Development of Techniques to Control Reasoning using SNePS

Pedro Neves
Instituto Superior Técnico, Portugal
pldn@mega.ist.utl.pt

Abstract: The Semantic Network Processing System (SNePS) is a knowledge representation formalism that implements a node-based inference mechanism on a semantic network. Processes are associated to nodes, communicate through channels and execute non-standard inference rules, allowing the assertion of new inferred nodes.
Other automatic deduction systems like Resolution, were born with the primary intention of finding proofs for theorems. PROLOG, a widely known logic programming system, searches for SLD-refutations for sets of clauses.
On this work we intend to develop mechanisms to control the inference made by SNePS. For that we describe and compare the SNePS Inference Package (called SNIP) with PROLOG. This comparison suggests improvements to SNIP that are latter implemented and tested.
A method to idle some processes, without affecting the number of solution found, is proposed and its theoretical foundation explained. Additionally a way to parameterize the search strategy is suggested, making the inference process more versatile.
Keywords: semantic network, SNePS, resolution, PROLOG, backward inference, automatic deduction.

1. Introduction

SNePS, (Shapiro, 1979), is a semantic network directed to represent natural language expressions, (Shapiro, 1999), and to reason about them. On this work, we focus on the automatic deduction capabilities of this system, more specifically on the backwards node-based inference mechanism implemented by the SNePS Inference Package, called SNIP, (Hull, 1986).
Our main objective is to understand the potential of SNIP and to propose methods to control and improve the reasoning performed by this system.
To achieve this aim, we start by describing SNIP and comparing it with PROLOG, a logic-programming system that will help us to find the advantages and drawbacks of the automatic deduction capabilities of SNePS. After this comparative study, we propose and implement some improvements to this system.

2. Understanding SNePS

On the first section of this chapter, we give some introductory concepts about SNePS. Afterwards, we explain how backwards node-based inference is performed by the system, through the association of processes to nodes.

2.1. Basics of SNePS

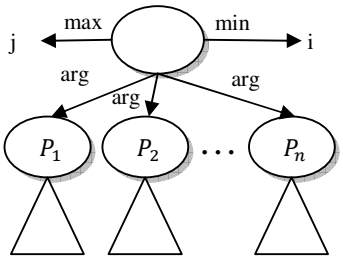
The definition of semantic network is not consensual. The fact that, according to Brachman ,"virtually every networklike formalism that appeared in the literature since 1966 has at one time been branded by someone a semantic net" lead us to, first of all, position SNePS on the taxonomies described on the works of (Brachman, 1979) and (Sowa, 1992):

- According to Sowa's characterization, SNePS is an *assertional* network;
- SNePS is a *propositional semantic network* since logical propositions are represented by a node. Therefore it is settled at the Brachman's *logical* level;
- It is possible to perform inference on SNePS through SNIP. Thus according to the Sowa's taxonomy, SNePS is also an *executable* network.

The syntax and semantics of the formalism can be found on (Shapiro, 1979) and (Martins, 2005). However, we stress four features that distinguish SNePS from other networks: concepts represent intensional entities, i.e. "objects of thought" - each intensional entity may correspond to more than one object of the real work or to no object at all (Shapiro, et al., 1987); the *Uniqueness Principle* guarantees that whenever two nodes represent the same intensional concept, then they should be the same; arcs represent non-conceptual relations between nodes and consequently arcs are structural rather than assertional; finally SNePS can represent higher order propositions and reason about them.

SNIP is capable of performing three types of inference: reduction, path-based (Shapiro, 1978) and node-based inference. On this paper we are interested in the last, more specifically on **backward node-based inference**.

This inference mechanism has a set of predefined case-frames (predefined arc names and node structures) that represent non-standard connectives (Shapiro, 1979). There are five types of rule nodes available: and-entailment, or-entailment, numerical-entailment, and-or and thresh (check (Shapiro, et al., 2004) for details). Inference rules are associated to each rule node. Figure 1 presents an and-or rule node, which asserts that between *i* and *j* of the $P_i, 1 \leq i \leq n$, are true.



