Regex+: Synthesizing Regular Expressions from Positive Examples

Mark Barbone
UC San Diego
mbarbone@ucsd.edu

Elizaveta Pertseva
UC San Diego
epertsev@ucsd.edu

Joey Rudek
UC San Diego
jrudek@ucsd.edu

Nadia Polikarpova
UC San Diego
npolikarpova@eng.ucsd.edu

Abstract

Regular expressions are a popular target for programming by example (PBE) systems, which seek to learn regexes from user-provided examples. Synthesizing from only positive examples remains an unsolved challenge, as the unrestricted search space makes it difficult to avoid over- and under-generalizing. Prior work has approached this in two ways: search-based techniques which require extra input, such as user feedback and/or a natural language description, and neural techniques. The former puts an extra burden on the user, while the latter requires large representative training data sets which are almost nonexistent for this domain. To tackle this challenge we present Regex+, a search-based synthesizer that infers regexes from just a few positive examples. Regex+ avoids over/under-generalization by using minimum description length (MDL) learning, adapted to version space algebras in order to efficiently search for an optimal regex according to a compositional MDL ranking function. Our evaluation shows that Regex+ more than triples the accuracy of existing neural and search-based regex synthesizers on benchmarks with only positive examples.

1 Introduction

Suppose you are a data scientist searching for Brazilian CNPJs (company identification numbers) in a large document. You quickly identify two initial CNPJs, `60.701.190/0001-04` and `02.916.265/0001-60`, but manually searching for more examples is prohibitively expensive, and writing regular expressions has a steep learning curve [1].

One possible solution is to input the two examples into a regex synthesizer and get a pattern-matching expression. However, to our knowledge there exists no one-shot regex synthesizer that works reliably well with only few positive examples, and coming up with good negative examples is unintuitive [7].

PL'22, January 01–03, 2018, New York, NY, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The above scenario can be extended to a multitude of identification tasks which involve filtering data based on only very few known matches, illustrating a void in the regex synthesis field. To fill this need we introduce Regex+, a novel regex synthesizer that relies on *MDL learning* and *version space algebras* to generate results from only positive examples. For example, from the two CNPJ inputs mentioned above, in less than a second, Regex+ outputs

```
\d{2}\.\d{3}\.\d{3}/0001-\d{2}
```

the correct pattern for Brazilian CNPJs.

Challenge: Ranking Function. The first challenge in generating regular expressions is that the relatively unrestricted search space demands a high quality ranking function which penalizes both overly specific and overly generic regexes. For example, both

`.*` and `(60\.701\.190/0001-04)|(02\.916\.265/0001-60)`

are valid, but unhelpful, solutions for the given inputs. Existing techniques for regex synthesis include inferring a ranking function using machine learning [2, 3, 9, 10], synthesizing the simplest valid regex in a custom DSL [3, 6], and asking users for clarification [18]. However, these strategies do not extend well to synthesis from few positive examples: machine learning models struggle to generalize due to a lack of positive-examples-only training data, and overly simple regexes are often unhelpful to users.

Solution: MDL. To balance simplicity and specificity, Regex+ employs *minimum description length (MDL) learning* [15], where the main idea is to minimize the total amount of information required to both describe the regex and to identify the positive examples among all strings accepted by the regex. More concretely, for a candidate regex r, the ranking function includes a *simplicity* term and a *specificity* term. The simplicity term corresponds to the probability of generating r from a fixed probabilistic context-free grammar (PCFG); this probability decreases as the regex gets larger or uses more complex constructs. The specificity term corresponds to the probability of generating the observed positive examples by sampling strings accepted by r; this probability decreases as the regex gets more general (accepts more strings).

Challenge: Search Strategy. Once a suitable ranking function is computed, efficiently searching for the best regex presents a second challenge. Since there is a vast realm of acceptable regexes, computing the minimal description length for each regex is infeasible. Even less-naive ranking strategies such as top-down A* search face combinatorial explosion, as optimal regexes can have many components (for example, the CNPJ regex has 18). Neural synthesizers approach the combinatorial explosion using beam search [17]. However, this search strategy is not applicable to our ranking function, as the correct regex might start with an unlikely component such as an optional.

Solution: Compositional Ranking and VSAs. To overcome this challenge, we propose a compositional formulation of the MDL ranking function. In this formulation, the weight of a regex is the sum of the weights of its atomic components. We then use *version space algebras* (VSAs) [6, 8, 11, 16] in order to compactly represent the space of all regexes that fit the positive examples. Once a VSA has been constructed, the compositional weights allow us to efficiently extract the best regex by simply finding the cheapest path through a DAG.

Evaluation. We evaluate our tool, Regex+, on 122 real-world Stack Overflow benchmarks from the Regel benchmark suite [3], as well as 22 benchmarks with augmented positive examples. Our results demonstrate that MDL learning surpasses prior machine learning approaches on both benchmarks suites, achieving 77% accuracy on the latter.

Contributions. In summary, this paper contributes the following:

- An MDL ranking function for selecting regexes, accounting for both simplicity and specificity
- A VSA-based algorithm for efficient search guided by the MDL ranking function
- Regex+, an implementation of the above that is able to synthesize correct regexes based on only positive examples outperforming other regex synthesizers, neural and enumerative.

2 Overview

2.1 MDL Learning for Regular Expressions

To better motivate the core components of our technique, we turn to a more familiar example: email addresses. Imagine a user wants to determine the format of UCSD student emails. Having read the Regex+ paper, they know three examples: `epertsev@ucsd.edu`, `jrudek@ucsd.edu`, and `mbarbone@ucsd.edu`.

There are many regexes that match these examples, which brings us to the question of how to pick the best (or the top k) regexes to show to the user? One of the most popular approaches in program synthesis is to select the shortest valid program. However in the case of regexes, the shortest option is `.*`, which is not helpful to the user as it is overly general: it permits all inputs.

