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 CROWDBOOST 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 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.

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, CROWDBOOST, shares some of the high-level goals with systems such as TurkIt [27], Deco [33], and Collabode [13].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
POPL '15, January 15–17, 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3300-9/15/01...$15.00.
http://dx.doi.org/10.1145/2676726.2676973

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 (`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 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 handle

of @diegoperini managed to get all the answers right[1]. Twitter user @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 (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.

Merging 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 program 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.

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

Regex source	Regex length	True positive	True negative	Overall accuracy
@krijnhoetmer	115	.78	.41	.59
@gruber	71	.97	.36	.65
@gruber v2	218	1.00	.33	.65
@cowboy	1,241	1.00	.15	.56
@mattfarina	287	.72	.44	.57
@stephenhay	38	1.00	.64	.81
@scottgonzales	1,347	1.00	.15	.56
@rodneyrehm	109	.83	.36	.59
@imme_emosol	54	.97	.74	.85
@diegoperini*	**502**	**1.00**	**1.00**	**1.00**

Figure 1: Representative regular expressions for URLs obtained from http://mathiasbynens.be/demo/url-regex. For every possible solution we show its length, true and false positive rates, and the overall accuracy. The last row is the winner.

Combining static analysis checkers: The precision of static analysis *checkers* or bug finding tools can be improved via cleverly blending them together. One scheme could be to have multiple checkers for NULL dereferences vote on a particular potential violation. Another would be to blend machine learning and program analysis by training a decision tree which determines which checker to apply in response to a range of local static features.

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 blending 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

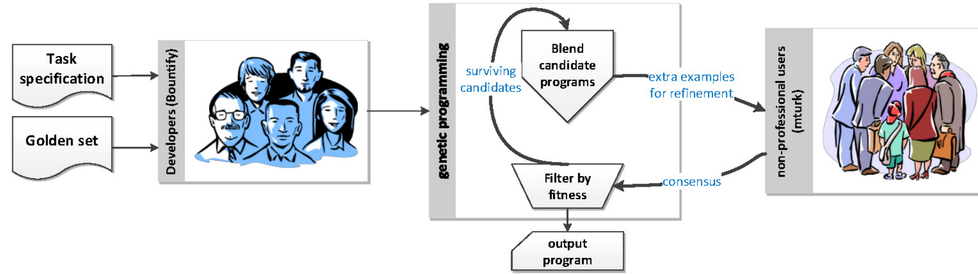


Figure 2: System architecture.

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 `^[0-9]{3}-[0-9]*-[0-9]{4}$` and `^[0-9]{3}-[0-9]{3}-[0-9]*$` can be combined to yield `^[0-9]{3}-[0-9]{3}-[0-9]{4}$`. While still probably not the final solution, the resulting regex avoids the obvious pitfalls in the initial two, namely, initial regexes allow digit sequences of arbitrary length.

2.1 Background on Crowd-Sourcing

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.

Bountify (`http://bountify.co`) 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 for a particular programming task. 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 will yield some possible regular expressions for URL filtering; so will exploring a programming advice site such as StackOverflow.

2.2 Overview of Program Boosting

Figure 2 shows the architecture of our system. We start with a textual task specification and an initial training set of (positive and negative) examples (also called "golden set"). To crowd-source a solution to the specified task, we take advantage of two crowds, the *skilled crowd* consisting of developers and the *untrained crowd* consisting of regular computer users. The former contains developers for hire via services such as Bountify, typically skilled in one or more languages such as Java and C++, the latter consists of regular computer users found on Mechanical Turk.

