Chapter 7

Searching the generalization/specialization graph

7.1 Basic idea

Given a constructive operator for generalization/specialization, for each example $e \in E^+$ we can build a directed graph (hierarchy) of hypotheses with two terminal nodes – the most specific element e, and the most general hypothesis $\top$. This graph will also include all correct hypotheses (not covering negative examples) that cover at least one positive example (e). And, as usual we will be looking for hypotheses covering more positive examples, i.e. maximally general ones. As this graph is a strict hierarchical structure, for each hypothesis h, the length of the path between h and e or h and $\top$ can play the role of a measure for the generality/specificity of h. Thus some standard graph search strategies as depth-first, breadth-first or hill-climbing can be applied. Depending on the starting point of the search we may have two basic search approaches:

- Specific to general search: starting from e and climbing up the hierarchy (applying generalization operators), i.e. searching among the correct generalizations of e for the one with maximal coverage.
- General to specific search: starting from $\top$ and climbing down the hierarchy (applying specialization operators), i.e. searching among the incorrect hypothesis which at the next specialization step can produce a correct hypothesis with maximal coverage.

The above discussed search procedures are usually embeded as step one in a covering learning algortihm. To illustrate this in the following two section we discuss two examples – searching the space of propositional hypothesis and heuristic general to specific search for relational hypotheses.

7.2 Searching the space of propositional hypotheses

Consder the MONKS concept discussed in chapter 3. Let us select a positive example, say $e = [octagon, octagon, no, sword, red, yes]$ (example 1 from Figure 3.1, where the attribute names are omitted for brevity). The most general hypothesis in this language is $\top = [\_,\_ ,\_ ,\_ ,\_ ,\_]$

(underscores denote "don't care" or any value). The first level of the specialization hierarchy starting from $\top$ is:

```
[octagon,_,_,_,_,_]
[_,octagon,_,_,_,_]
[_,_,no,_,_,_]
[_,_,_,sword,_,_]
[_,_,_,_,red,_]
[_,_,_,_,_,yes]
```

Note that the values that fill the don't care slots are taken from e, i.e. the choice of e determines completely the whole generaliztion/specialization hierarchy. Thus the bottom level of the hierarchy is:

```
[_,octagon,no,sword,red,yes]
[octagon,_,no,sword,red,yes]
[octagon,octagon,_,sword,red,yes]
[octagon,octagon,no,_,red,yes]
[octagon,octagon,no,sword,_,yes]
[octagon,octagon,no,sword,red,_]
```

A part of the generalization/specialization hierarchy for the above example is shown in Figure 7.1. This figure also illustrates two search algorithms, both version of depth-first search with evaluation function (hill climbing). The top-down (general to specific) search uses the hypothesis accuracy $A(h_k)$ for evaluation. It is defined as

$$A(h^k) = \frac{|\{e|e \in E^k, h^k \geq e\}|}{|\{e|e \in E, h^k \geq e\}|},$$

i.e. the number of examples from class k that the hypothesis covers over the total number of examples covered. The bottom-up search uses just the total number of examples covered, because all hypotheses are 100% correct, i.e. $A(h_k) = 1$ for all k. The bottom-up search stops when all possible specializations of a particular hypothesis lead to incorrect hypotheses. The top-down search stops when a correct hypothesis is found, i.e. the maximum value of the evaluation function is reached. The figure shows that both search algorithms stop at the same hypothesis – $[octagon, octagon, _, _, _, _]$, which is, if fact, a part of the target concept (as shown in Section 3.1)

The above described process to find the hypothesis $h = [octagon, octagon, _, _, _, _]$ is just one step in the overall covering algorithm, where the examples covered by h are then removed from E and the same process is repeated until E becomes empty.

