Machine Learning, 15, 299–319 (1994)
© 1994 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Associative Reinforcement Learning: A Generate and Test Algorithm

LESLIE PACK KAELBLING LPK@CS.BROWN.EDU
Computer Science Department
Box 1910
Brown University
Providence, RI 02912-1910 USA

Editor: R. Sutton

Abstract. An agent that must learn to act in the world by trial and error faces the *reinforcement learning* problem, which is quite different from standard concept learning. Although good algorithms exist for this problem in the general case, they are often quite inefficient and do not exhibit generalization. One strategy is to find restricted classes of action policies that can be learned more efficiently. This paper pursues that strategy by developing an algorithm that performans an on-line search through the space of action mappings, expressed as Boolean formulae. The algorithm is compared with existing methods in empirical trials and is shown to have very good performance.

Keywords: reinforcement learning, generalization, generate-and-test

1. Reinforcement learning

Reinforcement learning is the problem of learning a mapping from situations to actions by trial and error. An agent finds itself in a situation, then generates an action. It receives a scalar *reinforcement value*, which indicates, roughly, how good that action was in that situation. The agent must learn, over time, to take actions that tend to result in high reinforcement values. An important issue in reinforcement learning is the problem of exploration: especially when the results of actions are stochastic, it may be necessary for the agent to take actions that do not appear to be good in order to gain more information about their effects.

A companion paper (Kaelbling, 1994) explores the background of reinforcement learning and presents a number of algorithms for solving a reinforcement-learning problem. These include the IE algorithm, which gathers statistics about the results of every action in every situation and chooses actions based on a tradeoff between exploration and exploitation. This algorithm, because it is based on a table, is inefficient in time and space and exhibits no useful generalization. Artificial neural network algorithms, based on a linear associator (LARC) and on error back-propagation (BPRC) are also presented. Finally, two novel algorithms, (LARCKDNF and IEKDNF), which learn action mappings expressible in k-DNF are presented. The k-DNF based algorithms work well on simple problems with large spaces.

For simplicity in presentation, we begin by discussing the reinforcement-learning problem in which the following assumptions hold:

- the agent has only two possible actions
- the reinforcement signal at time $t+1$ reflects only the success of the action taken at time t
- reinforcement received for performing a particular action in a particular situation is 1 with some probability p and 0 with probability $1-p$, and each trial is independent
- the expected reinforcement value of doing a particular action in a particular input situation stays constant for the entire run of the learning algorithm

The companion paper discusses general techniques that can be used to extend algorithms making these assumptions to situations in which each of the assumptions is relaxed.

An agent's strategy can be seen as a mapping from its input space to its action space. In the case we are considering initially, the output space has only two elements, so the agent's action map can be thought of as a Boolean function.

2. A generate-and-test algorithm

GTRL is a highly parameterized generate-and-test algorithm for learning Boolean functions from reinforcement. With some parameter settings it is highly time- and space-efficient, but can learn only a restricted class of functions; with other parameter settings it can learn arbitrarily complex functions, but at a cost in time and space.

The generate-and-test reinforcement-learning algorithm, GTRL, performs a bounded, real-time beam-search in the space of Boolean formulae, searching for a formula that represents an action function that exhibits high performance in the environment. This algorithm satisfies the requirement of strict incrementality by performing its search incrementally, while using the best available solution to generate actions for the inputs with which it is presented.

The algorithm has, at any time, a set of hypotheses under consideration. A hypothesis has as its main component a Boolean formula whose atoms are input bits or their negations. Negations can occur only at the lowest level in the formulae.[1] Each formula represents a potential action-map for the behavior, generating action 1 whenever the current input satisfies the formula and action 0 when it does not. The GTRL algorithm generates new hypotheses by combining the formulae of existing hypotheses using syntactic conjunction and disjunction operators.[2] This generation of new hypotheses represents a search through Boolean-formula space; statistics related to the performance of the hypotheses in the domain are used to guide the search, choosing appropriate formulae to be combined.

This search is quite constrained, however. There is a limit on the number of hypotheses with formulae at each level of Boolean complexity (depth of nesting of Boolean operators), making the process very much like a beam search in which the entire beam is retained in memory. As time passes, old elements may be deleted from and new elements added to the beam, as long as the size is kept constant. This guarantees that the algorithm will operate in constant time per input instance and that the space requirement will not grow without bound over time. [3]

This search method is inspired by Schlimmer's STAGGER system (Schlimmer, 1987; Schlimmer & Granger, 1986), which learns Boolean functions from input-output pairs. STAGGER makes use of a number of techniques, including a Bayesian weight-updating component, that are inappropriate for the reinforcement-learning problem. In addition, it is not strictly limited in time- or space-complexity. The GTRL algorithm exploits STAGGER's idea of performing incremental search in the space of Boolean formulae, using statistical estimates of the "necessity" and "sufficiency" (these notions are made concrete in the following discussion) to guide the search.

It is possible to view the GTRL algorithm as being biased in favor of hypotheses representable as low-depth Boolean functions. One justification of this bias is the *Occam's razor argument* and similar complexity arguments that say that simple hypotheses are to be preferred over complex ones, given similar past performance. Saxena (1991) provides an excellent overview of these arguments.

The presentation of the GTRL algorithm is independent of any distributional assumptions about the reinforcement values generated by the environment; the process of tailoring the algorithm to work for particular kinds of reinforcement is described in section 4.

