PLANNING IN SEMANTIC NETWORK USING NODE NESTED GRAMMARS

Janusz Marecki
Academy of Computer Science and Management
Bielsko-Biala, Poland
jmarecki@wsi.edu.pl

Abstract: Detailed design assumptions and functional principles of the planning processes in a specific semantic network are presented in this paper. The unique hierarchic network structure, interconnected by relations on objects and categories, and equipped with node-nested grammars makes the planning a fast and effective process, similar to human's approach to the problem. Grammar productions and their respective actions allow the planning process to be reversed, therefore artificial perception of goals and plans from a raw array of actions becomes possible.

1. Introduction

Since the beginning of so called "Artificial intelligence" there has always been a question: how to implement an intelligent agent, and make it act as a human. Numerous ideas have emerged, but none of them seemed to be ideal, capable of integrating all the aspects of an intelligent behavior. Yet, some of them like Prolog or Clips definitely narrowed the gap between humans and computers. Rule Based expert systems [1] mostly written in the above mentioned tools constitute a proof, that even a best expert's reasoning can be implemented on the artificial machine.

The end of the XX century and a huge expansion of Internet focused the attention of intelligent agent developers on Semantic Networks, which are known for their clarity, expansibility and parallel processing possibility. The works of Hendrix [2] and Shapiro [3] contributed to their expressiveness and gave the basic concepts of integrating various programs with the network structure.

The new World Wide Web standard – DAML (actually created by W3C) will be based on local semantic networks, which will use unified ontology and structure. This will definitely open the new horizons for agent systems, which will move through the web, collecting knowledge in a very efficient way. The GEMO project launched recently is meant for integrating the data and knowledge distributed over the Internet.

1.1 Semantic Network Structure

Unlike many AI systems, where individual objects may exist separately and have nothing in common, the system presented in this paper is a construct of hierarchically placed objects, additionally connected by relations. An outlook of the network structure is depicted in figure 1.

All objects descend from a base class (Class Object), and deeper we go the better specified objects (classes) appear. It should be noticed, that an object can be inherited from many objects, so the structure of the network is not a classical tree, but a directed acyclic graph. Physical objects are the existing representatives of classes, but there are also abstract representatives of classes to allow the network to posses the knowledge about the abstract things.

There are practically no limits for relations placement in the network. Every relation can connect two network nodes (existing nodes, abstract nodes, or whole classes). Relations work not only on objects "attached" to it, but also on every descendant of an attached object, therefore an information surplus problem can be limited. The propagation of a relation is directed by the hierarchical links of the network.

