Program Boosting: Program Synthesis via Crowd-Sourcing

Robert A Cochran  
University of North Carolina  
rac@cs.unc.edu

Loris D’Antoni  
University of Pennsylvania  
lorisdan@seas.upenn.edu

Benjamin Livshits  
Microsoft Research  
livshits@microsoft.com

David Molnar  
Microsoft Research  
dmolnar@microsoft.com

Margus Veanes  
Microsoft Research  
margus@microsoft.com

Abstract

In this paper, we investigate an approach to program synthesis that is based on crowd-sourcing. With the help of crowd-sourcing, we aim to capture the “wisdom of the crowds” to find good if not perfect solutions to inherently tricky programming tasks, which elude even expert developers and lack an easy-to-formalize specification.

We propose 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 implement this approach in a system called CropdBoost and show in our experiments that interesting and highly non-trivial tasks such as writing regular expressions for URLs or email addresses can be effectively crowd-sourced. We demonstrate that carefully blending the crowd-sourced solutions 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.

Categories and Subject Descriptors  F.4.3 [Theory of Computation]: Formal languages; H.5.3 [Information Systems]: Collaborative Computing; D.1.2 [Programming Techniques]: Automatic Programming

Keywords  Program Synthesis; Crowd-sourcing; Symbolic Automata; Regular Expressions

1. Introduction

Everyday programming involves solving numerous small but necessary tasks. Some of these tasks are fairly routine; others are surprisingly challenging. Examples of challenging self-contained tasks include 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 while properly addressing all the tricky corner cases. Furthermore, there is plenty of 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 are valid email addresses or whether removing all characters outside of the [a−zA−Z] set is a valid sanitization strategy for preventing SQL injection attacks. These tasks are under-specified; there may not be absolute consensus on what solution is correct; moreover, different people may get different parts of the solution wrong.

What if we could pose these tricky programming tasks as a crowd-sourcing challenge? Ideally, we would be able to describe the task in question in English, admittedly, a very loose form of specification, with all its inherent ambiguities and under-specified corner cases. We would subsequently use the “wisdom of the crowds” to arrive at a solution, without having a precise specification a priori, but perhaps armed with some positive and negative examples, giving us a partial specification. This paper explores this deceptively simple idea and expands on previous investigations into the use of crowd-sourcing to help with technically challenging programming problems. Our implementation, CRODBoost, shares some of the high-level goals with systems such as TurkIt [27], Deco [33], and Collabode [13].

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. While RFCs may define formal grammars for URLs, it is non-trivial to construct a regular expression that can be used in practice from these specifications. 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 (ftp://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 on 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 han-
dle of @diegoperini managed to get all the answers right. @stephenhay came close, getting all positive inputs right, but missing some of the negative inputs. Notably, this experiment performed by Mathias was a specific form of program crowd-sourcing.

1.2 Key Observations

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

- The problem posed by Mathias 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 example set is very difficult;
- potential answers provided by developers range in length (median values 38–1,347) and accuracy (.56–1), a great deal, as measured on a set of examples. Note that the most accurate answer provided by diegoperini is in this case not the longest;
- developers get different portions of the answer wrong; while a particular developer may forget the ftp:// URL scheme but remember to include optional port numbers that follow the host name, another developer may do exactly the opposite;
- cleverly combining (or blending) partially incorrect answers may yield a correct one.

Inspired by these observations, program boosting is a technique that combines crowd-sourced programs using the technique of genetic programming to yield a solution of higher quality. A way of combining individual classifiers to improve the accuracy is referred to as classifier boosting in machine learning [37], hence of our choice of the term program boosting. This technique is especially helpful for problems that elude a precise specification.

1.3 Other Domains for Program Boosting

The experimental results in this paper focus heavily on the regular expression domain; regular expressions represent a limited but very important class of programming tasks, which maintain decidability of many important properties, unlike programs written in general-purpose languages like C or Java. However, we feel that a wide range of problems are captured by the mental model outlined above. Just like with other general techniques, we cannot really foresee all domains in which boosting may be valuable, but we give several examples below.