3. High-level description

As with the algorithms in the companion paper, we describe the GTRL algorithm in a standard form consisting of three components: s_0 is the initial internal state of the algorithm; $u(s, i, a, r)$ is the update function, which takes the state of the algorithm s, the last input i, the last action a, and the reinforcement value received r, and generates a new algorithm state; and $e(s, i)$ is the evaluation function, which takes an algorithm state s and an input i, and generates an action. It is shown in figure 1. The internal state of the GTRL algorithm consists of a set of hypotheses organized into levels. Along with a Boolean formula, each hypothesis contains a set of statistics that reflect different aspects of the performance of the formula as an action map in the domain. Each level contains hypotheses whose formulae are of a given Boolean complexity. Figure 2 shows an example GTRL internal state. Level 0 consists of hypotheses whose formulae are individual atoms corresponding to the input bits and to their negations, as well as the hypotheses whose formulae are the logical constants *true* and *false*.[4] Hypotheses at level 1 have formulae that are conjunctions and disjunctions of the formulae of the hypotheses at level 0. In general, the hypotheses at level n have formulae that consist of conjunctions or disjunctions of two formulae: one from level $n-1$ and one from any level, from 0 to $n-1$. The hypotheses at each level are divided into working and candidate hypotheses; the reasons for this distinction will be made clear during the detailed explanation of the algorithm.

The update function of the GTRL algorithm consists of two phases: first, updating the statistics of the individual hypotheses and, second, adding and deleting hypotheses.

Algorithm GTRL

```
$s_0 =$             array[0..$L$] of
                        record
                           working-hypoths: array[0..H] of hypoth
                           candidate-hypoths: array[0..C] of hypoth
                        end
$u(s, i, a, r) =$   update-hypotheses $(s, i, a, r)$
                    for each level in s do begin
                       add-hypotheses (level, s)
                       promote-hypotheses (level)
                       prune-hypotheses (level)
                    end
$e(s, i) =$         h := best-predictor (s)
                    if satisfies (i, h) then
                       return 1
                    else return 0
```

Figure 1. High-level description of the GTRL algorithm.

Level 2	$(a \vee b) \wedge (\neg b \vee \neg c)$	$(b \vee c) \wedge \neg a$	$(c \wedge \neg a) \vee (a \wedge \neg b)$					
Level 1	$a \vee b$	$b \vee c$	$c \wedge \neg a$	$\neg b \vee \neg c$	$a \wedge \neg b$			
Level 0	a	$\neg a$	b	$\neg b$	c	$\neg c$	t	f

Figure 2. Example GTRL internal state.

The evaluation function also works in two phases. The first step is to find the working hypothesis at any level that has had the best performance at choosing actions. If the chosen working hypothesis is satisfied by the input to be evaluated, action 1 is generated; if it is not satisfied, action 0 is generated.

The following sections examine these processes in greater detail.

4. Statistics

Associated with each working and candidate hypothesis is a set of statistics; these statistics are used to choose working hypotheses for generating actions and for combination into new candidate hypotheses at higher levels. The algorithms for updating the statistical information and computing statistical quantities are modularly separated from the rest of the GTRL algorithm. The choice of statistical module will depend on the distribution of reinforcement values received from the environment. A more detailed description of this work (Kaelbling, 1993) provides definitions of statistics modules for cases in which the reinforcement values are binomially or normally distributed; in addition, it contains a non-parametric statistics module for use when there is no known model of the distribution of reinforcement values. A statistics module supplies the following functions:

$age(h)$: The number of times the behavior, as a whole, has taken the action that would have been taken had hypothesis h been used to generate the action.

$er(h)$: A point estimate of the expected reinforcement received given that the action taken by the behavior agrees with the one that would have been generated had hypothesis h been used to generate the action.

$er\text{-}ub(h)$: The upper bound of a $100(1-\alpha)\%$ confidence interval estimate of the quantity estimated by $er(h)$.

$erp(h)$: A point estimate of the expected reinforcement received given that hypothesis h was used to generate the action that resulted in the reinforcement.

$erp\text{-}ub(h)$: The upper bound of a $100(1-\alpha)\%$ confidence interval estimate of the quantity estimated by $erp(h)$.

$N(h)$: A statistical measure of the probability that the expected reinforcement of executing action 0 when hypothesis h is not satisfied is greater than the expected reinforcement of execution action 1 when hypothesis h is not satisfied.

$S(h)$: A statistical measure of the probability that the expected reinforcement of executing action 1 when hypothesis h is satisfied is greater than the expected reinforcement of executing action 0 when hypothesis h is satisfied.

5. Evaluating inputs

Each time the evaluation function is called, the most predictive working hypothesis is chosen, by taking the one with the highest value of *pv*, defined as

$$pv(h) = \lfloor \kappa \;\; er(h) \rfloor + erp\text{-}ub(h) \;\; .$$

This definition has the effect of sorting first on the value of *er*, then breaking ties based on the value of *erp-ub*. The constant multiplier κ can be adjusted to make this criterion more or less sensitive to low-order digits of the value of $er(h)$.[5]

What makes this an appropriate criterion for choosing the hypothesis with the best performance? The quantity that most clearly represents the predictive value of the hypothesis is $erp(h)$, which is a point estimate of the expected reinforcement given that actions are chosen according to hypothesis h. Unfortunately, this quantity only has a useful value after the hypothesis has been chosen to generate actions a number of times. Thus, as in the interval estimation algorithm, we make use of $erp\text{-}ub(h)$, the upper bound of a confidence interval estimate of the expected reinforcement of acting according to hypothesis h.

So, why not simply choose the working hypothesis with the highest value of *erp-ub*(h), similar to the execution of the interval estimation algorithm? The reason lies in the fact that in the GTRL algorithm, new hypotheses are continually being created. If it always chooses hypotheses with high values of *erp-ub*(h), it will be in danger of spending nearly all of its time choosing hypotheses because little is known about them, rather than because they are known to perform well. The value of *er*(h) serves as a filter on hypotheses that will prevent most of this fruitless exploration. The quantity *er*(h) is not a completely accurate estimator of *erp*(h), because the distribution of instances over which it is defined may be different than the distribution of input instances presented to the entire algorithm,[6] but it serves as a useful approximation. We can use *er*(h) rather than *er-ub*(h) because the statistics used to compute *er*(h) get updated even when h is not used to generate actions, so that the statistic becomes valid eventually without having to do any special work. Thus, hypotheses that look good on the basis of the value of *er*(h) tend to get chosen to act; as they do, the value of *erp-ub*(h) begins to reflect their true predictive value. This method still spends some time acting according to untested hypotheses, but that is necessary in order to allow the algorithm to discover the correct hypothesis initially and to adjust to a dynamically changing world. The amount of exploration that actually takes place can be controlled by changing the rate at which new hypotheses are generated, as will be discussed in section 8.

