Abstract—Evolutionary algorithms aim to harness ideas from biological evolution in order to explore search spaces that may be hard to understand at a glance. Instead of viewing search spaces from an evolutionary perspective, we may also look at exploration from the perspective of a human learner. Instead of trying to tackle the most difficult problems in a problem domain at the outset, this approach initially looks at simpler problems from the domain and then learns useful solution concepts from these problems, which can then be applied to successively more difficult instances. The E.C. algorithm developed by Dechter et al. [1] aims to capture this perspective. The E.C. algorithm has previously been applied, with mixed results, to problems such as symbolic regression and Boolean function learning. Here we use the E.C. algorithm to learn regular expressions for binary languages.

I. INTRODUCTION

Suppose you are given a collection of circuit elements and asked to build an entire variety of circuits, ranging from a simple circuit that adds bits to a very complicated one used in noise-canceling headphones [12]. Suppose, furthermore, that you know nothing about circuits. How would you approach the problem?

Many human learners have the intuition to first attempt the simple circuits before trying the more difficult ones. This approach works well in a variety of situations, including learning how to solve a class of math problems, read a set of texts, or train certain body movements in sports. Why does this work? Supposing the learner knows nothing about the domain of problems except that certain problems are less difficult than others, the solutions to the simpler problems are less complicated and more likely to be intuitively generated by the learner. After receiving feedback regarding the success of each proposed solution, the learner uses parts of her previously successful solutions to solve increasingly difficult problems. In the case of circuits, this intuition has a very literal interpretation: many of the smaller components that comprise simpler circuits—such as XOR gates in a bit adder—are shared between solutions to circuit problems, including more complicated circuits—such as those used in CDMA receivers and decoders for error correction.

This concept of reusable subcomponents has a long history in artificial intelligence (AI) [1]. Herb Simon discusses the concept in [13]; examples of reusable subcomponents appear in search [11], inductive logic programming [8], and predicate invention [4]. Some of the most recent progress in work with reusable subcomponents is in genetic programming (GP), where the concept has been applied to evolving automatic function definitions [5] and circuits [6], [7].

How fundamentally different is this approach from the traditional GP approach? Koza, Keane, and Streeter provide a telling example of the difference between the GP approach and human intuition in the case of circuits [7]. Figure 1 (taken from [7]) depicts two circuits for cubic signal generators, one designed by a human and the bottom one evolved. Although the evolved circuit works, it is not understood how it works because there is no recognizable pattern to the subcomponents that the circuit uses, which makes it difficult to break down the purposes of different circuit parts. The evolved circuit even contains redundant parts (e.g. the purple transistor).

In our approach, we want to not only reward successful candidate solutions, but more importantly reward successful subcomponents that appear in many of the more successful candidate solutions. This intuition is formalized in the E.C. algorithm [1].

How fundamentally different is this approach from the traditional GP approach? Koza, Keane, and Streeter provide a telling example of the difference between the GP approach and human intuition in the case of circuits [7]. Figure 1 (taken from [7]) depicts two circuits for cubic signal generators, one designed by a human and the bottom one evolved. Although the evolved circuit works, it is not understood how it works because there is no recognizable pattern to the subcomponents that the circuit uses, which makes it difficult to break down the purposes of different circuit parts. The evolved circuit even contains redundant parts (e.g. the purple transistor).
computer vision [10] or bridge building [3], where the “optimal” solutions produced by GP indeed outperform human solutions according to the fitness function, but are incredibly difficult to understand. It is certainly a merit that GPs can generate solutions that humans would not have imagined, but in the traditional GP approach this seems to trade off with understandability of the solutions [7]. We believe our approach can reconcile these two objectives of optimality and understandability by rewarding the use of successful subcomponents, so that even the complicated solutions produced by this approach can be decomposed into a set of understandable parts. This approach additionally views the challenge of learning to solve a set of related problems not as one of optimizing some fixed fitness functions as quickly as possible, but rather as one of gradually acquiring the knowledge building blocks to solve the pre-specified problems and future related tasks that may arise.

