SNAP: A Marker-Propagation Architecture for Knowledge Processing

Technical Report No. CENG 89-10

Dan Moldovan, Wing Lee, Changhwa Lin

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California

September 27, 1989

[0]This research has been funded by the National Science Foundation Grant No. MIP-89/02426

Abstract

Semantic Network Array Processor (SNAP) is a highly parallel architecture targeted to Artificial Intelligence applications, in particular, natural language understanding. The knowledge representation used is a form of semantic network. The knowledge base is distributed among the elements of the SNAP array, and the processing is performed locally where the knowledge is stored. A set of powerful instructions specific to knowledge processing is implemented directly in hardware. SNAP is packaged into 256 custom designed chips assembled on four printed circuit boards. Each chip holds on an average 64 nodes and their associated pointers, thus SNAP can store a 16K node semantic network.

SNAP is a marker propagation architecture in which the movement of markers between cells is controlled by propagation rules. Various reasoning mechanisms are implemented with these marker propagation rules. SNAP's operation has been software simulated, and its performance compared to other knowledge processing systems. The results obtained, some of which are documented in this report, are very encouraging.

Contents

1 Introduction to Knowledge Processing 1

1.1 Main Issues in Knowledge Processing 1

1.2 Important Operations in Knowledge Processing 5

1.3 Architecture Requirements 7

2 Knowledge Representation on SNAP 9

3 SNAP Architecture 10

3.1 SNAP Array 11

3.2 Chip Architecture 13

3.3 SNAP Controller 19

4 SNAP Instructions 19

4.1 Instruction Set 19

4.2 Relations 25

4.3 Marker Propagation Rules 26

5 Reasoning on SNAP 27

5.1 Inheritance 27

5.2 Pattern matching 29

5.3 Classification 31

5.4 Numeric Processing 34

5.5 Graph 37

6 Simulation Results 38

6.1 Simulator 38

6.2 Results 39

7 Comparison with Other Knowledge Processing Systems 40

7.1 Spreading Activation 42
7.2 NETL 44
7.3 KL-ONE 44
7.4 Connection Machine 46

8 Conclusions **47**

1 Introduction to Knowledge Processing

In the SNAP project we wanted to design and implement a specialized, highly parallel architecture for knowledge representation and reasoning. We have selected *natural language understanding* as the main application domain. Our approach was to first identify key issues and operations for this problem domain, and then to map these requirements directly into hardware. This approach can provide performance advantages over the contrasting approaches of designing parallel languages for artificial intelligence or using commercially available multiprocessors.

The first SNAP design was done in 1983 [Moldovan 1983]. Some of its earlier applications were described in [Moldovan 1985]. Since then, considerable experience has been gained, and this report superseeds the eariler SNAP design.

In this section, some of the important issues in knowledge representation and reasoning are identified. They range from knowledge structuring and forms of reasoning, to speed and size of knowledge bases. Once these issues are identified, it is possible to find the basic operations used extensively in most knowledge processing systems. Any computer system aiming to achieve a significant performance improvement for this domain problem must be able to carry out knowledge-specific operations in hardware. The remarks in this introductory section constitute the background from where the SNAP architecture was derived.

1.1 Main Issues in Knowledge Processing

Knowledge Structuring

One of the most important issues in knowledge processing is how to structure the knowledge at hand. The organization of knowledge is often driven by two goals:

1. The desire to store knowledge in a compact manner, while also being able to derive as much implicit knowledge as possible, and
2. the desire to have fast access to information in an arbitrarily large knowledge base.

The human brain meets these goals. When we speak we say only enough for others to understand our intent. We are also able to recall and manipulate loosely related facts in a fraction of a second.

There is an agreement among a majority of researchers working in knowledge representation that *hierarchical structuring* of knowledge contributes towards both goals. Hierarchies imply some ordering. A main ordering criterion is generality, or level of abstraction; more general or abstract concepts are at the top of the hierarchy, while others more specific are placed lower in the hierarchy. The ordering relation is called the *subsumption* - or *is-a* relation. For example the concept `Person` subsumes the concept `Father`, which further subsumes concept the `Grandfather`. They can be written as `Person` > `Father` > `Grandfather`, where > represents the ordering relation *is-a*.

A primary advantage of hierarchical structuring is its efficient method of representation. This efficiency is derived from the fact that hierarchies provide a powerful mechanism for property inheritance. The properties of concepts may be derived based on the inheritance mechanism from the concepts above them in the hierarchy. For example, the properties of the concept `Person` such as two arms, two legs, two eyes and others need not be repeated for the concepts below, because the properties are inherited by all the subsumees. However, exceptions (one arm instead of two) or specific values (weight equals 160 pounds) may need to be specified for a particular person which is an instantiation of concept `Person`.

The speed advantage during retreival provided by hierarchical structuring is also derived from the ordering property. For example, it is easier to search a dictionary of alphabetized names than an unordered set.

Very Large Knowledge Bases

In order for an intelligent system to perform tasks such as natural language processing, its database must be populated with vast amounts of information. The construction of a huge, hierarchically structured knowledge base is not a trivial task. Their maintenance also raises serious computational problems. Constantly, new knowledge becomes available and needs to be integrated into the existing knowledge base. Inconsistencies between the new and the previous knowledge must be resolved, and this requires considerable processing.

Computational Effectiveness

Intelligent interactive systems must be capable of responding within a specified time frame. Machine processing time can often be compared to the time humans take to produce similar results. Many human activities such as recognizing objects, or understanding speech or text require only hundreds

of milliseconds. This should be the target for further intelligent systems. Today, computers are far from this target.

Consider, for example, activities such as understanding a sentence or recognizing objects in a certain context. It takes us only a fraction of a second to perform such functions. Considering that neurons, the basic elements of human brain, take milliseconds to fire, this implies only hundreds of such steps are needed.

There are also applications when we would like the machine to even outperform humans. Consider, for example, an AI system designed to help fighting pilots during a mission. Such systems must integrate a considerable amount of information and be able to assist pilots in making split second decisions.

Recognition

One of the most important forms of reasoning on a knowledge base is the recognition of concepts or situations on the knowledge base. The *recognition problem* may be described as: given a description of a set of properties, find a concept or a pattern of the concepts that best matches the description. Although similar to ordinary pattern matching, this problem is far more complex due to the fact that the properties of concepts are not available locally, and may have to be extracted via inheritance mechanisms. Moreover, the exact pattern commonly does not exist and the closest corresponding match must be determined.

Inheritance

Inheritance is another form of reasoning, the dual of recognition. The *inheritance problem* may be defined as: given a concept or a pattern of concepts in a hierarchical structure find the most likely properties inherited by that concept (or pattern) from the properties of other ancestor concepts. For example the input pattern may be "`Clyde is an elephant`". We would like the system to conclude rather rapidly that most likely `Clyde` is a large animal, has four legs, has a trunk and much more. Inheritance reasoning is the mechanism which locates properties attached to concepts within a certain "distance" specified by the reasoning mechanism. Often, conflicting information may be gathered by inheritance reasoning due to multiple inheritance channels. Resolving these conflicts may be quite difficult.

Handling Exceptions and Ambiguities

Exceptions occur in inheritance systems whenever a property applies to most members of a class but not to all. For example, most of the birds have

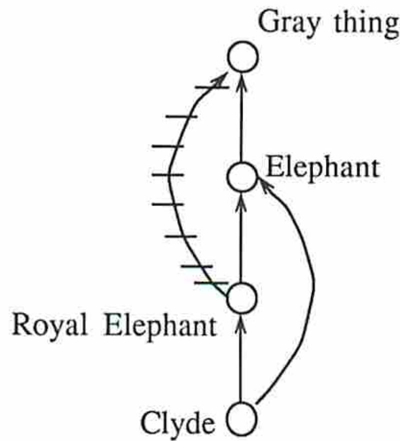


Figure 1: Exception

the ability to fly, but penguins do not fly. The problem with exceptions is that they lead to inconsistencies or conflicting information. Let us consider the following example, from [Fahlman 1979].

```
Elephants are gray.
Royal elephants are elephants.
Royal elephants are not gray.
Clyde is a royal elephant.
```

One possible interpretation may be: since Clyde is a royal elephant, and royal elephants are not gray, Clyde is not gray. On the other hand, another interpretation may be: Clyde is a royal elephant, royal elephant are elephants and elephants are gray, so Clyde is gray. This contradiction must be resolved by a mechanism built into the reasoning system. One technique to resolve the conflict is to keep track of the hierarchy, and let the information or property available to the nearest concept prevail over conflicting information. For example, we are inclined to believe that Clyde is not gray since he is closer to Royal elephant than to elephant.

This example introduces the notion of *distance* in the knowledge base. Its simplest definition is the number of semantic network links between two concepts. It can also be augmented by associating measures of belief to the concepts.

Another source of ambiguity is created by *multiple inheritances.* A concept may have several superconcepts or parents and it may inherit different and conflicting properties. Consider the following example:

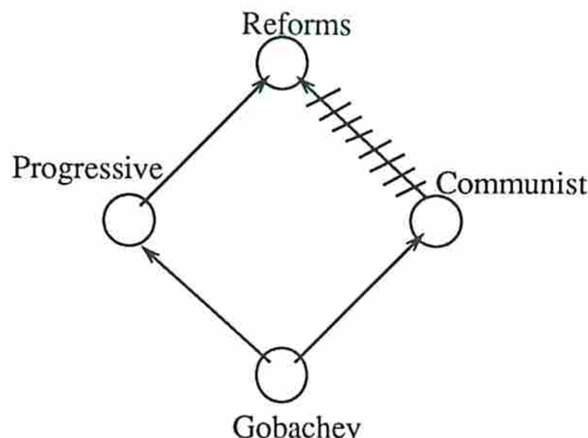


Figure 2: Multiple inheritance

```
Progressives want reforms.
Communists do not want reforms.
Gorbachev is a progressive.
Gorbachev is a communist.
```

Does `Gorbachev` want reforms or not? The problem here is because `Gorbachev` has two parent concepts, `Progressive` and `Communist`, which are not ordered. Note that the notion of distance used in exceptions, does not help here. However, there are ways to deal with multiple inheritances. One way is to simply enumerate all possible answers. Another way is to use additional relevant information to help resolve such conflicts. Yet, another way is to use a representation where concepts have associated to them a degree of belief. When reasoning is performed, beliefs propagate and a global hypothesis belief is generated which may serve for ordering hypotheses.

1.2 Important Operations in Knowledge Processing

In this section we identify some of the most frequently used operations in reasoning on knowledge bases. These operations constitute the building blocks for addressing the issues from the previous section.

Locate entities in knowledge bases that best match a set of features.

This operation is perhaps the most important operation carried out on knowledge bases because it is the "inner loop" operation used in reasoning schemes. The operation varies in complexity from a trivial word (or key)

search to locating the best match among properties dispersed through the knowledge base. An example of this latter case occurs when we try to find a situation similar to an existing problem, but with some modifications. Thus, features need to be sorted out and those which are most significant must play a larger role, while some less important features might be ignored. Such a requirement rules out fixed pattern matching techniques, hashing, or indexing schemes.

Find implicit properties of any concept in a knowledge base.

This operation refers to first identifying a particular concept, and then inferring properties or features which the concept contains implicitly, through the inheritance of properties from its ancestors. The main problem with such an operation is that the paths can bifurcate into multiple computation paths which increases the computational complexity.

Unification operation

This operation is derived from a reasoning technique called resolution. Unification produces a new pattern which will match whatever both input patterns would have match. This operation may be used, for example, to match an inference rule with a knowledge base to determine the applicability of that inference rule. It is also used to place a pattern within a context domain by unifying the pattern concepts with the rest of the knowledge base.

Set operations

Set operations such as intersection and union are essential for knowledge processing. Set intersection, for example, may be used to identify concepts which share two or more features, to identify features common to several concepts, and in general to restrict the members of a set by imposing a new condition. Sets can be represented in many ways. Simple ways to explicitly represent sets are listing members or using a set membership flag. Another more complex representation can be achieved implicitly, by generator-functions or predicates. Since we postulate large knowledge bases, it results that the operation described here is intersecting large sets. Clearly, we would like that this operation to be done in constant time, independent of the set cardinalities. Quite often, sequential AI algorithms avoid set intersections for the very reason that this operation is time consuming on sequential machines.

Global broadcasts

Broadcast operations are performed on knowledge bases to collect results, set initial conditions, and provide capability for a global controller to communicate with the networks. It is desired that parallel knowledge bases do not needlessly depend on a step-by-step operation of a central controller. We want the nodes to have some autonomy for deciding how to route messages and how to react to various conditions. While such local asynchronous processing is desired, there are, however, some operations which require simultaneous access to many nodes. These latter operations can be achieved via broadcasts from the central controller.

I/O access

As in any large computer system, I/O plays an important role in knowledge-base systems. It is necessary not only to initially construct the knowledge base, but to frequently update it with new information and retrieve results.

1.3 Architecture Requirements

Massive parallelism

A human's brain is massively parallel. Effective use of parallelism is essential for achieving increased computational effectiveness. More parallelism means more computing power, which in turn means more potential to perform the complex operations associated with intelligence. It is beneficial to have as many processors as concepts and to have a rich interconnection network to support a large number of relations between the concepts.