Once a working hypothesis is chosen, it is used to evaluate the input instance. An input vector i satisfies hypothesis h if h's formula evaluates to *true* under the valuation of the atoms supplied by input i. If the input instance satisfies the chosen hypothesis, action 1 is generated; otherwise, action 0 is generated.

6. Managing hypotheses

The process by which hypotheses are managed in the GTRL algorithm can be divided into three parts: adding, promoting, and pruning. On each call to the update function, the statistics of all working and candidate hypotheses are updated. Then, if it is time to do so, a new hypothesis may be constructed and added to the candidate list of some level. Candidate hypotheses that satisfy the appropriate requirements are "promoted" to be working hypotheses. Finally, any level that has more working hypotheses than the constant number allotted to it will have its working hypothesis list pruned.

6.1. Adding hypotheses

Search in the GTRL algorithm is carried out by adding hypotheses. Each new hypothesis is a conjunction or disjunction of hypotheses from lower levels.[7] On each update cycle, a candidate hypothesis is added to a level if the level is not yet fully populated (the total number of working and candidate hypotheses is less than the maximum number of working hypotheses) or if it has been a certain length of time since a candidate hypothesis was last generated for this level and there is room for a new candidate.

If it is time to generate a new hypothesis, it is randomly decided whether to make a conjunctive or disjunctive hypothesis.[8] Once the combining operator is determined, operands must be chosen.

The following search heuristic is used to guide the selection of operands:

When making a conjunction, use operands that have a high value of necessity; when making a disjunction, use operands that have a high value of sufficiency.

The terms *necessity* and *sufficiency* have a standard logical interpretation: P is sufficient for Q if P implies Q; P is necessary for Q if $\neg P$ implies $\neg Q$ (that is, Q implies P). Schlimmer follows Duda, Hart, Gashnig, and Nilsson (1976; 1979), defining the logical sufficiency of evidence E for hypothesis H as

$$\mathrm{LS}(E,H) = \frac{\Pr(E \mid H)}{\Pr(E \mid \neg H)}$$

and the logical necessity of E for H as

$$\mathrm{LN}(E,H) = \frac{\Pr(\neg E \mid H)}{\Pr(\neg E \mid \neg H)} \ .$$

If E is truly logically sufficient for H, then E implies H, so $\Pr(E \mid \neg H) = 0$, making $\mathrm{LS}(E,H) = \infty$. If E and H are statistically independent, then $\mathrm{LS}(E,H) = 1$. Similarly, if E is logically necessary for H, then $\neg E$ implies $\neg H$, so $\Pr(\neg E \mid H) = 0$, making $\mathrm{LN}(E,H) = 0$. As before, if E and H are independent, $\mathrm{LN}(E,H) = 1$.

What makes functions like these useful for our purposes is that they encode the notions of "degree of implication" and "degree of implication by."[9] Let $h^*(i)$ be the optimal hypothesis, that is, the action map that has the highest expected reinforcement in the domain. We would like to use these same notions of necessity and sufficiency to guide our search, estimating the necessity and sufficiency of hypotheses in the GTRL algorithm state for h^*, the Boolean function that encodes the optimal action policy for the environment. But, because of the reinforcement-learning setting of our problem, we have no access to or direct information about h^*—the environment never tells the agent which action it *should* have taken.

Let us first consider an appropriate measure of sufficiency. By the definition of conditional probability, we can rewrite the definition of logical sufficiency as

$$\begin{aligned}\mathrm{LS}(E,H) &= \frac{\Pr(E \mid H)}{\Pr(E \mid \neg H)} \\ &= \frac{\Pr(E \wedge H)\Pr(\neg H)}{\Pr(E \wedge \neg H)\Pr(H)} \\ &= \frac{\Pr(H \mid E)\Pr(\neg H)}{\Pr(\neg H \mid E)\Pr(H)} \ .\end{aligned}$$

We are interested in the sufficiency of a particular hypothesis, h, for the optimal hypothesis, h^*, or $\mathrm{LS}(h, h^*)$, which is equal to

$$\frac{\Pr(h^* \mid h) \Pr(\neg h^*)}{\Pr(\neg h^* \mid h) \Pr(h^*)} .$$

It is easiest to consider the case of deterministic Boolean reinforcement first. In this case, $\Pr(h^* \mid h) = \Pr(r = 1 \mid a = 1 \wedge h)$, which is the same as the expected reinforcement of executing action 1 given that h is satisfied, or $er(1 \mid h)$. So, we can express logical sufficiency as

$$\mathrm{LS}(h, h^*) = \frac{er(1 \mid h)}{er(0 \mid h)} \cdot \frac{\Pr(\neg h^*)}{\Pr(h^*)} .$$