The choice of shortest regex can be improved by restricting the DSL, a common technique used by prior regex synthesizers [4, 12]. Noting that learning all possible regular expressions from only positive examples is theoretically impossible [5], we similarly restrict our regex grammar, disallowing arbitrary alternation, and instead including a restricted class of optionals. For more details see Sec. 3.1. However, this only marginally improves the results. The simplest regex in our DSL for the emails example is

```
[a-z]+@[a-z]+\.[a-z]+
```

which is still overly generic.

Existing approaches remedy this issue by requiring negative examples or natural language descriptions. However, negative examples are inconvenient to come up with [7]. In this case the user runs into an almost infinite and overwhelming space of invalid emails: capital letters in emails, emails without `@`, emails not ending in `ucsd.edu`, etc. Similarly, writing natural language descriptions can be problematic and verbose.

Recent state of the art neural synthesizers like Github Copilot have attempted to approach the problem from a different angle by remedying the lack of information from inputs with an abundance of training data. However their response often underfits or overfits the regex as they struggle to capture the specific behavior of the input examples. For the three given inputs Github Copilot outputs

```
[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+
```

which is too permissive. Note that in many cases neural synthesizers are also unsound as there is no guarantee that the resulting regexes will actually match the inputs.

Our first *core insight* is that the over-generalization problem can be tackled by viewing the regex synthesis problem as an instance of minimum description length (MDL) learning. MDL learning [15] is a general framework for statistical learning which involves selecting the hypothesis that minimizes the total information necessary to encode both the hypothesis, and the data given the hypothesis. As a result, an MDL learner is forced to trade off the *simplicity* of the hypothesis and its *specificity* to the data. Specializing to regex synthesis, we select the best regex for the provided inputs by optimizing a combination of the probability of forming the selected regex and the probability of inputs given the regex, thus achieving a simple regex that is not overly permissive. Specifically, we compute information as the negative log of probability, minimizing the following over regexes r:

$$\text{Weight}_{MDL}(r) = -\log\left(P(r)\right) - \log\left(P(\mathit{inputs} \mid r)\right)$$

in which $-\log(P(r))$ is smaller for simpler regexes, and $-\log(P(\mathit{inputs} \mid r))$ is smaller for more specific regexes.

For the emails example, if we only minimize the simplicity ranking we get

```
[a-z]+@[a-z]+\.[a-z]+,
```

or the shortest regex in our DSL. On the other hand if we only minimize the specificity score Regex+ outputs

```
(epertsev)?(jrudek)?(mbarbone)?@ucsd\.edu,
```

which only permits a few inputs and is unhelpful to the user. However if we combine both of the rankings the top five choices are

1. `[a-z]+@ucsd\.edu`
2. `[a-z]{2,}@ucsd\.edu`
3. `[a-z]+e[a-z]?@ucsd\.edu`
4. `[a-z]{3,}@ucsd\.edu`
5. `[a-z]{2,}e[a-z]?@ucsd\.edu`

Where the top choice is clearly the desired result.

Scores. The top regex is chosen by Regex+ because it strikes the best balance between simplicity and specificity: its specificity score is 82.3 and its simplicity score is 55.4 giving the regex a score of 137.7 overall, the minimal score given these inputs.

The second from the top regex is more specific than the first as it mandates the use of at least two [a-z] thus its specificity is lower: 82.2. However it is also less simple then first as it adds an extra regex atom thus the simplicity score increases to 57.8 and the overall score becomes 140.0.

Similar checks and balances can be seen when analyzing the score of the most specific and most simple regex. The most simple regex

```
[a-z]+@[a-z]+\.[a-z]+,
```

has a simplicity score of 20.9 but a specificity score of 171.0 as it is too permissive thus giving it a total score of 191.0. On the other hand

```
(epertsev)?(jrudek)?(mbarbone)?@ucsd\.edu,
```

has a specificity score of 6.2 but a simplicity score of 173.5 giving it a total score of 179.7, which is once again much bigger than the top regex. The email example demonstrates that by balancing specificity and simplicity, MDL ranking is able to penalize both over permissive and overly strict regexes and choose the regex that is most helpful to the user.

2.2 MDL-Guided Search

Given an MDL score for each regex we run into the problem of efficiently enumerating all of the candidates and selecting one with a minimum $\mathrm{Weight}_{MDL}(r)$. We propose an approach based on version space algebras [16]. Our algorithm constructs a DAG data structure that represents all valid solutions, and then finds the optimal solution using a graph search algorithm (shortest path in a DAG). In the rest of this section we will first explain how to build a DAG that represents all valid regexes, and then show how to find the best one in a compositional way.

2.2.1 Representing valid solutions. To illustrate our VSA construction algorithm, we walk through another example with shorter inputs (to limit the size of the VSA) where the user is attempting to synthesize a regular expression that matches all numbers 10–19. As an input the user enters `15` and `18`. First, for each input, a version space algebra (VSA) is constructed, encoding the set of all regular expressions that match it. The VSA consists of a directed acyclic graph with designated start and end nodes, in which each edge is labelled with an atomic regex. The possible answer regexes are paths in the DAG from the start node to the end node. The VSAs generated for `15` and `18` can be found in Fig. 1. The VSAs for both of the inputs include 3 nodes (1 per each character and 1 for the end token). Each path represents a set of regexes that only accept the portion of the input from the source node until the destination, not including the destination. For example, in the VSA in Fig. 1 for the input `15` the edge from 1 to 5 represents possible regexes that accept 1.

