Semantic Networks And Intelligent Agents

Janusz Marecki

Contents

Index of figures

Chapter I Introduction

I. 1 Origin

Classification of problems

In general, we can say that technical, economical, organizational and computational problems can be classified as either:

- Deterministic problems
- Probabilistic problems
- Unique problems

We assume that our goal is not to introduce strict borders between the above classes, but to present characteristics for problems belonging to those classes.

Each deterministic problem has its structure and parameters well determined. Consider for example a well known traveling salesperson's problem (TSP). It consists in finding the shortest path through N different cities starting from the city 0 and finishing in the same city. The structure of this problem can be associated with the network of roads, whereas their lengths can be seen as problem parameters. This problem is hard to solve – from the point of view of the computational time. Although all parameters are well determined and we can start calculations from any moment, for big N values even the quickest computer would need more than a lifetime to find a solution.

The common point of all deterministic problems is that every parameter of the problem is known and we can begin to calculate the solution at any time. However, it does not mean that these problems are easy, though there are many algorithms which solve them.

Probabilistic problems either do not have a determined structure or their parameters remain unknown. Consider the problem of a fire department or emergency service functionality. There is no point in solving the scheduling problem for these institutions because the structure of this problem (set of operations with their locations) and its parameters (time of operation, time of transport) are unknown. However, some fire department or emergency service operations are well known. As a result, decision tables can be created to help to choose the rational decision in real-time conditions, depending on the situation.

Although probabilistic problems do not have their structures or parameters determined, we can define for them the set of permissible structures. For each structure there is a possibility to determine a rational (not optimal) solution – a decision. Relation on the set of structures and decisions creates so called decision tables.

The set of unique problems is empty! These are the problems which appear randomly - moreover, before their appearance they have neither a determined structure nor a known set of parameters. Finding solutions for unique problems consists in:

- Defining the structure
- Defining the parameters
- Defining the algorithm

A solved unique problem is no longer unique. A common point of all unique problems is that they are not defined before. The moment of their appearance is unknown. Thus, those are new problems which have to be solved quickly, minimizing the use of various resources.

Unique problems

New technical, economical, organizational or computational problems continuously appear in our life. Generally, they are represented by complexes of operations conditioned by time, space and other environmental factors. These problems are not defined before. There are no ready-to-use algorithms capable of solving them. Some new problems have to be coped with under the pressure of time. All these problems will be called unique.

Below, some of the unique problems will be presented. One day each of them was new, now none of them is still unique. The aim of this survey is to prove the proposition, that:

"Unique problems are solved through analogies to the problems solved before"

Consequently there are analogous (with the same structure and parameters type) problems which have been solved before.

Problem 1

The meltdown of the Tschernobil nuclear power plant. No-one suspected that such a problem would ever occur. The structure and parameters of this task were unknown before the disaster. The problem was coped with under serious time pressure. The discussion if the problem was being solved rationally remains up to this day.

Problem 2

The sinking of the TITANIC. No-one suspected that such a problem would ever occur. Before the disaster, structure (location of this tragedy) and parameters (meteorological conditions) of a task which could save the ship and its passengers were unknown. The problem was coped with under serious time pressure. The discussion if the problem was being solved rationally remains up to this day.

Problem 3

The space shuttle disaster. Space shuttles are objects with a unique structure. Nobody expects such disasters. The complexes of operations meant to save the shuttle from the catastrophe are unknown before the crash. Due to the pressure of time, some shuttle problems had not been solved. The discussion if other problems were being solved rationally remains up to this day.

Owing to extreme costs of space shuttles, their flight is tracked by computer systems, which try to identify a unique problem with an analogous problem that happened before.

Problem 4

The terrorist attack on the World Trade Center in New York. No-one suspected that such a problem would occur. Before the tragedy, the structure (the set of airplanes with their paths) and parameters (time, flight parameters) of a problem which had to be solved had been unknown. The discussion if the problem was being solved rationally remains up to this day.

Problem 5

The terrorist attack on the NORD-OST Moscow theatre. No-one suspected that such a problem would occur. Before the tragedy, the structure (the set of terrorists and their location) and parameters (terrorists' armament) of a problem that had to be solved had been unknown. The discussion if the problem was being solved rationally remains up to this day.

There is a common belief, that terrorist attacks are unique problems. However, some of them, like hijackings are no longer unique. For these problems, standard decision tables have been created. If there is a decision table for a unique problem, then this problem becomes probabilistic.

However, for dealing with other unique problems the so called brainstorming headquarters are being created. Brainstorming headquarters use the knowledge about analogoue problems to create a solution to each new unique problem. Most brainstorming headquarters work under the pressure of time.

Problem 6

The appearance of a new computer virus on the Internet. This fact usually emerges when companies indicate the first losses of their computer resources. The structure and parameters of this problem remain unknown until the virus strikes. The discussion about anti-virus protection tools is always a present issue.

Every day the new viruses appear on the Internet. Companies producing anti-virus software are forced to update their database of known viruses every few hours. By looking at the list of recent hardware / software virus protection solutions we may state that each attack of a hacker can constitute a unique problem.

From the above survey the following origin of this paper can be inferred:

- In various domains of life, there is a continuous appearance of new, unique problems, which have to be solved under the pressure of time.
- Unique problems are being solved analogously to similar problems that have appeared earlier.
- There is a need for a quick access to the knowledge about problems analogous to unique problems.
- The system based on the set of intelligent agents associated with specific life domains could be put into life.
- For such a system, the agent communication platform has to be created. The system headquarters should be able to gather the knowledge passed by widespread agents.
- There is a need to create an operating system for the network of multi-domain agents residing in various point of the network. This system should allow:
  - To create / destroy / clone agents associated with different parts of the network
  - To transmit the information from all agents to the headquarters, and back from the headquarters to individual agents.

I. 2 Prospects

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 [14] 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 20th century and a huge expansion of the 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 [7] and Shapiro [21] contributed to their expressiveness and gave the basic concepts of integrating various programs with the network structure.

The new World Wide Web standard – DAML[23] (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 [22] launched recently is meant to integrate the data and knowledge distributed over the Internet. The grouping algorithms [5] [11] will undoubtedly play a key role if this knowledge is to be stored in a classified, efficient way. To prepare a solid ground for all those tools, the unified network ontology [8] and knowledge engineering [9] should be developed.

It is highly unlikely, that one artificial intelligence method could simulate all possible intelligent behavior. Rather, it seems that hybrid expert systems [6] capable of integrating various AI tools can finally simulate human-like reasoning. These hybrid expert systems can efficiently classify problems and pass them over to smaller, more specialized AI tools.

Groups of agents collecting the information from the Internet can become one day a fully functional artificial intelligence unit. It is therefore important to develop a central command for each such a unit. This central command (Agent Operating System) should be capable of understanding human language.

The Internet will be the most probable environment for these groups of agents. Because of the fact that semantic networks could easily describe various internet security and business solutions, their role in the future information society is unquestionable [15][16][17][18][19].

This paper begins with the implementation of a common sense knowledge planning system [20]. Thanks to the use of semantic network node nested grammars, presented in Chapter 2, we obtain not only a powerful planning system, but also a tool that interprets the environment actions and provides them with meaning, sense and future predictions. It is shown how the planning system can emulate inference rules and the agent's communication platform.

Moreover, more complex plans can be seen as simple algorithms. The last Chapter demonstrates how to write Semantic Network algorithms distributed all over the network. Since those algorithms will not have their structure and parameters determined, we may consider them as solutions for new unique problems.

I. 3 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.

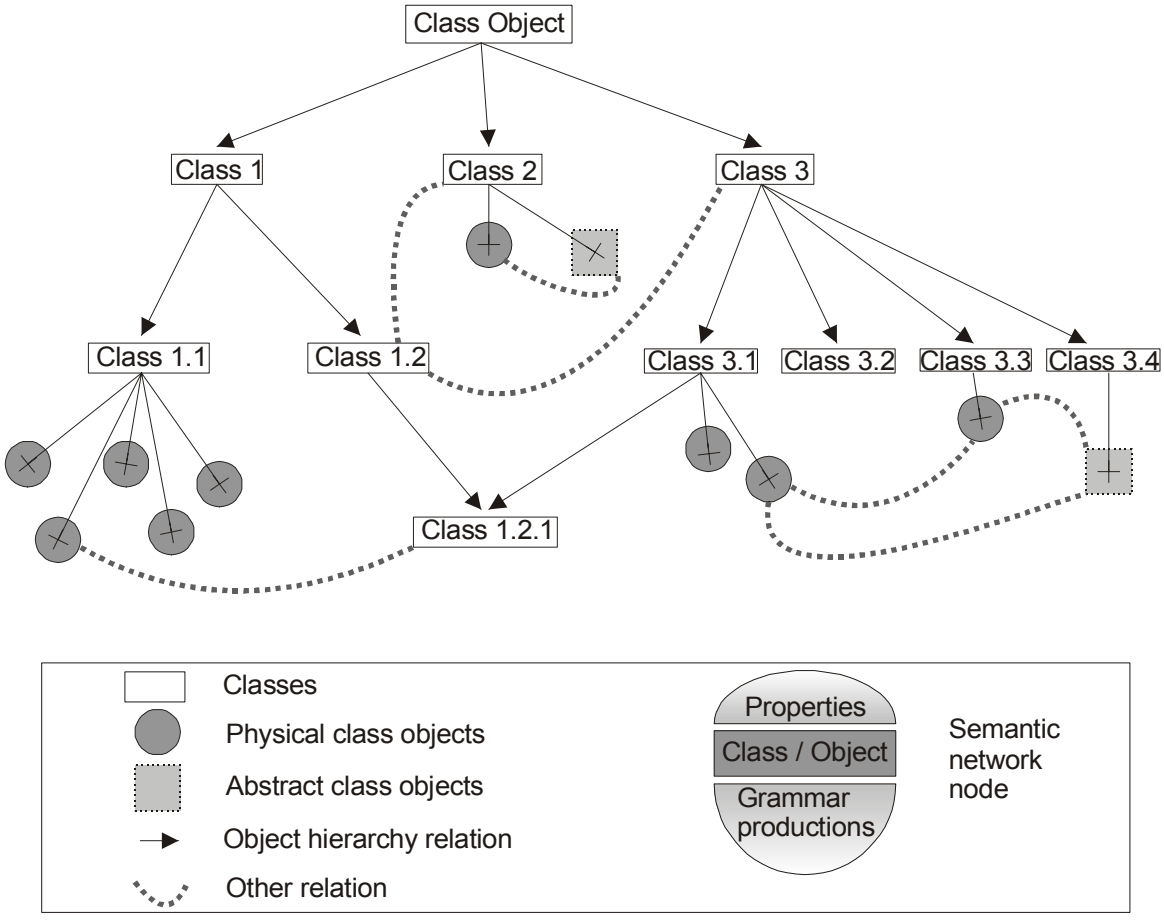


Figure 1 Semantic Network structure

As depicted in figure 1, every network node consists of a set of node properties, and node grammar productions. Node 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 descendants according to the hierarchical links of the network. Node grammar productions also propagate in the same way, and their syntax is introduced in the next chapter.

The semantic network's integral parts: the inference system, grouping system, and time/space system will be presented in the following chapters as they constitute other challenging subjects for development.

I. 4 Semantic Network management

In order to keep the semantic network up-to-date, there must exist some mechanisms responsible for applying changes to the network structure. These mechanisms can be divided into three categories:

- Human interactions.
- Perception & recognition.
- Programmed reasoning.

Human interactions

Whenever the Semantic Network interacts with a human, the latter can smartly update the network. At the beginning of the network's existence, it is the human being who designs the classes, objects, relations and grammar productions. Although the Semantic Network agent can start acting independently from that point, the human remains always the most intelligent source of information for this autonomous system.

When looking at a structure of objects / classes / relations he can identify the errors and improve the network efficiency and accuracy by applying updates to the knowledge base. Moreover, a human can transfer additional knowledge to the system concerning different world domains. He can teach the agent how to perform new plans, how to deal with unknown objects and how to understand new relations.

After a human adds some new facts to the network, the reasoning and perception mechanisms can infer from such a network additional knowledge about the environment. As an example let us consider the following situation: the human tells the agent that there is a new object in class 1.2.1 depicted in figure 1. Thanks to that fact, the pre-programmed reasoning mechanism can infer that this new object and an object from class 1.1 are linked by some relation.

Finally it is up to a human to modify the agent's scheduler (described in the next chapter) which represents the priority queue for various system goals.

Perception & Recognition

The Semantic Network perception and recognition element is the second mechanism which allows the network to undergo dynamic changes. Although the precise list of the agent's preceptors and sophisticated methods of picture/sound recognition are not presented in this paper, we will demonstrate their applicability to the network.

The agent associated with the semantic network performs a bunch of plans registered in its scheduler. Because of the fact that the agent can move within time its perception & recognition element should always be active, scanning the environment for possible knowledge base updates. We assume that the agent can percept picture and sound although it can also incorporate thermometers and smell sensors. All those perception elements are somehow combined together to answer the following question:

Does the recognized element E have a property P?

As it is shown later this question is elementary when the agent analyses an unknown object E.

Due to the fact that pictures, sounds, temperatures smells and other factors are all classified as objects, they belong to the same Semantic Network whose root node is Class Object. The recognition element should thus start by analyzing the new element E from the Semantic Network root node. The recognition element's goal is either:

- to identify the new element as being one of the existing objects in the network and return the category of that object,
- to attach the new element to the most accurate category for it and then return this category.

There is a hidden trap associated with the object recognition element. Since it will try to identify the new element by analyzing all its properties, it will surely come across the properties like time and space. Suppose it turns out, that the new element is exactly the same as the existing one except for the time and space properties. In this case, the recognition element will have to decide if this new element is in fact the existing one, or a new one. For example, when the agent finds a glass at a location where the glass was not supposed to be, then he can either classify it as a new glass, or treat it as an old one, that somebody brought to this place. If he classifies the glass as a new one, then the agent's Semantic Network should have the new object attached to the "Glass" category. If, on the other hand this glass already exists, the agent's Semantic Network should not only update the location property of this glass but also remember when this glass was located before. The in depth analysis of this process will be covered in Chapter "Time and Space".

Let us focus on the recognition algorithm itself. Function Add(*Element*, *Class*) has a task to either find the object *Element* in a sub-graph emerging from the node Class or to find the most accurate category for *Element* and then attach it to this category. In both cases the Add function returns the category *Element* is attached to. If the recognition element knows some general category *SomeCategory* the *Element* belongs to, then it can start directly from the node representing this category: Add(*Element*, *SomeCategory*), otherwise it must with the Semantic Network root node: Add(*Element*, *Class Object*). If the recognition algorithm starts from *SomeCategory*, then *Element* directly inherits all the properties and grammar productions for all nodes on various paths from *Class Object* to *SomeCategory*.

When being in the *Class* category, the Add function first applies to all properties and local inference rules to the *Element* (since the *Element* reached this category, it should have all properties and inference rules' resulting facts with it). Then the algorithm explores every sub-category *SC* of the *Class* node, and analyses all *SC*'s properties. If the *Element* possesses all *SC* properties, it means that *SC* is a less general category for the *Element* and the function Add(*Element, SC*) can be called. If all *Class*'s subcategories are explored, and none of them has all the properties coherent with the *Element*, then the algorithm checks all *Class*'s objects in order to find an object identical to the *Element*. Next, the Add function will analyze *Class*'s objects properties and compare them with the properties of the *Element*.

Here, we would like the algorithm to perform properties comparison in both directions. Given a *Class*'s object *O*, the algorithm should first check if the *Element* has all properties of *O* and then if *O* has all properties of the *Element*. If all of those properties are the same (except for the space property) it can be stated that the *Element* has been found in the network and the algorithm can return the category *Class*. If there are no objects identical to the *Element* then the *Element* is attached to the current category by calling the Attach(*Element, Class*) procedure.

This procedure can be default (simple network node attachment), but it can be individual for each category. Some properties of *Class*'s objects can have the values of other

artificial intelligence tools like the neural networks. Those tools are called within the body of the Attach procedure and their input values can be object's other properties.

The object recognition function written in the pseudo-code language is depicted below:

```
Function Add(Element, Class) returns ElementClass
Local variable: CoherentProperties : boolean

    Apply Class's inference rules for the Element,
    In order to derive some of its properties

    for each SC, a sub-class of Class do
        CoherentProperties = true
        For each SC's property p do
            If  Element does not have the property p then
                CoherentProperties = false
        If CoherentProperties = true then
            Return Add(Element, SC)

    for each O, an object of Class do
        CoherentProperties = true
        For each O's property p (except for the space property) do
            If  Element does not have the property p then
                CoherentProperties = false
        For each Element's property p (Except for the space property) do
            If  O does not have the property p then
                CoherentProperties = false
        If CoherentProperties = true then
            Update O's time and space properties if necessary
            Return Class

    Attach(Element, Class)
```

In the Chapter "Time and Space" it is shown how to implement the instruction:

Update O's time and space properties if necessary

Programmed reasoning

The last mechanism which can perform the updates to the Semantic Network is represented with node nested grammars. As it is shown in the next chapter, each grammar production consists of a set of Semantic Network updates. Those updates are stored locally when the planning process does its job, and take place immediately after it creates a plan.

Moreover, the grammar productions are also used to simulate inference rules, which clearly influence the facts in the Knowledge Base (see Chapter "Inference").

Chapter II Planning

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 between 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.

II. 1 Node-nested grammars

As mentioned above, the planning process will be directed both by hierarchical links of the network, and node-nested grammar productions. The grammar productions has 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 marked 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, then 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, then 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 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 an $NT(Objects_k - \{Obj\})$ goal **unifies** (the unification process is described later) with a property of the object *Obj*, then the goal is immediately satisfied, and the search for other goals can begin.
3. If an $NT(Objects_k - \{Obj\})$ goal unifies with the left side of one of the grammar productions in node *Obj*, then this production should be used for a recurrent call of the algorithm.

4. If an NT(Objects$_k$ – {*Obj*}) goal unifies neither with the set of properties nor with grammar productions of *Obj*, then it must be looked for in each of *Obj*'s children (see hierarchical links of the network)
5. If an NT(Objects$_k$ – {*Obj*}) goal is not found yet, the goal NT(Objects$_k$) cannot be satisfied.

If an NT(Objects$_k$) goal has not been found, then a new permutation of production goals should be taken into account. If all permutations are analyzed and the goals are not satisfied, then 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 the *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 of the simple plans this problem can be neglected but the example below shows the case when the standard algorithm fails:

Facts: not C, not D
Goals: A, B.
Productions: A→C | *Updates*(D)
B→D | *Updates*(C)

In the next section two versions of the planning algorithm are presented: the first, that does not tackle the goals overlapping problem and the second, much slower, with overlapping goals management.

II. 2 Planning problem

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 relations is a substitution for the name of an object in which the production is located. For example: if object Obj has production A(.)→ B(.) it is an equivalent to A(Obj)→ B(Obj).

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 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 SuperMarket node in 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 the local unification new goals can be defined: At(Agent, y) & Go(Agent, y, SM 2). By exploring the Agent object, 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 *Home* object (Step 9) is successful, because property Path(SM 2) has been found.

Since there are no more goals, the algorithm backtracks and performs **Go(Home, SM 2)** action. 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 the **Buy(Agent, Kiwi)** action.

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

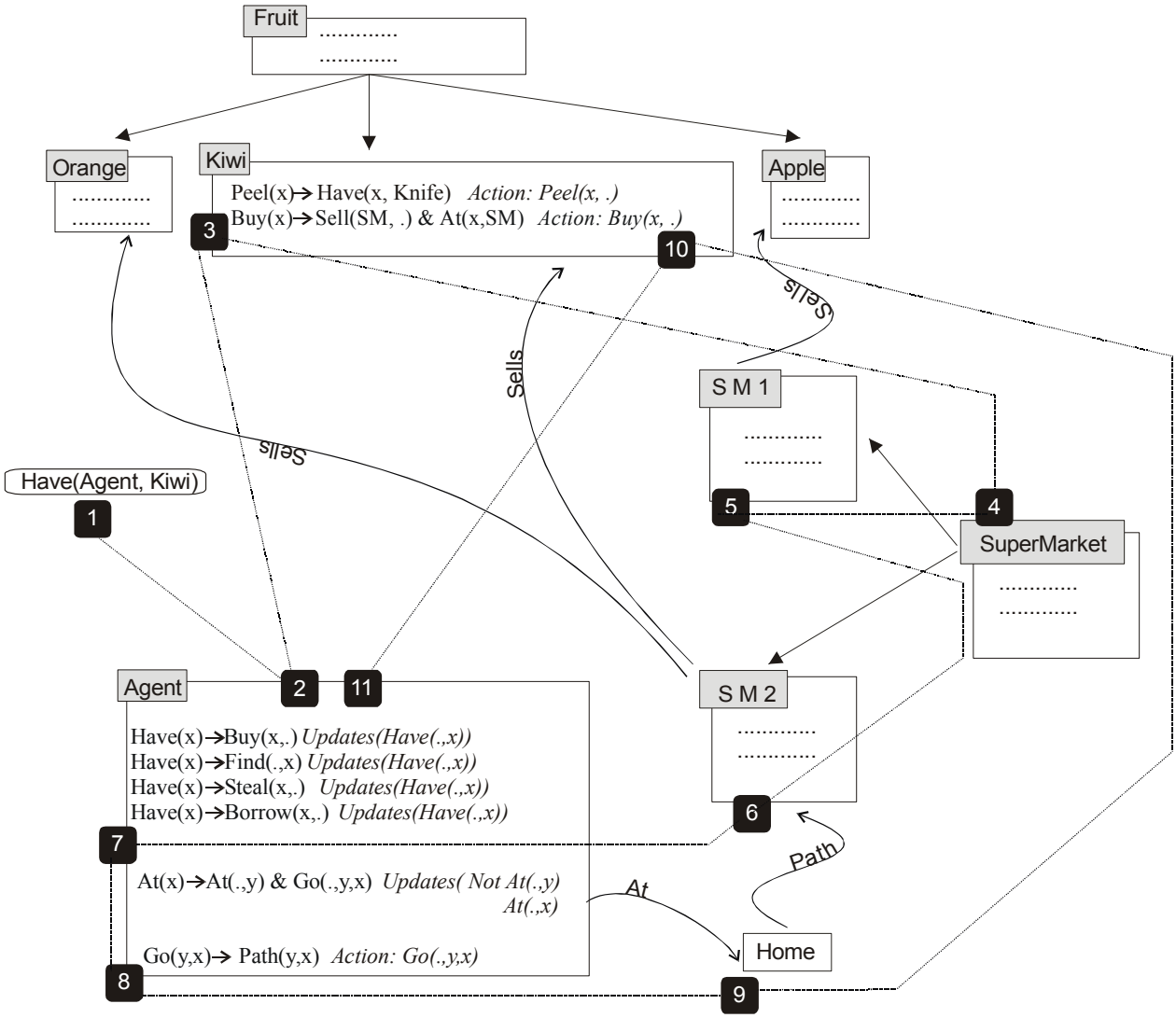


Figure 2 A planning process example.

