Knowledge abstraction using heuristic search I: A specialization approach

D. Riaño

Computer Science/Department of Algorithmics and Architecture

Report CS-R9469 December 1994

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Knowledge Abstraction using Heuristic Search I: A specialization approach

David Riaño

Universitat Rovira i Virgili, Carretera Salou s/n, 43006 Tarragona, Spain

E-mail: fibcls09@lsi.upc.es, drianyo@etse.urv.es

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

This work consists of two parts: first, the study of what the goodness of a descriptive conjunctive rule is, and second the application of several well-known searching strategies to the problem of finding the best conjunctive rule describing one problem. For the first purpose, three linear families of goodness functions are introduced and tested. For the second purpose, backtracking, best-first, hill-climbing, and A* algorithms are designed and tested independently, studing their possible prunings and heuristic functions. The work finishes with a comparison of the four algorithms in terms of both, the goodness of the rule produced, and also the cost of producing this rule in time.

AMS Subject Classification (1991): 68P10, 68T05, 68T30, 68T35.
CR Subject Classification (1991): I.2.4, I.2.6, I.2.8.
Keywords & Phrases: knowledge representation, heuristic search, Lisp structures, conceptual learning.
Note: This work was partially developed and finished at the Centrum voor Wiskunde en Informatica, afdeling Algoritmiek en Architectuur, Kruislan 413, 1098 SJ Amsterdam, The Netherlands.

1. Introduction

This report is about the application of *Heuristic Search* methods to construct *Knowledge in the Form of Rules.* Traditionally rules come represented in the form '**if** *premise* **then** *conclusion*' where *premise* describes a set of objects where the rule is applicable to via *modus ponens* . At the moment of deciding the sort of premise to use, one may opt for different types of expressions: *conjunctive forms*, *disjunctive normal forms* (DNF), *conjunctive normal forms* (CNF) and some others not so spread out. The problem created here is the generation of an expression in *conjunctive* form describing a subset (Θ^+, or *positive examples*) of elements within a certain domain (Θ) with the possibility of excluding a group of them (Θ^-, or *negative examples*). The expression obtained will have to be expressed in terms of the descriptors allowed ($\mathcal{D}$) and will have to maximize the *goodness function* that we will analyze further on.

Report CS-R9469
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Briefly we are able to show a scheme of the problem in the following form:

input:	Θ^+ set containing the objects to describe Θ^- set containing the objects to reject $\mathcal{D}$ set of the descriptors allowed
output:	'Best' *conjunctive predicate* which can be built up in terms of the descriptors in $\mathcal{D}$.

The goodness function, put in charge of giving sense to the concept of 'best', can be affected by several factors, some of them are enumerated here:

- Degree of acceptance and rejection of the *positive examples.*
- Degree of acceptance and rejection of the *negative examples.*
- Degree of acceptance and rejection of the elements in the *Domain.*
- *Simplicity* of the conjunctive expression.
- *Complexity* variable of the descriptors.

It should be indicated that the sets of *positive examples* and *negative examples* do not have to define an exhaustive partition of the domain; this is, it is possible that we have $\Theta^+ \cup \Theta^- \neq \Theta$ (see figure 0.1) although unavoidablely $\Theta^+ \cap \Theta^- = \emptyset$ (i.e. not noise-tolerant methodologies).

The subtitle of *specialization approach* is to indicate that this work is concerned with algorithms which, from the most general expression, and after successive refinements, arrive to a final expression [1]. With this restriction we do not consider all those methods that use *generalization* as a tool to construct expressions as well as all the possible hybrids of the two techniques.

A good approach to the understanding of this work is to consider the sections and algorithms in isolation and not to try to understand it as a whole.

Section 2 gives a brief overview to the knowledge representation in conjunctive form, giving rise, in section 3, to a serie of algorithms aiming at the task of searching the 'best' conjunctive expression; both algorithms that perform from blind search among all the possibilities (*exhaustive methods*) and heuristic algorithms, also considering *Hill-Climbing*, are studied. Section 4 is to compare all these methods in terms of *temporal cost* and *results obtained.* The last section (§5) is employed to list some new ideas on the process of constructing rules and consider the possibility of learning from *unknown*, *imprecise*, and *uncertain* data.

2. Conjunctive Rules and *goodness* functions

A *conjunctive classification rule* is defined as a rule expressing its *premise* in conjunctive form. A conjunctive classification rule is considered to be *active* when each one of the *conjunctants* in the premise is *true.* An *object* is said to *fire* a *rule* if the premise of the rule defines

[1] Historically a great number of systems use this approach: AQ, C4, CN2, CABRO, FOIL, PRISM, etc.

a generalization of the description of the object. From a point of view of set theory, a conjunctive classification rule in the form '*if* C *then* Θ^+' (also, $C \rightarrow \Theta^+$) divides the domain Θ into six subsets (figure 0.1):

$$
\begin{aligned}
[C]^+ &= \{\theta \in \Theta^+ : \theta \textit{ fires the rule}\} \\
[C]^- &= \{\theta \in \Theta^- : \theta \textit{ fires the rule}\} \\
\wr C \wr^+ &= \{\theta \in \Theta^- : \theta \text{ does not } \textit{fire} \text{ the rule}\} \\
\wr C \wr^- &= \{\theta \in \Theta^+ : \theta \text{ does not } \textit{fire} \text{ the rule}\} \\
|C|^+ &= \{\theta \in \Theta \setminus (\Theta^+ \cup \Theta^-) : \theta \textit{ fires the rule}\} \\
|C|^- &= \{\theta \in \Theta \setminus (\Theta^+ \cup \Theta^-) : \theta \text{ does not } \textit{fire} \text{ the rule}\}
\end{aligned}
$$

2.1 Goodness of a rule

Among the factors enumerated in the introduction as possible contributors to the computation of the *goodness* of a conjunction within a domain, we will put our attention on the first two (*acceptance indexes* and *rejection indexes*) keeping the rest constant (in the future they will be applied as a *second* criterion when the first one is not able to come to a decision). Expressed in terms of the 'lexicographic evaluation functional with tolerances' (LEF) [1], $Goodness(C) = < f([C]^+, [C]^-, \wr C \wr^+, \wr C \wr^-, |C|^+, |C|^-), \tau_1;\ simplicity(C), \tau_2;\ complexity(C), \tau_3 >$.

The *acceptance indexes*, represented by the sets $[C]^+$, $[C]^-$, and $|C|^+$ and the *rejection indexes*, represented by $\wr C \wr^+$, $\wr C \wr^-$, and $|C|^-$ [2], define the goodness function in the form:

$$Goodness(C) = F([C]^+, [C]^-, \wr C \wr^+, \wr C \wr^-, |C|^+, |C|^-)$$

Whereas the *set of positive examples* accepted by a conjunction C ($[C]^+$), as though the set of rejected negative examples ($\wr C \wr^+$), both contribute positively to the magnitude of the goodness, the *set of negative examples* accepted ($[C]^-$), and the set of positive examples rejected ($\wr C \wr^-$) act to the detriment of the goodness of the rule. It is remarkable the meaningless character of the sets *neutral examples* accepted ($|C|^+$), and neutral rejected ($|C|^-$) which is abstracted in the sentence '*whenever an element is not an example (nor counterexample) of the concept to learn, the learning process is independent of such element*', i.e.

$$Goodness(C) = F([C]^+, [C]^-, \wr C \wr^+, \wr C \wr^-)$$

Concretely, in this work we assume that all the elements within $[C]^+$ contribute in the same way to the computation of the goodness, likewise the elements within $[C]^-$, $\wr C \wr^+$ and $\wr C \wr^-$ do. This means that the goodness function can be defined in terms of the respective cardinalities; i.e. $[\![C]\!]^+$, $[\![C]\!]^-$, $\wr\!\wr C \wr\!\wr^+$, and $\wr\!\wr C \wr\!\wr^-$.

$$Goodness(C) = f([\![C]\!]^+, [\![C]\!]^-, \wr\!\wr C \wr\!\wr^+, \wr\!\wr C \wr\!\wr^-)$$

[2] Traditionally they get the respective names of *true positives* (TP), *false positives* (FP), *false negatives* (FN) and *true negatives* (TN). The two sets symbolized with vertical bars, traditionally ommited, will be called *uncertain positives* (UP) and *uncertain negatives* (UN) respectively.

2.2 Properties of a goodness function
Goodness of a rule

Property 1 *Any goodness function reaches the greatest value in the case of rules accepting all the positive examples and none negative, and the least value for rules accepting all the negative examples and none positive.*

Property 2 *Any goodness function is monotonically increasing with respect to* $[\![C]\!]^+$ *and* $\wr C \wr^+$;

$$\forall c_1, c_2 \in \wp^{\mathcal{D}} : [\![c_1]\!]^- = [\![c_2]\!]^-, \text{ and } \wr c_1 \wr^- = \wr c_2 \wr^-,$$

$$(a). [\![c_1]\!]^+ \geq [\![c_2]\!]^+ \Rightarrow Goodness(c_1) \geq Goodness(c_2) \quad (2.1)$$
$$(b). \wr c_1 \wr^+ \geq \wr c_2 \wr^+ \Rightarrow Goodness(c_1) \geq Goodness(c_2). \quad (2.2)$$

Property 3 *Any goodness function is monotonically decreasing with respect to* $[\![C]\!]^-$ *and* $\wr C \wr^-$;

$$\forall c_1, c_2 \in \wp^{\mathcal{D}} : [\![c_1]\!]^+ = [\![c_2]\!]^+, \text{ and } \wr c_1 \wr^+ = \wr c_2 \wr^+,$$

$$(a). [\![c_1]\!]^- \geq [\![c_2]\!]^- \Rightarrow Goodness(c_1) \leq Goodness(c_2) \quad (2.3)$$
$$(b). \wr c_1 \wr^- \geq \wr c_2 \wr^- \Rightarrow Goodness(c_1) \leq Goodness(c_2). \quad (2.4)$$

Goodness of combined rules In this section the goodness for complex rules is studied. One complex rule is composed of single conditions combined with the operators **and** and **or**. When two or more conditions are joined with an **and** operator, the resulting condition describes only those examples described by all the single conditions. The **or** operator describes all the examples which are described by some of the single conditions.