Mix processor and memory

There is a need for large amount of storage, and simultaneous processing of many data items. This is not possible with a single von Neumann computer or even a shared memory multiprocessor. It is appropriate to treat memory cells not only as storage, but also as active processing elements. This localizes the processing and, is compatible with a very large parallel network model, with simple processing elements. Thus the processing capability of such a computer is distributed across its memory.

Processing requirements

Each processing element must have its own processing capability, its own storage, and the ability to communicate with many other processors. Knowledge processing is not numerically intensive. Instead, operations such as quick access to data, associating a piece of information with another through

pointers, performing intersections over large sets and others become dominant features. Although, knowledge processing does not rely heavily on numeric processing, it is important that processors have the ability to perform some numeric instructions. One of the main reasons for this is that much effort has been done in developing probabilistic reasoning techniques which associate probabilities or measures of belief to concepts.

The processor granularity is related to the form of knowledge representation, and to the reasoning techniques used. The processor size is a tradeoff between the desires for numerous processors and numerous connections per processor. Eventually, when implemented, packaging and other design constraints tend to heavily influence processor granularity.

Communication

What makes knowledge processing networks distinct from other multiprocessors is their ability to support reasoning mechanisms. Knowledge processing is communication intensive and highly nondeterministic. The nature of knowledge is such that a slight perturbation of some concepts or relations sends waves of messages to numerous other destinations. The nondeterministic behavior derives from the requirement that the knowledge base adapts to new situations and moreover performs learning. This implies the need for interconnection networks with high bandwidths. In the brain, each neuron has on the order of 10^4 individual connections with other neurons. Present technology does not offer, by far, the ability to build so many dedicated links between processors. We must overcome this obstacle by cleverly controlling the flow of data on more modest, but realistic interconnection networks.

Traditional interconnection networks use centralized control to reconfigure, whereas here there is need for reconfiguration based on decisions at local nodes.

One possible communication mechanism which will be described in this report is *marker-propagation.* Markers are bit patterns which replace addresses. They are attached to data. Markers may satisfy many functions, and their identity may be coded in the bit patterns. The presence of some markers at a node may guide the utilization of information at that node. Markers may also be used to enable links, or to inhibit the passage of other markers. Markers may be used to color messages and thus represent different waves of activations that simultaneously advance through the network. Each such wave may be connected to a hypothesis, or part of a hypothesis.

Another use of markers is to construct temporary connections between different nodes in the network. For example, the simultenous presence of a marker on node A and node B may be regarded as a *virtual link* between

nodes A and B. If one wants to assign such a link specific properties, one can associate some node M with that link by assigning it the same marker.

The propagation of messages through the network must not be controlled by an outside controller, instead it should be guided by the nature of the messages via some propagation rules. Complex reasoning mechanisms can be implemented by controlling the movement of messages according to the hypothesis at hand.

Central controller

One of the most difficult aspects of knowledge processing networks is how to distribute control over the processing elements while still being able to make global decisions. The role of a central controller is to regulate and sometimes modify the node activities Also, it has the role of distributing the load throughout the network.

2 Knowledge Representation on SNAP

Semantic networks provide an efficent means to represent and process knowledge. The basic form of a semantic network is two nodes connected by some relation. Semantic networks can be stored in a parallel computer in a number of ways. The conventional methods are to store the two nodes as either a software data structure or a hardware processing element, with the connection between the two nodes provided by a named pointer.

Both approaches are direct mappings of the semantic network to the architecture. The problem with direct mapping, however, is that a typical semantic network can have hundreds or perhaps thousands of different relation types (`has-part`, `has-color`, `is-a`, `loves`, `hates`, etc). Thus, the architecture must be able to support the largest number of possible relations in a semantic network. This requires a large amount of memory to store the names of the pointers. In addition, a direct mapings can result in a loss of information. For example, deducing "`John does not hate Mary`" from "`John loves Mary`" is difficult, since no knowledge that `Love` is the opposite of `Hate` can be infered from storing semantic networks in this way.

For these reasons, we have chosen to represent information in SNAP differently. We felt that with 64 *basic relations* defined by the user (`is-a`, `role`, etc) SNAP could represent everything that the direct mapping technique could, without the associated problems. The 64 relations in SNAP are very primitive. In order to build meaningful relations like `Love` and `Has-Part`, *relation-nodes* are used. For example, representing the semantic

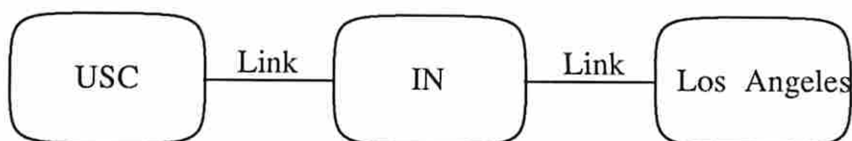


Figure 3: SNAP semantic network for "USC is in Los Angeles"

network for "`USC is in Los Angeles`" requires three nodes (see Figure 3). The node `IN` provides the means for linking the nodes `USC` and `Los Angeles`.

The disadvantage of representing knowledge in this way is that making the connection between `USC` and `Los Angeles` requires going through an intermediate node. However, having this intermediate node allows SNAP to represent *relationships between relations*. For example, the relation-node `Love` can have a relation emanating from it that indicates that it is the opposite of the relation-node `Hate`. This allows SNAP to conclude that "`John does not hate Mary`" if "`John loves Mary`".

A potential disadvantage is that many instances of relations like `IN` and `LOVE` can be present in a semantic network at one time. Distinguishing relationships can be difficult. For example, if the semantic network had both "`USC is in Los Angeles`" and "`Stanford is in Palo Alto`", keeping track of what `IN` means is not easy. SNAP solves this problem by creating multiple instances of relation-nodes. Thus in SNAP, the `IN` node for `USC` is placed in a different processor than the `IN` node for `Stanford`.

However, having multiple instances of a relation-node creates a uniqueness problem. In SNAP, this is taken care of by the *color* property. Each node in SNAP has both a *Node-ID* and a *color*. The Node-ID is the physical address of the node and is used to uniquely identify the node in question. For example, `USC` would have a pointer that points to the `IN` with the one Node-Id and `Stanford` would have a pointer that points to one with another Node-ID. The color property stores the type of node, in this case `IN`.

3 SNAP Architecture

As shown in Figure 4, SNAP consists of two parts: a *VLSI array* that performs the actual computations, and a *controller* which controls the operation of the SNAP array. The controller is connected to a SUN host computer which tasks SNAP with jobs.

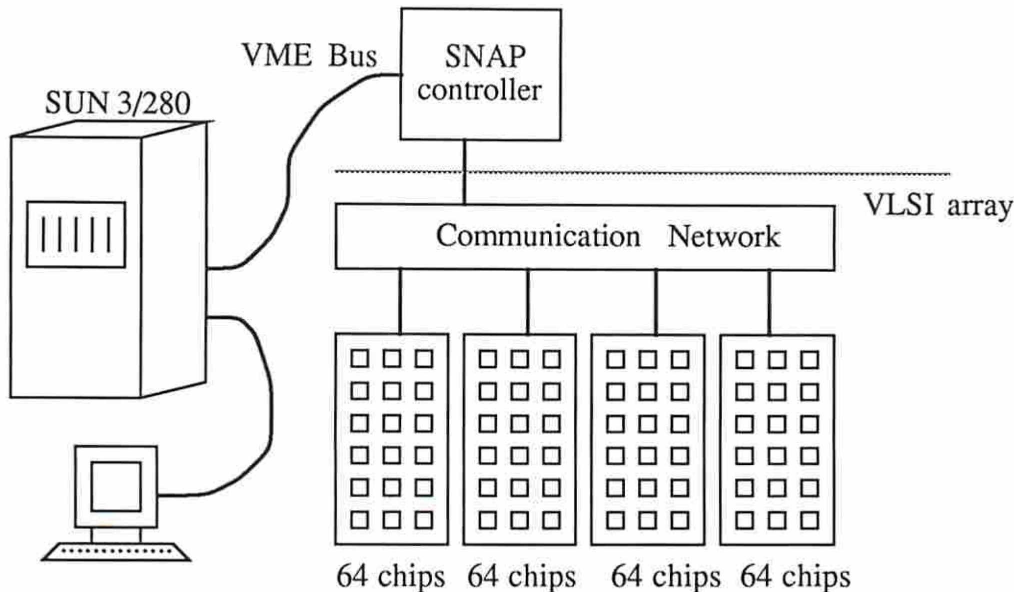


Figure 4: SNAP system

3.1 SNAP Array

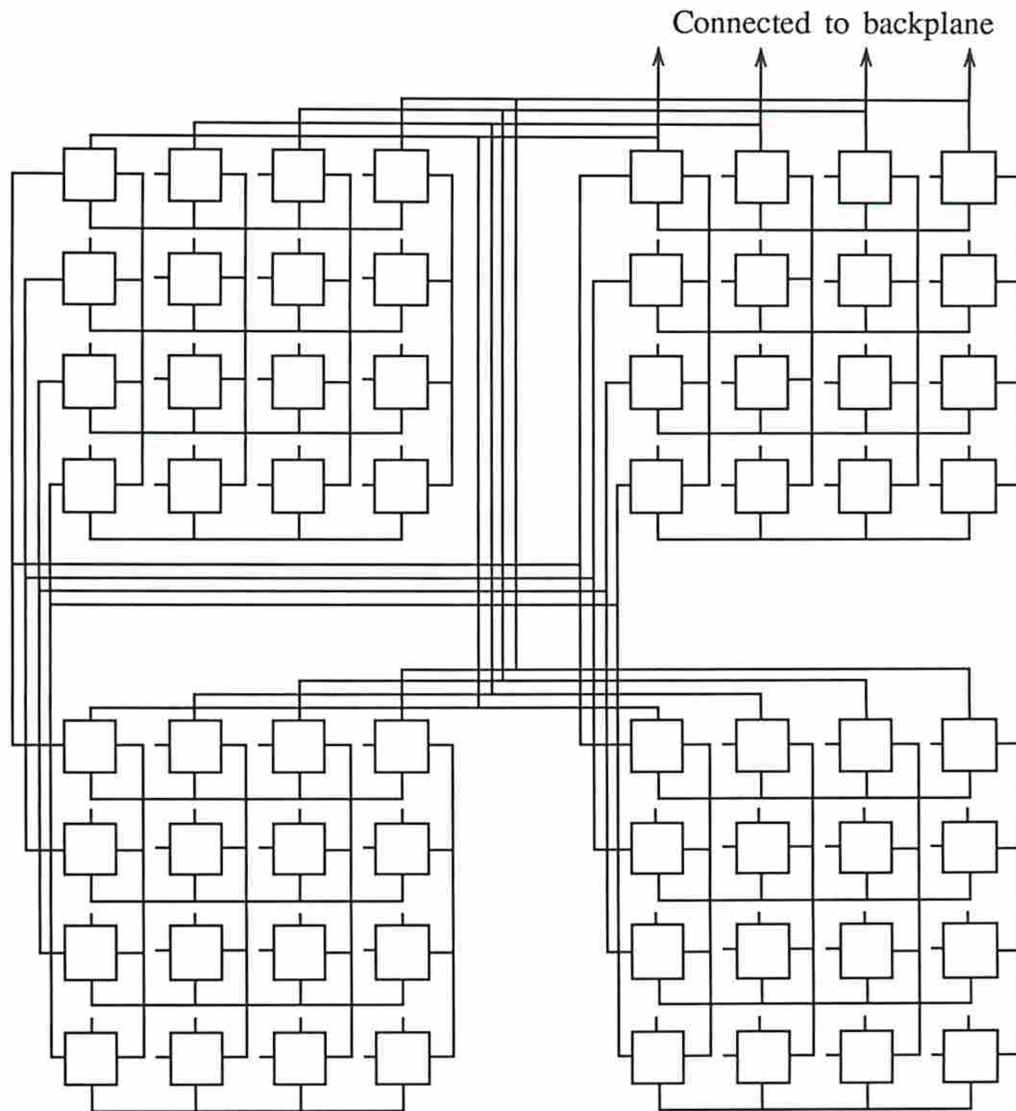
The SNAP array consists of 256 chips assembled on four 9U size printed circuit boards. Each of the chips contains 640 words of memory and 64 nodes. Thus, each node has reserved on the average 10 words.

The chips are connected to each other via a modified hypercube network (see figure 5). The chips are layed out into quadrants of 64. Each quadrant is on a separate board. Each chip has 4 data ports for communication with other chips. Assuming that the binary representation of the address of the processors is $(a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0)$, the configuration of the ports is :

Port #4 communicates with processors that vary in $(a_1\ a_0)$
Port #3 communicates with processors that vary in $(a_3\ a_2)$
Port #2 communicates with processors that vary in $(a_5\ a_4)$
Port #1 communicates with processors that vary in $(a_7\ a_6\ a_5\ a_4)$

Note that Port #1 is a superset of Port #2. Port #1 is used for communiation between quadrants (boards). Thus by putting 16 chips onto Port #1 instead of 4, the total number of buses between boards is reduced.

Typically, a message can travel on any of the four ports. For example, suppose chip (00 11 10 01) wants to send a message to chip (10 01 00 11).



Connections are replicated for all columns and rows.
There are 16 8-bit data buses to the backplane,
one for each chip in a 4×4 block.

Figure 5: SNAP Array

Using Port #1: we can send from chip (00 11 10 01) to chip (10 01 10 01)
Using Port #2: we can send from chip (00 11 10 01) to chip (00 01 10 01)
Using Port #3: we can send from chip (00 11 10 01) to chip (00 11 00 01)
Using Port #4: we can send from chip (00 11 10 01) to chip (00 11 10 11)

It takes a maximum of 4 (3 if Port #1 is used) chips to get from the source to the destination.

