Scalable Synthesis of Regular Expressions From Only Positive Examples

MARK BARBONE[*] and ELIZAVETA PERTSEVA[*], University of California, San Diego, USA

Category: Undergraduate.
Advisors: Nadia Polikarpova and Taylor Berg-Kirkpatrick.

Synthesizing regular expressions from user-provided examples is a popular research area for programming by example (PBE) systems. Yet, synthesis from only positive examples remains an unsolved challenge due to a lack of a clear criterion to select the best solution and an infinite search space. Existing tools [3, 4, 8, 12, 13] avoid this problem by requiring a wealth of additional information, such as negative examples or natural language descriptions. Our prior work Regex+ [9] tackled the first challenge by introducing a *pragmatic ranking function*, which tripled the accuracy of existing neural and enumerative synthesizers on positive-example-only benchmarks. This paper builds upon Regex+, by addressing the second challenge of scalability. We introduce an admissible A* heuristic that relies on the prior ranking function, achieving a **90x** decrease in memory usage and **1.9x** speedup on a novel suite of benchmarks collected from a human study.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; • **Context specific languages** → **Programming by example** ; • **Theory of computation** → **Regular languages**.

Additional Key Words and Phrases: Program synthesis, Regular expressions, A*

1 INTRODUCTION

Suppose you are a data scientist searching for Brazilian CNPJs (company identification numbers) in a large document. You quickly identify five initial CNPJs:

```
60.701.190/0001-04, 33.000.167/0001-01
02.916.265/0001-60, 00.623.904/0001-73
           00.000.000/0001-91
```

But you lack the domain knowledge to write negative examples or natural language descriptions. Writing regexes is difficult [2], so a possible solution is to input the examples into a synthesis tool. However, to the best of our knowledge, no existing one-shot regex synthesizer correctly solves this problem. Enumerative tools [4, 12, 13] require negative examples, while neural tools [3, 4, 8, 12] need natural language descriptions to perform well.

In prior work [9], we attempted to find a solution by introducing a *pragamatic ranking function*, which balanced simplicity and specificity. Our tool Regex+ was able to correctly generalize from only positive examples but could not scale past only a few inputs. For this specific example, Regex+ reaches memory overload with more than 3 CNPJS.

In this work, we address the scalability shortcomings with new search techniques, which, from the five inputs, output

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

the correct answer for the CNPJ example in less than a minute, using less than 30MB of RAM.

1.1 Challenges and Solutions

Synthesizing from positive examples has three main challenges, which we will briefly outline below. Our prior work focuses on the first one, while here, we attempt to address the next two.

[*]Both authors contributed equally to this research. ACM membership numbers: 9831285 and 4261444

Authors' address: Mark Barbone, mbarbone@ucsd.edu; Elizaveta Pertseva, epertsev@ucsd.edu, University of California, San Diego, 9500 Gilman Dr., La Jolla, California, USA, 92093.

1.1.1 *Ranking Function.*

In contrast to many synthesis problems, where the challenge lies in finding any program that matches the specification, in this case, solutions are readily found but are mostly unhelpful to the user. For example, both

`.*` and `(60\.701\.190/0001-04)|(33.000.167/0001-01)`

match the first two CNPJ examples, but one is too permissive, and the other is too specific.

In our prior work [9], we introduced a *pragmatic ranking function* that assumes that users act as rational speakers and choose examples in order to teach a concept. The idea of pragmatics has recently been explored in other domains in program synthesis but with different models. [10] Our pragmatic ranking function assigns each regex specificity and simplicity scores, which respectively come from the probability the regex generated the inputs and the probability of the regex itself. For the CNPJ example, our ranking function selects the correct answer.

1.1.2 *Search Strategy.*

Efficiently searching for the best regex presents a second challenge. Enumerative strategies [1, 4, 7] face combinatorial explosion, as the correct answer can be very complex: for instance, the regex for CNPJ's has 18 components. On the other hand, although representation-based search strategies as in BlinkFill or FlashFill [5, 11] and our prior work can make search faster by tailoring the search space to the particular problem, in our use case the resulting datastructures grow exponentially and take up too much memory.

Our solution uses A* search guided by our pragmatic ranking function, with a pragmatic admissible heuristic. Notably, our ranking function depends on the inputs and thus does not face the same issues as enumerative search with constant weights. Our A* search also constructs the version-space algebra (VSA) datastructure [6] from representation-based search on demand, and so it reaps the benefits of representation-based search without the associated memory demands.

1.1.3 *Benchmarks.*

Finally, we need benchmarks to evaluate our synthesis tool. To the best of our knowledge, there does not currently exist a comprehensive benchmark suite for regex synthesis from positive examples. While prior work has mainly relied on scraping stack overflow [4], the majority of posts rely on natural language descriptions, thus containing an insufficient number/quality of positive examples.