There are two further steps that we take to derive our heuristic measure of sufficiency. The first is to notice that the term $\Pr(\neg h^*)/\Pr(h^*)$ will occur in the sufficiency of every hypothesis h, and so may be eliminated without changing the ordering induced by the sufficiency function. The next step is to generalize this formulation to the case in which the world may be non-deterministic and and reinforcement non-Boolean. In such cases, the expected reinforcement values may be negative, making the ratio an inappropriate measure of their relative magnitudes. Instead, we define sufficiency as

$$S(h) = \Pr(er(1 \mid h) > er(0 \mid h)) .$$

This measure is strongly related to the difference of the two expected reinforcements, but is much more stable when estimates of the quantity are constructed on line. Any function that induces the same ordering on hypotheses may be used in place of S; in particular, if a statistical test such as Student's t is used, the raw t values may be used directly without translation back into the probability domain. Necessity can be analogously defined to be

$$N(h) = \Pr(er(0 \mid \neg h) > er(1 \mid \neg h)) .$$

Now we understand the definition and purpose of the necessity and sufficiency operators, but what makes them appropriate for use as search-control heuristics? In general, if we have a hypothesis that is highly sufficient, it can be best improved by making it highly necessary as well; this can be achieved by making the hypothesis more general by disjoining it with another sufficient hypothesis. Similarly, given a highly necessary hypothesis, we would like to make it more sufficient; we can achieve this through specialization by conjoining it with another necessary hypothesis. As a simple example, consider the case in which $h^* = a \vee b$. In this case, the hypothesis a is logically sufficient for h^*, so the heuristic will have us try to improve it by disjoining it with another sufficient hypothesis. If $h^* = a \wedge b$, the hypothesis a is logically necessary for h^*, so the heuristic would give preference to conjoining it with another necessary hypothesis.

Having decided, for instance, to create a new disjunctive hypothesis at level n, the algorithm uses sufficiency as a criterion for choosing operands. This is done by creating two sorted lists of hypotheses: the first list consists of the hypotheses of level $n - 1$, sorted from highest to lowest sufficiency; the second list contains all of the hypotheses from levels 0 to $n - 1$, also sorted by sufficiency. The first list is limited in order to allow complete coverage of the search space without duplication of hypotheses at different levels. Thus, for example, a hypothesis of depth 2 can be constructed at level 2, but one of depth 1 cannot.

Given the two sorted lists (another sorting criterion could easily be substituted for necessity or sufficiency at this point), a new disjunctive hypothesis is constructed by syntactically disjoining the formulae associated with the hypotheses at the top of each list. This new formula is then simplified and put into a canonical form.[10] If the simplified formula is of depth less than n it is discarded, because if it is important, it will occur at a lower level and we wish to avoid duplication. If it is of depth n, it is tested for syntactic equality against all other hypotheses at level n. If the hypothesis is not a syntactic duplicate, it is added to the candidate list of level n and its statistics are initialized. If the new hypothesis is too simple or is a duplicate, two new indices into the sorted lists are chosen and the process is repeated. The new indices are chosen so that the algorithm finds the non-duplicate disjunction made from a pair of hypotheses whose sum of indices is least. The complexity of this process can be controlled by limiting the total number of new hypotheses that can be tried before giving up. In addition, given such a limit, it is possible to generate only prefixes of the sorted operand lists that are long enough to support the desired number of attempts.

6.2. Promoting hypotheses

On each update phase, the candidate hypotheses are considered for promotion. The reason for dividing the candidate hypotheses from the working hypotheses is to be sure that they have gathered enough statistics for their values of N, S, and *er* to be fairly accurate before they enter the pool from which operands and the action-generating hypothesis are chosen. Thus, the criterion for promotion is simply the *age* of the hypothesis, which reflects the accuracy of its statistics. Any candidate that is old enough is moved, on this phase, to the working hypothesis list.

6.3. Pruning hypotheses

After candidates have been promoted, the total number of working hypotheses in a level may exceed the preset limit. If this happens, the working hypothesis list for the level is pruned. An hypothesis can play an important role in the GTRL algorithm for any of three reasons: its prediction value is high, making it useful for choosing actions; its sufficiency is high, making it useful for combining into disjunctions; or its necessity is high, making it useful for combining into conjunctions. For these reasons, we adopt the following pruning strategy:

To prune down to n hypotheses, first choose the $n/3$ hypotheses with the highest predictive value; of the remaining hypotheses, choose the $n/3$ with the highest necessity; and, finally, of the remaining hypotheses, choose the $n/3$ with the highest sufficiency.

This pruning criterion is applied to all but the bottom-most and top-most levels. Level 0, which contains the atomic hypotheses and their negations, must never be pruned, or the capability of generating the whole space of fixed-size Boolean formulae will be lost. Because its hypotheses will not undergo further recombination, the top level is pruned so as to retain the n most predictive hypotheses.

7. Parameters of the algorithm

The GTRL algorithm is highly configurable, with its complexity and learning ability controlled by the following parameters:

L: The number of levels of hypotheses.

$z_{\alpha/2}$: The size of the confidence interval used to generate *erp-ub*.

$H(l)$: The maximum number of working hypotheses per level; can be a function of level number, l.

$C(l)$: The maximum number of candidate hypotheses per level; can be a function of level number, l.

PA: The age at which candidate hypotheses are promoted to be working hypotheses.

R: The rate at which new hypotheses are generated; every R ticks, for each level, l, if there are not more than $C(l)$ candidate hypotheses, a new one is generated.

T: The maximum number of new hypotheses that are tried, in a tick, to find a non-duplicate hypothesis.

M: The number of input bits.

Because level 0 is fixed, we have $H(0) = 2M + 2$.

8. Computational complexity

The space complexity of the GTRL algorithm is

$$O(\sum_{j=0}^{L}(H(j) + C(j))2^j) \ ;$$

for each level j of the L levels, there are $H(j) + C(j)$ working and candidate hypotheses, each of which has size at most 2^j for the Boolean expression, plus a constant amount of space for storing the statistics associated with the hypothesis. This expression can be simplified, if H and C are independent of level, to

$$O(L(H + C)(2^{L+1} - 1)) \ .$$

which is

$$O(L(H+C)2^L) \ .$$

The time complexity for the evaluation function is

$$O(\sum_{j=0}^{L} H(j) + 2^L) \ ;$$

the first term accounts for spending a constant amount of time examining each working hypothesis to see which one has the highest predictive value. Once the most predictive working hypothesis is chosen, it must be tested for satisfaction by the input instance; this process takes time proportional to the size of the expression, the maximum possible value of which is 2^L. If H is independent of level, this simplifies to

$$O(LH + 2^L) \ .$$

The expression for computation time of the update function is considerably more complex. It is the sum of the time taken to update the statistics of all the working and candidate hypotheses plus, for each level, the time to add hypotheses, promote hypotheses, and prune hypotheses for the level.