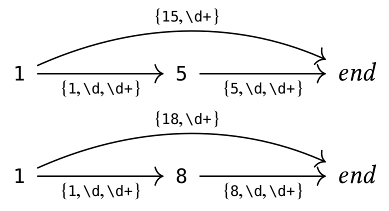

Figure 1. VSAs for inputs `15` and `18`. We omit `[a-zA-Z0-9]` and `[a-zA-Z0-9]+` for simplicity.

Next, we *intersect* the VSAs yielding a VSA representing the regexes that accept all of the inputs. For example, the intersection of the two VSAs from Fig. 1 is shown in Fig. 2. As in prior work [16], to intersect two VSAs we take all pairs of their nodes, and only keep those edges that are present in both VSAs. Unlike prior work, however, we also introduce *optional edges* to account for atomic regexes that are not shared by all inputs. For example, in Fig. 2 there is an optional edge `5?` from the node `(5,8)` to the node `(end,8)`, which corresponds to consuming the substring `5` from the first input (`15`) and consuming nothing from the second input. Note that when allowing optionals, the resulting DAG grows quite large, thus in Fig. 2 we omit the full set of labels for each edge, instead showing only the most specific one.

2.2.2 Finding the optimal solution. Upon constructing the DAG, a naive approach would be to enumerate all paths and then compute the MDL score for each one. However, this is prohibitively expensive: when one includes optionals the number of path grows superexponentially with each added input, and as shown in Fig. 2, even with a few inputs there is a proliferation of paths.

Thus our second *core insight* is that we can instead define a compositional approximation of the MDL score: we assign a score to each atomic regex (aka each edge of the DAG), and score of a path (the concatenation of the atomic regexes)

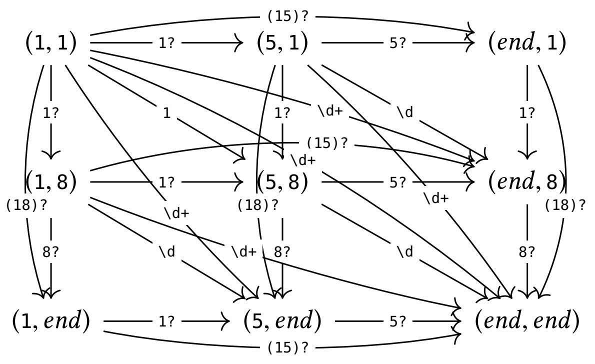

Figure 2. The intersection of VSAs from Fig. 1

becomes the sum of the scores of its edges. Similarly, during intersection, the score of the intersection of two edges may be computed from the scores of the two edges.

In our running example, we compare three likely candidate regexes, `1\d`, `\d\d`, and `\d+`, all of which appear along the diagonal in Fig. 2. (Although omitted for space, `\d` is an allowed label for the edge from (1, 1) to (5, 8).)

For the DAG above using the ranking function described in more detail in Sec. 3.2 Regex+ assigns

$$\text{Weight}_{MDL}(\texttt{1}) = 5.809$$

$$\text{Weight}_{MDL}(\texttt{\\d}) = 7.010$$

$$\text{Weight}_{MDL}(\texttt{\\d+}) = 17.295$$

The weight of `\d` is the same for all of its occurrences in this VSA.

Since in log space multiplication becomes addition we get that

$$\text{Weight}_{MDL}(\texttt{1\\d}) = 5.809 + 7.010 = 12.819$$

$$\text{Weight}_{MDL}(\texttt{\\d\{2\}}) = 7.010 + 7.010 = 14.020$$

Thus we can now represent the extraction problem as finding the shortest/cheapest path. The top five results for this example, together with their total composed weights, are presented below:

1. [12.819] `1\d`
2. [14.021] `\d\d` (simplified to `\d{2}`)
3. [17.295] `\d+`
4. [18.308] `1\d+`
5. [19.413] `1[A-Za-z0-9]`

The first is the one desired by the user, as it accepts all/only numbers 10-19.

2.3 Simplification and Deduplication

Since regexes can include arbitrary-count repetition, our grammar admits different representations, with different scores, for equivalent languages. For example, `1?[0-9]+` is equivalent to `[0-9]+`, but the latter would have a lower score as it is ostensibly simpler.

regex ::= ε | *edge regex*
edge ::= *atom* | `(`*atom*`)?`
atom ::= *class* | *class*`+` | `literal(`*string*`)`
class ::= `[0-9]` | `[a-z]` | `[A-Z]` | `[a-zA-Z]` | `[a-zA-Z0-9]`

Figure 3. The restricted regex grammar

While the duplication poses no computational toll, as the compositional ranking function facilitates the reuse of component calculations, it does pose a problem when outputting the top-k candidate regexes, since some of them could be duplicated.

Thus in order to avoid these equivalences, we gather $C \cdot k$ (in our implementation, $4 \cdot k$) of highest-ranked regexes and then we use the library `greenery` [14] to compare their languages, eliminate duplicates according to various reduction rules, and collect the top-at-most-k unique output regexes. Greenery also allows us to simplify complicated regexes via rewrites such as

`\d\d\d` ⇒ `\d{3}`

`(\d)?(\d)?(\d)?` ⇒ `\d{0,3}`

3 Algorithm

3.1 Grammar

Inspired by prior work [4, 6, 16] on synthesis of regexes and string transformations, we represent a regex as a concatentation of *atoms*, individual simple components which include literal strings, character classes, etc. We extend prior grammars by including optionals. The detailed DSL is shown in Fig. 3. Since the specific choice of character classes is inessential to our algorithm, it is simple to add more character classes such as `[0-9A-F]` or `[A-Za-z0-9_]` and tailor Regex+ to produce regexes for a specific domain.

The restrictions on our grammar are motivated by the theoretical results that show that with an unrestricted grammar synthesis from only positive examples is impossible [5]. As a result our grammar excludes complex regex features such as `not` and `startwith()`. These components are unlikely to be correctly inferred from only a few examples, thus excluding them not only allows for more efficient synthesis, but also likely improves our output.