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*.

```
1: Input: Programs σ, examples φ, crossover function β, mutation function μ,
   example generator δ, fitness function η, budget θ
2: Output: Boosted program
3: function Boost(⟨σ, φ⟩, β, μ, δ, η, θ)
4:  while (η̂ < 1.0 ∧ θ > 0) do                      ▷ Until perfect or no money
5:   ϕ = ∅                                      ▷ New examples for this generation
6:   for all ⟨σ_i, σ_j⟩ ∈ FindCrossoverCandidates(σ) do
7:    for all σ′ ∈ β(⟨σ_i, σ_j⟩) do                       ▷ Crossover σ_i and σ_j
8:     ϕ = ϕ ∪ δ(σ′, φ)                                ▷ Generate new examples
9:     σ = σ ∪ {σ′}                                  ▷ Add this candidate to σ
10:    end for
11:   end for
12:   for all ⟨σ_i⟩ ∈ FindMutationCandidates(σ) do
13:    for all σ′ = μ(σ_i) do                                       ▷ Mutate σ_i
14:     ϕ = ϕ ∪ δ(σ′, φ)                                ▷ Generate new examples
15:     σ = σ ∪ {σ′}                                  ▷ Add this candidate to σ
16:    end for
17:   end for
                                   ▷ Get consensus on these new examples via mturk
18:   ⟨φ_ϕ, θ⟩ = GetConsensus(ϕ, θ)                          ▷ and update budget
19:   φ = φ ∪ φ_ϕ                              ▷ Add the newly acquired examples
20:   σ = Filter(σ)                                        ▷ Update candidates
21:   ⟨σ̂, η̂⟩ = GetBestFitness(σ, η)
22:  end while
23:  return σ̂                                  ▷ Return program with best fitness
24: end function
```

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

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 attain 100% *state coverage* by considering non-obvious cases such as `212.555-1212` or `1-)212) 555-1212` as either valid or invalid phone numbers to be added to the evolving training set. Finally, the candidates with the highest fitness are chosen to continue to the next generation.

Crowd-sourcing initial programs and continuous refinement of the training set are key differeniators of program boosting and more standard genetic programming [2, 21, 35].

2.4 Program Boosting Algorithm

Figure 3 shows our program boosting algorithm as pseudo-code. Let Σ be the set of all programs and Φ be the set of all inputs. 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;
- $\beta : \Sigma \times \Sigma \rightarrow 2^{\Sigma}$ is the crossover function that takes two programs and produces a set of possible crossovers;
- $\mu : \Sigma \rightarrow 2^{\Sigma}$ is the mutation function that given a program produces a set of possible mutated programs;
- $\delta : \Sigma \times 2^{\Phi} \rightarrow 2^{\Phi}$ given a program and a set of examples generates a new set of training examples;
- $\eta : \Sigma \rightarrow \mathbb{N}$ is the fitness function;
- $\theta \in \mathbb{N}$ is the budget for Mechanical Turk crowd-sourcing.

Later 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.

Our implementation, CROWDBOOST, benefits greatly from parallelism. In particular, we make the two loops on lines 6 and 12 of 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 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, CROWDBOOST. 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 algorithm for complementing or intersecting them directly. We therefore opt for finite automata. 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 (UTF-16 has 2^{16} 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 7 the transition from state 10 to state 11 is labeled with the predicate `[^#--\/?\s]`. Because of the size of the UTF-16 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 $(\langle p, q\rangle, a, \langle p', q'\rangle)$. 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') (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 $(\langle p, q\rangle, \varphi \wedge \psi, \langle p', q'\rangle)$ where the guard is the conjunction of the two predicates φ and ψ. Moreover if the predicate $\varphi \wedge \psi$ 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 $\wedge$ (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 classic closure properties of regular language the set of predicates must also be closed under negation.

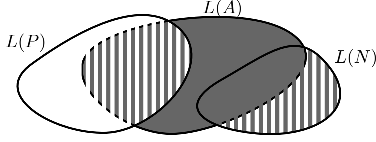


Figure 4: Fitness computation for an SFA A. The dashed regions represent the strings on which A is correct.

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 $f(\top) = \mathcal{D}_B$, $f(\bot) = \emptyset$, $f(\varphi \wedge \psi) = f(\varphi) \cap f(\psi)$, and $f(\neg\varphi) = \mathcal{D}_B \setminus f(\varphi)$. For $\varphi \in P_B$, we write $IsSat(\varphi)$ when $f(\varphi) \neq \emptyset$, and say that φ 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, 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. We say that A is deterministic if for every state $q \in Q$ there do not exist two distinct transitions (q, φ, q_1), and (q, ψ, q_2) in δ, such that $IsSat(\varphi \wedge \psi)$. We say that A is complete if for every state $q \in Q$ and every symbol $a \in \mathcal{D}_B$ there exists a transition (q_1, φ, 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-transitive closure of δ as, $\delta^*(q, \varepsilon) = q$, and for all $a \in \mathcal{D}$ and $s \in \mathcal{D}^*$, $\delta^*(q, as) = \delta(q', s)$, if $\exists(q, \varphi, q') \in \delta$ such that $a \in f(\varphi)$. The language accepted by A is $L(A) = \{s \mid \delta^*(q_0, s) \in F\}$.