$\wedge V_i^j\{P_1, \ldots, P_n\}$

$$\frac{P_a, \ldots, P_b}{\backsim P_k, \ldots, \backsim P_l}$$

where $P_a, \ldots, P_b$ are any j and $\sim P_k, \ldots, \sim P_l$ are the remaining $n - j$

$$\frac{\backsim P_a, \ldots, \backsim P_b}{P_k, \ldots, P_l}$$

where $\sim P_a, \ldots, \sim P_b$ are any $n - i$ and $P_k, \ldots, P_l$ are the remaining i

$\perp$: if more than j are true or more than $n - i$ are false, contradiction is derived

Figure 1 – And-or rule node, its linear representation and inference rules.

2.2. Backward node-based inference on SNIP

The match algorithm. In order to locate nodes and rules on the network in order to apply inference rules similar to those of Figure 1, SNePS implements an unification

algorithm (Shapiro, 1977). The *match algorithm* is applied to a node S and an initial *binding* β, producing a set of tuples that we will denote as $match(S, \beta)$. Each tuple has the form $\langle T, \tau, \sigma \rangle$, where T is a node, τ is a target binding and σ is a source binding. By *binding*[1] we mean a set of the form

$$\delta = \left\{ {}^{v_1}/_{t_1}, \ldots, {}^{v_n}/_{t_n} \right\},$$

where v_i are variable nodes and t_i are any nodes. We use $N\alpha$ to denote the result of the application of the binding α to a node *N*. Given the above definitions, if $\langle T, \tau, \sigma \rangle \in match(S, \beta)$, then $T\tau$ and $S\sigma$ are matched in the common sense of the word. The need for a source and target binding will became clear latter. Figure 2 shows the results of invoking $match(P_8, \{\ \})$ on the network.

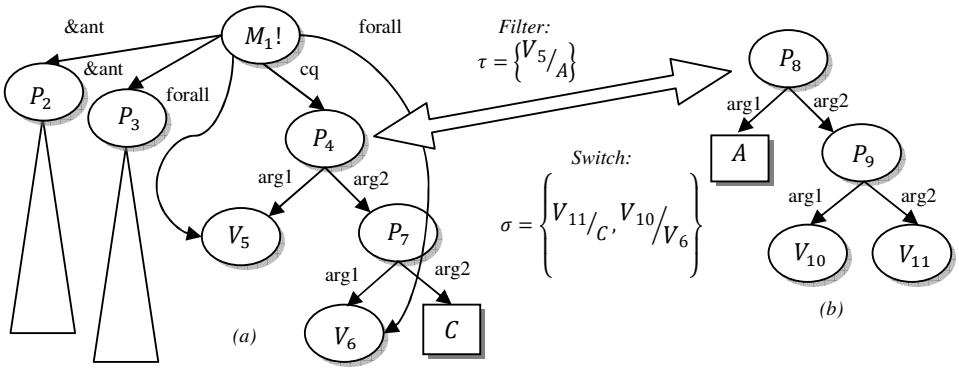


Figure 2 – Result of applying the operation of match on this network.

Associating processes to nodes. When a deduction is initiated by the user through the command **deduce**, SNIP creates and associates a process to each node involved on the deduction. There are two major types of processes: ***rule processes***, which implement inference rules and ***non-rule processes***.

Processes communicate using ***channels***, sending and receiving ***requests*** and ***reports***. When a node requests another, it is interested in finding solutions for that node. When a node finds new instances, it reports to the interested requestors.

Figure 3 will help to explain the kind of interaction that may exist in a network, when a backward node-based inference is started.

[1] We will use the terms *binding* and *substitution* interchangeably.