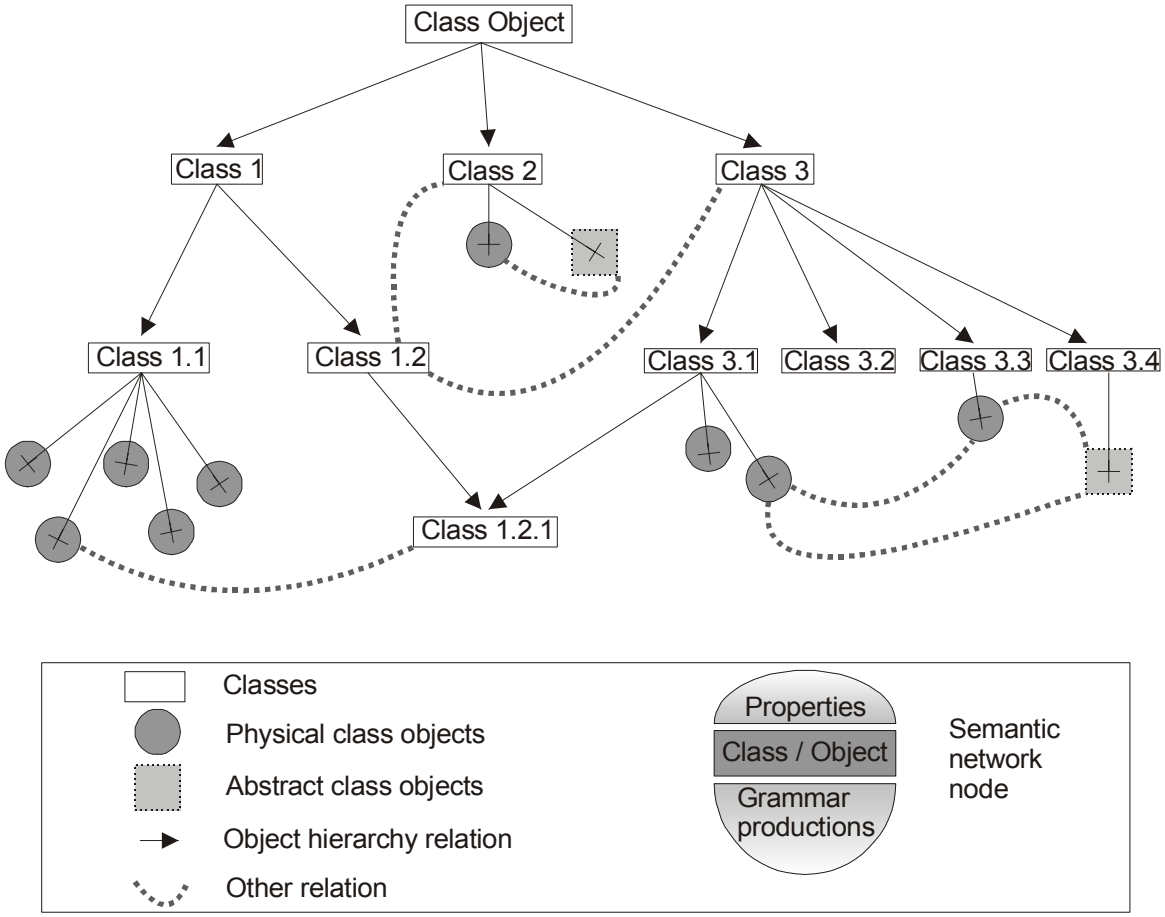


Figure 1 Semantic Network structure

As depicted in figure 1, every network node consists of the set of node's properties, and node's grammar productions. Node's properties function in the same way as relations do, they are always attached to the object they belong to, and they are propagated to the object's descendants according to the hierarchical links of the network. Node's grammar productions also propagate in the same way, and their syntax is introduced below.

1.2 Node-nested grammars

The planning process will be directed both by hierarchical links of the network, and node-nested grammar productions. The grammar productions will have a specific schema, depicted below: (NT = Non Terminal symbol)

$NT_0(Objects_0) \rightarrow$	$NT_1(Objects_1)$ … $NT_n(Objects_n)$	{SN Updates}	(Actions)

Let us consider the first part of the above production: $NT_0(Objects_0)$. There is a certain not-terminal symbol NT_0 associated with an array of objects $Objects_0$. This production will be used if the planning algorithm in a current state has a goal $NT_0(Objects_0)$. Suppose the production is chosen, than in the next state, the planning algorithm will have no longer the

goal $NT_0(Objects_0)$, but the set of new goals $NT_1(Objects_1)$ … $NT_n(Objects_n)$. The grammar productions that could possibly match the new sub-goals will be searched in the sub-graphs starting from the nodes associated with the first object of each sub-goal. If all the sub-goals of the production are satisfied, than Semantic Network updates can be made depending on the set {SN Updates}. Finally if the production if fulfilled and all its sub-goals satisfied, the agent can carry out actions from the array Actions.

It is important to notice, that there can be any order of the new production's goals, and therefore before the algorithm fails to find a plan, it must check every permutation of goals.

When the algorithm starts to search for $NT(Objects_k)$ it must:

1. Try to search $NT(Objects_k - \{Obj\})$ in the node pointed by *Obj* - the first object of $Objects_k$.
2. If a goal $NT(Objects_k - \{Obj\})$ **unifies** (the unification process will be described later) with a property of the object *Obj*, than the goal is immediately satisfied, and the search for other goals can begin.
3. If a goal $NT(Objects_k - \{Obj\})$ unifies with the left side of one of the grammar productions in the node *Obj*, than this production should be used for a recurrent call of the algorithm.
4. If a goal $NT(Objects_k - \{Obj\})$ does not unify neither with the set of properties nor with grammar productions of *Obj*, than it must be looked for in each of *Obj*'s children (see hierarchical links of the network)
5. If a goal $NT(Objects_k - \{Obj\})$ is not yet found, the goal $NT(Objects_k)$ cannot be satisfied.

If a goal $NT(Objects_k)$ has not been found, than a new permutation of production's goals should be taken. If all permutations are analyzed and the goals are not satisfied, than this production becomes useless for the current step of the planning algorithm.