II. 3 Planning algorithms

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

Index of symbols:

$G(n,i)$	– i-th gramar production embedded in a node *n*.
$\#G(n,i)$	– Number of relations in grammar production G(n,i).
$R^{G(n,i)}{}_j$	– j-th relation in grammar production G(n,i).
$R^{G(n,i)}{}_0$	– Left side of grammar production G(n,i).
$U^{G(n,i)}$	– Set of network updates associated with grammar production G(n,i).
U	– Set of temporary network updates.
$A^{G(n,i)}$	– Array of actions associated with grammar production G(n,i)
A	– Array of temporary actions.
C^n_i	– i-th child of node *n*.
P^n_i	– i-th property of a node *n*.
$\#C^n$	– Number of children of node *n*.
$\#P^n$	– Number of properties of node *n*.
$\#G^n$	– Number of grammar productions of 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 set F
$\#\varphi_i$	– Number of arguments in relation φ_i.
$\varphi arg_{i,j}$	– j-th argument in relation φ_i.
$\varphi_i - \varphi arg_{i,1}$	– Relation φ_i without the first argument
Id : Objects → Integers	– Function mapping Object names to their respective numbers.

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

Unification

There is also a procedure Alter(x → y in z) in the algorithm. It simply substitutes each appearance of x for y in set z. The Alter procedure is used to apply to the goals the set of local **unifications**. For further simplicity let us call the variables *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 Rel(x,y,Shop) goal and a Rel(Home,Home, z) property where Home and Shop are not variables, then the unification operation can be successful. In this case, the set of unifications is as follows:

$\Omega = \{x \rightarrow Home, y \rightarrow Home, z \rightarrow Shop\}$.

The unify function syntax is:

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

UNIFY(Rel_1, Rel_2, Ω) = false Rel_1 cannot 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 Alter(x → y in z) function was analyzing 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 become the altered set of goals)
        Do
            F' = F
            For each u∈Ω do Alter(u in F)
        While F' ≠ F
```

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
```

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 the 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

II. 4 Sequential actions and plan scheduler

So far, the planning algorithm was creating plans for single goal or a set of unordered goals. After a goal had been satisfied, the planning process returned a set of network updates and an array of actions which have to be executed in order to achieve this goal. In fact, we might consider each resulting action as a new starting goal for the planning algorithm. However, these goals associated with sequent actions must be achieved separately, one after another.

For example: suppose there is a goal 1.2 satisfied after executing actions 1.2.1 and 1.2.2. If Action 1.2.1 is not a low level action, then the system might try to apply the planning algorithm for Goal 1.2.1 If this goal is satisfied after executing the actions 1.2.1.1 and 1.2.1.2, we can wrap up the actions for Goal 1.2 obtaining an array:

(Action 1.2.1.1, *then* Action 1.2.1.2, *then* Action 1.2.2)

Now each plan can be extended like the scheme: it can consist of ordered sets of goals. Each set of goals can consist of either one low level action or a certain number of goals for a standard planning algorithm. In the latter case, this set equals to F_0 – the input value for the PickSubGoal function. An example plan which consists of ordered goals like low level actions and complex goals is depicted in figure 3.

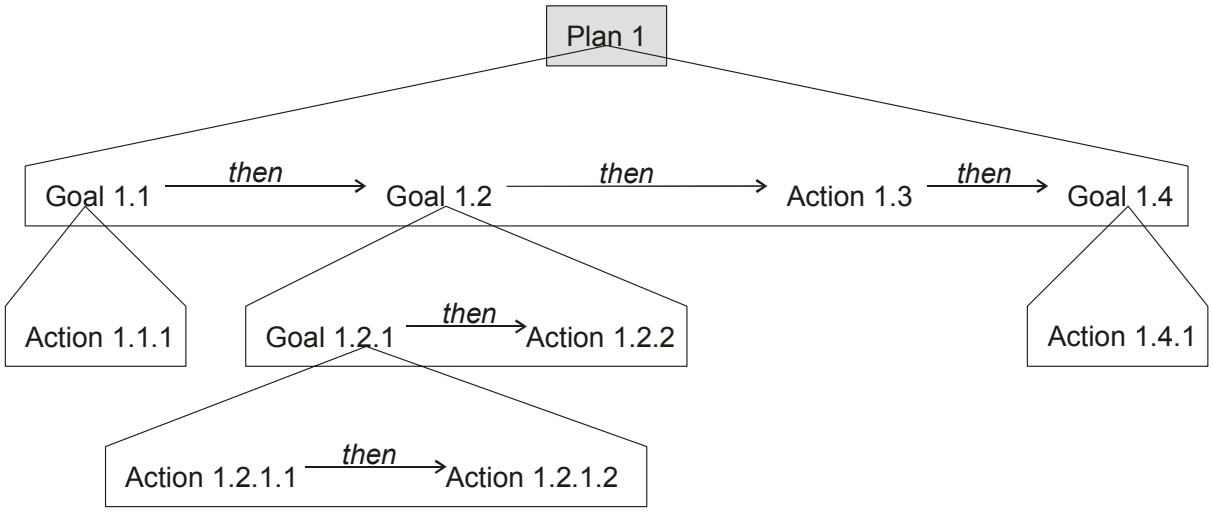


Figure 3 Ordered goals of a plan

The pseudo language algorithm which manages extended plans is depicted below:

Index of symbols:

P – The ordered array of sets of goals. $P = (F_1, \ldots, F_n, \ldots, F_N)$
F_n – The n^{th} set of goals

$\{\varphi^{(n)}_1, \varphi^{(n)}_2, \varphi^{(n)}_3 \ldots \varphi^{(n)}_n\} \in F_n$ –unordered goals of the F_n set.
LowLevelAction(φ) – The boolean function which returns true if φ is a low level action, and false if φ is a complex goal.
Execute(φ) – The function which executes low level action φ. Returns true if the execution of φ was successful, and false if the execution of φ failed.

```
Function Plan(P) returns true/false
While P ≠ ∅ do
        Pick F1          //the first set of goals from the array P
        If  #F1 = 1 and LowLevelAction(φ(1)1) then
                If Execute(φ(1)1) = false then return false
                Delete F1 from the array P.
        Else
                U:= ∅
                A:= ∅
                If PickSubGoal(F1, U, A, {}) = true then
                        Delete F1 from the array P.
                        P := A + P        //Concatenate the set of actions A with P
                        Apply the updates from U to the Semantic Network
                Else return false
        Return true
```

Remarks:

- Since there are no starting unifications for the *PickSubGoal* function, the function parameter Ω equals {}.
- If the function *Plan* fails, already executed actions and applied Semantic Network modifications stay intact.
- The function *Plan* is not recursive, therefore the implementation of loops of the same action is possible (see chapter "Semantic Network Algorithms")
- Alternatively, depending on the nature of the problem, the planning algorithm with goals overlapping management can be used.

We can imagine the intelligent agent as a hierarchical set of independent processes where each process is responsible for one part of agent's behavior. If the agent is a virtual computer program, then some of its processes can be executed simultaneously, but if the agent is associated with a real robot, then it should clearly have a scheduler which applies the priorities to the plans. The plans with higher priority values will be executed before those with lower priority values. Eventually, a priority value for a process can vary with the time making it more or less important.

If currently executed process loses its highest priority in favor of the other process, then it should be postponed at its actual execution state. Moreover, the postponed process

should track the Semantic Network updates to be able to check if it can normally resume when it is given back the highest priority.

In the Chapter "Inference" the process communication mechanism will be introduced, allowing different processes to exchange information and even stopping or resuming their execution.

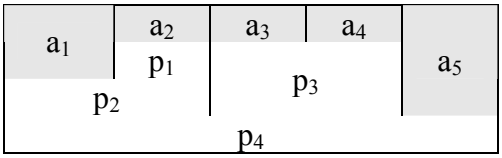
For some intelligent system a good idea might be to allow processes to create other processes and destroy them if necessary. The study of those issues, although not covered in this paper, is a challenging task for intelligent systems developers.

II. 5 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, knowing that somebody buys candles, red wine and a turkey in the evening it may indicate 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 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.

Since the in-depth study of the reverse planning process in not within the scope of this paper, it is not covered in this section.

Chapter III Inference

Abstract

This chapter provides detailed explanations of the implementation of inference rules for the semantic network being constructed in this paper. A bi-lateral approach to deal with the inference rules written in first-order logic will be presented: generalized modus-ponens and resolution. The first method will be greatly based on the planning algorithm, and we will show that it is not complete, whereas the latter will make up for the complete inference system.

III. 1 Introduction to inferences

The term "inference" is generally used to cover any process by which a conclusion can be reached. In some languages, this process can be strictly restricted to symbols that represent whole propositions (fact), in others it can take a more universal form.

In **propositional** logic, symbols that represent facts are whole sentences (like atoms) and cannot be changed in any way during the inference process. We might have atom C, which stands for a phrase "It is raining in Toronto", and this phrase can be either true or false. We are simply not allowed to influence the structure of this atom to change its meaning or logical value. Although proposition symbols can be combined using boolean connectives to generate more complex sentences, prepositional logic is a rather limited tool for modeling world and its behavior.

First-order logic commits to the representation of the world in terms of objects and predicates on objects, as well as using connectives and quantifiers which allow sentences to be written to present multiple things at once. The planning algorithm presented in this paper is intended to work with the first-order logic, because when it creates plans, it considers not only atoms but also general relations and grammar productions that simultaneously express knowledge about many objects in the Semantic Network.

As it was shown earlier, the Semantic Network being constructed in this work consists of multiple elements, like classes, objects (class instances), relations and node-embedded grammar productions. Generally, node's properties can be understood as relations linking this node with other elements of the network, like property values. Considering the fact, that inference rules are sometimes made up of simple facts without implications nor alternatives, it is important to know, that even without the inference mechanism, the constructed Semantic Network possesses some inference capabilities so far, notably for the sentences of the form: $True \rightarrow Relation_1 \,\&\, Relation_2 \,\&\, \ldots \,\&\, Relation_n$.

Inference rules of the form given above constitute of course only a small portion of all inference rules that can be expressed in the first-order logic.

The most fundamental inference rule in prepositional logic is Modus-Ponens. It says, that from an implication and the premise of that implication, we can infer the conclusion:

Knowing that: α, $\alpha \Rightarrow \beta$ *we can infer* β.

It turns out, that prepositional modus-ponens can be expanded to a generalized modus-ponens, a fundamental inference rule for the first-order logic. This is basically done by applying unification theory to the implication itself, its premises and conclusion. Even though the first-order logic system equipped with a generalized modus-ponens inference rule is

already a powerful tool for making complex inferences, it turns out, that it is not a complete system. There are still logical sentences that are not covered by the inference mechanism, although many artificial intelligence systems can already work fine.

The complete first-order logic inference system can be accomplished when we equip it with the resolution inference rule, which will be presented later.

Because of the fact, that on one hand generalized modus-ponens can be easily combined with the Semantic Network without having to add significant changes to it, and on the other hand the resolution is a complete system, two inference methods are presented.

III. 2 Modus Ponens

We have already seen that if there is the $R(x_1, x_2, \ldots, x_n)$ relation operating on objects $x_1, x_2, .., x_n$, then it is attached to at least one of the $x_1, x_2, \ldots, x_n$ objects. Suppose it is attached to object x_j, then the $R(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, ..., x_n)$ relation counts among x_i node properties.

On the other hand, if there is an inference rule as $A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow P(x_i)$ it means that x_i node property P should become true if only properties $A_1, A_2, \ldots, A_n$ are satisfied in any order. Therefore it is quite natural to place this inference rule in node x_i. The question which arises is "where exactly this rule can be located? ". Fortunately Semantic Network nodes store not only relations attached to this node, but also appropriate grammar productions that can easily represent inference rules emerging from modus-ponens.

First, let us focus on the standard modus-ponens rule from the point of view of propositional logic. In this case, every inference rule's elements (the premises and a consequence) can be directly associated with individual objects or classes. Consequently no unification takes place, and the rule can be easily represented using node-nested grammars. If we have an inference rule in the form of:

$$A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow P(x_i)$$

where x_i is a concrete object, then it can be re-written as an x_i grammar production:

$$P(.) \rightarrow A_1 \,\&\, A_2 \,\&\, ... \,\&\, A_n$$

Elements $A_1, A_2, ... , A_n$ point to various Semantic Network locations, and they may constitute either existing properties (relations) or other inference rules represented as corresponding grammar productions.

In fact, as long as grammar productions do not carry actions, they can be treated as inference rules. Hence, the standard planning algorithm with goals overlapping management may be used to draw inferences from the rules. The planning algorithm without goals overlapping management may not be sufficient to draw inferences when grammar productions entail Semantic Network updates. Typically, the inference rules do not entail network updates but when the inference mechanism based on the planning algorithm without goals overlapping management comes across a grammar production with an update (typical for planning tasks), it can fail to draw a conclusion.

Below, there is an example showing how a set of inference rules can be encoded using node-nested grammars:

Suppose we have the following inference rules:

Likes(Eve, Mark) ∧ Birthday(Mark) ⇒ Happy(Mark)
Month(Today, 11) ∧ Day(Today, 21) ⇒ Birthday(Mark)
Likes(Mark, Eve) ⇒ Likes(Eve, Mark)

And the following facts:
Likes(Mark, Eve)
Month(Today, 11)
Day(Today, 21)

The standard backward chaining algorithm if asked whether Mark is happy would easily deduce that he is in a good mood in deed. Let us translate the given situation to a form compatible with the Semantic Network.

First, we must assume the existence of objects like Mark, Eve and Today. Then the following relations must be fulfilled:
For the Today object: *Month(11) and Day(21)*
For the Mark object: *Likes(Eve)*

Finally, for all inference rules, respective grammar productions have to be added:
For Mark object: *Happy(.) → Birthday(.) & Likes(Eve, .)*
Birthday(.)→ Month(Today, 11) & Day(Today, 21)

For Eve object: *Likes(Mark) → Likes(Mark, .)*

Now when the planning algorithm without the goals overlapping management has to find plan "Happy(Mark)" it performs the following:

1. Cannot find *Happy* property in the *Mark* node, but finds the grammar production that unifies with *Happy(Mark)*. Its new sub-goals are *Birthday(Mark) & Likes(Eve,Mark)*
2. Tries to satisfy *Birthday(Mark)* sub-goal, explores the *Mark* object looking for *Birthday(.)* property. Instead of this property it finds a production for *Birthday(Mark)* : *Month(Today, 11) & Day(Today, 21)*. Those two new sub-goals are satisfied when the planning algorithm finds properties *Month(11)* and *Day(21)* in the *Today* object. Consequently the sub-goal *Birthday(Mark)* becomes true.
3. Only one sub-goal is not yet fulfilled: *Likes(Eve, Mark)*. The planning algorithm jumps to *Eve* node, and although it fails to find explicit property *Likes(Mark)*, finds production *Likes(Mark) → Likes(Mark, .)* that changes the actual sub-goal to *Likes(Mark, Eve)*. Since the *Likes(Eve)* property is located in the *Mark* node, it is immediately found when the algorithm enters the *Mark* node. As a result of that *Likes(Eve, Mark)* becomes true.
4. Since both sub-goals: *Birthday(Mark)* and *Likes(Eve,Mark)* are fulfilled, the *Happy(Mark)* goal also becomes true, and the planning algorithm terminates.

Consequently, the planning algorithm has accomplished the same task as the backward chaining algorithm.

The next step, after proving that node-nested grammars can tackle the inference problem for propositional logic, is to prove their effectiveness for the first-order logic. In order to do that, the generalized modus-ponens inference rule must be introduced. Let us recall the *Alter* procedure which applies the set of substitutions to a set of first-order logic predicates. Expression *Alter(x→ y in z)* substitutes each appearance of x for y in set z. More

generally, Alter(Q, Ω) takes all substitutions from set Ω, and applies it to every predicate from set Q.

It is important to mention that the structure of the proposed Semantic Network has some restrictions. It allows relations to have only the form of $R(x_1, x_2, \ldots, x_n)$, where $x_1, x_2, \ldots, x_n$ are objects (classes) that can be directly pointed. Consequently, x_i cannot be the function that returns some object being a parameter of relation R. As a result, relations like R(f(g(x),y),z) are not allowed which can cause some problems for the Semantic Network designers. Although it may look as though the above restriction constitutes a serious problem, it is only the first impression. The integration of Semantic Network nodes with their respective grammar productions constitutes an ideal solution to this problem. The mechanical process that changes nested relations to grammar productions dealing with them is covered later on.

As an example of this process, we may consider the following relation: *Likes(Mark, Mother(Eve))*. This fact can be expressed in the following way:

The Mark node has the following grammar production: $Likes(., x) \rightarrow Mother(Eve, x)$

The node Eve has the following property: *Mother(mrs)*

Where *mrs* is the real name of Eve's mother.

Generalized Modus-Ponens: For atomic sentences p_i, p_i' and q, where there is substitution Ω such that $Alter(p_i, \Omega) = Alter(p_i', \Omega)$, for all i:

Knowing that: $p_1', p_2', \ldots, p_n', \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)$

we can infer: $Alter(q, \Omega)$

To demonstrate the significance of this rule, an example of how it works can be shown. There are two facts and one inference rule written in the first-order logic:

Cd(MichaelJackson-Dangerous)

Has(Mark, MichaelJackson-Dangerous)

$\forall x \ Cd(x) \wedge Have(Mark, x) \Rightarrow Listen(Mark, x)$

Generalized modus ponens is able to draw a conclusion that Mark listens to Michael Jackson's album "Dangerous" in a very efficient way. Firstly there is a need to use the $UNIFY(Rel_1, Rel_2, \Omega)$ function, which creates (if possible) a minimal set of substitutions Ω, that satisfy condition: $Alter(Rel_1, \Omega) = Alter(Rel_2, \Omega)$. A need emerges to find two unifications for the example given above:

UNIFY(*Cd(x), Cd(MichaelJackson-Dangerous),* Ω_1)

UNIFY(*Have(Mark, x), Have(Mark, MichaelJackson-Dangerous),* Ω_2)

Now general substitution $\Omega = \Omega_1 \cup \Omega_2$ can be applied to infer $Alter(Listen(Mark, x), \Omega)$. Consequently the final conclusion can be drawn that Mark listens to Michael Jackson's album - "Dangerous".

Because of the fact, that Semantic Network relations have a very simple, restricted form which does not allow one relation to be embedded in the other, the unification function is in deed very fast and easy to implement (see Chapter "Planning").

Transformation from Horn sentences to grammar productions

Although the inference mechanism build with one inference rule – the generalized version of modus-ponens is not a complete one, it still represents great inference possibilities. This section shows that it can be simulated by node-nested grammars and the existing planning algorithm to which only few slight modifications will be applied.

First of all it must be noticed that all sentences in the knowledge base (represented in the Semantic Network) should be in a form that matches one of the premises of the generalized modus-ponens rule – otherwise, they could never be used. It means that each sentence in the knowledge base should be either an atomic sentence, or an implication having on its left hand side a conjunction of atomic sentences and a single atom on the right. Sentences having this form are called **Horn sentences**, and a knowledge base consisting of only Horn sentences is said to be in the Horn Normal Form.

Moreover, the structure of the proposed Semantic Network has some restrictions concerning the form of relations mentioned earlier. Those restrictions, although eliminated by node-nested grammars, impose additional requirements to the form of the Horn sentences.

From now on each Horn sentence having the following form:

$$R_1(x_1^1, x_2^1, \ldots, x_{n1}^1)$$
$$R_2(x_1^2, x_2^2, \ldots, x_{n2}^2)$$
$$R_3(x_1^3, x_2^3, \ldots, x_{n3}^3) \Rightarrow Q(y_1, y_2, \ldots, y_n)$$
$$\ldots\ldots$$
$$R_k(x_1^k, x_2^k, \ldots, x_{nk}^k)$$

will be called Semantic Network Horn Sentence (SNHS). In the definition given above, $R_1, R_2, \ldots, R_k$ are the atoms on the left hand side of the SNHS, and $x_1^i, x_2^i, \ldots, x_{ni}^i$ are the arguments of the i^{th} atom. There is also no difficulty to transform any original Horn sentence to a SNHS. Suppose there is a Horn sentence with nested relations:

$R_1(R_2(R_3(\ldots R_n(arg)))) \Rightarrow Q$

Since R_1 arguments must be atoms, $R_2(\ldots)$ must return atom a_2. Consequently R_2 arguments must also be atoms, so $R_3(\ldots)$ must also return an atom, this time a_3. By applying this rule to all nested relations we state that $R_n(arg)$ must return atom a_n. Thus we may write the Horn sentence in the following way:

$R_1(a_2) \wedge a_2 = R_2(a_3) \wedge a_3 = R_3(a_4) \wedge \ldots \wedge a_n = R_n(arg) \Rightarrow Q$

Atom "arg" should be an object without nested relations. It serves as the first pointing location for the planning algorithm EXPLORE function. In order for the original Horn sentence to draw the conclusion Q, fact $R_n(arg)$ must be a function in the knowledge base that returns an object a_n. In the Semantic Network this is a function (relation) characteristic for the "arg" object. It is therefore property $R_n(a_n)$ located in the arg node. It can be read as follows: "Relation R_n working on the arg object returns a_n" which is exactly what we desire from relation R_n to express.

Now atom a_n had been unknown until we have evaluated the $R_n(arg)$ function. The result a_n will be an argument for the next function - R_{n-1} etc. The same process is being executed by the planning algorithm, which enters the "arg" node and looks for any possible objects a_n satisfying relation $R_n(a_n)$. When it succeeds, a_n is no longer a variable, but unifies with an existing object (class). It becomes therefore a starting point for the next relation that

has to be fulfilled. The complete grammar production emulating the original Horn sentence is depicted below:

$$Q \rightarrow R_n(arg, a_n) \ \& \ R_{n-1}(a_n, a_{n-1}) \ \& \ R_{n-2}(a_{n-1}, a_{n-2}) \ \& \ \ldots \ \& \ R_1(a_2)$$

After a_n has been found, it serves as an anchor to find a_{n-1}, which is an anchor to find a_{n-2} etc. Finally when object a_2 is known, the planning algorithm can enter it, and when R_1 is true in the a_2 node, sentence Q also becomes true.

