HIERARCHY IN DESCRIPTIONS

VISION FLASH 46

by

Michael R. Dunlavey

May 73

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

Abstract

Organization of knowledge requires the flexible use of hierarchy in descriptions. This memo attempts to catalog the issues related to recognizing and executing such descriptions, drawing examples primarily from the blocks world.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

Foreword

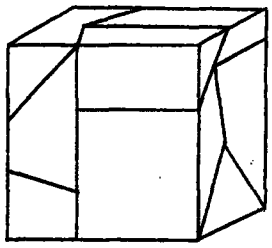
This memo is an octopus, touching just about every current abstract issue in Artificial Intelligence. My interest in hierarchy began as an interest in planning, stimulated by my advisor at Georgia Tech, Prof. Michael D. Kelly. Prof. Marvin Minsky encouraged me to explore the broader issues, giving rise to this memo. I am lucky in being able to build almost directly on Prof. Patrick Winston's thesis(14).

Table of contents

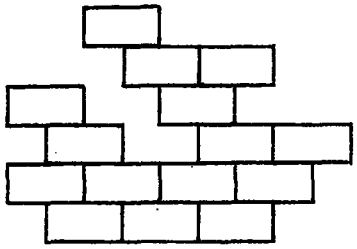
Wholes and parts

Properties of aggregates may or may not be related to properties of their parts.

For example, this aggregate has the property of being a cube, but this cannot in any obvious way be deduced from properties of its parts (example due to Winston).

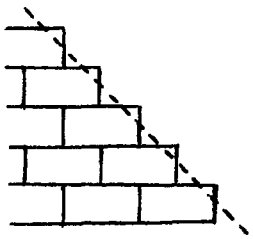


On the other hand, an aggregate may posess strong, simple local properties, like squareness, adjacency, and support, and yet have no interesting global properties, or the global properties may be difficult to determine, like stability.

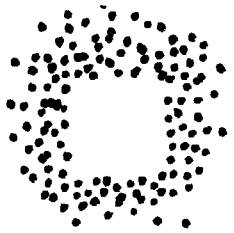


Wholes may not even have parts of their own, but be intimately related to context.

For example, the notion of an "edge" of a brick wall could be defined in terms of the properties of the bricks in the wall, although I suppose the imaginary line would be considered a part of the edge.

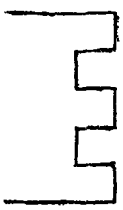


As before, even "partless" wholes can have global properties unobviously related to local properties, just as this hole in a field of random dots seems to be square.

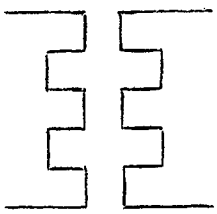


Of particular interest are aggregates in which a global property depends on some arbitrary and identifiable local property.

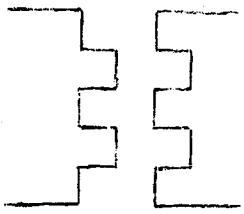
For example, suppose a brick wall is to have a ragged vertical edge.



Now if you take two such walls and try to fit them together, you may find that they don't fit snugly. We have a bug in our system!



However, if one of the walls had been built with its lowest edge brick indented instead of protruding, the walls would fit.



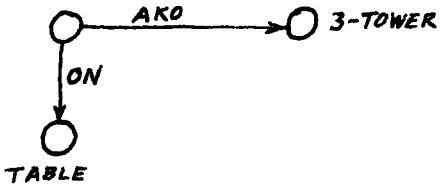
So the "fit" relation between two walls turns out to be related to an a priori insignificant and arbitrary "position" property of a brick. How do we know that there are effectively only two alternatives? I'll discuss this a little further along under patterns of choices.

Combining things

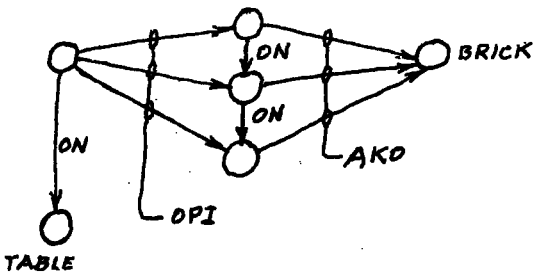
When enforcing global properties, it is often necessary to debug on a local level. The examples I give are primarily about binary properties, but they could just as well be n-ary.

Example

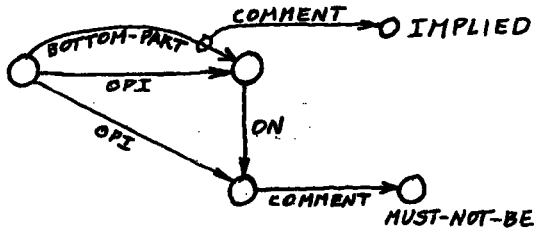
Suppose you want to build a 3-tower ON a table, as represented by this goal description. (AKO means "a-kind-of").

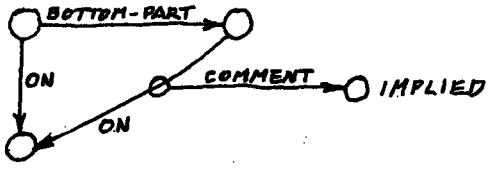


We don't know how to stack 3-towers, only bricks, so we expand the definition of the 3-tower. (OPI means "one-part-is".)

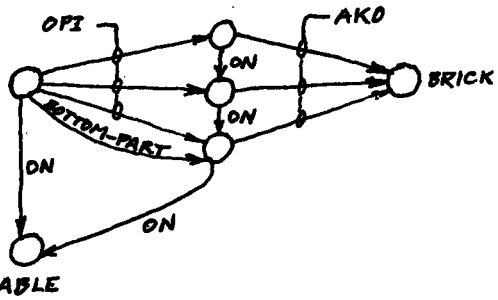


However, the stacker still complains because one of the bricks isn't ON anything, so we bring into play a couple of deductive rules associated with the ON relation. I call them "interface" knowledge because they help put things together.



