Altering the Description Space for Focussing

ALTERING THE DESCRIPTION SPACE FOR FOCUSSING

Jan Wielemaker *
Alan Bundy

D.A.I. RESEARCH PAPER NO. 262

Jan Wielemaker
University of Amsterdam
Institute for Cognitive Studies
Weesperplein 8
1018 XA Amsterdam
Netherland

Paper submitted to *Expert Systems-85*,
Warwick, December 1985.

ALTERING THE DESCRIPTION SPACE FOR FOCUSSING

Jan Wielemaker
Alan Bundy

University of Edinburgh
Department of Artificial Intelligence
Hope Park square
Edinburgh EH9 NW
Scotland

ABSTRACT

This paper discusses improvements on the heuristic rule and concept learning technique, called focussing. Central to this technique is the description space. It is a set of trees, representing knowledge about the domain in which the concepts (or rules) to be learned are described (see figure 2-1 for an example). The heuristic information is kept in the hierarchy of these trees. Unfortunately focussing can't learn a concept or rule if this hierarchy is wrong. A technique called "tree-hacking" is introduced to repair this flaw.

Also discussed is a way to build description spaces for focussing, given the properties that should be concerned and their possible values. The hierarchy is constructed by looking at a (large) set of concepts that share (parts of) their description space.

Acknowledgements

Jan Wielemaker would like the thank Jim Howe, Alan Bundy and Bob Wielinga for their work done to let me visit the AI department in Edinburgh. I also would like to thank Alan Bundy and Bernard Silver for their advise, and all the others who made this visit interesting.

keywords: concept learning, rule learning, focussing, tree-hacking, description space

1. INTRODUCTION

Focussing is a heuristic technique to learn concepts or rules from examples and counter examples (called "specimen"). Central to this technique is the description space. It is a set of trees, representing knowledge about the domain in which the concepts (or rules) to be learned are described. Each tree describes a certain aspect and its possible "values". For example a tree might describe "tense", with values "past", "present" and "future" (see figure 3-1). The structure of the tree represents heuristics for the learning process. Markers on the trees represent the state of the learning process. Focussing is discussed in more detail in section 2.

Focussing does not need to store all the specimens presented in the past. Together with its heuristic nature these are the main advantages of the technique. Unfortunately focussing also has a some serious flaws. One of them is that the hierarchy of the trees in the description space should be suitable for learning the concept you want it to learn. If this is not true focussing will eventually find a contradiction in the learning process.

[Bundy '82] and [Bundy e.a. '84] describe a technique, called "tree hacking" to change the hierarchy of the trees when focussing faces a contradiction. Section 3 describes refinements on this technique and its implementation. Section 4 introduces another technique, not only to enable focussing to cope with wrongly structured trees, but also to build trees by examining a set of concepts. Suggestions for further research are presented.

2. FOCUSSING

Remark

Learning concepts or rules from examples and counter examples is exactly the same process. In the rest of this report only learning of concepts is mentioned.

2.1. A description of focussing

Focussing is a heuristic technique to learn concepts from specimen. It uses a "description space" to represent its rule and heuristics. This description space consists of trees representing the important aspects to decide whether a specimen is an example or a counter-example. For example the trees used to learn the concept of an arch are: "shape" tree, "support" tree, "touch-relation" tree and "orientation" tree. The state of the learning program is represented by two markers on each tree: the "upper marker" and the "lower marker". See figure 2-1.

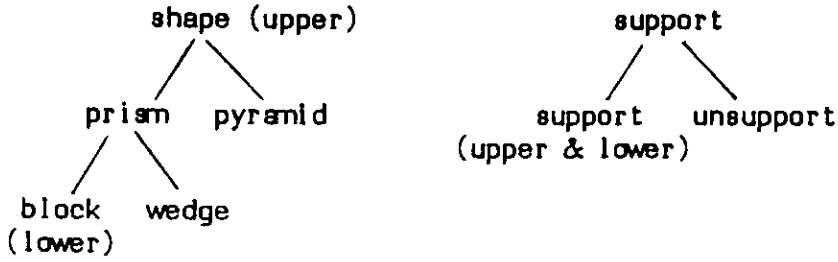

figure 2-1: Two of the trees used to learn the arch concept.