The E.C. algorithm—arguably the first formalization of this approach—was recently introduced and has only experimentally been tested in boolean function learning, symbolic regression [1], and learning of graphical models to fit datasets [2]. It has had promising but mixed performance on these problems and has yet to be applied elsewhere as further tests of its robustness. Here we apply the E.C. algorithm to the learning of regular expressions (regexes) that match given sets of strings. In Section 2, we outline the E.C. algorithm and previous work on learning regular expressions. In Section 3, we describe the setup of our experiments with the E.C. algorithm for learning specific regex tasks. In Section 4, we present our results, and Section 5 discusses the advantages and specific problems encountered by the E.C. algorithm on the variety of regex learning tasks. Section 6 concludes and proposes future directions for algorithms within our learning-based approach.

II. RELATED WORK

How does the E.C. algorithm formalize the aforementioned intuitive approach to learning? Each regex “task” is a set of positive examples—strings we wish the learned regex to match—and negative examples—strings we wish not to match. We will denote each task $t_k$ by $p_1, p_2, \ldots, p_m; n_1, n_2, \ldots, n_l$ where $p_i$ is a positive example (for example, “01”) and $n_j$ is a negative example. This gives a set of tasks $T$. Note that we can alternatively view each task as a function taking a regex to 1 if the regex solves the task, and 0 otherwise:

$$T := \{t_k\}_{k=1}^K \text{ where } t_k : \mathcal{L} \to \{0, 1\}$$

where $\mathcal{L}$ is the set of regular expressions in our language. Our goal is to solve as many tasks as possible.

Then the E.C. algorithm accomplishes this goal by doing the following [1]:

1) Enumerate the frontier, the $N$ most probable regexes in $D$, and

2) Update $\mathcal{D}$ by compressing the good regexes into common subregexes, up-weighting the subregexes’ probabilities.

In the following sections, we compress [1]’s discussion about the E.C.’s algorithm’s details into two questions: how to maintain a distribution $\mathcal{D}$ over regexes $\mathcal{L}$ and how to perform the steps of enumerating the frontier and updating $\mathcal{D}$ based on common subregexes.

A. Maintaining a Distribution over Regular Expressions

The first step to maintaining a distribution over regexes is to represent each regex as a binary tree. In an approach similar to the representation of functional programs by binary trees [1], we first note that any function can be expressed as the applications of a set of combinators:

- $I \ x \rightarrow x$
- $S \ f \ g \ x \rightarrow f(x)(g(x))$
- $C \ f \ g \ x \rightarrow f(x)(g)$
- $B \ f \ g \ x \rightarrow f(g(x))$

(Note that in $S$, $f(x)$ returns a function applied to $g(x)$, and in $C$, $f(x)$ returns a function applied to $g$.) Combining these combinators from the representation of functional programs with a set of regex primitives—0, 1, union (denoted $|$), concatenation (denoted $+$), and Kleene Star (denoted $*$)—we can form any regex to solve a task $t$. To see this, if $t = p_1, \ldots, p_m; n_1, \ldots, n_l$, then the trivial regex $\langle p_1 \ldots p_m \rangle$ solves $t$, where this regex as represented in terms of our combinators and primitives is $\langle \langle \langle e_1 \langle \langle e_2 \ldots (\langle (e_{m_0} e_{m}) \ldots) \rangle \rangle \rangle \rangle$, and each $e_i$ is the regex for $p_i$, represented as $(++ p_{i,0} (+p_{i,1} (\ldots (++ p_{i,m_i-1} p_{i,m_i}) \ldots)))$ where $p_{i,j}$ is the $j$-th letter of $p_i$. Of course, we would hope for more intelligent and concise regexes than this to solve our tasks, but this demonstrates that any regex to solve a task $t$ can be written in terms of combinators and primitives.

[Diagram of a binary tree with a star and 1 as the root nodes.]

Fig. 2. An example of representation of the regex “.*” for binary strings in terms of primitives and combinators in a tree. The leaves are the only nodes containing primitives and operators; all other nodes represent the application of the left child function to the right child argument. In this tree, we start with the bottom-most subtree and apply $|$ to 0, then to 1; then we apply $*$ to the previous result to get the regex “(0|1)*”, which for binary strings is equivalent to “.*”.