The time to update the hypotheses is the sum of the times to update the individual hypotheses. The update phase requires that each hypothesis be tested to see if it is satisfied by the input. This testing requires time proportional to the size of the hypothesis. Thus we have a time complexity of

$$O(\sum_{j=0}^{L} (H(j) + C(j))2^j)$$

which simplifies to

$$O(L(H+C)2^L) \ .$$

The time to add hypotheses consists of the time to create the two sorted lists (assumed to be done in $n \log n$ time in the length of the list) plus the number of new hypotheses tried times the amount of time to construct and test a new hypothesis for duplication. This time is, for level j,

$$O(H(j-1)\log H(j-1) + (\sum_{k=0}^{j-1} H(k)) \log(\sum_{k=0}^{j-1} H(k)) + T2^j(H(j) + C(j))) \ .$$

The last term is the time for testing new hypotheses against old ones at the same level to be sure there are no duplicates. Testing for syntactic equality takes time proportional to the size of the hypothesis and must be done against all working and candidate hypotheses in level j. There is no explicit term for simplification of newly created hypotheses because GTRL uses a procedure that is linear in the size of the hypothesis.

The time to promote hypotheses is simply proportional to the number of candidates, $C(j)$.

Finally, the time to prune hypotheses is 3 times the time to choose the $H(j)/3$ best hypotheses which, for the purpose of developing upper bounds, is $H(j)\log H(j)$.

Summing these expressions for adding, promoting, and pruning at each level, and making the simplifying assumption that H and C do not vary with level yields a time complexity of

$$O(L(H\log H + LH\log(LH) + T2^L(H+C) + C + H\log H)) \ ,$$

which can be further simplified to

$$O(L^2H\log(LH) + T2^LL(H+C)) \ . \tag{1}$$

The time complexity of the statistical update component, $O(L(H+C)2^L)$, is dominated by the second term above, making expression 1 the time complexity of the entire update function. This is the complexity of the longest possible tick. The addition and pruning of hypotheses, which are the most time-consuming steps, will happen only once every R ticks. Taking this into account, we get a kind of "average worst-case" total complexity (the average is guaranteed when taken over a number of ticks, rather than being a kind of expected complexity based on assumptions about the distribution of inputs) of

$$O(L(H+C)2^L + \frac{1}{R}L^2H\log(LH) + \frac{T}{R}2^LL(H+C)) \ .$$

The complexity in the individual parameters is $O(2^L)$, $O(H\log H)$, $O(1/R)$, $O(T)$, $O(C)$. Clearly, the number of levels and the number of hypotheses per level have the greatest effect on total algorithmic complexity. This complexity is not as bad as it may look, because 2^L is just the length of the longest formula that can be constructed by the algorithm. The time and space complexities are linear in this length.

9. Choosing parameter values

This section explores the relationship between the settings of parameter values and the learning abilities of the GTRL algorithm.

9.1. *Number of levels*

Any Boolean function can be written with a wide variety of syntactic expressions. Consider the set of Boolean formulae with the negations driven in as far as possible, using DeMorgan's laws. The *depth* of such a formula is the maximum nesting depth of binary conjunction and disjunction operators within the formula. The *depth* of a Boolean function is defined to be the depth of the shallowest Boolean formula that expresses the function.

An instance of the GTRL algorithm with L levels of combination is unable to learn functions with depth greater than L. Whether it can learn all functions of depth L or less depends on the settings of other parameters in the algorithm. The time and space complexities of the algorithm are, technically, most sensitive to this parameter, both being exponential in the number of levels.

9.2. Number of working and candidate hypotheses

The choice of the size of the hypothesis lists at each level also has a great effect on the overall complexity of the algorithm. The working hypothesis list needs to be at least big enough to hold all of the subexpressions of some formula that describes the target function. Thus, in order to learn the function described by $i_0 \wedge (i_1 \vee i_2) \wedge (i_3 \vee \neg i_4)$, level 1 must have room for at least two working hypotheses, $i_1 \vee i_2$ and $i_2 \vee \neg i_4$, and levels 2 and 3 must have room for at least one working hypothesis each.

This amount of space will rarely be sufficient, however. There must also be room for newly generated hypotheses to stay until they are tested and proven or disproven by their performance in the environment. Exactly how much room this is depends on the rate, R, at which new hypotheses are generated and on the size, $z_{\alpha/2}$, of the confidence intervals used to generate *erp-ub*. To see this, consider the case in which a representation of the optimal hypothesis, h^*, has already been constructed. The algorithm continues to generate new hypotheses, one every R ticks, with each new hypothesis requiring an average of j ticks to be proven to be worse than h^*. That means there must be an average of R/j slots for extra hypotheses at this level. Of course, it is likely that during the course of a run, certain non-optimal hypotheses will take more than j ticks to disprove. This can cause h^* to be driven out of the hypothesis list altogether during the pruning phase. Thus, a more conservative strategy is to prevent this by increasing the size of the hypothesis lists, but it incurs a penalty in computation time.

Even when there is enough space for all subexpressions and their competitors at each level, it is possible for the size of the hypothesis lists to affect the speed at which the optimal hypothesis is generated by the algorithm. This can be easily understood in the context of the difficulty of a function for the algorithm. Functions whose subexpressions are not naturally preferred by the necessity and sufficiency search heuristics are difficult for the GTRL algorithm to construct. In such cases, the algorithm is reduced to randomly choosing expressions at each level.