Property 4 *Define an order relation over the set of the parts of* $\mathcal{D}$*:*

$$\forall c_1, c_2 \in \wp^{\mathcal{D}} : Goodness(c_1) \geq Goodness(c_2) \; or \; Goodness(c_2) \geq Goodness(c_1)$$

Property 5 (*upper bound) For all goodness function* $g : \wp^{\mathcal{D}} \rightarrow \Re^+$,

(a). $g(C_1 \textbf{ and } C_2) \leq g(C_1) + g(C_2)$
(b). $g(C_1 \textbf{ or } C_2) \leq g(C_1) + g(C_2)$

Proof: directly obtained from property 7 (see later).□

Property 6 (*lower bound) For all goodness function* $g : \wp^{\mathcal{D}} \rightarrow \Re^+$,

(a). $0 \leq g(C_1 \textbf{ and } C_2)$
(b). $0 \leq g(C_1 \textbf{ or } C_2)$

Proof: considering the definition of function g, or from property 7 and taking into account the *upper bound.*□

Three examples of the goodness function The first approximation to the computation of the *goodness* for a conjunction C is based on figure 0.1 and consists of the linear combination of the number of positive examples ($[\![C]\!]^+$) with the number of negative examples ($[\![C]\!]^-$) that it accepts, as well as with the positive examples ($\|\!| C |\!|^+$), and negative examples ($\|\!| C |\!|^-$) that it rejects. The result is a *linear goodness* function, combined by means of the coefficients a_1, a_2, b_1, and b_2 described in table 0.1 [3] :

$$f_1(tp, fp, tn, fn) = (a_1 \cdot tp - a_2 \cdot fp) - (b_1 \cdot fn - b_2 \cdot tn)$$

This representation of the goodness of a rule takes values within the interval Imf_1 ($\equiv [-a_2 \cdot \#\Theta^- - b_1 \cdot \#\Theta^+, a_1 \cdot \#\Theta^+ + b_2 \cdot \#\Theta^-]$) expressed in function of the number of positive and negative examples taken. If we want to limit the goodness function to the interval $[a, b]$ previously selected and independent of the problem domain, a function in the following form will be needed:

$$f(tp, fp, tn, fn) = k_1 \cdot f_1(tp, fp, tn, fn) + k_2$$

where

$$k_1 = \frac{b - a}{(a_1 + a_2) \cdot \#\Theta^+ + (a_2 + b_2) \cdot \#\Theta^-}$$

$$k_2 = \frac{(a \cdot a_1 + b \cdot b_1) \cdot \#\Theta^+ + (b \cdot a_2 + a \cdot b_2) \cdot \#\Theta^-}{(a_1 + a_2) \cdot \#\Theta^+ + (a_2 + b_2) \cdot \#\Theta^-}$$

Concretely and for the interval $Imf_2 = [0, 1]$ the *normalized linear goodness* is obtained;

$$f_2(tp, fp, tn, fn) = \frac{(a_1 + b_1) \cdot tp + (a_2 + b_2) \cdot tn}{(a_1 + b_1) \cdot \#\Theta^+ + (a_2 + b_2) \cdot \#\Theta^-}$$

Unlike the above functions whose construction is the result of trying to control their boundaries, a third approach to the definition of a feasible linear goodness function is to take into consideration the two ideas underlying Property 1. The result, called *standarized linear goodness*, is:

$$f_3(tp, fp, tn, fn) = \frac{(a_1 \cdot tp - a_2 \cdot fp) \cdot b_1 \cdot \#\Theta^+ + a_2 \cdot (b_1 \cdot fn - b_2 \cdot tn) \cdot \#\Theta^-}{a_1 \cdot b_1 \cdot (\#\Theta^+)^2 - a_2 \cdot b_2 \cdot (\#\Theta^-)^2}$$

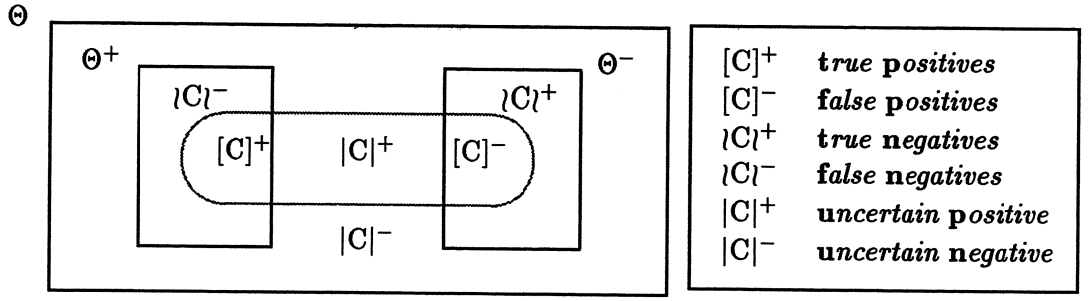
whose valuation is limited to the interval Imf_3 ($\equiv [-k_1 \cdot \max\{b_1 \cdot \#\Theta^+ , b_2 \cdot \#\Theta^-\} , k_2 \cdot \max\{a_1 \cdot \#\Theta^+ , a_2 \cdot \#\Theta^-\}]$) where k_1 and k_2 are the constants

$$k_1 = \frac{a_2 \cdot \#\Theta^-}{a_1 \cdot b_1 \cdot (\#\Theta^+)^2 - a_2 \cdot b_2 \cdot (\#\Theta^-)^2} \quad k_2 = \frac{a_1 \cdot \#\Theta^+}{a_1 \cdot b_1 \cdot (\#\Theta^+)^2 - a_2 \cdot b_2 \cdot (\#\Theta^-)^2}$$

[3] Notice that table 0.1 is a simplification of the general table,

	Θ^+	Θ^-	$\Theta \setminus (\Theta^+ \cup \Theta^-)$
acceptance	a_1	a_2	a_3
rejection	b_1	b_2	b_3

defined over the partition of the domain shown in figure 0.1.



When applied a predicate C over the domain Θ, this last comes divided into four significant subsets: $[C]^+$ or *positive examples accepted*, $[C]^-$ or *negative examples accepted*, $\langle C\rangle^+$ or *negative examples rejected*, and $\langle C\rangle^-$ or *positive examples rejected*. Among all four sets, only the first and the third have a positive contribution to the goodness (property 2) meanwhile the rest contribute negatively to the goodness (property 3).

Figure 0.1: Basic partition of the domain.

When a detailed analysis is done about the meaning of the sets tp, fp, tn, and fn (figure 0.1) the equivalences "$fn = \#\Theta^+ - tp$" and "$tn = \#\Theta^- - fp$" are obtained and the goodness functions change to the form;

$$
\begin{aligned}
f_1(tp, fp) &= (a_1 + b_1) \cdot tp - (a_2 + b_2) \cdot fp - b_1 \cdot \#\Theta^+ + b_2 \cdot \#\Theta^- \\
f_2(tp, fp) &= \frac{(a_1 + b_1) \cdot tp + (a_2 + b_2) \cdot (\#\Theta^- - fp)}{(a_1 + b_1) \cdot \#\Theta^+ + (a_2 + b_2) \cdot \#\Theta^-} \\
f_3(tp, fp) &= \frac{(a_1 \cdot \#\Theta^+ - a_2 \cdot \#\Theta^-) \cdot b_1 \cdot tp + a_2 \cdot (b_1 \cdot \#\Theta^+ - b_2 \cdot \#\Theta^-) \cdot (\#\Theta^- - fp)}{a_1 \cdot b_1 \cdot (\#\Theta^+)^2 - a_2 \cdot b_2 \cdot (\#\Theta^-)^2}
\end{aligned}
$$

An interesting study of the goodness functions consists on notice their behaviour for the basic premises: *true* and *false*.

$$
\begin{cases}
Goodness_1(true) = a_1 \cdot \#\Theta^+ - a_2 \cdot \#\Theta^- \\
Goodness_1(false) = b_2 \cdot \#\Theta^- - b_1 \cdot \#\Theta^+
\end{cases}
$$

$$
\begin{cases}
Goodness_2(true) = \frac{(a_1+b_1)\cdot\#\Theta^+}{(a_1+b_1)\cdot\#\Theta^+ + (a_2+b_2)\cdot\#\Theta^-} \\
Goodness_2(false) = \frac{(a_2+b_2)\cdot\#\Theta^-}{(a_1+b_1)\cdot\#\Theta^+ + (a_2+b_2)\cdot\#\Theta^-}
\end{cases}
$$

$$
\begin{cases}
Goodness_3(true) = \frac{a_1\cdot b_1\cdot(\#\Theta^+)^2 - a_2\cdot b_1\cdot\#\Theta^+\cdot\#\Theta^-}{a_1\cdot b_1\cdot(\#\Theta^+)^2 - a_2\cdot b_2\cdot(\#\Theta^-)^2} \\
Goodness_3(false) = \frac{a_2\cdot b_1\cdot\#\Theta^+\cdot\#\Theta^- - a_2\cdot b_2\cdot(\#\Theta^-)^2}{a_1\cdot b_1\cdot(\#\Theta^+)^2 - a_2\cdot b_2\cdot(\#\Theta^-)^2}
\end{cases}
$$

$$
Goodness_2(true) + Goodness_2(false) = Goodness_3(true) + Goodness_3(false) = 1
$$

An alternative to the above functions is the valuation of goodness taking into account the concepts of *sensitivity*, *specificity*, *positive predictive value*, *negative predictive value*, and *accuracy* introduced in [2] and [3]:

	Θ^+	Θ^-
acceptance	a_1	a_2
rejection	b_1	b_2

Table 0.1: Representation of the weights: a_1 represents the weight given to the fact of *doing right*, b_1 alternatively represents the weight of the fact *doing wrong*, a_2 the weight of *don't do it right*, and b_2 the weight of *don't do it wrong*.

	f_1	f_2	f_3
sensitivity	$a_1 = 1/\#\Theta^+$ $a_2 = b_1 = b_2 = 0$	a_1, b_1 free $a_2 = b_2 = 0$	$a_2 = 0$ a_1, b_1, b_2 free
specificity	$a_1 = a_2 = b_1 = 0$ $b_2 = 1/\#\Theta^-$	$a_1 = b_1 = 0$ a_2, b_2 free	$b_1 = 0$ a_1, a_2, b_2 free
accuracy	$a_1 = b_2 = 1/(\#\Theta^+ + \#\Theta^-)$ $a_2 = b_1 = 0$	$a_1 + b_1 = 1$ $a_2 + b_2 = 1$	$a_1 = b_1 = a_2 = b_2 = 1$

Table 0.2: Globalization of *sensitivity*, *specificity*, and *accuracy* ratios.

$$
\begin{aligned}
\textit{sensitivity}(tp, fp, tn, fn) &= \frac{tp}{\#\Theta^+} \\
\textit{specificity}(tp, fp, tn, fn) &= \frac{tn}{\#\Theta^-} \\
\textit{positive predictive value}(tp, fp, tn, fn) &= \frac{tp}{tp + fp} \\
\textit{negative predictive value}(tp, fp, tn, fn) &= \frac{tn}{tn + fn} \\
\textit{accuracy}(tp, fp, tn, fn) &= \frac{tp + tn}{\#\Theta^+ + \#\Theta^-}
\end{aligned}
$$

See the table 0.2 to have a definition of some of the above functions in terms of the functions *linear goodness*, *normalized linear goodness*, and *standarized linear goodness*. The table shows the values for the coefficients a_1, a_2, b_1, and b_2 used to make feasible such definition. Given that our three models here are linear models, the concepts of *predictive values* (as much positive as negative) are expressable by no means.