Writing regexes in terms of combinators and primitives helps us represent the regexes as binary trees. Take the regex
".*" as an example (Figure 2). The leaves are the only nodes containing primitives and operators; all other nodes represent the application of the left child, as a function, to the right child, as an argument to this function. For example, the bottom-most subtree with children \(|\) and 0 is the application of the function \(|\) to the argument 0, returning a curried function which takes a single argument \(x\) and returns the regex "0\.|x". This function is then the left child of the next-highest subtree, which, when applied to 1, returns the regex "0\.|1". This regex is the right child of the topmost tree, so it becomes the argument to the Kleene Star; thus the entire tree represents "(0\.|1)*", which for binary strings is equivalent to ".*".

The purpose of representing regexes as binary trees is to allow us to put a probability distribution over possible trees, namely in the form of a stochastic grammar. Suppose we have assigned probabilities \(p_1, \ldots, p_N\) to each of the primitives and combinators; then the probability of a regex tree \(e\) is given by

\[
p(e) = \prod_{c \in C_e} p(c|\tau(c))
\]

where \(C_e\) is the set of primitives and combinators in the leaf nodes of regex \(e\), \(\tau(c)\) is the type of the primitive or combinator \(c\), and \(p(c|\tau(c))\) denotes the the probability of picking the specific primitive or combinator \(c\) among all type-consistent possibilities. We now have the ability to specify a probability distribution over regexes.

### B. Frontier Enumeration and Updating \(\mathcal{D}\)

At each step of the E.C. algorithm, we first enumerate the frontier, identifying the \(N\) most likely regexes in the distribution \(\mathcal{D}\) (see [1] for more details on this). After enumerating the frontier, we want to be able to describe the tasks that we solved succinctly. One way to achieve succinctness is to minimize the number of unique subtrees that exist in all of the trees representing complete solutions to tasks in the frontier. This rewards having modular concepts common to many solutions, as having a concept appearing many times decreases the number of unique subtrees present. However, as solving this problem is computationally quite difficult, instead we consider tasks iteratively. For the \(i\)'th task solved by a tree in the frontier, we look at all the trees in the frontier solving this task, and look at the tree we chose for the \(i-1\)'st task and choose for the \(i\)'th task the tree that has the fewest unique subtrees not found in the \(i-1\)'st task’s tree. This runs in runtime quadratic in the number of frontier trees that correspond with a task, so in worst case quadratic in frontier size. In the EC algorithm as it currently stands, for computational purposes we only consider a small constant number of frontier trees for the \(i\)'th task.

Now that we have chosen a succinct set of solutions, the E.C. algorithm uses the Nevill-Manning algorithm to compress the grammar so that subtrees that occur many times get turned into primitives. The weight in the grammar is then determined by the number of times each new primitive is used in the grammar.

### C. Experiments with the E.C. Algorithm

The E.C. algorithm has been run previously on three experiments. The first of these is symbolic regression, in particular trying to fit quadratic polynomials with coefficients in \([0, 9]\). Each tree represents a function from \(\mathbb{N} \to \mathbb{N}\) and is made up of basic combinators, 0, 1, +, and \(\times\).

![Fig. 3. Symbolic regression performance for different frontier sizes.](image1)

![Fig. 4. Boolean function learning performance. Top graph is for task set made from random combinations of and/or/not gates and different frontier sizes. Bottom graph is for task set consisting of all Boolean functions on three input variables and frontier size 2000.](image2)

The second task the E.C. algorithm has been applied to is Boolean function learning. For this algorithm one task set used was random circuits built out of and, or, and not gates. Another task set was simply all Boolean functions on three input variables. Each tree represents a Boolean function and is made up of basic combinators, True, False, and NAND gates.

Results for this follow the same form as results for symbolic regression.
The third and final task was to learn graphical models, with 12 tasks each corresponding to some specific type of graphical model. The algorithm was eventually able to solve 7/12 or 8/12 of the tasks with frontier size 1000 and 5000, and able to learn small versions of Hidden Markov Models and Ising Models.