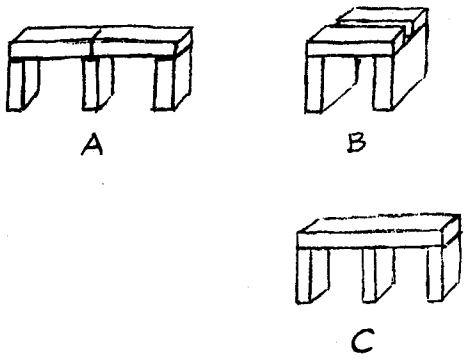


One rule identifies the bottom brick and the other tells what it is ON, so the stacker's complaint is satisfied.

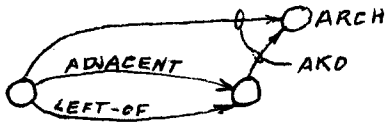


Example - Sharing parts

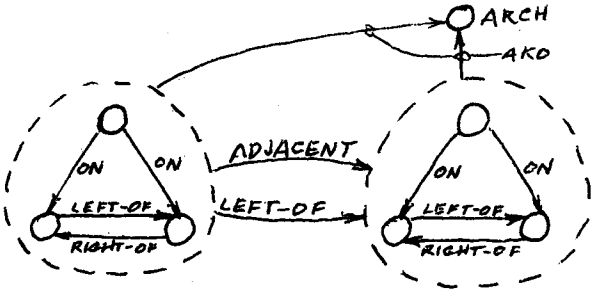
Sharing parts is risky if you don't know when to stop. For example, A is obviously two arches, but is B two arches? or C? Obviously your salvation is the notion of a "hole".



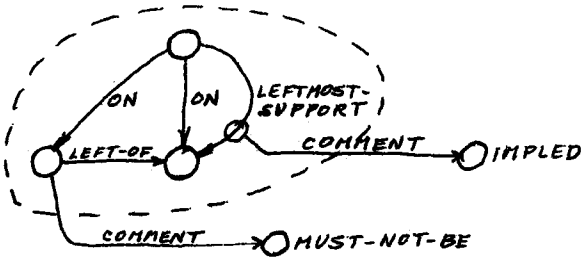
Anyway, suppose you want to build two arches adjacent to one another.



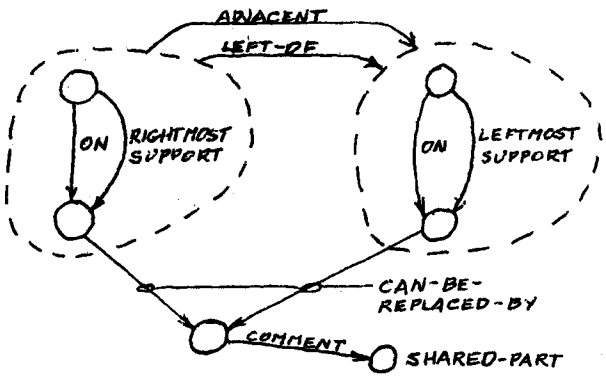
First expand the arch definition. (I'll use dashed balloons from now on to represent OPI aggregates.)



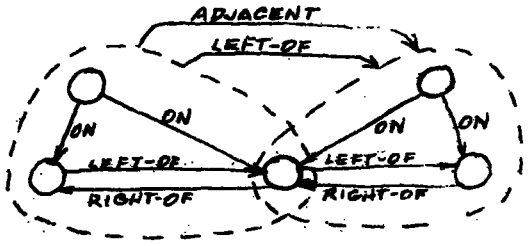
Then this rule and its mirror image identify the leftmost and rightmost supports.



Then this rule identifies the fact that the two adjacent supports can be replaced by one.



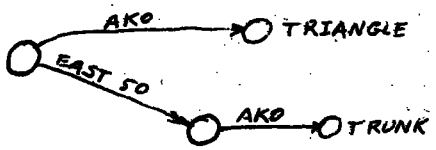
So the description is modified to share the center support.



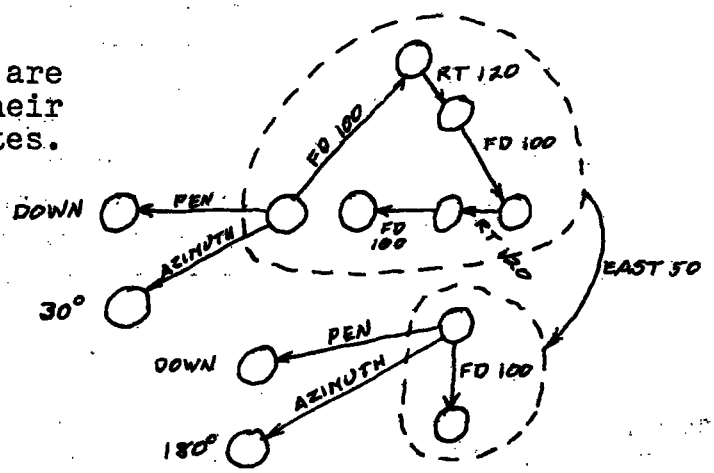
Example - Adding intermediate parts

Sometimes new parts must be added in order for two aggregates to be placed in a relationship. This example is adapted from one by Ira Goldstein.

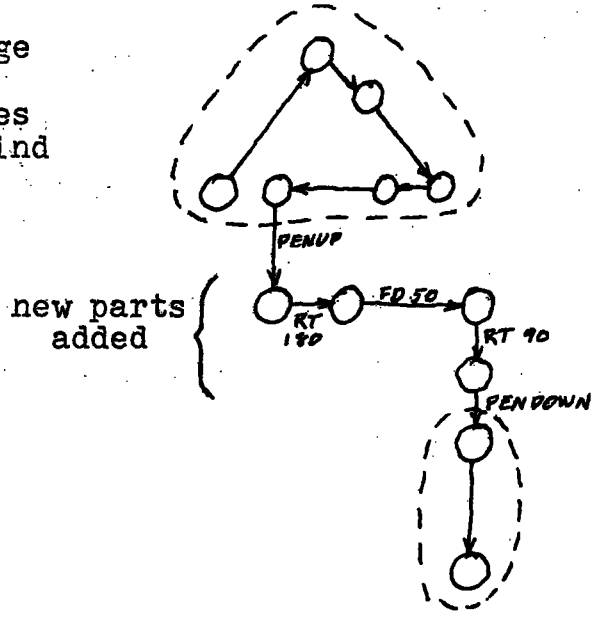
The goal is to make a program that draws a tree, given that you already have subroutines for a triangle and a trunk.



