LEARNING CONCEPTS BY ASKING QUESTIONS

Claude Sammut

Department of Computer Science
University of New South Wales
Sydney, Australia

Ranan B. Banerji

Department of Mathematics and Computer Science
Saint Joseph's University
Philadelphia, PA 19131

ABSTRACT

Two important issues in machine learning are explored: the role that memory plays in acquiring new concepts; and the extent to which the learner can take an active part in acquiring these concepts. This chapter describes a program, called Marvin, which uses concepts it has learned previously to learn new concepts. The program forms hypotheses about the concept being learned and tests the hypotheses by asking the trainer questions.

Learning begins when the trainer shows Marvin an example of the concept to be learned. The program determines which objects in the example belong to concepts stored in the memory. A description of the new concept is formed by using the information obtained from the memory to generalize the description of the training example. The generalized description is tested when the program constructs new examples and shows these to the trainer, asking if they belong to the target concept.

1. INTRODUCTION

From personal experience we know that it is hard to learn new concepts unless we already understand quite a lot about the subject we are studying. For example, when mathematics is taught in school, the teacher begins with simple problems like: what are numbers? Later, more complicated concepts such as addition, followed by multiplication are presented. Usually many years of patient accumulation of knowledge in a given field are required before one can fully understand the most complex concepts in that field. To be able to learn in stages, the student must have a memory which he can use to integrate all the knowledge acquired. As new concepts are added to the memory, the student's vocabulary broadens, enabling him to describe still more complex concepts.

As well as using his memory, a human student also asks questions to help him learn. Often a teacher will show the student an example to explain a new concept. However, a few examples are not always enough to completely define the concept. It is easy to generalize from a small number of examples and come up with an incorrect idea of the concept which the teacher wants the student to learn. Questions often help to identify those parts of the concept which have misunderstood.

Marvin is a program which can learn complex concepts by using its memory of simpler concepts to help describe the newer concepts. Marvin also asks the trainer questions to check that its description of the concept is correct. A concept is informally defined as the description of a set of objects in some universe. We say that an object is a *positive example* of a concept if the object is in the set defined by the concept description. A *negative example* is an object not contained in the set.

Marvin begins learning a new concept when the trainer shows it an object which is a positive example of the concept to be learned (the *target concept*). The description of the example represents a concept which contains only one object, the example itself. The target concept is a generalization of the example because it contains that object and perhaps many other objects. One concept, *P*, is a *generalization* of another concept, *Q*, if *Q* describes a subset of *P*. Alternatively, we say that *Q* is a *specialization* of *P*. Marvin's task is to discover the description of the target. It will do this by searching for a generalization of the initial example which contains all the positive examples of the target and none of the negative examples.

Suppose *P* is a generalization of the initial example. Also suppose that *P* describes a subset of the target. That is, *P* only contains objects which are positive examples of the target. We say that *P* is a *consistent generalization* of the initial example. If *Q* is a generalization which contains an object not in the target then we will say that *Q* is an *inconsisent generalization* of the target. A generalization of the initial example is called a *trial concept*.

Marvin's search for the target concept is *specific-to-general*. That is, the program begins with the initial example and creates a new trial concept by generalizing the example slightly. If the generalization is consistent, then Marvin creates a new trial which is a generalization of the previous one. To find out if a generalization is inconsistent, the program constructs objects which are contained in the trial and shows them to the trainer. If the trainer answers that one of the objects is not contained in the target then the generalization is inconsistent. When this occurs, Marvin tries to create a new trial which is more specific than the previous one. That is, it tries to create concept which excludes the negative examples of the target. If specialization fails to produce a consistent trial then a different generalization is tried. Marvin will continue to make as many generalizations as it can as long as can maintain consistency. When it has run out of generalizations, the concept description is stored in the the program's memory.

So far we have not specified how Marvin describes concepts. The concept description language is a subset of first order predicate logic very similar to Prolog. Marvin constructs example objects by taking a concept description and executing it as if it were a program. For Marvin, learning a concept is equivalent to synthesizing a logic program.

The program consists of four major components:

- A memory which contains the descriptions of concepts that have been learned or provided by the trainer as background knowledge.
- A pattern matcher which determines if objects shown to the program belong to concepts stored in the memory.
- A simple theorem prover which, given a concept description, generates objects which satisfy the conditions in the description.
- A search strategy which directs the operation of the other components in order to find a description for the concept that the trainer is trying to teach the program.

The work on Marvin has grown out of earlier efforts by Banerji (1964) and Cohen (1978) who both stressed the importance of a learning system being able to extend its power to describe concepts through learning. In the following section we describe the representation language used by Marvin.

2. REPRESENTING CONCEPTS IN FIRST ORDER LOGIC

A concept is represented by a set of *Horn clauses*. These are expressions in first order predicate calculus which have the form:

$$P(X) \leftarrow Q(X) \;\&\; R(X) \;\&\; S(X).$$

That is, an object, X, belongs to the concept, P, if the predicates Q and R and S are true. The clause will be called a definition of the concept P.

As an example, let us describe the concept *tee*. An object, X, is a *tee* if it consists of two objects. One object, A, may have any shape and lies on top ofga another object, B, which is a brick and which is standing up.

```
tee(X) ←
        A is_part_of X &
        B is_part_of X &
        A is_on B &
        any_shape(A) &
        lying(A) &
        brick(B) &
        standing(B).
```

We can think of this clause as describing the set of all objects, X, which satisfy the conditions on the right hand side of the arrow. *any_shape* is defined as:

```
any_shape(X) ← brick(X).
any_shape(X) ← wedge(X).
```

That is, the shape of an object in this world may be a brick or a wedge.

