Playing Regex Golf with Genetic Programming

Alberto Bartoli
DIA - University of Trieste
Italy
bartoli.alberto@units.it

Andrea De Lorenzo
DIA - University of Trieste
Italy
andrea.delorenzo@phd.units.it

Eric Medvet
DIA - University of Trieste
Italy
emedvet@units.it

Fabiano Tarlao
DIA - University of Trieste
Italy
fabiano.tarlao@phd.units.it

ABSTRACT

Regex golf has recently emerged as a specific kind of code golf, i.e., unstructured and informal programming competitions aimed at writing the shortest code solving a particular problem. A problem in regex golf consists in writing the shortest regular expression which matches all the strings in a given list and does not match any of the strings in another given list. The regular expression is expected to follow the syntax of a specified programming language, e.g., Javascript or PHP.

In this paper, we propose a *regex golf player* internally based on Genetic Programming. We generate a population of candidate regular expressions represented as trees and evolve such population based on a multi-objective fitness which minimizes the errors and the length of the regular expression.

We assess experimentally our player on a popular regex golf challenge consisting of 16 problems and compare our results against those of a recently proposed algorithm—the only one we are aware of. Our player obtains scores which improve over the baseline and are highly competitive also with respect to human players. The time for generating a solution is usually in the order of tens minutes, which is arguably comparable to the time required by human players.

Categories and Subject Descriptors

I.5.4 [**Pattern Recognition**]: Applications—*text processing*; H.4.1 [**Information Systems Applications**]: Office Automation—*word processing*

Keywords

Regular Expressions, Genetic Programming, Programming by Example, Machine Learning

1. INTRODUCTION

Regex golf has recently emerged as a specific kind of code golf, i.e., unstructured and informal programming competitions aimed at writing the shortest code solving a particular problem. A problem in regex golf usually consists in writing the shortest regular expression which matches all the strings in a given list and does not match any of the strings in another given list. Examples of such lists could be the names of all winners of an US presidential election and of the names of all losers (the specific constraints on the contents of these lists will be clarified later, e.g., their intersection must be empty). A trivial way for generating systematically a regular expression with these requirements consists in building a disjunction of all the desired matches—i.e., all the matches glued together by the | character which, in common regex syntax, means "or". To reward non trivial solutions, the *score* assigned to a given solution is higher for more compact expressions.

There has recently been a growing interest toward regex golf in the programmers' communities, motivated more by the challenge itself than by the actual utility of any given problem. Such interest has been fueled further by a blog post of a famous researcher—Peter Norvig—in which a simple yet powerful algorithm for solving regex golf problems systematically is proposed [16]. Norvig points out that problems of this sort are related to set cover problems, which are known to be NP-hard, and describes a greedy algorithm which is very efficient and works well in a number of cases, while at the same time identifying the fundamental trade-offs made in his proposal.

In this paper, we propose a methodology based on Genetic Programming (GP) for generating solutions to regex golf problems—a *regex golf player*. We generate a population of candidate regular expressions represented as trees and constructed with carefully selected regular expression operators and constants. We evolve such population based on a multi-objective fitness which maximizes the correct handling of the provided matches and unmatches while minimizing the length of the regular expression represented by the individual.

We implemented our proposal and assessed its performance on a recently proposed suite of 16 regex golf problems which is very popular. We used as baseline the algorithm proposed by Norvig—the only one we are aware of—and an existing GP-based system for generating regular expressions for text extraction tasks by examples [3]. Our proposal compares very favorably to the baseline and obtains the highest score on the full suite. We also attempted the construct a baseline based on scores obtained by human players, which is difficult because no structured collections of human players results are available: however, we collected several results by crawling the web and found that our proposal is ranked in the top positions.

A prototype of our regex golf player is available at `http://regex.inginf.units.it/golf`.

2. RELATED WORK

The only algorithm explicitly designed for solving regex golf-related problems which we are aware of is the one by Peter Norvig mentioned in the introduction. We used this algorithm as baseline for our proposal.

Several proposals for learning regular expressions from examples exist for *text extraction* problems [18, 13, 1, 12, 5, 2, 3]. Text extraction from examples is radically different from regex golf in several crucial points. First, regex golf assumes an input stream segmented so that the input strings listed in the problem specification are processed by the solution one segment at a time. Text extraction requires instead the ability to identify and extract specific portions from a longer stream. In other words, regex golf consists in binary classifying input strings whereas text extraction requires the identification in the input string of the boundaries of the substring(s) to extract, if any. Second, a regex golf problem places no requirements on how strings not listed in the problem specification will be classified. Text extraction requires instead a form of generalization, i.e., the ability of inducing a general pattern from the provided examples. Third, text extraction requires the ability to identify a context for the desired extraction, that is, a given sequence of characters may or may not constitute a match depending on its surroundings. A requirement of this form is not meaningful in regex golf.