BDD algebra: We describe the Boolean algebra of binary decision diagrams (BDDs), which is used in this paper to model sets of UTF-16 characters that are used in regular expressions. 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 non-negative integers less than 2^k, or equivalently, all k-bit bit-vectors. 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 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 UTF-16, we use the BDD algebra $2^{bv_{16}}$ to naturally represent sets of UTF-16 characters (bit-vectors). We consider the SFA and BDD implementations from the symbolic automata toolkit [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 $L(N) \setminus L(A)$ (see dashed regions in Figure 4), both of which can be computed fast using SFA operations. The accuracy can be then computed as $(|L(A \cap P)| + |L(N \setminus A)|)/|L(P \cup N)|$ 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.

1: **Input:** SFAs $A_1 = (Q_1, q_0^1, F_1, \delta_1)$, $A_2 = (Q_2, q_0^2, F_2, \delta_2)$
2: **Output:** All crossovers of A_1 and A_2
3: **function** CROSSOVERS(A_1, A_2)
4: $C_1 :=$ COMPONENTS(A_1)
5: $C_2 :=$ COMPONENTS(A_2)
6: **for all** $c_1 \in C_1$ **do**
7: $C_1' := \{c_1' \mid c_1 \prec_{A_1} c_1'\}$
8: **for all** $t_\rightarrow = (p_1, \varphi, p_2) \in$ EXIT_MOVES(c_1, A_1) **do**
9: **for all** $c_2 \in C_2$ **do**
10: **for all** $i_2 \in$ ENTRY_STATES(c_2, A_2) **do**
11: $C_2' := \{c_2' \mid c_2 \preceq_{A_2} c_2'\}$
12: **for all** $c_2' \in C_2'$ **do**
13: **for all** $t_\leftarrow = (q_1, \varphi, q_2) \in$ EXIT_MOVES(c_2', A_2) **do**
14: **for all** $c_1' \in C_1'$ **do**
15: **for all** $i_1 \in$ ENTRY_STATES(c_1', A_1) **do**
16: $t_\rightarrow' := (p_1, \varphi, i_2)$, $t_\leftarrow' := (q_1, \varphi, i_1)$
17: $\delta_{new} := \delta_1 \cup \delta_2 \setminus \{t_\rightarrow, t_\leftarrow\} \cup \{t_\rightarrow', t_\leftarrow'\}$
18: **yield return** $(Q_1 \cup Q_2, q_0^1, F_1 \cup F_2, \delta_{new})$
19: **end function**
20: **function** EXIT_MOVES(c, A)
21: **return** $\{(p, \varphi, q) \in \delta_A \mid p \in c \wedge q \notin c\}$
22: **end function**
23: **function** ENTRY_STATES(c, A)
24: **return** $\{q \in Q_A \mid (\exists(p, \varphi, q) \in \delta_A . p \notin c \wedge q \in c) \vee (q = q_0^A)\}$
25: **end function**

Figure 5: Crossover algorithm.

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].

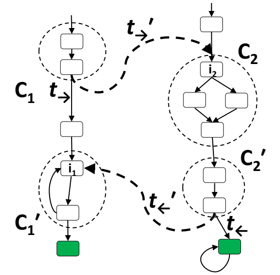


Figure 6: Crossover.

An SFA can have many transitions and trying all the possible crossovers can be impractical. Concretely, if A has n_1 states and m_1 transitions, and B has n_2 states and m_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, φ, 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 $q_1 \prec q_2$), and 2) if we reach B in a state p_1, and leave it by redirecting a transition (p_2, φ, p), then p_2

is reachable from p_1 (we write $p_1 \preceq p_2$). 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 $\{11\text{–}12\}$. 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 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.

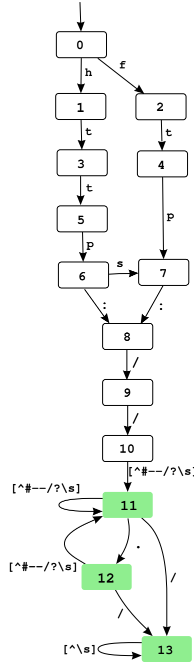


Figure 7: Identifying components.