If there is rule $R(A(arg_a), B(arg_b)) \Rightarrow Q$, then R clearly has more than one argument being a result of an embedded relation, and we have to repeat the above reasoning twice. As a result we obtain the following grammar production:

$Q \rightarrow A(arg_a, x_1) \ \& \ B(arg_b, x_2) \ \& \ R(x_1, x_2)$

So far we have demonstrated how a Horn sentence can be written in the Semantic Network using node-nested grammar productions. Now we should take a closer look at the Horn sentences, and the way they are obtained from the first-order logic sentences. In other words we must get rid of an existential and universal quantifier. In order to achieve it we have to apply two inference rules:

Universal elimination: For any sentence α that includes a universal quantifier $\forall x$, where x is a variable that occurs in α, the quantifier is dropped and each occurrence of x is substituted for a **universal variable**, that doesn't exist elsewhere in the network, for example X_i where "i" is the sentence id.

Existential elimination: For any sentence α that includes an existential quantifier $\exists x$, where x is a variable that occurs in α, the quantifier is dropped and each occurrence of x is substituted for an **existential variable**, that doesn't exist elsewhere in the network, for example x_i where "i" is the sentence id.

As an example, let us suppose there is a sentence with id=37:

$\exists x \ \forall y \ \forall z \ \ A(x, y) \wedge B(z) \Rightarrow Q$

which takes the following form:

$A(x_{37}, Y_{37}) \wedge B(Z_{37}) \Rightarrow Q$

Each SNHS will be located in one of the Semantic Network node. The appropriate node for the SNHS can be chosen depending on the category to which all of its variables belong. If for example a rule states facts only about cars, then it can be placed in the car node. Consequently, each variable can have an unique id that consists of the node's name, and the local number of a grammar production it represents.

The deeper a SNHS is, the faster an inference process can be, since the planning algorithm explores the Semantic Network via the hierarchical links that go from a more general object to its sub-objects.

It is important to remember, that from now on, each appearance of a big letter variable will be associated with a universal symbol, and a small letter variable with an existential symbol.

Modifications to the planning algorithm:

Although it looks as though all Horn sentences can be dealt with using the standard planning algorithm, the truth is that there are some vital modifications that have to be made. At this step we should be aware of the fact that the planning process can use some inference rules when they are needed. The example planning process depicted in the Chapter "Planning" showed that relation Have(x) could be a consequence of other relations like Buy(x), Create(x) etc. Even though grammar productions predestined for creating plans can be used when making inferences, many of them are useless from the point of view of the inference mechanism. The problems occur when:

1) The planning algorithm tries to explore a relation via the argument which is not yet defined (a variable before the unification takes place). In this case, it should return false without trying to perform Explore function.
2) A goal or a sub-goal is preceded by the negation symbol. Now we have to distinguish two situations: the negation sign precedes a relation that is preceded by a quantifier in first-order logic, or the negation sign precedes a quantifier that precedes a relation. Fortunately the following logical sentences are true:

$$\begin{aligned} \forall x \neg P(x) &\equiv \neg\exists x\, P(x) \\ \neg\forall x\, P(x) &\equiv \exists x \neg P(x) \\ \forall x\, P(x) &\equiv \neg\exists x \neg P(x) \\ \exists x\, P(x) &\equiv \neg\forall x \neg P(x) \end{aligned}$$

Thanks to those rules we can always move the negation sign before the quantifier and in some cases get rid of two negations. If the inference system supports existential and universal quantifiers, this allow us to limit negation problem to the case when they always stay before a quantifier. Consequently, the Explore function should work only on positive relations. If a system receives a rule with a negated relation, it should be translated to the appropriate form before placing in the Semantic Network.

3) A grammar production has some attached actions that will be made if all of this production's sub-goals are fulfilled.

4) A Semantic Network Horn sentence has a relation with a universal variable. Typically if a variable is an existential one, the planning algorithm tries to find only one object that satisfies it. If a variable is universal, all objects from the category to which this variable is assigned must satisfy a certain relation.

5) A Semantic Network Horn sentence has a relation with an existential variable. Typically if a variable is an existential one, the planning algorithm tries to find an object that unifies with this variable, and if it succeeds, returns true. At this moment we would like to clone the inference process, to obtain two processes:

    a. The first process returns true just like the standard planning algorithm. The instance for the existential quantifies has been successfully located
    b. The second process behaves as if the found object did not unify with the existential variable. Consequently the inference process continues its search.

The above problems (except for the 3-rd) also apply to the planning process, therefore we should first write the enhanced version of the planning algorithm. Then, it can be easily transformed to an inference algorithm by simply not allowing the planning process to consider productions with attached actions.

Since the enhanced version of the planning algorithm has to deal with the 5-th problem, it will definitely be more hardware demanding. It is up to the system designer to choose the best programming language that allows process cloning. We also have to be aware of the fact that the enhanced version of the planning algorithm will be much slower than the standard version because it will try to find all solutions, not just the first one.

The implementation of cloning is relatively easy, since all we have to do, is to clone the current inference process when it comes across a successful unification. A communication platform must be implemented to allow cloned processes to send signals to other cloned processes if one of them finds a solution. When one of the cloned processes reaches a starting goal, all the others can be destroyed if we are not interested in other solutions. When making inferences, other cloned processed should continue until they succeed or fail.

Cloned processes do not have to run simultaneously. They can be executed in a sequence in which the priority belongs to the old process. The old process holds a successful unification for some relation, and continues the search to satisfy its remaining goals. The cloned process simply ignores the unification found by the old process, therefore its execution should start, when the old process reaches a dead end (fail return value). The clone should be identical to the old process up to the point where the old process's successful unification took place.

As an example of process cloning the following situation is considered. Let us suppose there is a plan:

$plan_1 \rightarrow A(Node_1, x) \,\&\, B(Node_2, x)$ associated with process p_1.

The variable x is existential, so the appropriate unification must be found for it. We also assume that there are Semantic Network nodes: $Node_1$ and $Node_2$ with the following properties assigned to it:

$Node_1$
$A(obj_1)$
$A(obj_2)$

$Node_2$
$B(obj_3)$
$B(obj_2)$

The standard planning algorithm does the following: first, tries to satisfy a sub-goal $A(Node_1, x)$, so it enters $Node_1$ to find a unification for A(x). $A(obj_1)$ unifies with A(x),**(*)** so x becomes obj_1 and $A(Node_1, obj_1)$ is satisfied. Now the second sub-goal, notably $B(Node_2,obj_1)$ must be satisfied (x's unification had an influence on B's arguments). Process p_1 enters $Node_2$ and since there is no unification for $B(obj_1)$, returns a fail value. That however is not a total failure. P_1 knows, that it can try a different permutation of sub-goals to find the solution, so it tries to find $B(Node_2, x)$ first and then $A(Node_1, x)$. After entering $Node_2$, B(x) unifies with $B(obj_3)$ **(**)** and x becomes obj_3. Now p_1 tries to satisfy a sub-goal $A(Node_1, obj_3)$. Since there is no property $A(obj_3)$ in $Node_1$, process p_1 fails again. All possible sub-goal permutations have been analyzed and there is no plausible plan, so p_1 signals a total failure.

The enhanced algorithm does the same up to the point **(*).** At this moment the first cloning takes place. The old process p_1 becomes $p_{1,1}$, and the now process becomes $p_{1,2}$. The process $p_{1,2}$ will start from this point (ignoring the property $A(obj_1)$) when $p_{1,1}$ signals a total

failure. Process $p_{1,1}$ up to the point **(**)** behaves as p_1 in the standard planning algorithm. Now we have another cloning. $P_{1,1}$ forks into $p_{1,1,1}$ and $p_{1,1,2}$, where $p_{1,1,1}$ is similar to p_1 in the standard planning algorithm. Process $_{1,1,1}$ having unified x with obj_3 tries to find $A(obj_3)$ in $Node_1$. That leads it to a total failure. It is a signal for $p_{1,1,2}$ to resume from the point **(**)**. It ignores the property $B(obj_3)$, and soon finds $B(obj_2)$, which unifies with B(x). Now $p_{1,1,2}$ tries to satisfy a sub-goal $B(Node_1, obj_2)$. Since $B(obj_2)$ is a $Node_1$'s property, process $p_{1,1,2}$ reaches a success and sends a signal "destoy" to all processes $p_{1,x,x,x,x,..}$.

The planning algorithm for making inferences on SNHS's that deals with problems (1, 2, 4, 5) mentioned above will use a slightly altered FindSubGoal function, and a modified version of the Explore function. To change the enhanced planning algorithm to an inference algorithm, we would simply not allow the planning algorithm to consider productions with attached actions.

The problem no. 2 is fully eliminated by the $FindSubGoal_{enhanced}$ function.

Since the enhanced function differs from its original version in one place, we will only present a region of FindSubGoal for which the modification has been applied (marked as bold instruction).

```
…
            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(φp(i,j) – φarg p(i,j),1 , φarg p(i,j),1 , U, A, Ω)= true then
                        If Positive(φp(i,j) ) then
                              If φarg old≠φarg p(i,j),1 then Ω:=Ω ∪ {φarg old→φarg p(i,j),1}
                              ALTER(F, Ω)
                        If Negative(φp(i,j) ) then
                              Result = false
                              Exit {For j:=1 to (#F) }
                  Else
                        If Positive(φp(i,j) ) then
                              Result = false
                              Exit {For j:=1 to (#F) }
                        If Negative(φp(i,j) ) then continue
…
```

Function **Positive(sub-goal)** returns **true** is sub-goal is preceded by the negation sign. In the other case it returns **false**.

The problem with negations is tricky and can lead to some confusion. The inference mechanism can return true in case the goal $\varphi_{p(i,j)}$ was negative, and no unification was found for it. It would not be a problem if $\varphi_{p(i,j)}$ is a goal with only one variable, notably the one that was explored. If on the other hand, $\varphi_{p(i,j)}$ has an other existential non-initialized variable, then it will not be unified although the goal $\varphi_{p(i,j)}$ is met. It is due to the fact, that the inference mechanism did not find any object that could satisfy the $\varphi_{p(i,j)}$ without the negation sign. It should therefore return true although no object providing unification for an existential variable was found. Typically we should construct logical rules in such a way, that negative

goal's arguments are initialized (except for the quantified one) and the inference mechanism does not have to find any variable unification. This is a small limitation to the Semantic Network planning system. Since the resolution algorithm for first-order logic is presented at the end this chapter, this limitation is rather not important.

The $Explore_{enhanced}$ function is depicted below (bold instructions constitute updates to the standard Explore function; letters preceding them are the reference points).

```
Explore_enhanced(Rel, var : Obj, U, A, Ω)  returns boolean
LocalVariables: Obj_old , Ω_old, Ω_temp, UV_count

a)      If  Variable(Obj) then return false
        For i:=1 to #P^id(Obj) do
                If UNIFY(Rel , P^id(Obj)_i , Ω_temp )   then
b)                      clone(); if(new process) then continue
                        Ω := Ω ∪ Ω_temp
                        return true

        For i:=1 to #G^id(Obj) do
c)              If A^G(id(Obj),i) ≠ ∅ then continue
                Ω_old := Ω
                If UNIFY(Rel , R^G(id(Obj),i)_0 , Ω_temp)  then
d)                      clone() if(new process) then continue
                        Ω := Ω ∪ Ω_temp
                        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)
                                U := U ∪ U^G(id(Obj),i)
                                ALTER(U, Ω)
                                ALTER(A, Ω)
                                return true
                        else Ω := Ω_old
e)      UV_count = 0
        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
f)                      If UniversalVariable(Obj) then UV_count := UV_count + 1
                        Else return true
                Else Obj := Obj_old
g)      If UniversalVariable(Obj) and UVcount = #C^id(Obj) then return true

        Return false
```

The $Explore_{enhanced}$ function uses the following local elements:

UviversalVariable(x)– the function which returns true if x is a universal variable, and false if x is an existential variable

$\mathbf{UV_{count}}$ – the number of actual node's children, for which the relation Rel(…) was found.

Instruction a) has been written to solve problem no. 1. If the algorithm tries to explore a relation via a variable, that has not yet been initiated, it immediately fails.

Instruction c) **could be** written to solve problem no. 3. When the algorithm finds a grammar production with attached actions, it is not even considered. Instruction "Continue" loops the entire "*For"* section for the current *i* value without doing anything.

Instructions e), f), g) have been written to solve problem no. 4. If the $Explore_{enhances}$ function receives a universal variable *Obj* already initiated with a value, then the function must check if every child emerging from *Obj* fulfills the *Rel* relation. UV_{count} keeps a record of the number of *Obj* children which fulfill *Rel*, it is therefore initiated to 0 at the point e). Whenever an *Obj*'s child satisfies *Rel*, the variable UV_{count} is incremented (at the point f)). At the point g) the algorithm checks if for a universal variable all of *Obj*'s children have fulfilled *Rel*, and if yes, returns true. It is clear, that the $Explore_{enhances}$ function explores the whole sub-graph emerging from the *Obj* node when *Obj* is a universal variable. Sometimes it stops in a Class node without having to explore deeper (if a Class satisfies *Rel*, all of its children also do). Sometimes it must reach the Semantic Network bottom (individual objects) and check if every of them fulfills *Rel*.

Instructions b), d) have been written to solve problem no. 5. Relation Rel can unify either with some property or some grammar production, hence there are two points where the process can clone. The new process waits for his parent's failure signal, and executes the instruction "continue" which ignores the current unification.

It must be mentioned, that universal variables should not be altered by the ALTER procedure. A universal variable is a starting point for the inference algorithm rather than a variable for which a specific value has to be found. For example consider the following sentence:

$$[\forall_{Y \in UCSBStudents}\ At(Y, UCSB) \wedge Have(Y, Pencil)\ ] \Rightarrow AnExamAt(UCSB)$$

In this case, Y will be initiated to UCSB students, and the inference algorithm will jump to the node UCSB students, and then check if every one of them is at the UCSB. The universal variable Y will then propagate do deeper elements of the Semantic Network, but it should not change its form.

A set of inference rules

Up to now, we were only focusing on the inference making algorithm that works on one single sentence. This was functioning relatively well (the in-depth analyze of all possible Horn sentences the inference algorithm is able to work with is beyond the scope of this paper), but if we were given a set of inference rules with one rule influencing another, then the situation could become more complicated.

First of all, if there are many inference rules with existential quantifiers, in some cases they relate to the same existential variable. So far the initiation of all existential variables took place only within the current inference process and did not carry forward to other inference processes. Thanks to the fact that grammar productions can perform Semantic Network updates, the inference process, while drawing its own conclusion, can transmit some

knowledge to other inference processes. This knowledge can be either a fact about the world, or an actual variable value.

Each inference rule can be associated with a separate planning process running all the time. The Semantic Network Scheduler shown in Chapter "Planning" can be a tool responsible for managing the priority of various planning processes.

An example below shows how two separate processes can exchange facts:

$Process_1 \rightarrow A \ \& \ B$ | Updates: $Process_1$ $A \rightarrow$ true | Updates: **$Fact_1$** $B \rightarrow$ true	$Process_2 \rightarrow C \ \& \ D$ | Updates: $Process_2$ $C \rightarrow Fact_1$ $D \rightarrow B$

If $Process_1$ terminates, there are two new facts in the Semantic Network: $Process_1$ and $Fact_1$. Thanks to the latter, $Process_2$ can also terminate.

An example below shows how two separate processes can exchange a variable value. Let us suppose we know the value of $Node_1$, but the value of an existential variable x remains unknown. This value will be found by the first process, and used further by the second process. The first process constitutes a following plan:

$Process_1 \rightarrow A(Node1, x)$ \| Updates**: $Process_1$, Equals(T_1, x)** in the node **TempValues**

In the node $Node_1$ there is property $A(object_{24})$, so A(x) unifies with it. As a result, variable x is initialized to $object_{24}$. When $Process_1$ terminates, two new facts are stored in the Semantic Network: $Process_1$ (in some node)
Equals(T_1, $object_{24}$) in the node TempValues

First of all, considering the fact that $Process_2$ will fail as long as it cannot find the value for variable x, we are sure that this inference rule will not fire.

Secondly, we can directly initialize the value of x to T_1:

$Process_2 \rightarrow$ **Equals(TempValues, T_1, x)** & D(x) \| Updates: $Process_2$

$Process_2$ will terminate when $object_{24}$ (or one of its descendants) has either property D, or a grammar production whose left part unifies with D is satisfied.

The fact that various inference processes can exchange additional knowledge and local unifications makes the inference mechanism very flexible. It is up to the human to use the full potential of the Semantic Network structure to encode the inference rules in the most efficient way.

Finally we can demonstrate an example inference process for the following problem:

> There is an Oxford rule which states that if all students fail to pass an exam at the end of the A course, then if the lecturer of A has been corrupted in the past, he is no longer the lecturer of the A course.

There are many ways to write this rule; one of them is as follows:

Rule $\rightarrow$ *Corrupted(teachers)& Conduct(teachers ,courses) &* $\neg$*Passed(Students, courses)*

\| Updates: not *Conduct(teachers, courses)*

That grammar production assumes several facts:

1) There is node *Students*, to which all Oxford students belong.
2) There is node *Courses*, to which all Oxford courses belong.
3) If student A passed an exam from the C course, then there is a Semantic Network link from A to C representing relation *Passed*.
4) If teacher T is a lecturer of the C course, then there is a Semantic Network link form T to C representing relation *Conduct*.
5) If a teacher is corrupted, then his Semantic Network node has property *Corrupted*.

Existential variables are: *teachers, courses*
Universal variables are: *Students*

Those variables respectively set starting points for the inference algorithm. It should be added, that since the inference algorithm starting permutation analyses premises (sub-goals) from left to right, it will find the unification for the variable *courses* before it explores a negative relation involving this variable.

III. 3 Resolution

The resolution mechanism is a tool which manages inferences on any first-order logic sentence, not necessarily having the form of the generalized modus-ponens. The inference mechanism for modus-ponens type rules was presented, because it was *using* the full potential of the Semantic Network structure and rules encoded in node's grammar productions. Moreover, it was very similar to the planning process. The resolution mechanism that will be shown in this section is completely different from the previous inference mechanism and takes no advantage of the Semantic Network structure with its grammar productions.

First of all, it should be clear why the resolution in so important. Consider the following set of rules:

$\forall x\ A(x) \Rightarrow B(x)$
$\forall x\ \neg A(x) \Rightarrow C(x)$
$\forall x\ B(x) \Rightarrow D(x)$
$\forall x\ C(x) \Rightarrow D(x)$

It is obvious, that for any x, D(x) is true, because either B(x) or C(x) are true.

Unfortunately the generalized modus-ponens is not able to draw this conclusion, because it cannot unify the variables for negative relations. The second rule: $\forall x\ \neg A(x) \Rightarrow C(x)$ cannot be transformed into Horn-like sentence and therefore C(x) will not necessarily be true for some objects x.

It was the German mathematician Kurt Gödel who presented in 1930 his completeness theorem which proved that, for first-order logic, any sentence that is entailed by another set of sentences can be proved from that set. That is, if sentence α is entailed from the certain Knowledge Base KB, then there exists a resolution process using sentences from KB which can prove α.

Gödel showed that there exists a resolution process that proves α, but it was up to Robinson in 1965 to publish the resolution algorithm.