7.3 Searching the space of relational hypotheses

In this section we shall discuss a basic algorithm for learning Horn clauses from examples (ground facts), based on general to specific search embeded in a covering strategy. At each pass of the outermost loop of the algorithm a new clause is generated by θ-subsumption specialization of the most general hypothesis $\top$. Then the examples covered by this clause are removed and the process continues until no uncovered exampes are left. The negative examples are used in the inner loop that finds individual clauses to determine when the current clause needs further specialization. Two types of specialization operators are applied:

1. Replacing a variable with a term.
2. Adding a literal to the clause body.

Figure 7.1: Generalization specialization graph for MONKS data

These operators are minimal with respect to θ-subsumption and thus they ensure an exhaustive search in the θ-subsumption hierarchy.

There are two stopping conditions for the inner loop (terminal nodes in the hierarchy):

- Correct clauses, i.e. clauses covering at least one positive example and no negative examples. These are used as components of the final hypothesis.

- Clauses not covering any positive examples. These are just omitted.

Let us consider an illustration of the above algorithm. The target predicate is $member(X, L)$ (returning true when X is a member of the list L). The examples are
$E^+ = \{member(a, [a, b]), member(b, [b]), member(b, [a, b])\}$,
$E^- = \{member(x, [a, b])\}$.
The most general hypothesis is $\top = member(X, L)$. A part of the generalization/specialization graph is shown in Figure 7.2. The terminal nodes of this graph:
$member(X, [X|Z])$
$member(X, [Y|Z]) : -member(X, Z)$
are correct clauses and jointly cover all positive examples. So, the goal of the algorithm is to reach these leaves.

A key issue in the above algorithm is the search stategy. A possible approach to this is the so called iterative deepening, where the graph is searched iteratively at depths 1, 2, 3,..., etc. until no more specializations are needed. Another appraoch is a depth-first search with an evaluation function (hill climbing). This is the approach taken in the popular system FOIL that is briefly described in the next section.

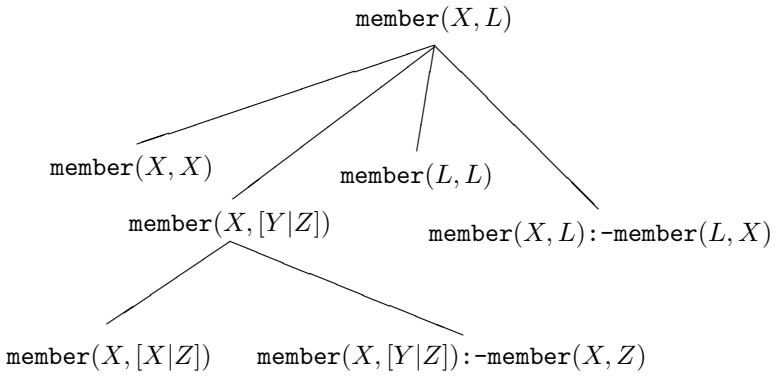

Figure 7.2: A generalization/specialization graph for $member(X, L)$

7.4 Heuristic search – FOIL

7.4.1 Setting of the problem

Consider the simple relational domain also discussed in Section 6.3 – the *link* and *path* relations in a directed acyclic graph. The background knowledge and the positive examples are:

$BK = \{link(1,2), link(2,3), link(3,4), link(3,5)\}$
$E^+ = \{path(1,2), path(1,3), path(1,4), path(1,5),$
$path(2,3), path(2,4), path(2,5), path(3,4), path(3,5)\}$

The negative examples can be specified explicitly. If we assume however, that our domain is closed (as the particular *link and path* domain) the negative examples can be generated automatically using the *Closed World Assumption (CWA)*. In our case these are all ground instances of the *path* predicate with arguments – constants from E^+. Thus

$E^- = \{path(1,1), path(2,1), path(2,2), path(3,1), path(3,2), path(3,3),$
$path(4,1), path(4,2), path(4,3), path(4,4), path(4,5), path(5,1), path(5,2),$
$path(5,3), path(5,4), path(5,5)\}$

The problem is to find a hypothesis H, i.e. a Prolog definition of *path*, which satisfies the *necessity* and *strong consistency* requirements of the induction task (see Chapter 2). In other words we require that $BK \wedge H \vdash E^+$ and $BK \wedge H \nvdash E^-$. To check these condition we use *logical consequence* (called also *cover*).