Marvin's *long term* memory is a database of such clauses. The program also has a *short term* memory which contains the set of *facts* which describe individual objects which the trainer presents as examples. By fact we mean a *unit clause*, that is, a clause with no right hand side. Thus an instance, x, of the concept *tee* would appear in the short term memory as:

```
a is_part_of x.
b is_part_of x.
b is_on a.
brick(a).
wedge(b).
standing(a).
lying(b).
```

This describes a wedge lying on top of a brick.

Originally we defined generalization and specialization in terms of sets of objects. These definitions help to understand what those operations mean, but they do not give us an effective way of performing generalizations or specializations. To construct a description of a concept, *P*, which is more general than *Q*, the program must transform the expression in the description language which represents *Q* into another expression which represents *P*. That is, generalizations and specializations must be defined as operations which manipulate the description language.

3. GENERALIZATIONS

Suppose *x* is an object which is defined by the following predicates: '$Q_1(x)$ & $R_1(x)$ & $S_1(x)$'. If there is a concept which is represented by:

$$P(X) \leftarrow Q_1(X) \ \& \ R_1(X) \ \& \ S_1(X).$$
$$P(X) \leftarrow Q_2(X) \ \& \ R_2(X) \ \& \ S_2(X).$$

then *x* is a positive example of the concept *P*. Note that the conjunction of predicates which describes *x* matches the conjunction on the right hand side of the first clause. Now suppose *x* is an object which appears in some visual scene which we show to Marvin. Since *x* is an example of *P*, the program can try to generalize the description of the scene by asking: "If I replace *x* with some other example of *P*, has the scene been changed in any essential way?" In the last example of the previous section, *b* was a wedge and also an instance of *any_shape*. The description of *x* can be generalized by replacing *wedge(b)* by *any_shape(b)*. *Wedge(b)* matches the right hand side of one clause of *any_shape*.

When we perform replacement operations, we think of clauses as rewrite rules. Rewrite rules are most commonly used to define the grammar of a language. In fact, the set of clauses stored in Marvin's memory defines a language. Sentences in the language describe objects which belong to concepts known to Marvin. We should be careful to distinguish between the concept description language (i.e Horn clauses in first order logic) and the language which describes the set of all objects recognizable by the program. The latter is a subset of the description language. Shapiro (1981) gives a more rigorous discussion of the relationship between these two languages.

A *replacement* such as changing *wedge(b)* to *any_shape(b)*, is the fundamental operation used in generalization. When there is a large number of concepts stored in the long term memory, many such replacements are possible. The description of one instance of a concept may be transformed into many different generalizations. The main problem for the learning program is to efficiently search through the space of all such generalizations. Before giving a formal definition of generalization, let us look at another example.

Suppose we wish to describe a column of bricks as a brick standing on the ground or a brick standing on another column. This is a recursive description consisting of two clauses:

```
column(X) ←
        brick(X) &
        standing(X) &
        X is_on Y,
        ground(Y).

column(X) ←
        brick(X) &
        standing(X) &
        X is_on Y &
        column(Y).
```

Variables such as Y are implicitly existentially quantified. Suppose that these clauses, as well as others, form the set of clauses in the long term memory, $S_i \leftarrow M_i$. Also assume that we have a set of facts in short term memory, C_0, which consists of the unit clauses:

brick(a).	(1)
standing(a).	(2)
a is_on b.	(3)
brick(b).	(4)
standing(b).	(5)
b is_on c.	(6)
ground(c).	(7)

If we substitute *b* for the variable *X* and *c* for the variable *Y* then predicates 4, 5, 6 and 7 match the right hand side of the first clause in the definition of 'column'. That is, *b* is an example of the concept *column*. The *substitution* for this match is the set of pairs {X/a, Y/B}. We can *elaborate* C_0 by adding the predicate *column(b)* to form C_1.

brick(a).	(1)
standing(a).	(2)
a is_on b.	(3)
brick(b).	(4)
standing(b).	(5)
b is_on c.	(6)
ground(c).	(7)
column(b).	(8)

Now predicates 1, 2, 3, and 8 match the right hand side of the second clause of 'column' with the substitution {X/a, Y/b}. Thus we can construct a new set C_2 by adding *column(a)* to C_1.

The process we call elaboration simply entails finding out which objects in the short tem memory are examples of concepts that Marvin knows. This requires the program to match object descriptions with the right hand sides of clauses and then adding the adding the left hand side to the short term memory.

We will now define the pattern matching operations used in the elaboration. It is convenient for us to think of conjunctions of predicates as being equivalent to sets of predicates. In what follows, sets will almost always contain predicates or clauses, not objects.