It must be mentioned, that there should be two versions of the algorithm to satisfy *goals overlapping problem*. The problem appears if at least two goals cannot be satisfied without the partial exploration of both of them. In other words, goals cannot be satisfied one after another. For most simple plans this problem can be neglected, but the example below shows when the standard algorithm fails:

Facts: not C, not D
Goals: A, B.
Productions: $A \rightarrow C \mid Updates(D)$
$B \rightarrow D \mid Updates(C)$

In the next section, two versions of the planning algorithm will be presented: the first, that doesn't tackle goals overlapping problem, and the second, much slower, with overlapping goals management.

2. Planning

To avoid ambiguity we denote each plan discussed in this paper as an ordered sequence of operations with restrictions associated with each operation.

2.1 Planning problem

The figure 2 shows an example of how the algorithm without overlapping goals management finds a specific plan. For the reasons of clarity, a dot in the production's relations will be a substitution for the name of an object in which the production is located. For example: if an object Obj has a production A(.)→ B(.) it is equivalent to A(Obj)→ B(Obj).

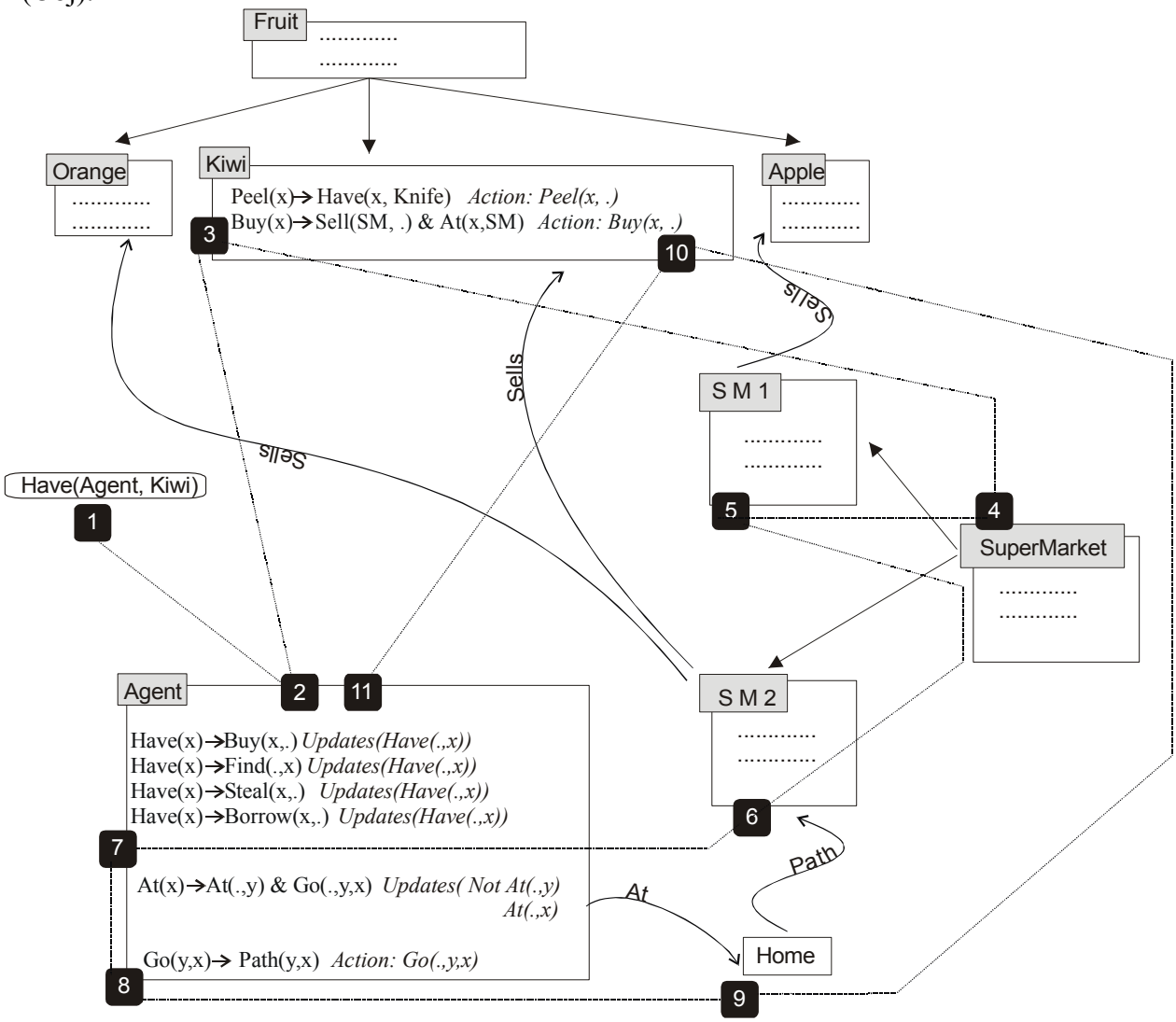


Figure 2 Example of a planning process.