The resolution inference rule

The standard resolution rule for the propositional logic is as follows:
Knowing that $\alpha \vee \beta$, and $\neg\beta \vee \delta$ we infer $\alpha \vee \delta$

The above rule can be understood in the following way: given a sentence β, it can be true or false, so either β or $\neg\beta$ is false. Now, if β is false then since $\alpha \vee \beta$ is true, α is also true. If β is false then since $\neg\beta \vee \delta$ is true, δ is also true. In any case, $\alpha \vee \delta$ is true.
This rule can be also expressed in the implicative form, more readable for humans:
Knowing that $\neg\alpha \Rightarrow \beta$, and $\beta \Rightarrow \delta$ we infer $\neg\alpha \Rightarrow \delta$

The standard resolution rule can be expanded to the resolution rule for first-order logic sentences. We can use the unification function $\text{UNIFY}_{\text{enhanced}}$, that is similar to the standard UNIFY function presented in Chapter "Planning". $\text{UNIFY}_{\text{enhanced}}$, function allows predicates (relations) to have more complex arguments, like function's return values etc. The simplest $\text{UNIFY}_{\text{enhanced}}$ procedure can be found in [3] page 303. The ALTER function that applies updates to a sentence is also necessary to be able to draw inferences from the rules of the following form:

Generalized resolution rule (disjunctive form)

Suppose there is unification Ω for the literals p_j and $\neg q_k$: $\text{UNIFY}(p_j, \neg q_k) = \Omega$
Knowing that $p_1 \vee \ldots \vee \boldsymbol{p_j} \vee \ldots \vee p_m$ and $q_1 \vee \ldots \vee \boldsymbol{q_k} \vee \ldots \vee q_n$
We infer $\text{ALTER}(\Omega, (p_1 \vee \ldots \vee p_{j-1} \vee p_{j+1} \vee \ldots p_m \vee p_1 \vee \ldots \vee p_{k-1} \vee p_{k+1} \vee \ldots p_n))$

Generalized resolution rule (implicative form)

Suppose there is unification Ω for the atoms p_j and q_k: $\text{UNIFY}(p_j, q_k) = \Omega$
Knowing that $p_1 \wedge \ldots \wedge \boldsymbol{p_j} \wedge \ldots \wedge p_{n1} \Rightarrow r_1 \vee \ldots r_{n2}$ and
$s_1 \wedge \ldots \wedge s_{n3} \Rightarrow q_1 \vee \ldots \vee \boldsymbol{q_k} \vee \ldots \vee qn_4$
We infer
$\text{ALTER}(\Omega, p_1 \wedge \ldots \wedge p_{j-1} \wedge p_{j+1} \wedge \ldots \wedge p_{n1} \wedge s_1 \wedge \ldots \wedge s_{n3} \Rightarrow r_1 \vee \ldots \vee r_{n2} \vee q_1 \vee \ldots \vee q_{k-1} \vee q_{k+1} \vee \ldots \vee q_{n4}))$

The resolution rule forces inference rules to be either:

- A disjunction of literals. Sentences that go along with this schema are in a **conjunctive normal form (CNF).**
- An implication with a conjunction of atoms on the left and a disjunction of atoms on the right. Sentences that go along with this schema are in an **implicative normal form (INF).**

Below are some example sentences written in both forms:

CNF	INF
$A(x) \vee B(x)$	$\text{True} \Rightarrow A(x) \vee B(x)$
$\neg A(x) \vee B(x)$	$A(x) \Rightarrow B(x)$
$A(x) \vee \neg B(x)$	$B(x) \Rightarrow A(x)$
$\neg A(x) \vee \neg B(x)$	$A(x) \wedge B(x) \Rightarrow \text{False}$

Conversion to normal form

Since first-order logic sentences can be in various forms, and the resolution inference mechanism works only on normal forms, there is a need to transform every sentence to an appropriate CNF or INF. The conversion to normal form procedure is depicted below in sequential steps. Each of them changes the form of the rule without influencing its meaning:

- **Elimination of implications**: every sentence having the implication $p \Rightarrow q$ becomes a sentence with the alternative $\neg p \vee q$.
- **Moving the negations inwards**: due to the fact, that the normal form allows negations to precede only individual atoms (not literals), sentences need to have their negations repositioned. The following de Morgan's laws involving quantifier equivalence and double negation elimination are used to tackle this problem:

  $\neg(p \vee q)$ *becomes* $\neg p \wedge \neg q$
  $\neg(p \wedge q)$ *becomes* $\neg p \vee \neg q$
  $\neg \forall x\ q$ *becomes* $\exists x\ \neg q$
  $\neg \exists x\ q$ *becomes* $\forall x\ \neg q$
  $\neg\neg q$ *becomes* q
- **Variable standardization**: for sentences of the form $\forall x\ Q(x) \vee \exists x\ F(x)$ in which one variable appears more than once after different quantifiers, there is a necessity to change the name of each variable to make it unique. This avoids confusion later, when the quantifiers drop.
- **Moving the quantifiers left**: without changing the sense of a sentence, its quantifiers are moved to the left, for example:

  $p \vee \exists x\ q(x)$ *becomes* $\exists x\ p \vee q(x)$
- **Skolemization**: is a process destined to eliminate all existential quantifiers. If an existential variable is not nested inside some universal quantifier, skolemization is trivial; if substitutes the quantified variable with a constant that does not appear elsewhere in the knowledge base, as it was presented earlier. In the case when the existential variable *x* is nested inside a universal quantifier, there is a need to substitute it with an appropriate **Skolem** function, which takes as arguments all universally quantified variables influencing x, for example:

  $\forall y\ A(y) \Rightarrow \exists x\ B(x) \vee C(x,y)$ *becomes* $\forall y\ A(y) \Rightarrow B(\mathbf{F}(y)) \vee C(\mathbf{F}(y), y)$

  where **F** is a Skolem function for the variable x.
- **Moving the quantifiers left**: without changing the sense of a sentence, its quantifiers are moved to the left, for example:

  $p \vee \exists x\ q(x)$ *becomes* $\exists x\ p \vee q(x)$
- **Distribution of $\wedge$ over $\vee$**: $(a \wedge b) \vee c$ *becomes* $(a \vee c) \wedge (b \vee c)$
- **Flattening of nested conjunctions and disjunctions**: every sentence of the form

  $(a \wedge b) \wedge c$ *becomes* $(a \wedge b \wedge c)$
  $(a \vee b) \vee c$ *becomes* $(a \vee b \vee c)$

At this point, the sentence which still keeps its original meaning is in conjunctive normal form CNF, and can be analyzed by the resolution algorithm. However, it may be

difficult for a human to see the sense staying behind this rule before we convert it to implicative normal form (INF). To do so, we scan the sentence, create set A_n of its negative atoms, and set A_p of its positive atoms. Then we create a new implication with the left part being a conjunction of A_n's elements, and the right part - a disjunction of A_p's elements.

Resolution implementation

The simplest resolution algorithm simply takes every two sentences from the Knowledge base and tries to apply for them the generalized resolution rule. It repeats this step until every combination on two sentences is analyzed, and no fact has been added to the KB. It there is a couple that may be used for the resolution rule, the algorithm starts from the beginning.

Although this resolution algorithm will surely find all possible facts that can be inferred from the Knowledge Base, its weakest point is the computational complexity. Due to that, several strategies have been proposed to make the search toward a proof more efficient.

One of them is to create a **set of support**, which is a subset of all sentences from the KB. Every resolution combines the sentence from the set of support with another sentence, and adds the resolvent into the set of support. If the size of the set of support is small in comparison with the size of the KB itself, this strategy will cut the search time dramatically. However, there exist a risk that not all inference making sentences are placed in the set of support. In this case the resolution will not draw all available conclusions.

The Semantic Network structure gives us the possibility to use two strategies to make the resolution algorithm work more efficient:

- We can create a set of support that combines:
  1. The grammar productions from all network nodes converted to the conjunctive normal form.
  2. The additional first-order logic sentences that could not be represented by node-nested grammars, converted to the normal form.

  All the other knowledge accumulated in the Semantic Network consists of simple relations on objects and can be therefore left outside the set of support. Using this strategy, we should remember, that the Semantic Network now consists of the network itself and the set of additional resolution rules.
- Since an additional first-order logic sentence is a composition of disjointed atoms and the atoms represent various relations in the Semantic Network, we can imagine each additional sentence as a sub-graph lying above the Semantic Network. This sub-graph could span some Semantic Network relations on a specific level. Different additional sentences could occupy different levels above the Semantic Network. Each sub-graph edge (a,b) connecting objects a and b, with label R could represent Semantic Network relation R(a,b). Edges could be either positive or negative, depending on the atom associated with them.

  By taking a snapshot through some Semantic Network object (class) O_n perpendicular to the plane of the Semantic Network, we could gather the sub-graphs that are using the same object O_n. Now we could analyze the relations that are connected to the object O_n for all gathered graphs. We should also create a temporary sub-graph from the node O_n and its Semantic Network properties. If for two sub-graphs G_i, G_j we found edge (O_n, x) with the same label R, but different logical values (one edge positive, one negative), we could

splice graphs G_i, G_j into new graph G_{ij}, without the edge (O_n, x) with label R. If the graph G_{ij} had only one edge, we could interpret it as a fact and encode this fact in the Semantic Network structure. The figures 4, 5 and 6 show in brief how such a process could look like.

We may also make multiple parallel snapshots that pass through Object 1.2 descendants, which would allow us to gather sub-graphs for the generalized resolution rule.

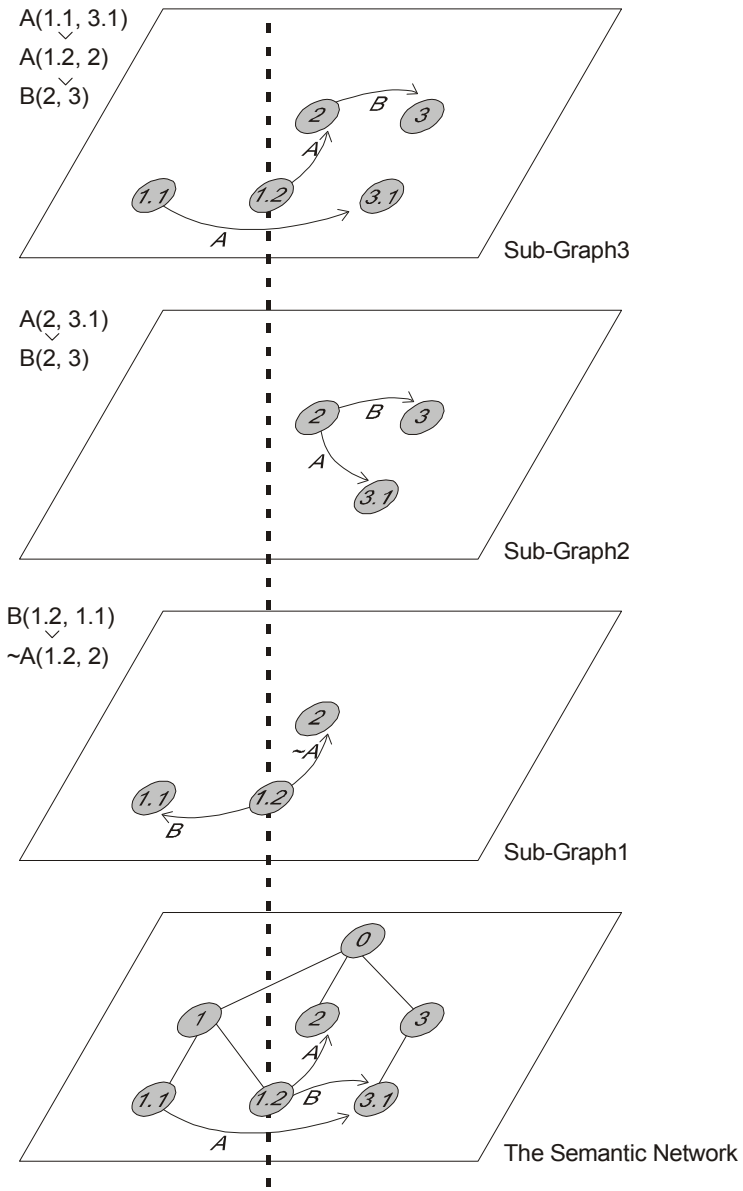


Figure 4 Snapshot through the object 1.2.

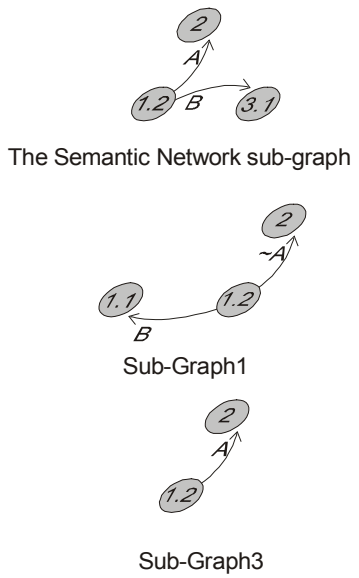


Figure 5 Sub-graphs with relations attached to the object 1.2

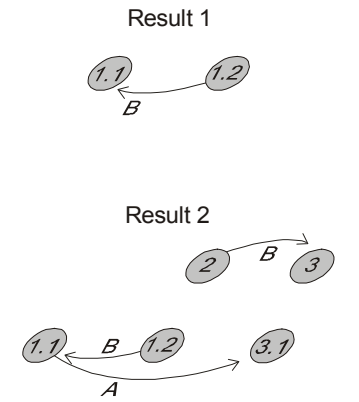


Figure 6 Two graphs representing two resolution algorithm results

This resolution strategy is roughly a theoretic concept, and its deeper analyze is beyond the scope of this paper.

Chapter IV Grouping

Abstract

This chapter will provide some ideas regarding the problem of grouping in the Semantic Network. The Semantic Network is a complex graph structure, which can eventually act autonomously, therefore object / relation management tools need to be present to keep the network structure well organized. Two independent grouping techniques will be covered: grouping by complexes using the CLUSTER/2 algorithm, and grouping by probabilities using the COBWEB algorithm.

IV. 1 Introduction to groupings

When the Semantic Network is being created, its designer can take care of the appropriate network structure. Since he knows how the environment looks like, he can implement a system with the right object, category and property distribution. The designer not only supervises the category tree, but also introduces new divisions among categories with too many independent objects.

When the system loses the contact with the designer it must act on its own. It must not only take care of keeping the Knowledge Base up-to-date, but also make it flexible, effective and (if possible) readable for a human.

Along with the system's perception element, which is usually responsible for making updates to the Knowledge Base, Semantic Network grammars also encode the information of how to update the network structure. Eventually it turns out, that even the most actual Knowledge Base will become ineffective if it is not kept flexible. For example, if a system has a fixed category tree, and during its live encounters mainly objects belonging to one category, then the category tree does not add to the functionality of this system. In this case we would like this one category to divide into smaller categories incorporating fewer objects. Objects located in a smaller category should be similar, and should differ from the objects from other categories.

Two grouping algorithms presented in this paper highlight the appropriate network category layout. The first one (CLUSTER/2) covers the space of examples with disjointed complexes representing categories. Each category possesses some properties, common for all its objects. The second one (COBWEB) represents a probabilistic approach to the issue of grouping. It analyses the similarities of various objects from the point of view of all their properties, and creates a tree of categories for them. Instead of inheriting common properties of objects, each category stores probability estimations for arguments of objects belonging to this category. Once started, COBWEB algorithm never stops, rearranging the category tree with the arrival of new objects.

The grouping algorithms mentioned above possess different advantages which make them useful for different grouping tasks. Grouping by finding complexes is a crucial element of every self-organizing network; Network nodes with common properties are stored in a more general node. This reduces the risk of having network nodes overloaded with independent, non-categorized elements. Grouping by probabilities on the other hand is a perfect tool for categorizing objects with the maximum alikeness of properties. It can be used in such areas as the space clusterisation, where object properties can represent space coordinates. In this case some objects can have all their coordinates different, yet still belong to the same category.

Let us assume there is a family of attribute sets: $A_1, A_2, \ldots, A_n$, and a set of objects P. Every object O, such that $O \in P$ is characterized by a vector of properties:

$O = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$

For the two given objects: o_1, o_2, we can assume that if their respective vectors of parameters are alike, objects o_1 and o_2 have generally much in common and a grouping algorithm should put them together into the same category C. On the other hand, if there is object o_3 that belongs to the C category, and some of its properties remain unknown, then it can inherit some of these properties from the C category.

IV. 2 Cluster/2 algorithm

Covering the space of examples

The CLUSTER/2 algorithm, proposed in 1983 by Michalski and Stepp [4] creates categories, which are represented by so called **complexes**. More precisely, each of the groups the learning set P is split into consists only of examples (objects) covered by some complex. The grouping process can be thus understood as a method for finding a set of complexes, each of them representing one group-category. Since every example should belong to one specific category, complexes must cover the whole set P, and they must be disjointed. Every complex not only stores examples, but also provides a description of the category associated with this complex.

Complexes provide a clear logical description of the property ranges of their examples. If all the examples are characterized by *n* attributes, then each complex is described by *n* **selectors** – one selector for one attribute. Selectors impose restrictions within the confines of one argument for all the examples of selector's complex. For example, if we wanted to describe the domain of people, we might consider three sets of attributes:

$A_1 = \{small, medium, tall\}$
$A_2 = \{light, medium, heavy\}$
$A_3 = \{child, teenage, adult, retired\}$

Now, there can be complex C describing people being fit. Its selectors: V_1, V_2, V_3 are the subsets of consecutive argument sets. The selector marked as "?" accepts every value of the argument it refers to.

$C = \{ V_1, V_2, V_3\}$
Where $V_1 =$? $V_2 = light \vee medium$ and $V_3 = child \vee teenage \vee adult$

In this example the set of examples can contain up to $|A_1| \cdot |A_2| \cdot |A_3|$ different people. The C complex on the other hand is a grouping for up to $|V_1| \cdot |V_2| \cdot |V_3|$ people. Because the set of complexes must cover the whole set of examples, clearly at least one additional complex is needed.

Due to the fact, that for a given set of examples, there are a lot of possibilities how to cover it with a set of complexes, we must introduce the measure of grouping effectiveness: its **dispersion**. The dispersion of complex C is big, when despite C covers a big part of the whole space of examples, there are not many of them in C. In other words, the C selectors are relatively general, but that does not change the fact, that there are few examples in it. The formal definition of complex dispersion is the following:

Given the set of all examples P, the complex C = $\{V_1, \ldots, V_n\}$ and a set of examples $P_c \subset P$ covered by C, the dispersion of C can be calculated using the formula below:

$$\xi_c(P) = | V_1 \times \ldots \times V_n - \{<a_1(x), \ldots, a_n(x)> : x \in P_c\} |$$

or $$\xi_c(P) = | V_1 | \cdot \ldots \cdot | V_n | - | P_c |$$

where $a_i(x)$ is a function that returns the i^{th} argument of x.

The dispersion of a grouping is a sum of dispersion for all complexes that make up for this grouping. The smaller the grouping dispersion, the more effective and accurate is this grouping.

Finding the best grouping

So far, we have been able to formulate a grouping problem and evaluate the possible groupings. The aim is now to construct an algorithm which generates a set of disjointed complexes covering the whole space of examples, and makes it as effective as possible. Since the selectors of each complex force the examples to have common arguments within a complex, we can expect similar examples to be stored in the same category and different examples in different categories.

Complexes describing individual categories are generated in a way which is similar to the concept used in the AQ algorithm (known from the induction of rules). Each complex is a result of a search for all possible complexes located around a chosen example, which will be called "*seed*". In order to create *k* independent complexes, the CLUSTER/2 algorithm first picks at random k different examples. It is however useful to possess a measure which calculates the distance between any two examples and choose seeds which are located as far as possible one from the other. After picking *k* different seeds, for each of them a star of possible complexes is created (see fig. 7). Next, the possible groupings are generated by picking one complex from each star. For each grouping we should then check if its categories are disjointed and if that is not the case, call a procedure which applies some modifications to the set of complexes.

Since the measure of the grouping efficiency can be its dispersion, the best set of categories can be easily identified. If this grouping satisfies a certain condition, the algorithm terminates; in the other case it repeats the process of generating stars, this time around seeds located in algorithm's previous iteration complexes.

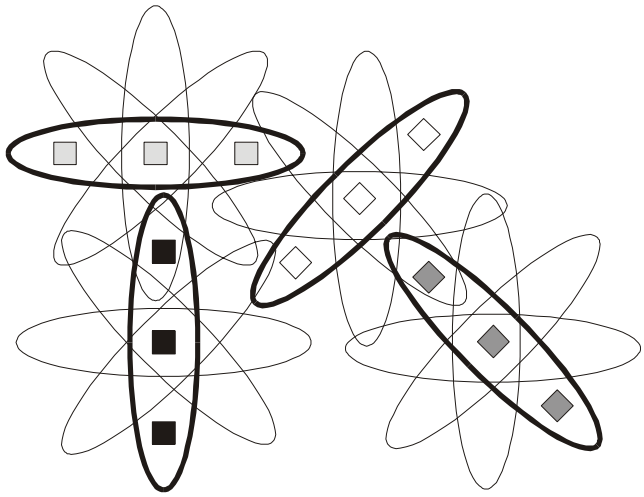


Figure 7 Stars in Cluster/2 system

Before we write the CLUSTER/2 pseudo-language code, we will show how to create stars around seeds.

When creating stars, the following symbols must be explained:

Positive seed: an example around which the star is being created - must be included in every complex added to the star.

Negative seed: an example which cannot be covered by any complex added to the star.

The algorithm generating stars takes as an input a positive seed and a set of negative seeds and returns the complexes which make up for a star. It also has access to all sets of arguments for the space of examples. First, the star is marked as empty. Then, for each set of arguments A_i, the local set of arguments A_{local} becomes equal to A_i. Next, each negative seed is analyzed, and its i^{th} argument $a_i(x)$ is deleted from the set A_{local}. Finally, if a complex of the form C_{local}=<?,…,A_{local},…,?> still covers the positive seed, it is added to the star. A detailed description of this process is depicted below:

```
Function CreateStar(PositiveSeed, NegativeSeeds) returns Star
LocalVariables: A_local, C_local
    Star := {}
    For each set of argument A_i do
        A_local := A_i
        For each negative seed S ∈ NegativeSeeds do
            A_local := A_local − {a_i(S)}
        Complex_local = <?,…,A_local,…,?>
        If Cover(Complex_local, PositiveSeed) then
            Star := Star ∪ {Complex_local}
```

The CLUSTER/2 algorithm receives on its input only two parameters: P - the set of examples, and n - the number of categories P should be divided into. It will be discussed later that if the set of complexes cannot cover the whole set P, additional complexes have to be generated. Every algorithm's iteration (associated with the variable k) creates set C^* of n complexes covering examples from P. Those complexes are extracted from n stars located around seeds. The seeds are chosen at random only for the first iteration of the algorithm (when k=0). Later (when $k = k_1 > 0$), they are selected from the complexes C^* found by previous algorithm iterations (when $k = k_1 - 1$).

After fixing n seeds x_s^1, x_s^2, …, x_s^n, n reduced stars S_i are generated using the CreateStar algorithm. Then, the algorithm analyses all possible sets of complexes C which are created by respectively extracting one complex from one reduced star (for i=1, …, n). After every set of complexes C is modified to guarantee its disjunction, it is compared with the previous set of complexes. If the grouping evaluation function υ considers the new set of complexes to be better than the previous one, then the actual best grouping C^* is modified. When the stop condition is not reached for the best grouping C^*, the algorithm starts its next iteration; in the other case it returns the best grouping C^*.

Although the stars are created in such a way, that none of them covers two different seeds, there is sometimes a need to call EnsureDisjunction function for the set of complexes.

It may turn out, that a set of complexes extracted from the set of reduced stars does not cover all examples P. This incompleteness can be a useful factor when evaluating the effectiveness of a current set of complexes. The second approach to this problem is to apply a small modification to the standard CLUSTER/2 algorithm. This modification would create a

minimal number of additional complexes for all uncovered examples. Each of these additional complexes should be disjointed not only with other additional complexes, but also with the complexes from the set K^*. The additional complexes would be extracted from the additional stars located around additional seeds which were not covered by any complex from K^*. After successfully finding the additional complexes for all uncovered examples, they can be added to the set K^*.

The CLUSTER/2 pseudo-language code is depicted below:

```
Function CLUSTER/2(P, n) returns C*
Local variables: υ* - the efficiency of the best grouping

    C* := 0;  υ* := – ∞
    Repeat
        For k=1, 2, …, n do
            x_s^k := ChooseSeed(P, k, C*)
        X_s := ⋃_{k=1}^{n} { x_s^k }
        For k=1, 2, …, n do
            S_k := ReducedStar(P, x_s^k, X_s – {x_s^k}, m)
        For each C = {c_s | c_s ∈ S_k, k=1, 2, …, n} do
            C := EnsureDisjunction(C)
            If υ_C(P) > then
                C* := C;  υ* := υ_C(P)
    Until StopCondition(C*, P)
```

- The function **ChooseSeed** picks n seeds from the set of examples P.
- The function **ReducedStar** calls a **CreateStar** function with two additional parameters:
  - **P** – the input set of examples
  - **m** – the upper limit of complexes the Star can posess.
- The function **EnsureDisjucntion**(C) modifies the set of complexes C to ensure their disjunction.
- The StopCondition can be formulated in many ways. On of them is to terminate the algorithm if there is no improvement to the best grouping for a fixed number of iterations.