D. Learning Regular Expressions

We briefly cover previous work related to the task of finding regexes matching certain strings and not matching others. One stronger version of the problem, in which the goal is to find regexes that not only match and fail to match the appropriate strings, but also are as short as possible, is popularly called “regex golf” in the computer science community. Peter Norvig recently showed that solving regex golf reduces to the set cover problem and demonstrated an approximation algorithm that performs decently [9]. Beyond Norvig’s recent work, the authors do not know of much work in this area.

III. METHODS

Each tree that we construct in the E.C. algorithm represents a regular expression over binary strings. For operations, we include union, concatenation, and Kleene Star, in addition to the default basic combinators in the E.C. Algorithm. We chose in particular to not include such basic operations such regular expression metacharacters as dot and plus so that our algorithm hopefully could learn these as concepts.

We run the E.C. Algorithm over two different task sets. A task consists of a set of strings that we require to be matched and a set of strings that we require not to be matched. We state that a task is solved by a regular expression if it matches all of the strings we want it to match and none that we do not want it to match.

The first task set consists of “simple” tasks and is similar to the tasks the E.C. algorithm has previously been used to solve [1]. It consists of finding regular expressions for subsets of the strings

“”, “0”, “1”, “00”, “01”, “10”, “11”

i.e. for all subsets of binary strings of length at most 2, which are. Note the similarity between this and previous analysis of learning circuits for all Boolean functions on 3 inputs using the base unit of a NAND gate.

The second task set contains more interesting tasks. Tasks in this set include learning building block tasks such as regular expression metacharacter concepts (for example, “.” for matching any possible single character—which in our case is equivalent to “0|1”—and “x*”, which is equivalent to “xx*”), strings of length exactly 2, and (most challengingly) regexes for divisibility by 3, 5, and 6.

IV. RESULTS

A. First Task Set

For the simple task set, the E.C. algorithm struggled with tasks such as “all strings except the empty string and 0” or “all strings except 0 and 1” as it is difficult to describe these with a very succinct regular expression without negation (which we excluded from out initial set of operations). In general the tasks that were difficult to solve had a small number of negative examples, or had many negative examples of length 1 and positive examples of length 2. A sampling of typical tasks solved and not solved is shown here (Table 1).

<table>
<thead>
<tr>
<th>Sampling of Results on First Task Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0,1,10,11;0,10,01</td>
</tr>
<tr>
<td>.0,1,01;0,011,11</td>
</tr>
<tr>
<td>1,0,01,11;0,10</td>
</tr>
<tr>
<td>0,01,11;0,01,10</td>
</tr>
<tr>
<td>0,1,01,10,11,0</td>
</tr>
<tr>
<td>.0,0,01,11,0,10</td>
</tr>
<tr>
<td>.0,0,10,11,1,01</td>
</tr>
<tr>
<td>.0,0,01,0,1,10</td>
</tr>
<tr>
<td>.0,0,01,10,11</td>
</tr>
<tr>
<td>.0,1,00,0,1,10,11</td>
</tr>
<tr>
<td>.0,1,00,0,1,10,11</td>
</tr>
<tr>
<td>.0,0,0,1,0,1,10,11</td>
</tr>
</tbody>
</table>

B. Second Task Set

For the complex data set, for all frontier sizes between 1000 and 10000, the E.C. algorithm solved 24 out of the 35 tasks. Most of the tasks solved were the simple tasks, with a number of the moderately interesting tasks solved as well, although the most interesting tasks such as divisibility by 3 were not solved. The grammar was also able to learn...
some interesting concepts over this task set, a sampling of which we include here (Table 2). That the expression we learned for plus is actually correct will be demonstrated in the discussion.

| Dot         | (0|1) |
|-------------|------|
| Plus        | S ++ * |
| Doubling a regex | S ++ I |

**TABLE II**
INTERESTING CONCEPTS LEARNED ON SECOND TASK SET

A sampling of tasks solved and not solved is included as well (Table 3).