Such components can be computed in linear time by using a variation of depth-first search starting in each node with in-degree smaller or equal 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 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.

```
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(C)
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 8: Example generation.

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^{16} 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 (s is a positive example but it doesn't belong to $L(A)$, or s is a negative example but it belongs to $L(A)$). 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 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', 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 the symbol a_i. Given a string of length k, this mutation can generate at most $2k$ mutated SFAs. Given the state $q \in F$ such that $\delta^*(q_0, s) = q$, we also output The SFA $A' = (q_0, Q, F \setminus \{q\}, \delta)$, in which the input SFA is mutated by removing a final state.

Augmenting mutations: Given a positive example and an SFA A such that $s \notin L(A)$ generate an SFA A', such that $L(A) \subseteq L(A')$ and $s \in L(A')$. Given a string $s = a_1 \dots a_n$ that is not accepted by A, the algorithm 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 an 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 the SFA $A' = (q_0, Q, F \cup \{q\}, \delta)$, 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

```
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) Examples generated randomly

```
Whttp//youtu:e.com        http://y_:outube.com
0.http//youtu:e.com       ht:tpWWW://youtube.com
h_ttp//youtu:e.com        ht:tpWWW0://youtube.com
WWW000http://youtu:e.com  http://youtube.com/
```

(b) Examples generated with the edit distance approach starting with the string `http://youtube.com`.

Figure 9: Two approaches to examples generation compared.

the following invariant: we attain *full state coverage* for the SFAs we allow to proceed 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 $s \notin L(A)$, 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.[2] 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 specification). Before we describe our experiments, we first outline our crowd-sourcing setup.

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 10: Specifications provided to Bountify workers. The last two columns capture the number of positive and negative examples (a subset of the golden set) given to workers in the task specifications.

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			asked question		
Wed 11:01 PM			asked question		
Wed 11:36 PM				posted solution	
Thu 10:03 AM			updated solution		
Thu 10:03 AM				posted solution	
Fri 1:02 AM					posted solution
Mon 7:00 PM	left 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 11: Illustrating the experience of crowd-sourcing initial regexes for URLs at `https : //bountify.co/5f`.

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"

[2] The algorithm in [42] 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.

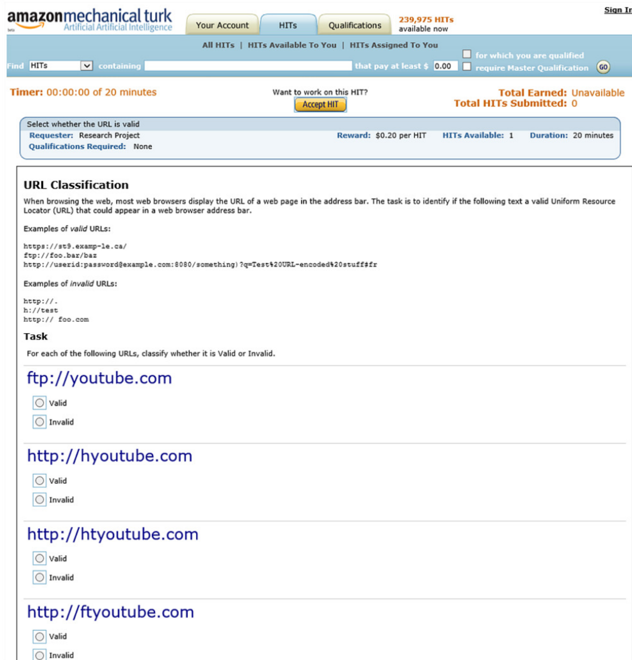


Figure 12: Mechanical Turk UI for string classification.

or requests, each consisting of a high-level specification, and asked for a regular expressions implementations of these specifications.

Over the course of our experiments, freelance developers on Bountify submitted 14 regular expressions that were used in the experiments. 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 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 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.

We supplemented the regular expressions created by users on Bountify with regular expressions gathered from Regexlib.com and other online sources total of 33 regular expressions, as detailed in Figure 14.

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, 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.

Overall, the workers we encountered on Mechanical Turk have been fairly positive towards our automatically-generated tasks. A

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://t.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 13: Examples of real-life Mechanical Turk feedback.