This interconnection network has been software simulated and compared with other types of networks (see [Lee 1989]). The simulator operates by introducing messages into the network and recording the time messages took to reach their destinations. The messages were placed into the network at frequencies varying from (1 new message)/clock to (32 new messages)/clock. Thirty-two thousand messages were sent and the results are shown Figure 6. The virtue of the modified hypercube is that the communication time has little variance. The average communication time was almost the same for lightly loaded networks as it as for the heaviest loads.

The controller communicates to all the chips via a global bus shared by all the cells in the array. The global instruction bus is used to transfer instructions from the controller to the cells for execution. The chips can send data back to the controller using Port #1.

3.2 Chip Architecture

The SNAP chip shown in Figure 7 consists of four units: a *processing unit* (PU), a *relation memory* (RM), a *marker control unit* (MU) and a *communications unit* (CU). All four of the units are connected via the *P-Bus*. The P-Bus is used for transfering instructions from the PU to the other units, and for receiving data from the other units to the PU. In addition, the RM and the MU are connected via the *M-Bus*, and the CU and the MU are connected via the *C-Bus*. Both, the M-Bus and the C-Bus are used during marker propagation.

Processing unit

The PU controls the operation of the SNAP chip. Its function is to decode the instructions received from the controller, and to provide the necessary commands to the other three units. The PU is microprogrammed and contains a microsequencer and a microprogram memory.

Relation memory

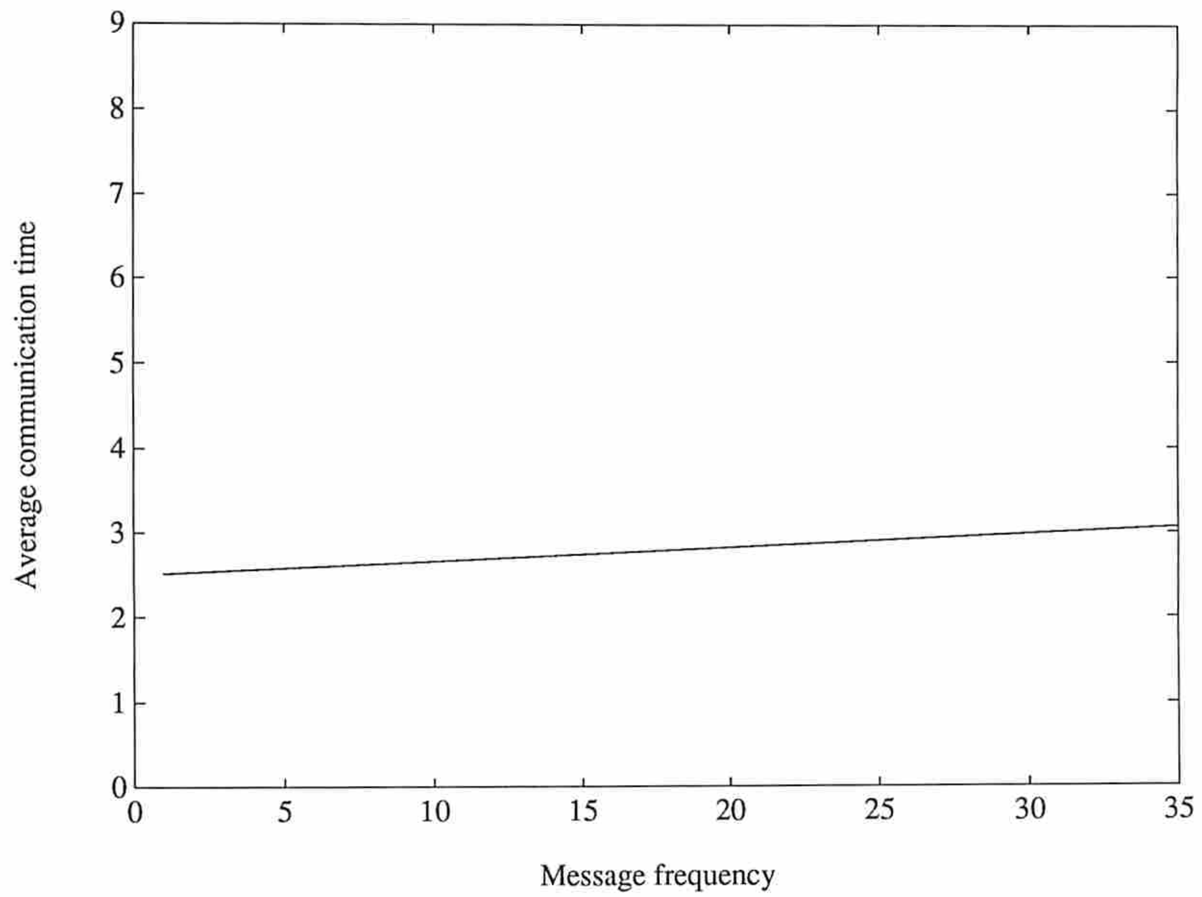


Figure 6: Performance of the SNAP network

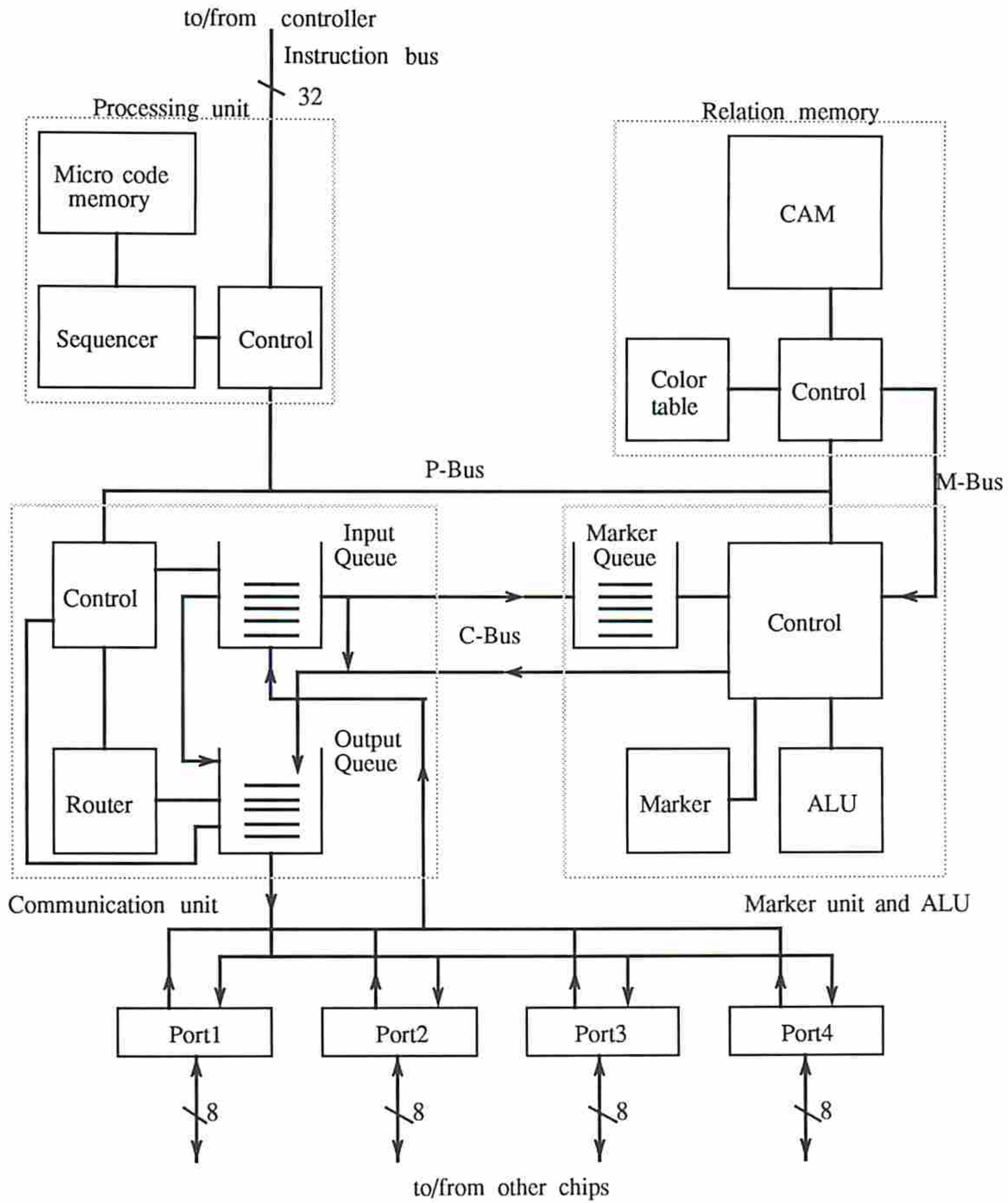


Figure 7: SNAP chip

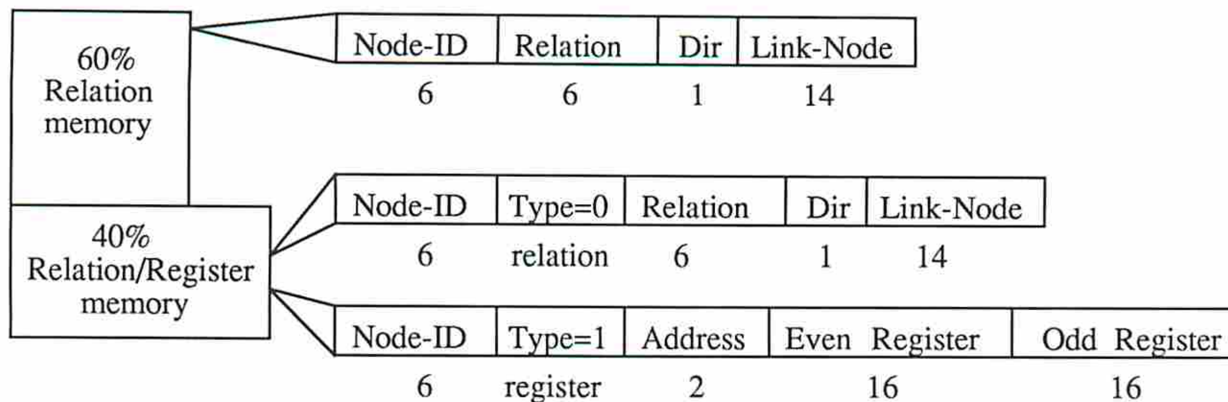


Figure 8: Relation Memory

The RM provides the main storage of the chip. It stores both the relation names coresponding to the relationships in a semantic network, and the registers for numeric processing. In order to provide parallelism within each chip, the RM is built around a content addressable memory (CAM). This CAM consists of 640 words, thus there are an average of ten words per semantic network cell. Six of the words can only hold relations, but the other four can hold either relations or registers.

Relation pointers are stored as four fields in the CAM (see Figure 8). Node-ID (6-bit) is the address of the cell to which the pointer is attached. The relation field (6-bit) is the name of the relation connecting this node with another node. Since relations in a semantic network are directed, a one bit direction field is used to indicate the direction of the relation. Link-node (14 bits) is the address of the node that the relation connects to.

The registers are packed as two registers/word. They are stored as four fields in the CAM. Node-ID (6-bit) is the address of the node that the registers belong to. Reg-Address (2-bit) contains the upper bits of the register address. Even-Register (16-bit) and Odd-Register (16-bit) are the actual data registers. There are a maximum of 8 registers per node. R1 is the Equal register, R2 is the Less-than register and R3 is the Greater-than register. GT, LT and EQ are used to denote the cardinality of a node. For example, to represent that a person has from 1 to 4 children, the LT register for `children` can be set to 5 and the GT register to 0. Likewise, to show that a person has 2 legs, the EQ register for `legs` is set to 2. R0, R4, R5, R6 and R7 are general purpose registers and can be used to store numbers or ALU flags. ALU flags are generated by the ALU and are used to tell whether the result of an operation is Zero, Negative, Postive, Overflow or has a Carry

out.

The allocation of words in the RM is not fixed. Some of the cells may have many relation pointers, while other cells in the chip may have just a few. Likewise, some cells may have many registers, while others have none.

Marker unit

Markers provide the inferencing capability of SNAP. Markers are assigned to certain semantic network nodes. These markers are then propagated to other nodes by way of messages. The way markers propagate is dictated by a set of *propagation rules.*

It is the job of the MU to control the propagation of these markers. The MU is divided into three parts: a *control unit* which processes the markers, an *associative memory* which stores the markers, and an *input queue* which stores the messages received by the CU.

Each node in a chip has 32 single-bit markers. When a message is received by a node, the marker it corresponds to is set. The marker is then propagated to other nodes using a set of propagation rules. After this, the user can ask for a list of all nodes that have a certain marker set. For example, sending a marker from the node `monkey` up its subsumption links would set markers in the nodes: `primate`, `mammal`, `animal`, and `thing`. Thus, if the user then asks for all the nodes with the marker set, he would get a list of all of the ancestors of `monkey`.

The marker unit also contains an ALU capable of doing add, subtract, multiply, and divide. The ALU can be driven by either regular SNAP instructions or by marker messages. Typical uses for the ALU under marker control are computing *semantic distance* and changing the *cardinality* of a node.