3.2 VSAs and Intersection

To facilitate efficient search, we encode the set of possible regexes in a version space algebra (VSA). In Regex+, a VSA consists of a directed acyclic graph (V, E), where V, the set of vertices, is labelled by source positions in the input strings, and E, the set of edges, is labelled by possible atomic regexes. The possible regexes are represented by paths through the directed graph from the start to end nodes. Examples are shown in Figures 1 and 2.

VSAs [16] are a representation-based search technique which works in three steps:

1. A VSA is built for each input, encoding the set of all possible solutions given only that input.
2. These VSAs are intersected, yielding a single datastructure that encodes the intersection of all the individual solution sets.
3. Lastly, the best solution (according to our cost metric) is extracted from the VSA.

Initial VSA Construction. To construct a VSA from an input word, we create vertices for each position in the word, and between each two vertices we put edges for each atomic regex from our grammar that matches that part of the word. More formally, given an input word $W = w_0 w_1 \ldots w_{n-1}$:

$$V_W := \{v_i \mid 0 \le i \le n\}$$
$$E_W := \bigcup_{0 \le i < j \le n} \{(v_i \to v_j;\ \mathit{atom}) \mid \mathit{atom} \text{ matches } w_i \ldots w_j\}$$

where $(v_i \to v_j;\ \mathit{atom})$ denotes an edge from v_i to v_j labelled by the regex component *atom*.

VSA Intersection. Next, the VSAs representing each input are intersected. From two VSAs (V_1, E_1) and (V_2, E_2), we build their intersection $(V_\cap, E_\cap)$ as follows. Since we want a path through $(V_\cap, E_\cap)$ to be *both* a path through (V_1, E_1) and (V_2, E_2), the vertices in $V_\cap$ are pairs of vertices from V_1 and V_2. Then the edges from (v_1, v_2) to (w_1, w_2) consist of the intersection of the sets of edges v_1 to w_1 in E_1 and edges v_2 to w_2 in E_2.

Optionals. With the intersection mechanism as described so far, each edge in the intersection VSA always consumes a non-empty substring of both inputs. To support optionals, we must allow edges that consume characters only from one of the inputs. We account for these components by taking each edge $(v \to w;\ \mathit{atom})$ in one of the input VSAs, turning *atom* into an optional, and adding it as an extra edge in the intersected VSA between vertices of the form (v, a) and (w, a), for each vertex a of the other input VSA. These optional edges can be seen in Fig. 2. All the horizontal optional edges, such as `(15)?` from `(1,1)` to `(end,1)`, consume a part of the first input (`15`) but not the second (`18`), while all of the vertical optional edges consume a part of the second input but not the first.

The process can be formally summarized as

$$\begin{aligned}
V_\cap &:= V_1 \times V_2\\
E_\cap &:= \{((v_1, v_2) \to (w_1, w_2);\ \mathit{atom})\\
&\qquad \mid (v_1 \to w_1;\ \mathit{atom}) \in E_1,\ (v_2 \to w_2;\ \mathit{atom}) \in E_2\}\\
&\quad \cup \{((v_1, v_2) \to (w_1, v_2);\ (\mathit{atom})\texttt{?})\\
&\qquad \mid (v_1 \to w_1;\ \mathit{atom}) \in E_1,\ v_2 \in V_2\}\\
&\quad \cup \{((v_1, v_2) \to (v_1, w_2);\ (\mathit{atom})\texttt{?})\\
&\qquad \mid v_1 \in V_1,\ (v_2 \to w_2;\ \mathit{atom}) \in E_2\}
\end{aligned}$$

Finally, after our ranking function assigns a weight to each edge, we extract the top regexes using shortest path in a DAG.

3.3 Ranking

We use an MDL learner composed of simplicity and specificity scores. We approximate minimum description length with a compositional ranking function by computing these scores for each regex atom separately and then multiplying the probabilities (or equivalently summing up the log scores) of the atoms to get the full score of the regex given the inputs.

In other words, we use Bayes rule and compute

$$P(r|\mathit{inputs}) \sim P(\mathit{inputs}|r) \cdot P(r)$$

where $P(\mathit{inputs}|r)$ corresponds to the specificity scores and $P(r)$ corresponds to simplicity. In log space, this allows us to compute the two scores independently and then add them. However, as log space can be unintuitive, we show all our calculations using probabilities. We first explain how specificity scores are computed, followed by simplicity scores.

3.3.1 Specificity.

Composition. Since our goal is to use a shortest path algorithm, we seek to compute the specificity scores or $P(\mathit{inputs}|\mathit{regex})$ by focusing on one input at a time and breaking each regex into its atomic components. Ideally we want the combined probability to be the product of atomic probabilities; however, in reality its computation is more complex: for a regex $r = r_1 r_2$ and an input string s, we can compute

$$P(s|r) = \sum_{s_1, s_2 | s_1 ++ s_2 = s} P(s_1|r_1) P(s_2|r_2)$$

where the sum is over all ways to break s up into pieces s_1 and s_2.

Nevertheless we claim that we can simplify this sum to a single product with out losing any solutions, taking only one (non-zero) term from the sum. In cases where the split of the string into matching components is unambiguous, such as when $r =$ `[a-z]\d+` and the input string is `a9`, this approximation is exact as the sum simplifies to a product of two probabilities

$$P(s|r) = P(s_1|r_1) P(s_2|r_2)$$

since all other partitions of s yield probabilities of 0 for the given atomic regexes.