Evaluating the efficiency of a grouping

Although the CLUSTER/2 system ensures that similar examples are being put into the same category, there are better and worse possible groupings. The choice of the best grouping can be guided by the following criterions:

- The measure of fitness of examples to the categories - the overall sum of dispersion among all complexes. Groupings with less dispersion are generally more accurate
- The complexity of category description - the overall number of non-universal selectors in all complexes. Groupings with less number of selectors are more readable.

- The number of examples not covered by the set of complexes. The less non-covered examples we have, the better grouping we deal with.

CLUSTER/2 and the Semantic Network

As it was said at the beginning of this chapter, network nodes with common properties should be stored in a more general node. If the system acts autonomously, it will have to execute the CLUSTER/2 algorithm for various Semantic Network nodes. If the initial condition, like the number of node children being too great is satisfied then, this node along with the children nodes and the projected number of categories are the input values for the CLUSTER/2 algorithm.

An example demonstrating how the algorithm works in the Semantic Network is depicted below.

Let us suppose there is the network node N, which has 6 children O_1, O_2, O_3, O_4, O_5, O_6. There are also two relations R_1 and R_2, with their values presented in figure 8.

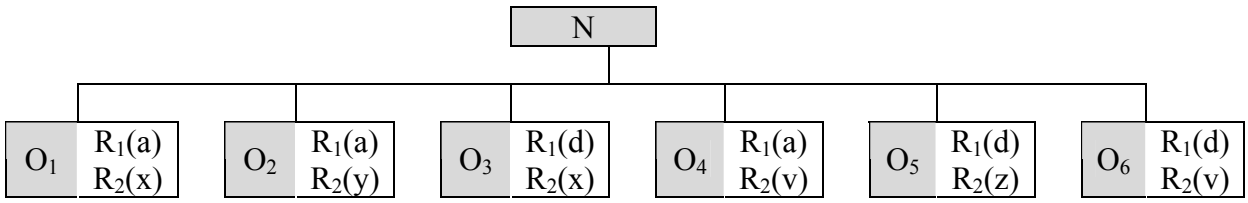
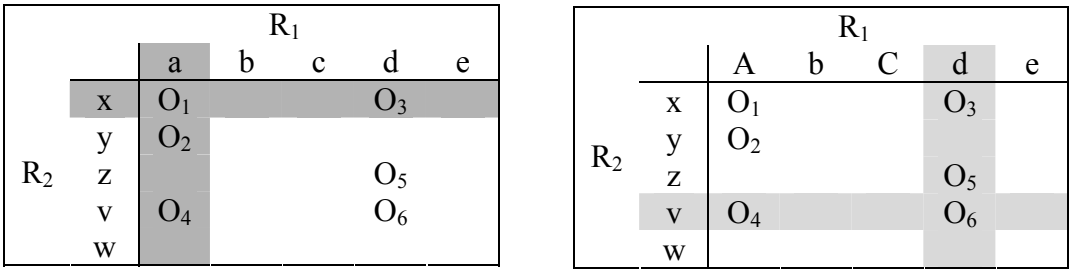


Figure 8 Example starting nodes for Cluster/2 algorithm

The goal of CLUSTER/2 algorithm is to find two categories which cover all N's children. First, the algorithm chooses two seeds among those children. Let them be O_1 and O_6. Figure 9 shows the space of examples and two reduced stars for a given problem:



Star for the object O_1
First complex = <a, ?>
Second complex = <?, x>

Star for the object O_6
First complex = <d, ?>
Second complex = <?, v>

Figure 9 Two reduced stars for an example Semantic Network node

After generating stars for the objects O_1 and O_6, the algorithm generates all possible groupings by picking one complex from each star, and repeating this process 4 times. At the end, only one grouping covers the whole set of examples. This grouping has the following complexes:

C_1 = <a, ?> C_2 = <d, ?>

Those complexes introduce two categories: Cat_1 with relation $R_1(a)$, and Cat_2 with a relation $R_1(d)$, which are depicted in figure 10.

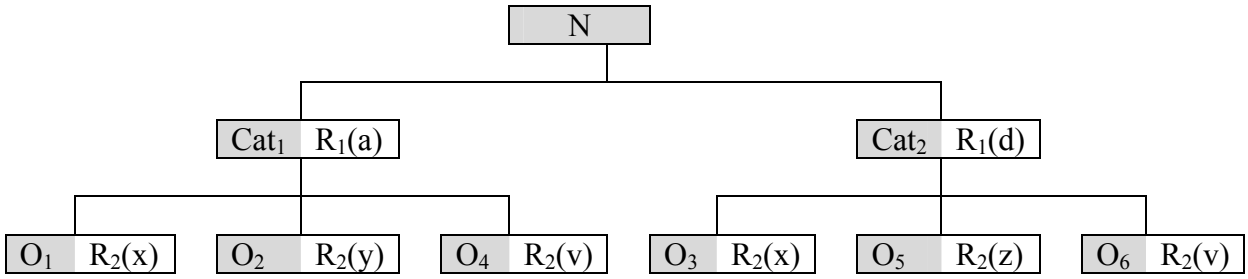


Figure 10 Result of Cluster/2 algorithm for an example Semantic Network node

IV. 3 Cobweb algorithm

Unlike the previous approach where grouping was represented by the set of complexes, the COBWEB system [5] presented first by Fisher in 1987 generates categories which do not store attribute values. Instead, a possible category node stores the probabilities of each attribute having a specific value. They are calculated by dividing the number of examples within this category having a specific argument value into the overall number of examples of this category. Because of that, the category representation does not force the examples from one category to be similar, and the examples from different categories to be different. These are the algorithm goals realized by the **evaluation function**, which is at the heart of the system.

The evaluation function estimates the performance for the specific hypothesis **h** which is the following function: $h: X \rightarrow C_h$, where X is the set of examples chosen with the probability Ω, and C_h is the set of categories specified by h. Furthermore, we also assume the presence of the argument functions $a_i: X \rightarrow A_i$, for i=1, 2,…, n. Although the original algorithm can deal with continuous values, our approach will be restraint to discrete parameters. One of the biggest advantages of the COBWEB system is that it never stops, modifying its categories with the arrival of new examples.

To properly describe the COBWEB system, the following issues must be covered:

- representation of grouping,
- operators that modify the existing hypothesis, creating a new grouping,
- evaluation function, which guides the grouping search process,
- search strategy which constitutes the algorithm itself.

Evaluation function

The main task of the evaluation function is to promote groupings which provide big similarities among the objects from one category, and big differences among objects from different categories.

Probabilities and similarity

Suppose category $d \in C_h$ is chosen. We are now interested in the probability that attribute a_i has the value v, if we know that this concerns only examples classified to be in the category d. This probability can be calculated with the use of the following formula (examples are being chosen with the probability distribution Ω) :

$$Pr_{x\in\Omega}(a_i(x) = v \mid h(x) = d)$$

In order to calculate these probabilities for all attributes and all attribute values we should rewrite this formula for i=1, 2, …, n and $v\in A_i$. This set of probabilities can represent the degree of similarities among the objects classified to be in the category d. If, for a considerable part of attributes A_i, only one or few A_i values have big probabilities $Pr_{x\in\Omega}(a_i(x) = v \mid h(x) = d)$, and other values have very small probabilities, then if an example is classified to be in d, we can predict its arguments values with a relatively good accuracy. In other words, if there are less big values and more small values then the examples which belong to the category d are more alike.

Now suppose the category d, the argument A_i and its value v are fixed. By analyzing the probabilities of the form below:

$$Pr_{x\in\Omega}(h(x) = d \mid a_i(x) = v)$$

it can be deduced that if some of those values are big, and the other relatively small, there are big similarities among the objects in d, and these objects are not likely to be in other categories. The bigger those probabilities are, the less common attributes have examples classified to different categories. By calculating those probabilities for all categories from C_h, and all attributes and their values we can measure the diversity among categories.

Since the evaluation function should measure the degree of similarities among objects from one category, and diversity among different categories, it seems this goal can be achieved by combing both presented probability measures. Finally we obtain the following formula for the evaluation function:

$$\sum_{d\in C_h} \sum_{i=1}^{n} \sum_{v\in A_i} Pr_{x\in\Omega}(a_i(x)=v) \cdot Pr_{x\in\Omega}(h(x)=d \mid a_i(x)=v) \cdot Pr_{x\in\Omega}(a_i(x)=v \mid h(x)=d)$$

The element $Pr_{x\in\Omega}(a_i(x)=v)$ stands for increasing the importance of more frequent attribute values.

Probabilities and inference

Using the Bayesians rules, the formula for evaluation function can be transformed to the form:

$$\sum_{d\in C_h} Pr_{x\in\Omega}(h(x)=d) \cdot \sum_{i=1}^{n} \sum_{v\in A_i} Pr_{x\in\Omega}(a_i(x)=v \mid h(x)=d)^2 \qquad (*)$$

which allows us to enrich its interpretation with the following reasoning:

Let us consider the random example x assigned to the category d, and a task to identify all its attribute values given the set of probabilities: $Pr_{x\in\Omega}(a_i(x)=v \mid h(x)=d)$ for i=1, 2, …, n and $v\in A_i$. This attribute identification could be made at random. In that case the probability of attribute a_i value v would be $p_i(v)$. Consequently, the probability of the correct $a_i(x)$ value would be:

$$\sum_{v\in A_i} p_i(v) \cdot Pr_{x\in\Omega}(a_i(x)=v \mid h(x)=d)$$

and the following formula would evaluate the degree of correct values for all attributes:

$$\sum_{i=1}^{n} \sum_{v \in A_i} p_i(v) \cdot Pr_{x \in \Omega}(a_i(x)=v \mid h(x)=d)$$

If we wanted to maximize this formula, then we should use the probability fitness strategy which sets $p_i(v)$ to $Pr_{x \in \Omega}(a_i(x)=v \mid h(x)=d)$, the real probability of this value's appearance. Finally, the formula (*) can be interpreted, an estimated number of attributes values, that can be correctly predicted for any example from the category d.

With the benefit of this additional interpretation we can precise the evaluation function's criterion for choosing the best grouping: it is the increase of the estimated number of attributes values that can be correctly predicted for grouping d, with relation to the estimated number of predictions without knowing this grouping.

Finally, we obtain the following formula for the evaluation function over hypothesis h:

$$\upsilon(h) = \frac{1}{|C_h|} \sum_{d \in C_h} Pr_{x \in \Omega}(h(x)=d) \cdot$$

$$\left[\sum_{i=1}^{n} \sum_{v \in A_i} Pr_{x \in \Omega}(a_i(x)=v \mid h(x)=d)^2 - \sum_{i=1}^{n} \sum_{v \in A_i} Pr_{x \in \Omega}(a_i(x)=v)_2 \right]$$

In this formula, the sum of squared probabilities obtained without knowing the grouping is subtracted from the sum of squared probabilities obtained with the knowledge about the grouping. Since the first sum does not depend on the hypothesis h, it does not have to be evaluated. The factor $1/|C_h|$ stands for the normalization.

Representation of grouping

Grouping tree

COBWEB system always creates a hierarchical grouping. It means that the division into groups is realized at different levels of a grouping tree. The more specified a group, the further from the root of the grouping tree it is. At the most general level (which we call level 0), we can assume, that all examples belong to one category, which divides at level-1 into a certain number of sub-categories. Each of those sub-categories divides at level-2 into their respective sub-categories etc. This process continues until a node of the grouping tree stores only one example. All sub-categories of each category are disjointed, and all examples of each category must be located in one of its sub-categories.

It is up to us to decide which level of a grouping tree is a satisfying division into categories.

The hierarchical grouping represented by a grouping tree is not evaluated as a whole. The evaluation function is always applied to the set of sub-categories of a given category. In other words, each hypothesis is associated with one node of a tree, and the evaluation function analyses only a part of this tree.

Evaluating the nodes

Each node of the grouping tree stores the probabilities of argument values of all objects within the node. Those probabilities are represented as counters, and can be easily evaluated given a set of node examples. To demonstrate a hypothesis concerning a specific

grouping suppose there is a grouping tree node n_0 and its related children n_1, n_2, …, n_m (see fig. 11). We assume, that the hypothesis $h_0 : X \rightarrow C_{h0} = \{d_1, d_2, \ldots, d_m\}$ is associated with the node n_0, with d_1, d_2, …, d_m representing the categories for nodes n_1, n_2, …, n_m. If n_0 is not a root of the grouping tree, it has a parent node n_{-1}, associated with the hypothesis $h_{-1} : X \rightarrow C_{h1}$. Consequently, we assume that the node n_0 represents the category $d_0 \in C_{h-1}$.

In order to evaluate the hypothesis h_0 associated with the node n_0, the following formula has to be calculated:

$$\upsilon(h_0) = \frac{1}{m} \sum_{d \in \{d_1,\ldots,d_m\}} \Pr_{x\in\Omega}(h_0(x)=d_0 \mid h_{-1}(x) = d_0) \cdot$$

$$\left[\sum_{i=1}^{n} \sum_{v \in A_i} \Pr_{x\in\Omega}(a_i(x)=v \mid h_0(x)=d)^2 - \sum_{i=1}^{n} \sum_{v \in A_i} \Pr_{x\in\Omega}(a_i(x)=v \mid h_{-1}(x)=d_0)^2 \right]$$

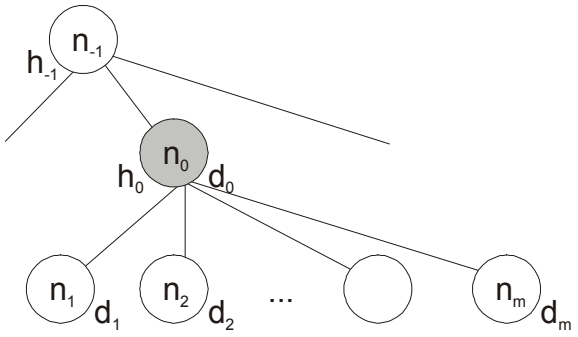


Figure 11 Cobweb grouping tree

All that is left to do is to show how to calculate the probabilities in the above formula: Suppose the number of examples in n_0 is $|T^{h_{-1}d_0}|$, and the number of examples in n_k (k=1, 2, …, m) is $|T^{h_0 d_k}|$. Each node also stores $|A_1|\cdot|A_1|\cdot\ldots\cdot|A_p|$ counters for respective possible values of all p attributes the examples can have. Consequently counters of the node n_0 will be the following: $|T_{A_i v}^{h_{-1}d_0}|$ for i=1, 2, … p and $v \in A_i$. Children of the node n_0 also have their set of counters of the following form: $|T_{A_i v}^{h_0 d_k}|$ for i=1, 2, … p; k=1, 2…, m and $v \in A_i$. Having all these counters we can finally calculate all the probabilities necessary for the evaluation function:

$$\Pr_{x\in\Omega}(h_0(x)=d_0 \mid h_{-1}(x) = d_0) = \frac{\left|T^{h_0 d}\right|}{\left|T^{h_{-1}d_0}\right|}$$

$$\Pr_{x\in\Omega}(a_i(x)=v \mid h_{-1}(x)=d_0) = \frac{\left|T_{A_i v}^{h_{-1}d_0}\right|}{\left|T^{h_{-1}d_0}\right|}$$

$$\Pr_{x\in\Omega}(a_i(x)=v \mid h_0(x)=d) = \frac{\left|T_{A_i v}^{h_0 d}\right|}{\left|T^{h_0 d}\right|}$$

Since the COBWEB algorithm starts every time a new example arrives, the set of examples T will eventually contain the growing number of examples, and the respective

counters can change. If an example has some missing attribute values then it can be assumed that this attribute takes any value with equal probability. Consequently the counters for the mentioned attribute will have fractional values.

Operators

Every time a new example arrives, the COBWEB algorithm must associate it with a category, either existing or a new one. In order to assure the maximum value of the evaluation function, the system deals with a new example using one of four operators:

- associating the example with an existing category,
- creating a new category for the example,
- splicing two existing categories together, and associating the example with the spliced category,
- splitting an existing category into a certain number of disjointed categories and then associating the example with one of them.

Above operators are chosen at different levels of the grouping tree starting from its root node. Every operator applies modification to the structure of this tree and updates the counters in the nodes it visits. Although the example moves through the tree from the root to the leaf node, it is not physically kept in this tree until it actually reaches a leaf node. In this case the AddToLeaf procedure is executed. This procedure attaches to this leaf (L) two new leafs L_1 and L_2 , moves the existing example from the node L to the node L_1, and then puts the new example to the node L_2.

Due to the fact that the in-depth analysis of all the operators is relatively easy, it will not be covered in this paper.

The algorithm

The COBWEB algorithm receives two parameters on its input: $\mathbf{x}^*$, a new example, and **n**, the node x^* should be added to. The recursive function responsible for processing x^* is as follows:

Function ***Cobweb***(x*, n)
If *n* is a leaf then
Create node *parent* with all the examples from the leaf *n* and example x^*, and make *n* the descendant of *parent*
Create node $leaf_2$ with example x^* and make $leaf_2$ the descendant of the *parent*
Else
Choose the best operator for node *n* using the evaluation function:
- Create new leaf *l* with example x^*, and attach this leaf to *n*.
- Put x^* to the node **n'** – the best descendant of *n*, and call the function Cobweb(x^*, **n'**)
- Splice two best descendants of *n* into node **n'** and call Cobweb(x^*,**n'**)
- Choose n' - the best descendant of n, split n' and call Cobweb(x^*, n).

Cobweb with continuous attributes

So far we have been focusing on the Cobweb algorithm with discrete attribute values, but its true potential reveals when the attributes are continuous.

The evaluation function for discrete attributes was calculated using the counters for each parameter values. For continuous attributes, instead of evaluating the counters, the probability density function should be used. If we fix category d, then for all its examples chosen from the domain with the probability Ω we can denote $g_\Omega^{i,d}$ as the probability density function for these examples' attribute A_i values. Similarly, g_Ω^{i} can be denoted as the probability density function for examples from any category. With probability density functions defined, the evaluation function for continuous attributes will be the following:

$$\upsilon(h) = \frac{1}{|C_h|} \sum_{d \in C_h} \Pr_{x \in \Omega}(h(x)=d) \cdot \left[\sum_{i=1}^{n} \int_{v \in A_i} g_\Omega^{i,d}(v)^2 dv - \sum_{i=1}^{n} \int_{v \in A_i} g_\Omega^{i}(v)^2 dv \right]$$

This formula can be used only if all attributes are continuous. If there are some discrete attributes, then instead of integrals, standard sums must be reintroduced.

The only real difficulty is to find the probability density function, different for each category. If we know that continuous attributes values have a standard probability distribution then we can analyze all category examples to find the parameters of this distribution like the variation and the estimation. After applying these parameters to a known probability distribution we obtain a nice probability density function.

For an unknown probability distribution we should try different probability distributions, determine their parameters, and finally choose a distribution which is the most consistent with the observed values.

The ability to generate categories for examples with continuous variables is particularly useful when there is a need to group locations in the space. In case of a 2-D space, we deal with locations represented by examples having two continuous attributes. By applying Cobweb algorithm to this set of examples we may obtain a nice set of multi-layer clusters as shown in figures 12 and 13.

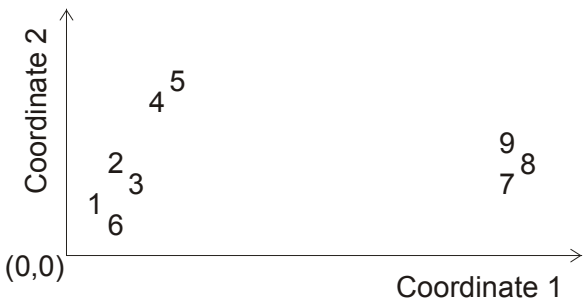


Figure 12 Cobweb example input data.

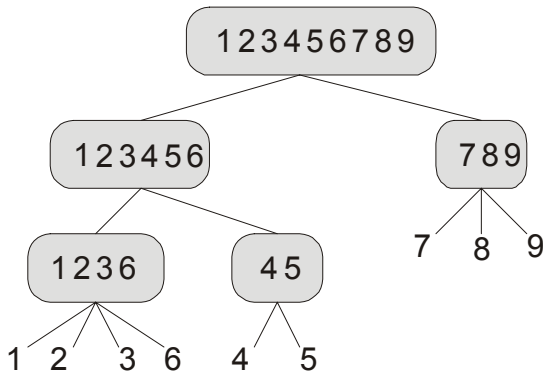


Figure 13 Cobweb example output data

Chapter V Time & Space

Abstract

Two important Semantic Network relations: time and space are discussed in this chapter. Time is a specific directed graph whose nodes represent time events; each time event is pointed by either: out-of-date, actual or future relation from the Semantic Network. Time graph's arcs link time events according to their start/finishing times. Space is represented by a tree, whose nodes represent different locations. Hierarchical structure of the space tree allows locations to form multi-level clusters.

V. 1 Time

Connection with the Semantic Network

So far, we have been only discussing relations which were actually present, without considering past of future properties of the network. Even if a relation becomes obsolete, it might still constitute an important fact for the system, and should be therefore archived in the knowledge base. Since every relation has its starting and finishing time, the natural way of keeping track of the history and future is to connect each relation with its time event (see fig. 14).

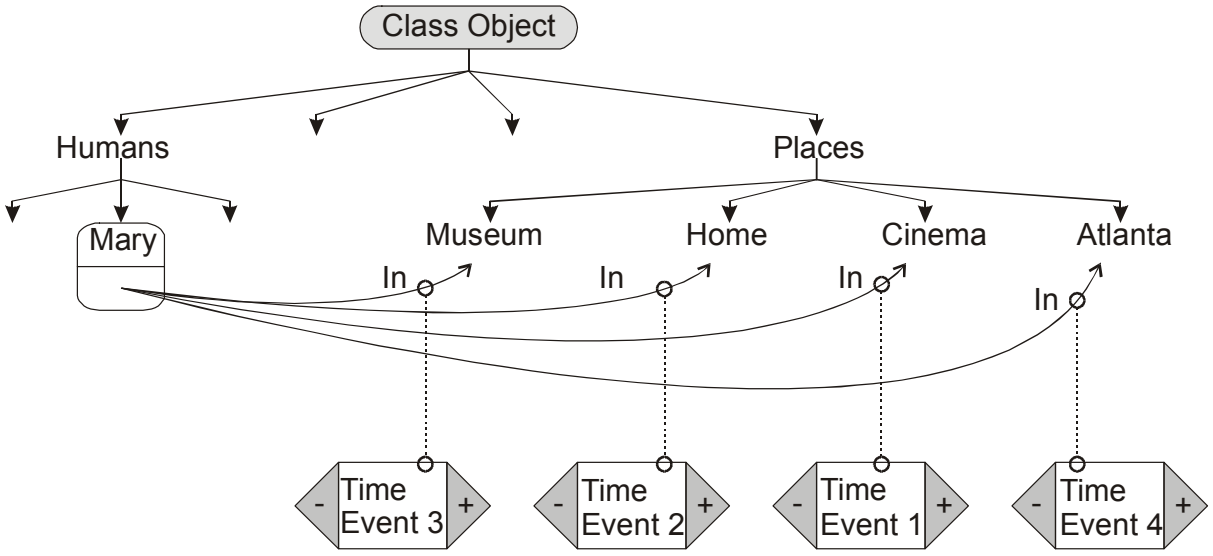


Figure 14 Connections between time events and the Semantic Network

The above figure shows the situation where Mary was (or is) at 4 different locations. If Museum, Home or Cinema can be found in Atlanta, it means then Mary can be at more than one location simultaneously.