One interesting case to analyze is the way in which *accuracy* is interpreted for *normalized linear goodness* (f_2); here the concepts of *wrong* and *right* are similar to probability variables which define the coefficients a_1, a_2, b_1, and b_2 as the probabilities $a_1 = P_r(x \in [C]^+)$, $a_2 = P_r(x \notin [C]^+)$, $b_1 = P_r(x \in [C]^-)$, and $b_2 = P_r(x \notin [C]^-)$.

3. Algorithms

This section is devoted to the presentation of a list of algorithms in increasing difficulty, all of them coping with the task of inducing knowledge from raw data. The approach used for this purpose is from the standpoint of *heuristic search* where the compilation of a rule is seen as a process of searching the most relevant descriptors which conform it.

Principle 1 (*Commutativity of* **'and'**) *the conjunctive operator* **and** *is commutative.*

Principle 2 (*Absorption of* **'and'**) *For all conjunction* C, $[C] \equiv [C \textbf{ and } C]$.

Property 7 *Under a representation of the goodness as a linear combination of the concepts 'be right', 'be wrong', 'don't be right', and 'don't be wrong'; the goodness of the intersection comes in the form:*

$$Goodenss(C \textbf{ and } C') = Goodness(C) + Goodeness(C') - Goodness(C \textbf{ or } C') \qquad (3.5)$$

Proof: It is an aftermath of the linearity of the function.□

Property 8 *Under the representation of the goodness as a normalized linear combination* (f_2), *the goodness of the intersection comes in the form shown in (3.5).*

Property 9 *Under the representation of the goodness as a standarized linear combination* (f_3), *the goodness of the intersection comes expressed in the form (3.5).*

3.1 Exhaustive Methods

They produce the best conjunctive rule, the one that best represents the positive examples and worst the negative examples. Methods under this name realize a search across the *whole* searching tree till the conjunctive expression that maximize the choosen *goodness function* is found. The space in which the searching is achieved would be in the order of $\mathcal{O}((n+1)!)$ (i.e. all the possible variations of terms defined on the set $\mathcal{D}$) if it was not possible to use some factors (*pruning reasons*) discussed in the next lines.

But, before going on such pruning reasons, two new concepts are helpful to characterize searching methods in this work. While pure exhaustive methods share a *syntactic* behaviour, when a pruning strategy is applied, the algorithms can come supported by some kind of background information, drawn out of the data, or supplied by an external agent. Those last methods are defined as *semantic* methods.

First Pruning Reason As a consequence of the *commutativity of 'and'* it is only necessary to explore the space of the possible combinations of descriptors in $\mathcal{D}$. On the contrary, in the spaces of search that are not commutative each one of the permutations for all the conjunctions have to be explored. The scheme in figure 0.2 shows a case where the exploration of the right subtree is meaningless since $[d_1 \textbf{ and } d_2 \textbf{ and } \mathcal{A}] \equiv [d_2 \textbf{ and } d_1 \textbf{ and } \mathcal{A}]$, for any tree $\mathcal{A}$.

$$\frac{\sum_k \binom{\#\mathcal{D}}{k}}{2^{\#\mathcal{D}}} = \frac{\#\mathcal{D}!}{2^{\#\mathcal{D}}} \cdot \sum_k \frac{1}{(\#\mathcal{D}-k)!}$$

This first pruning reason reduces the space of search in a factor $\frac{\#\mathcal{D}!}{2^{\#\mathcal{D}}} \cdot \delta$ of the space in a blind search. To have an more concrete idea, if the number of descriptors allowed was 10, we should have to pass from a searching between $\sum_{k=0}^{10} 10!/(10-k)!$ (= 9.869.141) expressions to a searching in a space of 2^{10} (= 1.024) expressions ($\approx$ 99.99% of reduction).

Second Pruning Reason This pruning tries to analyze the case depicted in figure 0.3 where the searching down the branch d_j can be avoided under several conditions on the already explored paths (conjunctions in the form 'C **and** d_i **and** $\mathcal{A}$'). The most general condition outcomes from the formula (3.5) which stablishes that the goodness of C **and** C' (for any C') is greater than the goodness of the single expression C when and only when the goodness of C' is greater than the goodness of the disjunction C **or** C'.

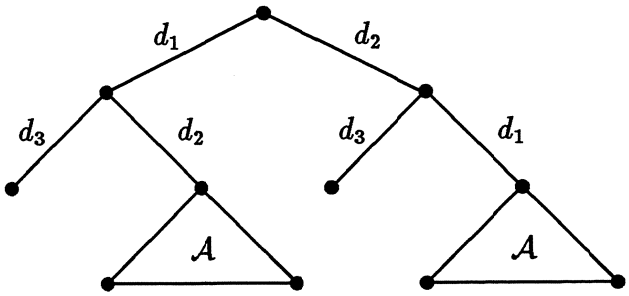


Figure 0.2: *First Pruning Reason* Diagram.

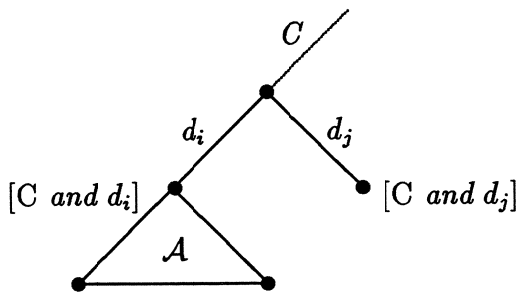


Figure 0.3: *Second Pruning Reason* Diagram.

Third Pruning Reason Analizes the pruning of a whole tree only considering the first searching level. It comes schematized in figure 0.4 and tries to answer the question *what do the sets* [C **and** d_i] *have to obey for any set in the form* [C **and** C'] *to have a goodness inferior to the goodness of* [C]*?*

Neither the second nor the third trimming reasons are easily applicable to the goodness functions introduced here. Instead, they are trivially employed in cases where goodness is computed as much in terms of the *simplicity* of the conjunctive expression (*minimal description*) as in terms of the *complexity* of the descriptors, as was mentioned in the introduction.

Some other trimming reasons are feasible: the concept of *incompatibility* defined as the impossibility of having a conjunctive premise containing a certain subset of descriptors (e.g. the concept descriptors `voluminous` and `two-dimensioned` cannot come together in the same conjunction since they define an *incompatibility*); as the concept of *disabling* defined as a

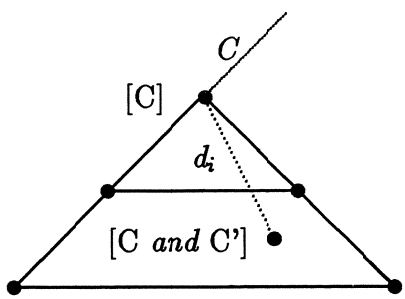


Figure 0.4: *Third Pruning Reason* Diagram.

hierarchycal incompatibility (e.g. the concept descriptor `not-supreme` disables the possibility of having `maximum` but not in the contrary) are two classical examples of semantic trimming reasons.

Naive Exhaustive Search It consists on the method called until this moment *blind exhaustive search*, where by mean of *backtracking* strategy, an analysis of all the possible variations defined on the descriptors is achieved. Two approaches to this philosophy are implemented, the first in the form of a *hard-memory-consumer* algorithm (`NESsc`) where all the combinations are compiled at once, and the second algorithm (`NESi`) in the form of a *sequential in-order generator* of all the premises.

In order to make this algorithm self-contained, the system should have a function `makelist` producing a list in the form (1 2 ... d) where d is the positive integer input parameter, and a function `but-nth` capable of remove the n*th* element of a list.

Algorithm 1: Naive Exhaustive Search (space consumer).

```
(defun NESsc (D)
    (cond ((null D) '(nil))
          (t (cons
               nil
               (apply #'append
                      (mapcar
                        #'(lambda (n)
                            (mapcar
                             #'(lambda (y) (cons (nth (- n 1) D) y))
                             (NESsc (but-nth n D))))
                        (makelist (length D))))))))
```

This second algorithm does need not only the set of possible descriptors $\mathcal{D}$, but also a conjunctive premise being the seed to produce the next immediate premise in the sense of an inorder search through the searching tree.

Algorithm 2: Naive Exhaustive Search (incremental).

```
(defun NESi (conj S)
  (let* ((last-S
           (remove-if #'(lambda (x) (find x conj :test #'equalp)) S))
         (last-1 (butlast conj))
        )
    (cond
      ((null (car conj)) (list (car S)))
      ((null last-S)
         (if (equalp (reverse conj) S)
             conj
```

```
        (do* ((l-1  (last conj)           (last prem))
              (l-2  (last (butlast conj)) (last (butlast prem)))
              (prem (butlast conj)        (butlast prem)))
             ((> (position (car l-1) S) (position (car l-2) S))
              (append (butlast prem)
                      (list (nth 1 (member (car l-2)
                                           (remove-if
                                             #'(lambda (x)
                                                 (find x (butlast prem)
                                                       :test #'equalp)) S)
  ) )   )    ))       )     ))
  (t (append conj (list (car last-S)))))))
```

The *cost* of the NESsc algorithm will come in the form of a function on the number of possible descriptors $\mathcal{D}$, i.e. d,

$$Cost(d) = d \cdot d_* \cdot Cost(d-1) + d + d_*$$

where $d_* = \begin{cases} 1 & \text{if makelist, but-nth, and nth are O(1).} \\ d & \textit{otherwise.} \end{cases}$

This cost is understood to be $\mathcal{O}(d! \cdot d_*!)$.

The cost of applying the second algorithm should be computed as the cost of applying $\mathcal{O}(d!)$ times the algorithm NESi. This is, $\mathcal{O}((d+1)!)$.

First Pruning Reason Search It only takes profit of the *fist pruning reason* and consists on the generation of all the commutative forms that are able to be generated from $\mathcal{D}$, attending the first pruning reason.