Consider the case in which $h^* = (i_0 \wedge \neg i_1) \vee (\neg i_0 \wedge i_1)$, an exclusive-or function. Because h^* neither implies nor is implied by any of the input bits, the atoms will all have similar, average values of N and S. Due to random fluctuations in the environment, different atoms will have higher values of N and S at different times during a run. Thus, the conjunctions and disjunctions at level 1 will represent a sort of random search through expression space. This random search will eventually generate one of the following expressions: $i_0 \wedge \neg i_1$, $\neg i_0 \wedge i_1$, $i_0 \vee i_1$, $\neg i_0 \vee \neg i_1$. When one of these is generated, it will be retained in the level 1 hypothesis list because of its high necessity or sufficiency. We need only wait until the random combination process generates its companion subexpression, and they will be combined into a representation of h^* at level 2.

Even with very small hypothesis lists, the correct answer will eventually be generated. However, as problems become more difficult, the probability that the random process will, on any given tick, generate the appropriate operands becomes very small, making the algorithm arbitrarily slow to converge to the correct answer. This process can be made to take fewer ticks by increasing the size of the hypothesis list. In the limit, the hypothesis list will be large enough to hold all conjunctions and disjunctions of atoms at the previous level and as soon as it is filled, the correct building blocks for the next level will be available and apparent.

9.3. Promotion age

The choice of values for the age parameter depends on how long it takes for the *er*, N, and S statistics to come to be a good indication of the values they are estimating. If reinforcement has a high variance, for instance, it may take more examples to get a true statistical picture of the underlying processes. If the value of R is large, causing new combinations to be made infrequently, it is often important for promotion age to be large, ensuring that the data that guides the combinations is accurate. If R is small, the effect of occasional bad combinations is not so great and may be outweighed by the advantage of moving candidate hypotheses more quickly to the working hypothesis list.

9.4. Rate of generating hypotheses

The more frequently new hypotheses are generated, the sooner the algorithm will construct important subexpressions and the more closely it will track a changing environment. However, each new hypothesis that has a promising value of *er* will be executed a number of times to see if its value of *erp* is as high as that of the current best hypothesis. In general, most of these hypotheses will not be as good as the best existing one, so using them to choose actions will decrease the algorithm's overall performance significantly.

9.5. Maximum new hypothesis tries

The attempt to make a new hypothesis can fail for two reasons. Either the newly-created hypothesis already exists in the working or candidate hypothesis list of the level for which it was created or the expression associated with the hypothesis was reducible to a syntactically simpler expression, making it inappropriate for this level. It is possible, but very unlikely, to have more than $H + C$ failures of the first type. The number of failures of the second type is harder to quantify.

10. Experimental results

This section reports the results of a set of experiments designed to compare the performance of GTRL with the algorithms described in the companion paper on a set of problems that illuminates their relative strengths and weaknesses.

10.1. Algorithms and tasks

The following algorithms were tested in these experiments:

Table 1. Parameters of test tasks.

Task	M	*expr*	p_{1s}	p_{1n}	p_{0s}	p_{0n}
1	3	$(i_0 \wedge i_1) \vee (i_1 \wedge i_2)$	.6	.4	.4	.6
2	3	$(i_0 \wedge \neg i_1) \vee (i_1 \wedge \neg i_2) \vee (i_2 \wedge \neg i_0)$	.9	.1	.1	.9
3	10	$i_2 \wedge \neg i_5$	.9	.6	.5	.8

- IE: Table-based statistical algorithm
- LARC: Linear associator with reinforcement comparison
- BPRC: Two-layer, fully connected backpropagation algorithm with reinforcement comparison
- LARCKDNF: Linear associator with fixed layer of conjunctive units
- IEKDNF: Statistical algorithm based on Valiant's technique for learning functions in k-DNF
- GTRL: Generate-and-test algorithm described here

The regular interval-estimation algorithm IE is included as a yardstick; it is computationally much more complex than the other algorithms and may be expected to out-perform them.

Each of the algorithms was tested in three different tasks. The tasks are called *binomial Boolean expression worlds* and can be characterized by the parameters M, *expr*, p_{1s}, p_{1n}, p_{0s}, and p_{0n}. The parameter M is the number of input bits; *expr* is a Boolean expression over the input bits; p_{1s} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance satisfies *expr*; p_{1n} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance does not satisfy *expr*; p_{0s} is the probability of receiving reinforcement value 1 given that action 0 is taken when the input instance satisfies *expr*; p_{0n} is the probability of receiving reinforcement value 1 given that action 0 is taken when the input instance does not satisfy *expr*. Input vectors are chosen randomly by the world according to a uniform probability distribution.

Table 1 shows the values of these parameters for each task. The first task has a simple, linearly separable function; what makes it difficult is the small separation between the reinforcement probabilities. Task 2 has highly differentiated reinforcement probabilities, but the function to be learned is a complex exclusive-or. Finally, Task 3 is a simple conjunctive function, but all of the reinforcement probabilities are high and it has significantly more input bits than the other two tasks.

10.2. Parameter tuning

In the GTRL algorithm, many of the parameters were fixed arbitrarily: the parameter L was set to 3, $z_{\alpha/2}$ to 2, C to be equal to H, and T to 100. The remaining parameters PA (the age at which candidates are promoted), R (the rate at which new hypotheses are

Table 2. Best parameter values for the GTRL algorithm on each task.

TASK	1	2	3
PA	35	10	25
R	200	100	450
H	30	30	20

Table 3. Average reinforcement for tasks over 100 runs of length 3000.

ALG-TASK	1	2	3
IE	.5827	.8966	.7205
LARC	.5456	.7459	.7644
BPRC	.5456	.7406	.7620
LARCKDNF	.5783	.8903	.7474
IEKDNF	.5789	.8900	.7939
GTRL	**.5667**	**.7968**	**.7986**
random	.5000	.5000	.7000
optimal	.6000	.9000	.8250

generated), and H (the number of hypotheses per level), were chosen experimentally, to optimize the behavior on each task. The success of an algorithm is measured by the average reinforcement received per tick, averaged over the entire run. The other algorithms' parameters were also tuned experimentally (Kaelbling, 1994).