The simplification underapproximates the probabilities only in cases where the split is ambiguous. For example, when $r =$ `[a-z]+[a-z]+` and the input string is `xyz`, it can be split both as `xy` followed by `z` and as `x` followed by `yz`, so the true probability is

$$\begin{aligned}
P(\texttt{xyz}|\texttt{[a-z]+[a-z]+}) &= P(\texttt{x}|\texttt{[a-z]+})P(\texttt{yz}|\texttt{[a-z]+})\\
&\quad + P(\texttt{xy}|\texttt{[a-z]+})P(\texttt{z}|\texttt{[a-z]+}).
\end{aligned}$$

However, the ambiguous regexes similar to `[a-z]+[a-z]+` are already expressed in our grammar by unambiguous regexes such as `[a-z][a-z]+`, whose probability will be correctly computed according to the simplification and higher than the underaproximation. Thus our algorithm will still find a correct regex.

After establishing the theoretical basis for splitting up the probability by splitting up the input among the atomic regexes, we can implement the split by relying on our VSA representation. Since in the VSA each node already corresponds to a position in the input string, each path has a canonical splitting of the inputs along those regex components. For example, considering the path `1\d` through the VSA for `15` in Fig. 1, the first atomic regex `1` consumes `1`, and `\d` consumes `5`, so we can naturally split the input between these atomic regexes.

Going forward, two challenges remain to be solved: how to actually compute the probabilities for each atomic regex, and how to factor in multiple inputs.

Probabilities. The first challenge can be symbolically expressed as computing $P(i|r)$ for each atomic regex r, where i is a single matching string.

For this task the atomic regexes can be split into two classes: finite such as `\d` and infinite such as `\d+`. In both cases, to compute the probability we consider a finite state automaton for the component and assign a probability to the input by assuming that each transition in the automaton is equally likely.

For example, consider the atomic regex $(a)?$, whose NFA representation is in Fig. 4. There are two possible transitions from beginning to end so $P(i|\texttt{(a)?}) = \frac{1}{2}$, for any matching input. In general, for any finite regex r matching $|r|$ strings (including literal strings, optional literal strings, character classes, and optional character classes) the probability can be computed as 1 over the number of possible transitions:

$$\frac{1}{|r|}.$$

For the atomic regexes that match infinitely many strings, we consider an input as a sequence of $n+1$ transitions in the automaton, where n is the input's length, and thus only multiply the probabilities of each of the $n+1$ transitions.

For example consider `[a-z]+`, which has a loop in its NFA as can be seen in Fig. 5. Any input of length n makes $n+1$ transitions (accounting for the transition to the end) through this NFA. For the first transition we have 26 options and for the rest we have 27, thus the probability for any matching input of length n is

$$\frac{1}{26} \cdot \left(\frac{1}{27}\right)^n.$$

The infinite atomic regexes in our grammar can be further subdivided into $C+$ and $C*$ (the simplified form of $(C+)?$), for a character class C. The probability of an input of length

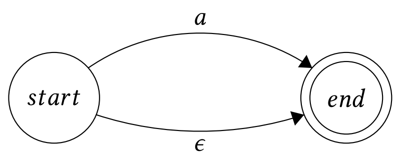

Figure 4. NFA for `(a)?`

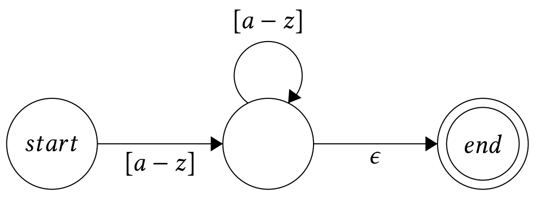

Figure 5. NFA for `[a-z]+`

n given $C+$ can be computed as

$$\frac{1}{|C|} \cdot \left(\frac{1}{|C|+1}\right)^n$$

and the probability of the input given $C*$ would be

$$\left(\frac{1}{|C|+1}\right)^{n+1}.$$

Multiple inputs. Finally, to account for multiple inputs, we assume that the probabilites $P(i_1|r)$ and $P(i_2|r)$ are independent as there is no reason for them to influence each other. Thus the overall probability can be computed as the product

$$P(i_1, \ldots, i_n|r) = \prod_{j=1}^{n} P(i_j|r).$$

3.3.2 Simplicity. The simplicity score of a single regex atom is derived using the probability of a regular expression, computed using the Probabilistic Context Free Grammar (PCFG) shown in Fig. 6. The PCFG encodes domain knowledge about the simplicity of regexes: `[0-9]` is simpler than `[A-Za-z0-9]` which is simpler than a constant which is simpler than an optional. One can use the PCFG to derive the simplicity weight for a single atom by following the PCFG until no transitions are left. For example let us consider computing the simplicity weight for `([a-z])?` and `[a-z]`. To find the score of the first atom we take

$$P(\texttt{Atom}) * P(\texttt{Opt}) * P(\texttt{Char Class}) * P(\texttt{[a-z]})$$

$$0.95 * 0.25 * 0.40 * \frac{57}{60} = 0.09025$$

For the second atom:

$$P(\texttt{Atom}) * P(\texttt{Char Class}) * P(\texttt{[a-z]}$$

$$0.95 * 0.30 * \frac{57}{60} = 0.27$$

Clearly `[a-z]` is simpler than `([a-z])?`, and thus has a higher probability.

`Regex`	$\longrightarrow$	`end of string`	0.05
		`Atom Regex`	0.95
`Atom`	$\longrightarrow$	`Optional`	0.25
		`Char Class +`	0.15
		`Char Class`	0.30
		`Literal`	0.30
`Literal`	$\longrightarrow$	`Printable ASCII char`	$\frac{1}{95}$
`Char Class`	$\longrightarrow$	`[a-z]/[0-9]/[A-Z]`	0.95
		`[A-Za-z]`	0.03
		`[A-Za-z0-9]`	0.02
`Optional`	$\longrightarrow$	`Char Class`	0.4
		`Char Class +`	0.2
		`Literal`	0.4

Figure 6. The Simplicity PCFG

By using a PCFG, the simplicity score can also be computed compositionally: for a regex r built from the atomic components $r_1 \dots r_n$, its probability given by the PCFG is

$$P(r) = 0.05 \cdot \prod_{i=1}^{n} 0.95P(r_i),$$

so analogously to the specificity scores, it can be feasibly computed by multiplying a function of the edges in the VSA, or in log space, adding their negative logs.