Definitions

The **current marker** in a tree is defined to be the node corresponding to the value of the property described by that tree of the specimen we are studying. A node is **under** a marker if that node belongs to the subtree whose root the marker marks (the node can be the root itself), a node is **above** a marker if it is not below that marker. A specimen can be classified in three ways:

- As an example ("yes"). A specimen is classified this way if all current markers are below the lower markers.
- As being out of the concept ("no"). A specimen is classified "no" if one or more of the current markers is above the upper marker.
- Undecided ("grey"). A specimen is classified this way if all the current markers are below the upper marker and at least one current marker is above the lower marker (otherwise it would be classified "yes").

2.2. Learning a concept

To learn a concept we should start by presenting an example of this concept. On each tree the program places the lower marker on the current marker and the upper marker on the root. The lower markers determine the part of the universe definitely in the concept (the most specific view), the upper markers determine the most general view: all except what is **known** to be out of the concept.

We now present the program a set of examples and counter examples and adjust our markers in the following way:

- In case of an example:
  - * If the classification is "yes": do nothing (correctly classified).

* If the classification is "grey": raise the lower marker on all the trees where the current marker is above the lower marker (grey trees). The lower markers should be placed on the root of the smallest subtree containing the old lower marker and the current marker (then both are below the new lower marker).

* If the classification is "no": contradiction, see discussion below.

- In case of a counter example:

* If classification is "yes": contradiction, see discussion below.

* If classification is "grey": If there is **one** tree with the current marker above the lower marker (a tree whose current marker is below the upper and above the lower marker is called a "grey" tree) we call this a "near miss". If there are more of these grey trees we call it a "far miss". In general we should place at least one upper marker on a node so that the current marker on that tree is above the upper marker (classification should become "no"). If the counter example is a near miss the grey tree should be used to discriminate upon. Otherwise we have a choice point: in theory we can discriminate on every non-empty subset of the grey trees. See the discussion below for more details.

* If classification is "no" there is nothing to be done.

Termination

The learning process is terminated if, on each tree, the upper and lower markers are in the same position. (*)

2.3. Problems with focussing

- In case a far miss is encountered the program is in trouble. It can not decide which subset of the grey trees to use in order to discriminate. Possibilities to handle this case are discussed in [Bundy e.a. '84]. Under them are: setting up a search space, using a teacher, neglecting and avoiding (= adapting the training instances).

- The concept described by the markers is conjunctive. Focussing is not able to cope with disjunctive concepts [Bundy e.a. 1984].

(*) This is only true if the hierarchy of the trees is right.

- Focussing is not capable to deal with noisy data.

- It is possible that the description space is not suitable for learning the concept we want to learn. There are three possible flaws in a description space:

  * There is a tree missing (an aspect relevant to decide what is in or out of the concept is not considered).

  * A tree is not detailed enough. For example it might be necessary to refine the shape tree of figure 2-1 by splitting the concept "pyramid" in "pyramid_4" and "pyramid_3" (a pyramid with a square c.q. a triangle at the bottom).

  * The hierarchy of one or more of the description trees is wrong. (see section 3.1 for an example). In theory there are two ways to handle this situation: change the hierarchy or split the concept into a disjunction.

In the following sections two techniques to change the hierarchy of a description tree are described.

