Program Boosting or Crowd-Sourcing for Correctness

Robert Cochran
UNC, Chapel Hill

Loris D'Antoni
University of Pennsylvania

Benjamin Livshits
Microsoft Research

Abstract

A great deal of effort has been spent on both trying to specify software requirements and on ensuring that software actually matches these requirements. A wide range of techniques that includes theorem proving, model checking, type-based analysis, static analysis, runtime monitoring, and the like have been proposed. However, in many areas adoption of these techniques remains spotty. In fact, *obtaining* a specification or a precise notion of correctness is in many cases quite elusive.

In this paper we investigate an approach we call *program boosting*, which involves crowd-sourcing imperfect solutions to a difficult programming problem from developers and then *blending* these programs together in a way that improves their correctness.

We show how interesting and highly non-trivial tasks such as writing regular exepressions for URLs or email addresses can be effectively crowd-sourced. We demonstrate that carefully blending the crowd-sourced results together consistently produces a *boost*, yielding results that are better than any of the starting programs. Our experiments on 465 program pairs show consistent boosts in accuracy and demonstrate that program boosting can be performed at a relatively modest monetary cost.

1. Introduction

Everyday programming involves solving a sequence of many smaller tasks. Some of these tasks are fairly routine; others are surprisingly challenging. Examples of challenging self-containing tasks are coming up with a regular expression to recognize email addresses or sanitizing an input string to avoid SQL injection attacks. Both of these tasks are easy to describe to most developers succinctly, yet both are surprisingly difficult to get right, i.e. to implement, properly addressing all the tricky corner cases. Furthermore, there is room for ambiguity in both tasks: for example, even seasoned developers can disagree as to whether `john + doe@acm.org` or `john..doe.@acm.com` is a valid email address or whether removing all characters outside of the $\texttt{a} - \texttt{zA} - \texttt{Z}$ set is a valid sanitization strategy for SQL injections. These examples illustrate several important points about these tricky programming tasks: they are typically *under-specified*, they may not have absolute consensus on what solution is correct, moreover, different people may get different parts of the problem wrong.

What if we could crowd-source the answer to these tricky tasks? We would be able to describe the task in question in English, with all its ambiguities and under-specified corner cases. We would subsequently use the "wisdom of the crowds" to arrive at the answer, without knowing what the proper answer might be, *a priori*, but perhaps armed with positive and negative examples. This paper explores this deceptively simple idea.

1.1 In Search of Perfect URL Validation

In December 2010, Mathias Bynens, a freelance web developer from Belgium set up a page to collect possible regular expressions for matching URLs. URL matching turns out to be a surprisingly challenging problem. To help with testing the regular expressions, Mathias posted a collection of both positive and negatives examples, that is, strings that should be accepted as proper URLs or rejected. While some example URLs are as simple as `http://foo.com/blah_blah`, others are considerably more complex and require the knowledge of allowed protocols (`ftps://foo.bar/` should be rejected) or the range of numbers in IP addresses (which is why `http://123.123.123` should be rejected).

Mathias posted this challenge to his followers of Twitter. Soon, a total of 12 responses were collected, as summarized in Figure 1. Note that the majority of responses were incorrect at least in part: while all regular expressions correctly captured simple URLs such as `http://www.cnn.com`, they often would disagree on some of the more subtle inputs. Only one participant with a Twitter handle of `@diegoperini` managed to get all the answers right[1]. `@stephenhay` came close, getting all positive inputs right, but missing some of the negative inputs.

Key Insight: While a detailed analysis of this experiment is available at `http://mathiasbynens.be/demo/URL-regex`, a few things are clear:

- The problem of writing a regular expression for URLs is surprisingly complex; moreover, it is a problem where it is easy to get started and get to a certain level or accuracy, but getting to perfect precision on the training set is very tough;
- potential answers provided by developers range in length (median values 38–1,347) and accuracy (.56–1), a great deal, as measured on a training set. Note that the most accurate answer provided by `diegoperini` is in this case *not* the longest;
- developers get *different* portions of the answer wrong, as can be seen from the results table at the URL above;
- cleverly combining (or blending) partially incorrect answers may yield a correct one.

We experienced a similar situation when trying to crowd-source *security sanitizers* [13]. Sanitizers are short self-contained string-manipulation routines that are crucial in preventing cross-site script attacks in web applications. As part of our experimentation, we asked developers on oDesk

[1] The full regex from `@diegoperini` can be obtained from `https://gist.github.com/dperini/729294`.

Regex source	Regex **length**	**True positive**	**True negative**	**Overall accuracy**
Spoon Library	979	.39	.39	.67
@krijnhoetmer	115	.78	.78	.59
@gruber	71	.97	.97	.65
@gruber v2	218	1.00	1.00	.65
@cowboy	1,241	1.00	1.00	.56
Jeffrey Friedl	241	.56	.56	.59
@mattfarina	287	.72	.72	.57
@stephenhay	38	1.00	1.00	.81
@scottgonzales	1,347	1.00	1.00	.56
@rodneyrehm	109	.83	.83	.59
@imme_emosol	54	.97	.97	.85
@diegoperini*	**502**	**1.00**	**1.00**	**1.00**

Figure 1: Regular expressions for URLs obtained from `http://mathiasbynens.be/demo/url-regex`.

to implement sanitizers. We then proceeded to test the obtained sanitizers against a well known test suite (cross-site scripting cheat sheet) [27]. All seven implementations we considered correctly escape angle brackets. However, we found that some of the implementations do not escape the string `&#`, potentially yielding an attack. Most developers made mistake when it came to corner cases. Only one implementation of `HTMLEncode` made it impossible for all of the strings our test set from appearing in its output. Once again, this is a problem for which is is easy to get partially-correct answers but quite difficult to get a correct solution, due to tricky corner cases.

Our experimental results in this paper focus on the regular expression domain. However, we feel that a wide range of problems falls into the category outline above. For example, imagine building a better renderer for tricky HTML pages by "blending" the result of renderers of popular browsers such as IE, Firefox, and Chrome. These browsers will "agree" on easy-to-parse web pages, but will likely differ on tricky and partially broken ones. One option for the resulting "hyper-browser" is to vote among the underlying browsers to decide how to display the more tricky pages.

1.2 Contributions

Our paper makes the following contributions:

- We propose a technique we dub *program boosting*. Program boosting is a semi-automatic program generation or synthesis technique that uses a set of initial crowd-sourced programs and combining (or blends) them to provide a better result, according to a fitness function. While other formulations are possible, we primarily focus on improving the accuracy of blended programs on a training set of positive and negative examples. The training set is *evolved* as a result of automatically refining answers to corner cases by asking the crowd to provide disambiguations.
- We show how to implement our approach for programs or tasks that can be expressed via regular expressions. Our technique is a specific version of genetic programming with custom-designed crossover (*shuffle*) and *mutation* operations.
- We represent regular expressions using *Symbolic Finite Automata* (SFAs), which enable succinct representation while supporting large alphabets and naturally representating the behaviour of regular expressions. We also adapt classical algorithms, such as *string-to-language* edit distance, to the symbolic setting.
- We evaluate program boosting techniques on four case studies. In our experiments on a set of 465 pairs of regular expression programs, we observe an average boost in accuracy of 0.1625%. The boosting effect is *consistent* across the tasks and sources of initial regular expressions, which enhances our belief in the generality of our approach. The average time to run the boosting process is 45 minutes. The cost of pairwise boosting average from 41¢ to $3, depending on the complexity of the underlying task.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 gives background on both crowd-sourcing and how to use it for programming tasks. Section 3 gives an outline of our approach of using two crowds in tandem to generate programs. Section 4 gives the detains of our implementation based on symbolic finite automata or SFAs. Section 5 provides an experimental evaluation in the context of four case studies. Section 6 contains a discussion of some of the outstanding challenges we see for future research. Finally, Sections 7 and 8 describe related work and conclude.

2. Background

In the last several years we have seen a rise in the use of crowd-sourcing for both general tasks that do not require special skills (recognize if there is a cat in the picture, reformat text data, correct grammar in a sentence) and skilled tasks such as providing book illustrations or graphic design assignments on request or perhaps writing short descriptions of products.