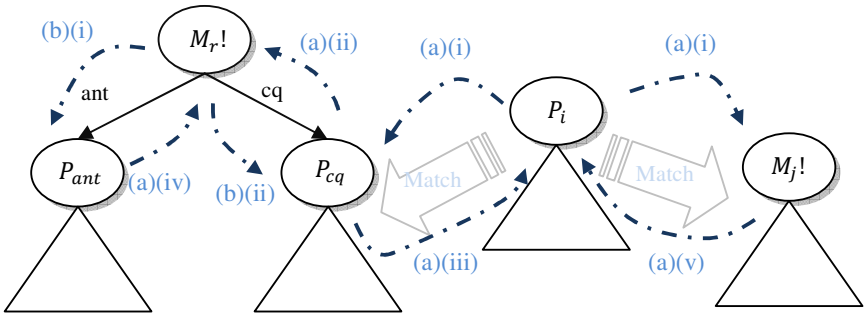


Figure 3 – Interactions between nodes on the network.

The behavior of ***non-rule nodes*** can be summarized as follows (see Figure 3):

- *Sending Requests*: a non-asserted non-rule node tries to find instances of itself matching and requesting other nodes (node P_i on interactions (a)(i)). Additionally, a node that is a consequent of a rule node may also request that rule node, as interaction (a)(ii) shows for node P_{cq}. The rule node will afterwards try to apply the inference rule associated and derive conclusions.
- *Replying Reports*: an asserted non-rule node (like $M_j!$) corresponds to a solution and can reply a report (see interaction (a)(v)). Non-asserted non-rule nodes may also report if they receive reports from other nodes: P_{cq} can report to P_i if it receives reports from the rule node $M_r!$ (interaction (a)(iii)), and antecedent nodes like P_{ant} may also report if they receive reports from matched nodes (interaction (a)(iv)).

The behavior of ***rule nodes*** can also be sketched in a similar manner:

- *Sending Requests*: when an asserted rule node receives a request from a consequent, it tries to apply the inference rule sending requests to all the antecedents of the rule in parallel (as interaction (b)(i) shows). Additionally, if the rule node is not asserted, then the rule node acts as an ordinary non-rule node and tries to find instances of the rule sending requests through consequent arcs that may exist (not shown on the picture).
- *Replying Reports*: when an asserted rule node like $M_r!$ receives reports from the antecedents, it tries to apply the inference rule and derive conclusions. If that is possible, reports are sent to the consequents (interaction (b)(ii)). If a non-asserted rule node receives instances of the rule, then it tries to apply those instances and produce reports to the consequents.

Processes interact under a ***consumers-producers model***, i.e. the same node (a producer) can report to more than one "boss" (which is a consumer). Producers keep all the inferred instances on a register, never reporting the same solution twice to the same consumer and, when a new "boss" is accepted, sending all the instances already derived to that "boss". These features mean that nodes are ***data-collectors***.

With this model, given that the same producer can report to more than one consumer, two problems may emerge: some solutions may be desired by one "boss" but not by the other and the namespace of variables may differ from consumer to consumer. That is why a fractioned version of the unifier is needed and those are the roles of both ***filter*** and ***switch***, respectively (check Figure 2 again).

The production of solution on rule-nodes is uniformly handled for all the connectives using the ***rule-use-information*** (RUI) data structure:

$$(\langle substitution\rangle, \langle non-negative\ integer\rangle,\ \langle non-negative\ integer\rangle, \langle flagged-node\ set\rangle).$$

When the antecedents of a rule node report solutions, they return substitutions for universally quantified variables. The consistency of the substitutions returned by antecedents (i.e. the correct instantiation of shared variables) is assured on the rule node, as RUI tuples are created for each consistent solution. These substitutions are stored as the first element of the tuple. The second and third elements are integers containing the number of true and false antecedents for that solution. $\langle flagged-node\ set\rangle$ indicates, for each solved antecedent, whether a true of false instance was found.