3. TREE HACKING

3.1. The purpose of tree hacking

Tree hacking is an extension on focussing, described in section 2. One of the drawbacks of focussing is that is relies on a suitable set of description trees. Otherwise focussing will over-generalise and/or over-discriminate. See -for example- the following tree:

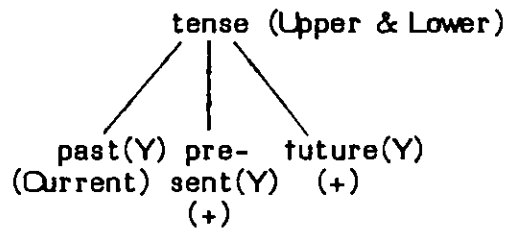

figure 3-1: The status of the tree after two positive instances.

Because of the two positive training instances on present(Y) and future(Y) the lower mark is lifted one level. If, at this moment, we present the program

with a negative training instance with the current marker for this tree on "past" and also for all other trees below the lower marker (*) a conflict is detected. The simple focussing algorithm is in trouble.

On the other hand if the program discriminates on a tree the upper marker is lowered in such a way that an as small as possible subtree is eliminated. Nevertheless it is possible that this part is too large.

These are the two typical situations where a wrong hierarchy of one of the trees in the description space leads to a contradiction in the learning process. Tree hacking is a technique to deal with these situations: it changes the hierarchy of the tree in such way that it becomes consistent with the old data and the specimen that made the inconsistency visible. In order to do this it needs information about the status of each node with respect to the past training instances. Tree hacking splits the original tree in consistent parts and constructs a new tree from these parts.

3.2. When to use tree hacking

Tree hacking might be the right action to perform if a contradiction is found during the learning process. This means that either an example should be classified -on the bases of the rules learned so far- "no" or a counter example should be classified "yes".

To be sure that hacking one or more of the trees is the right action to perform at this moment one should know that the contradiction in the learning process is caused by a faulty hierarchy of one or more trees (and thus excluding all other possible reasons for the detected contradiction). Unfortunately we don't know a good tactic to figure this out in general.

3.3. Information, needed for tree hacking

To be able to hack the tree in an -for this state of the learning process- acceptable hierarchy we have to know the status of each tip-node with respect to the previous training instances. We distinguish between four classes of tip-nodes:

- "Positive" marked tip-nodes.
- "Negative" marked tip-nodes.
- "Visited" marked tip-nodes.

(*) If, on another tree, the current marker is above the lower marker focussing will discriminate on that tree. This is a far miss situation that can not be detected by the focussing algorithm.

- "Undefined" marked tip-nodes.

The program starts with marking all tip-nodes undefined. During learning the process described in the table below is used to mark the nodes.

specimen	classification	action
example	don't care	Mark all current nodes positive.
counter-example	yes	Mark current node in the tree you are going to hack, negative.
counter-example	grey	Mark current node in the tree you are going to use for discrimination negative.
counter-example	no	If only one of the current nodes is marked undefined: mark it negative. If there are more of these nodes mark them visited. These marks are necessary to make sure that, once correctly classified, a specimen will always be correctly classified.

table 3-1: Marking of nodes to obtain necessary information for tree hacking.

3.4. The algorithm

Fortunately, the hacking algorithm for the two cases in which tree hacking is useful are identical. However the cases differ in the algorithm to detect which tree should be hacked.

Example presented, but classification gives "no":

- Hack all trees with the current mark above the upper mark. It is advisable not to hack the whole tree, but the smallest subtree containing both the current mark and the upper mark (and thus the lower mark). This is the smallest part to restore consistency in the whole tree.

Counter-example presented, but classification gives "yes":

- Find those trees for which the the current marker is on a not earlier marked tip-node. If there exist more of these trees then we have to deal

with an analogous problem to the far miss problem during focussing: we have to choose which tree to hack. At this moment the program just takes one of these trees.

- Hack the selected tree. The smallest possible tree to hack is the subtree with as root the lower mark. In spite of this it is not advisable to hack just that subtree because the upper mark will be lowered by the hacking program to somewhere in the changed subtree, excluding the whole grey area that existed. Therefore it is better to hack the subtree with as root the upper marker.