The planning process goal is : Have(Agent, Kiwi) (step 1). The algorithm explores this goal by trying to find Have(Kiwi) in the Agent object. Since Have(Kiwi) unifies with the first grammar production in the node Agent, further exploration is possible (step 2). Now, the algorithm has a new goal: Buy(Kiwi, Agent) and it explores the Kiwi object (step 3), where it finds a production beginning with Buy(x). After a local unification takes place, the new goals are: Sell(SM, Kiwi) & At(Agent, SM).

Now, the algorithm explores the sub-graph starting from the node SuperMarket in a search for a relation Sells(Kiwi) (steps 4, 5, 6). After finding that SM 2 sells Kiwi, SM 2 substitutes all occurrences of SM in the set of local goals, which now reduces to At(Agent, SM 2). Again, the algorithm explores the Agent object to find the unification for At(SM 2). It succeeds in step 7, and after local unification, the new goals are: At(Agent, y) & Go(Agent, y,

SM 2). By exploring the object Agent, the algorithm finds a property At(Home), and y unifies with Home leaving only one goal Go(Agent, Home, SM 2) to satisfy. The appropriate production Go(y,x) is found in the Agent object (Step 8), and there is now a new goal Path(Home, SM 2). The exploration of the object *Home* (Step 9) is successful, because a property Path(SM 2) has been found.

Since there are no more goals, the algorithm backtracks and makes an action **Go(Home, SM 2)**. Now, the production in step 7 is finished, so Semantic Network update At(Agent, SM2) is applied. The algorithm jumps back to step 3 (or 10) and since this production is finished it makes an action **Buy(Agent, Kiwi)**.

Further backtracking moves the algorithm to step 11, where an update to the network: Have(Agent, Kiwi) finishes the planning process.

2.2 Planning algorithms

Before we show the source code of the planning algorithms, detailed mathematical notation must be introduced:

2.2.1 Index of symbols:

$G(n,i)$ – „i-th" gramatic production embedded in a node "n".
$\#G(n,i)$ – Number of relations in a grammar production G(n,i).
$R^{G(n,i)}{}_j$ – „j-th" relation in a grammar production G(n,i).
$R^{G(n,i)}{}_0$ – Left side of a grammar production G(n,i).

$U^{G(n,i)}$ – Set of network updates associated with a grammar production G(n,i).
U – Set of temporary network updates.
$A^{G(n,i)}$ – Array of actions associated with a grammar production G(n,i)
A – Array of temporary actions.
C^n_i – i-th child of a node n.
P^n_i – i-th property of a node n.
$\#C^n$ – Number of children of a node n.
$\#P^n$ – Number of properties of a node n.
$\#G^n$ – Number of grammar productions of a node n.

$\{\varphi_1, \varphi_2, \varphi_3 \ldots \varphi_n\} \in F$ – Set of goals for the algorithm (Relations with objects)
F_0 – Starting goal (The planning result)
$\#F$ – Number of items in the set F
$\#\varphi_i$ – Number of arguments in a relation φ_i.
$\varphi arg_{i,j}$ – j-th argument in a relation φ_i.
$\varphi_i - \varphi arg_{i,1}$ – Relation φ_i without the first argument
$Id : Objects \rightarrow Integers$ – Function mapping Object names to their respective numbers.

The planning algorithms will use a sub-procedure that generates permutations of the arguments in the set F. Given $\{\varphi_1, \varphi_2, \varphi_3 \ldots \varphi_k\} \in F$, we will call $(\varphi_{p(i,1)}, \varphi_{p(i,2)}, \varphi_{p(i,3)} \ldots \varphi_{p(i,k)})$ an i-th permutation of F.

2.2.2 Unification

There is also a procedure Alter($x \rightarrow y$ in z) in the algorithm. It will simply substitute each appearance of x for y in the set z. The Alter procedure will be used to apply to the goals

the set of local **unifications**. For simplicity let's call the variables symbols x, y, z etc. All the variables can take any value, but every appearance of a variable in a production must be associated with the same value. Consequently if there is a goal Rel(x,y,Shop) and a property Rel(Home,Home, z), where Home and Shop are not variables, than the unification operation can be successful. In this case, the set of unifications is as follows: Ω = {x→Home, y→Home, z→Shop}.