As an example, let's admit that on Figure 2, P_2also dominates variable V_5 and P_3 dominates variable V_6. If P_2 reports a true solution with substitution $\{V_5/C\}$ and P_3 a false solution with substitution $\{V_6/A\}$, then the following RUI would be created for the and-entailment node M_1!:

$$(\{V_5/C\ , V_6/A\}, 1, 1, \{P_2: true, P_3: false\}).$$

The decision to trigger inference is based uniquely on the integers of the RUIs.

Deciding which process gets into action – the MULTI system. MULTI is a LISP based multiprocessing system which allows "various control structures such as recursion, backtracking, coroutines and generators" **(McKay, et al., 1980)**.

SNePS uses MULTI processes, which are scheduled on one of two queues with different priorities: processes having reports to threat are scheduled on the HIGH priority queue and processes with pending requests are inserted in the LOW priority queue.

The HIGH priority queue allows a fast propagation of solutions and the low priority processes are only executed when no processes remain on the HIGH priority queue. Both queues are maintained using a FIFO policy.

If process p1 sends a request or report to process p2, then p1 is in charge of scheduling p2 on the LOW or HIGH priority queue, respectively.

3. The SLD-resolution mechanism

Resolution is an automatic theorem proving system created by Robinson (Robinson, 1965). The system was continually improved across time and is still the basis for state of the art systems like OTTER (McCune, et al., 1997).

Understanding resolution will give us a greater insight of SNIP and may suggest interesting concepts and ideas that can be used on that system.

We will focus on **SLD-resolution** and **PROLOG**. An introduction to the basics of resolution can be found on (Duffy, 1991) and we will assume that the reader knows the basics of that method.

3.1. SLD-Resolution

The ***SLD-Resolution*** system works exclusively with ***Horn Clauses***, which are sets of literals with at most one positive literal. If the positive literal is present, i.e. $C = \{\sim L'_1, \sim L'_2, \dots, \sim L'_n, L\}$, we have a ***program clause*** which can be more suggestively written as $C = L \leftarrow L'_1, L'_2, \dots, L'_n$. Otherwise, assuming L does not exist, we have a ***goal*** or ***query***, $G = \leftarrow L_1, \dots, L_k, \dots, L_m$.

We will assume the existence of an ***unification algorithm*** such that given two literals, L_1 and L_2, returns a substitution $\theta = unify(L_1, L_2)$ such that $L_1\theta = L_2\theta$.

Given the previous goal G, the previous program clause C and assuming that $\theta = unify(L_k, L)$ the application of the ***resolution rule*** to G and C is defined as:

$$R(G, C, L_k, L) = (\leftarrow L_1, \dots, L'_1, \dots, L'_n, \dots, L_m)\theta.$$

The application of the resolution rule to a goal G can be restricted through the use of a ***selection function***, (Kowalski, et al., 1983). This function receives G and returns a literal such that $\psi(G) = L_i$ and $L_i \in G$. We restrict the application of the rule only to the literal chosen by ψ.

Having a set of program clauses $P = \{C_1, \dots, C_p\}$ and an initial query G, a ***linear refutation*** of G, (Loveland, 1978), is a *n*-tuple $D = (C_1, \dots, C_p, G, G^1, \dots, G^l, \square)$, where all $G^i, 1 \le i \le l$ are obtained applying the resolution rule to G^{i-1} and one of the program clauses. The symbol $\square$ denotes the empty clause $\{\ \} \leftarrow \{\ \}$.

The SLD-Resolution method defines a **S**election Function, restricts the general resolution to ***D***efinite clauses (i.e. Horn Clauses), and only allows **L**inear Refutations.

3.2. PROLOG, a logic programming system

PROLOG implements an efficient depth-first search strategy with backtracking on the space of SLD-refutations assuming:

- that the selection function always chooses the ***left-most literal*** of the subgoal, i.e. if $G = \leftarrow L_1, \dots, L_k, \dots, L_m$, then $\psi(G) = L_1$;
- that the set of program clauses is ordered: $P = (C_1, \dots, C_p)$. It will be called as a ***program***.
- that whenever a given literal unifies with more than one clause of the program, the interpreter chooses first the one that comes first in the list: ***standard ordering rule***.