A good example of a crowd-sourcing site for unskilled work is Amazon's Mechanical Turk (frequently abbreviated as *mturk*); oDesk is another widely-used platform, this one primarily used for skilled tasks. Both Mechanical Turk and oDesk can be used for sourcing programming tasks, although neither is specialized for that. Note that one can consider StackOverflow and other similar programming assistance sites as an informal type of crowd-sourcing. Indeed, these sites are so good at providing *ingredients* for solving difficult programming problems that some developers routinely keep StackOverflow open in their browsers as they code.

So far, we have only seen a single dedicated platform for crowd-sourcing programming, Bountify (`http://bountify.co`). It allows people to post programming tasks, some involving writing new code from scratch (*Write a JavaScript function to generate multiple shades of a given color*), and others involving fixing bugs in existing code (*why does my HTML table not look the way I expect and how should I tweak my CSS to make it look right?*). These programming tasks generally are not overly time-consuming; a typical task pays about $5. Responses are posted publicly, leading to other developers learning from partial answers. Finally, the poster decides which developer(s) to award the bounty to.

Note that interactive crowd-sourcing is not the only source of code. Indeed, one can easily use a code search engine to find the insight one is looking for in open-source projects. Searching for terms such as `url_regex` using a dedicated code engine is will yield some possible regular expressions for URL filtering as well, as will exploring a programming advice site such as StackOverflow.

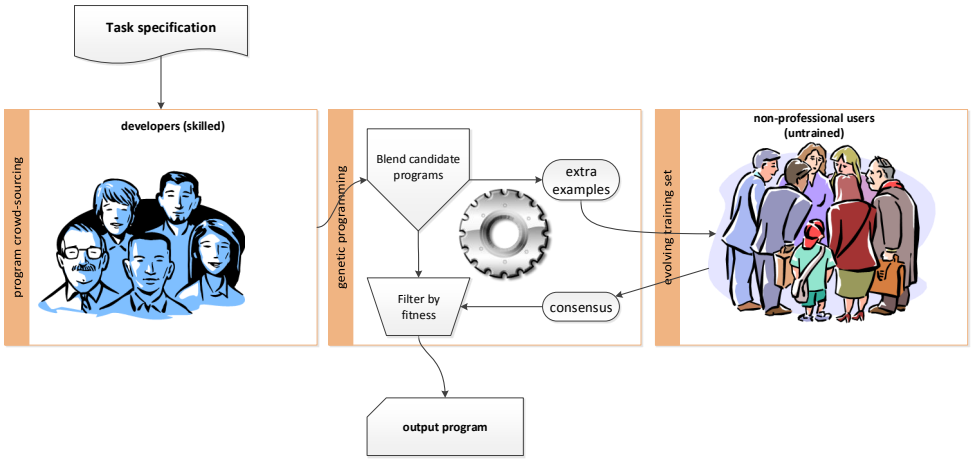


Figure 2: System architecture.

3. Overview

In this section, we provide an outline of our approach to program boosting. Note that our approach is general in that it applies to different kinds of programs. However, it relies on two operations that may be familiar to the reader from genetic programming literature: shuffle (also called *crossover*) and mutation. Section 4 discusses how we implement these for regular expressions. However, in many ways, there is a lot of ingenuity that goes into providing sensible implementations of these operations that do not greatly increase the size of resulting programs.

Difficult programming tasks: In this paper, we focus on a specific class of difficult programming problems, as exemplified by coming up with tricky regular expressions or sanitizers for security. To summarize, these difficult programming tasks share the following qualities:

- their specification is provided as text and is open to interpretation;
- virtually all developers get obvious cases right;
- virtually all developers get *some* corner cases wrong;
- frequently, different developers often get *different* corner cases wrong;

The hope is that piecing together diffferent solutions will yield a solution that is "more correct".

Binary classification tasks: for practical reasons in this paper we focus on programs that

- consume a single input
- produce a binary (yes/no) output;
- for any input, a non-specialist computer user can decide if the answer for it should be a *yes* or a *no*.

Our observation is that while the generic crowd is not going to help us to source programs, they will be able to *recognize* correct or incorrect program behaviors. By way of analogy, while a layperson may not be able to write a computer vision program that recognizes the presence of a cat in an image, humans are remarkably good at recognizing whether a given picture has a cat in it. This two-crowd approach helps us to both *collect* or *source* candidate programs and to *refine* them by asking the untrained crowd about the correct behavior on questionable cases.

Our approach can be expanded beyond these restrictions, but our implementation is greatly simplified by these assumptions.

Architecture: Figure 2 shows the architecture of our system. To crowd-source a solution to the specified task, we take advantage of two crowds, the *developer crowd* and the *user crowd*; the former contains developers for hire, typically skilled in one or more languages such as Java and C++, the latter consists of laypeople.

3.1 Iterative Genetic Algorithms

Figure 3 shows our program boosting algorithm as pseudocode. Let Σ be the set of all programs and Φ be the set of all examples. In every generation, we update the set of currently considered programs $\sigma \subset \Sigma$ and the set of current examples $\phi \subset \Phi$.

Note that the algorithm is iterative in nature: the process of boosting proceeds in *generations*, similar to the way genetic programming is typically implemented. The overall goal is to find a program with the best fitness in Σ. At each generation, new examples in Φ are produced and sent to the crowd to obtain consensus. The algorithm is parametrized as follows:

- $\sigma \subset \Sigma$ is the initial set of programs;
- $\phi \subset \Phi$ is the initial set of positive and negative examples;
- $\beta : \Sigma \times \Sigma \to 2^{\Sigma}$ is the crossover shuffle function that takes two programs and produces a set of possible shuffles;
- $\mu : \Sigma \to 2^{\Sigma}$ is the mutation function that produces a set of possible shuffles;
- $\delta : \Sigma \times 2^{\Phi} \to 2^{\Phi}$ generates new training examples;
- $\eta : \Sigma \to \mathbb{N}$ is the fitness function;
- $\theta \in \mathbb{N}$ is the budget for Mechanical Turk crowd-sourcing;

In Section 4 we show how to implement operations that correspond to functions β, μ, δ, and η for regular expressions using SFAs. Note that in practice in the interest of completing faster we usually limit the number of iterations to a set limit such as 10.

3.2 Optimizations

Our implementation benefits greatly from parallelism. In particular, we make the two loops on lines 6 and 12 of

1: **Input:** Programs σ, examples ϕ, shuffling function β, mutation function μ, example generator δ, fitness function η, budget θ
2: **Output:** Boosted program
3: **function** $Boost(\langle\sigma, \epsilon\rangle, \beta, \mu, \delta, \eta, \theta)$
4: **while** $(\hat{\eta} < 1.0 \wedge \theta > 0)$ **do** ▷ Til perfect or no money
5: $\varphi = \emptyset$ ▷ New examples for this generation
6: **for all** $\langle\sigma_i, \sigma_j\rangle \in FindShuffleCandidates(\sigma)$ **do**
7: **for all** $\sigma' \in \beta(\langle\sigma_i, \sigma_j\rangle)$ **do** ▷ Shuffle σ_i and σ_j
8: $\varphi = \varphi \cup \delta(\sigma', \phi)$ ▷ Generate new examples
9: $\sigma = \sigma \cup \{\sigma'\}$ ▷ Add this candidate to σ
10: **end for**
11: **end for**
12: **for all** $\langle\sigma_i\rangle \in FindMutationCandidates(\sigma)$ **do**
13: **for all** $\sigma' = \mu(\sigma_i)$ **do** ▷ Mutate σ_i
14: $\varphi = \varphi \cup \delta(\sigma', \phi)$ ▷ Generate new examples
15: $\sigma = \sigma \cup \{\sigma'\}$ ▷ Add this candidate to σ
16: **end for**
17: **end for**
▷ Get consensus on these new examples via mturk
18: $\langle\phi_\varphi, \theta\rangle = GetConsensus(\varphi, \theta)$ ▷ and update budget
19: $\phi = \phi \cup \phi_\varphi$ ▷ Add the newly acquired examples
20: $\sigma = Filter(\sigma)$ ▷ Update candidates
21: $\langle\hat{\sigma}\rangle = GetBestFitness(\sigma, \eta)$
22: **end while**
23: **return** $\hat{\sigma}$ ▷ Return program with best fitness
24: **end function**

Figure 3: Program boosting implemented as an iterative genetic programming algorithm.

the algorithm parallel. While we need to be careful in our implementation to avoid shared state, this relatively simple change ultimately leads to near-full utilization on a machine with 8 or 16 cores.