Algorithm 3: First Pruning Reason Search (space consumer).

```
(defun FPRsc (D)
   (cond ((null D) '(nil))
         (t (cons
              nil
              (apply #'append
                     (mapcar #'(lambda (n)
                                 (mapcar #'(lambda (y)
                                             (cons (nth (- n 1) D) y))
                                         (FPRsc (from-nth n D))))
                             (makelist (length D))))))))
```

Example: Let's imagine a concept we want to describe in terms of the five *descriptors* a, e, i, and o. Purely syntactic methods like *NESsc* and *FPRsc* algorithms will produce the following combinations:

```
(NESsc descriptors) => (true a ae aei aeio aeo aeoi ai aie aieo
  aio aioe ao  aoe aoei aoi aoie e ea  eai eaio eao eaoi ei eia
  eiao eio eioa  eo eoa eoai eoi eoia i ia iae iaeo iao iaoe ie
  iea ieao ieo ieoa io ioa ioae ioe ioea o oa oae oaei oai oaie
  oe oea oeai oei oeia oi oia oiae oie oiea)

(FPRsc descriptors) => (true a ae aei aeio  aeo ai aio  ao e ei
  eio eo i io o)
```

Whereas the naive methodology produces 65 possible premises, the first pruning application reduces the possibilities to 15 possible premises (76.92% of reduction).

One alternative is the algorithm for *sequential generation* which allows a sequentiality at the moment of generating the searching tree, such that this one can be explored in depth, step by step. It has the particularity of permit an exploration from any point within the search tree; this is, if we dispose of *a priori* knowledge in the form "the elements we want to describe agree for de descriptors d_1, d_2, ...", the exploration is not needed to be started from nil, and the call `(FPRi '(d1 d2 ...) rest-of-descriptors)` will produce the next possible expressions where 'd1 **and** d2 **and** ...' keeps fixed.

Algorithm 4: First Pruning Reason Search (incremental).

```
(defun FPRi (conj S)
  (let* ((Last-S (cdr (member (car (last conj)) S)))
         (Last-1 (butlast conj))
        )
    (cond ((null (car conj)) (list (car S)))
          ((null Last-S)
             (if (null Last-1) conj
                 (append  (butlast Last-1)
                          (list (nth 1 (member (car (last Last-1)) S))))))
          (t (append conj (list (car Last-S)))))))
```

3.2 Hill-Climbing or Gradient

The *Hill Climbing* methodology is a symplistic greedy method technique where decisions are taken locally with all the information available at such moment and with no reconsideration. This is, when a decision is made there is no form to reconsider it. This particular behaviour is what figure 0.5 is reflecting. Of course, this method do not ensure the best expression, but only an approximation.

Hill-Climbing algorithm is the first *heuristic* method introduced. From the diagram in figure 0.5 two main outcomes arise:

- the algorithm is *not independent* to the state.
- a *special function* is needed to decide the next state.

The first idea, related to the concept of *semantic search*, comes reflected in the argument `state` in the algorithms `HCSsc` and `HCScs`. This argument describes not only the set of feasible descriptors, but also the sets of *true positive*, *false positives*, *true negatives*, and *false negatives* elements. Secondly, a function (*weight function*) is needed to decide at each moment which descriptor to choose. There is not reason for this function to be the same as the *goodness* function, since their meanings are different; meanwhile the goodness function is defined on the goodness of the expressions themselves (global viewpoint), the other one comes to define the goodness of the chosen path (local viewpoint), in such a way that they only coincide in the limit.

Two points remain unclear: the *selection of the best descriptor* and the *modification of the problem state*, both easily identificable in figure 0.5. The following two tables try to put in clear what those two actions have to do and how.

function `Select-the-Best-Descriptors:`
1. Take the set of descriptors $\mathcal{D}_i = \{d_{i1}, d_{i2}, ...\}$. 2. Modify locally the state s for each descriptor in $\mathcal{D}_i$: $\{s_{i1}, s_{i2}, ...\}$. 3. Apply the function f to each of the new states: $\{f(s_{i1}), f(s_{i2}), ...\}$. 4. Take the set $\mathcal{D}_{i+1} = \{d_{ij} \in \mathcal{D}_i : f(s_{ij}) \geq f(s_{ik})\,;\ k = 1, 2, ...\}$. 5. If $f(s) < f(s_{ij})$ (where $d_{ij} \in \mathcal{D}_{i+1}$) then return $\mathcal{D}_{i+1}$ else return $\emptyset$.

function `Modify-State:`
$C_{i+1} \equiv C_i$ **and** α_{i+1} $[C_{i+1}]^{\pm} = [C_i$ **and** $\alpha_{i+1}]^{\pm} = \{x \in [C_i]^{\pm} : x$ satisfies $\alpha_{i+1}\}$ $\imath C_{i+1} \imath^{\pm} = \imath C_i$ **and** $\alpha_{i+1} \imath^{\pm} = \imath C_i \imath^{\pm} + ([C_{i+1}]^{\mp} - [C_i]^{\mp})$ $\mathcal{D}_{i+1} = \{\alpha \in \mathcal{D}_i : \alpha$ not incompatible with $C_{i+1}\}$

This approach generalizes the case of ID3 [4] where the *weight* function is the local entropy of the features.

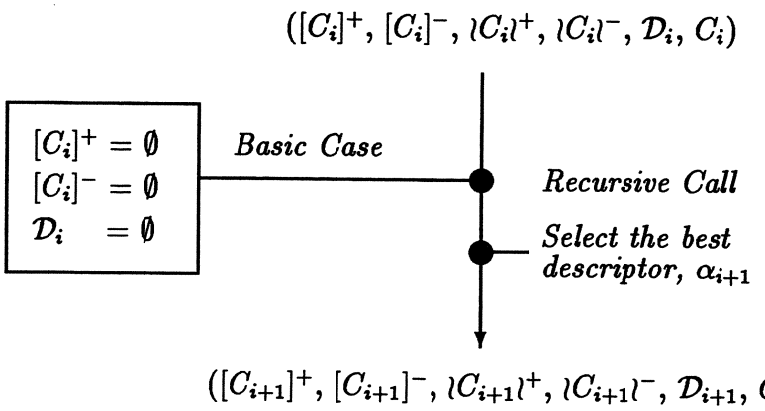


Figure 0.5: *Hill Climbing* diagram for rule generation.

Some considerations of the weight function The *weight function* is used to take a decision over all the p possible alternatives at each point. The maximal criteria, described by formula (3.6), selects the option with the greatest weight value.

$$weight(C_i \textbf{ and } d_k) = \max_{l=1..p}\{weight(C_i \textbf{ and } d_l)\} \tag{3.6}$$

Property 10 (desirable property) *For any* f_w (weight function);

$$f_w(C_i) \geq f_w(C,) \Rightarrow f_w(C_i \textbf{ and } C) \geq f_w(C_j \textbf{ and } C).$$

This will mean that any decision taken to maximize the local goodness of the expression is reflected as a maximization of the global goodness (i.e. the weight function can be assumed to be equal to the goodness function). Unfortunately this strong assumption is not easily achieved when the goodness of the expression is based on the sets depicted in figure 0.1, and the *weight function* has to be an aproximation to the global *goodness function*[4].

Some considerations have to be done concerning the interpretation of formula (3.6).

Case 1: Absolute Interpretation Here every time the weight of a conjunction C_i is asked, it is computed in terms of the sets $[C_i]^+$, $[C_i]^-$, $\wr C_i \wr^+$, and $\wr C_i \wr^-$; recursively described

$$\begin{aligned}
[C_i]^+ &\equiv [C_i \textbf{ and } d_i]^+ = \{x \in [C_{i-1}]^+ : x \text{ satisfies } d_i\} \\
[C_i]^- &\equiv [C_i \textbf{ and } d_i]^- = \{x \in [C_{i-1}]^- : x \text{ satisfies } d_i\} \\
\wr C_i \wr^+ &\equiv \wr C_i \textbf{ and } d_i \wr^+ = \wr C_{i-1} \wr^+ + ([C_i]^- - [C_{i-1}]^-) \\
\wr C_i \wr^- &\equiv \wr C_i \textbf{ and } d_i \wr^- = \wr C_{i-1} \wr^- + ([C_i]^+ - [C_{i-1}]^+)
\end{aligned}$$

where $C_0 = true$ defines the former sets as: $[C_0]^+ = \Theta^+$, $[C_0]^- = \Theta^+$, $\wr C_0 \wr^+ = \emptyset$, and $\wr C_0 \wr^- = \emptyset$.

[4] Using the terminology in [7], the *goodness function* stands for f^*, meanwhile the *weight function* is represented by f.

Case 2: Incremental Interpretation Here the weight of the conjunctions, C_i, is obtained applying the *weight function* not on the global behaviour of C_i (Case 1) but on the increment of the weight when passing from C_{i-1} to C_i. This is, the weight function is computed on the sets

$$\begin{aligned}
[C_i]^+_\Delta &\equiv |[C_{i-1} \textbf{ and } d_i]^+ - [C_{i-1}]^+| \\
[C_i]^-_\Delta &\equiv |[C_{i-1} \textbf{ and } d_i]^- - [C_{i-1}]^-| \\
\wr C_i \wr^+_\Delta &\equiv |\wr C_{i-1} \textbf{ and } d_i \wr^+ - \wr C_{i-1} \wr^+| \\
\wr C_i \wr^-_\Delta &\equiv |\wr C_{i-1} \textbf{ and } d_i \wr^- - \wr C_{i-1} \wr^-|
\end{aligned}$$

For linear *weight functions* both, absolute and incremental interpretations come to the same conclusions.

Case 3: Conditional Interpertation Here again, the weight of the conjunctions C_i is understood in a different form. The idea underlying this interpretation is that decisions have to be taken neither within the the whole initial problem environment (*global interpretation*), nor in the incremental gain of the decision to take (*incremental interpretation*), but in the valuation of the new decision given that the search process is in the actual state. This approach is defined using the conditional sets

$$\begin{aligned}
[C_i]^+_{[C_{i-1}]} &\equiv [C_{i-1} \textbf{ and } d_i]^+ / [C_{i-1}]^+ \\
[C_i]^-_{[C_{i-1}]} &\equiv [C_{i-1} \textbf{ and } d_i]^- / [C_{i-1}]^- \\
\wr C_i \wr^+_{[C_{i-1}]} &\equiv \wr C_{i-1} \textbf{ and } d_i \wr^+ / \wr C_{i-1} \wr^+ \\
\wr C_i \wr^-_{[C_{i-1}]} &\equiv \wr C_{i-1} \textbf{ and } d_i \wr^- / \wr C_{i-1} \wr^-
\end{aligned}$$

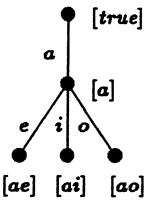
where the initial expression $C_0 = true$ defines the above sets as: $[C_0]^+_{[true]} = \Theta^+$, $[C_0]^-_{[true]} = \Theta^+$, and $\wr C_0 \wr^+_{[true]} = \wr C_0 \wr^-_{[true]} = \emptyset$.

Consider the following example to lightly differentiate the behaviour of the three above interpretations.