For example, a regex golf problem requiring the match of all winners of US presidential elections and no loser may be solved with a disjunction of `ls` and several short regexes [16]. Such a regular expression is not useful for the text extraction problem, because applying it to a superstring of a winner would provide no information about the substring which actually identifies the winner. Furthermore, any string containing the substring `ls` will thus be matched by the regex. On the other hand, a regex generated for text extraction might be applied to regex golf but it would be largely suboptimal: the solution generation process must induce a general pattern and there is clearly no syntactical pattern capable of predicting the names of future US presidents. In other words, learning approaches tailored to text extraction purposefully attempt to prevent any overfitting of the examples which is instead a necessity in regex golf.

Our proposal builds on the text extraction method in [3], which we modified and specialized by taking the specific requirements of regex golf into account. We included the cited method in the baseline because, although it was designed for a different problem, it is available as a webapp[1] and its inclusion demonstrates that solving regex golf effectively calls for a specialized solution.

Another proposal for learning regular expressions from examples is [6], but this work considers a problem whose requirements are a mix of regex golf and text extraction. On the one hand, the problem consists in merely classifying input strings without the need of identifying the boundaries of the matching substrings. On the other hand, the problem assumes input streams not necessarily segmented in advance at the granularity of the desired matches and unmatches. Moreover, and most importantly, the cited work aims at inferring a general pattern capable of solving the desired task beyond the provided examples.

Since a regular expression may be obtained from a deterministic finite automata (DFA), approaches for learning a DFA from labelled examples and counterexamples could be used (e.g., [14, 4]; see [7] for a survey). On the other hand, such proposals assume the number of states of the target DFA is known and, most importantly, they are not concerned with minimizing the length of the regular expression corresponding to the generated DFA. While approaches of this form may deserve further investigation, they do not appear to match the specific requirements of regex golf. Similar remarks may be applied also to proposals for induction of non-deterministic finite automata (NFA) from labelled examples [10, 17].

Finally, regex golf might be seen as a problem in the broader category of *programming by examples* (PBE), where a program in a given programming language is to be synthesized based on a set of input-output pairs. Notable results in this area have been obtained recently for problems of string manipulation [11, 15] and some of the corresponding algorithms have been included in the latest release of Microsoft Excel (Flash-Fill functionality). While such approaches are able to deal with *context-free grammars* and are thus potentially able to solve classification problems of the form encountered in regex golf, they use an underlying language which is much richer than regular expressions and thus may not generate solutions useful for regex golf.

3. THE PROBLEM

While the term "regex golf" may indicate any challenge requiring the generation of a regular expression, its usual meaning is the one described in the introduction and formalized as follows.

We consider strings constructed over a large alphabet $\alpha =$ UTF-8. Strings may potentially include arbitrary characters in the alphabet, including spaces, newline and so on. A problem instance is defined by $\mathcal{I} = (M, U)$, where M and U are sets of strings whose intersection is empty.

The problem consists in generating a regular expression which:

1. matches all strings in M;
2. does not match any string in U; and,
3. is shorter than the regular expression constructed as a disjunction[2] of all strings in M.

Note that, for a given problem instance, it might not be known whether a regular expression satisfying the above requirements actually exists. Furthermore, given a solution r' satisfying the three requirements, it might not be known whether there exists a shorter solution r'' satisfying requirements 1 and 2.

[1] `http://regex.inginf.units.it`

[2] More precisely, the disjunction of all strings in M, where each string is prefixed by the "start of string" anchor `^` and postfixed by the "end of string" anchor `$`.

Solutions may satisfy requirements 1 and 2 in part. That is, a solution might fail to match one or more strings in M and/or match one or more strings in U. Solutions are thus given a *score* quantifying their behavior in terms of the desired matches and unmatches, as well as their compactness.

We use the score definition in `http://regex.alf.nu`, from which we have also collected the suite of problem instances for our experimental evaluation. The definition is as follows. Let r be a candidate solution, let n_M and n_U denote the number of elements in M and U, respectively, which are matched by r. The score of r on instance $\mathcal{I} = (M, U)$ is:

$$w_{\mathcal{I}}(n_M - n_U) - \text{length}(r)$$

where $w_{\mathcal{I}}$ is a statically defined value which is meant to weigh the "difficulty" of the problem instance $\mathcal{I}$. Note that the numerical value of the score, as well as the range of possible values, is problem instance-dependent and that a solution may obtain a negative score.