For each task, a series of 100 trials of length 3000 were run with different parameter values. Table 2 shows the best set of parameter values found for each task.

10.3. Results

Using the best parameter values for each algorithm and task, the performance of the algorithms was compared on runs of length 3000. The performance metric was average reinforcement per tick, averaged over the entire run. The results are shown in table 3, together with the expected reinforcement of executing a completely random behavior (choosing actions 0 and 1 with equal probability) and of executing the optimal behavior.

These results do not tell the entire story, however. It is important to test for statistical significance to be relatively sure that the ordering of one algorithm over another did not arise by chance. Figure 3 shows, for each task, a pictorial representation of the results of a 1-sided t-test applied to each pair of experimental results. The graphs encode a partial order of statistically significant dominance, with solid lines representing significance at the .95 level.

With the best parameter values for each algorithm, it is also instructive to compare the rate at which performance improves as a function of the number of training instances. Figures 4, 5, and 6 show superimposed plots of the learning curves for each of the algorithms. Each point represents the average reinforcement received over a sequence of 100 steps, averaged over 100 runs of length 3000.

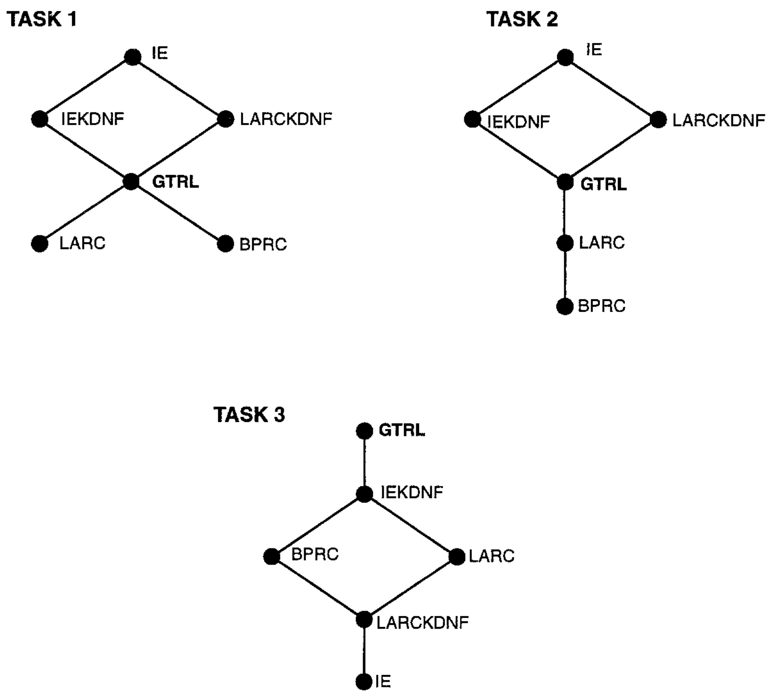

Figure 3. Significant dominance partial order among algorithms for each task.

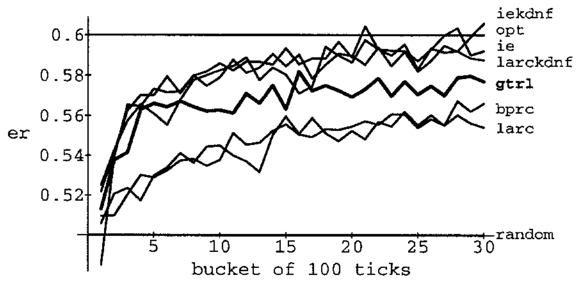

Figure 4. Learning curves for Task 1.

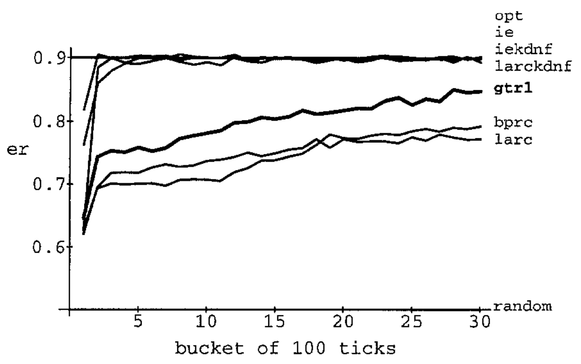

Figure 5. Learning curves for Task 2.

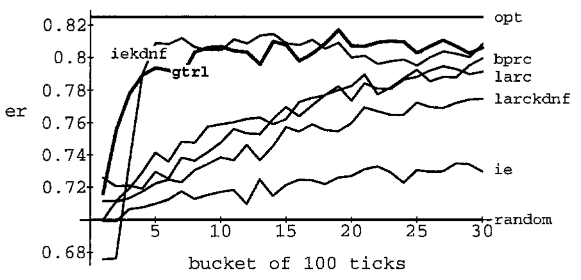

Figure 6. Learning curves for Task 3.

10.4. Discussion

The GTRL algorithm performs significantly better than the LARC and BPRC algorithms, but worse than the IE, IEKDNF, and LARCKDNF algorithms, on tasks 1 and 2. One reason for this is that there is no explicit mechanism in GTRL to cause it to quit exploring. We can see in the learning curves that, although it improves quickly early in the run, it does not reach as high a steady-state level of performance as the other algorithms. It does not converge to a fixed state, because it is always entertaining new competing hypotheses. This flexibility causes a large decrease in performance. If the GTRL algorithm is to be applied in a domain that is stationary (as in the assumptions stated in the first section), performance can be improved by decreasing over time the rate at which new candidate hypotheses are generated. This will cause the algorithm to spend less time experimenting and more time acting on the basis of known good hypotheses. However, the current behavior may be an asset in situations in which the task can change over time.