The subroutines are expanded into their constituent states.



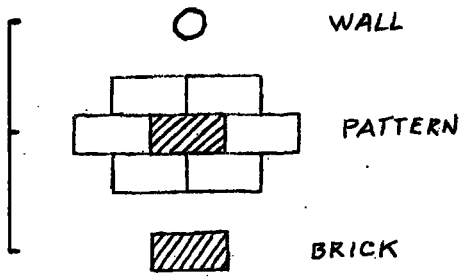
Then interface knowledge detects incompatible beginning and end states and does a search to find a connecting sequence of states.



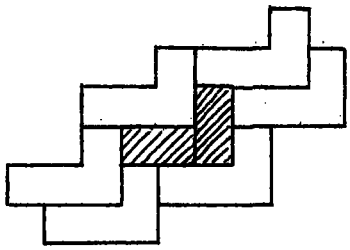
Patterns

A pattern is an aggregate which repeats to form a still larger aggregate.

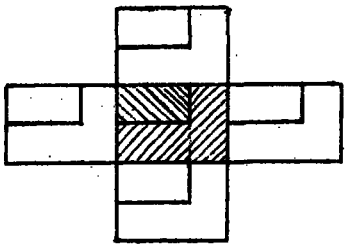
So the description of a brick wall might have three levels of detail instead of just two.



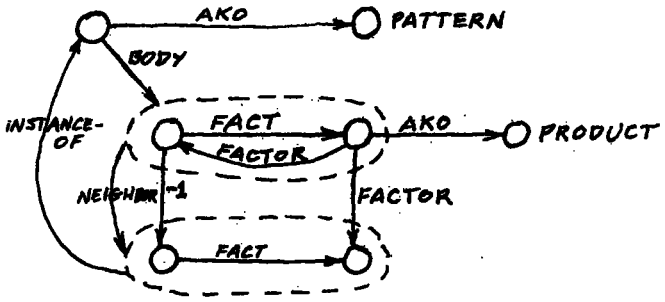
An example of a brick pattern which is an aggregate (since the pattern body contains two bricks):



A pattern may be multi-level:

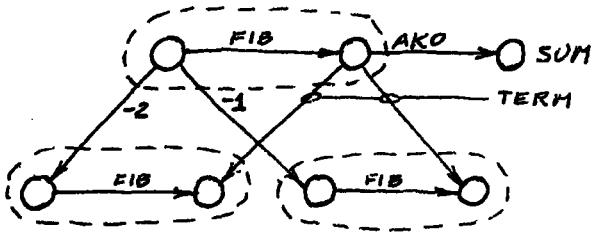


The recursive part of the factorial function is easily described as a pattern.

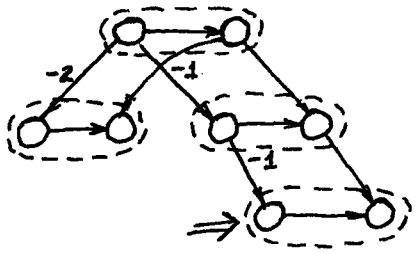


Using the notion of sharing parts and some simple rules about patterns, we can optimize the fibonacci function:

The recursive part of the fibonacci function would look something like this:

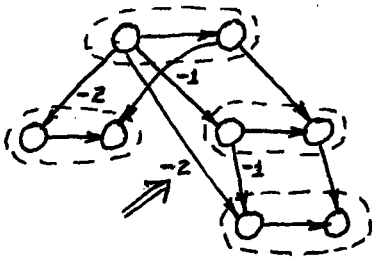


However, this definition takes exponential time to compute, so it can be reworked as follows. First one more neighbor of the pattern can be instantiated.

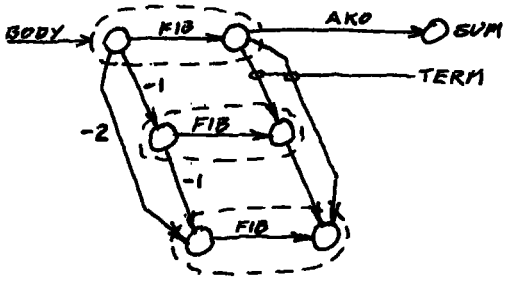


Then a little rule that we planned to try to use wakes up

and puts in its two cents worth.

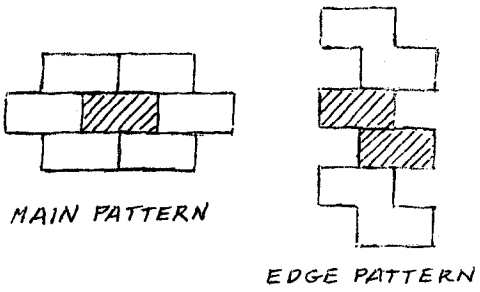


Next the two bottom leaves are seen to be identical and are merged.



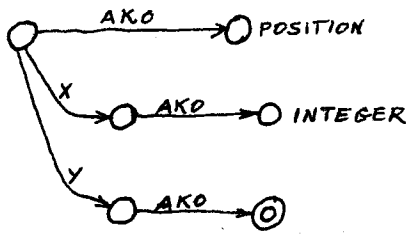
This description now only takes linear time to compute. Though a simple example, this illustrates the possible overlap in ways we handle patterns and aggregates and ways we handle programs.

A more complicated example of an exception is the edge pattern of a wall, in which the exception is itself a pattern.