4. OUR APPROACH

The system requires a description of the problem instance $\mathcal{I} = (M, U)$ and generates a Javascript-compatible regular expression. A prototype is available at `http://regex.inginf.units.it/golf`.

Our proposal builds on the text extraction method in [3], which we modified and specialized by taking the specific requirements of regex golf into account. We will summarize the differences at the end of this section.

Every individual of the GP search process is a tree τ, where labels of leaf nodes are taken from a specified *terminal set* and labels of internal nodes from a specified *function set* as follows.

The function set consists of the following regular expressions operators (the central dot · represents a placeholder for a child node): *possessive quantifiers* (`·*+`, `·++`, `·?+` and `·{·,·}+`), *group* `(·)`, *character class* `[·]` and *negated character class* `[^·]`, *concatenator* `··`—a binary node which concatenates its children—and *disjunction* `·|·`. We did not include greedy or lazy quantifiers [9] because, as indicated in [3], these operators lead to execution times which are not practically acceptable.

The terminal set consists of a set of terminals which do not depend on the problem instance $\mathcal{I}$ and other terminals which depend on $\mathcal{I}$. Instance independent terminals are: the alphabetical *ranges* `a-z` and `A-Z`, the start of string *anchor* `^` and the end of string anchor `$`, and the *wildcard character* `.`. Instance dependent terminals are: all characters appearing in M, *partial ranges* appearing in M, and *n-grams*.

Partial ranges are obtained as follows. We (i) build the sequence C of all characters appearing in M (without repetitions), sorted according to natural order; (ii) for each maximal subsequence of C which includes *all* characters between subsequence head c_h and tail c_t, build a partial range c_h-c_t. For example, if $M = \{$`bar`, `den`, `foo`, `can`$\}$, then the partial ranges are `a-e` and `n-o`.

n-grams are obtained as follows. We (i) build the set N of all n-grams occurring in M and U strings, with $2 \leq n \leq 4$; (ii) give a score to each n-gram as follows: $+1$ for each string in M which contains the n-gram and -1 for each string U which contains the n-gram; (iii) sort N according to descending score and (iv) select the smallest subset N' of all top-scoring n-grams such that the sum of their scores is at least $|M|$ and each individual score is positive. For example, if $M =$ $\{$`can`, `banana`, `and`, `ball`$\}$ and $U = \{$`indy`, `call`, `name`, `man`$\}$, then the n-grams are `an` and `ba`, as they are the two top-scoring n-grams and the sum of their scores is $2 + 2$.

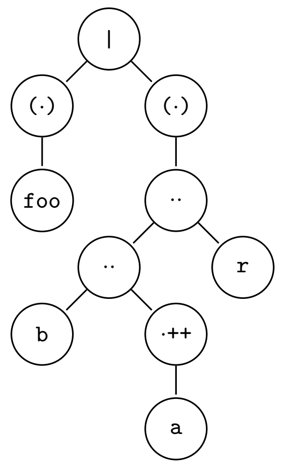

Figure 1: Tree representation of the regular expression `(foo)|(ba++r)`.

A tree τ is transformed into a string r_τ which represents a regular expression by means of a depth-first post order visit—Figure 1 shows an example of a tree and the corresponding regular expression (in the caption). In our implementation, each regular expression is evaluated by the Java regular expression engine, which works with possessive quantifiers. However, the regex golf competition being considered accepts only Javascript-compatible regular expressions and Javascript regular expression engine does not work with possessive quantifiers. Hence, we further transform r_τ into a Javascript-compatible regular expression by means of a mechanical transformation [9].

The initial population is generated as follows. Let $n_P = 500$ be the size of the population to be generated. For each string s in M, we generate an individual corresponding to s, built using only the concatenator node and single characters of s as terminals. We generate the remaining $n_P - |M|$ individuals randomly, with the ramped half-and-half method and depth of 1–5 levels.

We drive the evolutionary search based on two fitness indexes associated with each individual. Let r be an individual and let n_M and n_U be the number of elements in M and U, respectively, which are matched by r. The two fitness indexes are: $n_M - n_U$, which has to be maximized (the upper bound being $|M|$), and the length of r (in the Javascript-compatible version), which has to be minimized. We use the *Non-Dominated Sorting Genetic Algorithm II* (NSGA-II) [8] to rank individuals according to their fitness values.