<table>
<thead>
<tr>
<th>*</th>
<th>Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.*</td>
<td>S ++ * (0(1*))</td>
</tr>
<tr>
<td>0+</td>
<td>S ++ * 0</td>
</tr>
<tr>
<td>(01)+</td>
<td>S ++ * (0 ++ 1)</td>
</tr>
<tr>
<td>(1(00)+)+</td>
<td>S ++ * (1 ++ (S ++ * 0))</td>
</tr>
<tr>
<td>Divisibility by 2</td>
<td>Solved</td>
</tr>
<tr>
<td>Strings of length exactly 2</td>
<td>S ++ I 0(1)</td>
</tr>
<tr>
<td>Strings with exactly one 0</td>
<td>S ++ (++0) 1*</td>
</tr>
<tr>
<td>Strings of length exactly 3</td>
<td>Not Solved</td>
</tr>
<tr>
<td>Not Solved</td>
<td>++(0) (S ++ (U 0 1)) 0</td>
</tr>
<tr>
<td>Strings of length 2 or 3</td>
<td>Not Solved</td>
</tr>
<tr>
<td>Not Solved</td>
<td>(S ++ <em>(S ++ (++) 0 1)</em></td>
</tr>
<tr>
<td>Strings with exactly two 0s</td>
<td>Not solved</td>
</tr>
<tr>
<td>Not solved</td>
<td>(++) (S ++ (++) 1*)</td>
</tr>
<tr>
<td>Divisibility by 3</td>
<td>Not solved</td>
</tr>
<tr>
<td>Not solved</td>
<td>(11)<em>0</em></td>
</tr>
<tr>
<td>Divisibility by 5</td>
<td>Not solved</td>
</tr>
<tr>
<td>Not solved</td>
<td>(101)<em>0</em></td>
</tr>
<tr>
<td>Divisibility by 6</td>
<td>Not solved</td>
</tr>
<tr>
<td>Not solved</td>
<td>(11)<em>0</em></td>
</tr>
</tbody>
</table>

**TABLE III**

**V. DISCUSSION**

**A. First Task Set**

We see that for the first task set, the general behavior is similar to behavior found in Dechter et al. [1], and in particular the example of Boolean functions and NAND gates. As in [1], a larger frontier size allows us to solve more problems, and as with Boolean functions and NAND gates, with a frontier size in the thousands we end up solving around half of the tasks. The most noticeable difference for us is the speed of convergence—our examples all converge essentially within one iteration of the E.C. algorithm. This is because there are many fewer useful submodules in these simple regexes than there are for Boolean functions using NAND gates.

When using NAND gates, it is initially difficult to make most of the boolean functions by randomly connecting NAND gates, which accounts for the low number of tasks solved in the first iterations of E.C. [1]. However, in these initial steps, intermediate useful gates such as OR gates and AND gates are learned. These gates are especially useful in the case where the tasks are functions formed by randomly throwing together OR, AND, and NOT, gates. As the algorithm learns these intermediate useful gates one by one, it is able to solve more and more tasks. Ultimately, after the algorithm has learned all of the intermediate useful gates, task success rate plateaus. The same phenomenon seems to occur in the symbolic regression case—much time is spent simply generating things such as (+6) because the algorithm is only given 0 and 1 as terminals—giving the algorithm 0 through 9 would naturally make the learning occur more quickly.

A similar phenomenon is occurring in our case, except that we start with the intermediate useful structures that we will end up using. Concept-wise, the things that we end up learning in the first step of the E.C. algorithm are rudimentary combinations of these structures (such as taking the union of some existing regex structure with 0 or 1, or concatenating it with 0 or 1), and afterward we do not learn anything else. This seems to be not due to the E.C. algorithm, but due to the fact that there really are not many useful submodules that we can learn in this simple case. There are some submodules that are useful, but they do not occur often enough to justify giving them significant weight in the grammar.