Unfortunately, our call-outs to the crowd on line 16 to get the consensus are synchronous. This does lead to an end-to-end slowdown in practice, as crowd workers tend to have a latency associated with finding and starting new tasks, even if their throughput is quite high. In the future, we envision a slightly more streamlined architecture where allowing speculative exploration of the space of programs may allow us to call crowd calls asynchronously.

4. Regular Expression Manipulation

We first describe Symbolic Finite Automata (SFA) and motivate their choice as an alternative to classical automata. Next, we present algorithms for shuffling, mutation, and examples generation, used in the algorithm in Figure 3.

4.1 Symbolic Finite Automata

While regular expressions are succinct and relatively easy to understand, they are not easy to manipulate algebraically. In particular, there is not direct algorithm for complementing or intersecting them. Because of this, we opt for finite automata instead. Classic deterministic finite automata (DFAs) enjoy many closure properties and friendly complexities. However, each DFA transition can only carry one element of the alphabet, causing the number of transitions in the DFA to be proportional to the size of the alphabet. When the alphabet is large (UTF16 has 2^{16} elements) this representation becomes impractical.

Symbolic Finite Automata (SFAs) [31] extend classical automata with symbolic alphabets. In an SFA each edge is labeled with a predicate, rather than a single input character. This allows the automaton to represent multiple concrete transitions succinctly. For example, in the SFA of Figure 4 the transition from state 10 to state 11 is labeled with the predicate `[~#--\/?\s]`. Because of the size of the UTF16 set, this transition in classical automata would represented by thousands of concrete transitions.

Before defining SFAs we first need to introduce several preliminary concepts. Since the guards of SFA transitions are predicates, operations such as automata intersection needs to "manipulate" such predicates. Let's consider the problem of intersecting two classical DFAs. In classical automata intersection, if the two DFAs respectively have transitions (p, a, p') and (q, a, q') the intersected DFA (also called the product) will have a transition $(\langle p, q\rangle, a, \langle p', q'\rangle)$. Now if we want to intersect two SFAs this simple synchronization would not work. If two SFAs respectively have transitions (p, φ, p') and (q, ψ, q') from A_2 (where φ and ψ are predicates), the intersected DFA will need to synchronize the two transitions only on the values for which they are both "triggered", therefore the new transition will be $(\langle p, q\rangle, \varphi \wedge \psi, \langle p', q'\rangle)$. Moreover if the predicate $\varphi \wedge \psi$ is not satisfiable (does not have any character triggering it), this transition should be removed. These examples shows how the set of predicates used in the SFA should at least be closed under $\wedge$ (conjunction), and the underlying theory should be decidable (we can check for satisfiability). It can be shown that in general in order to achieve the classical closure properties of regular language the set of predicates must also be closed under negation.

Definition 1. *A Boolean algebra B has components $(\mathcal{D}_B, P_B, f, \bot, \top, \wedge, \neg)$. $\mathcal{D}_B$ is a set of domain elements, and P_B is a set of predicates closed under Boolean connectives $\wedge, \neg$, and $\bot, \top \in P_B$. The denotation function $f : P_B \mapsto 2^{\mathcal{D}}_B$ is such that $f(\top) = \mathcal{D}$, $f(\bot) = \emptyset$, $f(\varphi \wedge \psi) = f(\varphi) \cap f(\psi)$, and $f(\neg\varphi) = \mathcal{D} \setminus f(\varphi)$. For $\varphi \in P_B$, we write $IsSat(\varphi)$ when $f(\varphi) \neq \emptyset$,and say that φ is* satisfiable*. B is decidable if $IsSat$ is decidable.*

We can now define symbolic finite automata.

Definition 2. *A Symbolic Finite Automaton, SFA, A is a tuple (B, Q, q_0, F, δ) where B is a decidable Boolean algebra, called the* alphabet*, Q is a finite set of* states*, $q_0 \in Q$ is the* initial state*, $F \subseteq Q$ is the set of* final states*, and $\delta \subseteq Q \times P_B \times Q$ is a finite set of moves or transitions.*

In the following definitions we refer to a generic SFA A. A is deterministic if for every state $q \in Q$, there do not exists two distinct transitions (q, φ, q_1), and (q, ψ, q_2) in δ, such that $IsSat(\varphi \wedge \psi)$. If A is deterministic, we define the reflexive-transitive closure of δ as, for all $a \in \mathcal{D}$ and $s \in \mathcal{D}^*$, $\delta^*(Q, as) = \delta(Q', s)$, if $\forall q' \in Q \exists(q, \varphi, q') \in \delta$ such that $q \in Q$, and $a \in f(\varphi)$, and $\delta^*(Q, \varepsilon) = Q$. The language accepted by A is $L(A) = \{s \mid \delta^*(\{q_0\}, s) \subseteq F\}$.

BDD algebra: We describe the Boolean algebra of BDDs, which is used in this paper to model regular expression characters. A BDD algebra 2^{bv_k} is the powerset algebra whose domain is the finite set bv_k, for some $k > 0$, consisting of all nonnegative integers less than 2^k, or equivalently, all k-bit bit-vectors. A predicate is represented by a BDD [30] of depth k. The variable order of the BDD is the reverse bit order of the binary representation of a number, in particular, the most significant bit has the lowest ordinal. The Boolean operations correspond directly to the BDD operations, $\bot$ is the BDD representing the empty set. The denotation

$f(\varphi)$ of a BDD φ is the set of all integers n such that a binary representation of n corresponds to a solution of φ. For example, in the case of URLs over the alphabet UTF16, we use the BDD algebra $2^{bv_{16}}$ to naturally represent sets of UTF16 characters (bit-vectors). We consider the SFA and BDD implementations from the library [31].

4.2 Fitness Computation

Recall that as part of the genetic programming approach employed in program boosting, we need to be able to assess the fitness of a particular program. For regular expressions, this amounts to calculating the *accuracy* on a training set. The process of fitness calculation can by itself be quite time-consuming. This is because running a large set of examples and counting how many of them are accepted correctly by each produced SFA is a process that scales quite poorly when we consider thousands of SFAs and hundreds of examples. Instead, we "melt" our positive and negative examples into SFAs P and N, which represents the languages of all positive and all negative examples, respectively. For any SFA A, we then compute the cardinality of intersection sets $A \cap P$ and $N \setminus A$, both of which can be computed fast. The accuracy can be then computed as

$$\frac{|A \cap P| + |N \setminus A|}{|P| + |N|}$$

A challenge inherent with our refinement technique is that our evolved example set can greatly deviate from the initial gold set. While imperfect, we still want to treat the gold set as a more reliable source of truth; to this end, we use weighting to give the gold set a higher weight in the overall fitness calculation. In our experimental evaluation, we get reliably good results if we set gold:evolved weights to 9:1.

4.3 Shuffles

A shuffle (crossover) operation interleaves two SFAs into a single SFA that "combines" their behaviours. An example of this operation is illustrated on the right. Given two SFAs A and B, the shuffling algorithm redirects two transitions, one from A to B, and one from B to A. The goal of such operation is that of using a component of B inside A.

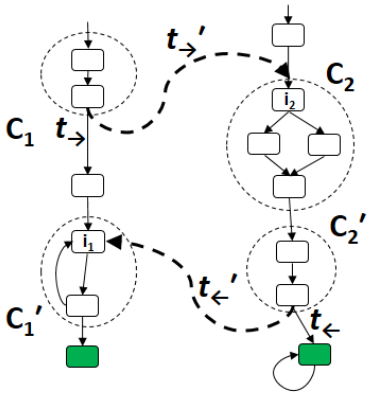
An SFA can have many transitions and trying all the possible shuffles can be impractical. Concretely, if A has n_1 states and m_1 transitions, and B has n_2 states and m_2 transitions, then there will be $O(n_1 n_2 m_1 m_2)$ possible shuffles. Checking fitness for this many SFAs would not scale. The shuffling algorithm is shown in Figure 5.

We devise several heuristics that try to mitigate such blowup by limiting the number of possible shuffles. The first technique we use is to guarantee that: 1) if we leave A by redirecting a transition (q, φ, q_1), and come back on state q_2, then q_2 is reachable from q_1, but different from it (we write $q_1 \prec q_2$), and 2) if we reach B in a state p_1, and we leave it by redirecting a transition (p_2, φ, p), then p_2 is reachable from p_1 (we write $p_1 \preceq p_2$). With these two approaches, we avoid generating shuffles for which the redirected transition would not lead to any final state.