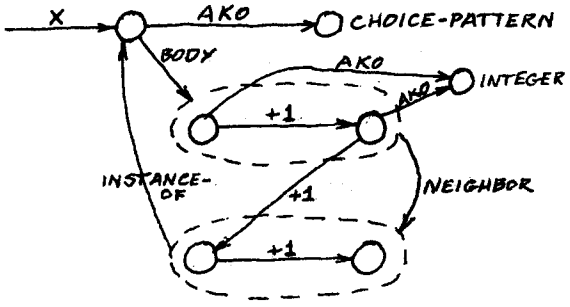


Since choices are aggregates, they can also contain their own patterns. This is useful for getting a handle on the global properties of a brick wall.

When building a wall, the first brick has to be put someplace, and this decision is arbitrary. Assume our coordinate system is over the integers. Then the place we can put the first brick is (integer, 0).



However, one position is equivalent to another if they generate identical walls, so there is a pattern among the alternatives, such that each component of the pattern corresponds to an equivalence class of alternatives. So in a sense there are only two alternatives.



Pattern induction

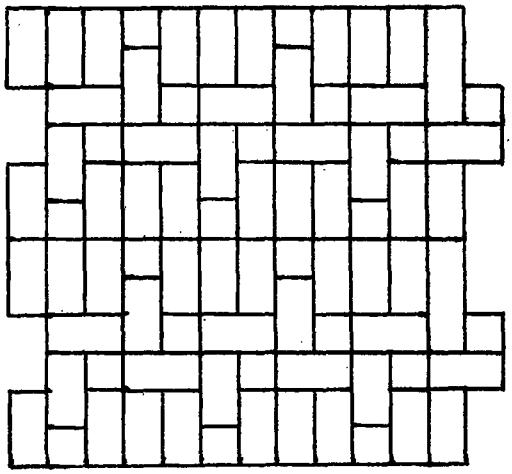
The generic induction problem is: "Given a field of data, find the pattern in it." This requires some syntax which can enumerate all patterns. The brute force method is: for each segmentation of the data, for each pattern, see if the pattern matches the data.

A trick used by Winston is to first look for specific simple subpatterns in the data, like a chain of pointers of the same type, or a bunch of pointers of the same type pointing into or out of a single node. These are easy to spot even in a haystack of data. A subpattern suggests a segmentation of the data, and from there on the brute force method can proceed quickly.

I think the key to pattern induction is to have a well organized hierarchy of subpatterns to look for. By "well organized" I mean that if a subpattern is perceived or is almost perceived, it suggests some other related subpatterns to try next - another Winston idea.

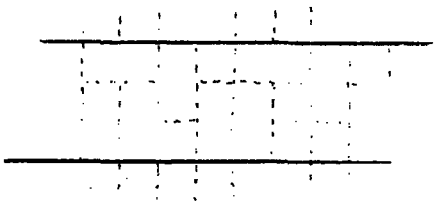
The following (hard) problem illustrates the idea:

What is a small description of this arrangement of bricks?

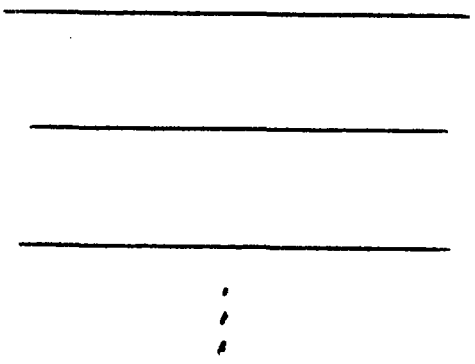


Having shown this to our secretary and interpreted her comments according to my own bias, I think it is safe to say that she did the following steps:

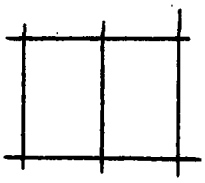
1. The first thing she noticed was one or more <u>lines</u> running through the picture.



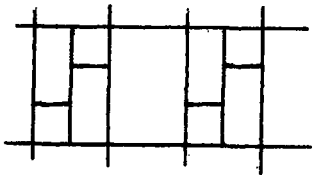
2. Then she noticed that these long lines repeated in parallel.



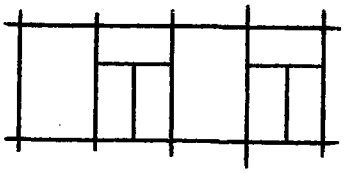
3. Then she noticed some perpendicular lines doing the same thing, and then noticed the squares.



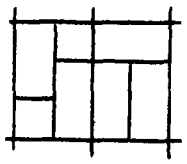
4. At this point the number of bricks within one square was reasonably small, so she tried seeing if each or every other square was the same, and every other one was.



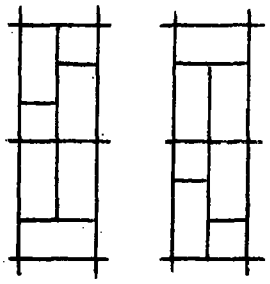
5. She then looked at the in-between square, and it also repeated.



6. Then she simply fastened together the two generic squares and she had found the pattern body. However, she didn't discard the two squares.



7. She knew that the pattern repeated horizontally, but would it repeat vertically? However, she did find that each square always had the other, inverted, as a vertical neighbor, and she was done.



Obviously, other correct answers are possible simply by switching components from left to right or up to down.

In steps 1-3 Suzin was, in a sense, planning. A straight line, like Winston's chain, is a pattern which is very simple, easily looked for and verified. It acted as a powerful clue which not only suggests more elaborate patterns but segments the data.

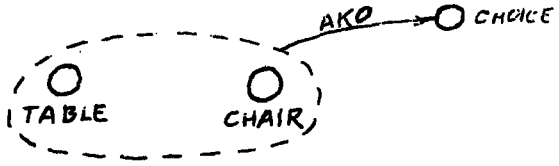
Step 4 was the first point at which she broke out from her little hierarchy of basic patterns and used syntax. She simply described something and looked for something similar elsewhere. Steps 5-7 are back in the network of pattern forms, and she finds the answer.

Notice that she wasn't upset that her description didn't explain the boundaries of the arrangement. I don't even think she noticed.

