Efficient Bayesian Synthesis with Version Spaces

Mark Barbone*[1], Elizaveta Pertseva*[2], Taylor Berg-Kirkpatrick[1], and Nadia Polikarpova[1]

[1] UC San Diego, USA {mbarbone,tberg,npolikarpova}@ucsd.edu
[2] Stanford University, USA pertseva@stanford.edu

Abstract. Programming-by-Example embraces program synthesis from an incomplete specification: just a few examples. A program synthesizer must therefore choose among many programs that satisfy the specification, and traditional approaches—such as choosing the simplest program—ignore much of the information present in the input examples. In this work, we examine synthesis of regular expressions (*regexes*) from a few positive examples as an exemplar of a synthesis problem considered hopelessly underconstrained. We show that learning (a restricted class of) regexes from positive examples is possible using a Bayesian cost function, which takes into account not only the simplicity of a regex, but also its fit to the examples.

The technical challenge with this approach is how to efficiently search the space of all regexes to find one that minimizes the Bayesian cost. To address this challenge, we (1) adapt *version space algebra* techniques to enable use of the Bayesian cost function as a search objective, and (2) develop an efficient search algorithm we dub *guided version space exploration* (GVSE). We implement this approach in a synthesizer called Regex+, and evaluate it on a corpus of 231 regex learning tasks with human-generated examples. Our evaluation shows that Regex+ outperforms non-Bayesian baselines, and that GVSE can find optimal or near-optimal solutions while exploring only a fraction of the version space.

1 Introduction

The task of program synthesis is to find a program satisfying a given specification. Specifications, however, are often incomplete, leaving many candidate solutions for the synthesizer to choose from. This is especially true in Programming-by-Example, where the specification is given as a (typically small) set of input-output examples, which merely showcase representative program behavior. How should the synthesizer choose among the many candidate solutions?

A common option is to search for the shortest program [33,1,18], or to use other metrics of program complexity, defined by probabilistic grammars [23,6]. Importantly, all these metrics are all purely *syntactic*: they take into account the program, but not the specification; the specification is only used as a boolean test to filter out unacceptable programs.

* Equal contribution.

Such syntactic metrics are known to fall short when the synthesis problem is severely underspecified [2]. As an example, consider the task of learning a regular expression (*regex*) from positive examples, *i.e.* a set $\{s_1, \ldots, s_n\}$ of strings which must be accepted by the regex. Independently of the examples, the regex that accepts any string, `.*`, is a valid and simple solution, but is utterly unhelpful. At the same time, we observe that humans are often able to guess meaningful regexes from just a few positive examples, as long as the examples are *informative*. Intuitively, this requires extracting more information from the specification: rather than just using it as a boolean test, we need a *semantic* objective function, which balances the simplicity of a regex with its fit to the given examples.

Bayesian Synthesis. This intuition can be formalized using a *Bayesian* approach to program synthesis, where the objective is to find the program with the highest *posterior probability* for the given specification. Although Bayesian program synthesis has been explored in the past in restricted domains (*e.g.* [20,29]), efficient algorithms for optimizing the posterior remain a challenge: existing tools either use stochastic search, which is slow and unpredictable [20], or require explicitly enumerating all solutions before choosing the optimal one [29].

Efficient Search with Version Spaces. In this work, we propose a new search algorithm for Bayesian synthesis based on *version spaces* [21,17,32], targeting a restricted class of regexes, which we call *format regexes.* A version space is a graph data structure that compactly represents all candidate solutions to a synthesis problem. We observe that the structure of the version space allows us to decompose the Bayesian cost of a problem into a sum of costs of its subproblems, thereby reducing Bayesian synthesis to graph search.

Although version spaces are more compact than the search space they represent, their size still grows quickly with the number and length of examples. To overcome this scalability challenge, we propose a new search algorithm we dub *guided version space exploration* (GVSE). Unlike prior work, GVSE constructs the version space on the fly, only exploring promising solutions. We present two variants of the algorithm: (1) GVSE-A*, which uses an admissible heuristic to guide the search and preserves the optimality guarantee; and (2) GVSE-Beam, which gives up the optimality guarantee in exchange for more efficiency.

Regex+. We implement GVSE in Regex+, a regex synthesizer capable of learning format regexes from only a few positive examples. We evaluate Regex+ on a new benchmark suite with 231 regex synthesis tasks with informative positive examples. We demonstrate the effectiveness of the Bayesian cost function through a comparison with a syntactic cost function, as well as human learners. We also compare the two variants of GVSE with naive version space search and show that (1) GVSE-A* uses an order of magnitude less memory than naive search, which allows it to solve one extra benchmark; and (2) GVSE-Beam further reduces the synthesis time by over an order of magnitude, while staying within 98% of the optimal cost.

Contributions. In summary, this paper contributes the following:

- A Bayesian cost function for learning regexes from positive examples.

- *Guided version space exploration* (GVSE), a new search algorithm that efficiently explores version spaces to minimize a semantic cost function.
- Regex+, an implementation of GVSE that is able to learn format regexes from informative positive examples, on average exceeding the accuracy of human learners.

2 Overview

In this section we illustrate the three components of our approach—the Bayesian cost function, version spaces for regular expressions, and guided version space exploration—via a running example.

2.1 Bayesian Learning for Regular Expressions

Consider a user who wants to determine the format of student emails at their university and provides the following two input strings to Regex+:

```
lpage@stanford.edu        twoods@stanford.edu
```

There are infinitely many regexes that match these strings; how should we pick the one (or the top k) that is most likely to describe the intended format?

In program synthesis, the most common approach to dealing with ambiguous specifications is to return the shortest solution. This approach is a complete non-starter for our problem, however: the shortest solution to almost *any* regex synthesis problem is `.*`, which is not helpful to the user as it is overly general (it permits all strings). On the other hand, aiming for the *most specific* regex that matches our inputs—`(lpage|twoods)@stanford.edu`—would also be problematic, as it does not generalize beyond the two strings at all.

The DSL of Format Regexes. Another common approach to dealing with ambiguity is to restrict the target DSL (domain-specific language), *i.e.* the class of regular expressions that the synthesizer is considering [28,12,17,32]. In this paper we target the DSL of *format regexes*, which disallows arbitrary disjunctions, but supports optionals and a fixed set of character classes. As such, a format regex is a sequence of *factors*, each of which can be optional and contain either literals or possibly repeated character classes.[3]

Restricting the DSL, however, does not by itself solve the problem of over- and under-generalization. Assuming the default set of character classes in our tool (which includes lower-, upper-, and mixed-case letters, digits, and alphanumerics, but does not include the `.` wildcard), respectively, the shortest[4] and the most specific solutions for our running example are:

```
[a-z]+@[a-z]+\.[a-z]+                    (1)
(lpage)?(twoods)?@stanford\.edu          (2)
```

[3] The formal grammar for the DSL is given in Sec. 3.

[4] Other solutions of the same length can be obtained by replacing `[a-z]` with `[a-zA-Z]` or `[a-zA-Z0-9]`.

Clearly, neither of these solutions is satisfactory.

Bayesian Inference. Our first *key insight* is that the over- and under-generalization problem can be tackled by viewing the regex synthesis problem through the lens of Bayesian inference. Given a set of input strings S, we can find the most likely regex r that produced the strings by maximizing its *posterior probability*, determined via the Bayes' rule:

$$P(r \mid S) \propto P(r) \cdot P(S \mid r).$$

Here $P(r)$ is the *prior probability* of the regex r, *i.e.* the probability of selecting r from the space of all expressions in the DSL; a natural prior assigns higher probabilities to programs that are shorter and use more common constructs. The second term, $P(S \mid r)$, is the *likelihood* of the inputs given the regex r, *i.e.* the probability of generating the inputs by sampling strings accepted by r. Likelihood is higher for more specific regexes, since there are fewer alternative strings that could have been generated instead of S. Therefore, the Bayesian formulation forces us to trade-off the simplicity of the regex against its specificity.

In our running example, the regex with the highest posterior probability is:

$$\texttt{[a-z]+@stanford\textbackslash.edu} \tag{3}$$

which is the intended solution. To give some intuition for why the regex (3) is more optimal than (1) and (2), we will now discuss how we compute the prior and likelihood terms.

Computing the Prior. A common way to describe a prior over expressions [23,6] is to use a *probabilistic context-free grammar* (PCFG), which essentially assigns a fixed probability to each construct that makes up a regex. So, for example,

$$P(\texttt{[a-z]+@stanford\textbackslash.edu}) = P(\texttt{+}) \cdot P(\texttt{[a-z]}) \cdot P_c^{|\texttt{@stanford.edu}|}$$

where P_c is the probability of an individual character. With a PCFG-based prior, we expect $P(\texttt{[a-z]+}) > P(\texttt{(lpage)?(twoods)?})$, since the latter expression is both longer and uses the less common optional constructs. Hence, it is not surprising that the prior component gives the intended solution (3) an edge over the candidate (2).