The next heuristics limit the number of "interesting" edges and states to be used in the algorithm by grouping multiple states into single component and only considering those edges that travel from one component to another one. In the algorithm in Figure 5, the reachability relation $\prec$ is naturally extended to components (set of states). The function COMPONENTS returns the set of state components computed using one of the heuristics described below.

Strongly-connected components: Our first strategy collapses states that belong to a single strongly connected component (SCCs). SCCs are easy to compute and often capture interesting blocks of the SFA.

Collapsing stretches: In several cases SCCs do not collapse enough states. Consider the SFA in Figure 4. In this example, the only SCC with more than one state is the set {11–12}. Moreover, most of the phone number regexes are represented by acyclic SFAs causing the SCCs to be completely ineffective. To address this limitation we introduce a collapsing strategy for "stretches". A *stretch* is a maximal connected acyclic subgraph where every node has degree smaller or equal to 2. In the SFA on the right $\{1, 3, 5\}$, $\{2, 4\}$, and $\{9, 10\}$ are stretches.



Single-entry, single-exit components: Even using stretches the collapsing is often ineffective. Consider again the SFA on the right. The set of nodes $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ looks like it should be treated as a single component, since it has a single entry point, and a single exit point, however it is not a stretch. This component clearly captures an independent part of the regex which accepts the correct protocols of a URL. Such components are characterized by the following features:

1. it is a connected direct acyclic subgraph,
2. it has a single entry and exit point,
3. it does not start or end with a stretch, and
4. it is *maximal*: it is not contained in a bigger component with properties 1–3.

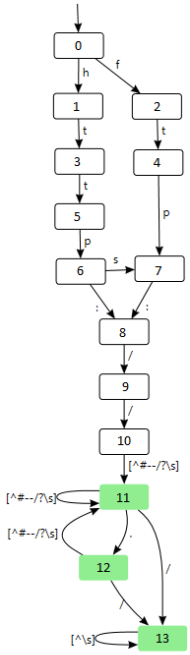


Figure 4: Identifying components.

Such components can be computed in linear time by using a variation of depth-first search starting in each node with in-degree smaller than 1. The requirement 4) is achieved by considering the nodes in topological sort (since SCCs are already collapsed the induced graph is acyclic). Since this technique is generally more effective than stretches, we use it before the stretch collapsing.

In the SFA on the right, the final components will then be: $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $\{9, 10\}$, $\{11, 12\}$, and $\{13\}$. Finally, if A has c_1 components and t_1 transitions between different different components, and B has c_2 components and t_2 transitions between different different components, then there will

Input: SFAs $A_1 = (Q_1, q_0^1, F_1, \delta_1)$, $A_2 = (Q_2, q_0^2, F_2, \delta_2)$
Output: All shuffles of A_1 and A_2

function SHUFFLES(A_1, A_2)
 C_1 := COMPONENTS(A_1)
 C_2 := COMPONENTS(A_2)
 for all $c_1 \in C_1$ **do**
  $C_1' := \{c_1' \mid c_2 \prec_{A_1} c_1'\}$
  for all $t_\rightarrow = (p_1, \varphi, p_2) \in$ EXIT_MOVES(c_1, A_1) **do**
   for all $c_2 \in C_2$ **do**
    for all $i_2 \in$ ENTRY_STATES(c_2, A_2) **do**
     $C_2' := \{c_2' \mid c_2 \preceq_{A_2} c_2'\}$
     for all $c_2' \in C_2'$ **do**
      for all $t_\leftarrow = (q_1, \varphi, q_2) \in$ EXIT_MOVES(c_2', A_2) **do**
       for all $c_1' \in C_1'$ **do**
        for all $i_1 \in$ ENTRY_STATES(c_1', A_1) **do**
         $t'_\rightarrow := (p_1, \varphi, i_2)$, $t'_\leftarrow := (q_1, \varphi, i_1)$
         $\delta_{new} := \delta_1 \cup \delta_2 \setminus \{t_\rightarrow, t_\leftarrow\} \cup \{t_\rightarrow, t'_\leftarrow\}$
         yield return $(Q_1 \cup Q_2, q_0^1, F_1 \cup F_2, \delta_{new})$
end function
function EXIT_MOVES(c, A)
 return $\{(p, \varphi, q) \in \delta_A \mid p \in c \wedge p' \notin c\}$
end function
function ENTRY_STATES(c, A)
 return $\{q \in Q_A \mid \exists (p, \varphi, q) \in \delta_A . p \notin c \wedge p' \in c \vee q = q_0^A\}$
end function

Figure 5: Shuffle algorithm.

be $O(c_1 c_2 t_1 t_2)$ possible shuffles. In practice this number is much smaller than $O(n_1 n_2 m_1 m_2)$.

One-way shuffles: One way shuffles are a variant of those described above in which we redirect one edge from A to B but we do not come back to A on any edge. This roughly corresponds to removing lines 11–15 from the algorithm in Figure 5. If A has t_1 transitions between different different components, and B has c_2 components, then there will be $O(c_2 t_1)$ possible one-way shuffles.

4.4 Mutations

In its classical definition a mutation operator alters one or more values of the input producing a mutation. In our setting, the inputs have too many values to be altered (every transition can carry 2^{16} elements), and a "blind" approach would produce too many mutations. Instead we consider a guided approach, in which mutations take as input a SFA A and a counterexample s, such that s is incorrectly classified by A (s is in the target language but not in $L(A)$, or s is not in the target language but it is in $L(A)$). Using this extra bit of information we mutate A only in those ways that will cause s to be correctly classified. The intuition behind such operations is to perform a minimal syntactical change in order to correctly classify the counterexample.

Let L_T be the target language. Based on whether the counterexample s belongs or not to L_T, we devise two types of mutations.

Diminishing mutations: Given a string $s \notin L_T$ and a SFA A such that $s \in L(A)$ generates a SFA A', such that $L(A') \subseteq L(A)$ and $s \notin L(A')$.

Given a string $s = a_1 \dots a_n$ that is accepted by A, the algorithm finds a transition (q, φ, q') that is traversed using the input character a_i (for some i) when reading s and either removes the whole transition, or simply shrinks the guard to $\varphi \wedge \neg a_i$ disallowing symbol a_i. Given a string of length k, this mutation can generate at most $2k$ mutated SFAs. When there exists a state $q \in F$ such that $\delta^*(q_0, s) = q$ we also generate output $A = (q_0, Q, F \setminus \{q\}, \delta)$, in which the input SFA is mutated by removing a final state.

Augmenting mutations: Given a string $s \in L_T$ and a SFA A such that $s \notin L(A)$ generates a SFA A', such that $L(A) \subseteq L(A')$ and $s \in L(A')$.

The input $A = (q_0, Q, F, \delta)$ is a partial SFA (some nodes have undefined transitions). Given a string $s = a_1 \dots a_n$ that is not accepted by A, the algorithms finds a state q such that, for some i, $\delta^*(q_0, a_1 \dots a_i) = q$, and a state q' such that, for some $j > i$, $\delta^*(q', a_j \dots a_n) \in F$. Next, it adds a path from q to q' on the string $a_{mid} = a_{i+1} \dots a_{j-1}$. This is done by adding $|a_{mid}| - 1$ extra states. It is easy to show that the string s is now accepted by the mutated SFA A'. Given a string of length k and a SFA A with n states this mutation can generate at most nk^2 mutated SFAs. When there exists a state q such that $\delta^*(q_0, s) = q$ we also output $A = (q_0, Q, F \cup \{q\}, \delta)$, in which the input SFA is mutated by adding a new final state.

4.5 Example Generation

Generating one string is often not enough to "characterize" the language of an SFA. For each SFA $A = (Q, q_0, F, \delta)$, we generate a set of strings S, such that for every state $q \in Q$, there exists a string $s = a_1 \dots a_n \in L(A)$, such that for some i $\delta^*(q_0, a_1 \dots a_i) = q$. Informally, we want to generate a set of strings covering all the states in the SFA. This technique is motivated by the fact that we keep the SFA minimal, and in a minimal SFA each state corresponds to a different equivalence class of strings. The example generation algorithm is described in Figure 7 and it terminates in at most $|Q|$ iterations. The algorithm simply generates a new string at every iteration, which is forced to cover at least one state which hasn't been covered yet.

