solver should meet a difficulty by finding an organized body of knowledge that applies to it. For example, Marcus's <1975> "wait-and-see parser" uses knowledge about choosing between parses to avoid having to back up. It doesn't have problems requiring large intersections for solution, because it always phrases its recognition

problems in terms which it knows are correct.

A language system knows that "Fido" IS a noun, but it also knows under what circumstances to do pattern-matching on "Fido" (say, in the phonology), and when to call it a noun (in the syntax). It would be possible to do Fahlmanesque "marker sweeps" to implement demon calls triggered by structures the system builds, but, given these guidelines, which are needed on independent linguistic grounds, there are cheaper ways.

Another example comes from Waltz's <1972> research on vision of scenes with shadows. His method would seem to benefit greatly from parallel marker sweeps, until closer examination reveals that enough simple knowledge has been brought to bear on the problem to make the parallelism unnecessary.

It seems at present that the evidence favors frame theory's account of the complexities of intelligence, but the issues are too complex for a verdict now. At any rate, data base efficiency does not seem to be the issue.

Acknowledgments--
This paper is the result of conversations with Scott Fahlman, Bob Moore, and David Marr. The cxt indexing scheme of Sect. IV is a descendant of Moore's solution to the symbol- mapping problem. Gerry Sussman and others as referenced are responsible for most elements of the TDBM. Sussman, Ronald Rivest, and others made helpful suggestions for improving early drafts of this paper.

Bibliography

Bobrow, Daniel G. and Allan M. Collins (1975) (eds) ***Representation and Understanding***, New York: Academic Press.

Boyer, R.S. and J.S. Moore (1972) "The Sharing of Structure in Theorem-Proving Programs," in Meltzer and Michie (1972).

Burstall, R.M., J.S. Collins, and R.J. Popplestone (1971) ***Programming in POP-2***, Edinburgh: Edinburgh Univ. Press.

Fahlman, Scott (1973) ***A Hypothesis-Frame System for Recognition Problems,***

Cambridge: MIT AI Lab Working Paper 57.

Fahlman, Scott (1975) *Thesis Progress Report: A System for Representing and Using Real-World Knowledge*, Cambridge: MIT AI Lab Memo 331.

Hewitt, Carl (1972) *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*, Cambridge: MIT AI Lab TR-258.

Kuipers, Benjamin J. (1975) "A Frame for Frames: Representing Knowledge for Recognition," in Bobrow and Collins (1975).

Levin, Michael I., John McCarthy, Paul W. Abrahams, Daniel J. Edwards, and Timothy P. Hart (1965) *LISP 1.5 Programmer's manual*, (Second edition) Cambridge, Massachusetts: The M.I.T. Press.

Marcus, M. (1973) *Wait-and See Strategies for Parsing Natural Language*, Cambridge: M.I.T. A.I. Lab Working Paper 75.

Marcus, M. (1975) "Diagnosis as a Notion of Grammar," Pre-prints of Workshop on Theoretical Issues in Natural Language Processing, June 10-13, 1975, M.I.T.

McDermott, D. (1974a) *Assimilation of New Information by a Natural Language-Understanding System*, Cambridge: MIT AI Lab TR 291.

McDermott, D. (1974b) *Advice on the Fast-Paced World of Electronics*, Cambridge: MIT AI Lab Working Paper No. 71.

McDermott, D. and G.J. Sussman (1973) *The Conniver Reference Manual*, Cambridge: MIT AI Lab Memo 259a.

Meltzer, Bernard, and Donald Michie (1972) (eds.) *Machine Intelligence 7*, New York-Toronto: John Wiley & Sons.

Minsky, Marvin (1974) *A Framework for Representing Knowledge*, Cambridge: MIT AI Lab Memo 306.

Moon, David A. (1974) *MACLISP Reference Manual*, Cambridge: MIT Project MAC Report.

Moore, J. and Allen Newell (1974) "How Can Merlin Understand?" in Gregg, L. (ed.) *Knowledge and Cognition*, Potomac, Maryland: Lawrence Erlbaum Associates.

Newell, Allen (1962) "Some Problems of Basic Organization in Problem-Solving Programs," in Yovitts, M., G.T. Jacobi, and G.D. Goldstein (eds.) *Self-Organizing Systems--1962*, New York: Spartan.

Rivest, Ronald L. (1974) *Analysis of Associative Retrieval Algorithms*, Stanford: Memo STAN-CS-74-415.

Robinson, J.A. (1965) "A Machine-oriented Logic Based on the Resolution Principle," *JACM 12*.

Rubin, Ann D. (1975) *Hypothesis Formation and Evaluation in Medical*

Diagnosis, Cambridge: MIT AI Lab TR-316.

Rulifson, J.F., J.A. Derksen, and R.J. Waldinger (1972) *QA4: A Procedural Calculus for Intuitive Reasoning*, Menlo Park: SRI Technical Note 73.

Sussman, G.J. and D.V. McDermott (1972) "From PLANNER to CONNIVER -- A Genetic Approach," (*Proc. FJCC 41*, p. 1171.

Sussman, G.J (1975) *A Computer Model of Skill Acquisition*, New York: American Elsevier Publishing Company.

Waltz, D.W. (1972) *Generating Semantic Descriptions from Drawings of Scenes with Shadows*, Cambridge: MIT AI Lab TR 271. Also in Winston (1975).

Winograd, Terry (1974) "Frame Representations and the Declarative/Procedural Controversy," in ???(ed.) *Essays in Memory of Jaime Carbonell*, ????.

Winston, Patrick W. (1975) *The Psychology of Computer Vision*, McGraw-Hill.

Woods, William A. (1975) "What's in a Link: Foundations for Semantic Networks," in Bobrow and Collins (1975).