Computing the Likelihood. Likelihood of a string s under a regex r can be computed using the formalism of *stochastic regular expressions* (SREs) [30,25]. Let us illustrate this calculation using the string $s = \texttt{stanford.edu}$ and two different regexes: $r_1 = \texttt{stanford\textbackslash.edu}$, the suffix of (3), and $r_2 = \texttt{[a-z]+\textbackslash.[a-z]+}$, the suffix of (1). Clearly, r_1 generates s with probability 1, as it is the only string accepted by the regex. Instead, for r_2 the likelihood can be computed as

$$\begin{aligned} P(s \mid r_2) &= P(\texttt{stanford} \mid \texttt{[a-z]+}) \cdot P(\texttt{.} \mid \texttt{\textbackslash.}) \cdot P(\texttt{edu} \mid \texttt{[a-z]+}) \\ &= \left(\frac{p}{26}\right)^8 (1-p) \cdot 1 \cdot \left(\frac{p}{26}\right)^3 (1-p) \quad \text{where} \quad 0 < p < 1 \end{aligned}$$

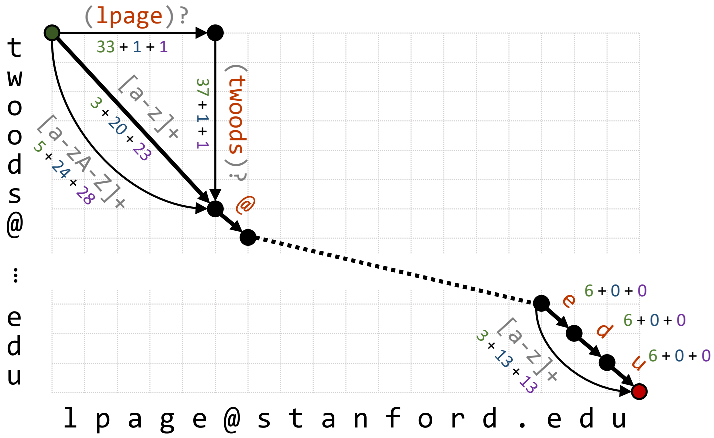


Fig. 1: A sub-graph of the version space for two input strings `lpage@stanford.edu` and `twoods@stanford.edu`. Nodes correspond to source positions in the input strings (*i.e.* how much of each string has already been matched). Source and goal nodes are highlighted in green and red, respectively. Each edge is labeled with a regex factor and a numeric cost, composed of the simplicity term (in green), and specificity terms for the two inputs (in blue and purple).

Intuitively, both occurrences of the factor `[a-z]+` have to repeatedly make a choice between generating one of the 26 lowercase letters with probability $p/26$, or ceasing to generate more letters with probability $1-p$. Again, it is not surprising that the intended solution (3) wins out over the candidate (1) thanks to its much higher likelihood.

2.2 Representing Solutions as a Version Space

Now that we have defined how to compute the posterior probability of a given regex, how do we search the space of all regexes to find the one with the highest posterior? The good news is that within the restricted class of format regexes the space of solutions we need to consider is finite.[5] The bad news, however, is that this space is astronomically large, so we need to find a way to search it without explicitly enumerating all solutions.

To this end, we follow prior work on synthesis using *version space algebras* (VSA) [21,17,27,32], whose main idea is to represent the space of all solutions compactly as a DAG, called a *version space*. Fig. 1 shows a small portion of the version space for our running example. In our case, a node in the DAG corresponds to a vector of source positions in the input strings, which indicates how much of each input has been matched so far; in the rest of this section,

[5] The full solution space is infinite if we allow optionals that are never taken in any of the inputs, but such optionals are provably suboptimal (see Sec. 4).

we will denote nodes by $\langle s_1, s_2 \rangle$, where s_1 and s_2 are the prefixes of the input strings that have been matched. An edge in the DAG corresponds to a regex component (a factor) that can take us from one node to another. For example, an edge labeled `[a-z]+` connects the *source node* $\langle \epsilon, \epsilon \rangle$ to the node $\langle$`lpage`, `twoods`$\rangle$. A solution is any path from the source node to the *goal node*, where both inputs have been matched in their entirety, $\langle$`lpage@stanford.edu`, `twoods@stanford.edu`$\rangle$.

The main benefit of the version space representation is its *exponential compactness*: note how the DAG in Fig. 1 represents not only the three candidate solutions we discussed in Sec. 2.1, but also *all combinations* of their factors, such as *e.g.* `[a-z]+@stanford\.[a-z]+` or `[a-z]+@[a-z]+\.edu`.

The compactness of the version space, however, would not do us much good if we had to explicitly enumerate all complete paths in the DAG in order to find the most likely solution. Fortunately, we can do better: our *second key insight* is that we can map the *posterior probability* of a solution to a *cost of a path* in the DAG, which enables using standard graph search algorithms to find the cheapest path without enumerating all of them.

Mapping Probabilities to Costs. To this end, we associate each edge in the DAG with a non-negative *cost*, which correspond to the negative log of its posterior probability. For example, let e be the edge in Fig. 1 that connects the source node to $\langle$`lpage`, `twoods`$\rangle$ and is labeled `[a-z]+`. The (unnormalized) posterior of this edge can be computed as:

$$P(\texttt{[a-z]+}) \cdot P(\texttt{lpage} \mid \texttt{[a-z]+}) \cdot P(\texttt{twoods} \mid \texttt{[a-z]+})$$

Taking the logs, we define the cost of e as a sum of a *simplicity term*—*i.e.* the negative log of the prior—and two *specificity terms* for the two inputs—*i.e.* the negative logs of the likelihoods:

$$\mathsf{cost}(e) = -\log P(\texttt{[a-z]+}) - \log P(\texttt{lpage} \mid \texttt{[a-z]+}) - \log P(\texttt{twoods} \mid \texttt{[a-z]+})$$

Fig. 1 shows the costs of the edges in the DAG for our running example, split into the three terms.

Importantly, both the prior and the likelihood defined in Sec. 2.1 are *compositional wrt.* the factor structure of the regex: that is, the probability of a regex is the product of the probabilities of its factors.[6] Hence, if we take the cost of a path to be the sum of the costs of its edges, then *minimizing* the path cost corresponds to *maximizing* the posterior of its entire regex. In Fig. 1, the cheapest path, which corresponds to `[a-z]+@stanford\.edu`, is highlighted in bold.

2.3 GVSE

Even though version spaces are compact relative to an explicit list of all solutions, they can still grow quite large, especially as the number and length of the input

[6] This is not exactly true for the likelihood of ambiguous regexes, which leads to underapproximating their probability, as we discuss in Sec. 3.2; this is not a problem in practice, however, since most format regexes are unambiguous.

strings increases. Even for our running example, which only has two inputs, the full version space contains $18 \times 19 = 342$ nodes and around 50 thousand edges. As a result, VSA-based search easily runs out of memory for even moderately-sized inputs. Our *third key insight* is that instead of building the full version space and *then* searching it, as in traditional VSA synthesis [17,27,32], we can construct the version space on the fly, *during* search. As long as the search only explores a fraction of the DAG, this approach can alleviate the memory bottleneck.

The baseline graph search algorithm is known as *uniform-cost search* or *Dijkstra's algorithm*. At a high level, it explores a graph starting from the source node, and maintains a frontier of nodes that have been visited but not yet expanded; at each step, it expands the frontier node v with the lowest *cost-so-far* $\mathsf{g}(v)$, *i.e.* the cost of the cheapest path from the source to v; once a full path to the goal is found, it is guaranteed to be optimal. Unfortunately, uniform-cost search is not very helpful when it comes to saving memory, since it is known to explore a large portion of the DAG before reaching the goal. Instead, we turn to two popular guided search algorithms—*A* search* and *beam search*—which are known to significantly reduce the proportion of explored paths, and we adapt them to searching a version space of regular expressions.

GVSE-A*. A* search [19] improves upon uniform-cost search by considering not only v's cost-so-far, $\mathsf{g}(v)$, but also its estimated *cost-to-go*, $\mathsf{h}(v)$—*i.e.* the cost of the cheapest path from v to the goal. A* requires a *heuristic* function that estimates the cost-to-go, and as long as the heuristic is *admissible*—*i.e.* it never overestimates the cost—the algorithm is guaranteed to find the optimal solution. The main challenge, therefore, is to come up with a heuristic that is admissible, computationally efficient, and sufficiently accurate to guide the search. A* search has been used in program synthesis before [23], but only with purely syntactic cost functions.

The design of our heuristic is based on a simple observation: *the best regex that matches n strings is necessarily more expensive than the best regex that matches any $m < n$ of those strings.* Because the complexity of our problem is exponential in the number of input strings, it makes sense to pick a small m (say, $m = 1$ or 2), build complete version spaces for sub-problems of size m, and use them to compute the heuristic for the full problem.

For example, going back to Fig. 1, let us estimate the cost-to-go for the node $v_1 = \langle \texttt{lpage}, \epsilon \rangle$ using the two sub-problems with $m = 1$. The suffixes of the two inputs that are still left to match are, respectively, `@stanford.edu` and `twoods@stanford.edu`, and the best regex for either is the corresponding literal. We conclude that the best solution for v_1 is at least as expensive as the *costlier* of these two, *i.e.* the regex `twoods@stanford\.edu`; in fact, the optimal solution for this node is `(twoods)?@stanford\.edu`, and our estimate was pretty close.