We evolve the population for a number of generations $n_g = 1000$, according to the following iterative procedure. Let P be the current population. We generate an evolved population P' as follows: 10% of the individuals are generated at random, 10% of the individuals are generated by applying the genetic operator "mutation" to individuals of P, and 80% of the individuals are generated by applying the genetic operator "crossover" to a pair of individuals of P. We select individuals for mutation and crossover with a *tournament* of size 7, i.e., we pick 7 individuals at random and then select the best individual in this set, according to NSGA-

Table 1: **Salient information for the 16 problems.**

	Problem name	$\|M\|$	$\|U\|$	$w_{\mathcal{I}}$	Ideal score	Best human score	Best human solution
1	Plain strings	21	21	10	210	207	`foo`
2	Anchors	21	21	10	210	208	`k$`
3	Ranges	21	21	10	210	202	`^[a-f]*$`
4	Backrefs	21	21	10	210	201	`(...).*\1`
5	Abba	21	22	10	210	193	`^(?!.*(.)(.)\2\1)`
6	A man, a plan	19	21	10	190	177	`^(.)[^p].*$`
7	Prime	20	20	15	300	286	`^(?!(..+)\1$`
8	Four	21	21	10	210	199	`(.)(.\1){3}`
9	Order	21	21	10	210	199	`^.5[^e]?$`
10	Triples	21	21	30	630	596	`00($|3|6|9|12|15)|4.2|.1.+4|55|.17`
11	Glob	21	21	20	420	397	`ai|c$|^p|[bcnrw][bnopr]`
12	Balance	32	32	10	320	289	`^(<(<(<(<<?>?>|.9)>)>)>)$`
13	Powers	11	11	10	110	93	`^(?!(.(..)+)\1*$)`
14	Long count	1	20	270	270	254	`((.+)0 \2?1 ){7}`
15	Long count v2	1	21	270	270	254	`((.+)0 \2?1 ){7}`
16	Alphabetical	17	17	20	340	317	`.r.{32}r|a.{10}te|n.n..`
	Total				4320	4072	

II. Finally, we generate the next population by choosing the individuals with highest fitness among those in P and P'. The size of the population is kept constant during the evolution. Upon generation of a new individual, we check the syntactic correctness of the corresponding expression: if the check fails, we discard the individual and generate a new one.

In order to generate a solution for a problem instance $\mathcal{I}$, we evolve $n_e = 32$ independent populations, with different random seeds, obtaining 32 candidate regular expressions. Finally, we choose the regular expression with the highest score.

We remark the key features of our proposal (w.r.t. [3]):

1. a method for constructing the terminal set based on the problem instance $\mathcal{I}$, rather than being defined once and for all;
2. a method for initializing the population based on the problem instance $\mathcal{I}$, rather than being completely random;
3. a different functions set which includes, in particular, a disjunction operator—which is difficult to use in text extraction because it tends to promote overfitting;
4. fitness definitions based on the number of examples handled correctly—definitions proven to be inadequate for text extraction [3];
5. usage of all learning information for synthesizing candidate solutions, that is, without reserving any partition as validation set for assessing the generalization capabilities of those solutions.

5. EXPERIMENTAL EVALUATION

We considered the 16 problem instances along with the accompanying scores proposed in `http://regex.alf.nu`. Salient properties of these instances are summarized in Table 1. The table shows, for each problem instance, the ideal score—i.e., the score equal to $w_{\mathcal{I}}|M|$ which could be obtained with a zero-length regular expression matching all strings in M and no strings in U. The table also shows, for each problem instance, the highest score obtained by (different) human players[3].

5.1 Baseline

We used as baseline the algorithm by Peter Norvig, which we call Norvig-RegexGolf, and the system for generating regular expressions for text extraction presented in [3], which we call GP-RegexExtract.

We provide a brief outline of Norvig-RegexGolf below. Full details, including the (partially for fun) motivations and design trade-offs can be found in [16]. The solution for a given problem instance $\mathcal{I} = (M, U)$ is obtained as a disjunction of a set of *components*, a component being a short regular expression which matches at least one string in M and does not match any string in U. Initially, a pool of components is built with several heuristics, including the generation of a component for each n-gram of each string in M (up to n=4) and, for each such component, the generation of a component for every possible substitution of a single character with the dot character (meaning "match any" in regular expression syntax). Next a set of components from that pool is built, such that each string in M is matched by at least one component in the set. Components in the resulting set are then glued together by the or regular expression operator `|`.