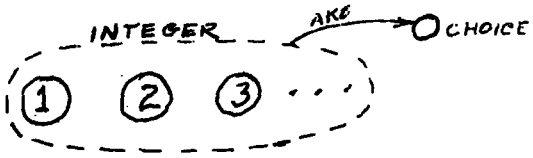
Choices

A choice is an aggregate from which one or more parts may be chosen.

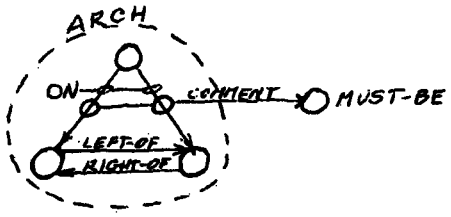
An example of an explicit choice would be the OR of "table" and "chair".



All of the concepts we have already seen implicitly describe choices, for example, "integer" is like a choice from 1, 2, 3, ...

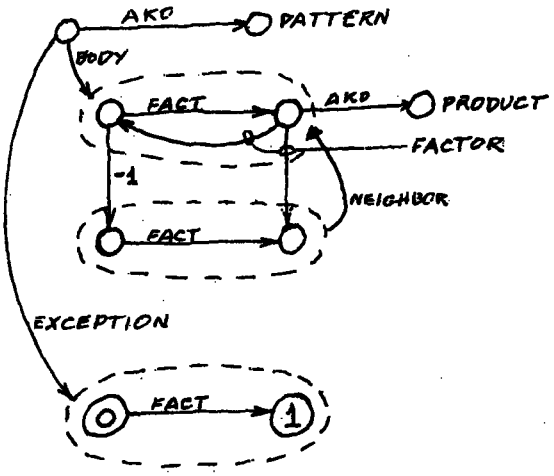


and Winston's arch is like a choice among all structures that match it.



Another kind of choice is what I call an "exception" to a repeating pattern. An exception intercepts an instance of a repeating pattern and overrides it in some way.

This is easiest to illustrate for the factorial function.



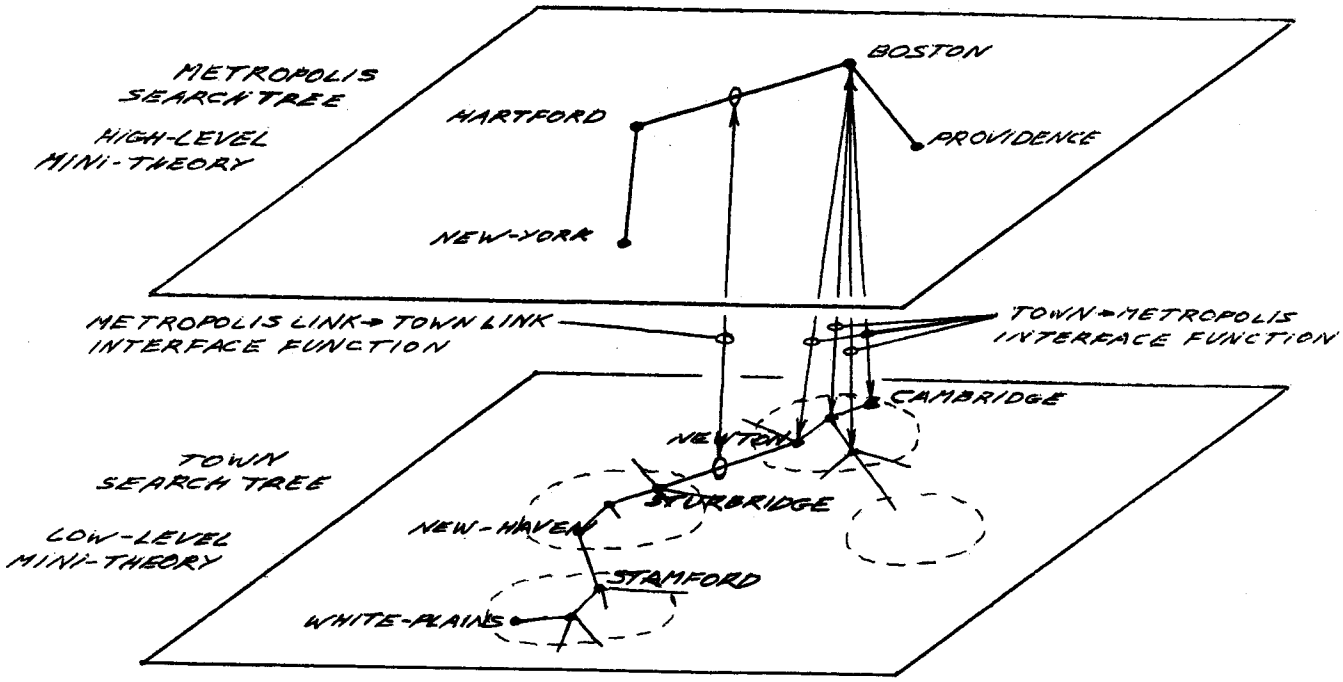
Notice that, with choices, descriptions attain computational universality.

Two paradigms of mini-theory interaction

These two paradigms are not the final answer to anything, but we needed them as an intellectual crutch in understanding the representation problem.

Figure 1. illustrates the paradigms in the context of a search problem. Assume you're trying to drive from one town to another in New England. Assume you have two maps, or mini-theories, to use, but these maps contain no directional information, only connectedness. One map displays connectedness of towns, and the other displays connectedness of metropolitan areas. In addition, you have two tables, one giving for each town its metropolis, and the other giving for each link between metropoli its representative link between towns. For these two tables we have coined the term interface knowledge because they relate one mini-theory to the other. Refer to figure 1. to see how the problem is solved.