Unfortunately, this naïve approach tends to generate strings that look "random" causing untrained crowd workers to be overly conservative by classifying them as negative examples, even when they are not. For example, we

`https://f.o/..Q/`	`https://f68.ug.dk.it.no.fm`
`ftp://1.bd:9/:44ZW1`	`ftp://hz8.bh8.fzpd85.frn7..`
`http://h:68576/:X`	`ftp://i4.ncm2.lkxp.r9..:5811`
`http://n.ytnsw.yt.ee8`	`ftp://bi.mt..:349/`

(a) Random examples

`ÃŕÂżÂ£Whttp://youtu:e.com`	`http://y_:outube.com`
`0.http//youtu:e.com`	`ht:tpWWWÃŹÂă://youtube.com`
`h_ttp://youtu:e.com`	`ht:tpWWW0://youtube.com`
`WWhttp://youtu:e.com`	`ht:tpWWWÃŹÂă0://youtube.com`
`WWWhttp://youtu:e.com`	`ht:tpWWW00://youtube.com`
`WWWÃŹÂăhttp://youtu:e.com`	`http://yo:u((t))ube.com`
`WWW0http://youtu:e.com`	`ht:tpWWWÃŹÂă00://youtube.com`
`http:/()/youtube.com`	`ht:tpWWW000://youtube.com`
`WWWÃŹÂă0http://youtu:e.com`	`http://yo:ut((u))be.com`
`WWW00http://youtu:e.com`	`http://%@youtube.com`
`http:/()()/youtube.com`	`http://[]/youtube.com`
`http:/(())youtube.com`	`http://!@youtube.com`
`http:/()(())youtube.com`	`http://%y@outube.com`
`WWWÃŹÂă00http://youtu:e.com`	`http://%00youtube.com`
`WWW000http://youtu:e.com`	`http://%00youtube.c:0`

(b) Examples generated with the edit distance approach.

Figure 6: Two approaches to examples generation.

```
1: Input: SFA A = (Q, q, F, δ) and "seed" string w
2: Output: Set of new training strings
3: function COVER(A, w)
4:     C := Q
5:     while C ≠ ∅ do
6:         q := REMOVE_FIRST(Q)
7:         A' := SFA_PASSING(A, q)
8:         s := CLOSEST_STRING(A, w)
9:         C := C \ STATES_COVERED(s, A)
10:        yield return s
11:    end while
12: end function
13: // Language of all strings in A passing through state q
14: function SFA_PASSING(A, q)
15:    return CONCATENATE((Q, q0, {q}, δ), (Q, q, F, δ))
16: end function
```

Figure 7: Example generation.

have obverved a strong negativity bias towards strings that use non-Latin characters. In the case of URLs, we often get strings containing upper Unicode elements such as Chinese characters, which look unfamiliar to US-based workers. Ideally, we would like to generate strings that *look* as close to normal URLs as possible.

Edit distance: We solve this problem by using the knowledge encoded in our training set of inputs. We choose to look for strings in A that are close to some example string e. We can formalize this notion of closeness by using the classical metric of string edit distance. Formally, an edit is a

character insertion: given a string $a_1 \dots a_n$, and a symbol $b \in \Sigma$, create $a_1 \dots a_i b a_{i+1} \dots a_n$,

character deletion: given a string $a_1 \dots a_n$, create $a_1 \dots a_{i-1} a_{i+1} \dots a_n$, or

character replacement: given a string $a_1 \dots a_n$, and a symbol $b \in \Sigma$ different from a_i, create $a_1 \dots a_{i-1} b a_{i+1} \dots a_n$.

The edit distance between two strings s and s', $ED(s, s')$ is the minimum number of edits that transforms s into s'. Next, we can reformulate the problem as: given a string e (from the training input), and an SFA A, find a string $s \in L(A)$ such that the edit distance between e and s is minimal. Let $witnesses(A, e) = \{s \mid \forall t \in L(A). ED(s, e) \leq ED(t, e)\}$ be the set of strings in $L(A)$ at minimal edit distance from e. Our algorithm will have to output one such string. We use the algorithm in [32] to compute the minimum edit distance, and we modify it to actually generate the witness string.[2] The algorithm has complexity $O(|s|n^2)$, where n is the number of states in the SFA.

Example generation in action: Figure 6a shows some examples of randomly generated strings, and Figure 6b several strings generated using the edit distance technique. Clearly, the second set looks less "random" and less intimidating to an average user.

[2]The algorithm in [32] actually has a mistake in the base case of the dynamic program. When computing the value of $V(T, S, c)$ in page 3, the "otherwise" case does not take into account the case in which $T = S$ and T has a self loop on character c. We fix the definition in our implementation.

Task	Specification	Examples +	Examples −
Phone numbers	`https://bountify.co/5b`	5	4
Dates	`https://bountify.co/5v`	9	9
Emails	`https://bountify.co/5c`	10	7
URLs	`https://bountify.co/5f`	14	9

Figure 8: Specifications provided to Bountify workers.

5. Experimental Evaluation

Broadly speaking, we are interested in the following measures of success for evaluating our program boosting approach:

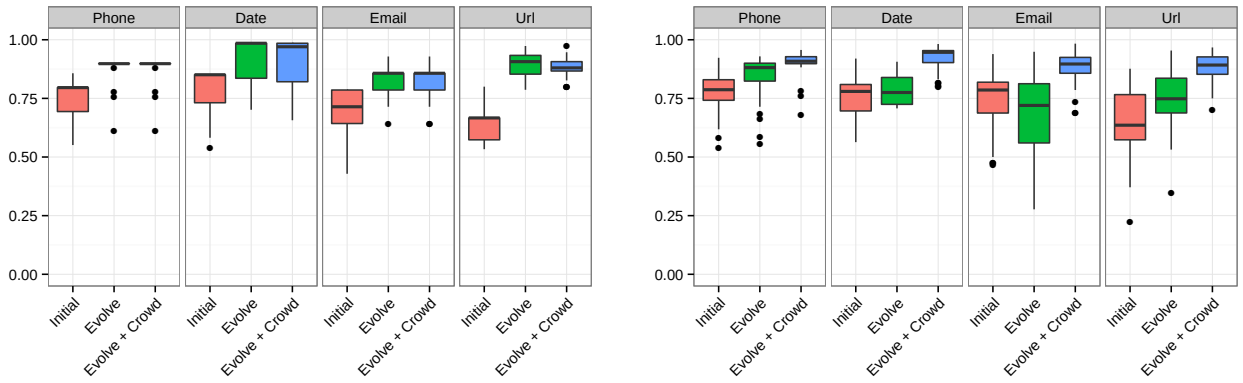
- overall boost obtained via our algorithm;
- time required to perform the boosting;
- monetary cost of boosting;
- expansion of the test suite for the task we are interested in created via crowd-sourcing;
- accuracy and size of the resulting programs.

Time	iurisilvio	vmas	7aRPnjkn	alixaxel	shobhit
Wed 7:32 PM	posted solution				
Wed 7:32 PM	updated solution				
Wed 8:15 PM	posted solution				
Wed 8:15 PM	updated solution				
Wed 8:17 PM	updated solution				
Wed 9:04 PM&	posted solution				
Wed 9:13 PM	updated solution				
Wed 11:00 PM			ask for clarification		
Wed 11:01 PM			ask for clarification		
Wed 11:36 PM			updated solution		
Thu 10:03 AM		updated solution			
Thu 10:03 AM				posted solution	
Mon 7:00 PM	left a comment				
Mon 7:01 PM	**updated solution**				
Mon 7:08 PM	left comment				
Mon 7:10 PM			updated solution		
Mon 7:12 PM			left comment		
Mon 7:12 PM			left comment		

Figure 10: Crowd-sourcing regex for URLs at `https://bountify.co/5f`.

	Examples		Candidate regexes				Regex length				State count			
	+	-	Total	Bountify	Regexlib	Other	25%	50%	75%	Max	25%	50%	75%	Max
Phone numbers	20	29	8	3	0	5	44.75	54	67.75	96	14.75	27	28	30
Dates	31	36	6	3	1	2	154	288	352.25	434	19	39.5	72	78
Emails	7	7	10	4	3	3	33.5	68.5	86.75	357	7.25	8.5	10	20
URLs	36	39	9	4	0	5	70	115	240	973	12	25	30	80

Figure 9: Case studies summarized.