Let us now compare the nodes $v_1 = \langle \texttt{lpage}, \epsilon \rangle$ and $v_2 = \langle \texttt{lpage}, \texttt{twoods} \rangle$. If we only consider the cost-so-far, then v_1 looks more promising than v_2 (because the best path to v_1 is very specific):

$$\begin{aligned} \mathsf{g}(v_1) &= \mathsf{cost}(\texttt{(lpage)?} \mid \langle \texttt{lpage}, \epsilon \rangle) = 35 \\ \mathsf{g}(v_2) &= \mathsf{cost}(\texttt{[a-z]+} \mid \langle \texttt{lpage}, \texttt{twoods} \rangle) = 45. \end{aligned}$$

As a result, uniform-cost search would expand v_1 first, which would be a mistake: in retrospect, we know that v_2 lies on the optimal path. If we take into account the heuristic, however, then v_2 becomes more promising:

$$\begin{aligned} \mathsf{g}(v_1) + \mathsf{h}(v_1) &= 35 + 87 = 122 \\ \mathsf{g}(v_2) + \mathsf{h}(v_2) &= 45 + 61 = 106. \end{aligned}$$

As a result, A* search prefers v_2 and finds the optimal solution faster.

GVSE-Beam. Although the A* heuristic helps curb the number of explored paths, it is not always accurate, especially for the nodes that are far from the goal. In our running example, consider the node $v_3 = \langle \epsilon, \texttt{tw} \rangle$, whose best regex is `(tw)?`; its full cost estimate is $\mathsf{g}(v_3) + \mathsf{h}(v_3) = 104$, better than that of the "optimal" node v_2 (106), so v_3 is expanded first by A*. At the same time, it is intuitively clear that the only reason v_3 appears so promising is that is has made so little progress through the input strings, that the heuristic has trouble accurately estimating its cost-to-go.

A popular alternative to A* is *beam search* [31], which trades off the optimality guarantee for improved efficiency. Beam search gives up on keeping track of all nodes in the frontier; instead, it divides nodes into "buckets", only keeping the best k nodes in each bucket and discarding the rest (where k is known as *beam size*). Mostly commonly nodes are bucketed by the number of search steps it took to reach them, but not necessarily; intuitively, buckets should group together nodes that have made "the same amount of progress" towards the goal, so that their costs can be compared meaningfully.

In our setting, we propose to *bucket the nodes by the total number of characters they have parsed* from all input strings. For example, node v_3 is in the bucket 2 (because it has only parsed `tw`) and node v_2 is in the bucket 11 (because it has parsed `lpage` and `twoods`). When deciding whether to keep either of these nodes in the frontier or to discard it, v_3 will be compared with other bucket-2 nodes (whose costs are all low), and v_2 will be compared with other bucket-11 nodes (whose costs are all higher). As a result, v_2 will make it to the top k, while v_3 might be discarded (given low enough k). Our evaluation shows that this bucket structure enables our beam search to obtain high precision even with very small beam sizes.

3 Language and Cost Function

3.1 Format regexes

As only some regex features are amenable to learning from positive examples, we consider a restricted DSL of regular expressions, which we call *format regexes*,

$$\begin{array}{rlrl} \textit{regex} & ::= \epsilon \mid \textit{factor regex} & \textit{atom} & ::= \textit{class} \mid \textit{class}\,\texttt{+} \mid \texttt{literal(}\textit{string}\texttt{)} \\ \textit{factor} & ::= \textit{atom} \mid \texttt{(}\textit{atom}\texttt{)?} & \textit{class} & ::= \texttt{[0-9]} \mid \texttt{[a-z]} \mid \texttt{[A-Z]} \mid \texttt{[a-zA-Z]} \mid \texttt{[a-zA-Z0-9]} \end{array}$$

Fig. 2: The grammar of format regexes

representing a regex as a sequence of *factors*, individual simple components which include literal strings, character classes, etc. This is a common restriction for algorithms synthesizing regular expressions and string transformations from positive examples [17,32,12], and in fact our grammar extends that of existing work by including optionals. The grammar for the DSL is shown in Fig. 2. The specific choice of character classes is inessential to our algorithm.

3.2 Bayesian cost function

To determine the best regular expression from the grammar, we use a Bayesian cost function, computing

$$P(r \mid S) \propto P(r) \cdot P(S \mid r)$$

according to Bayes' rule. Our goal is to find the regex r which maximizes $P(r \mid S)$; as the constant of proportionality depends only on the input strings S and not on the regex r, this can be done by maximizing the right-hand side.

We reframe this in terms of a *cost function*: taking negative logs, we define

$$\mathsf{cost}(r \mid S) := -\log\left[P(r) \cdot P(S \mid r)\right].$$

Taking $\mathsf{simplicity}(r) := -\log P(r)$ and $\mathsf{specificity}(r \mid s) := -\log P(s \mid r)$, and assuming input strings are i.i.d., this becomes

$$\mathsf{cost}(r \mid S) = \mathsf{simplicity}(r) + \sum_{s \in S} \mathsf{specificity}(r \mid s).$$

In the next sections, we formalize how the simplicity and specificity terms are computed.

Simplicity. To compute the probability of a regular expression, we need a probabilistic model for the generation of regexes. For this, we use a probabilistic context-free grammar (PCFG), a framework in which each production rule of a grammar has an associated probability. To find the probability of a program, one takes a top-down derivation of the program from the grammar, and multiplies together the probabilities associated with each production rule used.

The production probabilities can encode prior knowledge about the domain: for instance, `[0-9]` is simpler than `[a-zA-Z0-9]` which is simpler than a constant string which is simpler than an optional.

By using a PCFG, the simplicity term can be computed compositionally from the factors used. Let p_{end} be the probability of the transition $regex \to \epsilon$, and let r be a regular expression composed of the factors $r_1 \dots r_n$. Then the probability assigned to r by the PCFG is

$$P(r) = p_{\mathsf{end}} \cdot \prod_{i=1}^{n} (1 - p_{\mathsf{end}}) P(r_i),$$

Overloading $\mathsf{simplicity}(f)$ to denote $-\log p_{\mathsf{end}} - \log P(f)$ for a factor f, we have:

$$\mathsf{simplicity}(r) = \sum_{i=1}^{n} \mathsf{simplicity}(r_i) + \text{a constant.}$$

Specificity. The specificity cost of a regular expression is computed in terms of the probability that the given inputs would be chosen from this regex. For this, we need a probabilistic model of generating strings from regexes. To this end, we employ the *stochastic regular expressions* (SREs) [25,30]:

$$S \quad ::= \quad \texttt{literal}(\mathit{string}) \quad | \quad S_1 S_2 \quad | \quad S_1 \mid_p S_2 \quad | \quad S^{*_p}$$

Here, p may be any real number in the interval $(0, 1)$. While a regular expression merely describes a set of strings, a stochastic regular expression additionally describes a probability distribution on those strings, by attaching probability information to disjunctive constructs and giving it a semantics as a *probabilistic generator* of text.

This semantics is best described using a probabilistic sample operation:

$$\begin{aligned}
\mathsf{sample}(\texttt{literal}(s)) &= s \\
\mathsf{sample}(s_1 s_2) &= \mathsf{sample}(s_1)\, \mathsf{sample}(s_2) \\
\mathsf{sample}(s_1 \mid_p s_2) &= \begin{cases} \mathsf{sample}(s_1) & \text{with probability } p \\ \mathsf{sample}(s_2) & \text{with probability } 1 - p \end{cases} \\
\mathsf{sample}(s^{*_p}) &= \begin{cases} \mathsf{sample}(s)\, \mathsf{sample}(s^{*_p}) & \text{with probability } p \\ \epsilon & \text{with probability } 1 - p \end{cases}
\end{aligned}$$

To apply stochastic regular expressions to our synthesis task, we translate format regexes into SREs. This requires selecting values for the probability annotations p, which we do in an unbiased way—for instance, the factor `literal(a)?` is translated to the SRE `literal(a)` $|_{0.5}\ \epsilon$.

Given the semantics of SREs, it is not hard to compute the probability of generating a given input. For instance, the regex `[0-9][0-9]` generates the string `42` with probability

$$P(\mathsf{42} \mid \texttt{[0-9][0-9]}) = P(\mathsf{4} \mid \texttt{[0-9]}) \cdot P(\mathsf{2} \mid \texttt{[0-9]}).$$

Importantly, just like the PCFG in the previous section, the full probability of a regex is a *product* of the probabilities of its factors, which is an essential property to be able to compute the specificity term compositionally.

Ambiguous parses. Unfortunately, this is not always the case. Consider the regex `[0-9]?[0-9]?`, encoding "up to two digits". The string `8` may be generated as `8` followed by ϵ, or as ϵ followed by `8`, so its probability is the *sum* of these two cases (which does not decompose after taking the log):

$$\begin{aligned} P(\texttt{8} \mid \texttt{[0-9]?[0-9]?}) &= P(\texttt{8} \mid \texttt{[0-9]?}) \cdot P(\epsilon \mid \texttt{[0-9]?}) \\ &\quad + P(\epsilon \mid \texttt{[0-9]?}) \cdot P(\texttt{8} \mid \texttt{[0-9]?}) \end{aligned}$$

This is an issue when a regex can parse the input strings in more than one way.

We bypass this issue by reframing the problem to ask for the most likely combination of a regex together with a *parse*—*i.e.* an assignment of input segments to factors—rather than merely the most likely regex. This is, of course, equivalent in the common case when the optimal regex is unambiguous, which is the case for all but four of the regexes used in our evaluation. In the sequel we will freely use "optimal regex" to refer to the most likely regex-parse combination.