Communication unit

The CU interfaces the chip with the rest of SNAP. It provides for communication with other chips as well as with the controller. The CU is connected to four ports. The CU inspects incoming messages and determines if any messages are for the chip. If a message is for the chip, the CU sends the message to the marker unit for further processing. Otherwise, the CU directs the message to the another port.

The CU can also use Port#1 for sending data from the chip back to the controller.

Messages

Message are used to enable the nodes in a semantic network to communicate with each other. Since typical inferences require the passing of many messages, efficient message passing is a key goal of SNAP. Thus, messages are processed simultaneously in the background. This allows, the PU of the SNAP chip to be executing instructions, while the CU is sending messages and the MU is processing messages.

The messages that the SNAP array sends are marker-passing messages. SNAP has two types of messages: *regular marker messages* and *arithmetic marker messages.*

Regular markers simply set a marker at the destination node and propagate according to the propagation rule. They are sent as five 8-bit messages:

Byte1: Destination-chip (8-bit)
Byte2: Msg-type = 0, Node-in-destination-chip (6-bit)
Byte3: Marker (5-bit), Rule (3-bit)
Byte4: Relation1 (7-bit)
Byte5: Relation2 (7-bit).

Arithmetic markers are identical to regular markers except they also perform an arithmetic operation at the destination node. Arithmetic marker are sent as eight 8-bit messages:

Byte1: Destination-chip (8-bit)
Byte2: Msg-type = 1, Node-in-destination-chip (6-bit)
Byte3: Marker (5-bit), ALU-operation (2-bit)
Byte4: Upper data byte (8-bit)
Byte5: Lower data byte (8-bit)
Byte6: Register1 (3-bit), register2 (3-bit), rule (2-bit)
Byte7: Rule (1-bit), Relation1 (7-bit)
Byte8: Relation2 (7-bit).

The overlapping of message transfers with the broadcasting and execution of nonconflicting instructions is an important characteristic of SNAP.

Since messages are handled in the background, the SNAP controller does not know when a marker has finished propagating. However, each chip sends a status signal to the controller indicating whether it has any more messages to process. Thus, the controller can monitor the state of these status signals and determine if any more messages are circulating around the network. If an instruction requires that all of the markers have finished propagating, the

controller must wait until all of the status signals become inactive before sending that instruction to the chips.

3.3 SNAP Controller

The Controller interfaces SNAP with the host computer (a SUN 3/280) and controls the operation of the SNAP array. The controller is connected to the SUN via a VME Bus. SNAP programs are compiled into SNAP primitives on the SUN and downloaded onto the controller over the VME Bus. When the program has completed running, processed results are sent back over the VME Bus for viewing by the user on the SUN.

The controller manages the operation of the SNAP array by providing program sequencing, data collection, and node management capabilities. Program sequencing involves sending instructions to the array, determining the completion of an instruction, and generating the address of the next instruction. Data collection involves monitoring the SNAP array for results, and binding the results to variables. In node management, the controller allocates new nodes of the semantic network to unused cells of the array. In addition, when all the cells are in use, the controller invokes garbage collection routines to see if any cells can be freed up.

4 SNAP Instructions

4.1 Instruction Set

SNAP instructions are broadcast by the central controller to all chips for execution on-chip. They have the general format:

$$< opcode > < arguments >$$

The *opcode* specifies the type of instruction, and the *arguments* may be either node addresses, node colors, relations, propagation rules or markers. Depending upon each instruction, some arguments may be conditions and some may be actions. In order to easily distinguish between the two, *conditions* are marked with [ ], while *actions* are marked <>. Markers are represented by the symbol # followed by the marker number. Thus, #3 means marker number 3. In addition, "Don't Care" is denoted by the symbol %. Figure 9 shows the instructions supported by SNAP.

Instruction	Function
SEARCH SEARCH-COLOR	Search
SET-COLOR CREATE DELETE LOAD	Node maintanence
AND OR NOT	Logical
REG-ADD REG-SUB REG-MULT REG-DIVIDE TEST	Arithmetic
MARKER CLEAR-MARKER STOP-MARKER CLEAR-STOP-MARKER EQUATE CLEAR-EQUATE	Marker
MARKER-ADD MARKER-SUB MARKER-DIVIDE MARKER-MULT MARKER-MIN MARKER-MAX MARKER MIN+	Marker-arithmetic
COLLECT COLLECT-RELATON	Data retreival.

Figure 9: Table of mneumonics

The operations performed by each instruction are described in the following listing:

- SEARCH [*node*], $< marker - number >$

  The SEARCH function causes the cell with [$node - ID$] to set $< marker - number >$ For example, (SEARCH (65 4), #1) means to set marker#1 in chip 65, cell 4.

- SEARCH-COLOR [$node - color$], [*relation*], $< marker - number >$

  SEARCH-COLOR provides two pattern strings to the array; the first is the color of the cell memory, and the second is a relation pointer. Every SNAP chip checks these two conditions and in the cells where the match is successful, the marker specified in the third argument is set. For example, (SEARCH-COLOR A, R1, #1) means to search cells with color A and relation R1, and set marker #1 in those cells.

- SET-COLOR [$Node - ID$], $< node - color >$

  SET-COLOR modifies the COLOR of Node-ID For example, (SET-COLOR (65, 4), `IN`) will modify the color of chips 65, cell 4, to `IN`.

- CREATE $< node1 >$, $< relation >$, $< node2 >$

  CREATE is function that creates new relations between nodes For example, (CREATE A, R, B) inserts a new relation R from node A to node B. If either A or B is new, the controller assigns them to new nodes.

  Initially, the array is loaded with nodes and links by the controller. This allocation procedure is built upon the function CREATE. Conceptually, this procedure employs as many CREATEs as the number of links in the semantic network.

- DELETE $< node1 >$, $< relation >$, $< node2 >$

  The DELETE function has the opposite effect of CREATE. It deletes a pointer relation between a pair of nodes. For example, (DELETE A, R, B) deletes the relation R linking node A to B. If either node becomes isolated after the DELETE, then the node can be permanently deleted when the controller invokes garbage collection.

- LOAD [*marker*], [*register*], [*data*]

  The LOAD instruction causes all cells with [marker] to load [data] into [register]. For example, (LOAD #1,R5,'9000') means that all nodes with marker #1 set will load '9000' into register R5.

- AND $[marker1]$, $[marker2]$, $< marker3 >$

  The AND function performs a logical AND of the first two markers. The ¡marker3¿ is set or reset based on the result of the AND. For example, (AND #1, #2, #3) will set marker #3 in those nodes where both markers #1 and marker #2 are set.

- OR $[marker1]$, $[marker2]$, $< marker3 >$

  The OR function performs a logical OR of the first two arguments. <marker#3> is set or reset based on the result of the OR. For example, (OR #1, #2, #3) will set marker #3 for those nodes where either marker #1 or marker #2 is set.

- NOT $[marker1]$ $< marker2 >$

  The NOT function negates the status of [marker#1] and places the result in <marker2 >. For example, (NOT #1, #4) means that if marker #1 is set for a node, then marker #4 becomes reset. Otherwise, marker #4 is set.

- MARKER $[marker1]$, $< marker2 >$, $< propagationrule >$ $(< relation1 >$ $, < relation2 >)$

  The marker instruction introduces a marker into the network. All nodes with [marker1] will begin propagating <marker2> according to the <propagation rule> The originating node does not set <marker2>. For example, (MARKER #1, #2, SEQ(LINK, ROLE) causes all nodes with marker #1 set to propagate marker #2 once along the link relation of the affected nodes, and then once through the Role relation. (see section 4.3).

- COLLECT $[marker]$, $< variable >$

  COLLECT is a content-addressable read. The controller collects the NODE-IDs and COLORs of the nodes that have [marker#] set. For example, (COLLECT #1, X) gets all the NODE-IDs and COLORs of nodes which have marker#1 set and binds them to variable X.

- COLLECT-RELATION $[marker\#]$, $< variable >$

  The function COLLECT-RELATION is the same as COLLECT except that it returns the contents of all relation names in the nodes which have [marker#] set. For example, (COLLECT-RELATION #1, Y) gets all the relation names in nodes which have marker #1 set, and binds them to Y.

- STOP-MARKER [*marker1*], [*marker2*], $< marker3 >$

  The STOP-MARKER instruction causes a stop <marker3> command to be placed on all cells with [marker1] and [marker2] set. Thus, these cells will not allow <marker3> to propagate. For example, (STOP-MARKER #1, #2, #3) will cause all cells with #1 and #2 set to not propagate #3.

- CLEAR-MARKER [*marker1*], [*marker2*], $< marker3 >$

  With this instruction, the cells with [marker1] and [marker2] set will clear <marker#3>. For example, (CLEAR-MARKER #1, #2, #3) will cause all cells with #1 and #2 set to clear #3.

- CLEAR-STOP-MARKER [*marker1*], [*marker2*], $< marker3 >$

  This instruction clears the STOP-MARKER at <marker3> for all cells with [marker1] and [marker2] set. For example, (CLEAR-STOP-MARKER #1, #2, #3) will cause all cells with #1 and #2 set to clear their STOP-MARKER #3.

- EQUATE $< relation1 >$, $< relation2 >$

  The EQUATE instruction causes the SNAP to treat <relation1> as if it were <relation2> during marker propagation. For example, (EQUATE R1, R2) will allow any marker that propagates along R2 to also propagate along R1.

- CLEAR-EQUATE $< relation1 >$, $< relation2 >$

  The CLEAR-EQUATE instruction causes SNAP to forget about the EQUATE between <relation1> and <relation2>.

- READ [*marker*], [*register*]

  The READ instruction causes all cells with [marker] to output [register]. For example, (READ #1,R5) means that all nodes with marker #1 set will output R5 to the controller.

- REG-ADD [*marker*], [*register1*], [*register2*], $< flag - reg >$

  The REG-ADD instruction causes all cells with [marker1] to ADD the data in [register1] and [register2] and store the result in <register1>. The ALU flags (Positive, Negative, Zero, Overflow, and Carry Out) are stored in the optional <flag-reg>. For example, (REG-ADD #1,R4,R5,R6) means that all nodes with marker #1 set, will add R4 and R5 and store the result in R4. The ALU flags go into R6.

- REG-SUB $[marker]$, $[register1]$, $[register2]$, $< flag - reg >$

  The REG-SUB instruction is identical to REG-ADD except that a subtract is done instead of an add.

- REG-MULT $[marker]$, $[register1]$, $[register2]$, $< flag - reg >$

  The REG-MULT instruction is identical to REG-ADD except that a multiply is done instead of an add.

- REG-DIVIDE $[marker]$, $[register1]$, $[register2]$, $< flag - reg >$

  The REG-DIVIDE instruction is identical to REG-ADD except that a divide is done instead of an add.

- TEST $[marker1]$, $[flag - reg]$, $[P|N|Z|OV|CO]$, $< marker2 >$

  The TEST instruction causes all cells with [marker1] to check to see if the result generated by an ALU instruction is either Positive, Negative, Zero, Overflow, or has a Carry Out. If the condition to be check is TRUE then <marker2> is set.

  For example, (TEST #1,R6,Z,#2) means that all nodes with marker #1 set will check to see if the zero flag in R6 is set. If the condition is true then marker #2 is set.

- MARKER-ADD $[marker1]$, $[register1]$, $< register2 >$, $< marker2 >$, $< prop - rule >$ ($< relation1 >$, $< relation2 >$)

  The MARKER-ADD instruction causes all cells with [marker1] to send the data in [register1] to other nodes according to <prop-rule> <relation1> and <relation2>. When the data gets to a destination node, the data is ADDed to the data in the node's <register2>. The result is stored in [register2]. <marker2> is then set in the destination node.

  For example, (MARKER-ADD #1,R4,R5,#2,SPREAD(ROLE,LINK)) means that all nodes with marker #1 set will send the data in R4 to all nodes connected to it by ROLE or LINK relations. When the message gets to one of these nodes, the data is added to the data in the nodes R5 register and the result is stored in R4. Marker #2 is then set, and this node tries to propagate the message to its ROLE and LINK neighboors.

- MARKER-SUB $[marker1]$, $[register1]$, $< register2 >$, $< marker2 >$, $< prop - rule >$ ($< relation1 >$, $< relation2 >$)

  MARKER-SUB is identical to MARKER-ADD except that a subtract is done at the destination node instead of an ADD.

- MARKER-MULT $[marker1], [register1], < register2 >, < marker2 >, < prop - rule > (< relation1 >, < relation2 >)$

  MARKER-MULT is identical to MARKER-ADD except that a multiply is done at the destination node instead of an ADD.

- MARKER-DIVIDE $[marker1], [register1], < register2 >< marker2 >, < prop - rule > (< relation1 >, < relation2 >)$

  MARKER-DIVIDE is identical to MARKER-ADD except that a divide is done at the destination node instead of an ADD.

- MARKER-MIN $[marker1], [register1], < register2 >, < marker2 >, < prop - rule > (< relation1 >, < relation2 >)$

  MARKER-MIN is identical to MARKER-ADD except that a minimum of the two numbers is done at the destination node instead of an ADD.

- MARKER-MAX $[marker1], [register1], < register2 >, < marker2 >, < prop - rule > (< relation1 >, < relation2 >)$

  MARKER-MAX is identical to MARKER-MIN except that a maximum of the two numbers is done at the destination node instead of an minimum.