The hack algorithm

There are many ways to hack a tree to a consistent tree. The algorithm below describes one that destroyes as little as possible from the original hierarchy and grey area.

- Split the tree into the following three sets of subtrees:

  * A set of **"consistent positive"** subtrees. A consistent positive subtree is an as large as possible subtree of the original tree, only containing positive marked tip-nodes and undefined marked tip-nodes, with at least one positive marked tip-node.

  * A set of **"consistent negative"** subtrees. A consistent negative subtree is an as large as possible subtree, only containing negative, visited and undefined marked tip-nodes.

  * A set of **"consistent undefined"** subtrees. A consistent undefined subtree is an as large as possible subtree, only containing undefined tip-nodes.

- Rearrange the subtrees to a complete tree in the following way:

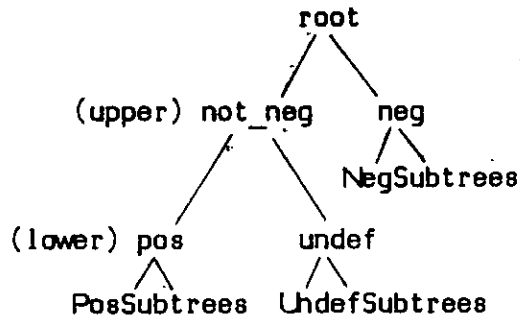

figure 3-2: Structure of the new tree

One may wonder what the purpose of the visited markers are. Their role is to make sure that a negative training instance that was once classified correct will still be classified correct after tree hacking. Without using the visited markers it is possible that a negative training instance that was once specified correct because it encountered an undefined marked tip-node above the upper marker will be classified wrong after the hacking process because the tip-node is included in the unmarked subtrees.

3.5. An example

This example is derived from Winston's arch program. I will discuss the focussing on the tree, that describes the shape of the upper part of the arch. The tree for describing the hierarchy of shapes is extended to show the working of the algorithm in a clear way. Suppose this tree starts with the following structure:

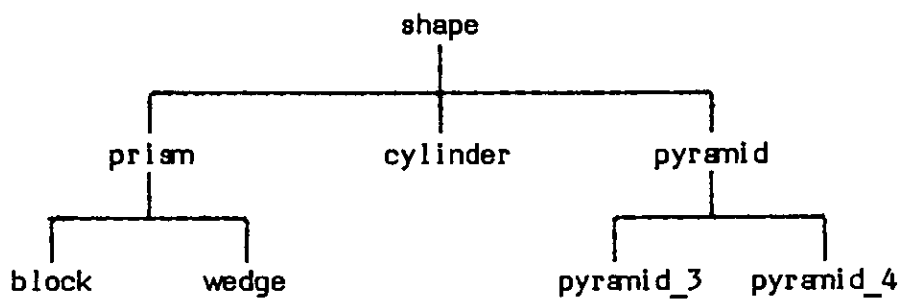

figure 3-3: Shape tree as used for testing the tree hacking algorithm.

The following table shows the given specimen, and the actions that the learning program performs (We just consider what happens at the top of the arch, it is assumed that the other parameters don't lead to far misses).

specimen	in concept	action.
block	yes	Put upper mark on shape, lower on block.
pyramid_4	yes	Lower mark is raised to shape.
pyramid_3	no	A contradiction is detected, the whole tree is hacked to the shape shown in the figure below.

