Dataflow architectures for knowledge representation systems*

by LUBOMIR BIC
University of California
Irvine, California

ABSTRACT

Most AI systems today are written in languages based on the sequential von Neumann model of computation and thus are not well suited to parallel processing. In this paper we propose to use logic programming as a bridge between the high-level operations performed by knowledge representation systems based on semantic nets and highly-parallel computer architectures. It will be shown how knowledge recorded in semantic nets, including the corresponding information retrieval operations, may be expressed in terms of logic programming. Such representation may be processed in a data-driven manner and thus permits computer architectures consisting of very large numbers of independent processing elements to be exploited.

*This work was supported by the NSF Grant MCS-8117516: The UCI Dataflow Databases Project.
INTRODUCTION

Most AI systems today are written in languages that are based on the sequential von Neumann model of computation and thus are executed on single-processor control-driven computer architectures. Even in the case of multiprocessors, the underlying model is still based on the basic principle of a fetch-execute cycle repeated by a processing element according to an instruction counter. The large semantic gap between the high-level operations performed by an AI system, e.g., logical inferences, pattern matches, or the searching of large spaces, and the typical repertoire of low-level machine instructions into which such operations must be mapped, is a serious limitation to high performance.

In recent years, logic programming has gained considerable attention in the realm of AI research. One of the main attractions of this approach is that logic programs do not presuppose a sequential von Neumann architecture and are therefore well suited to parallel processing. In this paper we propose to use logic programming as a bridge between AI systems based on the notion of semantic nets and computer architecture. We will first discuss the correspondence between semantic nets and logic programming. Then, by presenting a data-driven model for the execution of logic programs, we introduce a model for processing of semantic nets, suitable for execution on a computer architecture consisting of a very large number of homogeneous processing elements.

SEMANTIC NETWORKS

The use of semantic networks is a well-known approach to representation of knowledge in the realm of artificial intelligence. Even though there is not a precise, universally accepted definition of semantic nets, their common characteristics and advantages have been discussed extensively in the literature. A semantic net consists of a collection of nodes interconnected by arcs. Each node represents an object (physical object, set, relationship, event, logical entity) and each arc represents a binary relation. Usually, there is a finite set of such relations, e.g., “subset of”, “element of”, “agent”, “consequence”.

The main attraction of semantic nets largely centers around two factors. First, they provide sufficient expressive power to encode facts or concepts that are encodable in any other formal system. Second, the structures constituting a net serve themselves as a guide for information retrieval: information most closely related to a node is found simply by following arcs emanating from that node.

The main problem with semantic nets is the lack of exact rules or procedures governing the retrieval and modification of information represented by the semantic net. Most systems based on semantic nets assume the existence of some outside agent—a collection of programs—that embodies the knowledge about how to use the information in the net. This assumption makes parallel processing of semantic nets extremely difficult; it requires that the central processing agent divides computation into smaller tasks, distributes these to different processing elements, and coordinates their progress. As in other multiprocessor systems based on the notion of centralized control, these tasks are extremely difficult to solve and usually permit just a few processing elements to be exploited.

To solve the above problems, we propose a different model of computation: instead of assuming an outside agent to manipulate and use the network, the processing is incorporated into the network itself, i.e., each node is assumed to have some computational power and the ability to communicate with its neighbors via the network’s arcs. This point of view will permit the mapping of a semantic net onto an architecture consisting of large numbers of independent processing elements without the need for any centralized control.

SEMANTIC NETWORKS AND LOGIC PROGRAMMING

Dehnavi and Kowalski have proposed an extended form of semantic nets which may be mapped onto the firm grounds of first order predicate logic and thus provides an exact formalism for processing such nets. The following paragraphs describe briefly the correspondence between logic programming and the extended semantic nets.

A logic program is a set of clauses of the form

\[ p_0 \leftarrow p_1, \ldots, p_n \]

Each \( p_i \) is called a literal and has the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol and \( t_1, \ldots, t_n \) are terms. Terms may be constants, variables, or function symbols. \( p_0 \) is called the head or conclusion, and \( p_1 \) through \( p_n \) form the body or conditions of the clause.