Computing the specificity term. Having chosen a parse, we may now compute the likelihood $P(s \mid r)$. Let r be a regular expression composed of the factors $r_1 \ldots r_n$, and let $s = s_1 \ldots s_n$ be a string such that s_i is parsed by r_i. Then

$$P(s \mid r) = \prod_{i=1}^{n} P(s_i \mid r_i)$$

and, taking negative logs, we have

$$\mathsf{specificity}(r \mid s) = \sum_{i=1}^{n} \mathsf{specificity}(r_i \mid s_i).$$

4 Version space algebras

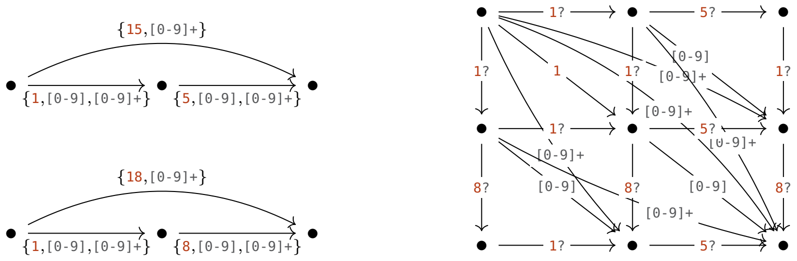
We encode the set of candidate solutions using a *version space algebra* (VSA). VSAs [21,17] are a synthesis technique which represents the search space using a data structure called a *version space*. VSA techniques are based on three core operations:

1. *Construction:* given a single input s, build a version space $\mathsf{VS}(s)$ which encodes the set of all candidate programs which work for that one input.
2. *Intersection:* given two version spaces V_1 and V_2, compute a combined version space $V_1 \sqcap V_2$ which encodes the intersection of programs in V_1 and V_2.
3. *Search:* given a version space, find the best program from it (according to some cost function).

VSA-based program synthesis works by creating a version space for each input example, intersecting them, and then extracting the best program from the resulting version space.

4.1 A version space algebra for format regexes

In Regex+, a version space is a directed acyclic graph (V, E), where each node $v \in V$ is labelled by a source position in the input strings, and each edge $e \in E$ is labelled by a factor. Possible regexes are encoded as paths though the DAG. We next discuss the three VSA operations in detail. For this discussion, we use a small running example consisting of the two input strings `15` and `18`.



(a)

(b)

Fig. 3: (a) The version spaces $\mathsf{VS}(\texttt{15})$ and $\mathsf{VS}(\texttt{18})$. (b) The intersection of the version spaces from (a). For clarity, longer optionals are omitted and only one factor is shown between any two nodes.

Initial Version Space Construction. To construct a version space from an input string, we create vertices for each position in the string, and we connect each pair of vertices by edges labelled by the factors which match the corresponding parts of the string. For our running example, the initial version spaces are shown in Fig. 3a. More formally, given an input string $s = c_0 c_1 \dots c_{n-1}$, we construct the version space $\mathsf{VS}(s) = (V, E)$ where:

$$\begin{aligned} V &\coloneqq \{v_i \mid 0 \le i \le n\} \\ E &\coloneqq \{(v_i \to v_j; r) \mid 0 \le i < j \le n, r \text{ matches } c_i \dots c_{j-1}\} \end{aligned}$$

and $(v_i \to v_j; r)$ denotes an edge from v_i to v_j labelled by the factor r.

Version Space Intersection. Next, the version spaces representing each input are intersected. The intersection version space for our running example is shown in Fig. 3b. From two version spaces (V_1, E_1) and (V_2, E_2), we construct 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, w_1) to (v_2, w_2) will be labelled by factors which appear both as edges $v_1 \to v_2$ and as edges $w_1 \to w_2$. For example, both version spaces in Fig. 3a have an edge from the start to the end labelled by the factor `[0-9]+`, so this factor is included in their intersection Fig. 3b across the diagonal.

However, the intersection mechanism as described so far does not account for optional factors. So far, each edge in the the intersection version space 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 factors by taking each edge $(v_1 \to v_2;\ r)$ in one of the input version spaces, and adding $\texttt{(}r\texttt{)?}$ as an edge between vertices of the form (v_1, w) and (v_2, w), for each vertex w of the other input version space. These optional edges can be seen in Fig. 3b. All the horizontal optional edges, such as those labelled by `5?`, 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.

Formally, we define

$$\begin{aligned}
V_\cap &\coloneqq V_1 \times V_2 \\
E_\cap &\coloneqq \{((v_1, w_1) \to (v_2, w_2);\ r) \mid (v_1 \to v_2;\ r) \in E_1,\ (w_1 \to w_2;\ r) \in E_2\} \\
&\quad \cup \{((v_1, w) \to (v_2, w);\ \texttt{(}r\texttt{)?}) \mid (v_1 \to v_2;\ r) \in E_1,\ w \in V_2\} \\
&\quad \cup \{((v, w_1) \to (v, w_2);\ \texttt{(}r\texttt{)?}) \mid v \in V_1,\ (w_1 \to w_2;\ r) \in E_2\}
\end{aligned}$$

Extracting the best solution. Having constructed a DAG, it remains to find the optimal candidate regex. Recall from Sec. 3.2 that a regex r composed of the factors $r_1 \ldots r_n$ where each factor r_i parses portion s_i of input string s is optimal when the cost function

$$\begin{aligned}
\mathsf{cost}(r \mid S) &= \mathsf{simplicity}(r) + \sum_{s \in S} \mathsf{specificity}(r \mid s) \\
&= \sum_{i=1}^{n} \left[\mathsf{simplicity}(r_i) + \sum_{s \in S} \mathsf{specificity}(r_i \mid s_i) \right] + \text{a constant}
\end{aligned}$$

is minimized. As an edge in the version space encodes *both* a factor and a portion of the inputs which it parses, optimizing this cost function thus becomes a shortest path problem, where the edge e corresponding to the factor r_i and parsing the set S_i of substrings of the input strings is assigned the weight

$$\mathsf{cost}(e) \coloneqq \mathsf{simplicity}(r_i) + \sum_{s \in S_i} \mathsf{specificity}(r_i \mid s).$$

So in principle, the best solution may be extracted by computing the shortest path in the DAG, using *e.g.* a dynamic program.

However, even constructing the DAG in the first place is often infeasible, as it grows quite large and requires too much memory. In Sec. 5, we present more efficient guided version space exploration (GVSE) algorithms to address this challenge.

4.2 Completeness

A key advantage of VSA techniques is that they are *complete*: out of all the infinitely-many format regexes, we guarantee that we can find the optimal one by merely searching for the optimal path in the finite-size version space. This is a consequence of the following theorem:

Theorem 1. *The optimal format regex is encoded by a path in the version space.*

Recall that, as per Sec. 3.2, the optimal format regex refers to the most likely combination of a format regex and a parse.

Proof. We show the contrapositive, that every regex and parse *omitted* from the version space is suboptimal. A regex-parse combination will only be omitted if one of its factors is omitted as an edge in the version space.

By construction, every factor which parses a nonempty segment of some input is included as an edge in the VSA, so this factor must not parse any input from any of the examples. So a strictly better regex-parse combination may be obtained by simply removing this factor.

Corollary 1 (Completeness). *The shortest path in a version space is the overall optimal format regex for those input strings.*

5 Guided Version Space Exploration

To overcome the scalability challenge of the VSA, we propose a new search algorithm which we call *guided version space exploration* (GVSE). As the version space graph is often too large to even be constructed in memory, our GVSE never fully constructs this graph, saving memory.

Typically, given input strings $S = \{s_1, s_2, \ldots, s_n\}$, one would construct the version spaces $\mathsf{VS}(s_1), \mathsf{VS}(s_2), \ldots, \mathsf{VS}(s_n)$, respectively, then intersect them one-at-a-time with a binary intersection operation:

$$\mathsf{VS}(S) = ((\mathsf{VS}(s_1) \sqcap \mathsf{VS}(s_2)) \sqcap \ldots) \sqcap \mathsf{VS}(s_n).$$

To avoid constructing any of the intermediate graphs in memory, we describe an n-way intersection operation on version spaces.

- Vertices of $\mathsf{VS}(S)$ are n-tuples $\langle i_1, \ldots, i_n \rangle$ where each i_k is an index into the string s_k.
- An edge between two different states $\langle i_1, \ldots, i_n \rangle$ and $\langle j_1, \ldots, j_n \rangle$ is labeled with a factor r iff for every $1 \leq k \leq n$, either (1) r accepts $s_k[i_k \ldots j_k)$, or (2) r is optional and $i_k = j_k$.

This graph may be computed on-the-fly rather than being fully realized in memory, enabling the use of efficient search algorithms that don't explore every node.

5.1 GVSE-A*

We first present GVSE-A*, a *complete* search algorithm which guarantees optimality of its output using an admissible heuristic.

We perform A* search starting from $\langle 0, \dots, 0\rangle$ and with a goal node of $\langle |s_1|, \dots, |s_n|\rangle$. A* [19] is a best-first search algorithm, which explores outwards from the start node, prioritizing nodes which may be a part of the best overall path from the start to the end. The cost of this path for a node v is given by $\mathsf{g}(v) + \mathsf{h}^*(v)$, where $\mathsf{g}(v)$ is the cost of the best path from the start to v, which is known, and $\mathsf{h}^*(v)$ is the cost of the best path from v to the goal—which is unknown. A* therefore instead ranks nodes by $\mathsf{g}(v) + \mathsf{h}(v)$ for an approximation $\mathsf{h}(v)$ of $\mathsf{h}^*(v)$, known as the *heuristic function.* If $\mathsf{h}(v) \leq \mathsf{h}^*(v)$ for all nodes v, $\mathsf{h}(v)$ is called *admissible* and A* is guaranteed to find the optimal path.