	Golden set		Candidate	Candidate regex source:		
	+	-	regexes	Bountify	Regexlib	Other
Phone numbers	20	29	8	3	0	5
Dates	31	36	6	3	1	2
Emails	7	7	10	4	3	3
URLs	36	39	9	4	0	5

Figure 14: The number of examples in the golden set and the number of candidate regexes in each case study.

few even chose to send us comments such as the ones shown in Figure 13 via email. Clearly, some workers are concerned about doing a good job and also about their reputation. Others asked questions about some of the corner cases.

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 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 6–9 of Figure 15 is generally relatively low, due to the natural compression achieved via symbolic representation.

	Regex character length				SFA state count			
	25%	50%	75%	Max	25%	50%	75%	Max
Phone numbers	44.75	54	67.75	96	14.75	27	28	30
Dates	154	288	352.25	434	19	39.5	72	78
Emails	33.5	68.5	86.75	357	7.25	8.5	10	20
URLs	70	115	240	973	12	25	30	80

Figure 15: Summarized size and complexity of the candidate regexes in our case studies.

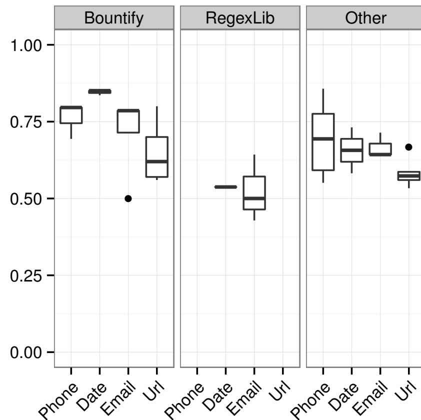


Figure 16: Fitness of candidate regular expressions.

EVALUATED ON...	GOLDEN SET			EVOLVED SET		
		Boosted			Boosted	
Task	**initial**	**no crowd**	**crowd**	**initial**	**no crowd**	**crowd**
Phone numbers	0.80	0.90	0.90	0.79	0.88	0.91
Dates	0.85	0.99	0.97	0.78	0.78	0.95
Emails	0.71	0.86	0.86	0.79	0.72	0.90
URLs	0.67	0.91	0.88	0.64	0.75	0.89

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".

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.

Task	Generations				Generated strings				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 18: Characterizing the boosting process. The Generated Strings column, demonstrates that the size of the evolved set varies for each experiment pair, depending on the number of strings created to cover the generated regexes.

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.

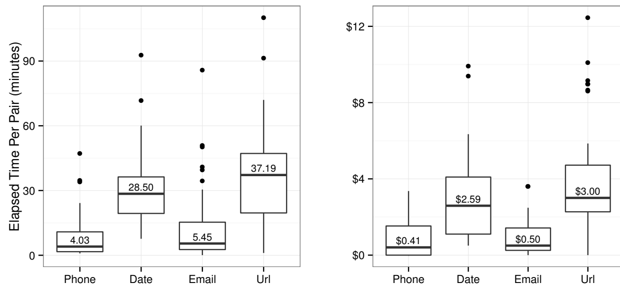
Figure 19 provides additional statistics for the crossover and mutation process across the tasks in the 25%, 50%, 75%, and Max format used before. Across the board, the number of crossovers produced during boosting is in tens of thousands. Yet only a very small percentage of them *succeed*, i.e survive 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 outlets on the high end costing about $12. Some pairs do not require Mechanical Turk call-outs at all, resulting in zero cost.

Task	Crossovers (thousands)				% Successful crossovers				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 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, 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.

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 DFAs: 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

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction*. 1997.

[3] 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[4] D. F. Barrero, D. Camacho, and M. D.R-Moreno. Automatic web data extraction based on genetic algorithms and regular expressions. In *Data Mining and Multi-agent Integration*. 2009.

[5] K. Chellapilla and D. Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the Congress on Evolutionary Computation*, 1999.

[6] B. Cody-Kenny and S. Barrett. Self-focusing genetic programming for software optimisation. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, 2013.

[7] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1995.

[8] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the Symposium on Principles of Programming Languages*, 2014.

[9] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. 1966.

[10] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, 2009.

[11] U. Galassi and A. Giordana. Learning regular expressions from noisy sequences. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation*, 2005.