Concerning GP-RegexExtract, we reimplemented the algorithm according to the details presented in [3] (see also Section 2). We set those parameters which determine the computational weight of GP to the same values for GP-RegexExtract and GP-RegexGolf, in order to allow a fair comparison of the results w.r.t. computational weight: $n_e = 32$, $n_g = 500$ and $n_P = 500$. Note that GP-RegexExtract requires that examples are partitioned in a training set and a validation set: while using it as a regex golf player, we chose to use half of M and half of U strings as training set, and the remaining string as validation set.

5.2 Results

[3]The information is obtained from `https://gist.github.com/jonathanmorley/8058871`.

We executed each of the algorithms on each problem instance and computed the score of the corresponding solution. Table 2 summarizes the resulting scores, which are presented as absolute value and as the percentage of the ideal and the best human score associated with each problem instance. Table 3 shows the regular expressions generated by GP-RegexGolf for each problem.

It can be seen that GP-RegexGolf outperforms both Norvig-RegexGolf and GP-RegexExtract: 3090 vs. -665 and 249, respectively. In particular, considering individual problem instances, GP-RegexGolf performs better than Norvig-RegexGolf in 6 problems, worse in 8 problems and obtains the same score in 2 problems. Despite obtaining a better score in 8 problems, Norvig-RegexGolf obtains a negative score on the full suite because on three problems (7, 12 and 13) it is not able to generate a non trivial solution: in these problems, the regular expression generated by Norvig-RegexGolf is the disjunction of all the M strings. Our algorithm, on the contrary, generates non trivial solutions for these problems. Concerning GP-RegexExtract, both its score on the full suite and its score on individual instances make it clear that this approach does not the requirements of regex golf.

Table 2 lists also, for each algorithm, the *competitive ratio* of the solutions [16], defined as the ratio between the length of a trivial solution (disjoining all the strings in M) and the length of the corresponding solution. Note that this index does not take matches or unmatches into account. It can be seen that both Norvig-RegexGolf and GP-RegexGolf generate solutions which are much shorter than the trivial solution for several problems: regular expressions generated by GP-RegexGolf are shorter than those of Norvig-RegexGolf in 9 problems, longer in 5 problems and with the same length in 2 problems.

Table 4 shows the time required by GP-RegexGolf and GP-RegexExtract for generating a solution. It is similar for all the problems (around 50 min) with the exception of 13, 14 and 15. For the latter problems, in which M is composed by very long strings, GP-RegexExtract attempts to generate a regular expression which extracts (rather than just matching) the each M string entirely: this leads to a population composed by very long regular expressions which require long times to be evaluated. The time required by Norvig-RegexGolf is practically negligible (less than a second per problem). All the experiments have been performed on a quad core Intel Xeon E5-2440 (2.40GHz) with 4 GB of RAM.

We wanted to investigate whether our approach can achieve better scores at the expense of increased computational weight. To this end, we repeated the experiments by setting $n_P = 1000$ and $n_P = 1500$, i.e., with an enlarged population: Table 5 shows the results in terms of score and competitive ratio. It can be seen that the full score does improve for larger values of n_P. Moreover, with $n_P = 1500$, the number of problems for which GP-RegexGolf score is not worse than Norvig-RegexGolf score is 11 vs. 8 with $n_P = 500$. The computation time for the full suite goes from 820 min for $n_P = 500$ to 1551 min and 2611 min for $n_p = 1000$ and $n_p = 1500$, respectively. As expected, with higher values for n_P GP-RegexGolf takes longer to generate a solution: yet, it is fair to claim that even such longer times may be acceptable for playing to a game of this kind.

Finally, we attempted to assess the performance of our proposal with respect to scores of highly skilled human players. We remark that there are several caveat concerning the

Table 4: Execution times of GP-RegexGolf and GP-RegexExtract algorithms in minutes.

Problem	GP-RegexGolf	GP-RegexExtract
1	53	51
2	52	52
3	53	65
4	38	25
5	34	18
6	20	23
7	33	43
8	19	27
9	46	21
10	45	25
11	44	47
12	56	71
13	71	269
14	94	289
15	95	173
16	66	64
Total	820	1262

Table 5: Scores and competitive ratio (C.R.) of GP-RegexGolf with different values for n_P.