To remedy this problem, we introduce a human study framework that gathers examples from different users for a custom set of regexes. We encode example generation as a game, incentivizing users to provide helpful examples. We introduce a fictional character Charlie, the recipient of the examples who is trying to understand the pattern, to cultivate empathy and increase user investment in the task.

In summary, our work contributes the following:

- An implementation of A* search for regex synthesis
- A pragmatic admissible heuristic for A* search
- A framework for collecting positive examples conveying regexes from real users

2 SEARCH

We first describe the graph on which we perform A*, and then describe our admissible heuristic.

2.1 The Graph

To efficiently represent the search space, we choose a fixed grammar of atomic regex components, and then limit our search to sequences $r_1 r_2 \dots r_k$ of atomic components. This results in a grammar which is simple enough to enable efficient synthesis while also sophisticated enough to express useful regular expressions.

We formulate the synthesis problem as a graph search on the DAG whose nodes are tuples $(i_1, \dots, i_N)$ of indices into the given examples, and the edges from $(i_1, \dots, i_N)$ to $(j_1, \dots, j_N)$ are atomic components matching the corresponding substrings. An acceptable regex is then a path from $(0, \dots, 0)$ to $(|e_1|, \dots, |e_N|)$ in this graph.

In this work our A* search never fully constructs this graph, saving memory. Instead, we compute edges and nodes as they are explored during the search process.

2.2 The Heuristic

Our pragmatic ranking function assigns each edge in the graph a weight as a sum of a simplicity score, and a specificity score for each example:

$$\textsc{Actual}(e_1, \dots, e_N) = \min_{R=r_1\dots r_k} \left[\textsc{Simplicity}(R) + \sum_{i=1}^{N} \textsc{Specificity}(R, e_i) \right] \tag{1}$$

Intuitively, the simplicity score rewards simpler regexes while the specificity score rewards regexes which match the inputs more closely. These scores are derived from probabilities, so that the most likely regex according to our probabilistic model receives the best score.

As the specificity scores are always nonnegative, (1) may be soundly underapproximated by considering some subset $S \subseteq \{1, \dots, N\}$ of the examples:

$$\begin{aligned}\textsc{Heuristic}_S(e_1, \dots, e_N) &= \min_{R=r_1\dots r_k} \left[\textsc{Simplicity}(R) + \sum_{\substack{i=1 \\ i \in S}}^{N} \textsc{Specificity}(R, e_i) \right] \\ &\le \textsc{Actual}(e_1, \dots, e_N).\end{aligned}$$

For small sizes of S, we may efficiently precompute the heuristic using VSAs, and then take the maximum over any number of choices of S. Empirically, we found that $|S| = 2$ strikes the best balance between not using too much memory while still successfully pruning the space and speeding up the search.

3 RESULTS

Using our human study framework, we collected 373 total responses from 20 users for 19 regexes taken from the Regel benchmark suite [4] and originally coming from Stack Overflow. We ran the human study on Prolific [1] and included \$12 an hour compensation, with the study on average taking 30 minutes. The number of provided examples varied from 0 to 13. We chose to remove any submission that had 0 examples provided. Users were screened for programming experience.

Our results show a 7% relative decrease in total percent failed, and a 5% relative increase in total correct, as illustrated in Table 1. Further, the average time per problem decreases 1.9x and the average RAM usage decreases 90x.

	Correct	Ran out of RAM (8GB Max)	Ran out of time (5 minutes max)	Total failed
VSA Search	25.46%	15.01%	0%	15.01%
A* Search	**26.00%**	**0%**	**14.20%**	**14.20%**

Table 1. Performance Comparison

[1]https://www.prolific.co/

REFERENCES

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[2] 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, 405–416.

[3] 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).

[4] 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).

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

[6] 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.

[7] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[8] 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

[9] Elizaveta Pertseva, Mark Barbone, Joey Rudek, and Nadia Polikarpova. 2022. Regex+: Synthesizing Regular Expressions from Positive Examples. *11TH Workshop on Synthesis* (August 2022). https://par.nsf.gov/biblio/10336574

[10] Yewen Pu, Kevin Ellis, Marta Kryven, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2020. Program Synthesis with Pragmatic Communication. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1111, 11 pages.

[11] 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

[12] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2020. Benchmarking Multimodal Regex Synthesis with Complex Structures. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 6081–6094. https://doi.org/10.18653/v1/2020.acl-main.541

[13] 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, 627–648. https://doi.org/10.1145/3379337.3415900