4 Evaluation

For our evaluation we set out to answer the following questions:

(RQ1) How does Regex+ perform on real-world regex problems compared to state-of-the-art neural and non-neural synthesizers?
(RQ2) How does Regex+ perform on regex problems tailored to be solved with only positive examples?
(RQ3) Are both simplicity and specificity weights needed to generate good regexes?

General Setup. All of the benchmarks ran by us are run on a consumer laptop with an Intel(R) Core(TM) i7-5557U CPU and 16 GB of memory.

4.1 StackOverflow Study

Benchmark Suite. To investigate how Regex+ performs on real world problems we took the benchmark suite from Regel [3] benchmark suites: a battery of 122 questions scraped from Stack Overflow endowed with positive and negative examples and natural language descriptions. It is vital to note that the Stack Overflow benchmarks are meant for multi-modal synthesis, using a combination of NLP, positive, and negative examples, and thus often provide a minimal number of positive examples, in some cases only one. For many cases, even human experts struggle to find the correct regex from only the examples provided. We proved this through an informal study where we asked 8 programmers who claimed reasonable knowledge of regex to solve 20 randomly-selected benchmarks given unlimited time. We received 14 correct answers, which is only 8.75% accuracy.

Since these benchmarks were posted on StackOverflow with 3 different modes of inputs; they were likely deemed as very difficult by the users. In fact, only 22 (18%) of the ground truth solutions to these benchmarks can be captured by our DSL.

Setup. For comparison, we list the results cited by the authors of Regel on the same benchmarks for DeepRegex (natural language only), Regel (enumerative synthesizer informed by machine learning) and Regel-PBE (one of Regel's baselines: an interactive synthesizer with positive and negative examples). The most relevant for comparing with Regex+ are the results from Regel-PBE, since both tools have no access to the natural-language description. Because the Regel-PBE results were obtained on a stronger computer (Intel Xeon(R) E5-1620 v3 CPU with 32GB of RAM), and multiple iterations per benchmark, we also report the results of a single Regel-PBE iteration without interaction when run with exactly the same setup we give Regex+, as well a second Regel-PBE run only given positive examples. In the last experiment we show Regel-PBE and Regex+ up to 4 positive examples (depending on how many examples the benchmark has) to test Regex+'s ability to synthesize the expected results with minimal information. All of these benchmarks are run with a 40 second timeout. Following prior work [10], we count a benchmark as correct if the result is in top five.

Results. The results are summarized in Tab. 1. Even when given less information, Regex+ triples the accuracy of both DeepRegex and the Single Iteration Regel-PBE. Furthermore, it more than quadruples the accuracy of Single Iteration Regel-PBE with negative examples withheld.

Interestingly, Regex+ does not have any overlap with Single Iteration Regel-PBE on which examples they get right; Fig. 7 shows an example comparing Regex+ to Regel-PBE given positive examples only. Note that although Regex+'s result is incorrect for benchmark 110 in Fig. 7, it still matches the examples well. Anecdotally we report that this is common: a large subset of the benchmarks that Regex+ gets wrong don't match the ground truth but are nevertheless likely helpful to users. This brings us to our second study.

4.2 Positive Examples Only Study

Benchmark Suite. To address the fact that the StackOverflow benchmark suite is not designed for synthesis from only positive examples, we consider the 22 benchmarks whose ground truth is within our DSL. We then augmented these

Table 1. Results on Stack Overflow Benchmarks Compared to other Synthesizers

	Benchmarks Correct	Percent	Positive Examples	NLP	Negative Examples	Multiple Iterations
Deep Regex	3	2.4%	X			
Regel	74	60.7%	X	X	X	
Full Regel- PBE	18	14.7%	X	X	X	X
Single Iteration Regel-PBE w/ out nlp	3	2.4%	X		X	
Single Iteration Regel-PBE w/ only positive examples	2	1.6%	X			
REGEX+	9	7.4%	X			

Benchmark 57
Inputs: `0,25`, `10,2`, `-7000`, `-175,33`
Correct: `-?\d+.?\d{0,2}`
Regel-PBE: `.*([0-9])`
REGEX+ Top 5:

1. `-?\d+,?\d{1,2}`
2. `-?\d+,?\d{0,2}`
3. `-?\d+,?\d+`
4. `-?\d*,?\d+`
5. `-?\d+,?\d*`

Benchmark 110
Inputs: `{foo}`, `{bar}`, `{nice}`
Correct: `\{.*`
Regel-PBE: `([{]).*`
REGEX+ Top 5:

1. `\{[a-z]+\}`
2. `\{[a-z]{2,}\}`
3. `\{[a-z]{3,4}\}`
4. `\{[a-z]{3,}\}`
5. `\{[a-z]{2,4}\}`

Figure 7. Left: REGEX+ produces the ground-truth regex as its second option (despite none of the examples having 0 decimals after the comma), whereas REGEL-PBE produces a regex that is too simple. Right: REGEL-PBE produces the ground-truth regex, whereas all answers given by REGEX+ are too constrained, since `.*` is not within our grammar.

22 benchmarks with 4 positive examples each, intended to make the results guessable by a human.