**B. Second Task Set**

For the second task set, the algorithm did about as expected, solving all of the easy tasks and a decent number of the medium tasks. Most interestingly, it learned two of the concepts that we hoped it would learn, the dot operator, (0—1), and the non-trivial plus operator, which is written as (S ++ * ) where S is the S combinator, ++ is concatenation, * is the Kleene Star operator. Written out fully, for a string x, this is

\[ S ++ * x = (++ x)(x*) = xx^* = x+ \]

It also learned a concept that we had not anticipated, the doubling operator, which is described in the grammar as (S ++ I). Written out fully, for a string x this is

\[ S ++ I x = (+++ x)(x) = xx \]

As a result of learning these concepts, it was able to learn some of the more difficult medium tasks such as (1(010)*+ (baabaaaaabaa) and binary strings of length exactly two. It also made partial progress towards divisibility by 3, learning the regex ((1(010)*1)*0*)*. This corresponds with numbers of the form

\[ (11)^*0^* = 2^0 \cdot (4^2 - 1) \]

We note that 4^2 — 1 is always divisible by 3, so this gives a subset of numbers that are divisible by 3. A regular expression for divisibility by 3 that closely resembles ours and that can be easily derived by looking at paths in the DFA for divisibility by 3 is ((1(010)*1)*0*)*, or more succinctly ((1(010)*1)*0*)*. For divisibility by 5, we found a similar result, learning the regex (101)*0* which corresponds to numbers of the form 5 · 2^i. A regular expression solving divisibility by 5 is (0(1(1(01))*00))*#. Thus, although the E.C. algorithm was not able to solve all the problems exactly, we really did not expect the E.C. algorithm to be able to solve the most difficult problems, and the E.C. algorithm made decent progress toward a solution.

Whereas divisibility by 3 and 5 is very difficult, the algorithm also failed on tractable tasks such as finding
strings of length exactly 2. It is interesting to note that these strings can be built simply out of concepts learned by the algorithm, as a regular expression that works for strings of length exactly 2 is just
\[(S + I)(S + (I + 0)1^*) = (S + I)(1^*01^*)
\]
\[= (I + 1^*01^*)(I + 01^*) = 1^*01^*01^*\]

In words, we are computing a doubling of strings which use 0 exactly once. However, a theme that will pervade our discussion is that it is difficult for the algorithm to come up with solutions of this form. Most of the tasks that the algorithm solved were solved through a combination of primitives, or through a simple application of a concept, i.e. doubling dot to get strings of length 2. We can see this simple applications of a concept as a one step jump. However, to solve a task such as using 0 exactly twice requires more or less a two step jump - we need to first construct the complicated expression for a string that uses 0 exactly once, and then apply the concept of doubling to it. As applying the doubling concept is a low probability event relative to simpler primitive concepts like concatenation sine not that many regex tasks require doubling for their solutions, that we would construct a solution to using 0 exactly twice is unlikely without having frontier sizes roughly one or two orders or magnitude larger than we currently have.

One thing worth pointing out is that the algorithm is very dependent on the task set and having simple stepping stones to reach solutions. If, knowing that a regular expression for divisibility for 3 were \((1(010)^*00)^*\), we were to include each of the pieces inside this regular expression as its own regular expression, it is possible that we would be able to learn the above regular expression for divisibility for 3. However, a priori building this into the task set defeats the point of trying to learn a regular expression for divisibility for 3.

The fact that changing frontier size does not impacting performance is due precisely to the importance of the choice of task set - 24 of the tasks are rather easily reachable, and the rest cannot be reasonably made out of one step jumps modular subproblems of the first 24, and thus are never reached barring a large frontier size, as estimate previously roughly one to two orders of magnitude larger than currently present. Were we to run a separate simulation for each of those unreached tasks and include each of their subtasks as a task, we could possibly reach them, but as we stated this somewhat defeats the point.

The issue with trying to use the E.C. algorithm to solve this problem is that, as we extend the problem away from the simple examples from our first task or the original paper, the frontier becomes too dispersed. Whereas before for things such as Boolean formulas on three variables, all of the formulas were essentially in a small ball centered around the origin in some high dimensional solution space, more complicated tasks such as divisibility by 3 are much further from the origin in this solution space. As the set of solutions considered in the frontier is always “near” previous solutions, we are never able to move far away from origin in the high dimensional space, and thus with the algorithm as is unless we directly provide a path to divisibility by 3, it will be very difficult if not impossible to solve.