[12] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.

[13] M. Goldman, G. Little, and R. C. Miller. Collabode: Collaborative coding in the browser. In *Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering*, 2011.

[14] A. González-Pardo and D. Camacho. Analysis of grammatical evolutionary approaches to regular expression induction. In *Proceedings of the Congress on Evolutionary Computation*, 2011.

[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the Symposium on Principles of Programming Languages*, 2011.

[16] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *Proceedings of the Converence on Computer Aided Verification*, 2011.

[17] 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*, 2011.

[18] Y. Inagaki. On synchronized evolution of the network of automata. *IEEE Transactions on Evolutionary Computation*, 6(2), 2002.

[19] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1), 1994.

[20] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1), 2009.

[21] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. 1992.

[22] B. Lambeau, C. Damas, and P. Dupont. State-merging DFA induction algorithms with mandatory merge constraints. In *Proceedings of the International Colloquim on Grammatical Inference*, 2008.

[23] K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *Proceedings of Workshop on Computational Learning Theory*, 1992.

[24] K. J. Lang, B. A. Pearlmutter, and R. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the International Colloquim on Grammatical Inference*, 1998.

[25] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek. Microtask programming: Building software with a crowd. In *Proceedings of the Symposium on User Interface Software and Technology*, 2014.

[26] 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*, 2008.

[27] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: tools for iterative tasks on mechanical turk. In *Proceedings of the SIGKDD Workshop on Human Computation*, 2009.

[28] S. M. Lucas and T. J. Reynolds. Learning DFA: evolution versus evidence driven state merging. In *Proceedings of the Congress on Evolutionary Computation*, 2003.

[29] 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), 2005.

[30] O. Maler and I. E. Mens. Learning regular languages over large alphabets. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[31] J. V. Nickerson, Y. Sakamoto, and L. Yu. Structures for creativity: The crowdsourcing of design. In *CHI Workshop on Crowd-sourcing and Human Computation*, 2011.

[32] J. Oncina and P. García. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, Series in Machine Perception and Artificial Intelligence. 1992.

[33] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *Proceedings of the Conference on Information and Knowledge Management*, 2012.

[34] L. Pitt and M. K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM*, 40(1), 1993.

[35] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

[36] A. J. Quinn, B. B. Bederson, T. Yeh, and J. Lin. Crowdflow: Integrating machine learning with mechanical turk for speed-cost-quality flexibility. Technical Report HCIL-2010-09, University of Maryland, College Park, 2010.

[37] R. E. Schapire. The boosting approach to machine learning: An overview. `https://www.cs.princeton.edu/courses/archive/spring07/cos424/papers/boosting-survey.pdf`, 2002.

[38] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[39] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the Symposium on Principles of Programming Languages*, 2010.

[40] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *Proceedings of the Conference on Constraints in Computational Linguistics*, 1994.

[41] M. Veanes and N. Bjørner. Symbolic automata: The toolkit. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2012.

[42] R. A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5), 1974.