Figure 4 shows a search tree for the execution of the program $P = (C_1, C_2, C_3, C_4, C_5)$ for the query $\leftarrow p(X)$.

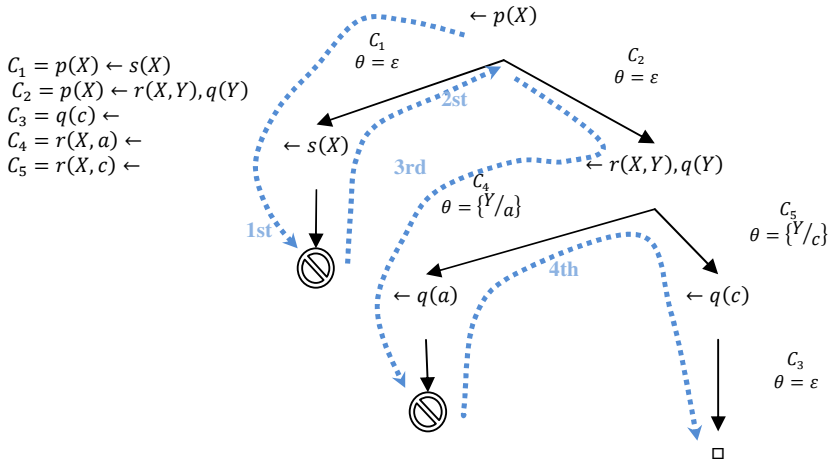


Figure 4 – Search tree generated by the execution of the program P, for the query $\leftarrow p(X)$

On the *SLD-refutation search tree* above we emphasize the following facts:

- Each node of the tree contains the goal clause that still remains to be solved;
- Each arrow contains the clause of the program that unified with the parent and each respective unification;
- The subgoal chosen on the node $\leftarrow r(X,Y), q(Y)$ was $r(X,Y)$, since we are assuming the *selection function* described before;
- The symbol ⊘ identifies the absence of a valid unification between the literal chosen by the *selection function* and the head of any clause of the program. Whenever that happens, the search backtracks to the parent goal, and searches for alternative unifications for the literal, only generating one successor of a node at a time.
- The dotted arrows show the sequence of exploration of the tree and make the standard ordering rule clear: $\leftarrow p(X)$ unifies first with C_1 and after backtracking it unifies with C_2.

4. A brief comparison

After the analysis of both systems, the following points of comparison can be stated:

- *Aim of the formalism*: while SNePS is first of all a knowledge representation formalism concerned with ***expressiveness*** and easiness of representation of natural language sentences (Shapiro, 1999), PROLOG is generally regarded as a logic programming environment that automatically proves theorems.
- *Complexity of the inference rules*: PROLOG applies one simple inference rule: the ***resolution rule***. On the other hand, SNIP implements five complex inference rules, which are associated to 5 predefined case-frames. The SNIP's inference process is also dependent on the structure of the semantic network, as messages are propagated using channels established between nodes.

- *Type of Reasoning*: also as a consequence of the complexity of the rules, the type of reasoning performed by SNIP is more "***human-like***" than the one made by resolution. SNePS inference rules are closer to the natural deduction system and resolution rule, although appropriate for automation, makes very small logical steps.
- *Search strategy used*:

- while the MULTI queues are managed using a ***FIFO*** policy giving rise to a ***breath-first*** search, PROLOG adopts a ***depth-first*** strategy with backtracking.
- SNIP inserts the antecedents of the rules in the queue by an order that is not specified and on PROLOG those antecedents are inserted by the order they appear in the rule.
- the unification operation on SNIP is extended to the entire network, on the contrary of what happens on PROLOG. In this last case the unification stops as soon as a literal matches the head of a rule. Additional matching rules are tried if backtracking occurs;

