The Pleasure Algorithm

Ben Goertzel

January 10, 2008

This document briefly describes a new program learning algorithm called PLEASURE, which stands for "Program Learning via Ensemble Analysis and SUbstitution of Repeated Entities."

How well it will work, in what situations, is unknown – at the moment, it is proposed purely based on intuition, guided by suggestive evidence from experimentation with other algorithms with related properties (as will be noted below).

Pleasure is actually more of a meta-algorithm than an algorithm. It requires some other program learning algorithm as a subroutine, and this other program learning algorithm (to be called the sub-algorithm, in a Pleasure context) must be capable of producing an ensemble of qualitatively different programs in response to a given fitness function. Example sub-algorithms that may be effective are genetic programming, decision tree forests, or MOSES. GP or MOSES can be expected to lead to better answers, whereas decision tree forests can be expected to yield faster performance. (Depending on how fast Pleasure with decision tree forests turns out to be, it could be possible to use Pleasure with decision tree forests within MOSES in place of hBOA or hill-climbing.) Experimentation will be the only way to tell which algorithms can be most effectively "Pleasured."

The basic goal of Pleasure is similar to the goal of SEAM and hBOA: to effectively make use of the strong bias toward hierarchically structured programs, which is believed to be implicit in many pragmatically-valuable fitness functions. What is different in Pleasure is its focus on repeated phases of clustering and search for repeated structures. The idea is that hierarchical structures may be defined in terms of contextually-defined functional categories of "elementary program components", rather than in terms of program components themselves.

The Pleasure approach is inspired in part by Tony C. Smith's algorithm for automated induction of English grammar, see http://citeseer.ist.psu.edu/104780.html; but the commonality with Smith's work is purely conceptual, not in terms of detailed algorithmics. It is also inspired, to a large degree, by work on mining patterns from ensembles of classification rules that we have done in the context of mining patterns from bioinformatic data.

Assume one has a fitness function F, and is seeking to learn a program P maximizing F. Assume for simplicity that P is represented as a dag, whose nodes are instances of nodes drawn from a node-type set T. Node-types fall into two categories: internal types, and terminal types. Part of the Pleasure algorithm involves iteratively modifying the node-type set used in programs.

The initial node-type set T is assumed to contain a fixed set of internal node-types (for instance, Boolean operators, arithmetic operators, programmatic constructs); and a fixed set of terminal node-types, which may fall into a number of different types (e.g. Boolean, integer, real number, real vector), and may include both constant types and variable types. Evaluating a program P consists of feeding specific values into its terminal nodes and then obtaining the program's output.

For instance a typical set of types defining Boolean programs would be {AND, OR, NOT, $I_{Boolean}$}, with $I_{Boolean}$ denoting a Boolean variable input. A typical set of types defining floating-point functions would be {*,/,+,-, I_{Float} , C_{Float}}, with C_{Float} denoting a float constant input. A program controlling an embodied agent might have types such as {AND, OR, NOT, IFELSE, Near, Visible, elementAt, I_{Agents}, $I_{FirstStructureAhead}$, I_{Happy}, $I_{Aggressive}$,...} where e.g.

- elementAt, append are list operators
- I_{Happy} is a terminal node-type that returns the degree of the agent's happiness, $I_{FirstStructureAhead}$ is a terminal node that returns the name of the first structure the agent detects in its line of sight
- I_{Agents} is a list of the agents currently close to the agent, etc.

The above is not intended as an exhaustive list, just a few examples. In general we can support the full scope of Combo node types.

In this context, the basic steps of Pleasure are as follows.

Let R = T
Optionally, define in advance a set of "possibly useful subgraphs" and "possibly useful properties."

1. Run the sub-algorithm, to find an ensemble E of programs using node-type set R, that maximize F to varying degrees
2. Divide E into three categories, in terms of their performance at maximizing F: Best, Middle, and Worst
3. Normalize the Best and Worst programs, using an appropriate set of reduction rules (if the sub-algorithm didn't already normalize them)
4. Study E to find
    a. the percentage of programs in E that use r, for each node r in R (this is the "importance" of r)
    b. a set S_{FU} of "frequently useful subgraphs": subgraphs that are significantly more frequent in Best than in Worst; this search may be primed by the set of possibly useful subgraphs provided in advance
    c. a set S_{SUE} of categories of "similarly utilized entities" (SUE categories), each entity consisting of entities of the same type signature that form a cluster in property-space: C_1, ..., C_k ; this search may be primed by the set of possibly useful properties provided in advance

5. Create a new node-type set NR, which appends to R the K_{FU} most significant elements of S_{FU}, and the K_{SUE} most significant elements of S_{SUE}
6. If NR has more than R_{max} elements, then remove the least important elements of NR to reduce the size to R_{max}
7. Rewrite the program trees in the population E to utilize the new node-types created. This may create multiple versions of many of the programs, which is okay – so long as we use a data structure in which it is made clear that they represent the same thing. However, a cap Pmax must be placed on the number of replicates of any particular program to be retained, so that prioritization must be made of which versions of a program to retain; as a heuristic, the versions may be ranked in order of total node importance
8. Set R = NR, and return to Step 1

What is not specified in the above is the definition of FU subgraphs and SUE categories.