- Security sanitizers: Security sanitizers are short, self-contained string-manipulation routines that are crucial in preventing cross-site scripting attacks in web applications, yet programming these sanitizers is widely-recognized to be a difficult task [17]. Sanitizers written in domain-specific languages have properties that allow for automated manipulation and reasoning about program behavior, but human intervention is still required when a specification is not precise. Reasoning about sanitizers amounts to reasoning about transducers; we feel that algorithms similar to those presented in this paper can be developed for transducers as well. Program boosting can incorporate powerful new techniques for program analysis with the on-demand efforts of both expert and non-expert human insight.

- Combining code patches: We envision the application of program boosting in other domains as well. For example, while it would be difficult to mix the code from several Java programs directly, we can imagine how the space of possible combinations of code patches written by different developers could be explored to find an optimal result as judged by the crowd.

Program boosting could be used to construct a layout engine that renders results which are most pleasing to the user.

- Web site layout: In web site design, A/B testing is often used to measure design changes and their observable influence on behavior [20]. Building on the idea that the customer is always right, program boosting could be used to construct a layout engine that renders results which are most pleasing to the user.

The key thread linking these application areas together is that a problem exists where the overall specification and how it should be implemented are difficult to pin down, but it is easy to indicate whether a given solution is correctly operating on a single input. Defining effective mutation and crossover operations for each of these domains remains a separate research challenge, which we only address for regular expressions.

1.4 Contributions

Our paper makes these 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 combines (or blends) them to provide a better result, according to a fitness function.

- We show how to implement program boosting for regular expressions. We propose a genetic programming technique [2, 21, 35] with custom-designed crossover and mutation operations. We propose a new genetic programming paradigm in which the fitness function is evolved along with the candidate programs.

- We implement our program boosting technique in a tool, CROWDBoost, to generate complex regular expressions. We represent regular expressions using Symbolic Finite Automata (SFAs), which enable succinct representation, while supporting large alphabets. We adapt classic algorithms, such as string-to-language edit distance, to the symbolic setting. To the best of our knowledge, ours is also the first work that uses genetic programming on automata over complex alphabets such as UTF-16.

- 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 16.25%, which is a significant improvement on already high-quality initial solutions. Importantly, the boosting effect is consistent.
across the tasks and sources of initial regular expressions, which enhances our belief in the generality of our approach.

1.5 Paper Organization
The rest of the paper is organized as follows. Section 2 gives an outline of our approach of using two crowds in tandem to generate programs. Section 3 gives the details of our approach to boosting regular expression-based programs based on symbolic finite automata or SFAs. Section 4 provides an experimental evaluation in the context of four case studies. Section 5 contains a discussion of some of the outstanding challenges we see for future research. Finally, Sections 6 and 7 describe related work and conclude.

2. Program Boosting
This section provides an overview of program boosting. Much of the motivation for our approach comes from experimenting with different tasks that can be captured via regular expressions. For example, the following imperfect initial regular expressions for matching US phone numbers can be combined to yield the final solution, the resulting regex avoids the obvious pitfall in the initial two, namely, initial regexes allow digit sequences of arbitrary length.

\[
\begin{align*}
\text{\([0-9\{3\}-[0-9\}]*\) & \text{\([0-9\{3\}-[0-9\}]+[0-9\{4\}]\)}
\end{align*}
\]

1.5 Paper Organization
The rest of the paper is organized as follows. Section 2 gives an outline of our approach of using two crowds in tandem to generate programs. Section 3 gives the details of our approach to boosting regular expression-based programs based on symbolic finite automata or SFAs. Section 4 provides an experimental evaluation in the context of four case studies. Section 5 contains a discussion of some of the outstanding challenges we see for future research. Finally, Sections 6 and 7 describe related work and conclude.