> **Example:** Used to make clear that, the global consideration of the weight function (*absolute interpretation*), the increase of benefit (*incremental interpretation*), and the benefit within the actual state (*conditional interpretation*), all of them can produce different expressions when applied.
>
> The *weight* function used in the example is in the form:
>
> $$f_w(C) = \frac{1 + [\![C]\!]^+}{1 + [\![C]\!]^-} + \frac{1 + \wr\!\wr C \wr\!\wr^+}{1 + \wr\!\wr C \wr\!\wr^-}$$
>
> where the increments have been introduced to avoid the case of division by zero.

$\mathcal{D}$

	a	e	i	o
1	1	1	1	1
2	1	1	1	0
Θ^+ 3	1	1	0	1
4	1	0	0	1
5	0	0	0	1
6	0	0	0	0
Θ^- 7	0	1	1	1
8	1	0	1	1



The weight functions are in the form:

- $f_w(C) = \frac{1+[\![\ C\ ]\!]^+}{1+[\![\ C\ ]\!]^-} + \frac{1+\ell\ C\ \ell^+}{1+\ell\ C\ \ell^-}$ (Case 1)
- $f_w^{\Delta}(C) = \frac{1+[\![\ C\ ]\!]^+_{\Delta}}{1+[\![\ C\ ]\!]^-_{\Delta}} + \frac{1+\ell\ C\ \ell^+_{\Delta}}{1+\ell\ C\ \ell^-_{\Delta}}$ (Case 2)
- $f_w^{[C]}(C\ \&\ d) = \frac{1+[\![\ C\ \&\ d\ ]\!]^+_{[C]}}{1+[\![\ C\ \&\ d\ ]\!]^-_{[C]}} + \frac{1+\ell\ C\ \&\ d\ \ell^+_{[C]}}{1+\ell\ C\ \&\ d\ \ell^-_{[C]}}$ (Case 3)

- Case 1: $f_w(ae) \geq f_w(ao) \geq f_w(ai) \Rightarrow ae$ is selected.
- Case 2: $f_w^{\Delta}(ai) \geq f_w^{\Delta}(ao) \geq f_w^{\Delta}(ae) \Rightarrow ai$ is selected.
- Case 3: $f_w^{[a]}(ao) \geq f_w^{[a]}(ai) \geq f_w^{[a]}(ae) \Rightarrow ao$ is selected.

Single Search Genuine Hill-Climbing where *one and only one* descriptor is chosen each time. If several of the available descriptors have the same weight, an alternative criteria is used to only proceed with one of them (e.g. herein the criteria is to proceed with the first one in order of computation; function `(defun one-only (l) (list (first l)))`).

Algorithm 5: Hill Climbing Search (single search).

```
(defun HCSsc (state function)
  (cond ((Basic-Case state) nil)
        (t (single-search
            (Select-the-Best-Descriptors state function)
            state
            function))))

(defun single-search (descriptors s f)
  (if (null descriptors) nil
      (apply #'append
             (mapcar
              #'(lambda (d) (cons d (HCSsc (modify-state s d) f)))
              (one-only descriptors)))))
```

Complete Search This alternative to the single search Hill Climbing is considered to avoid the arbitrarity in the selection of one single descriptor when several of them are equally heavy. Complete Search Hill Climbing extends the Hill Climbing search to all the descriptors whose weight is the greatest. This means that several expressions can be produced by this method,

each with not necessarily the same weight. Is up to the user to decide which one keep and which one drop.

Algorithm 6: Hill Climbing Search (complete search).

```
(defun HCScs (state function)
  (cond ((Basic-Case state) '(nil))
        (t (complete-search
            (Select-the-Best-Descriptors state function)
            state
            function))))

(defun complete-search (descriptors s f)
  (if (null descriptors) nil
      (apply
        #'append
        (mapcar
           #'(lambda (d)
               (mapcar #'(lambda (x) (cons d x))
                       (HCScs (modify-state s d) f)))
           (all descriptors))))
)
```

Example: Comparison of the HCSss and HCScs methods for rules generation. Both are testing with the same sample (s1) of 30 examples randomly generated, and described by ten feasible descriptors ($\mathcal{D} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$). The 70% of the sample are positive examples, and the remain negative examples.

$$\begin{aligned}\Theta^+ &= \{0100100001, 1100011101, 0100110010, 1101000001, 0011100011,\\ &\quad 0001110010, 1101010110, 1011101011, 0011110000, 0010010001,\\ &\quad 1011110100, 0010000000, 1110011111, 1011100011, 0000010010,\\ &\quad 0001111100, 0111001100, 0111010111, 1011000110, 1010110011,\\ &\quad 0001101110\}\\ \Theta^- &= \{0111101101, 1001010000, 0011101110, 1001111011, 1101001111,\\ &\quad 1100010001, 1010001100, 1111101100, 1101110110\}\end{aligned}$$

When the *weight function* is defined $f_w(tp, fp, tn, fn) = tp - 5 \cdot fp$, the search of HCSss (*single search*) comes embeded in the searching tree of the figure 0.6. This search only unfolds the nodes in the upper level and the nodes in the second upper left level. The rule produced is (3 6) which accepts de elements of Θ^+ 0011110000, 0010010001, 1011110100, 1110011111, 0111010111, and 1010110011 (28.57%); and none element of Θ^-. The final goodness value of the rule, reflected in the terminal node of the tree, is 6.

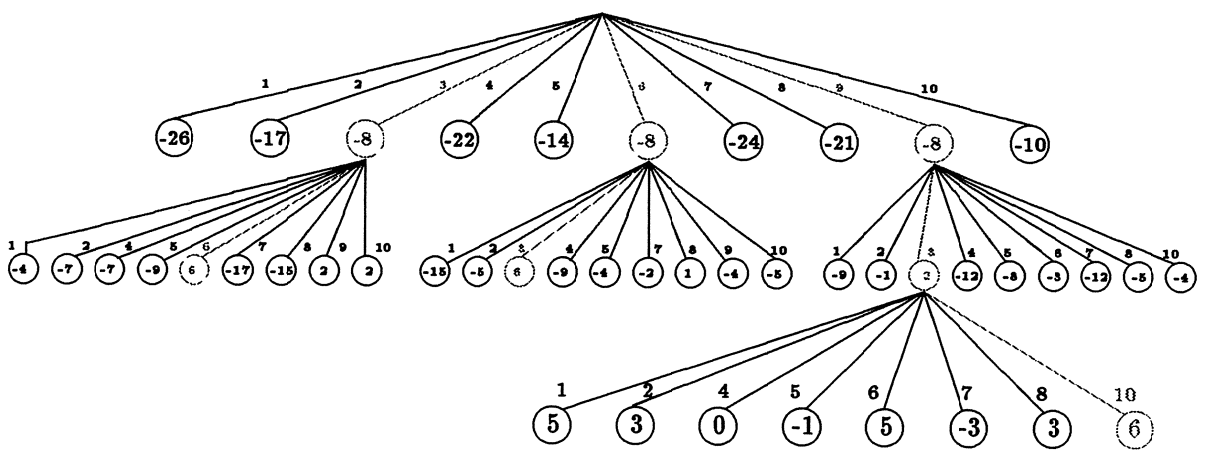


Figure 0.6: Seaching tree for the *Hill-climbing* example.

For the HCScs search, the whole tree in figure 0.6 is unfolded and three rules are obtained (3 6), (6 3), and (9 3 10); all of them with the same goodness, 6.

It can happens, and this is the case, that several permutations of a same rule are given as alternative results (in contadiction to the *first pruning reason*). This *nasty* behaviour is no way fixed if we want to keep the purity of the greedy method, which is unable to predict at the i-th level of the tree what is happening in the incoming levels. Nevertheless a forward *filtering* avoiding permutations is possible.

3.3 General Best First Search

This method does not have any improvement concerning the searching algorithm discused in section 3.1.2. Its incorporation to this work is more a matter of exhaustive contemplation of the most extended searchs than an innovative viewpoint of the problem.

Algorithm 7: Best First Search.

```
(defun BFS (state function)
  (cond ((Basic-Case state) '(nil))
        (t (complete-search
            (Order-all-the-Descriptors state function)
            state
            function))))

(defun complete-search (descriptors s f)
  (if (null descriptors) nil
      (apply
        #'append
        (mapcar
           #'(lambda (d)
```

```
            (mapcar #'(lambda (x) (cons d x))
                    (BFS (modify-state s d) f)))
          (all descriptors))))
)
```

3.4 Approach to the A Algorithm*

This algorithm works with a front of *opened* alternatives, taking at each moment the one more promising (i.e. weight function). The result is an algorithm that, under certain conditions [7], drives the search to one of *the best* possible solutions.

In the next section several heuristics performing the valuation of the alternatives are introduced and compared. Later on, the A* Algorithm is presented.

Heuristics Two big families of heuristics are considered here; (1) those related to the well-stablished goodness function (see section 2), and (2) those related to the length (in number of descriptors) of the rule.

(a) Heuristics on the goodness

Whether the *goodness* of a rule, $C \rightarrow \Theta^+$, is defined as a function of the sets $[C]^+$, $[C]^-$, $\wr C \wr^+$, and $\wr C \wr^-$. Here we are asked to define heuristic functions to measure the badness of a rule or, what is the same, how much extra goodness deserves one rule to become the best rule. And this must be expressed in terms of the goodness function.

The goodness of a future rule is computed as the addition of two components; i.e. the goodness already achieved (g_i), and the goodness that will be achieved in the path to the solution (h_i). Meanwhile g_i is the well known goodness function itself, the function h_i is unknown and must be approximated by mean of *heuristic functions*. The next list show some of the feasible heuristics.

1. Consider the *best rule*, the one which accepts all the positive examples, Θ^+, and rejects all the negative examples, Θ^-. The heuristic for the function h is defined then as the portion of goodness remaining to achieve the *best rule*.

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ f_i(|\Theta^+|, 0, |\Theta^-|, 0) - f_i([\![C]\!]^+, [\![C]\!]^-, \wr C \wr^+, \wr C \wr^-) & \text{otherwise} \end{cases} \tag{3.7}$$

2. The local interpretation of the above heuristic does not consider the utopy of achieving the *best rule*, but the relaxed utopy of ferreting out the best rule reachable from the state we are.

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ f_i([\![C]\!]^+, 0, [\![C]\!]^-, 0) & \text{otherwise} \end{cases} \tag{3.8}$$

3. The nearby function considers the case where the incoming increment of goodness is expressed in terms of the positive examples not yet considered, and the negative examples not yet misconsidered. This heuristic is transformed, in the case of linear goodness functions, to $f_i(\Theta^+, 0, \Theta^-, 0) + f_i([\![C]\!]^+, [\![C]\!]^-, \wr C \wr^+, \wr C \wr^-) - 2 \cdot f_i([\![G]\!]^+, 0, \wr C \wr^+, 0)$.