(a) Fitness on the golden set.

(b) Fitness on the evolved set.

Figure 13: Aggregate comparison of different strategies

5.1 Crowd-Sourcing Setup

Before we describe our experiments, we first outline our crowd-sourcing setup.

Bountify: In addition to using regular expressions from blogs, StackOverflow, etc., we crowd-sourced the creation of initial regular expressions using Bountify (`http//bountify.co`). Bountify is a service that allows users to post a coding task or technical question to the site, along with a monetary reward starting at as little as $1. Typical rewards on Bountify are $5–10. We posted four seperate "bounties" or requests, each consisting of a high-level specfication, and asked for a regular expressions implementations of these specifications.

Over the course of our expreiments, freelance developers on Bountify submitted 14 regular expressions that were used in the experiments. We supplemented those with regular expressions gathered from blogs and other web sites for a total of 33 regular expressions, as detailed in Figure 8.

Interactions with developers on Bountify sometimes get fairly involved, as illustrated in Figure 10. This figure captures a process of getting the best regular expressions for URLs. Each column of the table corresponds to an individual developer who participated in this bounty. The winner was `iurisilvio`, who was also the first to post a solution—this task was posted on Wednesday evening, with the first solution from `iurisilvio` arriving almost instantaneously. However, in this case, the winning soluton did not emerge until the following Monday, after several interactions and clarifications from the poster (us), and refinements of the original solution. Note that this was not done in "real time"; we could have been more agressive in responding to potential solutions to have this process converge more quickly.

Mechanical Turk: We used Amazon's Mechanical Turk to classify additional examples discovered as part of boosting and generated using the technique described in Section 3. For each example, we used 5 Mechanical Turk workers workers provided with a high-level specification of the task. For each string in batch the Mechanical Turk worked had to classify it as either *Valid* or *Invalid*.

These strings were grouped in batches containing up to 50 strings and workers were paid a maximum of $0.25 and a minimum of $0.05. These rates were scaled linearly depending on the number of strings within a batch. Classified strings were added to the training set assuming they reached an agreement consensus rate of 60%. Figure 17 shows additional data on Mechanical Turk consensus.

Overall, the workers we encountered on Mechanical Turk have been fairly positive towards our automatically-generated tasks. A few even chose to send us comments such as the ones shown in Figure 14 via email. Clearly, some workers are concerned about doing a good job and also about their reputation. Others expressed doubts regarding some of the corner cases.

5.2 Experimental Setup

We applied our technique to all unordered pairs of regular expressions (including reflective pairs $\langle x, x \rangle$) within each of the four specification categories: Phone numbers, Dates, Emails, and URLs. Overall, we considered a total of 465 initial pairs. We evaluated boosting in two seperate scenarios: first, using only the genetic programming techniques of shuffles and mutations and second, using these techniques *and* example generation and refinement with the help of Mechanical Turk workers.

Initial regular expressions: In Figure 9, we characterize the inputs used in our experiment by length and by the number of states in each resulting SFA. These values convey

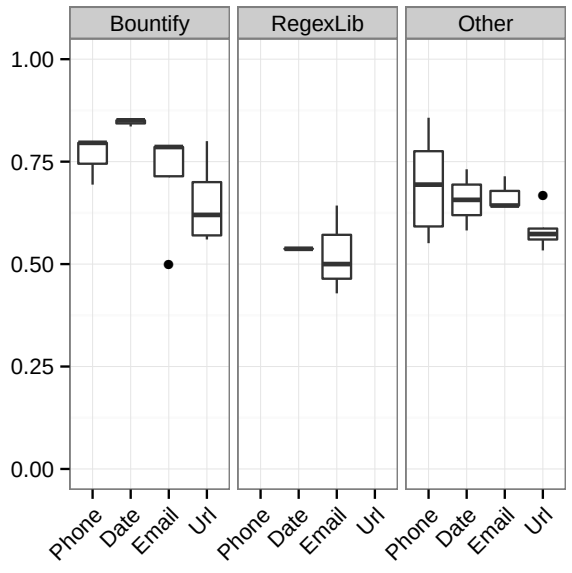


Figure 11: Initial Fitness Values

the varying levels of complexity across the input regular expressions. Columns 2 and 3 show the number of positive and negative examples we started with for each task. Columns 4–7 show where our 33 regular expressions come from. Bountify is the most popular source, with 14 coming from there. The regular expression length shown in columns 8–11 is quite high, with the median frequently being as high as 288 for Dates. To some degree, regular expression length reflects the complexity of these tasks. The state count shown in columns 12–15 is generally relatively low, due to the SFAs's ability to achieve good compression via sympolic representation.

Figure 11 shows the distribution of initial accuracy (fitness) values by source, diffirentiating beweeen Bountify, RegexLib, a widely-used regex repository, and other sources such as blogs, web sites, StackOverflow, etc. Surprisingly, the initial values obtained through Bountify are higher than

Task	Initial	Evolved	Evolved + Crowd
Phone numbers	0.80	0.90	0.90
Dates	0.85	0.99	0.97
Emails	0.71	0.86	0.86
URLs	0.67	0.91	0.88

(a) Fitness values measured on the *golden set*.

Task	Initial	Evolved	Evolved + Crowd
Phone numbers	0.79	0.88	0.91
Dates	0.78	0.78	0.95
Emails	0.79	0.72	0.90
URLs	0.64	0.75	0.89

(b) Fitness values measured on the *evolved set*.

Figure 12: Summary of boosting results across task specifications.

I did this HIT a few minutes ago and have a feeling that I did not do it right. If that is the case, could you please let me fix any mistakes so that I do not get rejected?
I have a doubt regarding Your valid URL finding HITs . Eg: http://âIJł.jw/s Can I consider the above URL as Invalid.
I look forward to doing your HITS. Thanks for posting them!
Thank you for providing such type of hits. Keep on posting jobs like this. It will be helpful to me as a financial support for my family.

Figure 14: Examples of Mechanical Turk feedback.

those obtained from RegexLib, a widely-used library of regular expression designed to be reused by a variety of developers. Overall, initial fitness values hover between .5 and .75, with none of the regexes being either too good or too bad.

5.3 Boosting Results

Our experiments center around pairwise boosting for the four chosen tasks: Phone numbers, Emails, Dates, URLs. We test the quality of the regular expressions obtained through boosting by measuring the *accuracy* on both positive and negative examples. Our measurments are performed both the *golden* set and the *evolved* set. We consider the measurements on the evolved set to be more representative, because the golden set is entirely under our contol and could be manipulated by adding and removing examples to influence accuracy measurements. The evolved set, on the other hand, evolves "naturally", through refinement and obtained Mechanical Turk consensus.

Figure 12 captures our high-level results obtained from the boosting process. Each cell captures the accuracy of the generated program variant. Comparing the values in columns 3 and 4 to column 2, we see that our process *consistently* does result in boosting *across the board*. It is worth pointing out that having a stable technique that produces consistent boosting for a range of programs is both very difficult and tremendously important to make our approach predictable.

Figure 13 contains a more detailed exporation of the boosting process portrayed as a distribution over the regular expression 465 pairs. Vertical bars are used to represent the shape of each distributions. Both the tables and the histogram show that the most significant boost is consistently obtained with the Evolve + Crowd strategy.

5.4 Boosting Process

Figure 17 characterizes the boosting process in three dimensions: the number of generations, the number of generated strings, and the measured consensus for classification tasks. For each of these dimentions, we provide 25%, 50%, 75%, and Max numbers in liu of a histogram.

Note that we artificially limit the number of generations to 10. However, about half the pairs for the Emails task finish in 5 generations only. For URLs, there are always 10 generations required — none of the results converge prematurely. The number of generated strings is relatively modest, peaking at 207 for Dates. This suggests that the total crowdsourcing costs for Mechanical Turk should not be very high. Lastly, the classification consensus is very high overall. This is largely due to the our candidate string generation technique in Section 4.5. By making strings look "nice" it prevents a wide spread of opinions.