2. Program Boosting
This section provides an overview of program boosting. Much of the motivation for our approach comes from experimenting with different tasks that can be captured via regular expressions. For example, the following imperfect initial regular expressions for matching US phone numbers can be combined to yield the final solution, the resulting regex avoids the obvious pitfall in the initial two, namely, initial regexes allow digit sequences of arbitrary length.

\[
\begin{align*}
\text{\([0-9\{3\}-[0-9\}]*\) & \text{\([0-9\{3\}-[0-9\}]+[0-9\{4\}]\)}
\end{align*}
\]

Two-crowd approach: For simplicity, in this paper we focus on binary classification tasks, that is programs that (1) consume a single input; (2) produce a boolean (yes/no) output; and (3) for any input, a non-specialist computer user can decide if the answer for it should be a yes or a no. Examples of such tasks include programs that decide if the input is a properly formatted phone number or programs that decide if the input image contains a giraffe. The last requirement is necessary for refinement, i.e., deciding the proper outcome for tricky corner cases with the help of an untrained crowd. Our observation is that while the untrained 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 giraffe in an image, humans are remarkably good at recognizing whether a given picture has a giraffe in it. This two-crowd approach helps us to both collect or source candidate programs and to systematically expand the space of considered inputs by asking the untrained crowd for disambiguation.

While other fitness criteria are possible, in this paper we focus on improving the accuracy of blended programs on a training set of positive and negative examples.

2.3 Program Boosting via Genetic Programming
To generalize the approach advocated in our example above, a natural technique to consider is genetic programming [2, 21, 35], a specialized form of genetic algorithms. Genetic programming is a search technique in the space of programs, whose goal is to improve program fitness over a series of generations. A successful genetic programming formulation relies on implementing two operations that may be familiar to the reader: crossover and mutation.
Given a set of initial crowd-sourced programs, the program boosting algorithm proceeds in generations. In the context of our phone number example, these initial programs may be the two initial regular expressions. At every generation, it performs a combination of crossover and mutation operations. (In our example, this may tweak individual parts of the regular expression to handle phone number separators like `-` and `.`) As a form of refinement, new examples are added to the training set. As an example, in our regular expression implementation the goal of refinement is to at least achieve full utilization on a machine with 8 or 16 cores. 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 in-line. 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 invoke crowd calls asynchronously.

### 3. Boosting Regular Expressions

In this section we instantiate the different parts of the program boosting algorithm for the regular expression domain. We first describe Symbolic Finite Automata and show how they are used to represent regular expressions in our implementation, CROWD-BOOST. Next, we present algorithms for crossover, mutation, and example generation, used by the algorithm in Figure 3.

#### 3.1 Symbolic Finite Automata

While regular expressions are succinct and relatively easy to understand, they are not easy to manipulate algebraically. In particular, there is no 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 character, causing the number of transitions in the DFA to be proportional to the size of the alphabet. When the alphabet is large (UTF16 has 216 elements) this representation becomes impractical.

Symbolic Finite Automata (SFAs) [41] extend classic automata with symbolic alphabets. In an SFA each edge is labeled with a predicate, rather than a single input character, and this allows the automaton to represent multiple concrete transitions succinctly. For example, in the SFA of Figure 3 the transition from state 10 to state 11 is labeled with the predicate `¬ϕ∨ψ∧χ`. Because of the size of the UTF16 set, this transition in classic automata would be 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 need to “manipulate” such predicates. Let’s consider the problem of intersecting two DFAs. In classic 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 `((p, q), a, (p', q'))`. 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')` (where `ϕ` and `ψ` are predicates), the intersected SFA will need to synchronize the two transitions only on the values that are both in `ϕ` and `ψ`; therefore the new transition will be `((p, q), ϕ ∧ ψ, (p', q'))` where the guard is the conjunction of the two predicates `ϕ` and `ψ`. Moreover if the predicate `ϕ ∧ ψ` defines an empty set of characters, this transition should be removed. This example shows how the set of predicates used in the SFA should at least be closed under `∧` (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 classic closure prop-
erties of regular language the set of predicates must also be closed under negation.

**Definition 1.** A Boolean algebra \( B \) has components \((D_B, P_B, f, \bot, \wedge, \neg)\). \( D_B \) is a set of domain elements, and \( P_B \) is a set of predicates closed under Boolean connectives \( \wedge \) and \( \neg \), with \( \bot \in P_B \). The denotation function \( f : P_B \rightarrow 2^{2^{D_B}} \) is such that \( f(T) = D_B \), \( f(f \wedge \neg) = f(f) \wedge f(\neg) \), and \( f(\neg) = D_B \setminus f(f) \). For \( \varphi \in P_B \), we write IsSat(\( \varphi \)) when \( f(\varphi) \neq \bot \), and say that \( \varphi \) is satisfiable. We say that \( B \) is decidable if IsSat is decidable.