A clause with an empty set of conditions is called an assertion and is used to represent explicit facts. If we restrict all predicates to be binary, each literal \( p(t_1, t_2) \) may be represented in the form of a directed arc as shown in Figure 1.

Figure 1—An arc corresponding to the predicate \( p(t_1, t_2) \)
The collection of all assertions then may be represented as a graph. Literals sharing the same term result in arcs connected to one another via the corresponding node. Since many terms may be shared among different literals, graphs of arbitrary complexity may result. This is illustrated by the sample logic program shown below (adopted from Reference 5):

1. \( \text{act}(e, \text{give}) \leftarrow \)
2. \( \text{obj}(e, \text{book}) \leftarrow \)
3. \( \text{actor}(e, \text{john}) \leftarrow \)
4. \( \text{rec}(e, \text{mary}) \leftarrow \)
5. \( \text{likes}(	ext{john, book}) \leftarrow \)
6. \( \text{likes}(X, Z) \leftarrow \text{act}(U, \text{give}), \text{obj}(U, Y), \text{actor}(U, X), \text{rec}(U, \text{mary}), \text{likes}(X, Y) \)
7. \( \leftarrow \text{likes}(	ext{john, mary}) \)

Lines 1 through 5 are assertions which form the semantic net shown in Figure 2. The constant \( e \) represents an act or event of giving, where \( \text{john} \) is the actor, \( \text{mary} \) is the recipient, and \( \text{book} \) is the object being given. The fact that \( \text{john} \) likes the \( \text{book} \) he is giving to \( \text{mary} \) is also recorded (line 5).

A clause which contains both a head and a body can be interpreted as recording implicit information. For example, line 6 of the above logic program can be paraphrased as follows: if it can be shown that \( X \) gives an object \( Y \) to \( Z \) (more formally, there exists an \( \text{act} \, U \) of giving, of which \( Y \) is the object, \( X \) is the actor, \( Z \) is the recipient), and \( X \) likes the object \( Y \), then it may be concluded that \( X \) likes the person \( Z \). To represent such statements graphically, Deliyanian and Kowalski introduce two types of arcs—one to represent conditions and another to represent conclusions. With such a representation, deductive information, which is normally part of the programs which process the semantic net, becomes actually part of the network itself.

A clause with an empty conclusion is interpreted as a request or goal which the system tries to solve through the principle of resolution—a sequence of unification steps attempting to derive an empty clause. In Figure 1, line 7 shows a goal that may be paraphrased as the query: “does \( \text{john} \) like \( \text{mary} \)?” To find an answer, the system will try to unify this goal with some other clause, in our example, the clause on line 6. The variables \( X \) and \( Z \) are bound to the constants \( \text{john} \) and \( \text{mary} \), respectively, thus generating five new goals—\( \text{act}(U, \text{give}), \text{obj}(U, Y), \text{actor}(U, \text{john}), \text{rec}(U, \text{mary}), \text{likes}(X, Y) \)—all of which must be solved in order to satisfy the original goal. Figure 3 shows the graphical representation of the above five goals. The resolution process is repeated until one (or more) solutions are found, or no further unifications are possible.

In the graphical representation, the body of each clause may be viewed as a graph template. Each unification within the resolution process then corresponds to the following graph matching problem: determine possible bindings of all free variables in the given graph template to constants such that the template matches some portion of the underlying semantic net. In the above example, binding the variables \( \text{U} \) and \( \text{Y} \) to the constants \( \text{e} \) and \( \text{book} \), respectively, yields a match between the template of Figure 3 and the semantic net of Figure 2, implying that the predicate \( \text{likes}(\text{john, mary}) \) is true.