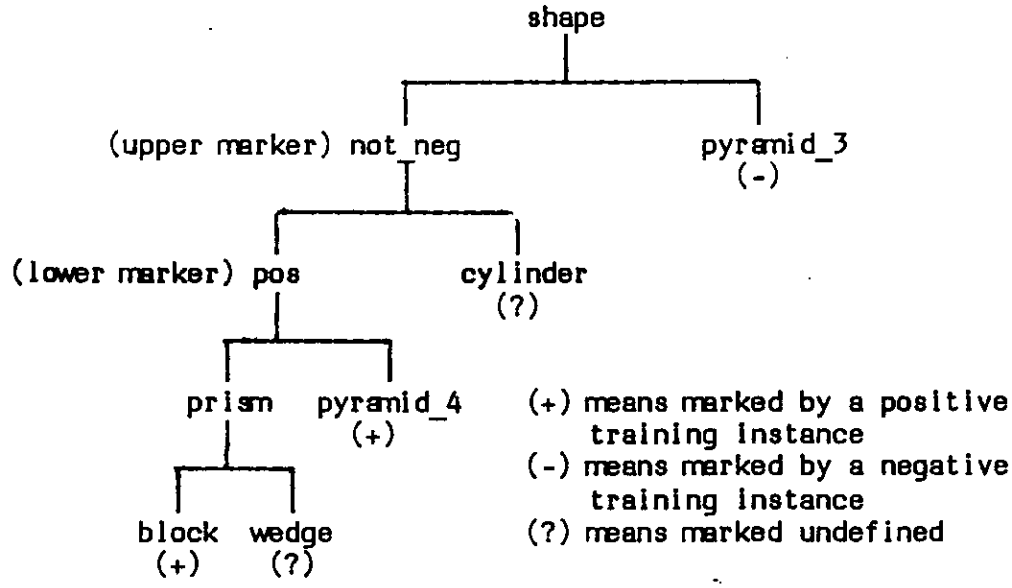

figure 3-4: Resulting tree after hacking.

table 3-2: A worked example of the extended focussing program.

3.6. Evaluation

Although this way of tree hacking satisfies if you just want to enable the program handling wrongly structured description trees it is far from optimal. The major drawbacks of the algorithm are the following:

- If a certain tree is used in more than one concept we are learning, and/or more than one time in a concept, then the program stores a copy of the tree for every occurrence of the tree. This is done because it is possible that the shape for one occurrence of the tree can't be equal to another occurrence.

- It is likely that if a tree is wrong shaped and used for various concepts it will get different shapes in most occurrences. I believe that it is preferable to check whether it is possible to hack the tree so that it satisfies as many as possible occurrences. I will discuss this idea in more detail in the next section.

4. BUILDING A DESCRIPTION SPACE FOR FOCUSSING

When using focussing for learning a concept there exist a lot of possible ways to build and alter the hierarchy of the trees which constitute the description space. In this section I present a way to build and maintain the description trees needed for focussing. A tree that describes a certain property can be used to learn properties of various objects or relations in a concept and for many concepts. I will call these occurrences of the tree.

4.1. Suitable and ideal description trees

A suitable tree

If a certain concept is completely learned the upper and lower markers on each tree are on the same node. The set of properties of each tree is then divided into two parts: a subset below the markers and a subset above the markers. This division is determined by the concept, and therefore it is only possible to learn a specific concept if each tree has a node in it so that all tip-nodes below that node refer to specimen in the concept and all other to specimen outside the concept. Apart from this it is unimportant (*) what the structure of the subtree below this node is and how the rest of the tree is structured.

A proposal for an "ideal" tree

An "ideal" tree is a tree whose hierarchy is likely to be suitable for learning a certain concept. One thing is sure: you can not build the "ideal" shaped tree by looking at just one occurrence of the tree. The reason for this is that all suitable trees (see above) are as ideal as each other if you just examine this one occurrence. This means that you should a) have more information about the properties you want to include in the tree and/or b) use information from other (earlier) learned concepts.

I choose to examine the possibility to use information from other occurrences of the tree (other concepts). An obvious solution is to build a tree that satisfies an as large as possible subset of the occurrences. I will call this tree the "consensus" tree. If the set of occurrences is representative for the universe

(*) It only affects the effectiveness of the heuristics.

of occurrences of this tree we build the tree most likely to be suitable for learning an **arbitrary** concept. This does not mean it is ideal for a particular concept, but without additional information we can't do better.