In the search for FU subgraphs, it is likely wise to restrict search to small subgraphs. If there are larger frequent subgraphs, in most cases they will be found via repeated iteration of small-subgraph search across multiple iterations of Pleasure. Also, as noted above, if a list of possibly-useful subgraphs (or templates thereof) is provided, it may be utilized to guide the search.

The one subtlety involving FU subgraphs involves the use of constants drawn from metric spaces, such as reals, real vectors or discrete metric spaces. In this case the occurrence of a subgraph in a program may be fuzzy rather than absolute, and these fuzzy values must be accounted for in calculating the total frequency of the subgraph.

As an example, consider a problem of classifying individuals into Case versus Control based on gene expression data associated with them. It may be that out of 100 Good classification models, 15 of them contain the subgraph (NM_95544 AND NM_4433) -- where the NM_xxx notation indicates a Boolean terminal variable that returns true if the expression level of the indicated gene in the individual is greater than a certain threshold. Then we may identify this as a FU subgraph, and create a new terminal node-type N that evaluates True for an individual only if the expression (NM_95544 AND NM_4433) is true for that individual.

In a problem of learning a program to control an agent, it may happen that a sequence of the form Ifelse(Near(I_{object}) , (Goto(I_{object}) ANDSeq Grab(I_{object}))) occurs frequently, in which case this sequence will become an FU subgraph.

In order to find SUE categories, we construct a set of properties Pr_i, each of which is a property that can be evaluated at any particular node N, to see the extent to which $Pr_i(N)$ is true. We then associate each node-type N_i with a vector V_i so that $V_{ij} = Pr_i(N_j)$, where the latter expression is interpreted as an average over all specific nodes of the node-type N_j. (The case of binary properties is the simplest one but not necessarily the only one worth considering.) Similarly, we associate each (terminal-node-type, value) pair with a property vector. We then cluster the vectors V_i, taking care to form only clusters that contain vectors corresponding to node-types with identical arity. Each cluster is a SUE category, with a certain cluster quality attached to it, and also a certain type signature.

Suppose one has a program tree with a SUE category node C included in it. How is C evaluated? The answer depends on C's type signature (which is identical to that of the nodes C clusters).

Suppose, for instance, C is of type *Object --> Boolean*. That is, C is a single-argument predicate. Then the question becomes: How does one evaluate C(x), where x is an input? The most natural answer seems to be: One averages $C_i(x)$ over all C_i in C. The average may be weighted, where the weight assigned to each C_i may be calculated in various ways, e.g.:

- Proportional the the incidence of Ci in the Good population
- Proportional to the average fitness of Ci
- Proportional to the differentiation that C_i, in itself, provides between Good and Bad in the population.

Next, suppose, C is a constant? Then to evaluate C, one randomly samples a member x of C, with the probability of choosing x proportional to the frequency of x in the population. The random sampling may be done according to method similar to the weighting mentioned above.

Or, suppose C is of type *Object --> T*. In this case, if type T forms a linear space, the most natural way to define C(x) seems to be as the average of $C_i(x)$ over all C_i in C. where the weighting is handled as mentioned above. On the other hand, if T does not form a linear space, then there are two options:

- Choose an object at random from the outputs of the $C_i(x)$, from an appropriate distribution
- Embed the set of objects of type T in a linear space, and choose the $C_i(x)$ that is closest to the centroid

What are the properties Pr_i to be considered? This is the subtlest part of the algorithm, and will no doubt depend on the problem domain. But some general suggestions may be made.

For instance, given an internal node-type N_i that has two children, and another internal node-type N_j, then we may create a property of the form "has N_i as parent and N_j as sibling under N_i". If the order of N_i's siblings is important, then we may create a property of the form "is left-child of Ni and has N_j as sibling under N_i" (and similarly for right-child).

Similarly, for the same N_i as above and a terminal node-type T_j, we may create a property of the form "has N_i as parent and, as sibling under N_i, has T_j with value V_k." In the case that T_j has values in a metric space, then this sort of property may apply to a fuzzy extent.

More complex properties may be created on a domain-specific basis. As the space of complex properties may become overly large, an a priori list of "possibly interesting complex properties" may be used to guide the search.

For instance, in the gene expression example, we might have a property of the form

Pr(N) is true iff (N AND NM_444) occurs

This could lead to a SUE category of items that co-occur with gene NM_444.

In the agent control example, we might have a property of the form

Pr(N) is true iff Grab(N) occurs

This could lead to a SUE category of items that are grabbed (in Good programs). Combining this with the property

Pr(N) is true if near(N) occurs

could lead to a SUE category of items that are assessed as nearby and are grabbed.