Each time event is an object which consists of two values:

"–" the starting time of the event

"+" the ending time of the event.

Starting time should be always a constant time value, whereas ending time can be either a constant time value (if the event's ending time is known), or label "**Now**" (if the event has started, and has not yet finished). Given the "Actual Time" integrated with the system, it can distinguish three categories for time events:

- **Obsolete Events** – have both "–" and "+" before the "Actual Time"
- **Actual Events** – have "–" before the "Actual Time", and "+" either after the "Actual Time" or marked as "Now"
- **Future Events** – have both "–"and "+" after the "Actual Time".

It is quite obvious, that for obsolete events the starting/ending times are constant time values. For the Actual Events, we may know their ending times or not. In the latter case, their "+" value is marked as "Now". For some future events, their "+" or "–" values may not be known. Future event becomes the Actual Event if Future Event's "–" is known to be before the "Actual Time".

The proposed notation for relations and time events is the following:

Given a relation A(x,y,z), we may directly influence its time values by writing $A(x,y,z).T_{event}^{-} = 13^{28}$, $A(x,y,z).T_{event}^{+}$ = Now. By default, the planning algorithm considers only relations which represent the actual state of the environment. The obsolete or future relations become visible only when the planning algorithm explores a relation equipped with the time event variable. In this case the algorithm should consider all possible relations, no matter if they are obsolete, present or future ones. If the time variable is initialized then it refers to a specific time event, and only relations pointing to the same time event are considered. If the time variable is not initialized then in can unify with other time event, for example the event pointed by some relation. Below a simple example of this process is presented:

Let us suppose the planning algorithm has to satisfy the following sub-goals:

$In_{e1}(Mary, y)$ & $In_{e2}(Mary, z)$ & $During(e_1, e_2)$

If in figure 14, *Mary* was in the *Cinema* during her stay in *Atlanta*, then *y* will unify with *Cinema*, *z* will unify with *Atlanta*, e_1 will unify with *TimeEvent1* and e_2 will unify with *TimeEvent4*. Since e_1 was during e_2, the sub-goal $During(e_1, e_2)$ will be satisfied. If all Mary's relations were analyzed and no appropriate time events were found, then the planning algorithm would eventually fail to satisfy the above sub-goals.

Time events representation

The first approach to find a data structure for time events is to use a basic set of independent elements. In most AI tools like for example prolog, time events (or situations – for situation calculus) used by time predicates are treated as any other object. The unification algorithm explores the whole knowledge base searching for objects which not only represent time events, but also have specific time properties. More advanced knowledge based systems use hashing tables for different types of objects and limit their search only to time associated events. However, the set of independent time events can be so huge that big common sense knowledge systems equipped with time reasoning mechanism would need enormous hardware resources to perform real time computations.

For time events e_1 and e_2, there is an important set of time relations:

$\forall e_1, e_2$ $\quad Meet(e_1, e_2) \Leftrightarrow e_1^{+} = e_2^{-}$

$\forall e_1, e_2$ $\quad Before(e_1, e_2) \Leftrightarrow e_1^{+} < e_2^{-}$

$\forall e_1, e_2$ $\quad After(e_1, e_2) \Leftrightarrow Before(e_2, e_1)$

$\forall e_1, e_2$ $\quad During(e_1, e_2) \Leftrightarrow e_1^{-} > e_2^{-} \wedge e_1^{+} < e_2^{+}$

$\forall e_1, e_2$ $\quad Overlap(e_1, e_2) \Leftrightarrow \exists e_3\ During(e_3, e_2) \wedge During(e_3, e_1)$

If we asked the first order logic system which of the above relations is satisfied for two known events e_1 and e_2, we would obtain the answer quickly. The situation would complicate dramatically if the system had to find for us two time events which satisfy one of the above relations. In the worst case scenario it would have to examine all time events to come up with the answer.

The answer to this problem may be to store some facts about the time in the lookup table and therefore speed up the searching of specific time events. Unfortunately, if we just wanted to store the relation *After* then for a set of *n* well ordered elements we would need *n!* entries in the lookup table. Since the other relations would also need a lot of space in the lookup table, this solution is good only for small knowledge bases.

The second approach to find an appropriate data structure for time events would suggest the implementation of a resizable array of events. This data structure would surely carry the information about the relations *Meet*, *Before* and *After*, but it would say nothing about the two others: *During* and *Overlap*.

The Semantic Network's approach towards the time representation problem is a **Time Graph** depicted in figure 15. Time graph connects different time events using two types of edges:

Next ↔ Previous	During ↔ When
If the event e_1 has a link *Next* to the event e_2, then $e_1^- < e_2^-$. *Next* and *Previous* are opposite: whenever there is the edge $e_1 \xrightarrow{Next} e_2$ there also must be the edge $e_2 \xrightarrow{Previous} e_1$ etc. Both relations keep track of the sequence of events.	If the event e_1 has a link *During* to the event e_2, it means that e_2 happens during e_1. In other words: $e_1^- < e_2^- \wedge e_2^+ < e_1^+$. *During* and *When* are opposite: whenever there is the edge $e_1 \xrightarrow{During} e_2$ there also must be the edge $e_2 \xrightarrow{When} e_1$ etc. Both relations create the hierarchical structure of the time graph.

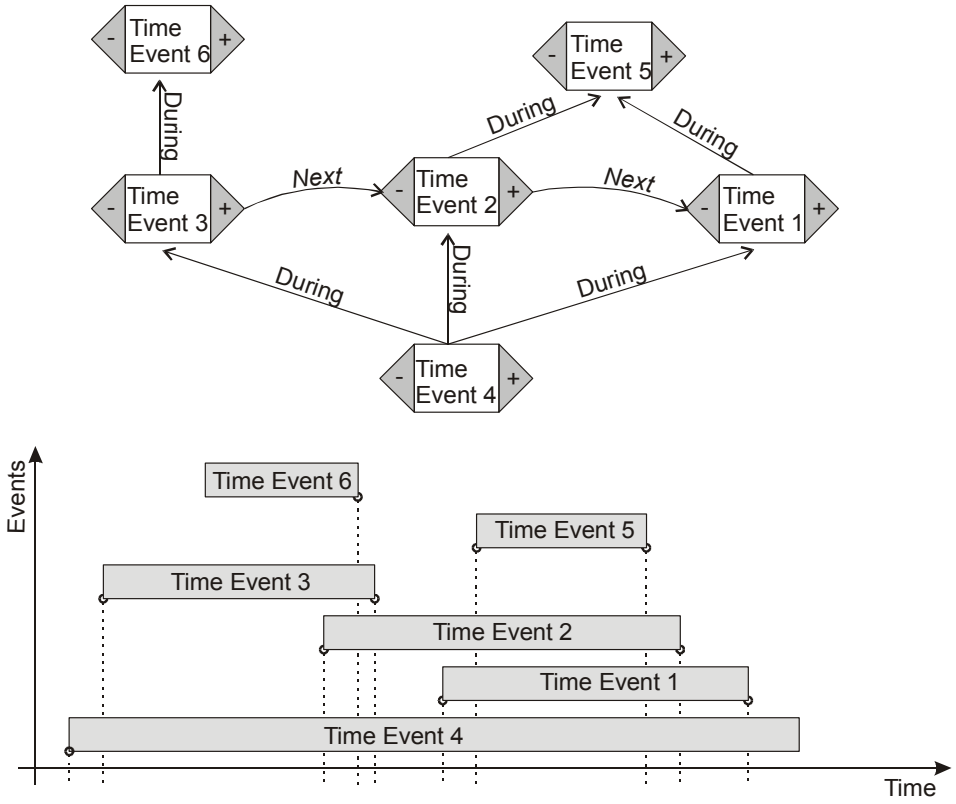


Figure 15 Time Graph and time axis

Relations Next/Previous can be called horizontal, and During/When – vertical.

The time graph can be understood as a pyramid. Its bottom is the longest, most general time event: Time. It is also the root of the time graph; all other events happen during the time root, and no event happens after or before it. The higher the pyramid we go, the shorter time events are found. If an event is at the top of the pyramid (there are many tops, confusing?) it means, that nothing happened during it. The general schema of a time event and its links is depicted in figure 16.

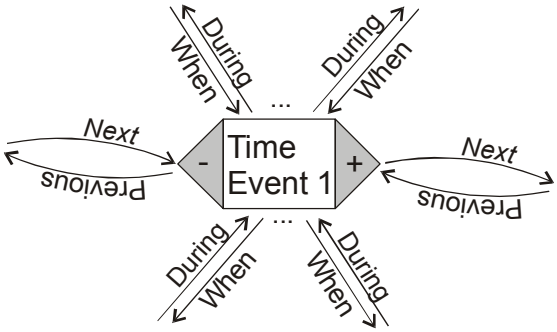


Figure 16 Time event as a Time Graph node

Relations *During* and *When* are transitive, therefore we can infer that all time events happen during root time event. Relations *Next* and *Previous* are also transitive, but in this case their transitivity is widened through the *During* relation: given the time events E_1, e_1, E_2, e_2, if $E_1 \xrightarrow{During} e_1$ and $E_2 \xrightarrow{During} e_2$ and $E_1 \xrightarrow{Next} E_2$ and $e_1 \neq e_2$, then it is obvious that $e_1 \xrightarrow{Next} e_2$.

Before we show the implementation of the Time Graph, a brief study of its advantages and disadvantages should be made. On the side of Time Graph's advantages the following points should be mentioned:

- Time Graph is a human readable representation of time events. Event big number of events can be represented in a transparent way.
- Since vertical relations apply to the Time Graph the hierarchical structure, the number of horizontal relations can be seriously reduced.
- For questions like: "When e_1 happened?" or "What happened during e_2" the answer is immediate. Moreover, because time events become more and more general when they approach the time root, the answers for the above questions are more and more general. Furthermore, if there is an event e_1 which happens during e_2 and e_3, then the fact that e_2 and e_3 overlap can be immediately inferred.

The biggest problem for the Time Graph is that it forces the hierarchical structure of time events, and therefore when a new event enters the graph it must be placed in the appropriate place updating some horizontal and vertical relations. In this section the event inserting algorithm will be depicted, along with a simple example of its execution.

The Time Graph carries some important properties:

1. **The first property of horizontal relations**. If there are time events e_1 and e_2 such that: $e_1 \xrightarrow{Next} e_2$, then not only $e_1^- < e_2^-$, but also $e_1^+ < e_2^+$. This fact is obvious, because if finishing time of e_1 was greater then finishing time of e_2, then since $e_1^- < e_2^-$ we would have $e_1 \xrightarrow{During} e_2$ which of course is false when $e_1 \xrightarrow{Next} e_2$.

2. **The first property of vertical relations**. Let us suppose there is a continuous set of sequential events: $\{e_i, e_{i+1}, \ldots, e_{j-1}, e_j\}$ and $e_i \xrightarrow{Next} e_{i+1} \ldots e_{j-1} \xrightarrow{Next} e_j$. If an event e_k is during e_i and e_j, then it must be also during $e_{i+1} \ldots e_{j-1}$. The proof of this property is indirect. Let us suppose that e_k is not during e_n ($i<n<j$). Since e_k is during e_i and e_j, then it is during the period $p = (e_j^-, e_i^+)$. Moreover, if e_n is after e_i and before e_j (property 1) then $e_n^- < e_j^-$ and $e_n^+ > e_i^+$, and therefore p is during e_n. Since e_k is during p and p is during e_n, we infer that e_k is during e_n – contradiction. Figure 17 shows the Time Graph which represents the first property of vertical relations.

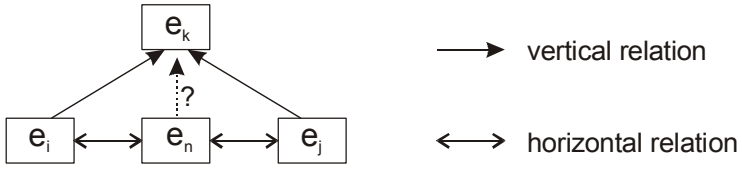


Figure 17 Improper Time Graph

3. **The second property of vertical relations**. If there are two sets of sequential events: $E_{ab}=\{e_a, e_{a+1}, \ldots, e_{b-1}, e_b\}$ and $E_{cd}=\{e_c, e_{c+1}, \ldots, e_{d-1}, e_d\}$, then for two events e_1 and e_2, if $e_1 \xrightarrow{Next} e_2$ and $E_{ab} \xrightarrow{During} e_1$ and $E_{cd} \xrightarrow{During} e_2$, then $a \le c$ and $b \le d$. Proving this property is the same as proving that e_2 is during e_d, for the graph depicted in figure 18.

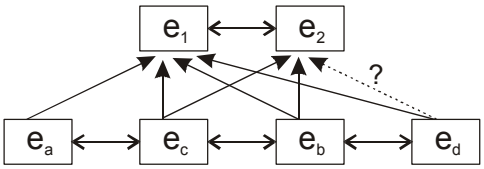


Figure 18 Second property of Time Graph vertical relations

In order to prove that e_2 is during e_d, we need to have: I) $e_2^- > e_d^-$ and II) $e_2^+ < e_d^+$.
Prove for I): If $e_1 \xrightarrow{Next} e_2$, then $e_1^- < e_2^-$. Moreover, if $e_d \xrightarrow{During} e_1$, then $e_d^- < e_1^-$. As a result we obtain $e_d^- < e_1^- < e_2^-$.
Prove for II): If $e_b \xrightarrow{During} e_2$, then $e_2^+ < e_b^+$. Moreover, if $e_b \xrightarrow{Next} e_d$, then $e_b^+ < e_d^+$. As a result we obtain $e_2^+ < e_b^+ < e_d^+$.

4. **The third property of vertical relations**. If there are time events e_1, e_2, e_3, e_4 such as: $e_1 \xrightarrow{Next} e_2$, and $e_1 \xrightarrow{During} e_3$ and $e_2 \xrightarrow{During} e_4$, but e_4 is not during e_1 and e_3 is not during e_2, then for set E_3 of all time events that happened during e_3, and for set E_4 of all events that happened during e_4 we have:

   $E_3 \xrightarrow{Next} E_4$ and
   $E_3 \cap E_4 = \varnothing$

   This property is depicted in figure 19:

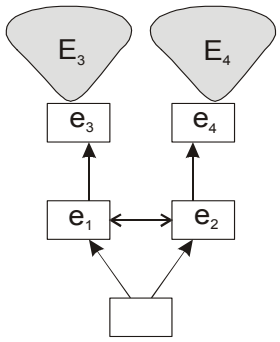


Figure 19 Third property of Time Graph vertical relations

5. **The second property of horizontal relations; graph levels connection**. One graph level consists of the nodes connected by the Next / Previous relations. The power of the Time Graph lies in the connections between time events belonging to different graph levels. If a new time event happens during e_x and e_y, where e_x and e_y are located at different graph levels, then this new event inherits all time properties for e_x and e_y. Consequently this new time event splices two different graph levels into a single level connected by Next / Previous relations. It should be remarked, that time events inherit the properties of the events they belong to, not the graph level properties.

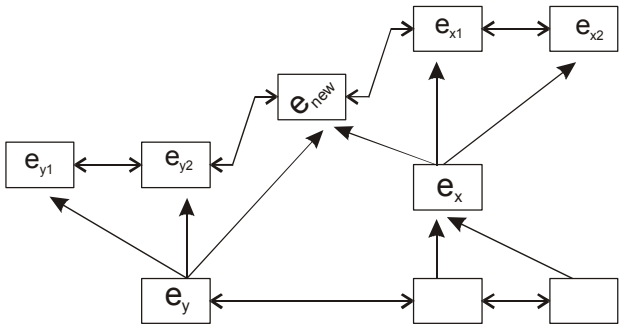


Figure 20 Time Graph level connections

Figure 20 shows an example graph where two different levels have been joined. The new, common level which consists of nodes e_{y1}, e_{y2}, e_{new}, e_{x1}, e_{x2} fulfills the properties 1, 2 and 3.

All time graph properties presented above help to understand the basis of the Time Graph functionality. Time Graph is hierarchical structure of graph levels. Whenever a new time event enters the graph, it is either inserted into a graph level, inserted between two different graph levels creating a common level, or put into a new graph level. In all cases, respective vertical relations link the new event with the events during which it happens.

Balanced Time Graph

The structure of the Time Graph along with all its properties provides one major advantage for the time representation: even with a huge number of time events, the system maintains its readability and simplicity. Instead of accumulating time relations in one node, and making the structure dense and complicated, the time graph expands nicely in horizontal

and vertical directions. As it is shown soon, the Time Graph functions use only local parts of this graph, keeping all the time their high efficiency.

Inserting a new time event

The biggest problem for the proposed Time Graph is to create an algorithm which inserts a new event into it, and updates some graph relations if necessary. The algorithm **Insert(e_{new})** depicted below consists of four stages:

1. Given a new event e_{new}, find the set *P* of parent nodes (set of events during which e_{new} occurs). Due to the fact that relation *During* is transitive, only the shortest events during which e_{new} occurs will be in P.
2. Attach e_{new} to all the nodes from the set P using the relations *During* and *When*.
3. Create or update an existing graph level formed from the events which occur during the events from the set P, and insert e_{new} into that level.
4. Add vertical relations linking e_{new} with all events which happen during this event.

Stage 1

This stage should determine the elements of set P along the way of e_{new} from the Time Root to its appropriate location. The algorithm uses a temporary set T consisting of actual events during which e_{new} occurs. Initially, the set T has only one element: TimeRoot. Then the algorithm picks in a loop each element e from T, and explores the set E of all events which occur during e. For each of those events e', it checks whether e_{new} occurs during e'. If that is the case, e' is put into T. If there is an event e' during which e_{new} occurs, e is deleted from T, in the other case e rests intact. After deleting possible duplicates from T, the algorithm checks if T has been modified in the current loop. If yes, the algorithm continues its loop, if no it terminates.

The stage one algorithm is written below:

```
Function DetermineP returns P
    T' := {TimeRoot};  T := ∅
    While T≠T' do
        T := T'
        For each e∈T do
            E:= Events during e
            For each e'∈E do
                If e' --During--> e_new then
                    T':=T' ∪ {e'} \ {e}
    Return T
```

Stage 2

This stage modifies the Time Graph by adding the following links:

$e \xrightarrow{During} e_{new}$ $\quad \forall e \in P$

$e_{new} \xrightarrow{When} e$ $\quad \forall e \in P$

Stage 3

At this stage, e_{new} will be put into an appropriate graph level. First, the set H of all events which happened during the events from P will be created. Then two events e_1 and e_2

closest to e_{new} from its both sides will be selected (if of course they exist). Finally appropriate horizontal relations will be added to the Time Graph.

Function **AddHorizontalLinks**

E:= ∅; $e_1^- := -\infty$; $e_2^- := +\infty$

For each e∈P do

H := *Events during* e

E := E ∪ H

For each e ∈ E do

If $e^- < e_{new}^-$ and $e^- > e_1^-$ then $e_1 := e$

If $e^- > e_{new}^-$ and $e^- < e_2^-$ then $e_2 := e$

If $e_1^- > -\infty$ then Add link $e_1 \xrightarrow{Next} e_{new}$ and $e_{new} \xrightarrow{Previous} e_1$

If $e_2^- < +\infty$ then Add link $e_{new} \xrightarrow{Next} e_2$ and $e_2 \xrightarrow{Previous} e_{new}$

Stage 4

At this final stage, vertical relations linking e_{new} with all events which happen during it will be added. In order to achieve it, we explore via the *During* relation all events located on the graph level of e_{new}. If we find an event e which happens during e_{new}, the relation $e_{new} \xrightarrow{During} e$ will added. To cut the branching factor of such a search, we would deal with events "on the left" and "on the right" of e_{new} separately.

Events "on the left" of e_{new} are located on the graph level of e_{new}, and start earlier than e_{new}. Suppose e_{left} is an event on the left of e_{new}. If there is an event e such that $e_{left} \xrightarrow{During} e$, then if that event starts after e_{new}, it must also be during e_{new}. If e starts and ends before e_{new}, then e and its descendants via the *During* relation cannot be during e_{new}. If e starts before e_{new}, but finishes after e_{new}, then although e cannot be during e_{new}, some descendants of e via the *During* relation can. The reasoning is analogous for the event "on the right" of e_{new}. The algorithm for stage 4 is presented below:

Function **AddVerticalLinks**

E:= ∅;

For each e∈P do

H := *Events during* e

E := E ∪ H

For each e∈E do

If $e^- < e_{new}^-$ then ExploreLeft(e)

If $e^- > e_{new}^-$ then ExploreRight(e)

Function **ExploreLeft**(e)

H := Events during e

For each h∈H do

If $h^- > e_{new}^-$ then

Add link $e_{new} \xrightarrow{During} h$

Add link $h \xrightarrow{When} e_{new}$

If $h^- < e_{new}^-$ and $h^+ > e_{new}^-$ then

ExploreLeft(h)

Function **ExploreRight**(e)

H := Events during e

For each h∈H do

If $h^+ < e_{new}^+$ then

Add link $e_{new} \xrightarrow{During} h$

Add link $h \xrightarrow{When} e_{new}$

If $h^+ > e_{new}^+$ and $h^- < e_{new}^+$ then

ExploreLeft(h)

Time Graph functions

Finally, we can demonstrate the efficiency of the Time Graph for the following questions: "What happened during the event E?", "When did E happen?", "What happened after E?" and "What happened before E?".