Task	Generations				Generated strings				Classification consensus			
	25%	50%	75%	Max	25%	50%	75%	Max	25%	50%	75%	Max
Phone numbers	7	8	10	10	0	6.5	20.25	83	1	1	1	1
Dates	10	10	10	10	29	45	136	207	1	1	1	1
Emails	5	5	6.5	10	2	7	17	117	1	1	1	1
URLs	10	10	10	10	54	72	107	198	0.99	1	1	1

Figure 17: Characterizing boosting process.

Task	Shuffles (thousands)				% Successful shuffles				Mutations (thousands)				% Successful mutations			
	25%	50%	75%	Max	25%	50%	75	Max%	25%	50%	75%	Max	25%	50%	75%	Max
Phone numbers	73	98	113	140	0.002	0.071	1.888	17.854	5	6	8	13	3.8	5.5	11.6	34.0
Dates	14	108	162	171	0.21	1.51	7.22	38.92	8	12	17	37	16	31	35	53
Emails	3	8	22	165	0.45	1.62	5.11	15.04	0	0	2	15	41	54	78	100
URLs	116	178	180	180	0.88	6.62	34.29	50.15	9	20	52	114	30	35	41	64

Figure 18: Successful propagation of candidates.

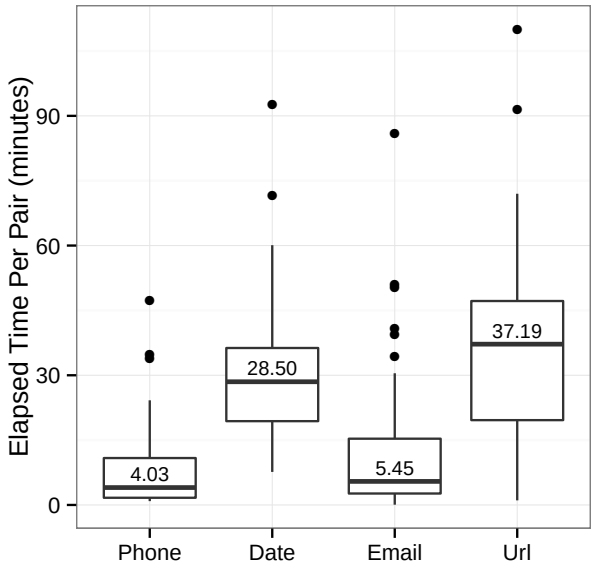


Figure 15: Running times for each task, aggregated across all pairs.

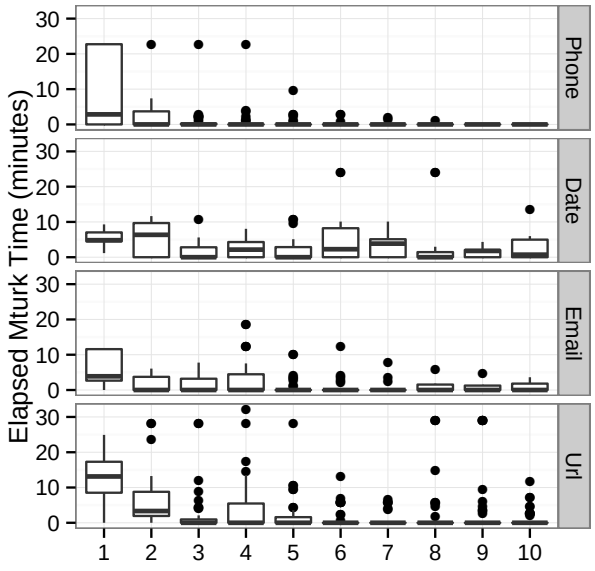


Figure 16: Latencies for Mechanical Turk across generations for all tasks.

Figure 18 provides additional statistics for the shuffling and mutation process across the tasks in the 25%, 50%, 75%, and Max format used before. Across the board, the number of shuffles produced during boosting is in tens of thousands. Yet only a very small percentage of them *succeed*, i.e survive to the next generation. This is because for the vast majority, the fitness is too small to warrant keeping them around. The number of mutations is smaller, only in single thousands, and their survival rate is somewhat higher. This can be explained by the fact that mutations are relatively local transformations and are not nearly as drastic as shuffles.

5.5 Running Times

Figure 15 shows the overall running time for pairwise boosting for each task. The means vary from about 4 minutes per pair and 37 minutes per pair. Predictibly, Phone numbers completes quicker than URLs. Note that for Emails, the times are relatively low. This correlates well with the low number of generated strings in Figure 17. Making the boosting process run faster may involve having low-latency Mechanical Turk workers on retainer and is the subject of future work.

Much of the delay is due to waiting for Mechanical Turk to complete. Figure 16 captures the latencies for Mechanical Turk calls, across generations. Clearly, as the boosting process proceeds, it spends less time waiting for Mechanical Turk, in part because there is less ambiguity to be resolved by Mechanical Turk workers. The only exception is Dates, which has a slight "bump" in Mechanical Turk latencies in the middle. Note, however, that Mechanical Turk latencies are influenced by other factors such as day of week, time of year, weather, etc.

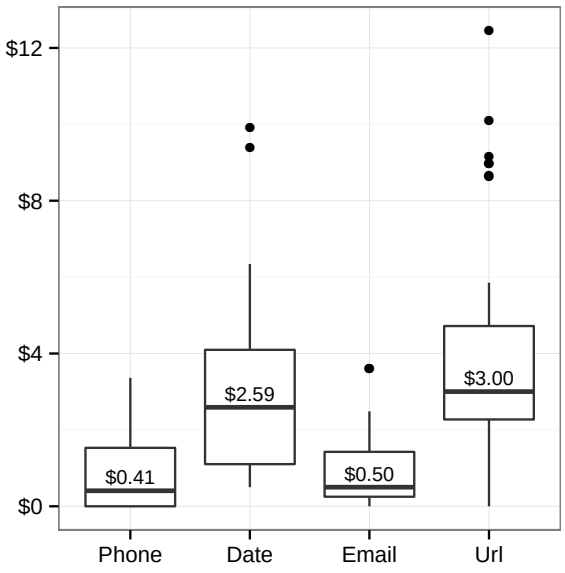


Figure 19: Costs for Mechanical Turk.

5.6 Boosting Costs

Figure 19 shows the costs of performing program boosting across the range of four tasks. The overall costs are quite modest, ranging between 41¢ and $3 per pair. We do see occasional outliers on the high end costing about $12. Some pairs do not requre Mechanical Turk call-outs at all, resulting in zero cost.

6. Discussion

We see the following main challenges with the program boosting approach. In this paper, we aim to provide solutions to only *some* of these challenges. Addressing all of them in a comprehensive manner will undoubtedly require much subsequent research.

Low quality of responses: just like with other crowd-sourcing tasks, our approach suffers from response quality challenges, both because the crowd participant is honestly mistaken (American Mechanical Turk workers think that Unicode characters are not allowed within URLs) or because they are trying to game the system by providing an answer that is either random or obviously too broad (such as `/.*/` for regular expression sourcing tasks).

Everyone makes the same mistake: analysis of *security sanitizers* in [13] illustrates that everyone gets the same (easy) part of the programming task correct. At the same time, everyone gets corner cases wrong as well. If everyone gets the same corner case incorrect, voting and consensus-based approaches are not going to be very helpful: everyone will incorrectly vote for the same outcome, falsely raising our confidence in the wrong solution.

Over-fitting on the training data: just like with any other learning tasks, over-fitting the answer (model) to the data is a potential problem. One way to mitigate this is to force generalization, either explicitly or through limiting the size (length or number of states or another similar metric) of the selected program. For instance, we could favor smaller regular expressions in our selection.

Program complexity is too high: while it is possible to blend programs together to achieve good results on training and testing data, it is desirable to produce resulting programs that are too complex to be understood. In some cases, since these programs will be used as black boxes, this is fine; in others, this is not the case.

Knowing when to stop: in the context of crowd-sourcing, knowing when to stop soliciting answers is difficult: even if you have absolute agreement among existing workers, it is not clear that asking more questions may not eventually yield disagreement about a non-obvious corner case. The current approach in this paper does not use a more flexible approach to getting the desired level of confidence, although several techniques have been proposed [].