To clarify the distinction between the two paradigms, they arose while we were considering ways to influence a strict AND-OR tree search from outside. Most planning programs or general search direction methods operate by trying to pick the best choice at each OR node. If you can influence OR's, why can't you influence AND's? What is an AND anyway? It is a bunch of things that exist together, that is, structure. That's what paradigm 2. does, it alters the structure of the problem, in this case by adding intermediate goals. So paradigm 1. in-



Original problem (low level):

(GO CAMBRIDGE WHITE-PLAINS)

Transformed problem (high level):

(GO BOSTON NEW-YORK)

Solution of transformed problem (high level):

(LINK BOSTON HARTFORD)
(LINK HARTFORD NEW-YORK)

Paradigm 1 says new low level problem is:

(GO CAMBRIDGE WHITE-PLAINS
(NOT-STRAYING-FROM-SEQUENCE
(BOSTON HARTFORD NEW-YORK)))

Paradigm 2 (plus paradigm 1) says new low level problem is:

(GO CAMBRIDGE NEWTON
(STRAYING-NOT-FROM BOSTON))
(LINK NEWTON STURBRIDGE)
(GO STURBRIDGE NEW-HAVEN
(STRAYING-NOT-FROM HARTFORD))
(LINK NEW-HAVEN STAMFORD)
(GO STAMFORD WHITE-PLAINS
(STRAYING-NOT-FROM NEW-YORK))

Figure 1.

cludes classic pruning and hill-climbing, while paradigm 2. is more like macro expansion.

Of course, this leaves out a lot. There are other ways mini-theories can interact, such as when one helps to interpret the other. Additional mini-theories, such as one preferring geometrically closer moves, would help greatly. This says nothing about debugging, which is deciding which choices to remake and how after running into trouble. However, the structure of a plan can help a lot in debugging by telling you which choices are independent.

Research topic - planning in constructing toy buildings

A good problem in which to study planning is to simulate the building of toy houses out of bricks. We would like to be able to state "rough" or "vague" descriptions like:

A box is four walls arranged to meet at four square corners.

A house is a box on which two opposed walls are topped with peaks, and roof panels are supported by the edges of the peaks, and one wall has a door.

A church is a house in which one of the peaked walls has a steeple in the middle of it.

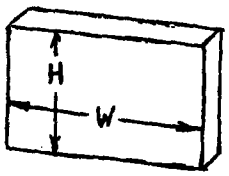
A fort is a house surrounded by a box with a door in it.

To build such objects in simulation, a program will need at least two domains of knowledge, one about walls and one about bricks.

Bricklaying is a very rich problem area compared to something like searching networks. For example, the interface knowledge will have to do problem solving of its own, and it will have to use a domain of knowledge about manipulating procedures into equivalent forms.

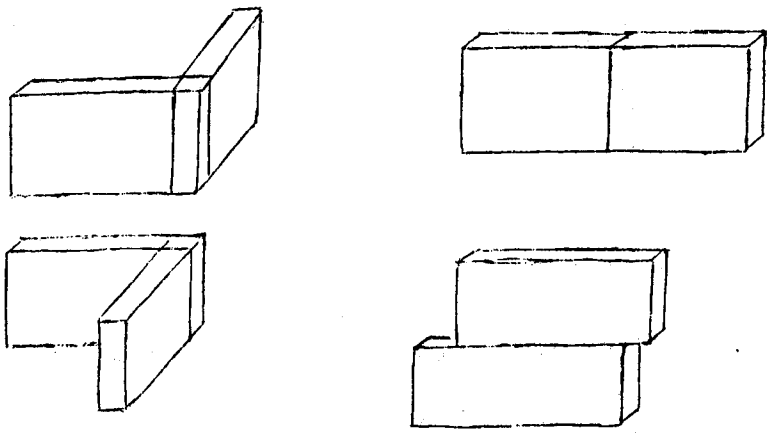
A domain of knowledge about walls

A wall is a rectangular parallelepiped having height, width, and unspecified thickness:



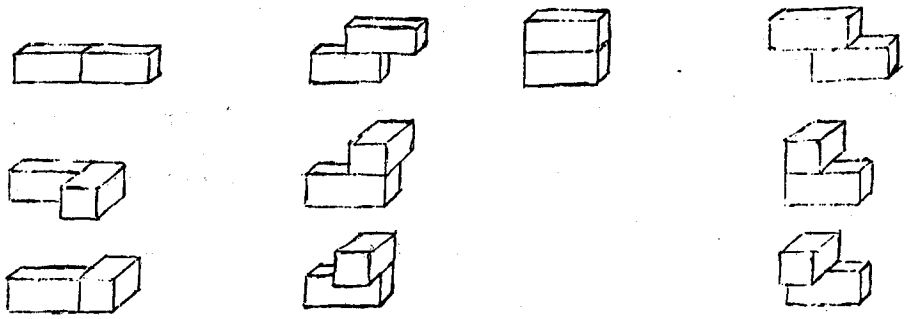
In this domain we're not saying anything about what a wall is made out of; it could be bricks, glass, concrete, or wood.

Walls may be placed in relation to one another in a small number of ways:

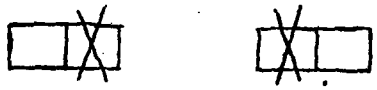


A domain of knowledge about bricks

A brick is an object whose shape is the same as two cubes joined at one face. Bricks may be put in relation to one another in a small number of ways:



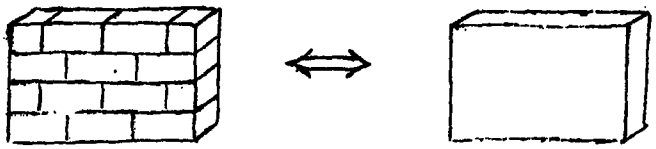
A half-brick is a cube formed by removing one cube from a brick. Any brick may be made into a half-brick by removing either half:



Bricks may not be placed in such a way as to intersect spatially. If such a condition occurs, it may be corrected by deleting either a half-brick or a whole brick.