Definition 1: Given a set of clauses:

$$S_1 \leftarrow M_1$$
$$\ldots\ldots\ldots\ldots$$
$$S_k \leftarrow M_K$$

and a set of predicates, C, we say that C´ is an *elaboration* of C if there is an M_i and a substitution, σ, such that

$$\sigma M_i \subseteq C \text{ and } C' = C \cup \sigma\{S_i\}.$$

The set, C, represents the set of facts which are presented to the program by the trainer as a description of an instance of the concept to be learned. This description is expanded, or elaborated, by finding those clauses stored in memory whose right hand sides match a subset of C. The left hand sides of the clauses are added to C to form C´. The pattern matcher uses a simple unification procedure (Robinson, 1965) to construct substitutions for the variables in the clause. The effect of this elaboration is to augment the description of the example using the knowledge that Marvin has acquired previously.

When predicate (8) was added to C, it enabled Marvin to find more matches in memory. In this way, sets of predicates can be elaborated repeatedly giving us a sequence of new sets derived from C_0.

Definition 2: Given a set, C_0, we define a sequence $C_0..C_n$ such that C_{i+1} is an elaboration of C_i and C_n cannot be elaborated further. We write

$$\text{All\_Elaborations}(C_0) = C_n,$$

representing the set of all predicates derived from C_0.

The information obtained through elaboration is used to construct hypotheses or *trials* for the concept being learned. The initial trial, T_0, is always equivalent to C_0. One trial, T_i, may be generalized to a new trial, T_{i+1} by replacing predicates in T_i which match the right hand side of a clause with the predicate on the left.

Again using the column example, we begin with T_0 equivalent to C_0. The first trial, T_1, may be obtained by replacing the predicates 4, 5, 6 and 7 with ‘column(b)’, giving:

brick(a).	(1)
standing(a).	(2)
a is_on b.	(3)
column(b).	(8)

Clearly this is a generalization of T_0 since *b* may now be a column of any height. The following definition shows exactly how we arrive at a replacement operation such as the one above.

Definition 3: If T_i is a trial concept and M is a subset of T_i, and there exists a clause in memory $S \leftarrow M'$ such that with the substitution σ, $M = \sigma M'$ then we define a replacement operation which will create a new trial, T_{i+1}, such that

$$T_{i+1} = T_i - M \cup \sigma\{S\}$$

T_{i+1} is called a *generalization* of T_i. If there is more than one clause in the definition of the concept S then T_{i+1} is more general than T_i, since S describes a larger set of objects than is described by M.

In the example, T_i can be generalized further. All of the remaining predicates match the second clause in the description of column. They can all be replaced by the single predicate *column(a).* Thus, we can produce a sequence of more general trials.

Definition 4: If T_0, .., T_k be a sequence of trials such that T_i generalizes to T_{i+1} then we say T_0 satisfies T_k.

Notice that making generalizations is exactly the same as recognizing that an object belongs to a concept. Although the program may construct many generalizations, only those which are less general than or the same as the target are of interest. Once a generalization has been created, we must be able to if it is consistent or not.

Definition 5: Trial T is consistent with the target, T´ , if any object which satisfies T also satisfies T´ .

The next section describes what is done when an inconsistent trial is constructed.

4. SPECIALIZATIONS

A generalization results in a set of predicates in a trial being replaced by a single, more general, predicate. The replacement operation throws away some specific information contained in the set M (i.e. those predicates which matched the right hand side of the clause) in favour of the more general statement, S (i.e. the left hand side of the clause). If the new trial is consistent then the information lost was not important. However, if the generalization is inconsistent then too much information was lost. To make T_{i+1} more specific, Marvin re-examines the predicates in M to determine which ones contain essential information.

Suppose we wish to teach Marvin what an arch is. Assume that, among other things, Marvin has already learned the concepts *any_shape* and *same_shape*, shown below.

any_shape(X) ← brick(X). (1)
any_shape(X) ← wedge(X).

same_shape(X, Y) ← brick(X) & brick(Y). (2)
same_shape(X, Y) ← wedge(X) & wedge(Y).

The following set of predicates describes the example of an arch shown by the trainer:

a is_part_of x.
b is_part_of x.
c is_part_of x.
a is_on b.
a is_on c.
b left_of c.
b does_not_touch c.
lying(a).
wedge(a).
standing(b).
brick(b). (3)
standing(c).
brick(c). (4)

This forms the initial trial, T_0. The final definition of *arch* should include the specification that the two columns *b* and *c* may have any shape as long as they are both the same. Now let us begin generating trial descriptions of *arch* by performing replacements.

1. Since *b* is a brick, it is an instance of *any_shape*, thus one possible trial, T_1, replaces *brick(b)* above with *any_shape(b)* using clause (1).
2. This generalization would allow *b* to be a wedge while *c* remains a brick. Note, that the generalization is not totally incorrect since *b* may be a wedge, however, additional information must be added to qualify the generalization. This additional information is obtained by searching for another replacement which involves at least one of the predicates removed from the original trial.
3. Predicates (3) and (4) match the right hand side of clause (2) and predicate (3) also matches the right hand side of clause (1). A new trial, T_2, may be formed by adding *same_shape(b, c)* to T_1. This specialization creates a consistent trial.

Note that since clause (2) completely subsumes clause (1) predicate, *any_shape(b)* can be ignored in the new trial. In general, this would not be the case. We now define specialization as follows:

Definition 6: Suppose T is a trial concept. Let

$$T^1 = T - M^1 \cup \sigma^1\{S^1\}$$

be a generalization of T obtained by replacing a subset of predicates, i.e. M^1, with a reference to the concept, S^1. Let

$$T^2 = T - M^2 \cup \sigma^2\{S^2\}$$

be another generalization of T, such that

$$M^1 \cap M^2 \neq \varnothing$$

then $T^1 \cup T^2$ is *more specific* than either T^1 or T^2.