- MARKER-MIN+ $[marker1], [register1], < register2 >, < marker2 >, < prop - rule > (< relation1 >, < relation2 >)$

  MARKER-MIN+ is identical to MARKER-MIN except that a 1 is added to [register1] before it is sent to the destination node. In addition, a minimum is performed even if <marker2> is already set.

  For example, (MARKER-MIN+ #1,R4,R5,#2,SPREAD(ROLE,LINK)) means that all nodes with marker #1 set will add 1 to the data in R4 and send it to all nodes connected by either ROLE or LINK relations. When the message gets to a destination node, a minimum is performed on the data and the result stored in R5. Marker #2 is set if it is not already. The MARKER-MIN+ is propagated until the incoming data is no longer less than the destination data.

4.2 Relations

Semantic networks in SNAP are built upon relations between nodes. SNAP supports 64 *primitive relations.* Primitive relations differ from regular relations as follows: primitive relations are pointers between semantic network nodes and stored as relations in the RM, where as other relations called *node relations* are stored as separate nodes. The primitive relations are user

definable. However, the following primitive relation conventions have adopted:

1. Superconcept: a relation between a supertype and its subsumee.
2. Individual: a relation between a type and a individual.
3. Split: a relation to seperate exclusive types.
4. Cancel: the relation to be used when an exception exists in the role of a special type (or individual) under a supertype.
5. Generic: the relation to group individuals with same characteristics.
6. Role: the relation to show the inherint properties of a type (or individual). For example, in sentence "`Animal has hair`", there is a role primitive relation between node `Animal` and node `Has-part`.
7. Link: similar to role except it is for non-inherent properties such as love, hate, etc.

4.3 Marker Propagation Rules

The propagation rules in SNAP govern how markers are passed. This is one of the basic features of this architecture which together with the instruction set makes it suitable for knowledge processing. Markers from one node are passed to other nodes via relations. In addition, SNAP has the capability that enables markers to travel through several relation types at the same time. These multi-relation propagation rules enables SNAP to create *virtual links*.

A *virtual link* is a property inheritance mechanism where a node inherits a property without a connection between the node and the property. The simplest virtual link involes inheritance from a node ancestor. For example, using the semantic network in Figure 10, `Clyde` inherits `Trunk` over a virtual link since `Clyde` and `Trunk` are not directly connected. More complex virtual links result from the use of the transitivity property. For example, the property `in` is transitive, thus if "`USC is in Los Angeles`", "`Los Angeles is in California`" and "`California is in United States`" a virtual link enables SNAP to infer that "`USC is in the United States`".

The propagation rules have the format of Rule, Relation1, Relation2, where Relation1 and Relation2 are the relations that rule affects. The following propagation rules have been defined for SNAP:

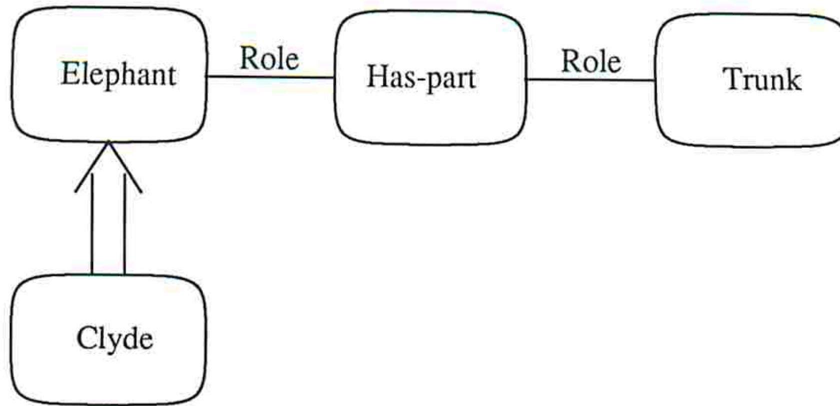


Figure 10: A simple virtual link

1. SEQ(R1, R2): the SEQUENCE propagation rule allows the marker to propagate through R1 once, then to R2 once.
2. SPREAD(R1, R2): the SPREAD propagation rule allows the marker to traverse through a chain of R1 links. For each cell in the R1 path, if there exist any R2, the marker switches to R2 link and continues to propagate until the end of the R2 link.
3. COMB(R1, R2): the COMBine propagation rule allows the marker to propagate to all R1 R2 links without limitation.
4. END-SPREAD(R1, R2): This propagation rule is the same as SPREAD except that it marks only the last cells in the paths.
5. END-COMB(R1, R2): This propagation rule is the same as COMB except that it marks only the last cells in the paths.

5 Reasoning on SNAP

5.1 Inheritance

The *inheritance problem* may be defined as: given a concept or pattern of concepts in a hierarchical structure, find the most likely properties inherited by that concept from the properties of other concepts.

Most queries are based on establishing an inheritance link between two nodes. However, many times the inheritance is not obvious, yet given the

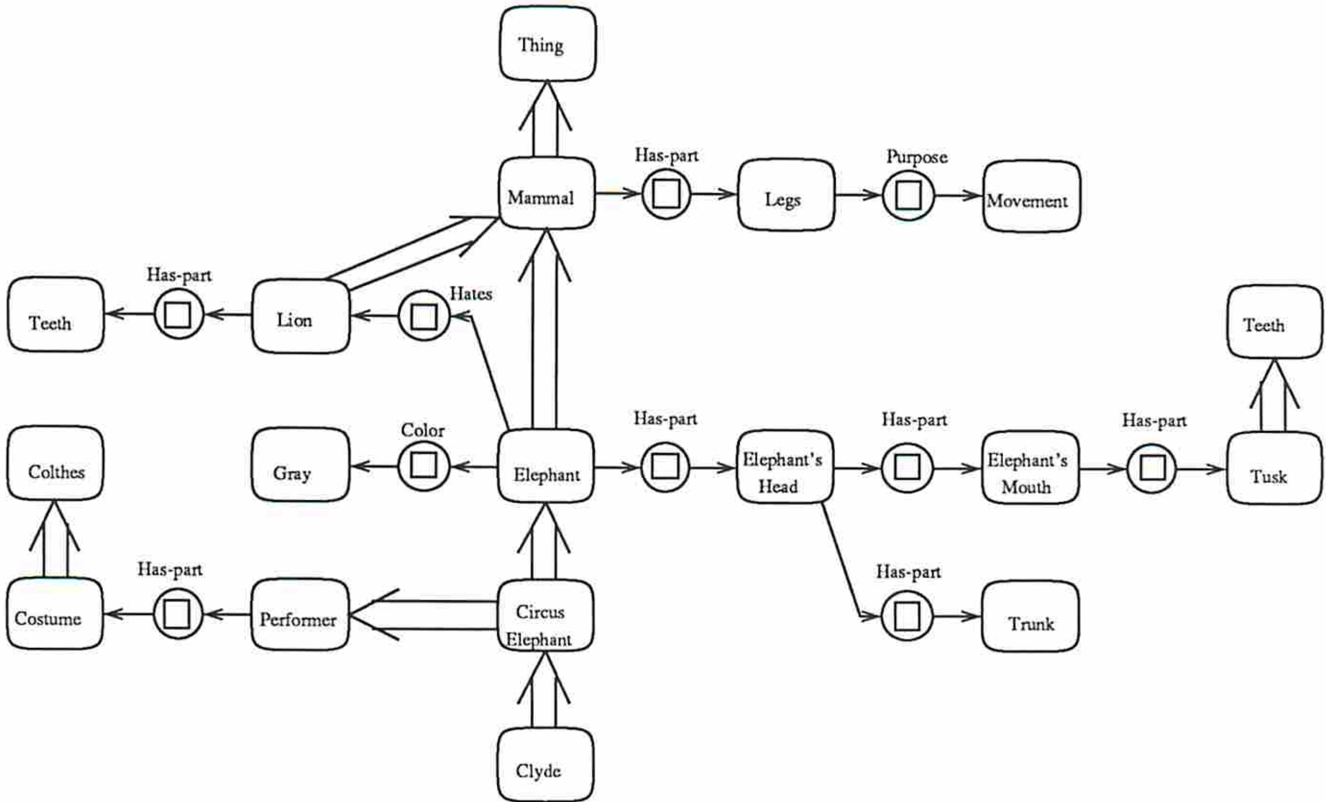


Figure 11: Semantic Network for Clyde

facts a human can easily see the connection. In the following example, we show how a non-obvious query is processed in SNAP.

Clyde the Circus Elephant

Clyde is one of the most famous figures in AI literature. Our example on inheritance includes Clyde and the semantic network in Figure 11. With this semantic network, we process the following query on SNAP:

"Does Clyde have teeth?"

The SNAP program to answer this query is very simple. It relies on the guiding of markers toward the destination (TEETH). Only valid nodes are allowed to propagate markers. Thus, markers cannot travel from ELEPHANT to HATES to TIGER to HAS-PART to TEETH. The program is shown below:

1. SEARCH-COLOR R-NODES % % #0

	this command sets marker #0 in all relation-nodes in the network (Has-Part, Has-Color, etc.).
2. STOP-MARKER #0 % %	this command prevents the propagation of any marker from all of the relation-nodes.
3. SEARCH-COLOR R-NODES HAS-PART % #1	
	this command sets marker #1 in all Has-Part relation nodes.
4. CLEAR-STOP-MARKER #1 % %	
	this command negates the previous STOP-MARKER instruction for all Has-Part relation nodes. Thus, only the Has-Part relation nodes will be able to propagate any markers. Regular nodes like Clyde never received a STOP-MARKER, and so continue to be able to propagate markers.
5. SEARCH CLYDE #2	sets marker #2 in node Clyde
6. SEARCH TEETH #3	sets marker #3 in node Teeth
7. MARKER #2 #2 COMB(SUPERCONCEPT F-ROLE)	
	this commmand causes Marker #2 to be propagated from CLYDE. The marker travels along all SUPERCONCEPT and F-ROLE links it encounters. F-ROLE is ROLE relation leaving the node.
8. WAIT-COMM-END	wait until markers finish propagating.
9. AND #2 #3 #4	
10.COLLECT #4	If the node TEETH gets marker #2, then marker #4 is set. Otherwise, marker #4 gets reset. If we are able to collect a node with marker #4 set, then the answer to query is YES. Otherwise, the answer is NO.

Thus, SNAP is able to respond to the query using just ten instructions.

5.2 Pattern matching

A marker-passing architecture can emulate a pattern matcher, but not vice versa. Consider that we want to search a knowledge as in Figure 12 (a) for a pattern

```
A --(R1)--> X1 --(R2)--> X2 <--(R3)-- B
```

Where `A` and `B` are name of nodes, `R1`, `R2` and `R3` are name of relations, `X1` and `X2` are variables. All nodes of the knowledge base matching this pattern will be bound to these variables. To search for this pattern, the following sequence of instructions need to be processed:

(a) Database

(b) First Pass:

(c) Second Pass

Figure 12: Search procedure

1	S-SEARCH R1 % 1	marker R1
2	SEARCH R2 % 1	mark R2
3	SEARCH R3 % 1	mark R3
4	MARKER 1 1 SPREAD(F-LINK)	propagate to all successor of R1,R2,R3 F-LINK is link relation leaving the node
5	SEARCH B % 2	mark A
6	SEARCH A % 3	mark B
7	SEARCH-COLOR R-NODES % # 0	mark all relation nodes
8	COMM-END	wait until propagation end
9	STOP-MARKER 0 % %	set all relation node stop marker propagation
10	CLEAR-STOP-MARKER 1 % %	allow only R1, R2, R3 propagate markers
11	MARKER 2 2 SEQ(F-ROLE)	propagate marker one step forward from B F-ROLE is a ROLE relation leaving the node.
12	MARKER 3 3 SEQ(F-ROLE)	propagate marker one step forward from A
13	COMM-END	wait until propagation end
14	MARKER 3 4 SEQ(F-ROLE)	propagate marker one step forward from A
15	COMM-END	wait until propagation end
16	AND 2 4 5	E should has both marker2 and marker4
17	COLLECT 5	get E, if nil then no solution.
18	MARKER 5 5 SEQ(R-ROLE)	if E exist, propagate backward for D R-ROLE is a ROLE relation leaving the node.
19	COMM-END	wait until propagation end
20	AND 5 3 6	D must have both marker3 and marker5
21	COLLECT 6	if nil then no solution or it is variable D

5.3 Classification

Consider the semantic network from Figure 13. This network includes a set of concepts (`person`, `parent`, `grandparent`, `mammal`, `dog` and `date`) and a set of roles (`child`, `birthday`, `pet`). Each concept denotes a set of objects and each role denotes a property that must be true for each concept to which that role is associated. The concepts have a hierarchical order, for example concept `mammal` includes concepts `person` and `dog`. A superconcept link connects a simpler concept to a more general one. We say that a more general concept subsumes a simpler concept. One inference problem is to find new subsumtion relations between the concepts of a given network and then to modify the structure of the network. For instance we want to determine if `parent` subsumes `grandparent`, and if it is true, connect `grandfather` to `parent` with a superconcept link.