Crowd-sourcing is too expensive: the cost of crowd-sourcing is certainly not the least important consideration. Untrained "recognition" tasks on services such as Mechanical Turk can cost about 2¢–5¢ a task, which is quite cheap. Programming assignments can easily cost from $5 to $50 on sites like Bountify. Our overall philosophy is to maximize the number of participants while keeping the wages relatively low as opposed to hiring one or two highly skilled and highly compensated developers. However, this is not the only approach.

Monetization and payment: it is not clear how to properly compensate the workers whose (programming) efforts become blended into the end-product. There are thorny intellectual property issues to grapple with. There is the question of whether the workers should be compensated beyond their initial payment, as the software to which they have contributed becomes successful.

Crowd latency: is a major issue in getting program boosting results faster. In the future, it may be possible to have a set of workers on retainer with faster response times. Another option is to design a more asynchonous approach that would speculatively explore the program space.

Sub-optimality: because we are evolving the training set, it is possible that in earlier generations we abandoned programs that in later generations would appear to be more fit. One way to compensate for this kind of sub-optimality of our technique is to either revisit the evaluation once the evolved set has been finalized, or to inject some of the previously rejected programs from past generations into the mix at later stages.

7. Related Work

Below we provide a brief overview of some of the related literature.

Genetic algorithms: Genetic programming methods alter structures that represent members of a population to produce a result that is better according to fitness or optimality conditions. Evolutionary approaches for building automata and state machines have been widely studied. Early work by Fogel et al. [6] evolved finite state machines to predict symbol sequences. Others have extended these techniques to build modular systems that incorporate independent FSMs to solve maze and grid exploration problems [4] or to predict note sequences in musical compositions [14]. In software engineering, genetic programming approaches have been applied to fixing software bugs [7] and software optimization [5].

Learning DFAs: Grammatical inference is the study of learning a grammar by observing examples of an unknown language. This problem was introduced by Gold [9], who showed that a learning algorithm can produce a new grammar that can generate all of the examples seen so far in polynomial time. Many variants of this problem have been studied, including different language classes and different learning models. Relevant to this paper is the study of producing a regular language from labeled strings, where the learning algorithm is given a set of positive and negative examples that have been labeled by an unknown target DFA and the task is to predict the output of the target DFA on new examples. This problem has been shown to be hard in the worst case [15, 25], but many techniques have been demonstrated to be practical in the average case. The L-star algorithm [1] can infer a minimally accepting DFA but assumes that the target language is known and that hypothesized grammars can be checked for equivalence with the target language. State merging algorithms [18] relax the requirement for a minimal output, and work by building a prefix-tree acceptor for the training examples and then merge states together that map to the same suffixes. A number of extensions to this technique have been proposed [16, 17, 23]. Evolutionary approaches to learning a DFA from positive and negative examples have also been proposed [21, 22].

Learning regular expressions: Automatic generation of regular expressions from examples has been explored in the literature for information extraction. Galassi et al. [8] presented a technique to extract events from DNA sequences, by learning simple regular expressions that anchor the relevant strings. Others have applied evolutionary approaches to infer regular expressions subject to different fitness metrics. These techniques use various types of transformations on the regular expressions themselves, rather than a DFA representation [3, 10, 19]. Furthermore, the alphabet size is minimized to either extracted tags from text processing tools or the ASCII character set. In contrast, our approach directly manipulates the automata representing the regular expression and our transformation techniques can handle more complex regular expressions and large alphabets.

Program synthesis: Recent work has investigated automatic synthesis of program fragments from logical and example based specifications [11, 12, 28, 29]. A common thread of this research is that when the high-level insights of how a solution can be described, errors often appear in the low-level details. These tools use formal methods to aid in the construction the low-level implementation of a specification. Our technique differs in that it aims to address the issue of corner-cases in specification implementations by blending a diverse collection of solutions through crowd-sourcing.

Crowd-sourcing: The emergence of crowd-sourcing platforms such as Amazon's Mechanical Turk has led to a variety of inquiry into the manner in which human computation can be incorporated into programming systems. Several platforms [2, 20, 24, 26]. have been designed to abstract the details of using a crowd-sourcing service away from the programmer, so that issues of latency, quality control and cost are easier to manage. Our work introduces a novel use of crowd-sourcing to automatically refine the training set in our genetic programming algorithm.

8. Conclusions

This paper presents a novel crowd-sourcing approach to program synthesis called *program boosting*. Our focus is difficult programming tasks, which even the most expert of developers have trouble with. Our insight is that the wisdom of the crowds can be brought to bear on these challenging tasks. In this paper we show how to use two crowds, a crowd of skilled developers and a crowd of untrained computer workers to successfully produce solutions to complex tasks that involve crafting regular expressions.

We have tested our approach to program boosting on four complex tasks, we have crowd-sourced 33 regular expressions from Bountify and several other sources, and performed pairwise boosting on them. We find that our program boosting technique is stable: it produces *consistent* boosts in accuracy when tested on 465 pairs of crowd-sourced programs. The cost of program boosting is generally quite modest, varying between several cents and about $3 for the untrained crowd and about $10–20 for the skilled crowd, depending on the complexity of the task.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[2] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor. Automan: a platform for integrating human-based and digital computation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 639–654, New York, NY, USA, 2012. ACM.

[3] D. F. Barrero, D. Camacho, and M. D. R-Moreno. Automatic web data extraction based on genetic algorithms and regular expressions. In L. Cao, editor, *Data Mining and Multi-agent Integration*, pages 143–154. Springer US, 2009.

[4] K. Chellapilla and D. Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages –1356 Vol. 2, 1999.

[5] B. Cody-Kenny and S. Barrett. Self-focusing genetic programming for software optimisation. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, GECCO '13 Companion, pages 203–204, New York, NY, USA, 2013. ACM.

[6] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, UK, 1966.

[7] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[8] U. Galassi and A. Giordana. Learning regular expressions from noisy sequences. In *Proceedings of the 6th international conference on Abstraction, Reformulation and Approximation*, SARA'05, pages 92–106, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[10] A. González-Pardo and D. Camacho. Analysis of grammatical evolutionary approaches to regular expression induction. In *IEEE Congress on Evolutionary Computation*, pages 639–646, 2011.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[12] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture*

Notes in Computer Science, pages 418–423. Springer Berlin Heidelberg, 2011.

[13] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *Proceedings of the USENIX Security Symposium*, August 2011.

[14] Y. Inagaki. On synchronized evolution of the network of automata. *Trans. Evol. Comp*, 6(2):147–158, Apr. 2002.

[15] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41(1):67–95, Jan. 1994.

[16] B. Lambeau, C. Damas, and P. Dupont. State-merging dfa induction algorithms with mandatory merge constraints. In A. Clark, F. Coste, and L. Miclet, editors, *Grammatical Inference: Algorithms and Applications*, volume 5278 of *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin Heidelberg, 2008.

[17] K. Lang, B. Pearlmutter, and R. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In V. Honavar and G. Slutzki, editors, *Grammatical Inference*, volume 1433 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1998.

[18] K. J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 45–52, New York, NY, USA, 1992. ACM.

[19] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[20] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 29–30, New York, NY, USA, 2009. ACM.

[21] S. Lucas and T. Reynolds. Learning dfa: evolution versus evidence driven state merging. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 1, pages 351–358 Vol.1, 2003.

[22] S. M. Lucas and T. J. Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1063–1074, 2005.

[23] J. Oncina and P. García. Identifying Regular Languages in Polynomial Time. In H. Bunke, editor, *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[24] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, CIKM '12, pages 1203–1212, New York, NY, USA, 2012. ACM.

[25] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, Jan. 1993.

[26] A. J. Quinn, B. B. Bederson, T. Yeh, and J. Lin. Crowdflow: Integrating machine learning with mechanical turk for speed-cost-quality flexibility. *Better performance over iterations*, 2010.

[27] RSnake. XSS (Cross Site Scripting) cheat sheet. `http://ha.ckers.org/xss.html`, 2006.

[28] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[29] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[30] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *IN PROC. OF THE 1ST INTERNATIONAL CONFERENCE ON CONSTRAINTS IN COMPUTATIONAL LOGICS*, pages 34–49. Springer-Verlag, 1994.

[31] M. Veanes and N. Bjørner. Symbolic automata: The toolkit. In *TACAS*, pages 472–477, 2012.

[32] R. A. Wagner. Order-n correction for regular languages. *Commun. ACM*, 17(5):265–268, May 1974.