**DATAFLOW VIEW OF PROCESSING**

In the approach described in the previous section, the processing of the (extended) semantic networks is defined in terms of executing logic programs. That is, the resolution principle of logic programming is used to extract knowledge from the semantic net. Graphically, each unification step corresponds to finding a match between the graph template representing the given clause and some portion of the underlying semantic net.*

The model presented in this paper is based on the idea of performing the unification steps by directly operating on graphs, instead of executing the corresponding logic programs. The necessary graph matching may be performed in a data-driven manner, which lends itself to an implementation on a highly parallel computer architecture. The basic idea is to view the semantic net as a dataflow graph. That is, each node of the network is not simply a passive element of information; rather, it is an active element, capable of receiving,
processing, and emitting value tokens traveling asynchronously along the network's arcs. The necessary graph matching is then accomplished by placing the graph template on tokens and injecting these into specific nodes of the semantic net. From their injection points, tokens replicate simultaneously into many possible directions thus searching for the given pattern. Since many processing elements may be engaged in the processing and replication of tokens, a computer architecture consisting of a large number of independent processing elements may be employed.

DATA-DRIVEN GRAPH MATCHING

In this section we develop the procedures for propagating the template-carrying tokens through the underlying semantic net. (These procedures are based on those presented originally in Reference 6.)

Matching of a Single Arc

We consider first the simplest form of a template, consisting of a single arc \( T_i \rightarrow T_j \). This arc corresponds to the literal \( p(T_i, T_j) \). To find a match, we need to determine possible bindings for the terms \( T_i \) and \( T_j \) such that the graph template \( T_i \rightarrow T_j \) matches some arc of the semantic net. Operationally, this is accomplished by placing the graph template on a token and injecting it into specific nodes of the semantic net. From each of these nodes the token is replicated along existing arcs in an attempt to find a match. We can distinguish the following four cases:

1. Both nodes \( T_i \) and \( T_j \) are bound to round terms \( t_i \) and \( t_j \), respectively. Since there can be only one occurrence of each of the nodes \( t_i \) and \( t_j \) in the semantic net, the token is injected into one of these, say \( t_i \). This node then replicates the token along all arcs labeled \( p \) that emanate from \( t_i \). If one of the nodes receiving the replicated template matches the second term \( t_j \), the subgoal is solved successfully; otherwise there is no direct match for this template. The same result is obtained when the token is initially injected into \( t_j \) from which it replicates in a search for \( t_i \). This will be denoted by reversing the direction of the arc: \( T_j \leftarrow T_i \).

2. The node \( T_i \) is bound to a ground term \( t_i \) while the node \( T_j \) is a free variable. As in the first case, the token is injected into the node \( t_i \) from which it is replicated along all arcs labeled \( p \). This time, however, any node \( t_j \) receiving the replicated token may be bound to the variable \( T_j \) and hence presents a solution to the given subgoal \( p(T_i, T_j) \).

3. The node \( T_j \) is bound to a ground term \( t_j \) while the node \( T_i \) is free. In this case, reversing the arc to \( T_j \leftarrow T_i \) yields a situation analogous to (b), where the first term is bound while the second is free. Hence the same approach can be taken.

4. Both variables \( T_i \) and \( T_j \) are free. This case differs from the previous three in that there is no unique injection point for the token. Rather any node of the semantic net is a potential binding for either variable and hence the token must be broadcast to all nodes of the semantic net. Each of these nodes binds the first variable \( T_i \) to its own content and replicates the token along all arcs labeled \( p \) in the same way as described under (b). In other words, the search is started in all nodes simultaneously.

Matching of Linear Chains of Arcs

In this section we extend the scheme for matching of individual arcs as discussed above to cope with sequences of arcs of the form:

\[ p_i(T_{i_1}, T_{i_2}), p_2(T_{i_2}, T_{i_3}), \ldots, p_n(T_{i_n}, T_{i_{n+1}}), \]

where each \( T_i \) must satisfy one of the following restrictions:

1. It must match the term \( T_{i-1} \rightarrow T_i \), or
2. It must be bound to a constant.

In the first case the arcs corresponding to the two adjacent literals \( p_i \) and \( p_{i+1} \) are connected, resulting in a graph template of the following form:

\[ \ldots \rightarrow T_i \rightarrow T_{i+1} \rightarrow \ldots \]

In the second case, where \( T_i \) is bound to a constant, we introduce a dummy arc, denoted as \( \rightarrow \), to create an artificial connection between the two adjacent arcs \( p_i \) and \( p_{i+1} \), i.e.,

\[ \ldots \rightarrow T_i \rightarrow T_{i+1} \rightarrow \ldots \]

A sequence of literals where each \( T_i \) satisfies one of the two above restrictions has the following important property: all literals in that sequence, except possibly the first, will have the left-most term \( T_i \) bound when the sequence is processed from left to right. We will refer to such a sequence as linear.