We can now define Symbolic Finite Automata.

**Definition 2.** A Symbolic Finite Automaton, SFA, \( A \) is a tuple \( (B, Q, q_0, F, \delta) \) 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 \( A \). We say that \( A \) is deterministic if for every state \( q \in Q \) there do not exist two distinct transitions \((q, \varphi, q_1)\) and \((q, \psi, q_2)\) in \( \delta \), such that IsSat(\( \varphi \land \psi \)). We say that \( A \) is complete if for every state \( q \in Q \) and every symbol \( a \in D_B \) there exists a transition \((q_1, \varphi, q_2)\) such that \( a \in f(\varphi) \). In this paper we only consider deterministic and complete SFAs, and for this class of SFAs we can then define the reflexive-closure of \( \delta \) as \( \delta^+ = \delta \cup \delta^2 \cup \ldots \), and for all \( a \in D_B \) and all \( \varphi \in B \), \( \delta^+ (a, \varphi) = \delta^+ (a, \varphi) \cup \delta^+ (a, \varphi^2) \cup \ldots \cup \delta^+ (a, \varphi^k) \).

**BDD algebra:** We describe the Boolean algebra of BDDs, which is used in this paper to model sets of UTF16 characters that are used in regular expressions. A BDD algebra \( B^{\text{BDD}} \) is the powerset algebra whose domain is the finite set \( b_{\text{BDD}} \), for some \( k > 0 \), consisting of all non-negative integers less than \( 2^k \), or equivalently, all \( k \)-bit bitvectors. A predicate is represented by a BDD [40] 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 \( \varphi \) is the set of all integers \( n \) such that a binary representation of \( n \) corresponds to a solution of \( \varphi \). For example, in the case of URLs over the alphabet UTF16, we use the BDD algebra \( B^{\text{BDD}} \) to naturally represent sets of UTF16 characters (bit-vectors). We consider the SFA and BDD implementations from the library [41].

### 3.2 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 construct SFAs \( P \) and \( N \), which represent the languages of all positive and all negative examples, respectively. For any SFA \( A \), we then compute the cardinality of the intersection sets \( L(A) \cap L(P) \) and \( L(N) \cap L(A) \) (see dashed regions in Figure 4), both of which can be computed fast using SFA operations. The accuracy can be then computed as \((L(A) \cap L(P)) + |L(N) \cap L(A)|) / |L(P) \cup L(N)| \) and will range from 0 to 1. A challenge inherent with our refinement technique is that our evolved example set can greatly deviate from the initial golden set. While imperfect, we still want to treat the golden set as a more reliable source of truth; to this end, we use weighting to give the golden set a higher weight in the overall fitness calculation. In our experimental evaluation, we get reliably good results if we set golden:evolved weights to 9:1.

### 3.3 Crossover

A crossover operation interleaves two SFAs into a single SFA that “combines” their behaviors. An example of this operation is illustrated in Figure 6. Given two SFAs \( A \) and \( B \), the crossover algorithm creates a new SFA by redirecting two transitions, one from \( A \) to \( B \), and one from \( B \) to \( A \). The goal of such an operation is that of using a component of \( B \) inside \( A \). The crossover algorithm is shown in Figure 5. In all of the following algorithms we assume the SFAs to be minimal [8].

An SFA can have many transitions and trying all the possible crossovers can be impractical. Concretely, if \( A \) has \( n_1 \) states and \( n_1 \) transitions, and \( B \) has \( n_2 \) states and \( n_2 \) transitions, there will be \( O(n_1 n_2 m_1 m_2) \) possible crossovers, and checking fitness for this many SFAs would not scale.