Additionally, not only is it necessary that we have simple stepping stones to reach each solution, we need that the stepping stones reaching to a solution are well-tread, i.e. often used. Our task set, although full of interesting regexes, is lacking in that regard since it is rather diffuse, in that we simply included interesting regexes rather than regexes directed toward a particular goal.

Thus, as we learn concepts, i.e. plus and dot, as these concepts are not directly related to each other they in some sense make our frontier “thinner” - with more concepts, we can explore the combinations with each concepts less. As our task set isn’t unified for a specific goal, our frontier is pulled thin in many directions in the high-dimensional solution space. Although we make our stepping stones such as dot and plus, with regexes there are too many stepping stones all around us, so it is difficult to walk very far from where we start on any path.

VI. CONCLUSION AND FUTURE WORK

Future directions for this research lie in using the algorithm in conjunction with partial matching. If we are only able to use the information of whether we match perfectly or not, it is inevitably difficult to reach complex tasks. Using partial matches in addition to complete matches in determining the next frontier seems like it would be useful, but what seems useful as well is maintaining more information about good matches in the previous frontier. As opposed to the current approach which gets stuck trying to combine a few nand/and/or gates to make an arbitrary 3-sat formula, what seems like a good future course is to, in addition to having the grammar as is, maintain a “best found so far” match for each unsolved task. That way, in essence instead of having frontier determination depend only on the current grammar, frontier determination will depend on the current grammar and the previous best solutions. This is in some way analogous to adding a stochastic hill climbing component to the algorithm.

In the case of having partial solutions, we then need to be very careful about the positive and negative examples that we construct for each regular expression. If we include too many positive examples, the regular expression will be too liberal and favor allowing more strings that it should, and if we include too many negative examples, the regular expression will be too strict and favor too few strings in the language.

As it currently stands the E.C. algorithm can be thought of as relying on having a ladder of submodules that can be formed by combining just a very small number of simpler submodules. Our suggested modifications would serve to allow the jump between these submodules to be larger and let us to do more complex combinations of these submodules to build difficult trees.

From our experience applying the E.C. algorithm to regular expressions, we see that it still has a ways to go to capture
our intuition about concept learning. The issues that we have raised correspond with discrepancies between our intuitive feelings about human learning and the algorithm as it stands. Inability to deal well with partial solutions corresponds with the discrepancy that the unlike the algorithm, people know if they are on the right track toward a solution, and the fact that the algorithm does not focus on a single task at once but rather just enumerates solutions that could solve some task corresponds with the discrepancy that, were a person trying to solve some set of problems, we would look at the problems individually and try to solve each on, rather than throwing together pieces and seeing if they solve any task. Nonetheless, that the algorithm does work and give interesting results is certainly a promising sign that it has the potential if modified correctly to be able to capture some amount of how we perceive humans to learn.

We hope in the future to be able to move on from regular expressions to subjects in computer vision, the original goal of this project but far beyond the grasp of the algorithm as it stand now. We would begin on the pixel level and first teach our object how to recognize simple objects such as generic boxes, eventually going on to real-life objects such as airplanes. During this process, the algorithm would first learn low level concepts such as edge detectors and blob detectors, then progressively more interesting concepts that extract higher level features of objects and eventually concepts corresponding to object classifiers able to identify objects such as dogs, cats, and sheep (baaaabaaaaaba).

CONTRIBUTIONS

Both Liu contributed roughly equally to the project. Andrew Liu was responsible for the idea for the project. Andrew Liu wrote most of the code to apply the EC algorithm to regular expressions, and David Liu wrote code adapting the tree structure of programs to work for regular expressions. David Liu wrote up the test cases and ran simulations.

ACKNOWLEDGMENT

The authors would like to thank Eyal Dechter and Kevin Ellis for software for running the E.C. algorithm and for their conversations regarding potential applications of the E.C. algorithm.

REFERENCES