The fitting of a linear sequence into the semantic net then corresponds to the following graph fitting problem: determine possible bindings for all terms \( T_i \) such that the graph template matches some path in the semantic net. Operationally, this is accomplished as follows: A token, carrying the entire graph template, is injected into nodes of the semantic net that may be bound to the first term \( T_1 \). (As was the case with individual subgoals, only one such node \( t_i \) will exist if \( T_1 \) is bound to a ground term; otherwise the token must be injected into all nodes of the semantic net.) Each node \( t_i \) receiving the injected token will replicate it along all arcs that match the template arc \( p_i \). Each of the nodes \( t_j \) receiving the replicated token, will attempt to bind itself to \( T_{i+1} \). At this point, the next arc of the template is either a regular arc \( p_{i+1} \) or it is a dummy arc \( \rightarrow \). In the first case, \( t_j \) would continue the propagation of the token along all arcs matching the name \( p_{i+1} \). In the second case, the token is simply sent to the node that matches the corresponding term \( T_{i+1} \). There is at most one such node since, by definition, \( T_{i+1} \) must be bound.

The same steps are performed by all nodes \( t_i \) receiving the token, which results in a stepwise expansion of the graph.
template into all possible directions of the semantic net. Each branch continues to grow until one of the following conditions occur:

a. A node $t_i$ is unable to bind itself to the corresponding node $T_i$ (i.e., $T_i$ is already bound to a term different from $t_i$), or, no arc emanating from $t_i$ matches the corresponding template arc $p_i$. This indicates that no match can be found along this path.

b. The last node $T_i$ of the template has been reached, implying the detection of a match for the given graph template.

**Matching of Tree-Structured Graph Templates**

We define a tree-structured template recursively as follows:

1. A linear chain of arcs
\[
p_i(T_1, T_2, p_i(T_1, T_2), \ldots , p_n(T_n, T_2)),
\]
   as defined in the previous section is a tree.

2. If $p_i(T_1, \ldots , T_i, \ldots )$ and $p_i(T_1, \ldots , T_i, \ldots )$ are trees, then the construct
\[
(p_i(T_1, \ldots , T_i, \ldots ))(p_i(T_1, \ldots , T_i, \ldots ))(p_i(T_1, \ldots , T_i, \ldots ))
\]
forms a tree, where $T_i$, which is shared by both literals $p_i$ and $p_i$, becomes the root of that tree. Note that by repeatedly applying the above rules, arbitrarily many subtrees may be created at each node.

The algorithm for fitting linear graph templates described in the previous section may be extended to cope with tree-structured templates as follows:

Assume that the node $t_i$ receives a tree-structured template of the form
\[
(p_i(T_1, \ldots , T_i, \ldots ))(p_i(T_1, \ldots , T_i, \ldots ))(p_i(T_1, \ldots , T_i, \ldots ))
\]
As a first step, $t_i$ will attempt to bind itself to the root variable $T_i$. If this is successful, it will start a matching process for each of the $n$ subtrees. That is, for each subtree $p_i(T_1, \ldots , T_i, \ldots )$, the node $t_i$ forms a new token and replicates it along all arcs that match the name $p_i$. As was the case with linear templates, it will await responses from all directions into which tokens have been sent. Solutions for the original tree-structured template are then constructed by combining the possible solutions for each of the individual subtrees.