The purpose of the specialization operation defined above is to conjoin two generalizations of T. That is, specialization will force the program to search for conjunctions of generalizations. Without specialization, Marvin could only discover generalizations which consist only of single replacements. Usually, Marvin will attempt one generalization, say, T^1 and if it is found to be inconsistent, then the program will look for T^2 to make the trial more specific.

5. CONSTRUCTING EXAMPLES TO TEST HYPOTHESES

To find out if a trial, T_{i+1}, is consistent or not, Marvin shows the trainer an instance of T_{i+1}. If the program can create an object which does not belong to the target, but does belong to T_{i+1} then T_{i+1} is inconsistent. However, since the program has not yet learned the description of the target concept, how can we guarantee that an object not in the target is shown to the trainer when the trial in inconsistent?

The set, All_Elaborations(T_0), contains all the predicates which can be inferred from T_0. If the target concept can be learned at all, then its description must be a subset of All_Elaborations(T_0). Any object which fails to satisfy any of the predicates in this set cannot belong to the target. To test the trial, T_{i+1}, Marvin constructs an object which does not satisfy any of the predicates in All_Elaborations(T_0), with the exception that T_{i+1} and anything that it implies, must be satisfied. This guarantees that, if at all possible, an object will be constructed so that it belongs to the trial concept but not to the target. If the trial is consistent, then the object must belong to the target.

Definition 7: Let T_{i+1} be a generalization of T_i. Any object which satisfies T_{i+1} but negates each element of

$$\text{All\_Elaborations}(T_i) - \text{All\_Elaborations}(T_{i+1})$$

is called a *crucial* object. If T_i is consistent with the target and T_{i+1} is not then no crucial object satisfies the target.

Recall that in the example in section 3, the description, T_0, of a column consisting of two blocks was generalized to the description, T_1, of a block resting on another column. By definition 7, an object which can be used to test the generalization may be constructed by the following procedure:

1. Find all elaborations of T_0:

```
brick(a).          (1)
standing(a).       (2)
a is_on b.         (3)
brick(b).          (4)
standing(b).       (5)
b is_on c.         (6)
ground(c).         (7)
column(b).         (8)
column(a).         (9)
```

2. Find the set of all predicates implied by T_1, i.e. all elaborations of T_1.

brick(a).	(1)
standing(a).	(2)
a is_on b.	(3)
column(b).	(8)
column(a).	(9)

3. Find All_Elaborations(T_0) − All_Elaborations(T_1)

brick(b).	(4)
standing(b).	(5)
b is_on c.	(6)
ground(c).	(7)

4. Construct an object which satisfi es T_1, but does not satisfy any of the four predicates 4, 5, 6 or 7. The resulting object will consist of a brick on top of at least two more bricks. Negation of predicates 4, 5, 6 and 7 prevents the bottom of the column from begin a single brick standing on the ground.

We have now developed methods for generalizing and specializing concept descriptions and we also have a way of testing whether those descriptions are consistent or not. In the following section, we bring all of these elements together to create the complete Marvin.

6. AN OVERVIEW OF THE LEARNING ALGORITHM

Let us now summarize Marvin's learning algorithm:

To learn a new clause:

1. The trainer gives the program an example of the concept.
2. Find the list of all replacement operations obtainable from the primary predicates.
3. Use these operations to generalize the initial trial.
4. Store the resulting concept in memory.

To fi nd the list of replacement operations:

1. Let C initially be the set of all facts describing the training example.
2. Find the set, I, of all concepts implied by C.
3. Append I to C.
4. Repeat this procedure until no more implied concepts can be found. The result is All_Elaborations(C). The list of replacement operations consists of all the clauses used to fi nd the implied concepts in All_Elaborations(P). See defi nitions 1 and 2.

To generalize a trial T_i:

1. Choose the fi rst concept in the list of replacement operations.
2. Perform the replacement to obtain T_{i+1}. (See defi nition 3).
3. Try to *qualify* T_{i+1}.
4. If T_{i+1} cannot be qualifi ed, abandon this replacement.
5. Repeat this procedure with the next replacement operation in list.

To qualify a trial, T_{i+1}:

1. Construct a crucial object which satisfi es T_{i+1} but does not satisfy

   All_Elaborations(T_i) − All_Elaborations(T_{i+1})

   (See defi nition 7).
2. Ask trainer if object satisfi es target. If it does then T_{i+1} has been qualifi ed.
3. If not then specialize T_{i+1} to a new trial T_{i+2}

4. Try to qualify T_{i+2}.

To specialize a trial, T_{i+1}:

1. Search list of replacement operations for a clause, $S \leftarrow M$, such that $M \cap M^{i+1} \neq \emptyset$, M^{i+1} comes from the replacement which produced T_{i+1}.
2. Perform replacement on T_{i+1} to obtain the new trial, T_{i+2}. (See definition 6).

To construct an example from a trial, T_{i+1}:

1. To construct an example from P & Q: Construct P, Construct Q.
2. To construct an example from an atomic predicate P when there is a set of clauses $\{P \leftarrow B_i\}$ in memory:
   - 2.1 Select a B_i such that

     $$B \cap (\text{All\_Elaborations}(T_i) - \text{All\_Elaborations}(T_{i+1})) = \emptyset$$

   - 2.2 Construct example using the selected B_i.
3. To construct an example from an atomic predicate P when there is no clause $P \leftarrow B$, add P to set of predicates representing example.