The unify function syntax is:

UNIFY(Rel_1, Rel_2, Ω) = true if there exist an unification Ω for Rel_1 and Rel_2.

UNIFY(Rel_1, Rel_2, Ω) = false Rel_1 can't unify with Rel_2. Ω is empty.

The function code is depicted below:

```
UNIFY(Rel1(Objects1), Rel2(Objects2), Ω)  returns Boolean

        Ω := ∅
        If  (#Objects1) ≠  (#Objects2) then return false
        For i:=1 to #Objects1 do
                If not variable(Objects1,i) & not variable(Objects2,i) then
                        If Objects1,i ≠ Objects2,i then
                                Ω := ∅
                                return false

                If not variable(Objects1,i) & variable(Objects2,i) then
                        Ω := Ω ∪ { Objects2,i → Objects1,i }
                        For j:= i to  #Objects1 do
                                If Objects2,j = Object2,i then Objects2,j := Objects1,i

                If variable(Objects1,i) & not variable(Objects2,i) then
                        Ω := Ω ∪ { Objects1,i → Objects2,i }
                        For j:= i to  #Objects1 do
                                If Objects1,j = Object1,i then Objects1,j := Objects2,i

                If variable(Objects1,i) & variable(Objects2,i) then
                        Ω := Ω ∪ { Objects1,i → Objects2,i }
                        For j:= i to  #Objects1 do
                                If Objects1,j = Object1,i then Objects1,j := Objects2,i
        Return true
```

So far, the function Alter(x → y in z) was analyzing the set z only once. Now, we would like to extend it to the function ALTER(F, Ω) which would apply the substitutions from Ω to the set of goals F, until there is no change for F. The need to repeat the application of substitutions Ω to F is due to the fact, that Ω contains substitutions of the form x → y, where both x and y are variables. For some variables, they can substitute many times before there is finally a substitution which gives them a specific value.

```
        ALTER(var F, Ω)      (F will be the altered set of goals)
                Do
                        F' = F
                        For each u∈Ω do Alter(u in F)
                While F' ≠ F
```

2.2.3 The algorithm without overlapping goals management:

Starting parameters:

$A := \varnothing$
$U := \varnothing$
$\Omega := \varnothing$
PickSubGoal(F_0, U, A, Ω)

```
PickSubGoal(F, var: U, A, Ω)  returns boolean
LocalVariables: Result (true/false), F_old, U_old, A_old, φarg_old , Ω_old

    //generate all permutations
    For i:=1 to (#F)! do
        F_old := F
        U_old := U
        A_old:= A
        Ω_old := Ω
        Result = true

        //loop for every relation in the permutation
        For j:=1 to (#F) do
            Ω := {}
            φarg_old := φarg_p(i,j),1
            If Explore(φ_p(i,j) – φarg_p(i,j),1 , φarg_p(i,j),1 , U, A, Ω)= true then      (*)
                If φarg_old ≠ φarg_p(i,j),1 then
                    Ω := Ω ∪ {φarg_old→φarg_p(i,j),1}
                ALTER(F, Ω)
            Else
                Result = false
                Exit {For j:=1 to (#F) }

        // If all relations successfully explored, end PickSubGoal routine
        If Result = true  then
            Return true
        F := F_old
        U := U_old
        A := A_old
        Ω := Ω_old

    //If all permutations analysed and goals not eleminated then return false
    Return false
```

The PickSubGoal function calls the Explore function depicted below:

```
Explore(Rel, var : Obj, U, A, Ω)  returns boolean                    (**)
LocalVariables: Obj old , Ωold, Ωtemp

    For i:=1 to #P^id(Obj) do
        If UNIFY(Rel , P^id(Obj)_i , Ωtemp )  then
            Ω := Ω ∪ Ωtemp
             return true

    For i:=1 to #G^id(Obj) do
        Ωold := Ω
        If UNIFY(Rel , R^G(id(Obj),i)_0 , Ωtemp)  then
            Ω := Ω ∪ Ωtemp
            If PickSubGoal(ALTER(⋃_{j:=1}^{#G(id(Obj),i)-1} R^G(id(Obj),i)_j, Ω),U, A, Ω) = true
            then                                                     (***)
                A := A + A^G(id(Obj),i)      //concatenation of actions
                U := U ∪ U^G(id(Obj),i)
                ALTER(U, Ω)
                ALTER(A, Ω)
                 return true
            else Ω := Ωold

    For i:=1 to #C^id(Obj) do
        Obj old := Obj
        Obj := C^id(obj)_i
        If Explore(Rel, Obj, U, A, Ω) = true then return true            (****)
        Else Obj := Obj old

    Return false
```