To illustrate this procedure, consider again the semantic net of Figure 2. Assume, however, that instead of the template shown in Figure 3, we are interested in finding matches for the tree-structured template of Figure 7. The fitting process will begin with injecting a token carrying the given template into the node of the semantic net that matches the template's root node; this is the node $john$. From there, a copy of the template is replicated along all arcs matching the label *act*. All receiving nodes will bind their own names to the variable $U$; in the above example, this will be the node $e$. At this point there are three subtrees, all of which must be fitted into the semantic net. The first will be sent along all arcs matching the label *act*; the receiving node *give* will match successfully against the corresponding template node *give*. The second subtree is sent along arcs matching the label *rec*, thus reaching the node *mary*. Finally, the third subtree is sent along all arcs that match the label *obj* and, since the template variable $Y$ may be bound to the node *book*, it continues along the arc *likes* back to the node *john*. Since all subtrees have successfully been fitted into the semantic net, the question "does *john* like *mary*?" has been answered affirmatively.

**Dynamic Graph Templates**

It is not always possible or convenient to express the requests for extracting some knowledge from a semantic network as a fixed graph template. To illustrate such a situation, consider Figure 4, which shows several concept nodes arranged into a "is-a" hierarchy. A property called *color* is associated with the node *typical-cat*.

Typical for such a hierarchy is the principle of *property inheritance*, which states that a property applies not only to the node to which it is attached but also to all descendants of that node within the hierarchy. For example, the property *color* in Figure 4 applies not only to the node *typical-cat* but also to the individuals *bertha* and *fritz*. In logic programming, this can be expressed conveniently as the following recursive clauses:

\[
\text{inherited-prop}(X, Y) : \leftarrow \text{loc-prop}(X, Y)
\]
\[
\text{inherited-prop}(X, Y) : \leftarrow \text{is-a}(X, Z), \text{inherited-prop}(Z, Y)
\]

Operationally, this program may be paraphrased as follows: Check first if $Y$ is $X$'s local property. If so, the query is satisfied, else go to $X$'s parent in the 'is-a' hierarchy and perform the same check. Repeat this process until the query is satisfied or the top of the hierarchy is reached, indicating a failure.

![Figure 4-An 'is-a' hierarchy](image-url)
By binding actual node names to one or both of the variables \( X \) and \( Y \), the semantics of the query can be changed. For example, the meaning of the query \( \text{inherited-prop}(\text{fritz}, \text{color}) \) is interpreted as “does fritz have a color”, and yields the value true or false. The meaning of \( \text{inherited-prop}(\text{fritz}, Y) \), on the other hand, is to “find all properties of fritz”; finally, the meaning of \( \text{inherited-prop}(X, \text{color}) \) is to “find all elements that have color as one of their properties”. The body of each of the above queries may itself be viewed as a network (or a collection of networks). Consider for example the query \( \text{inherited-prop}(\text{fritz}, \text{color}) \). From the above recursive definition of \( \text{inherited-prop} \) we obtain

\[
\text{inherited-prop}(\text{fritz}, \text{color}): \leftarrow \text{loc-prop}(\text{fritz}, \text{color})
\]

\[
\text{inherited-prop}(\text{fritz}, \text{color}): \leftarrow \text{is-a}(\text{fritz}, Z), \hspace{1cm} \text{inherited-prop}(Z, \text{color})
\]

which can be represented as an infinite sequence of networks shown in Figure 5. The task of answering that query is then to determine whether at least one of these networks matches some portion of the underlying database network. As indicated earlier, this may be performed by asynchronous propagation of tokens through the database network.

In the above simple case a query token is propagated from \( \text{fritz} \) to nodes at higher levels in the ‘is-a’ hierarchy and each such node checks for the existence of a \( \text{color} \) at its own level. If no such node is found when the top of the hierarchy is reached, the query reports a failure; otherwise a positive answer is given.

**TRANSFORMATION OF GRAPH TEMPLATES**

The procedures for graph fitting presented in the previous section assume that all templates have the form of a tree. (A linear chain is a special case of a tree.) In the most general case, however, graph templates of arbitrary interconnection topology may occur. Since tokens in dataflow systems propagate asynchronously, the detection of cyclic patterns is difficult. To solve this problem, each template is first transformed into a tree-form, for which token-propagation procedures have already been developed.