To make the demonstration more readable, we assume the presence of two functions: $BFS_{During}(E)$ and $BFS_{When}(E)$. BFS is a shortcut for **Breadth First Search** – the graph search algorithm, which explores the graph starting from the node N and returns a set of visited nodes. $BFS_{During}(E)$ explores the Time Graph via the *During* relation, and $BFS_{When}(E)$ explores the Time Graph via the *When* relation.

The answer for the first two questions is already done:

During(E) = $BFS_{During}(E)$

When(E) = $BFS_{When}(E)$

It can be remarked that the Time Graph structure along with Breadth First Search algorithm will return the more and more general answers for the first two questions.

Before we show how to answer the following two questions, we will introduce two auxiliary functions: LatestWhen(E) – which returns the latest event linked with E via the *When* relation, and EarliestWhen(E) - which returns the earliest event linked with E via the *When* relation. Functions A(E) and B(E) may return a set containing events which are during E, which must be deleted from the final return set.

Function **After**(E) returns P Return A(E) \ During(E)	Function **Before**(E) returns P Return B(E) \ During(E)
Function **A(E)** returns P E_1 := E ; P:= ∅ E_2 := Next(E) While exist E_2 do P:= P ∪ BFS(E_2) E_1 := E_2 E_2 := Next(E_1) Return P ∪ After(LatestWhen(E_1))	Function **B(E)** returns P E_1 := E; P:= ∅ E_2 := Previous(E) While exist E_2 do P:= P ∪ BFS(E_2) E_1 := E_2 E_2 := Previous(E_1) Return P ∪ After(EarliestWhen(E_1))

Example

We can demonstrate how the event insertion algorithm puts a new event to the Time Graph. Let us suppose we deal with a graph from Figure 15, and the new event e_{new} happens during e_6, e_3, e_2, e_4. After step 1, the set P will contain the elements e_2 and e_6. The new event will be therefore linked with *During / When* relations with e_2 and e_4. Consequently the graph level of e_{new} will contain only two sequential events: $e_{new} \xrightarrow{Next} e_5$. Since there is no event during e_{new}, the 4th step of the insertion algorithms will immediately terminate.

Now, if we asked the system a question "What happened after Time Event 3?" the *After* algorithm would go to e_2, explore it reaching e_{new} and e_5, before finally visiting e_1. Since there is no event after e_4, the algorithm would return a following set:

$P = \{e_2, e_{new}, e_5, e_1\} \setminus \{e_6, e_{new}\} = \{e_2, e_5, e_1\}$.

V. 2 Space

The Semantic Network is a directed graph whose hierarchical relations connect objects of a different type. Since all existing environment objects must have their specific locations, it is a good idea to create a Space Graph capable of integrating them into a coherent structure. The Space Graph covered in this section will be a tree of hierarchical objects connected by a relation *In*. We assume that, the root of the Space Graph will be a node *World*, and each of its children (via the *In* relation) will be a sub-area of *World*. Each sub-area will have its own children representing the locations this sub-area consists of etc. Since each node's children inherit the property of this node, the redundancy of information about the world's localizations will be eliminated.

Except for the relation *In*, some Space Graph nodes would be connected by the relation *Path*, which can be associated with bridges linking various areas. Typically every two nodes connected by the relation *In* will also be connected by the relation *Path*, therefore going from one Space Graph node to another would be possible only along the *Path* relation. Figure 21 shows an example Space Graph with some *Path* connections.

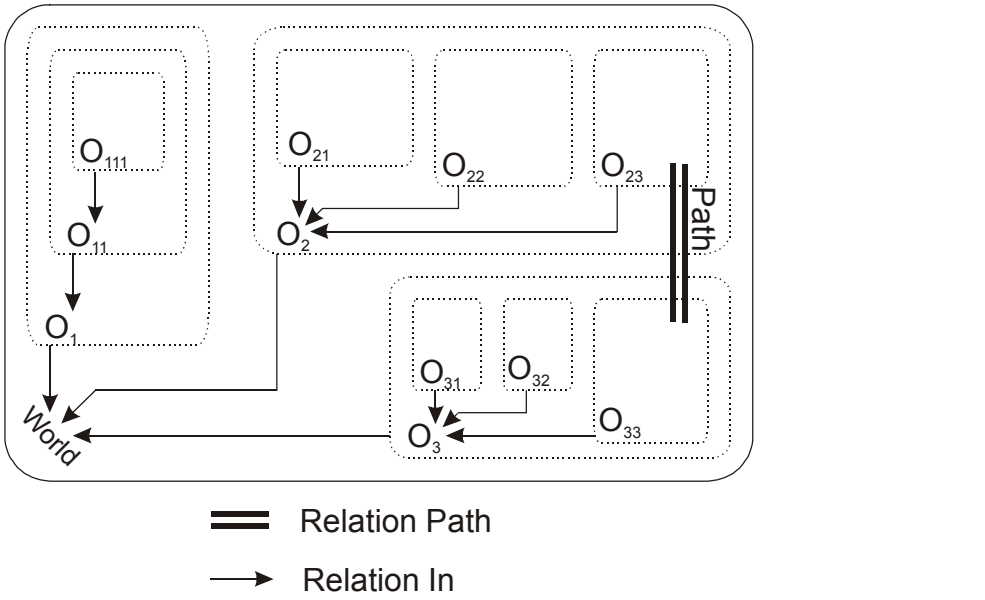


Figure 21 Space Graph

The passage from the node o_1 to o_2 would be possible if either:

- o_1 and o_2 are directly connected which means that there is the link $o_1 \xrightarrow{Path} o_2$
- o_1 and o_2 are indirectly connected which means that either
  - o_1 is located in x ($o_1 \xrightarrow{Path} x$) and there is the passage from x to o_2
  - o_2 is located in x ($o_2 \xrightarrow{Path} x$) and there is the passage from o_1 to x

Due to the fact that all existing environment objects already exist as Semantic Network objects, the Space Graph will add to the network only links representing relations *Path* and *In*.

The Semantic Network node-nested grammar productions can easily make each relation transitive. Let us assume there is relation *Rel* working on the set of objects which belong (not necessarily directly) to the class *C*. To make *Rel* transitive for all descendants of *C*, the following grammar production should be added to the node *C*:

Rel(. ,y) → Rel(. ,z) & Rel(z, y)

It is important to notice, that so far, we have been using grammar productions with unique existential variables (see Chapter "Inference"). When dealing with a production responsible for adding transitivity for *Rel* it is clear, that this production will be used many times and therefore the variable Z once unified, will always have the same value. Although the mechanism which makes these variables differ will be depicted in the Chapter "Semantic Network Algorithms", we assume it is already up and running.

This schema can be used to create the relation In_t – the *In* relation with the transitivity property.

In_t(. ,y) → In(. ,y)
In_t(. ,y) → In(. ,z) & In_t(z, y)

We assume that, if one location is in the other location, then there is a path between them:

Path(. , y) → In(. , y)
Path(. , y) → In(y, .)

The goal At(*Who*, *Where*) will be used to move the agent to the location *Where*. Since the agent can already be in *Where*, the grammar productions for At(x) located in the node *Agent* will be the following: (recall that "." is a substitution for the node the grammar production is located in)

1) At(. , x) → In(. , x)
2) At(. , x) → In(. , y) & Go(. , y, x) | *Updates*:{¬In(. , y), In(. , x)}

Finally, the productions for the goal Go(. , y, x) will be the following: (actions written in bold are the agent's low level functions)

1) Go(. , x, y) → Path(x, y) | *Actions*: (**GoTo(y)**)
2) Go(. , x, y) → In_t(x, y) & In(x, z) | *Actions*: (**GoTo(z)** , Go(. , z, y)
3) Go(. , x, y) → In_t(y, x) & In(y, z) | *Actions*: (Go(., x, z), **GoTo(y)**)
4) Go(. , x, y) → In(y, z) | *Actions*: (Go(., x, z), **GoTo(y)**)

The first grammar production is used when the two locations: x and y are joined by a *Path*. If there is no direct path between x and y, but x is somewhere deep in y (In_t(x,y)) then the agent should go to z – a one level more general location than x, and then go somehow from z to y. If there is no direct path between x and y, but y is somewhere deep in x (In_t(y,x)) then the agent should go somehow to z – one level more general location than y, and then go directly to y. If none of the above situations occur it means that neither x is a descendant of y nor y is a descendant of x. In this case the agent should go somehow to z – one level more general location than y, and then go directly to y.

We suppose the agent operating in the environment depicted in figure 21 is located in O_{33} and wants to go to O_{23}. In other words, the agent's goal is **At(O_{23})** which changes into Go(Agent ,O_{33}, O_{23}). Since there is no direct *Path* between O_{33} and O_{23}, the first *Go*

production will fail. Also, because neither x is somewhere in y, nor y somewhere in x, productions 2) and 3) will not be considered. Production 4 is applicable, because O_{23} is in O_2. Consequently the array of agents actions will be (Go(Agent, O_{33}, O_2) , **GoTo(O_{23})**). Now, the system will have to make a plan for a goal Go(Agent, O_{33}, O_2). This plan succeeds immediately because of the first grammar production for *Go*. The existence of a Path(O_{33}, O_2) guarantees a safe action **GoTo(O_2)**. Finally low level actions will be the following:

GoTo(O_2), *then* GoTo(O_{23}).

Now suppose that the agent is in the location *World* and wants to go to O_{111}. The starting goal for the planning algorithm is At(O_{111}). Since O_{111} is somewhere deep in the *World*, we can use the second *Go* production twice. At the end we use the first *Go* grammar production obtaining a nice array of low level actions:

GoTo(O_1), *then* GoTo(O_{11}), *then* GoTo(O_{111})

V. 3 Updating time and space

In the Chapter "Introduction", the function Add(*Element*, *Class*) was first presented. One instruction of this function was:

Update O's time and space properties if necessary

Now when we know how the time and space elements look like and how their respective graphs are embedded in the Semantic Network, we can present the Time/Space update process.

We assume there are two objects: *O* and *Element*, recognized as the same except for the *In* property:

In(O, loc_1)
In(Element, loc_2)

Consequently, the system assumes, that O = Element, and applies the following changes to the Semantic Network:

- In the node O, changes the existing property: In(O,loc_1).T_{event}^{+} = "actual_time"
- In the node O, adds a new relation: In(O, loc_2)
- In the node O, adds the following time properties:
  - In(O, loc_2).T_{event}^{-} = "actual_time"
  - In(O, loc_2).T_{event}+ = "now"
- Inserts the old event to the Time Graph: Insert(In(O,loc_1))

As a result, the old relation: In(O,loc_1) can be accessed only when the planning algorithm searches a goal with a time property, for example: In_e(O,loc_1) & During(e, $10^{\underline{20}}$).

Chapter VI Semantic Network algorithms

Abstract

This chapter provides the description of the principles of Semantic Network algorithms. Since node-nested grammars along with the planning algorithm provide a useful tool for creating standard programming instructions, the family of plans the system deals with expands significantly. Complex, non-deterministic plans facilitate the process of creating intelligent agents, like the Personal Human Assistant presented at the end of this section.

VI. 1 Concept of Semantic Network algorithms

Up to now, the structure of the Semantic Network could be modified either by grammar production updates, human interactions or perception & recognition element. Eventually, there can be some special programs which also influence the network, similar to grouping algorithms presented in Chapter "Grouping".

The concept of Semantic Network algorithms is an extension to the standard family of plans. The standard plan can be seen as fully deterministic whereas the Semantic Network algorithm can consist of loops, recurrent calls of goals, and standard programming instructions like case and if.

The Semantic Network algorithm is a unique form of actions description, because it incorporates the following mechanisms:

- Semantic Network knowledge – at any time, any network property can be an input value for the algorithm.
- Widespread grammar productions – at any time the algorithm can use other algorithms' parts. Each algorithm is constructed from smaller sub-goals, just like the house is constructed from the bricks.
- Widespread inference rules – at any time the inference mechanism embedded in nodes in the form of grammar productions can guide the algorithm execution process. Inference rules can also determine and update constraints for the algorithm.
- Perception and recognition element applies modifications to the Semantic Network structure influencing the input data for running algorithms.

The standard plan could initialize a specific variable only one time, therefore goals with variables could be used only once. In Chapter "Time and Space" we introduced the grammar productions for the action Go(. x, y) which were inevitably used more than once. Now, the small modification to the planning algorithm which fixes this problem will be presented.

Program counter

Let us suppose there is grammar production A(x,y,z) which becomes an actual goal. We can expect some (not necessary all!) variables to unify after the Explore function succeeds. Now, imagine the same grammar production is used again. In this case, since the set of unifications Ω already contains the entries for x, y or z, new unifications for those variables would lead to ambiguity.

To deal with this problem, we should distinguish the variables of a specific production used more than once. In other words, whenever a new production is taken into consideration, its variables should become unique for this production and their substitutions should not yet be present in the set Ω. The solution to this problem is to modify future production variables to make them unique before the production is analyzed. The easiest modification consists in attaching to each variable a label associated with the planning process **program counter** (pc).

The planning process program counter is an integer variable which represents the number of successful calls of the *Explore* function along the current planning process. The variable *pc*, zeroed at the beginning of each planning process should be passed to recurrent *PickSubGoal* and *Explore* calls just like the set of Semantic Network updates *U*.

Let us suppose there is a set of goals *F* and *pc's* value *k* when the planning process analyses the function *PickSubGoal*. For each permutation of goals, the temporary value pc_{temp} should be initialized to *pc*. The variable *pc incremented by 1* should be passed to every *Explore* function call and allow the *Explore* function to modify it. If for some of goals from the set F the *Explore* function returns false, *pc* should be set back to pc_{temp}. If it returns true, variable *pc* obtained from the *Explore* function should become actual.

Now, suppose the planning process analyses the *Explore* function. For each grammar production *G(n,i)*, where *n* is the network node number, and *i* is this node grammar production number, the algorithm locally attaches a specific label to each variable of *G(n,i)*. This label is the actual value of *pc*. Then, the production *G(n,i)* is analyzed just as if nothing has happened. Variable pc should be passed to every *PickSubGoal* function call, allowing this function to change *pc* value.

The use of the program counter finally eliminates the problem associated with ambiguous variable names and allows the planning algorithm to process the same goals more than once. Moreover, goals can be launched recursively, one from inside another.

Extracting the program counter value

As it is presented soon, it is comfortable to be able to know the value of the program counter at the specific stage of the planning process. When having a specific variable initialized to the program counter value, we can mark different blocks of a Semantic Network program to make them signify the same in the future.

So far the value of *pc* has been useful only for the planning algorithm, not for grammar productions lying on the planning process. The aim is to initialize a grammar production variable with the current value of the program counter. The solution consists in making a small adjustment to the planning algorithm, more precisely, to the first instruction of the $Explore_{enhanced}$ function introduced in the Chapter "Inference". The first line of this function was responsible for returning false, if the $Explore_{enhanced}$ function starting point was a variable. The modification consists in extracting the pc value when the $Explore_{enhanced}$ function goal is PC(*variable*). In order to achieve it, the first instruction:

If Variable(Obj) then return false

Should be extended into:

```
If  Variable(Obj) then
        If Rel = PC then
                Obj = pc
                Return true
        Return false
```

With all the above modifications, if there is a set of goals:
PC(x) & A(. , x) & B(. , x)

Then, even when using this goal multiple times, respective sub-goals A and B will have a common point represented by a specific program counter's value x.

Algorithms as network plugins

Since the Semantic Network algorithms are the extended versions of standard plans, they can be spread all over the network. To be able to distinguish which of the plans constitute the algorithms, we can create a Semantic Network node *Algorithm* incorporating plans associated with algorithms. Since the planning algorithm explores the network nodes in a search for specific relations, the introduction of different nodes for different algorithms seems a reasonable idea. The figure 22 shows how the set of algorithms can be placed in the network:

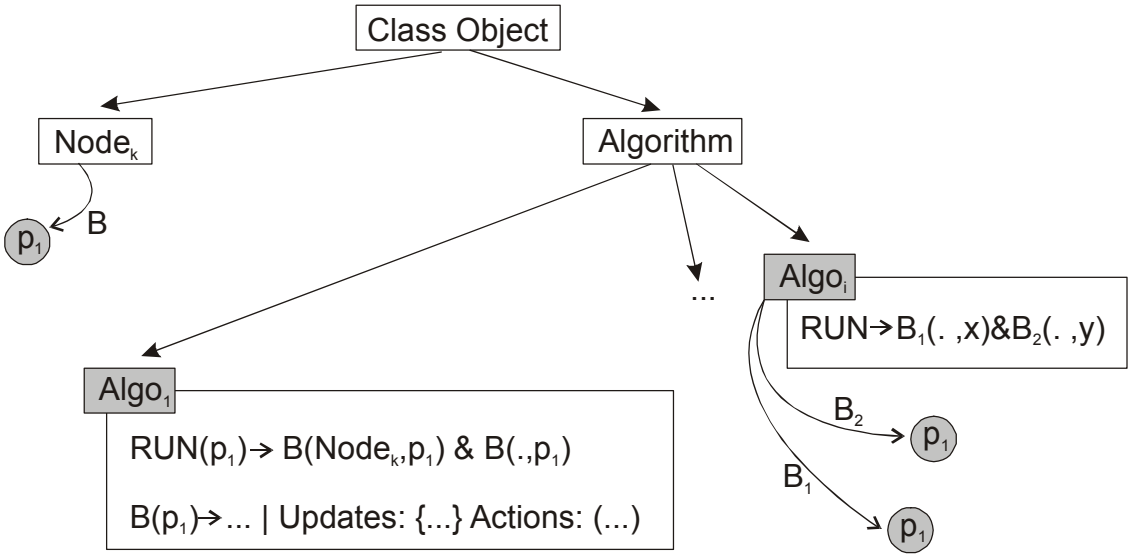


Figure 22 Semantic Network algorithm node

Every algorithm consists of its local variables and different blocks represented by respective grammar production which can be either sub-goals from different parts of the network or other grammar productions from the algorithm node. Because some algorithm parts, like loops etc. are characteristic only for one specific algorithm, their location in one node does not influence the other parts of the network keeping the network efficiency at the same level.

If each algorithm has the following starting goal: RUN(p_1, p_2, …p_n), then each grammar production can launch any algorithm in a standard way. Let us suppose there is the network node Node$_1$ with a production:

A(x, arg$_1$, …, arg$_k$) → B(x) & C(x) | *Actions*: (RUN(Algo$_i$, arg$_1$, …, arg$_k$))

Then if B(x) and C(x) are satisfied, the action RUN(Algo$_i$, arg$_1$, …, arg$_k$) will have to be executed after the current planning process terminates. Since RUN(Algo$_i$, arg$_1$, …, arg$_k$) is not a low level action, it will become a starting goal for the planning algorithm. The search for that goal will begin in the node Algo$_1$. If the Algo$_1$ is attached to the network, that it will have a production RUN(par$_1$, par$_2$, …, par$_k$). Consequently, each argument arg$_i$ could unify with parameter par$_i$, and the algorithm Algo$_i$ can start.

There are two approaches to write the sequent instructions of the algorithm:

- **Sequential actions**: In this approach, the sequence of actions is achieved through grammar productions *Actions* element. Let us suppose the program $Algo_i$ consists of two blocks: B_1 and B_2, and B_1 must be executed before B_2. In order to achieve this sequence of goals we might write the following grammar productions in the node $Algo_i$:

$Algo_i$
RUN → B_1(.) | *Actions*: (B_2(.))
B_1 → …
B_2 → …

  The biggest advantage of this approach lies in the fact that the planning algorithm tries to satisfy simple goals, then terminates and tries to satisfy another simple goal. The number of recurrent calls of the *Explore* and *PickSubGoal* function is reduced, and the set of unifications is zeroed for every new program block.

  However, the forecast for the success of a plan is also limited, and the planning process behaves just like the standard computer program: it executes sequent instructions (blocks) without considering the fact, that some future block would lead it to crash. If the planning process crashes, the actions it has made so far cannot be canceled what can cause many troubles.

- **Sequential sub-goals**: In this approach, the sequence of actions is achieved through the order of goals the planning algorithm searches. The program blocks must be written is such a way, that it is possible to terminate the program without considering other permutation of goals (blocks). Let us suppose the program $Algo_i$ consists of two blocks: B_1 and B_2, and B_1 must be executed before B_2. The standard grammar production:

$Algo_i$
RUN → B_1(.) & B_2(.)
B_1 → …
B_2 → …

  will be sufficient to execute all the program blocks. The biggest advantage of this approach lies in the fact that, if at some point a block cannot be executed, then the planning algorithm can return false without executing the set of actions it has gathered so far. Consequently the forecast for future instructions is bigger and the algorithm is more stable.

  However, much greater resources are needed to store the sets of unifications, actions and Semantic Network updates for nested recurrent calls of the *PickSubGoal* and *Explore* function. Moreover, some sophisticated grammar production tricks should be used to make the planning algorithm accept only the initial order of goals. In other words, no other permutation of goals (blocks) should be allowed. In the next section some of those tricks will be used for sequential sub-goals algorithms.

The best algorithm usually combines the elements from both approaches.

VI. 2 Standard Instructions

This section will provide a brief study of the standard programming instruction. Since the description of all programming instructions is beyond the scope of this paper, only three of them will be presented: if, case and while.

Instruction If

The general schema of the *If* instruction is the following:

B_1: If <case> then B_2 else B_3

When the algorithm $Algo_i$ using the sequential actions approach has to execute the block B_1, then its grammar productions must be the following:

$Algo_i$
$B_1 \rightarrow$ <case> | *Actions*: (B_2(….))
$B_1 \rightarrow$ | *Actions*: (B_3(…..))

The <case> segment can be of any form: a network property, an inference rule or another algorithm (plan). If the <case> element returns true, then goal B_1 is satisfied, and B_2(….) becomes a new action. If <case> returns false, the planning algorithm analyzes the second grammar production which is always true since it does not introduce new goals. Consequently B_3(….) becomes the new action.

When the algorithm $Algo_i$ using the sequential sub-goals approach has to execute the block B_1, then its grammar productions must be the following:

$Algo_i$
$B_1 \rightarrow$ PC(p) & IF(. ,p) & RESTIF(. ,p) | *Updates*: {¬TRUE(. , p), ¬FALSE(. , p)}
IF(p) → <case> | *Updates*: {TRUE(. , p)}
IF(p) → B_3(….) | *Updates*: {FALSE(. , p)}
RESTIF(p) → FALSE(. , p)
RESTIF(p) → B_2(….)