4.2. Computing the consensus tree

The main problem in this approach is computing the tree that fits for as many as possible occurrences. The formalisation of this problem is given below:

given:

- A set of properties F.

- A set of occurrences C. Each occurrence C(i) of this set has two other sets connected to them: Cmax(i) (the disjunction of the positive and undefined marked properties), and Cmin(i) (the positive properties).

  For example look at the tree of figure 3-4. Here F would be {block, wedge, pyramid_4, cylinder, pyramid-3, Cmin would be {block, pyramid_4} and Cmax {block, wedge, pyramid_4, cylinder}.

To compute:

A tree T that has the following properties:

- For as many as possible occurrences C(i) there exits a subtree S. The set of tip-nodes of S is a superset of Cmin(i), and a subset of Cmax(i).

Of course this algorithm should have an acceptable order: generating all tree and selecting the best is not a solution in a practical sense of view.

A heuristic approach

While learning the concepts the sets Cmax(i)\Cmin(i) (the grey area) will gradually disappear. This means that if we can solve the problem stated below we have a heuristic solution, that will get closer to the ideal solution as the grey area gets smaller.

To compute in a reasonable order:

A tree T that has the following properties:

- For as many as possible concepts C(i) there exist a subtree S. The set of tip-nodes of S is Cmin(i). After constructing this tree try (in a heuristic way) to include the remaining concepts by using the grey areas.

The first part of the problem is well defined and solvable in order N cubed (N is the number of concepts involved). Adapting the construction if only one subset Cmin(i) changes can be even more efficient. I have not done any research to the heuristic part of this algorithm but I believe it should be possible to write an acceptable algorithm for this part with order N.

4.3. A criterion for the existence of one consensus tree

Lemma 1

Given a set of property-sets Cmin. There exist a tree, so that for every Cmin(i) there is a subtree with a set of properties equal to Cmin(i) if and only if for every Cmin(i), Cmin(j) one of the following statements is true:

- Cmin(i) and Cmin(j) are disjoint.
- Cmin(i) is a subset of Cmin(j).
- Cmin(j) is a subset of Cmin(i).

Proof:

If this criterion is met it is possible to build the tree with the algorithm given below. This proves that such a tree exists.

If this criterion is not met there exist two sets Cmin(i) and Cmin(j) with a nonempty intersection I(i,j). The elements of I(i,j) have to be in two subtrees: the subtree with root S(i) for Cmin(i) and the subtree with root S(j) for Cmin(j). The subtree with root S(i) must have a tip-node set Cmin(i). The elements of Cmin(j)\I(i,j) should be outside this subtree, but in the subtree of S(j). Thus S(j) is not a node in the subtree of S(i). In the same way S(i) is not a node in the subtree of S(j). Because S(i) and S(j) are both in the tree there must be a loop (root - S(i) - "element of I(i,j)" - S(j) - root). Contradiction.

4.4. The algorithm

Given the set of properties F and a set of Cmin sets of which we know they satisfy lemma 1, we can compute the consensus tree as follows:

Let the tree be represented as:

```
tree ::= tree(<node name>,
              [list of properties in the subtree of the node],
              [list of subtrees])
```

- Initialise the tree as tree(<root>,F,[]).
- DO (for every Cmin) fit_in(Cmin,[Tree],[NewTree]) OD.

To fit Cmin in the list of trees the following actions have to be performed:

- Split the subtrees in the following 4 classes:

    1. A class of subtrees for which the property set is equal to Cmin.
    2. A class of subtrees for which the property set is disjoint to Cmin.
    3. A class of subtrees for which the property set is a superset of Cmin.
    4. A class of subtrees for which the property set is a subset of Cmin.

- Because of lemma 1 and the way the tree is built three different situations can occur:

    * If class 1 is nonempty no action has to be performed because there is an equal Cmin set fitted in the tree earlier.

    * If class 3 is nonempty we fit the Cmin set in the tree of class 3. There are never more then one set in class 3 because of lemma 1.