7. A TYPICAL LEARNING TASK

In this section we will discuss the steps involved in learning the description of an arch. This time, the description will be slightly more involved than the first version we discussed in section 4. Marvin will learn that an arch consists of an object of any shape lying on top of two columns of equal height. The columns are adjacent to each other, but must not touch.

Initially the program's memory is empty. Since concepts stored in the memory are essential to performing generalizations, any object shown to Marvin which it cannot recognize, is simply remembered, without any attempt at generalizations. So if the trainer states that *x* is a brick and *x* is an example of the concept *any_shape*, Marvin will remember this fact.

When Marvin begins operation, it often starts by learning basic concepts by rote. *any_shape* and *any_orientation* will be learned in this way.

```
any_shape(X) ← brick(X).
any_shape(X) ← wedge(X).

any_orientation(X) ← standing(X).
any_orientation(X) ← lying(X).
```

These concepts may be written out to file at the end of one learning session and reloaded at a later time so that they need not be relearned.

The description of *arch* will contain references to two quite complex concepts, namely, *column* and *same_height*. A concept, such as *arch*, cannot be learned unless the other concepts to which it refers are already in the memory. In an analogy to describing something in English: one cannot adequately describe a table unless one knows about words such as ‘leg’ or ‘flat’. After teaching Marvin about shape and orientation, the trainer presents examples of columns.

We will leave a discussion of the details of the learning session until the program starts learning about arches. It is important to note that the description of column is recursive, that is, it refers to itself. Because of this, the trainer must take care in presenting example of columns. The first example must teach Marvin about the non-recursive disjunct of the description. That is the first clause shown below. Having learned this, the program is then able to recognize the recursive nature of more complex examples.

```
column(X) ←
            brick(X) &
            standing(X) &
            X is_on Y &
            ground(Y).

column(X) ←
            brick(X) &
            standing(X) &
            X is_on Y &
            column(Y).
```

In this learning session, we will allow columns to consists only of bricks. The same consideration for recursive concepts applies to learning *same_height*. The heights of two columns may be compared by scanning down both columns to see if the ground is reached at the same time.

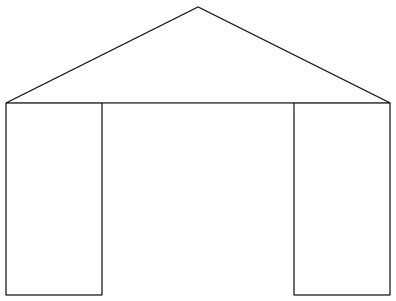
```
same_height(X, Y) ←
            ground(X) &
            ground(Y).

same_height(X1, X2) ←
            brick(X1) &
            standing(X1) &
            brick(X2) &
            standing(X2) &
            X1 is_on Y1 &
            X2 is_on Y2 &
            same_height(Y1, Y2).
```

To complete Marvin's background knowledge so that it can learn about arches, it must also learn the following concepts:

```
X adjacent_to Y ← X left_of Y.
X adjacent_to Y ← Y left_of X.

X may_touch Y ← X touches Y.
X may_touch Y ← X does_not_touch Y.

X may_be_on Y ← X is_on Y.
X may_be_on Y ← X is_not_on Y.
```

Assuming that all the above concepts are now present in Marvin's memory, we can begin to teach it what an arch is. The program is presented with the description of an instance of *arch*, shown below.

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
B is_on D
C is_on E
ground(D)
ground(E)
B left_of C
B does_not_touch C
lying(A)
wedge(A)
standing(B)
brick(B)
standing(C)
brick(C)



Having seen an example of the concept, the pattern matcher determines which objects in the example belong to concepts in the memory. This recognition process results in the generation of all possible replacements. These are shown below with the predicates to be replaced on the right hand side of the arrow and the predicate replacing them on the left.

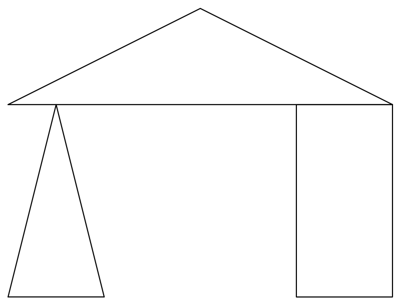
```
1 :     any_shape(B) ← brick(B)
2 :     any_shape(C) ← brick(C)
3 :     any_shape(A) ← wedge(A)

4 :     any_orientation(B) ← standing(B)
5 :     any_orientation(C) ← standing(C)
6 :     any_orientation(A) ← lying(A)

7 :     column(B) ← brick(B) & standing(B) & B is_on D & ground(D)
8 :     column(C) ← brick(C) & standing(C) & C is_on E & ground(E)

9 :     same_height(D, E) ← ground(D) & ground(E)
10 :    same_height(B, C) ←
            brick(B) &
            standing(B) &
            brick(C) &
            standing(C) &
            B is_on D &
            C is_on E &
            same_height(D, E)

11 :    B adjacent_to C ← B left_of C

12 :    B may_touch C ← B does_not_touch C

13 :    A may_be_on B ← A is_on B
14 :    A may_be_on C ← A is_on C
15 :    B may_be_on D ← B is_on D
16 :    C may_be_on E ← C is_on E
```

The program takes each replacement in turn and attempts to create a consistent trial. The first replacement attempts to generalize the shape of B. This results in the trial:

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
B is_on D
C is_on E
ground(D)
ground(E)
B left_of C
B does_not_touch C
lying(A)
wedge(A)
standing(B)
standing(C)
brick(C)
any_shape(B)

To test this, the program asks if the following example is an instance of arch.