- *Redundant deductions*: on SNIP, the ***producer-consumer*** model and the fact that all nodes are ***data collectors*** avoids redundant deductions. Known instances are stored on the producers and whenever a new consumer is accepted it immediately receives the already derived solutions. The same solution is never reported twice to the same consumer. PROLOG is not protected against redundant deductions: if the same literal appears twice on the goal, either due to backtracking or because the application of the resolution rule makes it appear twice, then it is solved redundantly.
- *Instantiation of common variables*: on PROLOG, when the resolution operation is applied, if any of the instantiated variables due to unification is shared by any other subgoal, then this subgoal will see that variable instantiated too. This makes ***parallelization*** impossible to happen, unless complicated communication of solutions among subgoals is performed. The same doesn't happen on SNIP, as all antecedents are requested by the rule node at the same time and consistency of variable instantiation is assured on rule nodes. Consequently there is potential for ***parallelization***.
- *Treatment of recursive rules*: recursive rules on PROLOG may generate infinite branches on the search tree. Only a careful ordering of the program clauses and of the antecedents of the clauses may avoid going into a loop. On SNIP, the fact that nodes are ***data-collectors*** and that the same solution is never reported twice also impedes the system to enter an infinite loop - (Shapiro, et al., 1980) and (McKay, et al., 1981).

5. Improving SNIP

5.1. Creating an IDLE queue to avoid the execution of some processes

Whenever an asserted rule node requests antecedent nodes, these requests are sent in **parallel** and all the requested nodes are scheduled on the low priority queue and executed later. This property is sometimes undesirable, since it can result in clear inefficiencies.

Let's admit that an asserted and-entailment node is requested by a consequent node and as a result requests all its antecedents. Let's also consider that the first antecedent doesn't report any solutions. Assuming this scenario, it is obvious that the and-entailment node won't produce any inferences, as a consistent report is needed from all the antecedents. However, on the current version of SNePS, all the antecedent processes that were already scheduled on the low priority queue are unnecessarily executed.

When we discussed the instantiation of common variables on section 4, we noticed that a different approach was adopted by PROLOG: antecedents were solved sequentially and if one of them fails to solve, backtracking immediately occurs.

In order to avoid this situation, we propose the creation of a third queue on the MULTI system: the **IDLE queue**. The reasoning behind scheduling a process on this queue is simple: *if none of the working processes report any solutions, then there is no need to run the idle processes because no additional inferences would be made by the rule nodes*. Scheduling processes on this queue and taking processes from the IDLE state would involve the following interactions:

- *Scheduling on the IDLE queue*: whenever a rule node sends requests to the antecedents and tries to schedule them on a queue, the MULTI system checks the inference state of the rule node and decides, for each process, whether it is appropriate to idle or set it to the working state.
- *Taking the process from the IDLE queue*: whenever the antecedents of a rule node report solutions, then we check the **updated** inference state of that rule node, i.e. the RUI set, and conclude whether we need to take another *brother* antecedent from the idle state and **reschedule** it on the low priority queue.

This modification can be implemented without affecting the internal behavior of the nodes and only with minor changes to the MULTI package.

At each moment, the number of working and idle antecedents for a certain rule node is calculated based on the type of rule node at stake and on its inference state, i.e. number of true and false antecedents proved. Table 1 shows how to calculate that number.