Our A* heuristic comes from the insight that for a small number of inputs, it is easy to fully compute all best paths in a version space. Moreover, if we simplify the problem by considering only a small subset of the inputs, the cost of the best solution to this restricted problem will be smaller than the cost of the best solution to the full problem.

Formally, for a small subset $S' \subset S$ of the input strings, we define a heuristic function $\mathsf{h}_{S'}$ as follows: Let $m = |S'|$ and without loss of generality suppose S' is $\{s_1, \dots, s_m\}$. Then the value of the heuristic $\mathsf{h}_{S'}(\langle i_1, \dots, i_n\rangle)$ is the length of the best path from $\langle i_1, \dots, i_m\rangle$ to $\langle |s_1|, \dots, |s_m|\rangle$ in $\mathsf{VS}(S')$. Choosing $|S'|$ to be sufficiently small, this version space is easily stored in memory and all best paths precomputed using *e.g.* the method described in Sec. 4.

In the implementation, we strengthen the heuristic by computing $\mathsf{h}_{S'}$ for a number of choices of S', and taking the maximum:

$$\mathsf{h}(v) := \max\{\mathsf{h}_{S'_1}(v), \mathsf{h}_{S'_2}(v), \dots\}$$

The larger the sets S', and larger the number of subsets used, the more precise the heuristic function will be, but more expensive to compute. We found that $|S'| = 2$ strikes a good balance between not using too much memory while still successfully pruning the space and speeding up the search, and so we use a collection of pairs.

Theorem 2 (Admissibility). *The heuristic function* $\mathsf{h}_{S'}(v)$ *is admissible for any* $S' \subset S$.

Proof. For a string s, let $s[i \dots]$ denote the suffix of s starting at index i.

Let $\langle i_1 \dots i_n\rangle = v$, and let $v' = \langle i_1 \dots i_m\rangle$ be the corresponding node in $\mathsf{VS}(S')$. A path from v to $\langle |s_1|, \dots, |s_n|\rangle$ represents a format regex which matches the suffixes $s_1[i_1 \dots]$, …, and $s_n[i_n \dots]$ of the inputs in S, and correspondingly a path from v' to $\langle |s_1|, \dots, |s_m|\rangle$ in $\mathsf{VS}(S')$ represents a format regex which matches the suffixes $s_1[i_1 \dots]$, …, $s_m[i_m \dots]$.

Let r be the format regex corresponding to the optimal path from v to $\langle |s_1|, \dots, |s_n|\rangle$, so that

$$\mathsf{cost}(r \mid s_1[i_1 \dots], \dots, s_n[i_n \dots]) = \mathsf{h}^*(v).$$

On the other hand, $\mathsf{h}(v)$ is the cost of the optimal regex for the subset of suffixes $s_1[i_1 \ldots]$, …, $s_m[i_m \ldots]$. Thus,

$$\begin{aligned} \mathsf{h}(v) &\leq \mathsf{cost}(r \mid s_1[i_1 \ldots], \ldots, s_m[i_m \ldots]) \\ &\leq \mathsf{cost}(r \mid s_1[i_1 \ldots], \ldots, s_n[i_n \ldots]) = \mathsf{h}^*(v). \end{aligned}$$

Corollary 2. *The heuristic function* $\mathsf{h}(v) = \max\{\mathsf{h}_{S'_1}(v), \mathsf{h}_{S'_2}(v), \ldots\}$ *is admissible, for any collection of subsets* $S'_i \subset S$.

Corollary 3 (Completeness). *GVSE-A* finds the optimal regular expression.*

5.2 GVSE-Beam

Our second GVSE search algorithm is based on beam search and, while no longer complete, significantly improves efficiency and performs extremely well in practice.

Beam search [31] is a greedy search algorithm commonly used in machine learning for decoding the output of neural networks. In that context, beam search is applied to a search *tree*; it works by greedily picking a *beam*, *i.e.* a collection of k candidate nodes *from each depth of the tree*, and discarding any nodes whose costs-so-far are not among the top k at that depth. This is the source of incompleteness in beam search: the optimal path may start with a high-cost prefix, and thus mistakenly be discarded early on. Nevertheless, beam search is highly effective in practice. The intuition is that two partial paths at the same depth in the tree tend to have solved similar amounts of the problem, so their scores can be meaningfully compared.

Adapting beam search to version space exploration is not entirely straightforward: a version space is a highly-connected DAG rather than a tree, so the notion of "depth" is not very useful to decide which nodes are meaningfully comparable (for example, you can argue that all nodes in the DAG in Fig. 3b are at the same "depth" of one). Hence, we need to design a different way to "bucket" the nodes together.

To this end, we group nodes by *total number of characters parsed*, defining

$$\mathsf{bucket}(\langle i_1, \ldots, i_N \rangle) = \sum_{n=1}^{N} i_n.$$

Intuitively, we expect that for two nodes v_1 and v_2 with $\mathsf{bucket}(v_1) = \mathsf{bucket}(v_2)$, as a path from v_1 to the end has the same number of characters as a path from v_2 to the end, $\mathsf{h}^*(v_1)$ is likely to be similar to $\mathsf{h}^*(v_2)$, and so their costs so far may be meaningfully compared.

One useful property of beam search is that the beam size k is a tunable parameter than governs how approximate the search is. With a beam size of $k = 1$, it becomes a completely greedy search, and as the beam size grows larger, beam search will explore all paths, regaining completeness in the limit. However, this of course has the inverse effect on performance, with larger beam sizes being proportionally more expensive to compute. Therefore, the beam size is a useful parameter for trading off efficiency and optimality.

6 Evaluation

We have implemented our Bayesian cost function and the GVSE search procedure in a prototype synthesizer called Regex+, which takes as input a set of strings and outputs the most likely regex that matches those strings. Our empirical evaluation aims to answer the following research questions:

RQ1: How accurate is the Bayesian cost function at recovering regexes from informative positive examples provided by humans?

RQ2: Do the proposed search procedures, both GVSE-A* and GVSE-Beam, offer benefits over the baseline VSA search?

6.1 Experiment Setup

Existing benchmarks for evaluating regex synthesizers have not focused on the setting where only positive examples are provided. Further, the positive examples that they do include were not designed in order to convey a regex in the absence of other supervision (e.g. negative examples, or a textual prompt) [10]. Thus, we collected and annotated a novel benchmark dataset designed for our setting. Specifically, we took regexes from prior benchmarks and then augmented them with human-designed informative positive examples, as described below.

Regex Selection. To obtain regexes for our experiments, we selected all regexes from Regel's StackOverflow dataset [10] that conform to Regex+'s regex grammar; this left us with 21 regexes. We decided to focus on this dataset because it is realistic (the regexes are extracted from StackOverflow posts); this is in contrast to benchmarks from [24,22], which are synthetic and/or use a restricted alphabet, and are not representative of the format of learning task we are interested in.

Human Data Collection. The StackOverflow benchmarks are intended for multi-modal synthesis, using a combination of natural-language descriptions, positive, and negative examples, and thus often provide a minimal number of positive examples, in some cases only one. We first conducted a small pilot study that confirmed that neither humans nor Regex+ were able to learn regexes from those minimal example sets (both had accuracy of less than 10%). Instead, our experiments require *informative* examples sets, the kind that a human would give if they were asked to communicate a regex with just positive examples. To this end, we conducted two human experiments: (1) a *speaker experiment*, where the humans were asked to generate positive examples for given regexes, and (2) a *listener experiment*, where the humans were asked to infer regexes from positive examples. We used the data from the first experiment to construct our benchmark suite, and the data from the second experiment as the human baseline for regex learning.

Both experiments were conducted as an online surveys using JSPhych. We recruited 11 and 15 participants for each experiment, respectively, with knowledge of regular expressions; the participants in the two studies did not overlap. They were CS students and faculty from our department and departments of our collaborators, and were not compensated.

Speaker Experiment. In the speaker experiment, each participant was given the regex in the Regex+ DSL and its natural-language description, and asked to provide as many or as few examples as they thought would be necessary to convey the regex to another person. Participants were aware of the DSL restrictions. Further, they were given four training examples, each requiring them to infer a regex from curated examples, to illustrate what informative and uninformative examples might look like.

As a result of this experiment, we obtained 231 benchmarks (*i.e.* 11 example sets for each of the 21 regexs) where each benchmark consists of a regex and a set of positive examples. The number of examples per benchmark varies from 1 to 7 (median 3); the length of the examples varies from 1 to 67 (median 6).

Listener Experiment. Each listener was asked to infer the regex for a set of examples corresponding to each of the 21 regexes in our evaluation dataset. In order to minimize the effect of speaker quality (*i.e.* some sets of examples for a given regex are more informative than others, depending on the speaker), we randomized which of the 11 example sets would be seen by each listener. We also randomized the order of the regexes to prevent learning effects. In two cases, the participants did not finish guessing all of the regexes, so we recruited two additional participants to complete the task. This resulted in 15 total participants. Participants entered their guesses in a text box; they were made aware of the Regex+ DSL, but were not required to adhere to it; instead, they could enter any regex using Javascript syntax. Listeners were given feedback if the regex they entered was not a valid regex, but were not told if it did not match some of the examples; our rationale was to prevent typos, but otherwise not to interfere with the human learning process.

Computational Experiments. All experiments were run on a server on a single core, with a 12GB RAM limit and a 60 minute timeout.