2.2.4 The algorithm with overlapping goals management

The planning algorithm with overlapping goals management is a small modification of the previous algorithm, but introduces a huge difference in the planning process. Because of the fact, that different goals may overlap one onto each other, there is a need to analyze conditions of all goals when exploring each individual goal. Therefore, recurrent calls of the PickSubGoal function must involve not only the explored goal, but a whole set containing the explored goal and the other goals. Consequently the search space becomes bigger, and the algorithm is significantly slower.

The difference in algorithm source is as follows: Explore routine is now equipped with an additional parameter F, which keeps track of all goals. Whenever there is a need to call recursively PickSubGoal routine from inside the Explore routine, we need to pass all actual goals, not only the goals from an explored grammatical production. The changes to the previous algorithm are:

(*)

If $\text{Explore}_2(F-\{\varphi_{p(i,j)}\}, \varphi_{p(i,j)} - \varphi\text{arg}_{p(i,j),1}, \varphi\text{arg}_{p(i,j),1}, U, A, \Omega) = \text{true}$

(**)

$\text{Explore}_2(F, Rel, var : Obj, U, A, \Omega)$ returns boolean

(***)

If $\text{PickSubGoal}(\text{ALTER}(F \cup \bigcup_{j:=1}^{\#G(id(Obj),i)-1} R^{G(id(Obj),i)}_j, \Omega), U, A, \Omega) = \text{true}$ then

(****)

If $\text{Explore}_2(F, Rel, Obj, U, A, \Omega) = \text{true}$ then return true

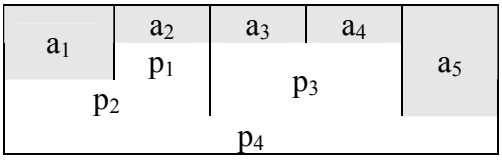
3. Conclusions

3.1 The reverse process

The reverse process, although not a trivial one, is an interesting approach to equip the agent with some important intelligent behaviors. With this process, the agent could:

- say what somebody or something was doing (or was intending to do) by observing its behavior. For example, by knowing that somebody buys candles, red wine an a turkey in the evening it may suppose a romantic supper,
- be taught how to make human defined plans. For example, a human tells an agent only the actions, and their final goal, and the agent can properly classify the actions in its network nodes,
- learn the actions it does not understand. Those actions can be easily described using node-nested grammar productions, and existing semantic network ontology.
- compare one plan with another, find similarities and apply modifications.

The reverse process should be capable of creating complete planning trees from an array of productions, for example, given the actions a_1, a_2, a_3, a_4, a_5 it should find the productions p_1, p_2, p_3, p_4 that lead to them.



In the complete planning tree:

- Action sequence should be saved.
- Semantic network updates made by sub-goals cannot exclude consecutive actions in the array of actions.
- The unifications must be unambiguous for every production in the planning tree.

3.2 Prospects

"Computers are machines and machines exist for one purpose
and that purpose is to serve people"

Prof. Alphonse Chapanis

This paper introduces a specific approach demonstrating how to create intelligent agents simulating human behavior. Although only a simple example of such an agent was presented it is likely, than one day all mechanical human behaviors could be performed by artificial units.

Recent trends meant to unify the ontology of the Internet will undoubtedly open new horizons for intelligent agent developers. Systems, like the one presented in this paper could play a key role in tomorrow's information society, where information is accessible both for humans and computers.

4. Bibliography

1. Niederliński, A. (2000) Regułowe systemy ekspertowe, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice.
2. Hendrix, G. G. (1975). Expanding the utility of semantic networks through partitioning. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 115-121, Tbilisi, Georgia.
3. Shapiro, S. C. (1979). The SNePS semantic network processing system. In Findler, N. V., editor, Associative Networks: *Representation and Use of Knowledge by Computers*, pages 179-203. Academic Press, New York.
4. Marecki, J. (2001) Metody Sztucznej Inteligencji, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice.
5. Marecki, J. (2003) Intelligent agent based on semantic network, monograph, Academy of Computer Science and Management.