First the program counter value p is extracted to join the elements of this *If* instruction. Properties TRUE(p) and FALSE(p) attached to the $Algo_i$ node guide the planning process through key grammar productions of the *If* instruction. Regardless of the logical value of <case>, plan B_1 always succeeds. This *If* instruction can be used in the planning process multiple times (even recursively!), because the program counter assures the unique values for the properties TRUE and FALSE. Each *If* instructions segment performs the cleaning of its variables TRUE / FALSE after their usage.

Instruction Case

The general schema of the *Case* instruction is the following:

$CASE_i$: <$case_1$> : B_1
<$case_2$> : B_2
……..
<$case_n$> : B_n
<else> : B_e

When the algorithm $Algo_i$ using the sequential actions approach has to execute the block $CASE_i$, then its grammar productions must be the following:

$Algo_i$
$CASE_i \rightarrow <case_1> \mid \textit{Actions}: (B_1(\ldots.))$
$CASE_i \rightarrow <case_2> \mid \textit{Actions}: (B_2(\ldots.))$
……..
$CASE_i \rightarrow <case_n> \mid \textit{Actions}: (B_n(\ldots.))$
$CASE_i \rightarrow \mid \textit{Actions}: (B_e(\ldots..))$

The $CASE_i$ segment is similar to *If* instruction segment. The only difference is that, for different cases, different actions are executed. Whenever one of the cases is fulfilled, its corresponding grammar production is taken. Consequently other grammar productions for other case values are omitted.

When the algorithm $Algo_i$ using the sequential sub-goals approach has to execute the block $CASE_i$, then its grammar productions must be the following:

$Algo_i$
$CASE_i \rightarrow PC(p)\ \&\ C_1(\ .\ ,p)\ \&\ \ldots\ \&\ C_n(\ .\ ,p)\ \&\ C_e(\ .\ ,p) \mid \text{Updates: } \{\neg TRUE(\ .\ ,p)\}$
$C_1(p) \rightarrow TRUE(\ .\ ,p)$
$C_1(p) \rightarrow \neg <case_1>$
$C_1(p) \rightarrow B_1 \mid \textit{Updates}: \{TRUE(\ .\ ,p)\}$
……..
$C_n(p) \rightarrow TRUE(\ .\ ,p)$
$C_n(p) \rightarrow \neg <case_n>$
$C_n(p) \rightarrow B_n \mid \textit{Updates}: \{TRUE(\ .\ ,p)\}$
$C_e(p) \rightarrow TRUE(\ .\ ,p)$
$C_e(p) \rightarrow B_e(\ .\ ,p) \mid \textit{Updates}: \{TRUE(\ .\ ,p)\}$

The above set of instructions uses the negation of consecutive cases to guide the planning process through the CASE instruction. When goal C_i is being explored, the algorithm checks if there has been some satisfying $<case_k>$ ($k<i$) so far. If no, it checks the condition $\neg<case_i>$. If this conditions does not hold, the instruction block B_i is executed and the variable TRUE(. ,p) is set to omit remaining case conditions ($<case_k>$ for $k>i$). After the end of CASE instructions segment, the variable TRUE(. ,p) is cleaned for future use.

Instruction While

Due to the fact that the instruction *While* can be an everlasting loop, there is a big risk to implement it using the sequential sub-goals approach, because the multitude of loops could generate the multitude of substitutions and unifications for all recurrent calls. Although this approach is possible, the sequential actions approach for a *While* instruction is only presented.

The general scheme of the *While* instruction is the following:
$WHILE_i$: while <case> do B_1

When the algorithm $Algo_i$ using the sequential actions approach has to execute the block $WHILE_i$, then its grammar productions must be the following:

$Algo_i$
$WHILE_i \rightarrow$ <case> | *Actions*: ($B_1(….)$, $WHILE_i(.)$)
$WHILE_i \rightarrow$

As long as a condition <case> is satisfied, the planning process executes an action $B_1(…)$ followed by an action $WHILE_i(.)$. The latter action is a loop instruction itself. If the condition <case> is not fulfilled, then the planning algorithm analyzes the second grammar production which immediately terminates the loop.

Optimizing Semantic Network algorithms

One of the ways to optimize the Semantic Network algorithms is to divide the algorithm set of productions into sub-categories. Due to the fact that some algorithm segments can use only a small part of this algorithm's grammar productions, we may allocate different algorithm segments in its respective sub-nodes. As a result of that, the planning algorithm has to deal with smaller sets of grammar productions which improve the efficiency of the planning process associated with the algorithm. The division of $Algo_i$ into its sub-programs is shown in figure 23.

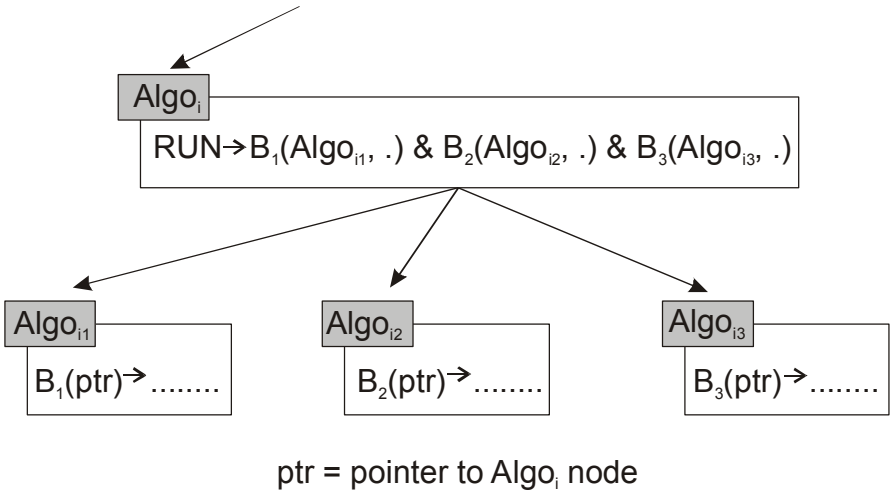


Figure 23 Decomposition of a Semantic Network algorithm

VI. 3 Basic algorithms

Neural networks

The Semantic Network along with its algorithms can be used to simulate almost every type of neural network. The neural network structure can be written by using Semantic Network nodes and relations connecting them. The neural network result can be associated with a respective grammar production responsible for launching a specific Semantic Network algorithm.

Most Neural networks can be emulated using the basic programming tools, like loops and condition instructions. Loops are used when a neural network learns from the facts carried by the set of examples. Conditional elements are used to implement neuron's activation

function. The neural network weights can be written in the Semantic Network as respective relations connected to neural network nodes. All those key elements of each neural network can be written as Semantic Network algorithms as it was shown before.

Any Semantic Network object can have either a fixed relation, or a grammar production pointing to a specific neural network which would return a value on the basis of this object's properties. Whenever there is a new training example available for the neural network, it can be passed over to this network's learning algorithm responsible for updating neural network nodes. The implementation of basic neural networks using Semantic Network algorithms is beyond the scope of this paper, and will not be covered in this section.

Reinforcement learning

Semantic Network algorithms are also a good platform to implement other artificial intelligence tools, like reinforcement learning. Since the nature of grammar productions allow the implementation of Markov processes, many of them can be integrated with the network. The following reinforcement learning tasks can be easily implemented:

- Actions with a feedback – a feedback of each action can be written as a grammar production network update.
- The perception element can scan the environment for changes made by a specific action and if they exist, encode them in grammar production's update section.
- Action strategy can be encoded using Semantic Network algorithms. Loops and conditional instructions along with the perception of action results may be used to incorporate in the Semantic Network complex learning strategies.
- Using advanced Semantic Network algorithms, there is a possibility to encode strategies which analyze all possible ways of achieving a goal and pick the best one.
- Action feedback (promotion) value can vary during the learning process.

Deep first search

Due to the fact that the Semantic Network is a huge graph whose objects are linked by various relations, the algorithm which explores this graph may by very useful. The graph search algorithm must be universal: there are a lot of different relations and there can be many starting points for the exploration process. The Deep First Search algorithm which performs these tasks can be written entirely in Semantic Network grammar language.

The recursive algorithm written in pseudo-language code is the following:

```
DeepFirstSearch(x)
        MARKED[x] = TRUE
        Do Action with x.
        For each node y adjacent to the node x via the relation Rel do
                If MARKED[y] = FALSE then DeepFirstSearch(y)
```

The algorithm uses a table of visited nodes noted as MARKED. For each explored node, the **Action** is executed. To perform the graph exploration via the relation *Rel* starting from the point SP, we ought to write: DeepFirstSearch(*SP*).

The Deep First Search algorithm written in Semantic Network grammar language will be a powerful tool allowing the exploration of the Semantic Network objects from any network point via any network relation. For each explored object, not just a single action, but a complex plan can be executed.

The algorithm temporary values can be stored in the algorithm node *DFS* attached to the node *Algorithm* (see fig. 24).

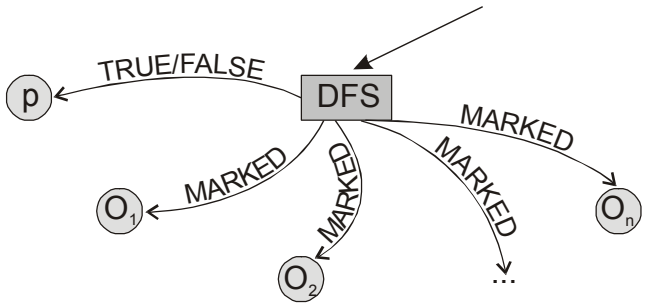


Figure 24 DFS algorithm temporary values

Let us suppose the algorithm starting point is **SP**, the relation connecting visited objects is **Rel**, and the action for each explored object is **Act**. The following set of grammar productions executes the DFS and then cleans its temporary variables.

DFS

RUN(SP,Rel,Act) → EXPLORE(. ,SP,Rel,Act) & CLEAN(.)
EXPLORE(SP,Rel,Act) → PC(p) & IF(. ,p,SP,Rel,Act) & RESTIF(. ,p,SP,Rel,Act)
|*Updates*: {¬FALSE(. , p)}
IF(p,SP,Rel,Act) → ¬MARKED(. ,SP)
IF(p,SP,Rel,Act) → |*Updates*: { FALSE(. ,p)}
RESTIF(p,SP,Rel,Act) → FALSE(. ,p)
RESTIF(p,SP,Rel,Act) → VISIT(. ,SP,Rel,Act) & SEARCH(. ,SP,Rel,Act)
VISIT(SP,Rel,Act) → |*Updates*: {MARKED(. ,SP)} *Actions*: (**Act(SP)**)
SEARCH(SP,Rel,Act) → **Rel(SP,x)** & ¬MARKED(. ,**x**)
|*Actions*: (EXPLORE(. ,**x**,Rel,Act), SEARCH(. ,**SP**,Rel,Act))
SEARCH(SP,Rel,Act) →
CLEAN → MARKED(. ,x) | *Updates*: {¬MARKED(. ,x)} *Actions*: (CLEEN(.))
CLEAN →

The above set of grammar productions nicely combines the Semantic Network programming elements shown earlier in this section. It uses both programming approaches: sequential actions (ex. for clean-up routine) and sequential sub-goals (ex. when checking if an object has been explored). Furthermore, the presented algorithm combines the execution of actions with reasoning: the inference mechanism justifies if a new action should be added to the array of actions.

There are few points marked as bold in the above productions. Firstly, **Act(SP)** is responsible for executing a specific action for the node *SP*. This action is fully independent from the DFS algorithm and can take place in a completely different part of the network. Moreover, there is no limit for the form of *Act*: it can be either a low level action, more

complex plan or a Semantic Network algorithm. Since **Act(SP)** is called using the sequential actions approach, its sets of updates, unifications and actions are local. **Act(SP)** can be easily extended to have more parameters before the DFS algorithm starts.

Rel(SP,x) is a goal which can be fulfilled only if there is a physical link $SP \xrightarrow{Rel} x$ between two Semantic Network nodes: *SP* and *x*. Since *x* is a variable, it can unify with any Semantic Network node adjacent to SP. EXPLORE goal is responsible for searching for new objects in-depth, whereas SEARCH goal is responsible for searching for new objects in-breadth.

VI. 4 Personal human assistant

The last section of this chapter demonstrates how an intelligent agent based on Semantic Network might look like. The agent will be equipped with several grammar productions simulating some human like behaviors in specific, predefined environment, therefore its name can be Personal Human Assistant.

Personal Human Assistant can be of a great use for all alone people living in a closed environment, whose disabilities make it hard (or impossible) for them to perform everyday tasks. Due to the variety of possible tasks, only few of them will be covered in this section.

Switching the lights off

Personal Human Assistant can easily perform a task which involves switching off the lights in a certain location. Let us recall from Chapter "Time & Space", that the location of all existing objects was represented by the relation IN. We can create a relation invIN reflexive to IN:

invIN(x,y) ↔ IN(y,x)

Let us suppose there is the space object House incorporating other objects like Kitchen, Hall, Boy's room etc. If in a certain location there is a light switch object, then it possesses either property "*ON*" or property "*OFF*". Naturally since a light switch is a physical object, it must be located somewhere. Consequently, this light switch must be linked to some more general location with the help of the relation *IN*. If the agent wants to turn off the light switch, it should be exactly at the location of this switch.

Because all house locations are connected in a tree with the common root *House*, the DFS algorithm introduced earlier can easily explore every part of this house:

RUN(*DFS, House, invIN, TurnTheLightOff*)

The action *TurnTheLightOff* should be performed whenever the agent knows, that a light switch is in a current location. The agent should in that case go to that location and perform a low level action **switch**. If in a current location there is no light switch, or the light is already off, the agent should not go to this place. All these premises force the following form of the action *TurnTheLightOff:*

TurnTheLightOff(object) → ON(object) & At(Agent, object)
| *Actions*: (**switch**) *Updates*: {¬ON(object), OFF(object)}
TurnTheLightOff(object) →

The second production is used to produce zero action if the explored location (object) is either not a light switch or a light switch already turned off.

Surveying the house

Personal Human Assistant can be also used for interactive surveying tasks. If a surveying object is a House, then the agent should repeatedly visit all locations in the house and in case of danger, apply specific actions. The everlasting house exploration process can be written in the following way:

Surveillance(*House*) → |*Actions*: (RUN(*DFS,House,invIN,CheckLoc*), Surveillance(*House*)

Now for each location *L* found by the DFS algorithm, the agent will execute an action *CheckLoc(L)*. Since it remains unknown if there is an emergency in the location L, the agent should first go to L, and then execute the scan for possible dangers.

CheckLoc(*object*) → At(Agent, *object*) | *Actions*: (EmergencyScan(*object*))

The EmergencyScan for a location *object* can have a different form, depending on the nature of the *object*. For most house locations, the standard scanning routine will have the following schema:

EmergencyScan(*object*) → FIRE(*object*) | *Actions*: ((Call Fire Department))
EmergencyScan(*object*) → SMOKE(*object*) | *Actions*: ((Call Fire Department))
EmergencyScan(*object*) → IN(Burglar,*object*) |*Actions*: ((Spray paralyzing gas), (Call Police Department))

However, some locations like the Child's room may have more sophisticated scanning routines:

EmergencyScan(*Child'sRoom*) → IN(*Child*, *Child'sRoom*) & CRY(*Child'sRoom*) |Actions:(At(*Agent*, *Parent'sRoom*), SAY(Child is crying))

The properties FIRE / SMOKE / CRY can be determined by the perception and recognition element of the agent.

Bring me something!

The last action covered in this section will be associated with a following human order: "Bring me an object O". In order to make the agent cope with this order, a set of additional grammar productions must be added to the agent's knowledge base.

First, we must teach the Personal Human Assistant how to search an object in a specific location. In other words, the agent must know how to satisfy a goal:

FIND(*What, Where*)

Naturally, we would like the agent to stop exploring sequent locations if the object's location is already known. The two following grammar productions cope with the problem of finding an item *Object* in the location *Location* (the second production is an instruction *If* which consists of several smaller productions, as it was shown earlier in this chapter).

FIND(*Object*, *Location*) → RUN(*DFS, Location, invIN*, $\text{Action}_{\text{find}}$(*Object*))

$Action_{find}$(*Object*, *loc*) → IF IN(*Object*, *loc*) THEN <nothing>
ELSE At(*Agent*, *loc*)

For all locations found by the DFS algorithm, action $Action_{find}$ is executed. As long as the *Object* location remains unknown, the agent explores sequent locations associated with variable *loc*.

If the agent knows the location of *Object*, then although its DSF algorithm keeps launching actions for all remaining locations, these actions are empty.

With the implementation of FIND(*What, Where*) done, we may focus on the implementation of the goal MOVE(*What, Where*).

Whenever the agent has to satisfy the goal MOVE(*Object, Location*) it must first determine the initial location of Object. Only if this location is known, it can pursue the MOVE process. The grammar productions responsible for fulfilling this goal are as follows:

MOVE(*Object*, *Location*) → IN(*Object*, *Agent*) & At(*Location*) | Actions: (Put(*Object*))
MOVE(Object, Location) → IN(*Object*, loc_0) & At(*Agent*, loc_0)
|*Actions*: (Pick(*Object*), GO(loc_0, *Location*), Put(*Object*))
Updates: {¬IN(*Object*, loc_0), IN(*Object*, *Location*)}

The first production is used in case the agent already carries the *Object*. The second production forces the agent to go to the initial *Object* location, pick it, and then transport it to the *Object* destination Location.

Finally, we can present a description of a process which fulfills a goal BRING(*Object*, *Location*):

BRING(*Object*, *Location*)→ B_1(*Object*) | *Actions*: (B_2(*Object*, *Location*))
B_1(*Object*)→ IF ¬IN(*Object*, *loc*) THEN FIND(*Object*, *House*)
B_2(*Object*, *Location*)→ MOVE(*Object*, *Location*)
GET(*Object*) | *Actions*: (At(*Location*), Put(*Object*))
GET(*Object*)→ BUY(*Object*)
GET(*Object*)→ CREATE(*Object*)
GET(*Object*)→ BORROW(*Object*)
……..

At the end, we have presented a complete mechanism responsible for bringing an object to a specific location. The description of a goal BUY(…) was shown back in the Chapter "Planning".

Summary

Personal Human Assistant is able to do many other interesting actions which are not covered in this section, like for example:

- It can answer the questions about the events that happened in the past:
  Sleepy(*Child*) → CRY_e(*Child*) & DURING($Night_{25\text{-}04\text{-}2003}$, e)

- It can create new algorithms by listening to the humans, for example:
  "Go to Mary's house, and if she is there give her a letter. If she is not in her house, send the letter at the post office. "

As a result, the Semantic Network becomes a powerful tool for fixing new problems defined in a human like language in a relatively short time.

Chapter VII Conclusions

VII. 1 Recapitulation

This paper was an attempt to demonstrate the effectiveness of semantic networks when developping intelligent agents. The Chapter "Introduction" introduced what such a network might look like and which knowledge base elements it could integrate. Node-nested grammars were introduced to increase the expressiveness of the Semantic Network.

In the Chapter "Planning", the simplest version of the most important algorithm of this paper was shown – the Semantic Network planning algorithm. This algorithm, although slightly enhanced in Chapters "Intefence" and "Semantic Network Algorithms" was sufficient for performing most intelligent tasks, like learning, making inferences or constructing plans.

The Chapter "Inference" investigated the possibility of equipping the Semantic Network with two different inference mechanisms: based on generalized modus-ponens and resolution rule. It was prooved, that the first mechanism can be emulated by the planning algorithm and node-nested grammar productions. The implementation of the complete first-order logic inference system to work with the Semantic Network was also covered.

Two effective grouping algorithms were shown in the Chapter "Grouping". We demonstrated their effectiveness and applicability when managing the strucure of the autonomous Semantic Network.

Chapter "Time & Space" explained how to implement the Time Graph and Space Graph in the Semantic Network. Time Graph was a brave attempt to connect various time events in a well-organized, human readable graph structure. Space Graph showed the basis of how to represent locations in the Semantic Network.

Chapter "Semantic Network Algorithms" finally demonstrated how to construct an intelligent agent based on Semantic Network. It was depicted how to create Semantic Network algorithms written in a special programming language – the language of Semantic Network grammar productions.

All these integral parts of the Semantic Network could succesfully simulate the behaviour of intelligent agents. Thanks to unified ontology and structure, agents can communicate performing various tasks in different parts of the network.

VII. 2 Final remarks

"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 (Personal Human Assistant) it is likely, that 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.

The Semantic Network ability to clone agents and send them in no-time to collect information from various network parts could constitute the biggest advantage of computers over humans.

VII. 3 Future applications

There were some points in this work where it was written that further investigation of a problem is not within the scope of this paper. These points can constitute in the future some challenging tasks for scientists. Here we depict some of them:

- Process interactions – it might be a good idea to allow processes to clone, create and destroy other processes. The specific precess interactions mechanism could be the subject of a future work.
- Reverse planning process – the algorithm guessing plan goals from observed array of actions and network states could be a powerful AI tool.
- Study of all possible Horn sentences the Semantic Network modus-ponens mechanism is able to work with.
- Semantic Network resolution strategy – the concept of sub-graphs associated with resolution sentences lying above the network structure could be further developed.
- Analyse of the efficiency of the Time Graph could lead to some interesting results for huge sets of time events. The question if Time Graph complexity in linear with the arrival of new time events is still without an answer.
- Study of more sophisticated programming instructions for Semantic Network algorithms.
- Detailed implementation of neural nerworks and other specialized AI tools in Semantic Network programming language.

The biggest project would be however to implement the presented artificial intelligence system on real hardware. As soon as this project is up and running, a great variety of AI tasks can be the domain of engineers, not scientists.

GLOSSARY

Semantics	Science about the meaning of words describing the world
semantic network	Directed acyclic graph whose nodes and arcs represent different words of a language. Local nodes and arcs provide the semantic description of a local region of the world or a domain of live
Semantic Network	Written with capital letters – the semantic network with the structure being described in this work
Plan	Ordered sequence of operations with restrictions associated with each operation
grammar production	Rewriting rule embedded in a certain node. Left side of grammar production can be substituted by its right side. Grammar production consists of network relations.
Semantic Network relation	Either network node's property or a semantic expression on the array of semantic words
Semantic Network update	Update to the knowledge base associated with the Semantic Network
Semantic Network action	Either a low level action or a network relation
low level action	Atom action performed by the agent associated with the Semantic Network
Semantic Network algorithm	More complex plan described by a set of Semantic Network grammar productions