According to [Lipkis 1983], concept `A` subsumes concept `B` if only if:

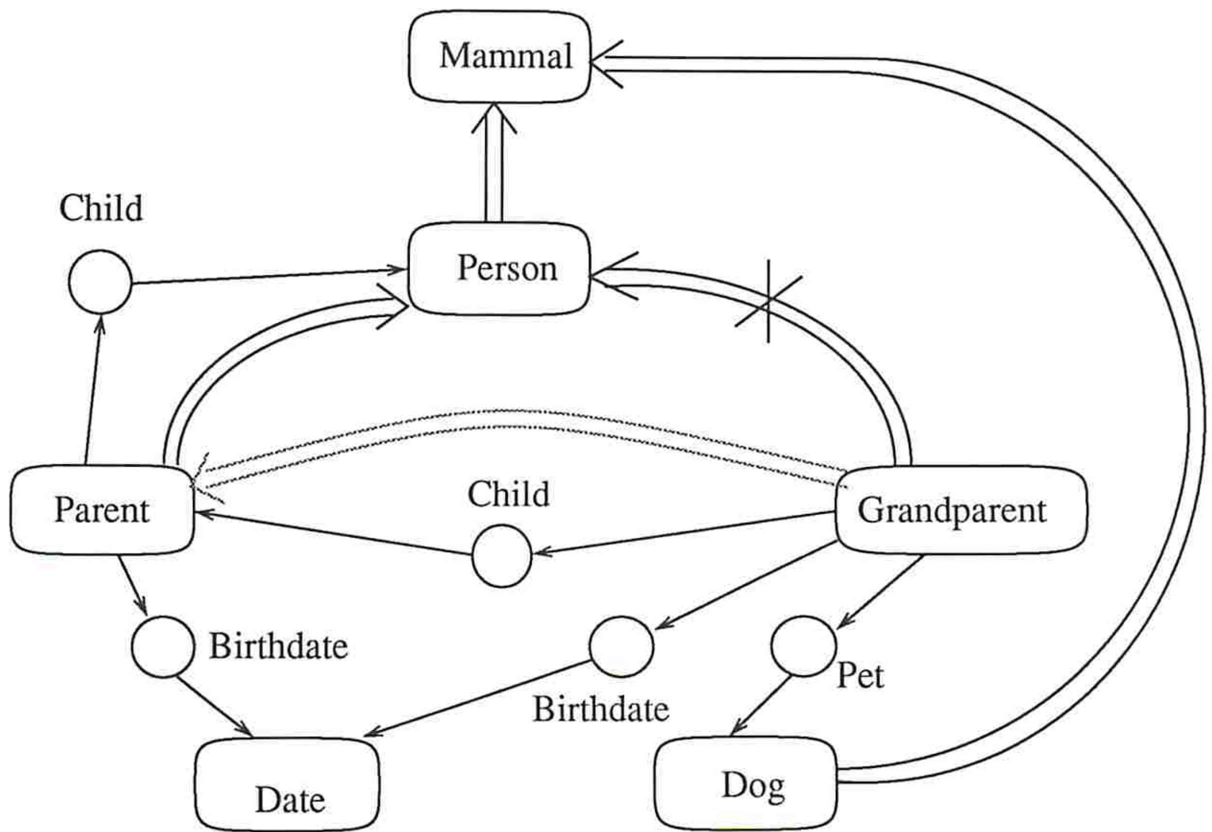


Figure 13: Classification on a semantic network

1. All primitive concepts that subsume `A` also subsume `B`. (Here `A` is `parent` and `B` is `grandparent`.)

2. For each roleset of `A`, some roleset of `B` denotes the same relation. (Roleset of `A` includes `birthdate` and `child` and is a subset of roleset of `B`.)

3. The value description of `A`'s roleset subsumes that of `B`'s (For the `birthdate` roleset, both `parent` and `grandparent` have the same value description namely `date`. For `child`, `parent`'s value description subsumes `grandparent`'s, namely `person` subsumes `parent`.)

This algorithm maps into the following SNAP primitive functions.

Condition I		
1	SEARCH GRANDPARENT % 1	Mark `grandparent`
2	SEARCH PARENT % 2	Mark `Parent`
3	MARKER 1 3 SEQ(SUB)	Mark all subsumers of `grandparent`
4	MARKER 2 4 SEQ(SUB))	Mark all subsumers of `parent`
5	COMM-END	Wait until propagation end
6	AND-MARKER 1 2 3	Get the intersection part of two subsumer sets
7	CLEAR-MARKER 3 %	Unmark the intersection part
8	OR-MARKER 1 2 4	Check if subsumers of `A` equal that of `B`
9	COLLECT 4	if nil than condition 1 valid
Condition II		
10	MARKER 2 5 SEQ(F-ROLE)	Mark roleset relations of `parent`
11	COLLECT 5 X	Get the colors of `parent`'s roleset from X we get `child` and `birthday`.
12	SEARCH-COLOR CHILD % 6	Mark the roleset relation nodes.
13	SEARCH-COLOR BIRTHDAY % 7	
14	MARKER 6 6 SEQ(R-ROLE)	Mark intersection between
15	MARKER 7 7 SEQ(R-ROLE)	`child` and `birthday`
16	COMM-END	
17	AND 6 7 8	Get intersection
18	AND 2 8 9	Check if `grandparent` is in the intersection
19	COLLECT 9	If not nil, then condition II valid.
Condition III		
20	MARKER 3 12 SEQ(F-ROLE)	Mark roleset of `grandparent`
21	MARKER 4 11 SEQ(F-ROLE)	Mark roleset of `parent`
22	COMM-END	
23	AND-MARKER 12 % 9	
24	MARKER 9 10 SPREAD(SUB)	Check if all `parent`'s roles subsume `grandparent`'s roles
25	AND-MARKER 11 % 12	

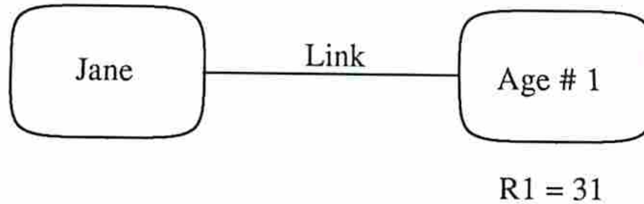


Figure 14: Representation of `Jane is 31`

26 COMM-END
27 CLEAR-MARKER 10 12
28 COLLECT 12 — if nil then value of `parent`'s roleset subsume `grandparent`'s

5.4 Numeric Processing

Some aspects of knowledge processing deal with numbers. Most frequently numbers are used to express values, cardinality of sets, measures of beliefs, and others. It is important, therefore, for SNAP to be able to process numbers efficiently. In this section, we present two examples where numbers play an important role and demonstrate how they are handled in SNAP.

The Age Problem

Given the following facts:

```
Jane is 31          David is 11
Phil is 7           George is 24
Sam is 43           Sue is 14
Mary is 5           Diane is 17
John is 20          Debra is 49
Carol is 12         Alice is 23
```

Answer the following query:

Who is an adult (i.e. age ≥ 18)?

In SNAP, the knowledge of each person's age is represented by two nodes (see Figure 14). The first node stores the person and the second node stores his age. Thus, if we identify those AGE nodes with a value greater than 17,

and then propagate a marker to the person the age belongs to, we can find all of the adults.

The SNAP program therefore is:

```
1. SEARCH-COLOR AGE,#1              ; select all AGE nodes
2. LOAD #1,R7,17                    ; load 17 into R7 of all AGE nodes
3. REG-SUB #1,R7,R1,R6              ; in all AGE nodes, R7=R7-R1; R6 = ALU flags
4. TEST #1,R6,N,#2                  ; set marker #2 in all AGE nodes GT 17
5. MARKER #2,%,#3,SEQ(LINK)         ; mark all persons older than 17 using marker #3
6. WAIT-COMM-END
7. COLLECT-COLOR #3                 ; get all adults
```

Most Children Problem

Given the following facts:

```
John, Karen, Diane, Sue and Matt live in Daywood.
John's children are Sam and Mary.
Karen's children are David, Jim, and Phil.
Sue's child is Sandy.
Diane has no children.
Matt's children are Ann, Alice.
```

Answer the following query:

Who has the most children in Daywood?

The semantic network for this problem is shown in Figure 15. SNAP solves this problem by having each person count the number of his children. Then, each person sends his number to the Daywood node, who finds the maximum. Then the Daywood node distributes the maximum back to the residents. Finally, each person compares the number of his children to the maximum. If the two numbers are equal, then that person has the most children.

Assume marker #1 is in Daywood, marker #2 is in the residents of Daywood, and marker #3 is in all of the children of the residents. The SNAP program is then:

```
1  LOAD %,R7,0                      ;load 0 into R7 of every node
2. LOAD #1,R7,1                     ;load 1 into R7 of the children
```

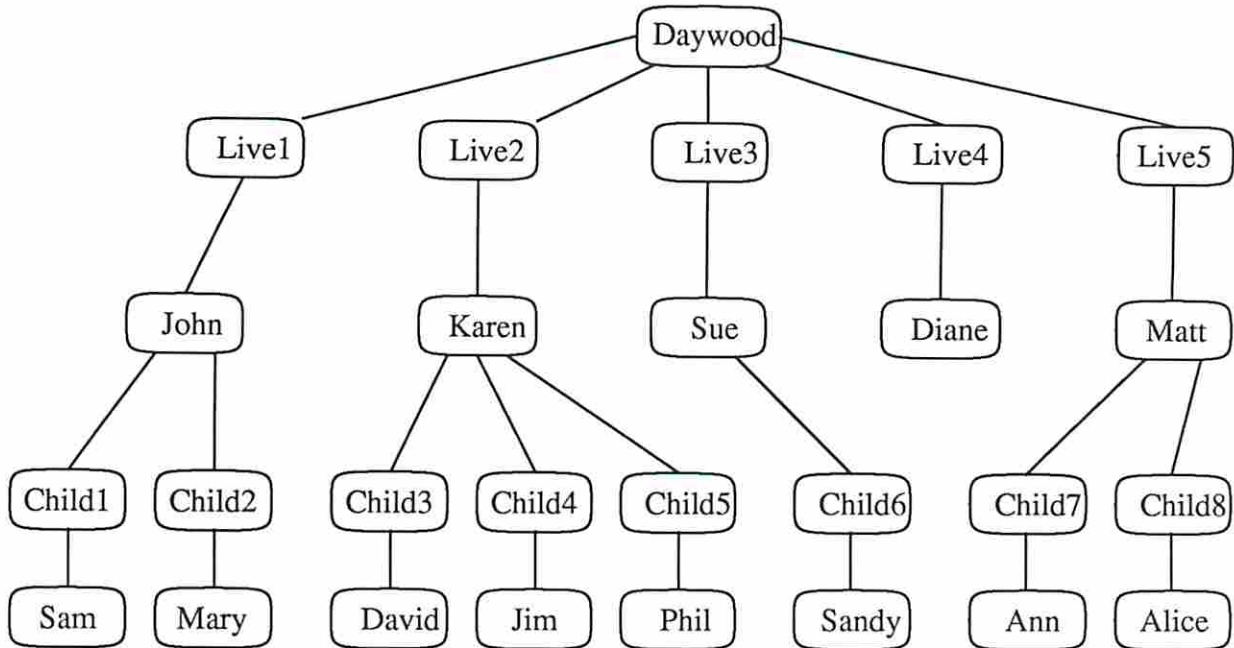


Figure 15: Semantic network for most children problem

```
3.  MARKER-ADD #1,R7,R7,#3,SEQ(R-LINK, R-LINK)
                                        ;add 1 to the number of children of the parent.
4.  WAIT-COMM-END
5.  MARKER-MAX #2,R7,R7,#4,SEQ(R-LINK, R-LINK)
                                        ; send the number of children to Daywood
                                        ; where a maximum is performed.
6.  WAIT-COMM-END
7.  LOAD %,R6,0                         ;load 0 into R6 of every node
8.  REG-ADD #1,R6,R7                    ;in Daywood: R6=R7
                                        ;(maximum number of children)
9.  MARKER-ADD #1,R6,R6,#5,SEQ(F-LINK, F-LINK)
                                        ; send the maximum back to the residents.
10. WAIT-COMM-END
11. REG-SUB #2,R7,R6,R5                 ;R7=R6-R7 with R5=flags.
12. TEST #2,R5,Z,#6                     ;set marker 6 in all residents whose
                                        children equal the maximum
13. COLLECT-COLOR #6                    ;get the person with the most children
```

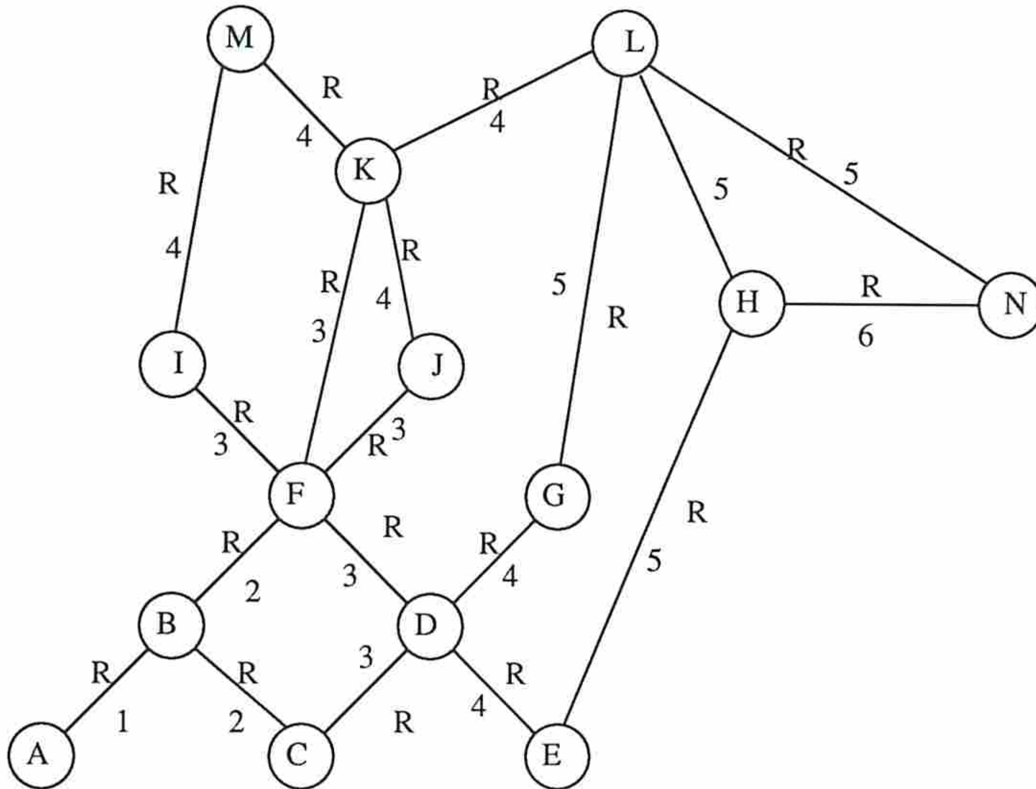


Figure 16: Find distance from A to any node

5.5 Graph

SNAP is well suited for graph problems. The nodes in a graph map directly to cells in the SNAP array. The links of the graph can easily be represented as relations between SNAP nodes. In this section we present a typical graph problem and show how it is solved using SNAP.