A. Appendix

To provide intuition on the type of results produced by our technique we present two examples. These are not necessarily the the highest fitness results; some of these are sampled after several generations. The complexity of the evolved results is due to two factors: the regular expressions are obtained using the classical transformation from automata which in general produces large results, and the classifiers have to take into account many corner cases.

```
Task: Email
Fitness: 0.935714285714286
Result:
^(([+-0-9A-Z\_a-z](([+-0-9A-Z_a-z]|\(\)*[+-0-9A-Z_a-z]))*
(@[-0-9A-Za-z]([-0-9A-Za-z])*.[A-Za-z][A-Za-z]|\(\)*@[-0-
9A-Za-z]([-0-9A-Za-z])*@[-0-9A-Za-z]([-0-9A-Za-z])*.[A-Za
-z][A-Za-z])(([A-Za-z]|))*|[+-0-9A-Z\_a-z](([+-0-9A-Z_a-z
]|\(\)*[+-0-9A-Z_a-z]))*(.[^\x0A][-0-9A-Z_a-z]|\(\)*(([\0
-\x09\x0B-*,./:-?[]^‘{-\uFFFF][-0-9A-Z_a-z]|@_)|@[-0-9A-Z
a-z]([-0-9A-Za-z])*_))(([-0-9A-Z_a-z]|\[^\x0A][-0-9A-Z_a-
z]))*@[-0-9A-Za-z]([-0-9A-Za-z])*.[A-Za-z][A-Za-z](([A-Za
-z]|))*))$

Task: Phone numbers
Fitness: 0.897959183673469
Result:
^((((((([02-9]|+1([0-9]|[\x20-.][0-9]))|((|+1((|[\x20-.](
))[0-9])([0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0
-9][0-9][0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9])|(([0
2-9]|+1([0-9]|[\x20-.][0-9]))|((|+1((|[\x20-.]())[0-9])[0
-9][0-9][\x20-.]([0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9
][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9]))|(([02-9]|+1([0-
9]|[\x20-.][0-9]))|((|+1((|[\x20-.]())[0-9])[0-9][0-9])((
[0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][\x20-
.][0-9][0-9][0-9][0-9])|[\x20-.]([0-9][0-9][0-9][0-9][0-9
][0-9][0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9]))
)|1(_)*((([0-9][0-9][0-9]([0-9][0-9][0-9][0-9][0-9][0-9](
[0-9])?|([0-9][0-9][\x20-.]|[0-9][0-9][0-9][\x20-.])[0-9]
[0-9][0-9][0-9])|([0-9]([0-9][0-9][0-9][0-9][0-9][0-9][0-
9][0-9][0-9]|[0-9][0-9][0-9][0-9][0-9][\x20-.][0-9][0-9][
0-9][0-9]))|(([0-9][0-9][\x20-.]|[0-9][0-9][0-9][\x20-.])
|([0-9][0-9][0-9][\x20-.])([0-9][0-9][0-9][0-9][0-9][0-9]
[0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9]))|(([0-
9][0-9])|[0-9][0-9][0-9]))|([0-9][0-9][0-9]))(([0-9][0-9]
[0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-
9][0-9][0-9])|[\x20-.]([0-9][0-9][0-9][0-9][0-9][0-9][0-9
]|[0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9]))))|1(_)*[\
x20-.](1_)*((((((((([0-9][0-9]([0-9][0-9][0-9][0-9][0-9][0
-9]([0-9])?|([0-9][0-9][\x20-.]|[0-9][0-9][0-9][\x20-.])[
0-9][0-9][0-9][0-9])|(([0-9][\x20-.]|([0-9][0-9][\x20-.])
([0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][\x20
-.][0-9][0-9][0-9][0-9]))|1[0-9][0-9][0-9]([0-9][0-9][0-9
][0-9][0-9][0-9]([0-9])?|([0-9][0-9][\x20-.]|[0-9][0-9][0
-9][\x20-.])[0-9][0-9][0-9][0-9]))|(([02-9]|+1([0-9]|([0-
9]))|1([0-9])([0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-
9]|[0-9][0-9][0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9])
)|(1([0-9][0-9][\x20-.]|[0-9][0-9][0-9][\x20-.])|(([02-9]
|+1([0-9]|([0-9]))|1([0-9])[0-9][0-9][\x20-.])([0-9][0-9]
[0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-
9][0-9][0-9]))|(+1[\x20-.]|1[\x20-.])(([0-9]|([0-9])([0-9
][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9]
[0-9][0-9][\x20-.][0-9][0-9][0-9][0-9])|([0-9]|([0-9])[0-
9][0-9][\x20-.]([0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9]
[0-9][0-9][\x20-.][0-9][0-9][0-9][0-9])))|(([0-9])|([0-9]
[0-9]))(([0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0
-9][\x20-.][0-9][0-9][0-9][0-9])|[\x20-.]([0-9][0-9][0-9]
[0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-9][0-
9][0-9])))|((1([0-9][0-9])|[0-9][0-9][0-9]))|(([02-9]|+1(
[0-9]|([0-9]))|1([0-9])[0-9][0-9]))|(+1[\x20-.]|1[\x20-.]
)([0-9]|([0-9])[0-9][0-9]))(([0-9][0-9][0-9][0-9][0-9][0-
9][0-9]|[0-9][0-9][0-9][\x20-.][0-9][0-9][0-9][0-9])|[\x2
0-.]([0-9][0-9][0-9][0-9][0-9][0-9][0-9]|[0-9][0-9][0-9][
\x20-.][0-9][0-9][0-9][0-9])))))$
```