Interface knowledge

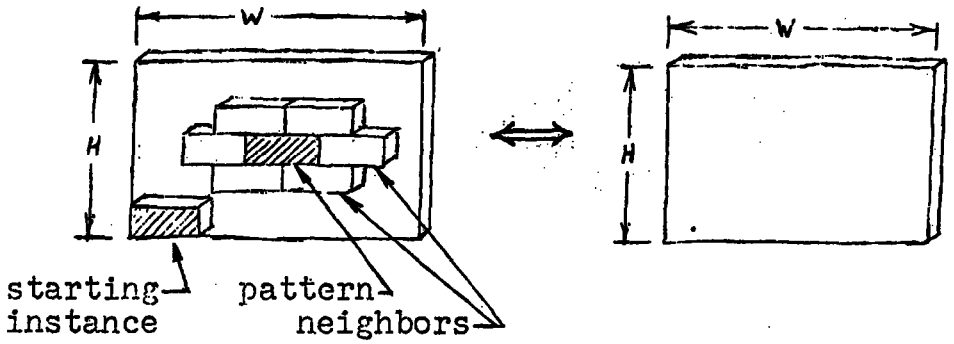
The main component of interface knowledge between the domains of walls and bricks is a description of the structure of bricks that constitutes a wall. If all walls had the same width and height, we would need only a single static description:



relational structure of bricks

wall

However, since we allow walls to have any height or width, we need a more flexible description of the possible brick structures. One kind of description employs a repeating pattern and some edges, plus a starting instance of the pattern. The pattern would start propagating at the starting instance and be delimited by the edges.



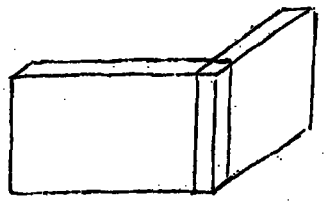
flexible description of brick structure

wall

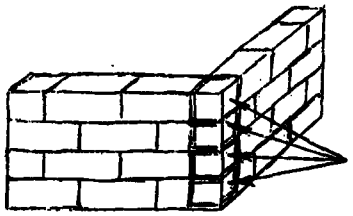
Another reason for using this type of description is that it conserves memory when compared to a brick-by-brick description of a wall.

Building a corner

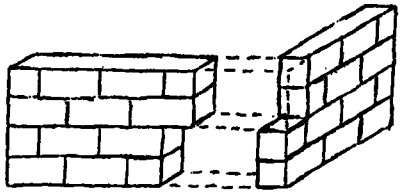
Now we can do a little scenario of building two walls which meet at a corner. The goal is to make one of these:



We first have to expand the walls into their equivalent brick structures, using the interface knowledge. Assume for now that we have enough memory to make brick-by-brick descriptions of the walls.


spatial conflict

However, the problem isn't solved yet because there is spatial conflict among bricks at the corner. So each pairwise case of conflict is resolved by removing a half-brick. One possible combination of removal choices produces this nonconflicting structure:



which can be directly built.

Notice that neither of these two modified brick structures any longer conforms to the definition of a wall, as defined by interface knowledge, yet we would still call them walls. They are, in fact, unforeseen variations of the concept of a wall. If we still call such a structure a wall, does that mean our idea of a wall is vague? While mathematicians eschew vagueness in the concepts they use, we would like to offer the opinion that some vagueness is essential to thinking about complex things, as this example shows.

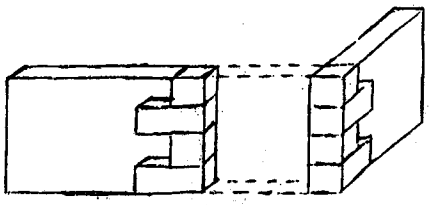
Programmers run into the same trouble in writing big systems. A big subroutine is written to do complex

task A, and another to do complex task B. However, if someone wishes to do both A and B, say in sequence, likely as not he will run into trouble because the two subroutines are incompatible in minor ways.

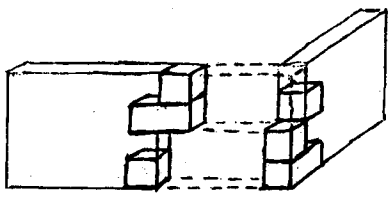
Conflict resolution under memory restrictions

Going back to the wall-wall example, the step where we generated all bricks in each of the walls would have been impractical for any but very small walls, due to memory and time restrictions. However, we can make use of the fact that the walls are described by repeating patterns.

The idea is to have a demon detect the region of possible conflict by looking at the dimensions and positions of the walls. Then a domain of knowledge that knows about pattern manipulation would walk the pattern all about the region of possible conflict so that only bricks that might be involved in conflict will be generated:



The conflicts are resolved as before:



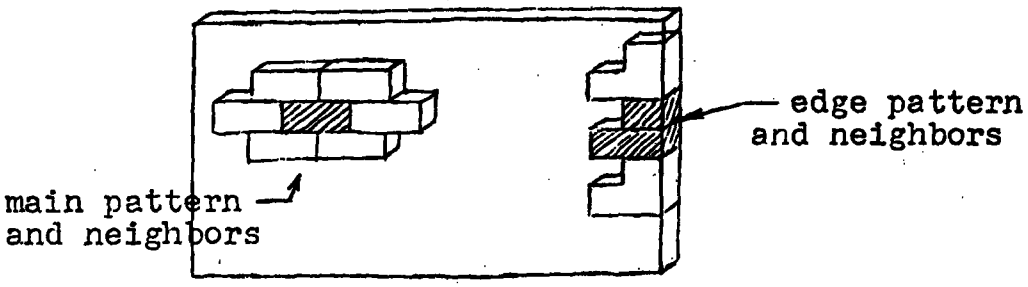
Then the walls are built from their patterns, working around the specific modified instances of the pattern.

Procedure manipulation

Going back again to the problem of resolving the conflict between the two walls, it could even be that it will be uneconomical to remember all bricks along a single edge. For example, imagine thinking about all the specific bricks on one corner of the Washington Monument. We are trying to suggest how to handle big problems!