We devise several heuristics that try to mitigate such blowup by limiting the number of possible crossovers. The first technique we use is to guarantee that: 1) if we leave \( A \) by redirecting a transition \((q, \varphi, q_1) \to B \), and come back to \( A \) from \( B \) with a transition that reaches a state \( q_2 \in A \), then \( q_2 \) is reachable from \( q_1 \), but different from it (we write \( q_1 \not< q_2 \)), and 2) if we reach \( B \) in a...
state $p_1$, and leave it by redirecting a transition $(p_2, \varphi, p)$, then $p_2$ is reachable from $p_1$ (we write $p_1 \preceq p_2$). Following these rules, we only generate crossover automata for which there always exists an accepted string that traverses both the redirected transitions.

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 or more 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 7. In this example, the only SCC with more than one state is the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Moreover, most of the phone number prefixes 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 sub-graph where every node has enough states. Consider again the SFA in Figure 7. In this example, the SFA is a maximal connected acyclic sub-graph where every node has degree smaller or equal than 2. In the SFA in Figure 7, $\{1, 3, 5\}$, $\{2, 4\}$, and $\{9, 10\}$ are stretches.

**Single-entry, single-exit components**: Even using stretches the collapsing is often ineffective. Consider again the SFA in Figure 7. 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 a component is characterized by the following features:

1. it is a connected direct acyclic sub-graph,
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.

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

In the SFA in Figure 7, 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 components, and $B$ has $c_2$ components and $t_2$ transitions between different components, then there will be $O(c_1c_2t_1t_2)$ possible crossovers. In practice this number is much smaller than $O(n_1n_2m_1m_2)$.

**One-way crossovers**: One way crossovers 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. If $A$ has $t_1$ transitions between different components, and $B$ has $c_2$ components, then there will be $O(c_2t_1)$ possible one-way crossovers.

### 3.4 Mutation

In its classic definition a mutation operator alters one or more values of the input program and produces a mutated one. SFAs have too many values to be altered (every transition can carry $2^{36}$ elements), and a “blind” approach would produce too many mutations. Instead we consider a guided approach, in which mutations take as input an SFA $A$ and a counterexample $s$, such that $s$ is incorrectly classified by $A$ (it’s a negative example but it belongs to $L(A)$ or $s$ is a negative example but it belongs to $L(A^\prime)$). 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 syntactic change in order to correctly classify the counterexample.

**Diminishing mutations**: Given a negative example and an SFA $A$ such that $s \in L(A)$ generate an SFA $A^\prime$, such that $L(A^\prime) \subseteq L(A)$ and $s \not\in L(A^\prime)$. Given a string $s = a_1 \ldots a_n$ that is accepted by $A$, the algorithm finds a transition $(q_i, \varphi, q_j)$ that traverses using the input character $a_i$ (for some $i$) when reading $s$ and either removes the whole transition, or simply shrinks the guard to $\varphi \land \neg a_i$ disallowing the symbol $a_i$. Given a string of length $k$, this mutation can generate at most $2^k$ mutated SFAs. Given the state $q \in F$ such that $\delta^*(\{q_0, s\}, q) = q$, we also output The SFA $A^\prime = (q_0, Q, F \setminus \{q\}, \delta)$, in which the input SFA is mutated by removing a final state.

**Augmenting mutations**: Given a positive example and an SFA $A$ such that $s \not\in L(A)$ generate an SFA $A^\prime$, such that $L(A^\prime) \subseteq L(A)$ and $s \in L(A^\prime)$. Given a string $s = a_1 \ldots a_n$ that is not accepted by $A$, the algorithm finds a state $q$ such that, for some $i$, $\delta^*(\{q_0, q_0 \ldots a_{i-1}\} = q$, and a state $q_j$ such that, for some $j > i$, $\delta^*(q, a_{i} \ldots a_{n}) \in F$. Next, it adds a path from $q$ to $q_0$ on the string $a_{i+1} \ldots a_{n}$. This is done by adding $|s_{mid}| = 1$ extra states. It is easy to show that the string $s$ is now accepted by the mutated SFA $A^\prime$. Given a string of length $k$ and an SFA $A$ with $n$ states this mutation can generate at most $nk^2$ mutated SFAs. When there exists a state $s$ such that $\delta^*(\{q_0, s\}, q) = q$ we also output the SFA $A^\prime = (q_0, Q, F \cup \{q\}, \delta)$, in which the input SFA is mutated by adding a new final state.