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
B is_on D
C is_on E
ground(D)
ground(E)
B left_of C
B does_not_touch C
lying(A)
wedge(A)
standing(B)
standing(C)
brick(C)
wedge(B)



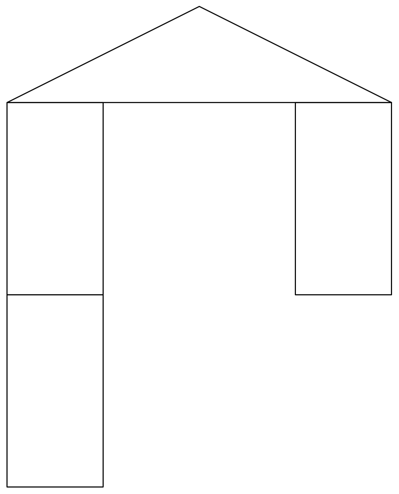
A new object has been created which differs from the original example by having the shape of B changed to a wedge. The trainer responds *no*, indicating that this is an inconsistent trial.

Since the trial is inconsistent, another trial which is more specific must be constructed. This is done by finding another replacement which involves the predicate *brick(B)*. In the order that the replacement operations were generated, replacement 7 is selected. This results in the new trial:

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
B is_on D
C is_on E
ground(D)
ground(E)
B left_of C
B does_not_touch C
lying(A)
wedge(A)
standing(C)
brick(C)
column(B)

This generates the training example:

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
C is_on E
ground(E)
B left_of C
B does_not_touch C
lying(A)
wedge(A)
standing(C)
brick(C)
brick(B)
standing(B)
B is_on _1
brick(_1)
standing(_1)
_1 is_on _2
ground(_2)



Since replacement 7 generalizes B to any column, Marvin constructs a new example in which B is a column, but not the same one as before. B is now a two brick column. _1 and _2 are names generated by the program to stand for new objects. Since the left and right columns of the arch must be the same height, the trainer responds *no* again. Thus, the trial must be specialized even further. Since the right hand sides of replacements 7 and 10 have a non-empty intersection, replacement 10 is selected to make the trial more specifi c. This new trial is:

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
B left_of C
B does_not_touch C
lying(A)
wedge(A)
column(B)
same_height(B, C)

The program now constructs a new example in which both columns are two bricks high. The trainer indicates that this trial is consistent, so Marvin can now resume generalizing.

We will not show in detail the remaining questions which are asked. However, Marvin will go on to test the shape of A. This can be generalized to any shape. However, A cannot assume any orientation. It must be lying down. Marvin will discover that B is adjacent to C, but must not touch C. Finally it will conclude that the description of an arch is:

A part_of X
B part_of X
C part_of X
A is_on B
A is_on C
column(B)
same_height(B, C)
column(C)
any_shape(A)
lying(A)
B adjacent_to C
B does_not_touch C

In all, 8 questions are asked before Marvin is fi nished. However, it should be noted that in general, the number of questions asked depends on the number of concepts in memory which match parts of the example. As the size of the memory increases, so will the number of questions.

8. CONCEPTS THAT MARVIN HAS LEARNED

In this section we describe some of the concept which Marvin has been able to learn.

Since concepts are represented as Horn clauses in fi rst order logic, they can be viewed as logic programs. Thus Marvin is able to perform as an automatic programming system as well as a learning system. The class of programs that can be synthesized is limited by the way in which variables are created. When the trainer presents an example to the program, each object in the example is given a name. When the description is generalized and turned into a concept, the object names become variables. Marvin has no ability to invent existentially quantifi ed variables other than those derived from the example. The program also cannot deal with universal quantifi cation.

Some of the concepts that have been learned are:

- List manipulation.
  We can represent lists by a recursive concept much like *column*. A column is an object with a top which is a brick and a bottom which is another column. Similarly, a list is an object which has a head which can be of some specifi ed type and a tail which is another list. Marvin can be taught to append lists by showing it examples consisting of input/output pairs for the concept *append*. using a Prolog-like notation, one example may be: *append([], [1], [1])*. Another may be *append([1], [2], [1, 2])*. These two examples, shown in that order provide Marvin with enough information to synthesize both clauses of the recursive concept. Once it knows how to

append lists, Marvin can then be taught to reverse them, again by showing examples of input/output pairs.