Task 3 reveals many interesting strengths and weaknesses of the algorithms. One of the most interesting is that IE suddenly becomes the worst performer. Because the target function is simple and there is a larger number of input bits, the ability to generalize across input instances becomes crucial. The IEKDNF algorithm is able to find the correct action function early during the run (this is apparent in the learning curve of figure 6). The GTRL algorithm has similar but significantly superior performance, finding the correct hypothesis promptly and using it for a large portion of its actions. The BPRC and LARCKDNF algorithms have very similar performance on task 3, with the LARC algorithm performing slightly worse, but still reasonably well. The good performance of the generalizing algorithms is especially apparent when we consider the size of the input space for this task. With 10 input bits, by the end of a run of length 3000, each input can only be expected to have been seen about 3 times. This accounts for the poor performance of IE, which would eventually reach optimal asymptotic performance on longer runs.

11. Conclusion

We have seen that the GTRL algorithm can be used to learn a variety of Boolean function classes with varying degrees of effectiveness and efficiency. This paper describes only a particular instance of a general, dynamic generate-and-test method—there are a number of other possible variations.

The algorithm is designed so that other search heuristics may be easily accommodated. An example of another, potentially useful, heuristic is to combine hypotheses that are highly correlated with the optimal hypothesis. One way to implement this heuristic would be to run a linear-association algorithm, such as LARC, over the input bits and the outputs of the newly created hypotheses, then make combinations of those hypotheses that evolve large weights. It is not immediately apparent how this would compare to using the N and S heuristics.

Another possible extension would be to add genetically motivated operators, such as crossover and mutation, to the set of search operators. Many genetic methods are concerned only with the performance of the final result so this extension would have to be made carefully in order to preserve good on-line performance.

Acknowledgments

Thanks to Stan Rosenschein for providing financial and moral support and to Rich Sutton for helpful discussions of neural-network and statistical reinforcement learning methods. An anonymous reviewer supplied a number of useful comments and questions.

This work was supported in part by the Air Force Office of Scientific Research under contract F49620-89-C-055, in part by the System Development Foundation, in part by Teleos Research IR&D, in part by National Science Foundation National Young Investigator Award IRI-9257592 and in part by ONR Contract N00014-91-4052, ARPA Order 8225.

Notes

1. Any Boolean formula can be put in this form using DeMorgan's laws. It may be, however, that a hypothesis that would be most simply expressed with an outer negation would be more difficult to learn in the current system. One possible extension would be to allow negation as a search operation, with the heuristic being to negate hypotheses that have both low necessity and low sufficiency.
2. Other choices of syntactic search operators are possible. Conjunction and disjunction are used here because of the availability of good heuristics for guiding their application.
3. An alternative would be to limit the total number of hypotheses, without sorting them into levels. This approach would give added flexibility, but would also cause some increase in computational complexity. In addition, it is often beneficial to retain hypotheses at low levels of complexity because of their usefulness as building blocks.
4. It is necessary to include *true* and *false* in case either of those is the optimal hypothesis. Hypotheses at higher levels are simplified, so even if $a \wedge \neg a$ or $a \vee \neg a$ were to be constructed, it would not be retained.
5. In all of the experiments described here, κ had the value 1000.
6. This difference in distributions depends on the fact that $er(h)$ is conditioned on the agreement between hypothesis h and whatever hypotheses are actually being used to generate actions.
7. Terminology is being abused here in order to simplify the presentation. Rather than conjoining hypotheses, the algorithm actually creates a new hypothesis whose formula is the conjunction of the formulae of the operand hypotheses. This use of terminology should not cause any confusion.
8. Schlimmer's STAGGER system generates new hypotheses in response to errors, using the nature of the error (false positive vs. true negative) to determine whether the new hypothesis should be a conjunction or a disjunction. This method cannot be applied in the general reinforcement-learning scenario, in which the algorithm is never told what the "correct" answer is, making it unable to know whether or not it just made an "error."
9. The LS and LN functions were designed for combining evidence in a human-intuitive way; their quantitative properties are crucial to their correctness and usefulness for this purpose. The S and N operators that will be proposed do not have the appropriate quantitative properties for such uses.
10. The choice of canonicalization and simplification procedures represents a tradeoff between computation time and space used in canonicalization against the likelihood that duplicate hypotheses will not be detected. Any process for putting Boolean formulae into a normal form that reduces semantic equivalence to syntactic equivalence has exponential worst-case time and space complexity in the original size of the formula. The GTRL algorithm currently uses a very simple simplification process whose complexity is linear in the original size of the formula and that seems, empirically, to work well. This simplification process is described in detail in elsewhere (Kaelbling, 1993).

References

Duda, R. O., Gaschnig, J., & Hart, P. E. (1979). Model design in the Prospector consultant system for mineral exploration. In D. Michie (Ed.), *Expert Systems in the Micro Electronic Age*. Edinburgh, U.K.: Edinburgh University Press.

Duda, R. O., Hart, P. E., & Nilsson, N. J. (1976). *Subjective Bayesian Methods for Rule-Based Inference Systems*. Technical Report 124, Artificial Intelligence Center, SRI International, Menlo Park, California.

Kaelbling, L. P. (1994). Associative reinforcement learning: Functions in k-DNF. *Machine Learning, 15*, 279-298.

Kaelbling, L. P. (1993). *Learning in Embedded Systems*. Cambridge, Massachusetts: The MIT Press. Also available as a PhD Thesis from Stanford University, 1990.

Saxena, S. (1991). *Predicting the Effect of Instance Representations on Inductive Learning*. PhD thesis, University of Massachusetts, Amherst, Massachusetts.

Schlimmer, J. C. (1987). *Concept Acquisition Through Representational Adjustment*. PhD thesis, University of California, Irvine, Irvine, California.

Schlimmer, J. C. & Granger, Jr., R. H. (1986). Incremental learning from noisy data. *Machine Learning*, 1(3), 317–354.

Final Manuscript July 19, 1993