Node Type	Chosen RUIs to calculate pos and $negs$	Number of working antecedents
&⇒	$pos_{\&\Rightarrow} = pos(RUI)$, where $RUI: \forall_{x \in RUISet}\, pos(x) \leq pos(RUI)$	$min\{n, pos_{\&\Rightarrow} + 1\}$
$\vee\Rightarrow$	Not applicable	n
$i\Rightarrow$	Same as for &⇒	$min\{n, n - i + pos_{i\Rightarrow} + 1\}$
$\wedge\vee_{min}^{max}$	$pos_{\wedge\vee} = pos(RUI_1)$, $negs_{\wedge\vee} = negs(RUI_2)$ where $RUI_1: \forall_{x \in RUISet}\, pos(x) \leq pos(RUI_1)$ $RUI_2: \forall_{x \in RUISet} negs(x) \leq negs(RUI_2)$	$max\{min\{num\_ants, num\_ants - max + pos_{\wedge\vee} + 1\}, min\{num\_ants, min + negs_{\wedge\vee}\}\}$
$\Theta_{\text{thresh}}^{\text{thmax}}$	$pos_{\Theta} = pos(RUI)$, $negs_{\Theta} = negs(RUI)$ where $RUI: \forall_{x \in RUISet}\, pos(x) + negs(x) \leq pos(RUI) + negs(RUI)$	$min\{num\_ants, thmax - thresh + 1 + pos_{\Theta} + negs_{\Theta}\}$

Table 1 – Number of working antecedents for each rule node. The functions pos and $negs$ return the second and third elements of the RUI tuple respectively.

As Table 1 suggests, we need to iterate the entire RUI set of a rule node to reach the RUI satisfying the properties of the second column of the table. (Choi, et al., 1992) created different data structures to handle RUIs in a more efficient way than linear sets and these differences must be taken into account while iterating RUI sets.
This mechanism can be shown to reduce the number of processes executed by SNePS in several situations.

5.2. Making the search strategy versatile

Controlling the search strategy used by a theorem prover is a desirable functionality, since it may allow to find proofs using less time and space resources.
Although the breath-first search used by SNIP is useful to exhaustively explore the search tree, if we are interested in finding only one solution in the quickest time possible, then other strategies may do better.
The independence of the MULTI processes, supported by the previously discussed non-instantiation of common variables on the antecedents and by the potential to parallelization of processes, implies the correctness of the deduction process even when other sorting policies are adopted for the MULTI queues.
On section 4, we alerted to the different depth-first search strategy used by PROLOG. However, the arguments presented during the discussion about the treatment of recursive rules hint that the parameterization of the strategy cannot be easily accomplished on PROLOG as it is on SNIP.
Thus, we propose two additional search strategies for the LOW priority queue:

- depth-first search: can be accomplished assuming a LIFO scheduling policy. Instead of inserting in the end, we schedule new processes in the beginning of the queue;
- heuristic search: if we create an additional register on the processes, *QUEUE-PRIORITY*, containing an integer estimating the distance to a possible solution, then a greedy search can be induced if we sort the queue by increasing number of priority. This priority can be calculated through the invocation of an heuristic function stored on another register, *PRIORITY-CALCULATOR*.

The criteria used to create appropriate heuristics can range from global network properties computed before inference takes place - (Smith, 1989) may be an interesting reference - to local features calculated while the inference process runs. Local features may take into account the analysis of:

- the neighborhood of the node, checking the nature of the non-rule nodes that can potentially match ours: whether potentially matched nodes are asserted or have consequent arcs pointing to them;
- the potential cost of applying a rule node, checking the number of antecedents needed by the inference rule and the estimated cost of solving each of them;

6. Conclusion

On this work, we described the SNePS node-based inference mechanism implemented by SNIP and the PROLOG logic-programming system.
We presented a comparison of both systems, with the purpose of familiarizing the reader with the advantages and drawbacks of SNIP. Besides this objective, we desired to find ideas that could be implemented on this system, so as to make it more efficient and controllable.
In the end, some of these emerging ideas were implemented. We proposed a method to avoid the execution of some deduction processes without sacrificing the number of solutions found. Additionally, we argued the importance of having multiple search strategies on SNIP and proposed a method to do it.

References

Brachman, J. Ronald. 1979. On the Epistemological Status of Semantic Networks. [ed.] Nicholas V. Findler. *Associative Networks, Representation and Use of Knowledge by Computers.* s.l. : Academic Press, 1979, 1, pp. 3-50.

Choi, Joongmin and Shapiro, Stuart C. 1992. Efficient Implementation of Non-Standard Connectives and Quantifiers in Deduction Reasoning Systems. 1992.