There is a number of ways to transform a given graph template into a tree. One possible approach is to find a depth first spanning tree, which partitions the template into two sets of edges—tree edges and back edges. (A procedure for constructing such spanning trees may be found, for example, in .) Figure 6 shows one possible spanning tree constructed for the template of Figure 3; back edges are shown as dotted lines. By replicating the nodes at which back edges terminate, we obtain the desired tree structure shown in Figure 7. At each node of the template, the corresponding subtrees represent independent goals which can potentially be processed in parallel since no free variables are shared among subtrees.

**ARCHITECTURAL REQUIREMENTS**

According to the model described in previous sections, a semantic net is a dataflow graph from which information is extracted by an asynchronous propagation of tokens. In this
section we examine the architectural requirements necessary to support this model.

Implementation of the Semantic Net

According to the model, each node of the semantic net is active, i.e., it must be capable of exchanging tokens with other nodes and of performing some computation on the received tokens. This implies the following requirements:

- Each node must be mapped onto a processing element (PE) which will perform the necessary token processing and communication operations on the node’s behalf; thus the PE is the incarnation of all nodes mapped onto that PE. There is no centralized control to start or supervise the execution of individual PEs; rather, the operation of each PE is triggered by the arrival of a token.
- Nodes must be able to communicate via logical connections corresponding to the arcs of the semantic net. This implies that any two PEs holding nodes connected via a logical arc must be able to communicate with one another; the communication may be direct or via other PEs.

The mapping of nodes constituting the semantic net onto a computer architecture consisting of \( p \) processing elements, may be accomplished by providing a global mapping function \( f \). Given a graph node \( n \), \( f(n) \) yields a number \( f(n) \), ranging between 1 and \( p \), which designates the PE that will hold the node \( n \) during execution. This mapping may be done statically or dynamically.

The lack of centralized control permits potentially a very large number of PEs to be employed since scheduling is performed implicitly by the arrival of tokens. The issue, however, is to provide the necessary communication network to permit the nodes of the semantic net to exchange tokens. Since the topology of the semantic net will not be known a priory, and may also change during execution, we must assume that each PE is capable of communicating (directly or indirectly) with any other PE. Unfortunately, since the number of PEs is expected to be large, a fully connected network will not be feasible; rather, the choice of a suitable interconnection scheme is a trade-off between an acceptable rate of token flow and the cost of the necessary hardware. (Simulation experiments are currently being conducted in our laboratory to investigate this issue.)

Assuming that a communication path may be established between any two PEs, a logical arc between two nodes \( n_1 \) and \( n_2 \) of the semantic net may be represented by recording the name of the node \( n_1 \) with the node \( n_2 \), and vice versa. To send a token along such an arc is then accomplished by finding a physical path between the PEs holding the two nodes and routing the token toward its destination.

Injection of Tokens

According to the model it must be possible to inject tokens into any node of the semantic net. In the implementation it implies finding the address of the PE holding the receiving node. This is very similar to the problem of providing communication channels (logical arcs) between nodes as discussed above and thus may be solved using the same mechanisms: a sender, which in the case of token injection would be an external source, as opposed to one of the PEs, can determine the PE that holds the destination node by applying to it the same mapping function \( f \) that has been used initially to map the graph onto the architecture. This PE number is carried by the token as its destination address when it is being routed through the physical network.

CONCLUSIONS

The main objective of the research presented in this paper was to find a bridge between the high-level operations performed to retrieve knowledge from a semantic net, and the underlying computer architecture. We have shown that the processing of semantic nets may be expressed in terms of logic programming, which in turn lends itself to non-von Neumann style of computation. The model is based on an asynchronous propagation of tokens through the semantic net in a search for graph patterns corresponding to given queries. Since a large number of independent processing elements may be engaged in the processing and forwarding of individual tokens, a highly parallel computer architecture may be employed. It should be noted that the architectural requirements of the model are very similar to those of general dataflow systems, which suggests that existing architectures, designed to execute dataflow programs, may easily be adapted to support the proposed model and thus provide a highly parallel execution environment for the processing of semantic nets.
REFERENCES