Distance from A to any node

Given the graph in Figure 16, find the distance from node `A` to any other node. In knowledge processing, this problem could be phrase as:

"Find the Semantic Distance from A to all other nodes"

The problem is solved very easily in SNAP using the MIN+ instruction. MIN+ allows SNAP to compute the minimum of two numbers, store the minimum in the cell, and then add one to the value before further propagation. Thus, starting a node A, we just propagate a MIN+ to all of the nodes in the array connected by `R` relations.

The SNAP program is then:

```
1. LOAD %,R7,32623                ;put the largest possible number in
                                  to R7 of all cells.
2. SELECT-COLOR A,#1              ;select node A
3. LOAD #1,R7,0                   ;put 0 in node A (distance from A to A is zero).
4. MIN+ #1,R7,SPREAD(R1)          ;use MIN+ to find distance
5. WAIT-COMM-END
6. READ %,R7                      ;get the distances from all of the nodes.
```

6 Simulation Results

For the purposes of verifying the design of SNAP and exploring the performance of SNAP, a simulator has been implemented on a SUN 3/280 using SUN Common LISP.

6.1 Simulator

The simulatior is configured for a full SNAP of 256 chips with each one having 64 nodes. Each node is structured using the same relation memory format described in section 3. The current SNAP simulation program supports the following features:

- The complete SNAP instruction set.
- User interactive instructions which initialize and restart the simulator.
- Instructions for reporting the simulator status after execution.
- Showing the marker activities during execution for program tracing and debugging.
- Showing the communication time for marker activities.
- User configuration of the nodes per chip, total relation memory, number of markers, and queue size.
- System clock for performance evaluation.
- The simulator also handles most of the controller functions.

In addition, some assumptions have been made in the design of the SNAP simulator:

- Although, the marker unit, processing unit and communication unit operate in parallel in the actual SNAP hardware, the simulator implements them sequentially.
- The marker unit forms messages in sequential manner.
- If there are conflicts between the marker unit and the processor unit, the marker unit has priority.
- If there are conflicts between the marker unit and the communication unit when accessing queues, the communication unit has priority.
- Each SNAP instruction has its own weight of execution time. When there is no message for the marker unit, the execution time of current step is based only on the instruction execution time.

6.2 Results

The simulator has been modified to reflect iterative improvements of the propagation rules, interconnection networks, and instruction sets. In this section, we will show some of the simulation results that indicate the improvement according to different modifications. For more detail information about the design of the SNAP simulator see [Lin 1989].

Many examples have been studied on the simulator. Through the simulation results, we were able to verify the feasibility of SNAP design, to test different allocation schemes, and to define an efficient knowledge representation methodology.

SNAP operation simulation

The proper operation of SNAP was verified by running examples on the simulator. With the functions for monitoring the status while execution, the trace of the examples proved that the all the primtive instructions of SNAP are conceptually correct and there are no hidden difficiencies in the design. Most of these results are documented in [Lin 1989].

Braching factor

The braching factor was studied through a semantic network which had a tree with the height of four. The first problem was to match query with

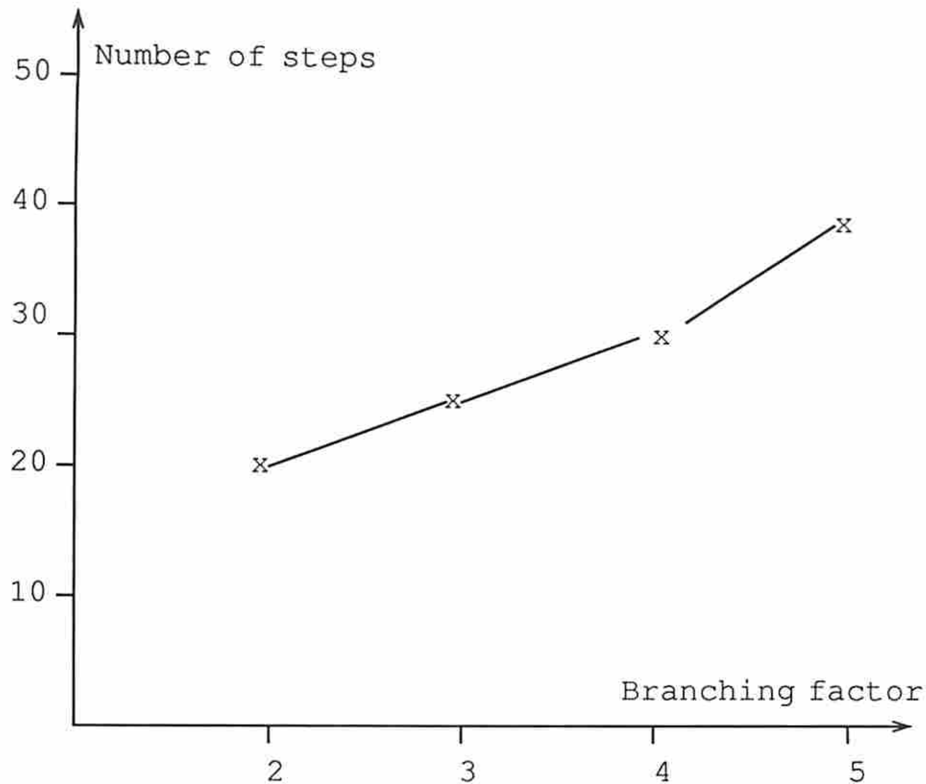


Figure 17: Simulation result for different allocation schemes

the the network having different number of nodes and braching factors. The output of this operation was the matching time as a function of branching factor. The result is shown in Figure 17

7 Comparison with Other Knowledge Processing Systems

Many approaches have been proposed in the past to provide knowledge representation and reasoning. In this section, we will examine some of these approaches and compare them with SNAP. A summary of this section is illustrated in Figure 18.

Approach		Number of threads of reasoning	Control of marker prop	Ability to restrict domain of marker propagation.	Numeric processing capaiblity	Implementation level	Limitation
Spreading activation	[Quillian 1966]	1	none	none	No	Model	Marker propagation irrespective of relation type. Different solution can result from different activation strength. Illogical inferences can result
	[Hendler, 1988]	1	none	none	No	Simulator	Correctness of solution done on host computer. Communication bottleneck between semantic network and host computer
NETL	[Fahlman 1979]	15	by controller	by controller	No	Simulator	Type of information that can be stored and reasoned on is limited
KL-ONE	[Brachman 1985]	1	Not applicable	Not applicable	Yes	Software	Implementation in software on a uniprocessor-inherently slow
Connection Machine	[Hillis 1985]	1	By nodes using software	By nodes using software	Yes	Hardware	Lacks specialized communication hardware to adequately take advantage of the parallelism in knowledge representation problems.
SNAP		32	By nodes using hardware	By nodes using hardware	Yes	Simulator Hardware, planned	

Figure 18: Comparison between Marker-Passing architectures (KL-ONE has been also included)

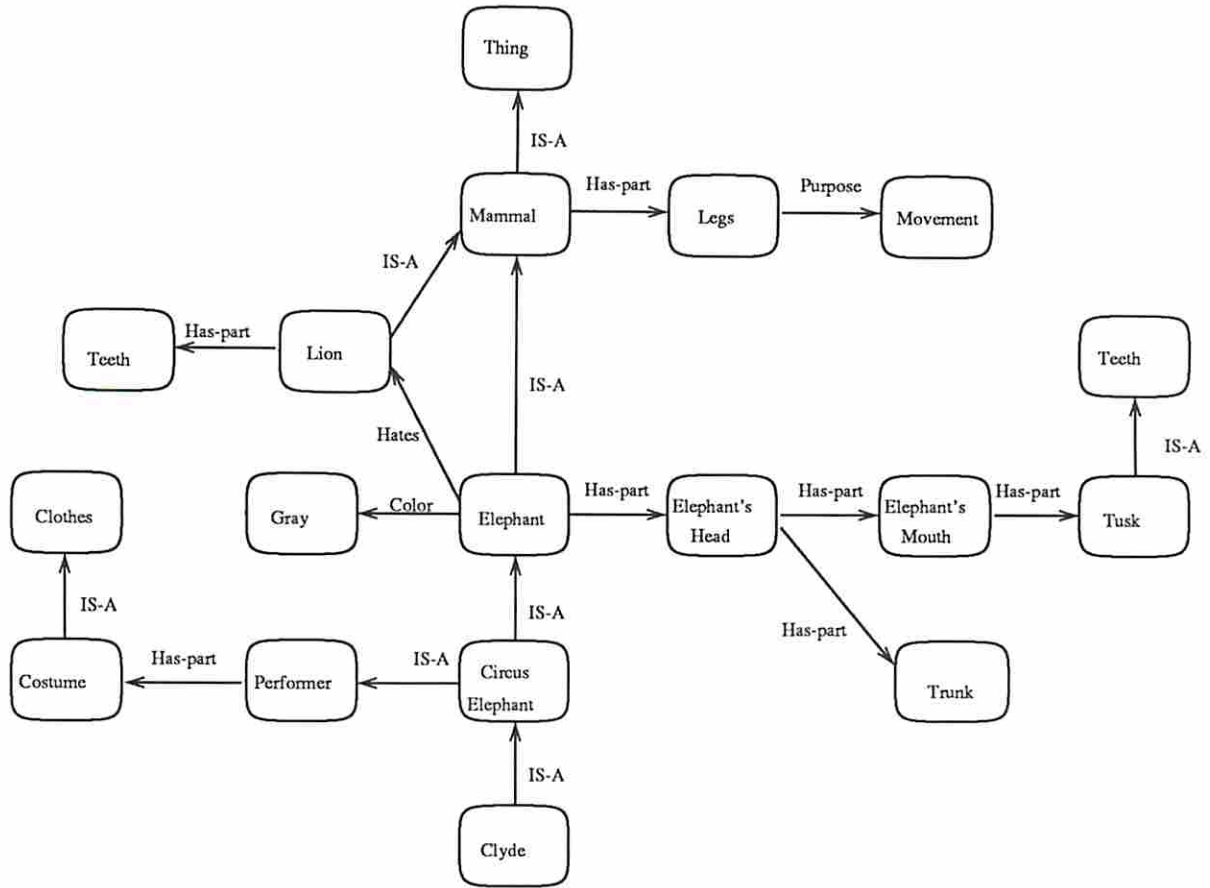


Figure 19: Quillian Semantic Network for Clyde

7.1 Spreading Activation

Marker passing semantic networks were first proposed by [Quillian 1966]. Quillian's semantic networks differed from SNAP semantic networks in that relations were shown as the labels on the arc connecting two nodes. Figure 19 shows a Quillian'ssemantic network for Clyde the Circus Elephant.

Marker passing in Quillian's scheme was very basic. A marker placed on a node would mark all other nodes connected to it by relations. From there the marker would pass to other nodes in the same manner. Thus, a marker M1 is placed on node Elephant it would propagate, in one phase, to the nodes LION, GRAY, MAMMAL, ELEPHANT'S HEAD, and CIRCUS ELEPHANT. In the next phase, TEETH, PERFORMER, CLYDE, ELEPHANT's MOUTH, TRUNK, LEGS, and THING would get marked.

One obvious problem, with the *spreading* of markers in this manner, is that it would eventually mark all of the nodes in the semantic network! Quil-

lian's solution to this problem was to assign an *activation* strength to each marker. The activation strength served as an *age* counter which decremented at each node that the marker visited. Thus, if the activation strength was set at 5, then only those nodes within a radius of 5 from the start node would get marked.

However, this solution makes Quillian's scheme unsatisfactory for inferencing. The first problem lies with the activation strength. If the activation strength is set too low then the marker may not get to the correct node. For example, if the activation strength is set to less than seven, than the node teeth does not get marked, and an incorrect answer would be returned. But, setting the activation strength too high causes more of the semantic network to be marked and leads to additional cases of the second problem.