6.2 RQ1: Accuracy of the Cost Function

We first compare the exact match accuracy—*i.e.* the fraction of benchmarks for which the ground truth regex is inferred—for Regex+ with its Bayesian cost function against two ablations that only use the simplicity and specificity terms, respectively. Note that the simplicity-based cost corresponds to a more traditional syntactic cost function [23,6]. Unfortunately, we cannot compare Regex+ to any existing synthesizers because they do not support learning from only positive examples. We do, however, compare with the accuracy of human listeners, which we take as a baseline for how many benchmarks we can reasonably expect to solve, given that the input examples are imperfect. The accuracy results are shown in Fig. 4, and sample solutions are shown in Tab. 1.

Comparison with Ablations. Overall, Regex+ significantly outperforms both ablations in terms of exact match accuracy: in total over all regexes, Regex+ solves 55% of benchmarks exactly correct (see the leftmost column of Fig. 4), the simplicity-only cost solves 20%, and the specificity-only cost does not solve a single benchmark, and is therefore omitted from the figure. Line 1 in Tab. 1

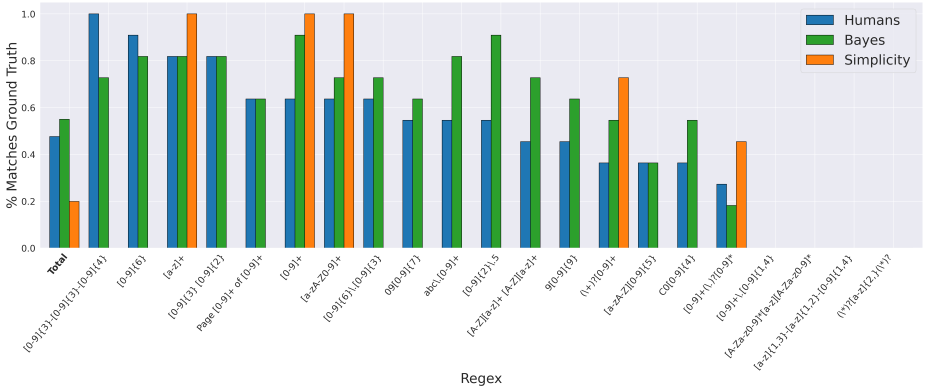


Fig. 4: Comparison of exact match accuracy against ground truth regex across human listeners, Regex+, and an ablated cost function that only includes simplicity. Results for the ablation that only includes specificity are not depicted because this system was unable to correctly infer any regexes. Accuracies are shown for each individual regex in our benchmark dataset.

Table 1: A sample of benchmarks and corresponding results inferred by humans, the Bayesian cost function, and the simplicity-only ablation.

#	Ground Truth	Examples	Humans	Bayes	Simplicity
1	C0[0-9]{4}	C09999 C01234 C05656	C0[0-9]{4}	C0[0-9]{4}	C[0-9]+
2	[a-zA-Z0-9]+	aA3aA3a a aaa	a(A3)?(a)?(A3)?(a)?	a([a-zA-Z0-9]+)?	[a-zA-Z0-9]+
3	[0-9]{3}-[0-9]{3}-[0-9]{4}	619-953-8114	[0-9]{3}-[0-9]{3}-[0-9]{4}	[0-9]+-[0-9]+-[0-9]+	[0-9]+-[0-9]+-[0-9]+
4	Page [0-9]+ of [0-9]+	Page 3 of 570 Page 1 of 60	Page [0-9]+ of [0-9]+	Page [0-9] of [0-9]+	[a-zA-Z]+ [0-9] [a-z]+ [0-9]+
5	[a-z]{1,3}-[a-z]{1,2}-[0-9]{1,4}	jwa-bu-9247 k-p-2 la-n-738	[a-z]+-[a-z]+-[0-9]+	[a-z]+-[a-z]{1,2}-[0-9]+	[a-z]+-[a-z]+-[0-9]+

shows an example where Regex+ picks up on the pattern present in all three inputs (that they all start with `C0` followed by exactly four digits), while the simplicity cost over-generalizes to an arbitrary sequence of digits.

When splitting the results by regex, we can see that simplicity does outperform Regex+ on 5/21 regexes. Closer inspection shows that this happens when the ground truth regex is very general, while the input examples feature spurious patterns. An example is the benchmark in line 2 of Tab. 1: the ground truth regex allows any sequence of alphanumeric characters, while all three human-provided examples start with a lower-case `a`, which confuses Regex+, but has no influence on the simplicity cost.

Comparison with Humans. Somewhat surprisingly, on average Regex+ performs slightly better than human listeners (55% vs. 48% accuracy). As seen in Fig. 4, humans outperform Regex+ on only two regexes; sample benchmarks

Table 2: Comparison of search algorithms in terms of: exact **match** accuracy, % of benchmarks **finished** within the timeout, avg **RAM** in megabytes per benchmark, average **time** in second per benchmark. Both RAM and time are evaluated only on benchmarks all algorithms could finish: 226 benchmarks total.

Algorithm	match	finished	RAM	Time
VSA	54.54 % (126)	97.84 % (226)	172.57 MB	4.76 seconds
A*	54.97 % (127)	99.13 % (229)	22.23 MB	5.8 seconds
Beam (size 5)	55.8 % (129)	99.57 % (230)	19.54 MB	0.17 seconds

for those regexes are shown in lines 3 and 4 of Tab. 1. In both of those cases, humans have an edge because they can infer the meaning of the regex, beyond just the syntactic pattern. The regex in line 3 is a common format for phone numbers; this familiarity allows humans to infer it from just a single example. The benchmark is line 4 is even more interesting: although both input strings only show a single digit for the page number, humans use their knowledge of the world to infer that page numbers can contain multiple digits.

The four regexes that Regex+ was not able to guess from *any* of the examples, could not be solved by humans either (see the last four columns in Fig. 4). We hypothesize that these four regexes are too complex to be learned from just a few positive examples. An example is shown in line 5 of Tab. 1: the underlying regex is quite complex, requiring the lengths of the three parts of a string to fall within specific ranges; to illustrate such a requirement properly, the speaker would need to generate many more examples, which our human speakers were understandably reluctant to do. If we omit these four regexes from our accuracy calculation, Regex+ achieves 68% and humans achieve 59%.

We also conducted a more in-depth analysis of the agreement between Regex+, ablations, and human listeners on those benchmarks, which at least one of them solved incorrectly. These results can be found in Appendix A.

6.3 RQ2: Efficiency of GVSE

To evaluate the efficiency benefits of GVSE, we implement a baseline VSA algorithm that uses the same objective function but always constructs the full version space and then performs search using Dijkstra's algorithm. We then compare our complete algorithm, GVSE-A*, to this baseline in terms of memory consumption and time spent per benchmark. Finally, we also compare GVSE-A* to its approximate version, GVSE-Beam. The results are shown in Tab. 2 and Fig. 5.

GVSE-A*. As seen in Tab. 2, the VSA baseline was able to finish 226 benchmarks out of 231—3 fewer than GVSE-A*—and ran out of memory on the rest. Further, VSA's average RAM consumption per benchmark is 7x more than that of GVSE-A*. While, on average, GVSE-A* was a little slower than the VSA baseline, when looking at the second plot on Fig. 5 we can see that the average is heavily skewed by the outliers and for the majority of the benchmarks GVSE-A* is faster. Additionally, VSA consumes more RAM for almost all of the

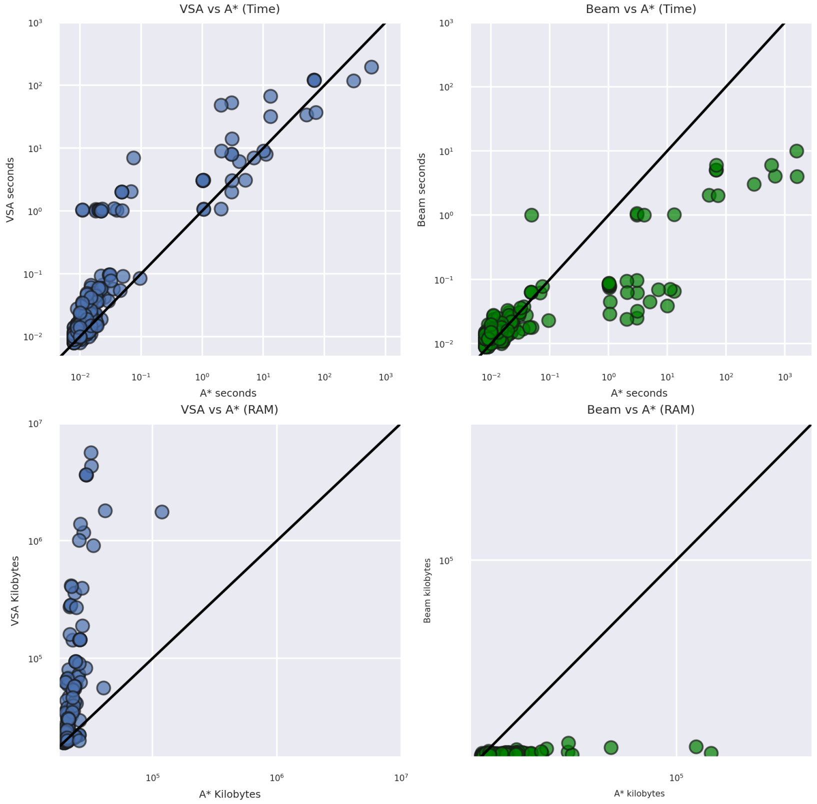


Fig. 5: Scatter plots of time and memory used by VSA baseline and GVSE-Beam v.s GVSE-A*. Each point represents a benchmarks. Only benchmarks that all algorithms could finish are considered (226 total)

benchmarks. This indicates that the GVSE-A* heuristic we are using is helpful in guiding the search to be more efficient, and not constructing the entire VSA substantially reduces memory consumption.