Setup. We compare the results of REGEX+ once again with a single iteration of REGEL-PBE. Since REGEL-PBE timed out on many of the 122 benchmarks we increase the timeout to 5 minutes. We also introduce Github Copilot, a state of the art neural program synthesizer. Although Copilot's primary use is not regexes, it is advertised as a general purpose synthesizer, so it can be used to synthesize regexes. It is also one of the few neural synthesizers that does not require verbose NLP descriptions. The prompt used for Copilot is shown in Listing 1, where the synthesizer auto completes the answer into `re.compile(r'')`. This was selected as the best of several prompts through repeated experimentation.

Results. A summary of results is presented in Tab. 2 and selected specific examples are listed in Tab. 3. Our results demonstrate that REGEX+ more than quintuples the accuracy of REGEL-PBE and more than triples that of Github Copilot.

Note that Copilot often produces unsound results. For example in Tab. 3, one can see that for benchmarks 3, 12 and 18 Copilot's results are not only wrong but they also do not accept the given inputs. A result of special interest is benchmark 3 where upon getting a large number Copilot automatically assumes it is a phone number. Out of these 22

```python
import re
"""
Regular expression which matches
- Example 1
- Example 2
- Example 3
- Example 4
"""
rx = re.compile(r'')
```

Listing 1. Prompt for Github Copilot

benchmarks, Copilot guesses

`\d{3}-\d{3}-\d{4},`

3 separate times, out of which it is correct only once, thus demonstrating the shortcomings of neural synthesizers in this domain.

REGEL-PBE also does poorly on these benchmarks. Although REGEL-PBE works well as a multi-modal synthesizer, without negative examples or natural language descriptions its enumerative synthesis quickly finds very short matching regexes, which are seldom correct. For example, for benchmark 4 its output, `startwith(<P>)` in its DSL, satisfies the specification but is too simple, as it ignores the other shared components between the 4 inputs.

The benchmarks also illustrated some of the shortcomings of Regex+. In benchmark 11 as seen in Tab. 3 our synthesizer fails to achieve `\d{1,4}`, this occurs because to generate `\d{1,4}` Regex+ must output

```
\d (\d)? (\d)? (\d)?
```

which is simplified to `\d{1,4}`. The initial regex is very long and thus due to the compositional ranking function it is more expensive than just `\d+`. One way to remedy this is to increase the specificity weight penalizing `\d+` or to add ranges to our grammar. Furthermore for benchmark 12 Regex+ only outputs 3 results as opposed to five. This occurs because 17 of the top 20 outputs generated by Regex+ are pruned as equivalent (see Sec. 2.3). One way to avoid this would be to either initially compute top 40 as opposed to top 20, or to prune a large portion of repeating paths prior to extraction. Nonetheless despite its shortcomings Regex+ outperforms both Regel-PBE and Copilot when given benchmarks with only positive examples, showing that it is a competitive with state of the art regex synthesizers.

Table 2. Results on Positive Examples Benchmarks Compared to other Synthesizers

	Benchmarks Correct	Percent
Regel-PBE	3	13.6%
Github-Copilot	5	22.7%
Regex+	17	77.3%

4.3 Ablation Study

Setup. Finally in order to answer RQ3 we preform an ablation study comparing full Regex+ to Regex+ with only specificity and simplicity.

Results. The results, summarized in Tab. 4, show that full MDL learning more than doubles the accuracy of Regex+ with only specificity or simplicity. In fact Regex+ with only specificity gets 0 benchmarks correct, as all of the outputs are way too tailored to the inputs. With only simplicity Regex+ gets 7 benchmarks correct, since as mentioned in Sec. 2, choosing the simplest regex within the DSL is a common strategy for many regex synthesizers. However Regex+ with only simplicity fails to account for the presence of constants and ranges.

Table 4. Results on Positive Examples Benchmarks for Specificity and Simplicity

	Benchmarks Correct	Percent
Only Specificity	0	0%
Only Simplicity	7	31.8%
Full Regex+	17	77.3%

5 Related Work

String transformations. Although not explicitly targeting regular expressions, string transformations as used in tools such as FlashFill [6] and BlinkFill [16] also involve identifying textual patterns. These tools synthesize string transformations in spreadsheets from input-output example pairs. As part of this synthesis problem, BlinkFill uses an InputDataGraph VSA to identify patterns in the inputs from a restrictive DSL; we apply a similar technique to the problem of general regex synthesis.

Non-neural synthesizers. Regular expression synthesis has also been approached in terms of more traditional program synthesis. AlphaRegex [9] uses an extension of top-down enumerative search to generate regular expressions over a very small alphabet (namely, the set $\{0, 1\}$). Regae [18] also employs top-down search, but instead of restricting its alphabet, it relies on extensive user interaction to narrow the search space.

Neural synthesizers. Many recent program synthesizers leverage machine learning, and the domain of regular expressions is no exception. In particular, machine learning is uniquely proficient at working with natural language descriptions. DeepRegex [10] attains 88.7% accuracy on its benchmark suite using only natural language descriptions of target regexes. Regel [3] also uses natural language descriptions, but only for ranking candidate regexes; because it uses traditional top-down synthesis to generate candidates, its output is guaranteed to be sound with respect to the given examples. Finally, GitHub Copilot [2] is not tailored to regular expressions, but as it is a general-purpose code synthesis tool it is completely capable of generating regular expressions. (It is worth noting, however, that Copilot regularly provides well-formed but unsound regexes; see Tab. 3.)

Pragmatic communication. Recent work on program synthesis with pragmatic communication [13] also seeks to increase the amount of information extractable from positive examples, by modelling synthesis as a two-player game. Their work uses a very simple DSL for controlling a robot on a finite grid. In future work, it would be interesting to explore whether this learning paradigm can be extended to regular expressions.

6 Limitations and Future Work

Regex synthesis from only positive examples is as yet an underexplored space. To the best of our knowledge, there is a limited number of other synthesizers tailored to the domain, as well as limited benchmarks facilitating our need to create our own battery of 22 benchmarks. We consider the creation of more extensive sets of benchmarks, as well as the subsequent benchmarking of our tool and others against them, to be important future work.