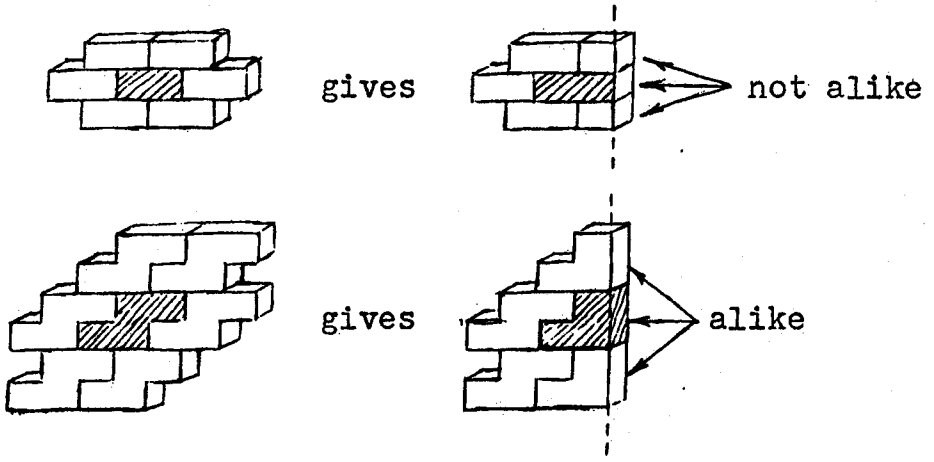
One could argue that it's not necessary to do all this conflict resolution in advance of actual construction, which is correct for some problems. Where the building material and fabrication methods are cheap, as in masonry, it's not necessary to plan so carefully. But in steel construction of buildings, it is necessary to plan carefully.

In the wall-wall example, it happens that the bricks along the conflict edge can be described by another repeating pattern:

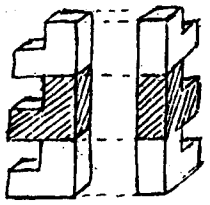


The problem now is to find out what that edge pattern is. This can be done with the help of a domain that knows how

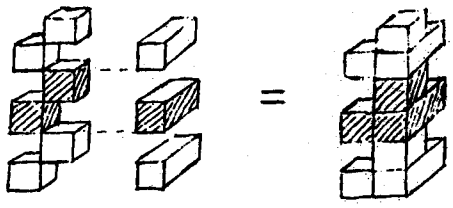
to manipulate procedures, because these patterns are actually simple procedures. The basic idea is to try transforming the main pattern into equivalent forms in such a way that, when scissored by the edge, it and all its vertical neighbors are alike:



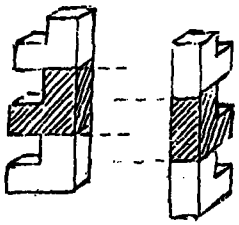
Once an edge pattern has been found for each wall, it is possible to try to resolve the conflicts by just resolving the conflicts in the patterns. For example, if the patterns come together like this:



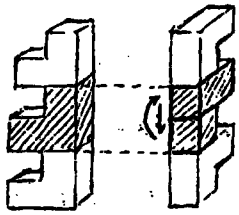
they can be resolved like this:



If the two patterns do not match up in their upper and lower boundaries, some additional procedural manipulation may be necessary to make them match. For example, these two patterns cannot be resolved because their boundaries do not match:



However, the pattern on the right can be transformed, by cyclically permuting its parts, so as to be comparable to the left hand pattern:



This should illustrate the importance of having a domain of knowledge about how to manipulate procedures, of which these patterns are simple cases. This knowledge is specific in the sense that it could be an independent problem domain, but it is general, like mathematics, in that it can be brought to bear in a wide variety of other problem domains.

Data Structure Design

These are some half-baked thoughts about data structure design, which is crucial to non-trivial learning as well as to automatic programming. I feel that there are strong parallels between data structure design and problem-solving, but I'm not sure what they are. Anyway here is some food for thought.

Basic ideas

1. A data structure's primary purpose is to provide primitives with which to describe problems.

2. A data structure design is a procedure which can generate any particular problem description.

3. A data structure design should come with something of a ready-made theory for the class of problems it can represent.

4. A subroutine distills repetition in a program description, while a variable distills repetition in an execution of a program.

5. A variable corresponds to a network node, and its values correspond to the node's possible properties.

How to make small additions to data structure design

1. If you perceive a pattern in the program you are writing, make it a subroutine.

2. If you perceive that there will be a pattern in the execution of the program you are writing, make a variable.

How to design a data structure globally

1. You have a catalog of basic data structure types and attendant theories. Find one in which you can state your problem and use it.

2. In one sense you just "collect" together all the things "needed" by the various subunits of your program and interface them, much like gathering ingredients before baking a cake.

Bibliography

1. Butcharov, Payanot. Resemblance and identity: an examination of the problem of universals. Indiana University Press, 1966.

2. Eberle, Rolf A. Nominalistic systems. D. Reidel publishing Company, Dordrecht, Holland, 1970.

3. Feigenbaum, Buchanan, Lederberg. On generality and problem-solving: a case study using the DENDRAL program. in Meltzer, Michie (Ed.) Machine Intelligence 6. Edinburgh, 1971.

4. Freuder, Eugene. Suggestion and Advice. Vision Flash 43 (internal memo), M. I. T. Artificial Intelligence Laboratory, March 1973.

5. Goodman, Nelson. The structure of appearance. Harvard University Press, Cambridge, Massachusetts, 1951.

6. _____, and Quine, W. V. O. Steps toward a constructive nominalism. Journal of Symbolic Logic, XII (1947), 105-123.

7. Hewitt, Carl. Teaching procedures in humans and robots. in Proc. Conference on structural learning, Philadelphia, April, 1970.

8. _____. PLANNER: a language for manipulating models and proving theorems in a robot. in Proc. International joint conference on artificial intelligence, 1969.

9. Kelly, Michael D. Edge detection in pictures by computer using planning. in Meltzer, Michie (Ed.) Machine intelligence 6. Edinburgh, 1971.

10. Loux, Michael J. Universals and particulars. Anchor Books, Doubleday and Company Inc., Garden City, New York, 1968.

11. Minsky, Marvin L. Artificial intelligence. in Project MAC progress report VIII. 1971.

12. Piaget, Jean. Structuralism. Basic Books, Inc., New York, 1970.

13. Polya, G. How to solve it. Princeton University Press, 1954.

14. Winston, P. Learning structural descriptions from examples. M. I. T. Project MAC TR-76, AD 713-988, 1970.