- Arithmetic on numbers represented as strings of binary digits.
  A string of bits can be represented as a list of objects which can be either 0 or 1. Marvin can learn to compare numbers by learning the concept, *less*. As examples, the program would be shown pairs of numbers, one less than the other. Input/output pairs can also be presented to the program in order to teach it about addition and other arithmetic operations.
- Sorting.
  Once the program knows how to do arithmetic and manipulate lists, it can learn how to sort lists of numbers. It is possible to learn a simple insertion sort. However, because of the program's inability to invent new existentially quantified variables, learning more efficient sorting algorithms, such as quick-sort, is beyond its capabilities.
- Grammar Rules.
  Marvin is capable of learning to recognize sentences in context free grammars. For example, the program has been taught to recognize a very limited subset of English. A string of words can be represented as a list. Concepts such as *verb*, *noun*, *etc* are taught first, so that the program can identify parts of speech. After that, Marvin can learn to recognize noun phrases and verb phrases and so on until the representation of a complete sentence has been acquired. One of the interesting problems encountered when teaching such grammatical concepts to Marvin is that concepts such as *noun phrase* and *verb phrase* can refer to each other. Since the concepts are disjunctive, the concepts can be taught in steps. One concept is partially learned, then part of the second and then the description of the first concept can be completed (Sammut, 1981). discusses this problem more fully.
- A more difficult language recognition problem which Marvin has learned is that posed by Hayes-Roth and McDermott (1977). The task was to learn the rules to transform a sentence in the active form, such as 'The little man sang a lovely song' to the passive form 'A lovely song was sung by the little man'.
- Geometrical concepts as described by Michalski (1980) have also been learned.

Marvin has proved to be capable of learning many concepts in a variety of domains. In the next section we discuss ways in which the program may be improved.

9. DETECTING ERRORS IN CONCEPT DESCRIPTIONS

We have already seen that the trainer has the responsibility for teaching Marvin all the necessary background knowledge required to describe any new concept that is to be learned. This means that he must carefully choose the training examples and the order in which they are presented. To see what happens concepts are not taught in the best order let us return to the description of columns.

We now wish to make the definition slightly more general by allowing cylinders as well as bricks to make up a column, but not wedges. Suppose that any shape in our blocks world is defined as:

$$\begin{aligned}&\text{any\_shape}(X) \leftarrow \text{brick}(X).\\&\text{any\_shape}(X) \leftarrow \text{cylinder}(X).\\&\text{any\_shape}(X) \leftarrow \text{wedge}(X). \qquad \text{(R1)}\end{aligned}$$

If Marvin is shown an example of a column, without first learning to distinguish wedges from bricks and cylinders, it will construct the wrong description for columns. Remember that to generalize a description, the program replaces predicates which match the right hand side of a clause with the corresponding left hand side. If the example of the column contained bricks, then Marvin would recognize the brick as belonging to *any_shape* and attempt to generalize. To test the generalization it constructs another instance of the concept. In this case, the program could construct a column with cylinders. This would be an instance of the target concept, even though the description it has created is too general.

If Marvin had first learned a concept such as *flat_top*, defined as:

```
flat_top(X) ← brick(X).
flat_top(X) ← cylinder(X).
```

then the correct description of column could be learned. The problem we discuss here is: how can we determine that a concept description has been learned incorrectly?

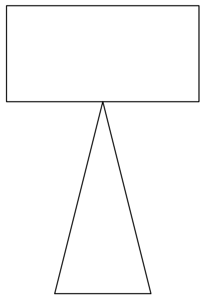
Let us suppose that we are carrying on an extended dialogue with Marvin and at some point, we see that the program has incorrectly stated that an object is a *tee* even though it is not. Marvin's description of *tee* must be incorrect. However, suppose that a *tee* is now defined as a brick lying on a column, is the bug the description of *tee* itself or in one of the concepts that it refers to, such as *column*? We can 'debug' the concept by using a method similar to Shapiro's backtrace (Shaprio, 1981). To illustrate this method, let us continue with the tees and columns. Assume that the concept description *any_shape*, above, has been learned as well as the following:

```
column(X) ←     (R2)
                ground(Y).

column(X) ←     (R3)
                any_shape(X) &
                standing(X) &
                X is_on_Y &
                column(Y).

tee(T) ←        (R4)
                X is_part_of T &
                Y is_part_of T &
                brick(X) &
                lying(X) &
                X is_on Y &
                column(Y).
```

The first clause in this version of *column* has been simplified to make the explanation easier.

Now suppose that the following object, X, has been incorrectly recognized as a tee:

```
1:  A part_of X
3:  B part_of X
4:  A is_on B
5:  B is_on C
6:  brick(A)
7:  lying(A)
8:  wedge(B)
9:  standing(B)
10: ground(C)
```



We will debug the concept descriptions by tracing through the steps which Marvin took to incorrectly recognize this object as an tee. Let us first list those steps.

1. Predicate 8 matches R1, the third clause of *any_shape*. Performing a replacement, this leaves us with:

1: A part_of X
3: B part_of X
4: A is_on B
5: B is_on C
6: brick(A)
7: lying(A)
9: standing(B)
10: ground(C)
11: any_shape(B)

2. Predicate 10 matches R2, the first clause of *column*. Replacing predicate 10 with the corresponding predicate in the left hand side of that clause:

1: A part_of X
3: B part_of X
4: A is_on B
5: B is_on C
6: brick(A)
7: lying(A)
9: standing(B)
11: any_shape(B)
12: column(C)

3. Predicates 5, 9, 11 and 12 match R3, the second clause of *column*. The replacement leaves:

1: A part_of X
3: B part_of X
4: A is_on B
6: brick(A)
7: lying(A)
13: column(B)

4. All of the remaining predicates match R4, the description of *tee*. Leaving:

14: tee(X)

Thus, the object has been incorrectly recognized as a tee.

When the trainer has told Marvin that this derivation is incorrect, the program retraces its steps, asking the trainer to confirm that each replacement should have taken place. The following questions correspond to each of the steps above.