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ f_i(|\Theta^+| - [\![\ C\ ]\!]^+, [\![\ C\ ]\!]^-, |\Theta^-| - \wr\ C\ \wr^+, \wr\ C\ \wr^-) & \text{otherwise} \end{cases} \quad (3.9)$$

4. The *general case* for the heuristic function h is based in the selection of the two values $k_1 \in [0, [\![\ C\ ]\!]^+]$ and $k_2 \in [0, [\![\ C\ ]\!]^-]$, which define the function h_i in the form;

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ f_i([\![\ C\ ]\!]^+ - \mathrm{k}_1, [\![\ C\ ]\!]^- - \mathrm{k}_2, \wr\ C\ \wr^+ + \mathrm{k}_2, \wr\ C\ \wr^- + \mathrm{k}_1) & \text{otherwise} \end{cases} \quad (3.10)$$

There are two particular values for k_1 and k_2, which depend directly on the remain descriptors, and for which the function h_i is the cheapest cost function h^*.

(b) Heuristics on the length

The heuristics here are think to calculate the number of descriptors that have to be *and*ed to the present conjunction C to reach a goal situation. To define the family of heuristics under this scheme some notations are introduced. Let $|C|$ represent the length of the conjunctive expression C, and defined as the number of descriptors used in C. Some of the heuristics use the functions h_i^* defined in the last paragraphs, which are renamed here to $\dot{h}_i^*$.

1. Whether arriving to the actual situation takes $|C|$ descriptors, going on to a desirable goal situation will take $Int(h_i^*)$ new descriptors. $Int(x)$ is the function which takes the integer part of x.

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ \frac{\dot{h}_i^*(C)}{f_i(C)} \cdot |C| & \text{otherwise} \end{cases} \quad (3.11)$$

2. Whether the spending of descriptors has followed the frequency $\frac{|C|}{|\mathcal{D}|}$. An approximation to the number of new descriptors, before the set of descriptors is empty, is the value.

$$h_i^*(C) = \begin{cases} 0 & \text{in the } \textit{Basic Case} \\ \frac{|C|}{|\mathcal{D}|} \cdot |Compatible(C)| & \text{otherwise} \end{cases} \quad (3.12)$$

The set $Compatible(C)$ is $\mathcal{D}$ when C is the basic conjunction *true*, and the intersection of compatibilities for all the descriptors in C when this is a complex conjunction (see the function `Modify-State` in page 13 for more accurate details).

Other many functions could be considered at this point in both, *heuristics on the goodness* and *heuristics on the length*. The purpose here is not to give an exhaustive list, but remark all those that are more representative. The A* algorithm in the next section is somehow conscient of this limitation, and it is intentionally designed to allow any new definition of the h^* function.

The A Algorithm* The algorithm in this section follows the guidelines in [7]. Some other implementations are possible, being some of them even more efficient than the one here.

Algorithm 8: A*.

```
(defun A* (s g h)
  (let ((OPEN (list s))(CLOSED nil)(n nil))
    (loop
      (if (null OPEN) (return nil))
      (setq n (minimal OPEN #'(lambda (x) (get x 'f-value))))
      (setq OPEN   (remove n OPEN :test #'equalp))
      (setq CLOSED (place n CLOSED))
      (if (goal n) (return (reconstruct-path n)))
      (dolist (s (expand n))
         (setf (get s 'g-value) (apply-f g s))
         (setf (get s 'h-value) (apply-f h s))
         (setf (get s 'f-value) (+ (get s 'g-value) (get s 'h-value)))
         (let ((OPEN-s   (extract s OPEN))
               (CLOSED-s (extract s CLOSED)))
           (setq OPEN   (remove OPEN-s   OPEN   :test #'equalp))
           (setq CLOSED (remove CLOSED-s CLOSED :test #'equalp))
           (cond ((and (null OPEN-s) (null CLOSED-s))
                  (setq OPEN (place s OPEN)))
                 ((null CLOSED-s)
                  (if (< (get OPEN-s 'g-value) (get s 'g-value))
                      (free s)
                      (progn (setf (get s 'descriptors)
                                   (union (get OPEN-s 'descriptors)
                                          (get s 'descriptors)))
                             (free OPEN-s) (setq OPEN (place s OPEN)))))
                 ((null OPEN-s)
                  (if (< (get CLOSED-s 'g-value) (get s 'g-value))
                      (free s)
                      (progn (setf (get s 'descriptors)
                                   (union (get CLOSED-s 'descriptors)
                                          (get s 'descriptors)))
                             (free CLOSED-s) (set OPEN (place s OPEN)))))
))))))
```

Example: The use of one or other heuristic can drive the algorithm to produce different results. The sequence of actions,

```
> (setq s1 (make-sample 50 100 .7))
> (setq e1 (make-initial-state s1 D))
> (setq n1 (make-initial-node e1 g h))
```

define one problem with 50 examples (70% positive examples) and 100 descriptors. Constants a_1, a_2, b_1, and b_2 are initialized with

```
> (setq a1 3 a2 5 b1 1 b2 7)
```

and the heuristics (3.7), (3.8), (3.9), and (3.10) are tested obtaining the following results

$h_i^*(C)$	*Expression*	*f value*	No. Θ^+	No. Θ^-
(3.7)	(99 100)	210	10	0
(3.8)	(44 78 80 95)	40	0	6
(3.9)	(44 78 95)	62	1	8
(3.10)	(10 44 78 80 95)	377/2	0	6
	(44 80 95)	21/2	0	7
The case (3.10) *is analized under two situations* 1. When $k_1 = tp/2$ and $k_2 = fp/2$, and 2. when $k_1 = tp/10$ and $k_2 = fp/10$.				

The last three columns represent the *goodness* of the result, the *number of positive examples* accepted and the *number of negative examples* accepted.

4. Analysis

Two comparative analysis of the methods in section 3 were realized and discussed here. The first analysis concerns the results obtained and the second the comparison of time costs.

For the first case a random problem is generated (`#:|SAMPLE:87141|`) consisting of the 40 examples and 10 descriptors. The next table describes the examples in terms of the descriptors for the 50% of positive examples Θ^+ (named p_i in the left-half table), and for the remaining negative examples Θ^- (named n_i in the right-half table). One example, p_i (or n_i), for a given descriptor, j, may have two values: 0 if j does not happen in p_i (or n_i), and 1 in the contrary.

Θ^+	1	2	3	4	5	6	7	8	9	10
p_1	1	0	1	1	1	0	0	0	1	0
p_2	0	1	0	1	1	1	1	1	0	0
p_3	0	0	1	0	1	1	0	0	0	1
p_4	0	1	0	0	1	0	1	0	0	1
p_5	0	0	0	1	0	0	0	1	0	0
p_6	1	1	1	0	1	1	0	0	1	0
p_7	0	1	1	0	0	0	0	1	0	0
p_8	1	0	1	0	1	0	0	0	0	0
p_9	1	0	1	1	1	0	1	1	0	0
p_{10}	1	1	0	0	1	1	1	0	0	1
p_{11}	1	1	1	0	0	0	0	1	0	1
p_{12}	0	1	1	0	0	1	0	1	1	1
p_{13}	1	1	0	1	0	1	0	0	0	1
p_{14}	0	0	1	0	0	1	0	0	0	1
p_{15}	1	1	0	1	0	0	0	0	1	1
p_{16}	1	1	0	1	1	1	0	1	0	1
p_{17}	0	1	1	0	0	0	1	1	1	0
p_{18}	1	0	0	0	1	0	1	0	0	0
p_{19}	0	0	0	1	0	1	0	0	1	0
p_{20}	1	0	1	1	1	0	0	1	1	0

Θ^-	1	2	3	4	5	6	7	8	9	10
n_1	0	0	0	0	1	1	1	0	1	1
n_2	0	1	1	1	1	0	1	0	0	0
n_3	1	0	0	0	0	0	0	1	0	1
n_4	0	1	0	0	0	0	0	1	0	0
n_5	0	0	0	1	0	1	1	1	1	0
n_6	0	1	1	0	0	1	1	0	1	1
n_7	1	0	0	1	1	0	1	1	1	0
n_8	1	1	0	1	0	1	1	0	0	0
n_9	1	0	0	0	0	1	1	1	0	1
n_{10}	1	0	1	1	1	0	1	1	1	1
n_{11}	0	1	1	1	0	0	0	0	0	0
n_{12}	0	1	0	1	0	1	0	1	0	1
n_{13}	1	1	0	1	0	1	0	1	0	0
n_{14}	0	1	1	1	0	0	1	0	0	0
n_{15}	0	0	1	1	0	1	1	1	1	0
n_{16}	0	0	0	1	0	1	1	0	0	1
n_{17}	1	1	1	1	1	0	0	1	1	0
n_{18}	1	0	1	0	0	1	0	0	0	1
n_{19}	0	1	1	0	1	0	0	0	1	0
n_{20}	0	0	1	0	0	0	0	0	0	0

Four algorithms have been considered; EXHAUSTIVE, HILL-CLIMBING-*, BEST-FIRST-*, and A*. For the ones in the middle (postfixed '-*') the version of multiple solutions is runned. The '-*' version for the EXHAUSTIVE method does not represent a significant increase of the *time* and the conjunctions obtained are the ones in the BEST-FIRST-* method avoiding permutations.

method's name	*best Conjunction*	*time*	*goodness*
EXHAUSTIVE	(2 3 4)	29350	135
HILL-CLIMBING-*	(7 9 4) (7 9 6)	550	135
BEST-FIRST-*	(2 3 4) (2 4 3) (3 2 4) (3 4 2) (4 2 3) (4 3 2) (4 7 8 9) (4 7 9) (4 8 7 9) (4 8 9 7) (4 9 7) (4 9 8 7) (6 7 9) (6 9 7) (7 4 8 9) (7 4 9) (7 6 9) (7 8 4 9) (7 8 9 4) (7 9 4) (7 9 6) (7 9 8 4) (8 4 7 9) (8 4 9 7) (8 7 4 9) (8 7 9 4) (8 9 4 7) (8 9 7 4) (9 4 7) (9 4 8 7) (9 6 7) (9 7 4) (9 7 6) (9 7 8 4) (9 8 4 7) (9 8 7 4)	1161610	135
A*	(9 7 6)	760	135

When the algorithms are applied the results on the previous table are obtained. The most important conclusions verify the intuitive sense; the fastest method is HILL-CLIMBING even in the case of finding all the 'possible' solutions. The slowest is the *best-first* method where all the possible alternatives are analyzed (like in *exhaustive*) but with the increased cost of deciding, at each point, which is the next best descriptor to choose. A* is competitive enough to be compared with the faster method, but with the security of arriving to the optimal solution whether an *admissible* heuristic is applied.