    * Otherwise we built a new subtree. The root of this subtree is the node S for Cmin. All subtrees of class 4 are attached to this node. The new list of subtrees is the union of class 2 and the new subtree.

4.5. Splitting the occurrences

To use the algorithm above we should first split the Cmin sets in an as large as possible part containing Cmin sets that can be together in a tree and the rest. The following algorithm does this:

- Build a graph. The vertices are the Cmin sets. If two sets do not meet lemma 1 they should be connected by an arc.

- Do until there are no arcs left in the graph:

    * Eliminate the vertex with most arcs connected to it. If there are two or more vertices with an equal number of arcs remove an arbitrary member of this set.

The remaining set of vertices is the largest non conflicting set of Cmin sets.

4.6. Experiments

On the basis of this theory I built a Prolog program to learn simultaneously a set of concepts. As mentioned I did not pay any attention to the heuristic part of constructing the trees. I planned to do this after writing and testing the version without, but ran out of time.

The program handles incoming specimens using focussing. If a contradiction is detected in the focussing process the program looks for trees to be altered in the same way as the tree hacking extension described in section 3. Then all occurrences of this tree are considered and a new set of trees created. There are many other ways to use this description space building algorithm in focussing. For a further discussion on this subject see section 4.7.

Because of the limited time I decided not to construct a "real life" testing environment. Instead I tested the program with a set of nine abstract concepts, sharing a description tree with ten properties. The concepts where designed to need three different trees.

The program was then tested by presenting it 90 specimens in a random order, but so that for a specific concept the first one always is an example (necessary for focussing). To reconstruct the three trees from one tree in which all properties were directly attached to the root it needed to change the description space 17 times.

4.7. Further research

In this section an algorithm is discussed that uses information of other occurrences of the same tree to build a hierarchy for description trees. The presented algorithm is not worked out well and needs further research on the following aspects:

- What is the optimal heuristic algorithm to maximise the number of occurrences of the tree for which one common hierarchy fits? Solving this

problem will especially improve the behaviour of the algorithm on partly learned concepts.

- How should the tree changing algorithm interact with the focussing program. Some possibilities are:

  * Change all occurrences of the tree after every specimen.

  * Change all occurrences of the tree if a contradiction occurs in the focusing process (the solution I choose).

  * If a contradiction in the learning process occurs look to see if it is possible to use another tree of this set. If not then change all occurrences.

  * Change part of the occurrences in one of the above situations.

- In what sense can the information stored from other concepts help to provide heuristics for choice points in the learning process (far misses, choosing between one of the alternative options described above etc.)

5. SUMMARY

This paper presents technique -called tree hacking- to enable focussing learning concepts while the hierarchy of one or more of the description trees is wrong. Ways and problems to select a tree which hierarchy should be changed are discussed. An algorithm that restores the consistency of the tree but changes as little as possible of its hierarchy is presented.

Section 5 starts a discussion about how to build an optimal hierarchy for description trees. The basic idea of this section is that the optimal hierarchy is the hierarchy that is most likely to be suitable to learn a new -arbitrary- concept. If other occurrences of the same tree is the only available information this tree is the tree that is suitable for as many as possible of these other occurrences. An outline of a concept learning program based on this idea is presented, together with suggestion for further research.

6. REFERENCES

[Bundy e.a. '84] Bundy, A., Silver,B., Plummer, D., "An analytical comparison of some rule learning programs", D.A.I Research Paper No. 215, 1984.

[Bundy '82] Bundy, A., "Changing a description space by tree

hacking", D.A.I. Note 106, 26 march 1982

[Clocksin & Mellish '81] Clocksin, W.F., Mellish, C.S., "Programming in Prolog", Springer-Verlag Berlin Heidelberg, 1981.

[Winston '75] Winston, P., "Learning structural descriptions from examples", in Winston P.H. (editor), The psychology of computer vision, McGraw Hill, 1975.