### 3.5 Example Generation

Generating one string is often not enough to “characterize” the language of an SFA. In generating examples, we aim to follow...
the following invariant: we attain full state coverage for the SFAs we allow to precede to the next generation. For each SFA \( A = (Q, q_0, F, \delta) \), we generate a set of strings \( S \subseteq L(A) \), such that for every state \( q \in Q \), there exists a string \( a_1 \ldots a_n \in S \), such that for some \( i, \delta'(q_0, a_1 \ldots a_i) = q \). The example generation algorithm is described in Figure 8; given an SFA with \( k \) state it terminates in at most \( k \) iterations. The algorithm simply generates a new string at every iteration, which is forced to cover at least one state which hasn’t yet been covered.

Unfortunately, this naïve approach tends to generate strings that look “random”, causing untrained crowd workers to be overly conservative by classifying virtually all of the generated strings as negative examples, even when they are not. For example, we have observed 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 \( e \) in the training set. We can formalize this notion of closeness by using the classic metric of string edit distance. The edit distance between two strings \( s \) and \( s' \), \( ED(s, s') \), is the minimum number of edits (insertion, deletion, and character replacement) that transforms \( s \) into \( s' \). Given an SFA \( A \) and a string \( s \in L(A) \), we want to find a string \( s' \in A \) such that \( ED(s, s') \) is minimal. In the case of DFAs there exists an algorithm that given a string \( s \) and a DFA \( A \) computes the metric \( \min \{ ED(s, s') | s' \in L(A) \} \), representing the minimum edit distance between \( s \) and all the strings in \( L(A) \) [42]. We symbolically extend the algorithm in [42] to compute the minimum string-to-language edit distance for SFAs, and we modify it to actually generate the witness string. The algorithm has complexity \( O(|s|n^2) \), where \( n \) is the number of states in the SFA.

As an illustration, Figure 9a shows some examples of randomly generated strings, and Figure 9b several strings generated using the edit distance technique. Clearly, the second set looks less “random” and less intimidating to an average Mechanical Turk worker.

4. Experimental Evaluation

To evaluate CROWDBOOST, our implementation of program boosting, we face a fundamental challenge; there is no easy definition of correctness because the problems we are attempting to solve do not have a clearly defined specification. Recall that our framework is designed to evolve a notion of correctness as the crowd provides more feedback, which is then incorporated into the solution. Therefore, our goal in this section is to describe the experiments on four different regular expression tasks and demonstrate that our technique can both refine an initial specification and construct a solution that performs well (on the both evolved and initial specifications). Before we describe our experiments, we first outline our crowd-sourcing setup.

Bountify: In addition to sourcing regular expressions from an online library of user-submitted regexes (Regexlib.com) and other sources (blogs, StackOverflow, etc.), we crowd-sourced the creation of initial regular expressions using Bountify, 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 separate “bounties”
Over the course of our experiments, freelance developers on Bountify submitted 14 regular expressions that were used in the experiments. The reader is encouraged to consult the (anonymized) URLs in Figure 10 to learn more about the program crowd-sourcing interactions, which are revealing, but too long to fit into this submission. We supplemented those with regular expressions gathered from Regexpal and other online sources total of 33 regular expressions, as detailed in Figure 14.

Interactions with developers on Bountify sometimes get fairly involved, as illustrated in Figure 11. 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 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 solution 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 aggressive 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 earlier. For each example, we used 5 Mechanical Turk workers provided with a high-level specification of the task. For each string in batch the Mechanical Turk workers had to classify it as either Valid or Invalid. As example of a Mechanical Turk user interface is shown in Figure 12.

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 18 shows additional data on Mechanical Turk consensus.

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://ł.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.

**4.1 Experimental Setup**

We applied our technique to all unordered pairs of regular expressions (including reflexive 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 separate scenarios: first, using only the genetic algorithm techniques of crossover and mutations and second, using these techniques and example generation and refinement with the help of Mechanical Turk workers.