The *goodness* of all the results is optimal for all the methods, which means that, for some particular problems, with all the methods the 'best' solution is reachable.

A study of the *robustness* of the methods was considered for the case of *very well informed* domains (domains where many positive examples are known) and for the case of *very bad informed* domains or domains described in terms of exceptions. The test was realized separately on samples where the percentage of positive examples were from 0% to 100%, with increases of 10%. For each percentage 20 different randomly-generated samples were taken and solved using the available algorithms. For each algorithm and percentage tested, the time was averaged for the 20 samples and the results tabulated.

The averaged time for the EXHAUSTIVE method confirmed the initial assumption of being a method independent of the percentage of positive and negative examples. The results were respectively the times 25580, 27433, 27431, 27463, 27689, 28330, 27738, 27694, 27824, 27600, and 25748 for the percentages considered.

More interesting is the comparison of the HILL-CLIMBING-* and A-* methods, represented in the figure 0.7. The curves are introduced to remark the *concave* conduct of HILL-CLIMBING-* and the *convex* conduct of the A-* algorithm. In other words, time in HILL-CLIMBING-* degenerates for domains where the percentage of examples is equivalent to the percentage of counterexamples, but it works quite fast when the number elements in one of the two sets is clearly greater. Just in the contrary of the A-* method.

5. FURTHER ISSUES

The algorithms in this work have been designed to realize learning by specification. Analogous methodologies could be applied to perform *learning by generalization* where the initial rule should be the conjunction of all the possible descriptors ($\mathcal{D}$), and the goodness function

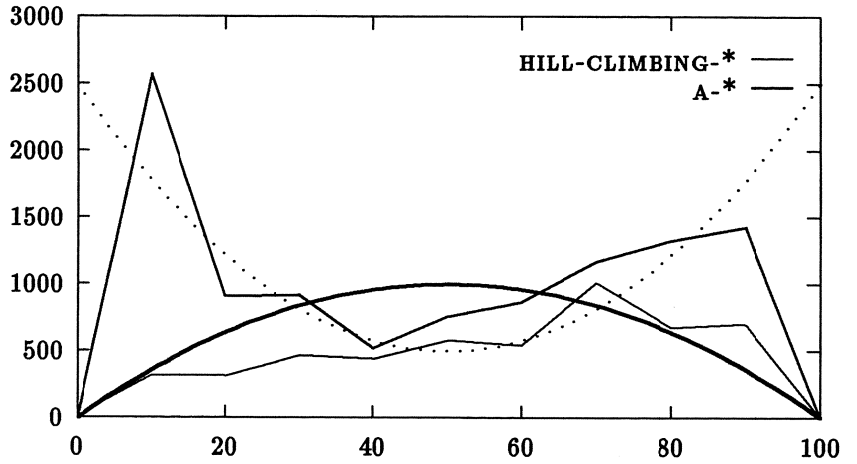


Figure 0.7: Comparative performance of *Hill Climbing* and A^*.

should be redefined to indicate which descriptors have to be removed for the sake of the rule improvement.

A second limitation of this work is that it has been centered exclusively in the production of conjunctive expressions. All the pruning and heuristics has been introduced keeping in mind this fact, and their extensibility to alternative expressions is called into question. In the other side, the algorithms themselves are hardly modifiable to produce alternative expressions (for this purpose see [8]).

It has not been considered the learning process in either a noise domain where the restriction $\Theta^+ \cap \Theta^- = \emptyset$ is slacken, or the case of unknown or missing values (this is, partially defined examples), or the case of imprecise or uncertain knowledge.

All the tests in this work have been performed in `CommonLisp` (CLISP), version `1994-10-26 (October 1994)` on the `SUN4C` workstation `wolf.cwi.nl [192.16.191.98]` with the soft `ANSI C` program, version `GNU C 2.6.0`. The unities of time are implementation-dependent and must be used only comparatively.

6. Acknowledgments

I want to express my deep gratitude to Peter Grünwald, always interested in my work and whose corrections and comments conducted this work to the way it is. I am also grateful to Prof. Doct. Paul Vitanyi, leader of the research group wherein this work was developed and the person who accepted me at the CWI. Finally, I thank Prof. Doct. Ulises Cortés, at the UPC, and Doct. Vicenç Torra, at the URV, all their efforts to arrange my bureaucratic problems.

REFERENCES

1. Michalski R.S., *AQVAL/1-Computer Implementation of a Variable-Valued Logic System VL1 and Examples of its Application to Pattern Recognition*, Proceedings of the First Int. Joint Conference on Pattern Recognition (1973), Washington DC., 3-17.

2. Williams B.T., *Computer Aids to Clinical Decisions*, (Vol. I & II), Boca Raton, FL: CRC Press (1982).

3. Weiss S.M., Galen R.S., Tadepalli P.V., *Optimizing the Predictive Value of Diagnostic Decision Rules*, Proceedings of the Sixth National Conf. on Artificial Intelligence, AAAI (1987), Seattle, 521-526.

4. Quinlan J., *Induction of Decision Trees*, Machine Learning 1 (1986), 81-106.

5. Cendrowska J., *PRISM: An Algorithm for Inducing Modular Rules*, Int. Journal on Man-Machine Studies 27 (1987).

6. Kononenko I., Bratko I., *Information-Based Evaluation Criterion for Classifier's Performance*, Machine Learning 6 (1991), 67-80.

7. Pearl J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Publishing Company, Ch. 2-3, pp. 33-112, 1984.

8. Riaño D., *Automatic Knowledge Generation from Data in Classification Domains*, Master Thesis, Facultat d'Informàtica de Barcelona (UPC), Barcelona (1994).

1. NESsc AND NESi CODES

Complete self-contained code for the searching strategies in section 3 without *first pruning reason search*; its use is examplified in Appendix 5.

```
;.....................................................

(defun NESsc (D)
  (cond ((null D) '(nil))
        (t (cons
             nil
             (apply #'append
                    (mapcar
                      #'(lambda (n)
                          (mapcar
                            #'(lambda (y) (cons (nth (- n 1) D) y))
                            (NESsc (but-nth n D))))
                      (makelist (length D))))))))

;.....................................................

(defun NESi (conj S)
  (let* ((last-S (remove-if #'(lambda(x)(find x conj :test #'equalp)) S))
         (last-1 (butlast conj)))
    (cond ((null (car conj)) (list (car S)))
          ((null last-S)
             (if (equalp (reverse conj) S) conj
                 (do* ((last-1 (last conj) (last prem))
                       (last-2 (last (butlast conj)) (last (butlast prem)))
                       (prem (butlast conj) (butlast prem)))
                      ((> (position (car last-1) S)
                          (position (car last-2) S))
                       (append (butlast prem)
                               (list (nth 1 (member (car last-2)
(remove-if #'(lambda(x) (find x (butlast prem) :test #'equalp)) S)))))))))
          (t (append conj (list (car last-S)))))))

;.....................................................

(defun makelist (n)
  (cond ((= n 0) nil)
        (t (append (makelist (- n 1)) (list (- n 1))))))

(defun but-nth (n l)
  (cond ((= n 1) (cdr l))
        (t (cons (car l) (but-nth (- n 1) (cdr l))))))
```

2. FPRsc AND FPRi CODES

Complete self-contained code for the searching strategies in section 3 applying the *first pruning reason search*; its use is examplified in Appendix 5.

```
;.....................................................

(defun FPRsc (D)
  (cond ((null D) '(nil))
        (t (cons
             nil
             (apply #'append
                    (mapcar #'(lambda (n)
                                (mapcar #'(lambda (y) (cons (nth (- n 1) D) y))
                                        (NESsc (from-nth n D))))
                            (makelist (length D))))))))

;.....................................................

(defun FPRi (conj S)
  (let* ((last-s (cdr (member (car (last conj)) s)))
         (last-1 (butlast conj)))
    (cond ((null last-s)
           (if (null last-1) conj
               (append (butlast last-1)
                       (list (nth 1 (member (car (last last-1)) s))))))
          (t (append conj (list (car last-s)))))))

;.....................................................

(defun from-nth (n l)
  (cond ((= n 1) (cdr l))
        (t (from-nth (- n 1) (cdr l)))))
```

3. HCcs AND HCsc CODES

Complete self-contained code for the *Hill Climbing* searching strategies in section 4; its use is exemplified in Appendix 5.

```
;.....................................................

(defun HCSsc (state function)
  (cond ((Basic-Case state) nil)
(t (single-search
    (Select-the-Best-Descriptors state function)
    state
    function))))

(defun single-search (descriptors s f)
  (if (null descriptors) nil
      (apply #'append
     (mapcar
      #'(lambda (d) (cons d (HCSsc (modify-state s d) f)))
      (one-only descriptors)))))

;.....................................................

(defun HCSsc (state function)
  (cond ((Basic-Case state) '(nil))
(t (complete-search
    (Select-the-Best-Descriptors state function)
    state
    function))))

(defun complete-search (descriptors s f)
  (if (null descriptors) nil
      (apply #'append
     (mapcar
      #'(lambda (d) (mapcar #'(lambda (x) (cons d x)) (HCSsc (modify-state s d) f)))
      (all descriptors))))
)

;.....................................................

(defun Basic-Case (s)
  (let ((Di (get s 'descriptors)) (tp (get s 'tp)) (fp (get s 'fp)))
    (or (null Di) (null tp) (null fp))))

(defun Select-the-Best-Descriptors (st f)
  (labels ((apply-f (function state)
             (funcall function
                      (length (get state 'tp)) (length (get state 'fp))
                      (length (get state 'tn)) (length (get state 'fn))))
   (select-the-maximum (descriptors valuations)
             (do* ((ld descriptors       (cdr ld)) ;List of Descriptors
   (lv valuations        (cdr lv)) ;List of Valuations
   (fd (car descriptors) (car ld)) ;First Descriptor
   (fv (car valuations)  (car lv)) ;First Valuation
```

```
   (lb (list fd)          (cond ((> fv bv) (list fd))
                                        ((= fv bv) (cons fd lb))
                                        ((< fv bv) lb)))
                                    ;List of the Best descriptors
   (bv fv                 (max bv fv));Best Valuation
  )
  ((null (cdr ld)) (reverse lb))))
  )
     (let* ((step1 (get st 'descriptors))
    (step2 (mapcar #'(lambda (d) (modify-state st d)) step1))
    (step3 (mapcar #'(lambda (s) (apply-f f s)) step2))
    (step4 (select-the-maximum step1 step3))
   )
       (if (< (apply-f f st)
      (apply-f f (modify-state st (car step4)))) step4 nil))))

(defun modify-state (st descriptor)
  (let* ((tp (get st 'tp)) (fp (get st 'fp))
 (tn (get st 'tn)) (fn (get st 'fn))
 (descriptors (get st 'descriptors))
 (f+ (FILTER-and tp descriptor))
 (f- (FILTER-and fp descriptor))
 (s  (gensym "STATE:")))
    (setf (get s 'tp) f+)
    (setf (get s 'fp) f-)
    (setf (get s 'tn) (union tn (set-difference fp f- :test #'equal)))
    (setf (get s 'fn) (union fn (set-difference tp f+ :test #'equal)))
    (setf (get s 'descriptors) (remove descriptor descriptors))
    (setf (get s 'ancestor) st)
    s))

(defun one-only (l) (list (first l)))
(defun all (l) l)
```