GVSE-Beam. GVSE-Beam is not only the most efficient, but also, surprisingly, the most accurate of the three search algorithms we consider (Tab. 2). Its average time spent per benchmark is almost 30x less than GVSE-A*, while at the same time using less RAM for all the benchmarks as seen in Fig. 5. Further, it is able to finish more benchmarks within the set resources constraints and despite being approximate, gets two more benchmarks correct than GVSE-A* (which we discuss in more detail below).

GVSE-Beam: Parameters. Finally we evaluated how beam size affects the efficiency and accuracy of GVSE-Beam. Fig. 6 (bottom) shows the resources consumption, which as expected, increases linearly with beam size. While at beam size 50 GVSE-Beam remains the best with respect to average time, it

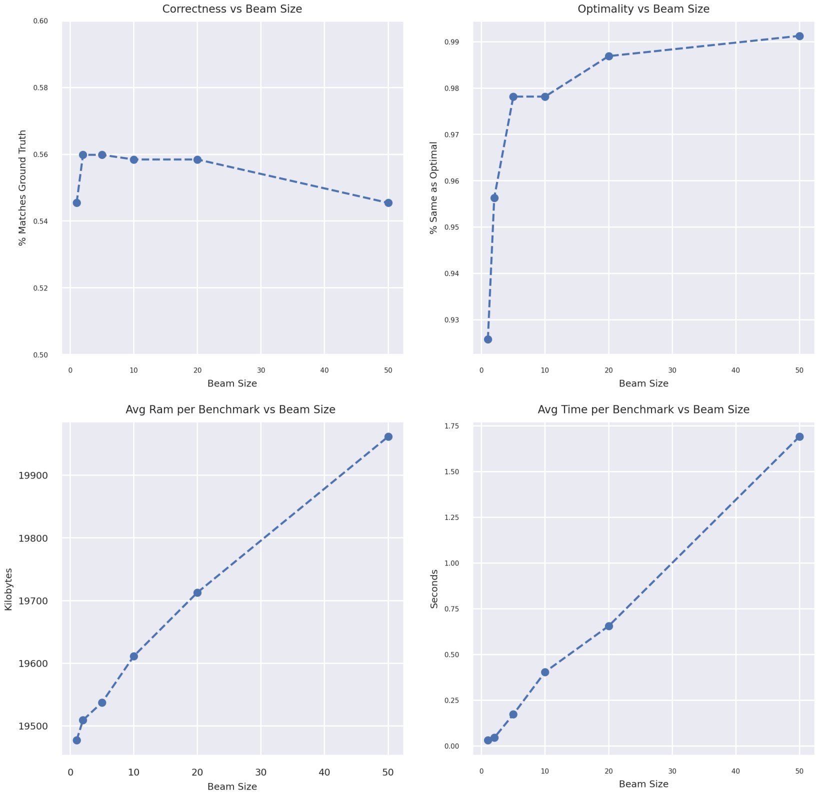


Fig. 6: Line plots of exact match accuracy, optimality, RAM, and time, as a function of beam size (for sizes 1, 2, 5, 10, 20, 50). The efficiency plots are only evaluated on the benchmarks that all beam sizes were able to finish within the resources constraints (226 benchmarks total).

loses to GVSE-A* in terms of RAM. Fig. 6 (top) plots exact match accuracy (*i.e.* percentage of benchmarks that match the ground truth) and optimality (*i.e.* percentage of benchmarks whose score matches the optimal under the cost function). As expected, optimality strictly increases with beam size. Specifically, beam of size 50 yields an optimally scoring result on 99.12% of the benchmarks, with only two benchmarks producing suboptimal results.

At the same time, accuracy peaks at beam sizes 2–5, and then decreases and converges to that of GVSE-A*. Our hypothesis is that the greedy bias of beam search with small beam sizes matches the bias used by human speakers, and therefore can get higher accuracy than the optimal search. In fact, we are not the first ones to observe this effect: this has been previously described in machine translation literature [37].

7 Related Work

7.1 Semantic Cost in Program Synthesis

Our work is related to other efforts to handle under-specification in program synthesis by incorporating semantic cost functions. One class of techniques [4,11] learns a model (*e.g.* a neural network) that maps a specification to a syntactic cost function (*e.g.* a probabilistic grammar), which allows them to use traditional search algorithms, while still incorporating information from the specification. This approach is very general, but requires hard-to-obtain training data.

Other approaches [2] add a semantic bias via data augmentation (*i.e.* adding more examples); depending on the domain, it might not be clear how to generate additional examples, and in addition, the extra examples slow down the search.

The closest approach to ours is Bayesian program synthesis [20,5,29], which uses Bayesian reasoning as the basis of semantic cost. (Closely related to it is the work on synthesizing symmetric lenses [25], which defines its cost function in terms of entropy instead of posterior probability.) We contribute to this line work by proposing a new search algorithm based on version spaces, which can efficiently optimize against a Bayesian cost function, as long as its structure is compositional in the version space.

7.2 Regular Expressions in Program Synthesis

Regular expressions have been a popular target for program synthesis, thanks their utility and relative simplicity. A common thread in this line of work [22,10,38,34] is using enumerative search to generate regexes from positive and negative examples. AlphaRegex [22] proposed top-down search with pruning based on over- and under-approximation, to synthesize regular expressions over a small alphabet ($\{0, 1\}$). Regae [38] and Regel [10] use similar top-down search, but target real-world regexes (over the full ASCII alphabet); to guide the search, the former relies on extensive user interaction, while the latter uses information extracted from natural language descriptions via semantic parsing. Paresy [34] contributes a bottom-up search algorithm for regex synthesis, which can be accelerated by GPUs, achieving significant speedups over top-down search. Importantly, none of these algorithms support optimization *wrt.* a Bayesian cost function. Although prior work has shown [23,6] how to use enumerative search to optimize against purely *syntactic* cost—like our simplicity—it is not clear how to extend it to handle *semantic*, data-dependent cost—like our specificity. On the other hand, the regex synthesizers listed above support a much richer class of regexes than our tool, and for those more complex regexes, additional input modalities (such as negative examples, natural language descriptions, or use interaction) are indispensible.

7.3 Regular Expressions in Machine Learning

Outside of the program synthesis community, regular languages have been a long-standing target in the machine learning, starting from the seminal work of [3]. Some of these technique do tackle learning from only positive data. A more theoretical line of work [16,13,12] is concerned with characterizing classes of regular expressions that are learnable *in the limit* (*i.e.* given enough data). Others propose algorithms for text classification [28,36], text extraction [8], or XML schema inference [7,14]. These techniques, however, are designed to work on a large amount of data, and many of them are approximate, whereas we are interested in learning from a small number of examples, and producing *sound* solutions (that are guaranteed to match the examples).

Recent advances in neural networks made it possible to generate regexes from a range of textual descriptions [24,15,26]. Although tools like Github Copilot [15] and ChatGPT [26] are capable of inferring high-quality regexes from a few positive examples, these tools are unpredictable and often generate unsound solutions.

7.4 Synthesis with Version Space Algebras

While not explicitly targeting regular expressions, tools such as FlashFill [17] and BlinkFill [32] also identify regular patterns as part of synthesizing string transformations, although the class of patterns they support is more restricted. The structure of our version spaces is directly inspired by the InputDataGraph data structure in BlinkFill, but we use it for a different purpose and extend it in several ways, most notably by adding optionals. The main challenge in VSA-based synthesis is that version spaces grow very large as the number and length of examples increases. Recent work by [9] proposes to address this via *guarded DSLs*, a special *local* ranking function that enables optimization without constructing the full version space; this function, however, has very restricted applicability, and would not work for our problem, where *all* atomic parts of a regex contribute to its score, and no single part dominates. Instead we propose GVSE, a more general approach to reducing memory demands of VSA-based synthesis, by combining it with efficient graph search algorithms, such as A* and beam search.

8 Conclusions and Future Work

We have presented an efficient algorithm for Bayesian synthesis of regular expressions from positive examples. Our algorithm is based on three key insights:

1. Version spaces are a good fit for Bayesian synthesis because an edge in a version space encodes not only a part of a *program*, but also a part of the *specification* that it solves, making it possible to decompose a Bayesian cost function into a sum of edge costs.
2. We can obtain an admissible A* heuristic for a version space for n examples by simply using the cost of the best program for some $m < n$ examples.
3. Beam search over a version space can produce accurate results as long as we bucket together nodes that have made comparable progress towards solving the specification.