Cruz-Filipe, Luís. 2006. *Programação em Lógica.* IST : Unpublished lecture notes, 2006.

Duffy, David. 1991. *Principles of Automated Theorem Proving.* s.l. : Wiley, 1991.

Hull, Richard G. 1986. *A New Design for SNIP, the SNePS Inference Package.* Department of Computer Science, State University of New York at Buffalo. 1986. 14.

Kowalski, R. and Kuehner, D. 1983. Linear Resolution with Selection Function. [book auth.] Jorg Siekmann and Graham Wrightson. *Automation of Reasoning: Classical Papers on Computational Logic.* s.l. : Springer, 1983, Vol. 2.

Loveland, Donald W. 1978. *Automated Theorem Proving: a logical basis.* 1st Edition. s.l. : North-Holland Publishing Company, 1978.

Martins, João Pavão. 2005. Semantic Networks. *Knowledge Representation.* Unpublished Lecture Notes. 2005, 6, pp. 213-267.

McCune, Willian and Wos, L. 1997. The CADE-13 competition incarnations. *Journal of Automated Reasoning.* 1997, Vol. 18(2), pp. 211-220.

McKay, Donald P. and Shapiro, Stuart C. 1980. *MULTI - A LISP Based Multiprocessing System.* Department of Computer Science, State University of New York at Buffalo. 1980.

—. 1981. Using Active Connection Graphs for Reasoning with Recursive Rules. 1981, pp. 368-374.

Robinson, J. A. 1965. A Machine Oriented Logic Based on the Resolution Principle. [ed.] Jorg Siekmann and Graham Wrightson. *Automation of Reasoning: Classical Papers on Computational Logic 1957 - 1966.* 1965, Vol. 1, pp. 397-415.

Schubert, Lenhart K., Goebel, Randolph G. and Cercone, Nicholas J. 1979. The Structure and Organization of a Semantic Net for Comprehension and Inference. [ed.] Nicholas V. Findler. *Associative Networks: Representation and Use of Knowledge by Computers.* s.l. : Academic Press, Inc., 1979, pp. 121-175.

Shapiro, Stuart C. and Group, The SNePS Implementation. 2004. *SNePS 2.6.1 User's Manual.* 2004.
Shapiro, Stuart C. and McKay, Donald P. 1980. Inference with Recursive Rules. *Proc. NCAI.* 1980, pp. 151-153.
Shapiro, Stuart C. and Rapaport, William J. 1987. SNePS Considered as a Fully Intensional Propositional Semantic Network. [ed.] Nick Cercone and Gordon McCalla. *The Knowledge Frontier: Essays in the Representation of Knowledge.* s.l. : Springer-Verlag, 1987, pp. 262-315.
Shapiro, Stuart C. 1978. Path-Based and Node-Based Inference in Semantic Networks. 1978.
—. 1977. Representing and Locating Deduction Rules in a Semantic Network. *SIGART Newsletter.* 1977, Vol. 63, pp. 14-18.
—. 1999. SNePS: A Logic for Natural Language Understanding and Commonsense Reasoning. *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language.* 1999.
—. 1999. SNePS: A Logic for Natural Language Understanding and Commonsense Reasoning. *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language.* 1999.
—. 1979. The SNePS semantic network procesing system. *Associative Networks.* 1979, pp. 179-203.
—. 1979. Using Non-Standard Connectives and Quantifiers for Representing Deduction Rules in a Semantic Network. 1979.
Smith, David E. 1989. Controlling Backward Inference. *Artificial Intelligence.* 1989, Vol. 39, pp. 145-208.
Socher-Ambrosius, Rolf and Johann, Patricia. 1997. *Deduction Systems.* 1st Edition. s.l. : Springer, 1997.
Sowa, John F. 1992. *Encyclopedia of Artificial Intelligence.* [ed.] Stuart Shapiro. 2nd edition. s.l. : Wiley, 1992.