7.4.2 Illustrative example

We start from the *most general* hypothesis

$$H_1 = path(X, Y)$$

Obviously this hypothesis covers all positive examples E^+, however many negative ones too. Therefore we have to specialize it by adding body literals. Thus the next hypothesis is

$$H_2 = path(X, Y) :- L.$$

The problem now is to find a proper literal L. Possible candidates are literals containing only variables with predicate symbols and number of arguments taken from the set E^+, i.e. $L \in \{link(V_1, V_2), path(V_1, V_2)\}$.

Clearly if the variables V_1, V_2 are both different from the head variables X and Y, the new clause H_2 will not be more specific, i.e. it will cover the same set of negatives as H_1. Therefore we impose a restriction on the choice of variables, based on the notion of *old variables*. Old variables are those appearing in the previous clause. In our case X and Y are old variables. So, we require *at least one* of of V_1 and V_2 to be an old variable.

Further, we need a criterion to choose the *best literal* L. The system described here, FOIL uses an *information gain measure* based on the ratio between the number of positive and negative examples covered. Actually, each newly added literal has *to decrease the number of covered negatives maximizing at the same time the number of uncovered positives*. Using this criterion it may be shown that the best candidate is $L = link(X, Y)$. That is

$$H_2 = path(X, Y) : -link(X, Y)$$

This hypothesis does not cover any negative examples, hence we can stop further specialization of the clause. However there are still uncovered positive examples. So, we save H_2 as a part of the final hypothesis and continue the search for a new clause.

To find the next clause belonging to the hypothesis we exclude the positive examples covered by H_2 and apply the same algorithm for building a clause using the rest of positive examples. This leads to the clause $path(X, Y) : -link(X, Z), path(Y, Z)$, which covers these examples and is also correct. Thus the final hypothesis H_3 is the usual definition of path:

```
path(X,Y):-link(X,Y).
path(X,Y):-link(X,Z),path(Z,Y).
```

7.4.3 Algorithm FOIL

An algorithm based on the above ideas is implemented in the system called FOIL (First Order Inductive Learning) [2]. Generally the algorithm consists of two nested loops. The inner loop constructs a clause and the outer one adds the clause to the predicate definition and calls the inner loop with the positive examples still uncovered by the current predicate.

The algorithm has several critical points, which are important for its efficiency and also can be explored for further improvements: +

- The algorithm performs a search strategy by choosing the *locally best branch* in the search tree and further exploring it *without backtracking*. This actually is a *hill climbing* strategy which may drive the search in a local maximum and prevent it from finding the best global solution. Particularly, this means that in the inner loop there might be a situation when there are still uncovered negative examples and there is no proper literal literal to be added. In such a situation we can allow the algorithm to add a new literal without requiring an increase of the information gain and then to proceed in the usual way. This means to force a further step in the search tree hopping to escape from the local maximum. This further step however *should not lead to decrease of the information gain* and also *should not complicate the search space* (increase the branching). Both requirements are met if we choose *determinate literals* (see Chapter 8) for this purpose.

  Using determinate literals however does not guarantee that the best solution can be found. Furthermore, this can complicate the clauses without actually improving the hypothesis with respect to the *sufficiency* and *strong consistency*.

- When dealing with *noise* the *strong consistency* condition can be weakened by allowing the inner loop to terminate even when the current clause covers some of the negative examples. In other words these examples are considered as noise.

- If the set of positive examples is *incomplete*, then CWA will add the missing positive examples to the set of negative ones. Then if we require strong consistency, the constructed hypothesis will be specialized to exclude the examples, which actually we want to generalize. A proper *stopping condition* for the inner loop would cope with this too.