Table 3. Selected benchmarks run on Regex+, Regel-PBE, and Github Copilot. Green cells indicate that a tool output the golden regex, whereas red cells indicate that the tool provided a regex which does not match all provided inputs.

#	Examples	Golden regex	Regex+	Regel-PBE	Github Copilot
1	t alex ramon bob	[a-z]+	- [58.977] [a-z]+ - [67.026] [a-z]* - [67.504] [a-z]*[A-Za-z] - [67.504] [A-Za-z][a-z]* - [68.901] [a-z]*[0-9A-Za-z]	([a-z]){1,}	[a-z]{1}
3	091239567 098764321 093334445 094388270	09\d{7}	- [91.068] 09\d+ - [92.927] 09\d{7} - [93.092] 09\d{2,} - [94.850] 0\d+ - [95.116] 09\d{3,}	([0-9]){9}	\d{3}-\d{3}-\d{4}
4	Page 2 of 20 Page 18 of 44 Page 107 of 109 Page 7 of 0	Page \d+ of \d+	- [112.868] Page \d+ of \d+ - [119.182] Page \d+ of \d{1,3}	([P]).*	Page \d+ of \d+
10	abc-de-1234 f-q-7 oh-no-33 coo-l-007	[a-z]{1,3}-[a-z]{1,2}-\d{1,4}	- [125.824] [a-z]+-[a-z]{1,2}-\d+ - [127.073] [a-z]+-[a-z]{0,2}-\d+ - [128.842] [a-z]{1,3}-[a-z]{1,2}-\d+	([a-z]).*	[a-z]+-[a-z]+- \d+
11	236.1 8736.9999 0.43 72.875	\d+\.\d{1,4}	- [87.976] \d+\.\d+ - [95.357] \d*\.\d+ - [95.357] \d+\.\d*	([0-9]).*	\d+\.\d+
12	tw *mcaaa *qqee* hello	(*)?[a-z]{2,}(*)?	- [97.374] *?[a-z]+*? - [98.643] *?[a-z]+(e*)? - [99.629] *?[a-z]{2,}*? - [99.911] *?[a-z]+(e{2}*)? - [100.897] *?[a-z]{2,}(e*)?	.*([a-z]).*	[a-z]{2,3}
13	4567 +9752 3015 +1	\+?\d+	- [55.339] \+?\d+ - [62.720] \+?\d* - [63.638] \d?\+?\d+	.*([1-9])	\+\d{3,}
18	Hello Bob Sunil Kumar Jack Sparrow Oh No	[A-Z][a-z]+ [A-Z][a-z]+	- [151.342] [A-Z][a-z]+ [A-Z][a-z]+ - [156.365] [A-Z][a-z]+ [A-Za-z][a-z]+ - [156.365] [A-Za-z][a-z]+ [A-Z][a-z]+ - [157.762] [A-Z][a-z]+ [0-9A-Za-z][a-z]+ - [157.762] [0-9A-Za-z][a-z]+ [A-Z][a-z]+	([A-Z]).*	[A-Z][a-z]+
20	H347gjdj 8 sdjW23 Q3QW	[a-zA-Z0-9]+	- [101.271] [0-9A-Za-z]+ - [107.243] [0-9A-Za-z]* - [111.804] (H3)?[0-9A-Za-z]+ - [111.804] (Q3)?[0-9A-Za-z]+	.*([1-9]).*	[A-Z]{3}[a- z]{3}

Our chosen grammar encodes many useful regexes, but to maximize speed and code simplicity, it is extremely restrictive. Notably absent from it are the class `.*`, arbitrary choice including custom character classes, and repetitions and optionals of multiple concatenated components (e.g., `(\d, )+`). Further work should expand upon our grammar while maintaining reasonable runtime.

7 Acknowledgements

This work was supported by the National Science Foundation under Grants No. 1955457 and 1943623.

References

[1] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. 2017. Exploring Regular Expression Comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*

(Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 405âĂŞ416.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (07 2021).

[3] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020).

[4] Henning Fernau. 2009. Algorithms for learning regular expressions from positive data. *Information and Computation* 207, 4 (2009), 521–541.

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

[6] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[7] Tessa Lau. 2009. Why PBD systems fail: Lessons learned for usable AI. (10 2009).

[8] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156.

[9] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. *SIGPLAN Not.* 52, 3 (oct 2016), 70âĂŞ80. https://doi.org/10.1145/3093335.2993244

[10] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 1918–1923. https://doi.org/10.18653/v1/D16-1197

[11] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[12] Paul Prasse, Christoph Sawade, Niels Landwehr, and Tobias Scheffer. 2015. Learning to identify concise regular expressions that describe email campaigns. *J. Mach. Learn. Res.* 16, 1 (2015), 3687–3720.

[13] Yewen Pu, Kevin Ellis, Marta Kryven, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2020. Program Synthesis with Pragmatic Communication. *ArXiv* abs/2007.05060 (2020).

[14] qntm. 2022. Greenery. https://github.com/qntm/greenery.

[15] Jorma Rissanen. 1978. Modeling by shortest data description. *Automatica* 14, 5 (1978), 465–471.

[16] Rishabh Singh. 2016. BlinkFill: semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9 (06 2016), 816–827. https://doi.org/10.14778/2977797.2977807

[17] Volker Steinbiss, Bach-Hiep Tran, and Hermann Ney. 1994. Improvements in beam search. In *Third international conference on spoken language processing*.

[18] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. *Interactive Program Synthesis by Augmented Examples*. Association for Computing Machinery, New York, NY, USA, 627âĂŞ648. https://doi.org/10.1145/3379337.3415900