**Initial test set:** In Figure 14 we characterize our initial test set. Columns 2 and 3 show the number of positive and negative examples included in the golden set for each task. Recall that only a subset of these examples was provided to the Bountify workers. Note, that it is difficult to select a comprehensive test set for evaluation, given the complexity of the tasks and the lack of a specification.

An ideal test set would contain all the corner cases of the final program, but that program is not known. Consider coming up with a comprehensive set of test cases for phone numbers; numbers direct from a phone book will not provide a lot of the needed complexity. Finding a set of “misshapen” phone numbers that is comprehensive is similarly tough, this is why testing on the evolved set presents a harder challenge for boosting than alternatives. In our experiments, the evolved examples “push the envelope” by considering difficult cases and are generated with help from the crowd, which are difficult to find elsewhere.

**Initial regular expressions:** In Figure 14 we also characterize the size and source of the candidate regular expressions used in our experiments. Columns 4–7 show where our 33 regular expressions come from. Bountify is the most popular source, with 14 coming from there. In Figure 15, we characterize the inputs used in our experiment by length and by the number of states in each resulting SFA. These values convey the varying levels of complexity across the input regular expressions. The regular expression length shown in columns 2–5 of Figure 15 is quite high, with the median frequency 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 6–9 of Figure 15 is generally relatively

<table>
<thead>
<tr>
<th>Golden set</th>
<th>Candidate regexes</th>
<th>Candidate regex source:</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone numbers</td>
<td>20 29</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Dates</td>
<td>31 36</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Emails</td>
<td>7 7</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>URLs</td>
<td>36 39</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure 15: Summarized size and complexity of the candidate regexes in our case studies.

Figure 16: Fitness of candidate regular expressions.

Figure 17: In each task category, boosting results are shown via fitness values measured on either the golden set or the evolved set for three separate regexes; initial, “no crowd” and “crowd”.

low, due to the SFA’s ability to achieve compression via symbolic representation.

Figure 16 shows the distribution of initial accuracy (fitness) values by source. Somewhat surprisingly, the initial values obtained through Bountify are higher than those obtained from RegExLib, a widely-used library of regular expression specifically 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”. Of course, starting with higher-quality initial regular expressions creates less room for growth.

4.2 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 measurements 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 control 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 17 captures our high-level results obtained from the boosting process. We display the the mean values for the experiments on each task. We show results as measured both on the golden set of examples and the evolved, expanded set. We see that our process consistently results in boosting across the board. Significant improvements can be observed by comparing columns 5 and 7. The average boost across all the tasks is 16.25%. 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. Note also that on the larger evolved set the advantage of having a crowd (columns 4 and 7) is more pronounced than on the smaller golden set.

4.3 Boosting Process

Figure 18 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 dimensions, we provide 25%, 50%, 75%, and Max numbers in lieu 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 crowd-sourcing 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. By making strings look “nice” it prevents a wide spread of opinions.

Figure 19 provides additional statistics for the crossover and mutation process across the tasks in the 25%, 50%, 75%, and Max format used before. Across the board, the number of crossovers 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 crossovers. The overall experience is not uncommon in genetic algorithms.

Running times: Figure 20a 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. Predictably, 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 18. Making the boosting process run faster may involve having low-latency Mechanical Turk workers on retainer and is the subject of future work.

Boosting costs: Figure 20b 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 require Mechanical Turk call-outs at all, resulting in zero cost.
Successful crossovers

<table>
<thead>
<tr>
<th>Task</th>
<th>Crossovers (thousands)</th>
<th>% Successful crossovers</th>
<th>Mutations (thousands)</th>
<th>% Successful mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone numbers</td>
<td>73 98 113 140</td>
<td>0.002 0.071 1.888 17.854</td>
<td>5 6 8 13</td>
<td>3.8 5.5 11.6 34.0</td>
</tr>
<tr>
<td>Dates</td>
<td>14 108 162 171</td>
<td>0.21 1.51 7.22 38.92</td>
<td>8 12 17 37</td>
<td>16 31 35 53</td>
</tr>
<tr>
<td>Emails</td>
<td>3 8 22 165</td>
<td>0.45 1.62 5.11 15.04</td>
<td>0 0 2 15</td>
<td>41 54 78 100</td>
</tr>
<tr>
<td>URLs</td>
<td>116 178 180 190</td>
<td>0.88 6.62 34.29 15.15</td>
<td>9 20 52 114</td>
<td>30 35 41 64</td>
</tr>
</tbody>
</table>

Figure 19: Successful propagation of candidates.