4. A* Code

```
;.....................................................
(defun A* (s g h)
  (let ((OPEN (list s))(CLOSED nil)(n nil))
    (loop
      (if (null OPEN) (return nil))
(format nil "OPEN: ~s CLOSED: ~s" OPEN CLOSED)
      (setq n (minimal OPEN #'(lambda (x) (+ (funcall g x) (funcall h x)))))
      (setq OPEN   (remove n OPEN :test #'equalp))
      (setq CLOSED (place n CLOSED))
      (if (goal n) (return (reconstruct-path n)))
      (dolist (s (expand n))
 (setf (get 'g-value s) (funcall g s))
 (let ((OPEN-s   (extract s OPEN))
       (CLOSED-s (extract s CLOSED)))
   (cond ((and (null OPEN-s) (null CLOSED-s))
                (setf (get 'h-value s) (funcall h s))
                (setf (get 'f-value s) (+ (get 'g-value s)
                                          (funcall c  n s)
                                          (get 'h-value s))))
               ((null CLOSED-s)
                (if (< (get 'g-value OPEN-s) (get 'g-value s))
                    (free s)
                    (progn (free OPEN-s) (set OPEN (place s OPEN)))))
               ((null OPEN-s)
                (if (< (get 'g-value CLOSED-s) (get 'g-value s))
                    (free s)
                    (progn (free CLOSED-s) (set OPEN (place s OPEN)))))
))))))

(defun minimal (S f)
  (car (remove-if #'(lambda (x) (> (funcall f x) (apply #'min (mapcar f S)))) S)))

(defun place (n S) (cons n S))

(defun goal (n)
  (or (null (get 'descriptors n))
      (null (get 'tp n))
      (null (get 'fp n))))

(defun reconstruct-path (n)
  (labels ((recons-path-rec (node)
             (cond ((null (get node 'antecedent)) nil)
                   (t (cons (get node 'descriptor)
                      (reconstruct-path (get node 'antecedent)))))))
    (reverse (recons-path-rec n))
))

(defun expand (n)
  (dolist (d (get 'descriptors n))
    (let* ((new-tp (FILTER-and (get 'tp n) d))
           (new-fp (FILTER-and (get 'fp n) d))
           (new-tn (union (get 'tn n) (set-difference (get 'fp n) new-fp)))
```

```
          (new-fn (union (get 'fn n) (set-difference (get 'tp n) new-tp)))
          (s (make-node new-tp new-fp new-tn new-fn d

(defun extract (n S) )

(defun make-initial-node (sample D g h)
  (let ((n (gensym "NODE:")))
    (setf (get n 'tp) (O+ sample))
    (setf (get n 'fp) (O- sample))
    (setf (get n 'tn) nil)
    (setf (get n 'fn) nil)
    (setf (get n 'descriptor) nil)
    (setf (get n 'g-value) (apply-f g n))
    (setf (get n 'h-value) (apply-f h n))
    (setf (get n 'f-value) (+ (get n 'g-value) (get n 'h-value)))
    (setf (get n 'descriptors) D)
    (setf (get n 'antecedent) nil)
    (setf (get n 'consequent) nil)
    n))

(defun make-successor-node (n d g h)
  (let* ((s (gensym "NODE:")))
     (setf (get s 'tp) (FILTER-and (get 'tp n) d))
     (setf (get s 'fp) (FILTER-and (get 'fp n) d))
     (setf (get s 'tn) (union (get 'tn n) (set-difference (get 'fp n) new-fp)))
     (setf (get s 'fn) (union (get 'fn n) (set-difference (get 'tp n) new-tp)))
     (setf (get s 'descriptor) d)
     (setf (get s 'g-value) (apply-f g n))
     (setf (get s 'h-value) (apply-f h n))
     (setf (get s 'f-value) (+ (get s 'g-value) (get s 'h-value)))
     (setf (get s 'descriptors) (set-difference (get 'descriptors n) d :test #'equalp))
     (setf (get s 'antecedent) n)
     (setf (get s 'consequent) nil)

     (setf (get n 'consequent) (cons s (get n 'consequent)))
     s))
```

5. TESTING

```
wolf.cwi.nl% lisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I I I I I I I      8         8           8     8        8    8
  I I I I I I I      8         8           8      ooooo   8oooo
  I  \ `+' /  I      8         8           8           8  8
   \  `-+-'  /       8     o   8           8     o     8  8
    `-__|__-'         ooooo    8ooooooo  ooo8ooo   ooooo   8
        |
  ------+------      Copyright (c) Bruno Haible, Michael Stoll 1992, 1993, 1994


> (load "general.lisp")
;; Loading file general.lisp ...
;; Loading of file general.lisp is finished.
T
> (setq f #'(lambda (tp fp tn fn) (- (- (* 7 tp) (* 3 fp)) (- (* 7 fn) (* 3 tn)))))
#<CLOSURE :LAMBDA (TP FP TN FN) (- (- (* 7 TP) (* 3 FP)) (- (* 7 FN) (* 3 TN)))>
> (setq g f)
#<CLOSURE :LAMBDA (TP FP TN FN) (- (- (* 7 TP) (* 3 FP)) (- (* 7 FN) (* 3 TN)))>
> (setq h #'(lambda (tp fp tn fn) (- (funcall f 0+ 0 0- 0) (funcall f tp fp tn fn))))
#<CLOSURE :LAMBDA (TP FP TN FN) (- (FUNCALL F 0+ 0 0- 0) (FUNCALL F TP FP TN FN))>
> (setq s (make-sample 7 4 .6))
#:|SAMPLE:725|
> (setq i (make-initial-state s (D s)))
#:|STATE:726|
> (setq n (make-initial-node i g h))
#:|NODE:727|
> (time (NESsc (D s)))

Real time: 0.192986 sec.
Run time: 0.17 sec.
Space: 23568 Bytes
(NIL (1) (1 2) (1 2 3) (1 2 3 4) (1 2 4) (1 2 4 3) (1 3) (1 3 2) (1 3 2 4) (1 3 4)
 (1 3 4 2) (1 4) (1 4 2) (1 4 2 3) (1 4 3) (1 4 3 2) (2) (2 1) (2 1 3) (2 1 3 4)
 (2 1 4) (2 1 4 3) (2 3) (2 3 1) (2 3 1 4) (2 3 4) (2 3 4 1) (2 4) (2 4 1) (2 4 1 3)
 (2 4 3) (2 4 3 1) (3) (3 1) (3 1 2) (3 1 2 4) (3 1 4) (3 1 4 2) (3 2) (3 2 1)
 (3 2 1 4) (3 2 4) (3 2 4 1) (3 4) (3 4 1) (3 4 1 2) (3 4 2) (3 4 2 1) (4) (4 1)
 (4 1 2) (4 1 2 3) (4 1 3) (4 1 3 2) (4 2) (4 2 1) (4 2 1 3) (4 2 3) (4 2 3 1)
 (4 3) (4 3 1) (4 3 1 2) (4 3 2) (4 3 2 1))
> (print "NESsc")

"NESsc"
"NESsc"
> (all-NESi (D s))
(NIL (1) (1 2) (1 2 3) (1 2 3 4) (1 2 4) (1 2 4 3) (1 3) (1 3 2) (1 3 2 4) (1 3 4)
 (1 3 4 2) (1 4) (1 4 2) (1 4 2 3) (1 4 3) (1 4 3 2) (2) (2 1) (2 1 3) (2 1 3 4)
 (2 1 4) (2 1 4 3) (2 3) (2 3 1) (2 3 1 4) (2 3 4) (2 3 4 1) (2 4) (2 4 1)
 (2 4 1 3) (2 4 3) (2 4 3 1) (3) (3 1) (3 1 2) (3 1 2 4) (3 1 4) (3 1 4 2) (3 2)
 (3 2 1) (3 2 1 4) (3 2 4) (3 2 4 1) (3 4) (3 4 1) (3 4 1 2) (3 4 2) (3 4 2 1)
 (4) (4 1) (4 1 2) (4 1 2 3) (4 1 3) (4 1 3 2) (4 2) (4 2 1) (4 2 1 3) (4 2 3)
 (4 2 3 1) (4 3) (4 3 1) (4 3 1 2) (4 3 2) (4 3 2 1))
> (NESi nil (D s))
(1)
> (NESi '(1) (D s))
(1 2)
> (NESi '(1 2 3 4) (D s))
(1 2 4)
> (NESi '(4) (D s))
(4 1)
> (time (FPRsc (D s)))

Real time: 0.094053 sec.
Run time: 0.06 sec.
Space: 8632 Bytes
```

```
(NIL (1) (1 2) (1 2 3) (1 2 3 4) (1 2 4) (1 2 4 3) (1 3) (1 3 2) (1 3 2 4) (1 3 4)
 (1 3 4 2) (1 4) (1 4 2) (1 4 2 3) (1 4 3) (1 4 3 2) (2) (2 3) (2 3 4) (2 4)
 (2 4 3) (3) (3 4) (4))
> (all-FPRi (D s))
(NIL (1) (1 2) (1 2 3) (1 2 3 4) (1 2 4) (1 3) (1 3 4) (1 4) (2) (2 3) (2 3 4)
 (2 4) (3) (3 4) (4))
> (FPRi nil (D s))
(1)
> (FPRi '(1) (D s))
(1 2)
> (FPRi '(1 2 3 4) (D s))
(1 2 4)
> (FPRi '(4) (D s))
(4)
> (time (HCSsc i h))

Real time: 0.151189 sec.
Run time: 0.12 sec.
Space: 10760 Bytes
(4 1)
> (time (HCScs i h))

Real time: 0.14166 sec.
Run time: 0.11 sec.
Space: 12056 Bytes
((4 1) (4 3))
> (time (A* n g h))

Real time: 0.186746 sec.
Run time: 0.16 sec.
Space: 16432 Bytes
(4 3)
```