We believe that all three of these insights are generalizable beyond regular expressions. In future work, we plan to explore the application of these ideas to other domains where version spaces [17,32] or the related data structure of *finite tree automata* [35] are used for synthesis.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), https://ieeexplore.ieee.org/document/6679385/
2. An, S., Singh, R., Misailovic, S., Samanta, R.: Augmented example-based synthesis using relational perturbation properties. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371124, https://doi.org/10.1145/3371124
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (nov 1987). https://doi.org/10.1016/0890-5401(87)90052-6, https://doi.org/10.1016/0890-5401(87)90052-6
4. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: Learning to write programs. arXiv preprint arXiv:1611.01989 (2016)
5. Barke, S., Kunkel, R., Polikarpova, N., Meinhardt, E., Bakovic, E., Bergen, L.: Constraint-based learning of phonological processes. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 6176–6186. Association for Computational Linguistics, Hong Kong, China (Nov 2019). https://doi.org/10.18653/v1/D19-1639, https://aclanthology.org/D19-1639
6. Barke, S., Peleg, H., Polikarpova, N.: Just-in-time learning for bottom-up enumerative synthesis. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428295, https://doi.org/10.1145/3428295
7. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise dtds from xml data. In: Proceedings of the 32nd International Conference on Very Large Data Bases. pp. 115–126. VLDB '06, VLDB Endowment (2006)
8. Brauer, F., Rieger, R., Mocan, A., Barczynski, W.M.: Enabling information extraction by inference of regular expressions from sample entities. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management. pp. 1285–1294. CIKM '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2063576.2063763, https://doi.org/10.1145/2063576.2063763
9. Cambronero, J., Gulwani, S., Le, V., Perelman, D., Radhakrishna, A., Simon, C., Tiwari, A.: Flashfill++: Scaling programming by example by cutting to the chase. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571226, https://doi.org/10.1145/3571226
10. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (2020)
11. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., Kohli, P.: Robustfill: Neural program learning under noisy I/O. In: Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. pp. 990–998 (2017)

12. Fernau, H.: Algorithms for learning regular expressions from positive data. Information and Computation **207**(4), 521–541 (2009)
13. Firoiu, L., Oates, T., Cohen, P.: Learning regular languages from positive evidence. In: Proceedings of the Twentieth Annual Conference of the Cognitive Science Society. pp. 350–355 (1998)
14. Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K.: Xtract: Learning document type descriptors from xml document collections. Data Mining and Knowledge Discovery **7**(1), 23–56 (2003). https://doi.org/10.1023/A:1021560618289
15. GitHub: Github copilot - your ai pair programmer. https://copilot.github.com/ (2023)
16. Gold, E.M.: Language identification in the limit. Information and control **10**(5), 447–474 (1967)
17. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)
18. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. SIGPLAN Not. **46**(6), 62–73 (jun 2011). https://doi.org/10.1145/1993316.1993506, https://doi.org/10.1145/1993316.1993506
19. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **SSC-4(2)**, 100–107 (1968)
20. Lake, B., Salakhutdinov, R., Tenenbaum, J.: Human-level concept learning through probabilistic program induction. Science **350**(6266), 1332–1338 (Dec 2015). https://doi.org/10.1126/science.aab3050
21. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by Demonstration Using Version Space Algebra. Machine Learning **53**(1), 111–156 (2003)
22. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. SIGPLAN Not. **52**(3), 70–80 (oct 2016). https://doi.org/10.1145/3093335.2993244, https://doi.org/10.1145/3093335.2993244
23. Lee, W., Heo, K., Alur, R., Naik, M.: Accelerating search-based program synthesis using learned probabilistic models. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 436–449. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192410, https://doi.org/10.1145/3192366.3192410
24. Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R.: 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. pp. 1918–1923. Association for Computational Linguistics, Austin, Texas (Nov 2016). https://doi.org/10.18653/v1/D16-1197, https://aclanthology.org/D16-1197
25. Miltner, A., Maina, S., Fisher, K., Pierce, B.C., Walker, D., Zdancewic, S.: Synthesizing symmetric lenses. Proc. ACM Program. Lang. **3**(ICFP) (jul 2019). https://doi.org/10.1145/3341699, https://doi.org/10.1145/3341699
26. OpenAI: Chatgpt. https://chat.openai.com/ (2023)
27. Polozov, O., Gulwani, S.: FlashMeta: A Framework for Inductive Program Synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 107–126 (2015)

28. Prasse, P., Sawade, C., Landwehr, N., Scheffer, T.: Learning to identify concise regular expressions that describe email campaigns. J. Mach. Learn. Res. **16**(1), 3687–3720 (2015)
29. Pu, Y., Ellis, K., Kryven, M., Tenenbaum, J.B., Solar-Lezama, A.: Program synthesis with pragmatic communication. ArXiv **abs/2007.05060** (2020)
30. Ross, B.J.: Probabilistic pattern matching and the evolution of stochastic regular expressions. Applied Intelligence **13**(3), 285–300 (Nov 2000). https://doi.org/doi:10.1023/A:1026524328760, http://www.cosc.brocku.ca/~bross/research/apin1303.pdf
31. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, 3 edn. (2010)
32. Singh, R.: Blinkfill: semi-supervised programming by example for syntactic string transformations. Proceedings of the VLDB Endowment **9**, 816–827 (06 2016). https://doi.org/10.14778/2977797.2977807
33. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: Transit: Specifying protocols with concolic snippets. SIGPLAN Not. **48**(6), 287–296 (jun 2013). https://doi.org/10.1145/2499370.2462174, https://doi.org/10.1145/2499370.2462174
34. Valizadeh, M., Berger, M.: Search-based regular expression inference on a gpu. In: PLDI (2023)
35. Wang, X., Dillig, I., Singh, R.: Synthesis of data completion scripts using finite tree automata. Proc. ACM Program. Lang. **1**(OOPSLA) (oct 2017). https://doi.org/10.1145/3133886, https://doi.org/10.1145/3133886
36. Xie, Y., Yu, F., Achan, K., Panigrahy, R., Hulten, G., Osipkov, I.: Spamming botnets: Signatures and characteristics. SIGCOMM Comput. Commun. Rev. **38**(4), 171–182 (aug 2008). https://doi.org/10.1145/1402946.1402979, https://doi.org/10.1145/1402946.1402979
37. Yang, Y., Huang, L., Ma, M.: Breaking the beam search curse: A study of (re-)scoring methods and stopping criteria for neural machine translation (2018)
38. Zhang, T., Lowmanstone, L., Wang, X., Glassman, E.L.: Interactive Program Synthesis by Augmented Examples, pp. 627–648. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3379337.3415900

A Additional Experiments

A.1 Agreement with Human Listeners

Since our human speakers were prone to error and had an untested knowledge of regular expressions, we do not know if our tool was unable to correctly infer regexes due to an imperfect cost function or as a result of the examples provided being fundamentally ambiguous . To gain more clarity, we inspect a more fine-grained breakdown of Regex+'s performance compared to humans. The high density of benchmarks in the diagonal in the first plot of Fig. 7 demonstrates that Regex+'s ability to correctly or incorrectly infer regexes is highly correlated with that of humans. This is consistent with the hypothesis that a large portion of Regex+ incorrect results could be ascribed to faulty examples. For example, row 1 in Tab. 3 illustrates a case where a speaker did not provide a single example with an upper case letter making it difficult to infer `[A-Z]` from the

Table 3: A sample of benchmarks illustrating agreement with human listeners

#	Ground Truth	Examples	Humans	Bayes	Simplicity
1	`[a-zA-Z][0-9]{5}`	q92837 j62910	`[a-z][0-9]{5}`	`[a-z][0-9]{5}`	`[a-z][0-9]+`
2	`[0-9](\.)?[0-9]`	1. 1.1234 1.1111 1.12345 3.1234 4.5	`[0-9]\.[0-9]*`	`[0-9]\.[0-9]*`	`[0-9]\.[0-9]*`
3	`[0-9]{2}\.5`	45.5 01.5 99.5	`[0-9]{2}\.[0-9]`	`[0-9]{2}\.5`	`[0-9]+\.[0-9]+`
4	`[0-9]{6}`	928365 657483 019284	`[0-9]+`	`[0-9]{6}`	`[0-9]+`

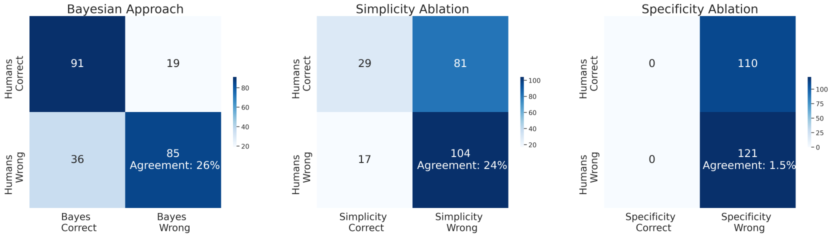


Fig. 7: Confusion matrices that show the breakdown of the number benchmarks achieved by system v.s humans. Separate matrices are shown for the Bayesian approach, simplicity only ablation and specificity only ablation. The cell that indicates benchmarks that both humans and system got wrong also contains a percentage for benchmarks that both humans and system guessed the same answer.

examples. Similarly, for row 2 one would not guess `(\.)?` since all of the speaker's examples had "." in them. On examples that humans were able to guess, and thus can be confidently considered to be of good quality, Regex+ achieves 84% accuracy. Further, on examples that humans and Regex+ could not solve, Regex+ matched humans examples 26% of the time (eg: rows 5 and 6) In cases where Regex+ got the example correct and humans did not, humans tended to be more general than Regex+. In row 3 of Tab. 3, a listener failed to identify the '.5' and in row 4 the listener either did not see that all examples had 6 digits or did not consider it important enough. It is hard to determine whether the unnecessary generality of these guesses was a result of careless or some other cognitive bias. While the underlying cause is beyond the scope of this paper, this observed pattern could explain the 24% agreement on the benchmarks that both humans and simplicity ablation got wrong. Nonetheless, as seen in Fig. 7, neither of the ablations demonstrated as dense a diagonal as Regex+, consistent with the conjecture that humans leverage both specificity and simplicity to reason about regular expressions.