1. The first replacement is checked for correctness by asking the trainer: “Is B an example of *any_shape*?” If the answer is “yes”, then the program continues. In this case, a wedge is an example of any_shape.
2. Checking step 2, Marvin asks: “Is C an example of a column?”. Again the answer is “yes”.
3. Finally, checking the third replacement, Marvin asks: “Is B an example of *column*?” This time, the trainer answer “no”. Thus the offending clause, R3, has been identified.

Note that the description of *any_shape* is not incorrect. It is the definition of column which is too general. The next step in debugging the concept, is to identify the predicate within the clause which is incorrect.

Since the object, B, was incorrectly classified as a column, Marvin asks the trainer to change the description of B so that it becomes a column. The trainer should make the least number of changes possible. Assume that the description is changed to a brick standing on the ground. Now the program may consider why the new object, X, is a column, while B is not. The two descriptions are:

wedge(B)	brick(X)
standing(B)	standing(X)
B is_on C	X is_on Y
ground(C)	ground(Y)

The properties which the two objects have in common can be ignored, leaving only the shape. The replacement, R1, caused *wedge(B)* to be generalized to *any_shape(B).* Since this replacement resulted in a misclassification, it should not be used in the clause R3. Marvin can now tell the trainer that it has located the bug, and ask him to teach it about columns again, this time not using *any_shape*. More specifically, the program can ask, ''Is there a distinction between *brick(B)* and *wedge(B)* that you haven't explained yet?''

10. CONCLUSION

When we characterize learning as a search process we assume the following:

1. There is a language which is used to describe concepts.
2. A state in the search space is a collection of descriptions written in this language. Given a set of examples, a goal state is a set of descriptions such that each positive example satisfies some description and no negative example satisfies any description.
3. There is a set of generalization rules for transforming states into new states.
4. A strategy for using the set of examples as a guide to choose a useful sequence of transformations which leads one to the goal state.

The problem of learning is to search through the space with a sequence of generalizations until a path to the goal is found. The search space is determined by the language used to describe concepts. Learning systems such as LEX (Mitchell, 1982), INDUCE (Michalski, 1983) and MIS (Shapiro, 1981), have languages which are fixed at the start of the learning task. In programs such as Marvin (Sammut, 1981) and its predecessor, CONFUCIUS (Cohen, 1978) we have investigated learning systems in which the description language changes depending on the state, i.e. on the set of descriptions learned. Put another way, while in previous work the transformations operated on single sentences in the state, in our work the rules take the whole state into consideration. This makes for a significant difference in the efficiency of the descriptions learned and, under many circumstances, on the efficiency of the search process itself.

Marvin is able to easily extend its language because it uses sets of Horn clauses as descriptions, just as MIS does. The program is designed to emulate teacher/student learning. Unlike LEX, it is not given a hierarchy of concepts initially. The relationships between concepts are built up as the trainer presents new concepts. As in human teacher/student interactions, it is expected that simple concepts will be taught before complex ones. For example, addition is taught as a concept before multiplication is taught, since the description of multiplication may use addition. The collection of concepts as a whole forms a model of the world in which the program exists.

Marvin does not passively accept data from the trainer. It 'performs experiments' to test its hypotheses by constructing its own training examples. This provides the trainer with feedback which indicates how well Marvin has 'understood' the concept. With improvements which will allow it to detect and correct misconceptions, a learning system, such as Marvin, could find applications in the interactive acquisition of knowledge for expert systems. However, its main contributions have been in demonstrating a program whose language grows in descriptive power as it learns new concepts and in being able to use those concepts as procedures to perform actions, that is, to build objects.

ACKNOWLEDGEMENTS

This paper was written while the first author was with the Department of Computer Science, University of Illinois at Urbana-Champaign. We thank R.S. Michalski for his help during the stay at Illinois. We also thank Ian Hayes for his helpful comments on a draft of this paper.

References

Banerji, R. B., “A Language for the Description of Concepts,” *General Systems,* 9 (1964).

Cohen, B. L., *A Theory of Structural Concept Formation and Pattern Recognition,* Ph.D. Thesis, Dept. of Computer Science, University of N.S.W (1978).

Hayes-Roth, F. and McDermott, J., “An Interference Matching Technique for Inducing Abstractions,” *Communications of the ACM,* 21, pp. 401-411 (1978).

Michalski, R. S., “Pattern Recognition as Rule-Guided Inference,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 2, 4, pp. 349-361 (1980).

Michalski, R.S., “A Theory and Methodology of Inductive Learning” in *Machine Learning: An Artificial Intelligence Approach,* ed. Michalski, Carbonell, Mitchell, Tioga, Palo Alto (1983).

Robinson, J. A., “A Machine Oriented Logic Based on the Resolution Principle,” *Journal of the ACM,* 12, 1, pp. 23-41 (1965).

Sammut, C. A., *Learning Concepts by Performing Experiments,* Ph.D. thesis, Department of Computer Science, University of New South Wales (1981).

Shapiro, Ehud Y., “Inductive Inference of Theories From Facts,” 192, Yale University (1981).

Utgoff, Paul E. and Mitchell, Tom M., “Acquisition of Appropriate Bias for Inductive Concept Learning” in *Proceedings of the National Conference on Artificial Intelligence,* pp. 414-417, Pittsburgh (1982).