The second problem is more serious and comes about because the markers in Quillian's domain propagate to nodes irrespective of the relations that connect them. Thus, if the link between TUSK and TEETH were removed, the answer returned by Quillian's markers would still be YES. This is because a marker on ELEPEHANT would cross the relation HATES and mark LION. From LION, it is very easy to get to TEETH. However, the logic behind this approach is wrong, since the fact that an ELEPHANT hates a LION that has TEETH, does not imply that the ELEPHANT has teeth.

Thus, Quillian's markers show the presence of a path between two nodes, but does not show whether the path is a valid one.

[Hendler 1985] used a method similar to Quillian's in the area of planning. Hendler added FROM fields to each marker. The FROM field indicated the node the marker came from. Once all of the markers had finished propagating, Hendler had the semantic network return all of the paths that lead to a solution back to the host computer. The host then reconstructed all of the paths and evaluated the best path to the solution. Hendler was more interested in the path to the goal than whether a path existed. Thus, this approach is quite effective for that purpose.

However, Hendler's modification still makes the spreading activation approach unsatisfactory for logic inferencing. The reason is because most of the effort involved in determining the solution is now on the host computer. This results in a loss of parallelism and more importantly in a communication bottleneck. For, if it is not inconceivable that many possible solutions to a query exist. Thus, a great deal of time would elapse while the semantic network communicated all of the possible paths back to the host computer. A more efficient technique is to have the semantic network determine the correctness of a path and return only a final solution back to the host computer.

7.2 NETL

[Fahlman 1979] used a marker passing scheme as the basis for the architecture of his machine NETL. Fahlman's markers differed from Quillian's in that they differentiated between types of relations.

A significant relation in Fahlman's model was the virtual copy (VC). The VC relation is very similar to the Superconcept relation present in SNAP. It is used for the inheritance of properties from parents and the passing of properties to children. Fahlman also used other relations such as EXISTENCE, CANCELLATION, and SPLIT. Figure 20 shows the semantic network for Clyde the Circus Elephant using Fahlman's relation and node types.

Fahlman's markers differentiated between VC links and other links. Markers could travel indefinetely through VC links but only to one node via the other relations. In addition, markers could only propagate in one direction of a VC link at a time. Thus from Circus Elephant a marker can travel to Clyde or it can travel to Elephant and Performer, but not both cases. These limitations on marker movement prevented the runaway propagation present in Quillian's scheme.

The main drawback with Fahlman's approach was that it assumes that the properties that a node needs to inherit are those directly connected to its parents. Thus, in order for Clyde to inherit TEETH, the node TUSK would have to be directly connected to ELEPHANT. This results from the fact that markers can only travel to one node through non-VC links. However this puts a severe limitation on the information that can be contained in a semantic network. For example, the relationship between TUSK and Elephant's Mouth and Elephant's Head are lost if TUSK is moved to ELEPHANT.

7.3 KL-ONE

KL-ONE is a knowledge representation language developed by [Brachman 1985]. It is a system for representing knowledge using a combination of semantic networks and frames. KL-ONE served as starting point for several other systems such as KL-TWO, NIKL, KRYPTON and most recently LOOM. KL-ONE is like semantic networks in that it has nodes with relations connecting them. But it also has elements of frames, with fillers that function much like slots in frames.

The heart of KL-ONE is a classification process. Once a concept is place in the knowledge hierarchy, it is very easy to retrieve information about it.

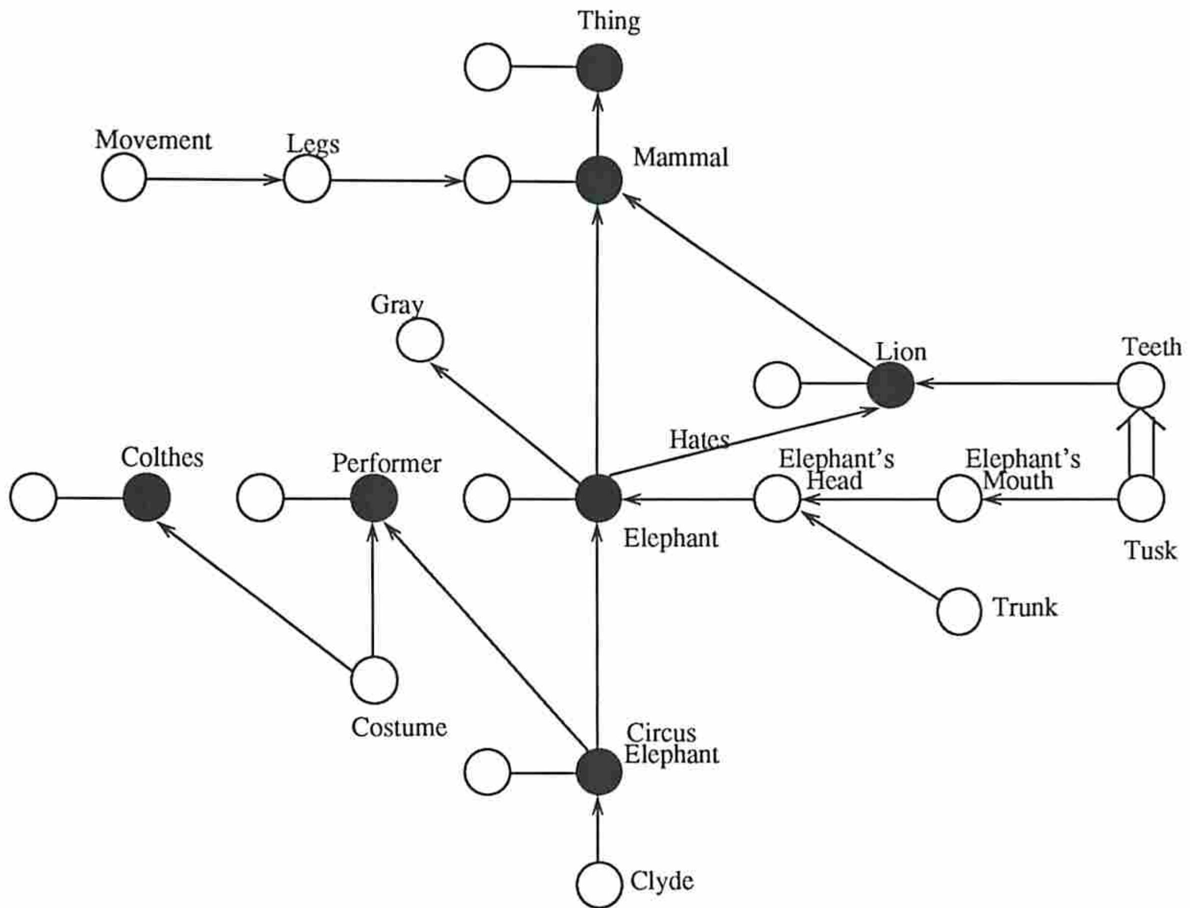


Figure 20: NETL Representation of Clyde

However, maintaining a consistent knowledge base is difficult. This is due to the fact that a change in one concept can ripple through the entire knowledge base. KL-ONE and its derivatives require each concept to maintain a list of pointers that identify which concepts it is dependent upon and which concepts are derived from it. Even using pointers a lot of computational overhead is still require to update the knowledge base to reflect new information. This is because KL-ONE is implemented without specialized hardware support. To date there are no KL-ONE machines, all of the work that has been done on the language has been done on general purpose computers or special LISP machines.

On our project, we have tried to utilize some of KL-ONE techniques in SNAP. Our knowledge structure, for example, can be traced to KL-ONE.

7.4 Connection Machine

The Connection Machine was developed by [Hillis 1985] as an implementation of Fahlman's NETL. Since then it has become to be realized as a much more general purpose tool.

The Connection Machine is a fine-grained array processor with programmable connections between nodes. As implemented in Hillis' prototype CM-1, the Connection Machine consists of 64K processors. Each processor has 4K bits of memory and a serial ALU. The processors operate in SIMD fashion, with messages being the method of communication.

The Connection Machine can operate as either a pattern matcher or a marker-passing machine. Pattern matching, however, is not suitable for general queries such as "Does Clyde have Teeth?".

As a marker passer, the Connection Machine suffers from two main problems. The first is due to the format of the messages in the Connection Machine. The messages lack the propagation rule field that SNAP has. Thus, the Connection Machine must execute actual instructions in order to process the message and determine the next destination. Consequently, markers must be processed in the foreground and in addition, only one marker at a time can be passed.

This limitation means that the Connection Machine can do nothing else while markers are being passed. Typically, only a small portion of the nodes in the network are actually participating in the marker passing. Thus, the majority of the nodes remain idle during the marker passing process. In order to solve most problems, several different markers are typically required. In the Connection machine, the passing of the markers must be done se-

quentially, one at a time. However, in SNAP the markers are passed in the background and have, as part of the message, their own propagation rule. Thus, as shown in section 5.1, all the markers can be propagating simultaneously. This results in a better mapping of the parallelism to the problem and a potential faster execution time.

Another limitation of the Connection Machine is that it is a serial machine. Each of the 64K processors can only process one 1 bit at a time. Thus much time is spent forming messages, decoding messages and determining the next destination of the marker. Even with its routing network, sending messages to other nodes takes long time.

[Chung 1989] programmed the "Does Clyde have teeth" example on the Connection Machine. Their results show that the Connection Machine requires over 2 million clock cycles to answer the query. However, it takes on the order of 10^3 clock cycles on SNAP to answer the query. If both machines run at the same clock speed, the speedup of SNAP over Connection Machine for this type of problem would be three orders of magnitude. Useful inferences over a large knowledge base requires much more processing than this simple query. Processing these types of problems on the Connection Machine could take a very long time, while on SNAP, only a fraction of that time would be needed.

8 Conclusions

We believe that SNAP represents a positive step in the building of machines that can understand natural language and represent knowledge in a useful manner.

As we have shown in this paper, the idea of marker passing and using semantic networks to represent knowledge is not new. What we have tried to do however is to create an approach that solves some of the weaknesses that we perceive other methods to have.

The use of Quillian's spreading activation resulted in inferences that may be illogical. Hendler's modification put too much load on the host computer. Fahlman's NETL put a severe limation of the type of network and thus the type of information that can be represented and processed. Hillis' Connection machine lacks the specialized hardware to effectively solve knowledge representation problems.

We believe that SNAP addresses these problems and is one step in the direction of building a machine that understands what we tell it. Hopefully,

one day such an intelligent machine shall exist. In the meantime, we feel that its pursuit is a worthwhile goal.

References

Brachman, Ronald J. and Schmolze, James G. [1985]. "An overview of the KL-ONE knowledge representation system", *Cognitive science* 9, 171-216.

Brachman, Ronald J. [1988]. "The basics of knowledge representation and reasoning", *AT&T Technical Journal*, 67:1, 7-24.

Chung, Sang-Hwa, Moldovan, Dan and Tung, Yu-Wen [1989]. "Reasoning on Connection Machine", Technical Report CENG-89-13. University of Southern California Department of EE Systems.

Fahlman, S. E. [1979]. "NETL: A system for representing and using real-world knowledge". The MIT Press, Cambridge, MA.

Hendler, James A. [1988] "Integrating Marker-Passing and Problem-Solving". Lawerence Erlbaum Associates, Inc.

Hillis, W. Daniel [1985] "The connection machine". The MIT Press, Cambridge, MA.

Hinton, Geoffery E. [1989]. "Implementing Semantic Networks in Parallel Hardware". In Geoffery E. and Abderson, James A. (Ed.) [1989] "Parallel models of associative memory". Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersy.

Lee, Wing. C. [1989]. "Bandwidth Analysis of Message-Passing Networks". Technical Report CENG89-24. University of Southern California-Department of EE Systems.

Lin, Changhwa and Moldovan, Dan [1989]. "SNAP Simulator Results", Technical Report CENG89-11. University of Southern California Department EE Systems.

Lipkis, Thomas A. and Schmolze, James G. [1983]. "Classification in the KL-ONE knowledge representation system", *Proceedings of the eighth international joint conference on artificial intelligence*, Vol. 1, 330-332.

MacGregor, Robert M. [1988] "A deductive pattern matcher", *AAAI88: The seventh national conference on artificial intelligence*, 403-408.

Moldovan, Dan I. [1983]. "An associative array architecture for semantic network processing", Techincal report ppp83-8, Electrical Engineering - Systems, Univ. of Southern California.

Moldovan, Dan I. and Tung, Yu-Wen, [1985]. "SNAP: A VLSI Architecture for Artificial Intelligence", Journal of Parallel and Distributed Computing, May 1985.

Quillan, M,R. [1966]. "Semantic Memory", PhD Dissertation, Carnegic Institute of Technology (Carnegic Mellon University).

Shastri, Lokendra [1988]. "Sematic Networks: An evidential formalization and its connectionist realization". The PITMAN Publishing Limited, London.

Sowa, J. F. [1984]. "Conceptual structure: information processing in mind and machine". Addision-Wesley Publishing.

Touretzky, David S. [1986]. "The mathmeatics of Inheritance systems" .The PITMAN Publishing Limited, London.

Tucker, Lewis W. and Roberston, George G. [1988]. "Architecture and applications of the Connection Machine", *Computer* 21:8, 26-38.

Woods, W. A. [1981]. "Research in knowledge representation for natural language understanding", Annual report no. 4785, Bolt Beranek and Newman Inc..