(a) Running times for each task.  (b) Costs for Mechanical Turk.

Figure 20: Running times and costs.

5. 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: If every developer 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. Analysis of security sanitizers in [17] illustrates that sometimes that may be the case.

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, 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.

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: The time it takes to gather solutions from the crowd 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 asynchronous 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 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.

6. Related Work

Crowd-sourcing: Recent work has investigated the incorporation of human computation into programming systems. Several platforms have been proposed to abstract the details of crowd-sourcing services away for the programmer, making the issues of latency, quality control, and cost easier to manage [3, 33, 36]. TurkIt [27] and Collabode [13] take different approaches to the problem of making programming easier; the former enabling usage of non-experts to solve tasks well suited for humans and the latter enabling collaboration between programmers. Another approach to using crowd-sourcing is to break large programming tasks into small independent “microtasks” [25]. Our work furthers progress towards the goal of solving difficult programming problems by leveraging a crowd of mixed skill levels and using formal methods to combine efforts from these multiple workers.

Genetic algorithms: In general, genetic algorithms alter structures that represent members of a population to produce a result that is better according to fitness or optimality conditions. Early work [9] 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 [5] or to predict note sequences in musical compositions [18]. In software engineering, genetic algorithms have been applied to fixing software bugs [10] and software optimization [6].

Genetic programming is a sub-area of genetic algorithms focused on evolving programs [2, 21, 35]. An evolutionary approach has also been applied to crowd-sourcing creative design, where workers iteratively improved and blended design sketches of a chairs and alarm clocks [31]. Our work introduces a novel use of crowd-sourcing to both crowd-source initial programs and to automatically refine the training set and the fitness function through crowd-sourcing.

Program analysis and synthesis: In the theory of abstract interpretation, widening operators are used to compute a valid abstraction for some function [7]. Our use of the mutation and crossover operations to refine or blend SFAs follows the spirit of this ap-
proach and may be one path towards applying program boosting to domains where abstract interpretation has seen much success, e.g., static analysis. Recent work has investigated automatic synthesis of program fragments from formal and example based specifications [15, 16, 38, 39]. A specification which defines program behavior can be viewed as a search problem for which constraint solvers or customized strategies can be applied. These approaches use formal methods to aid in the construction of the low-level implementation of a specification. In our technique, the initial specification may be open to interpretation or not fully defined. We take advantage of the crowd to refine our specification and improve the correctness of a collection of implementations.

**Learning regular expressions**: Automatic generation of regular expressions from examples has been explored in the literature for information extraction. In the context of DNA analysis, events can be extracted from DNA sequences by learning simple regular expressions that anchor the relevant strings [11]. Others use transformations on regular expressions themselves, rather than a DFA representation [4, 14, 26]. In contrast, our approach uses the power of symbolic automata to manipulate complex expressions containing large alphabets. We are not aware of traditional learning approaches suitable for learning regular expressions that contain Unicode, due to the large alphabet size.

**Learning DFA**: Grammatical inference is the study of learning a grammar by observing examples of an unknown language. It has been shown that a learning algorithm can produce a new grammar that can generate all of the examples seen so far in polynomial time [12]. 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. This problem has been shown to be hard in the worst case [19, 34], 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. Recent results [30] extend to large alphabets but still require strong assumptions about the oracle. State merging algorithms [23] 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 [22, 24, 32]. Evolutionary approaches to learning a DFA from positive and negative examples have also been proposed [28, 29].

### 7. 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 implemented program boosting in a tool called **CROWDBoost** and have tested it 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. Even when starting with initial programs of high quality (fitness), crowd-sourced from qualified developers, we are consistently able to achieve boosts in accuracy, averaging 16.25%.

### References


[23] K. J. Lang. Random DFA’s can be approximately learned from sparse uniform examples. In *Proceedings of Workshop on Computational