	$n_P = 1000$		$n_P = 1500$	
Problem	Score	C.R.	Score	C.R.
1	207	66.3	207	66.3
2	208	105.5	208	105.5
3	196	11.5	197	12.4
4	146	5.2	147	6.5
5	188	12.5	186	11.5
6	142	6.0	151	4.3
7	188	24.1	188	24.1
8	183	11.7	180	10.5
9	190	8.7	190	8.7
10	456	10.5	354	27.2
11	355	28.2	522	3.2
12	36	7.3	223	26.5
13	65	46.2	40	29.7
14	191	1.0	191	1.0
15	191	1.0	191	1.0
16	259	21.1	237	10.4
Total	3201	-	3412	-

Table 2: Results of the three algorithms as score, score %, score % w.r.t. to best human score and competitive ratio (C.R., see text). For each problem, the score of the best algorithm is shown in bold.

	Norvig-RegexGolf				GP-RegexGolf				GP-RegexExtract			
Problem	Score	Score %	Hum. %	C.R.	Score	Score %	Hum. %	C.R.	Score	Score %	Hum. %	C.R.
1	**207**	98.6	100.0	66.3	**207**	98.6	100.0	66.3	170	81.0	82.1	5.0
2	**208**	99.1	100.0	105.5	**208**	99.1	100.0	105.5	185	88.1	88.9	8.4
3	191	91.0	94.6	8.5	**195**	92.9	96.5	10.7	107	51.0	53.0	7.0
4	**175**	83.3	87.0	8.0	138	65.7	68.7	6.7	−70	<0	<0	4.0
5	**186**	88.6	96.4	11.5	184	87.6	95.3	17.2	77	36.7	39.9	4.4
6	**157**	82.6	88.7	5.1	136	71.6	76.8	7.0	−246	<0	<0	0.7
7	−398	<0	<0	1.0	**188**	35.3	37.0	24.1	−52	<0	<0	13.4
8	**192**	91.4	96.5	17.5	183	87.1	92.0	11.7	−45	<0	<0	7.0
9	**190**	90.5	95.5	8.7	186	88.6	93.5	7.3	−39	<0	<0	4.5
10	**589**	93.5	98.8	6.1	430	68.3	72.2	12.6	−106	<0	<0	2.4
11	**392**	93.3	98.7	25.2	340	81.0	85.6	17.7	−163	<0	<0	4.3
12	−1457	<0	<0	1.0	**130**	40.6	45.0	11.1	−85	<0	<0	20.9
13	−1969	<0	<0	1.0	**51**	46.4	54.8	109.4	−47	<0	<0	44.2
14	189	70.0	74.4	1.0	**191**	70.7	75.2	1.0	**191**	70.7	75.2	1.0
15	189	70.0	74.4	1.0	**191**	70.7	75.2	1.0	**191**	70.7	75.2	1.0
16	**294**	86.5	92.7	18.8	132	38.8	41.6	8.0	181	53.2	57.1	11.0
Total	−665	-	-	-	**3090**	-	-	-	249	-	-	-

Table 3: Regular expressions generated by GP-RegexGolf.

Problem	Regular expression
1	`foo`
2	`k$`
3	`(^..[a-f][a-f])`
4	`v|[^b][^o][^p]t|ngo|lo|[n]o|rp|rb|ro|ro|rf`
5	`z|.u|nv|st|ca|it`
6	`oo|x|^k|ed|^m|ah|^r|v|^t`
7	`^(?=((?:x[A-Zx])+))\1x`
8	`ell|j|W|ele|o.o|Ma|si|de|do`
9	`ch|[l-p]o|ad|fi|ac|ty|os`
10	`24|55|02|54|00|95|17`
11	`lo|ro|^p|(?=((c)+))\1r|en|^w|y.|le|^p|rr`
12	`((?=((?:<<\>\>\>)*))\2(?=((?:<<<(?=(<*))\4\>\><<<<)*))\3(?=((?:<<<<<\>\>\>(?=(<*))\6\>\>\>)*))` `\5(?=((?:<<<<<<)*))\7^(?=((?:<<\><<)*))\8(?=((?:<<<\>\>\>)*))\9<<)`
13	`^(?=(((x|^)x)+))\1$`
14	`0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111`
15	`0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111`
16	`tena|[^et][^etren](?=((?:(?:ren|eren.(?=((?:(?:ren|[^ren]))+))\2|eren.(?=((?:(?:ren|` `[^ren]))+))\3))+))\1|eas`

assessment of the results obtained by human players. The web site hosting the challenge does not, at the time of this writing, provide a score ranking computed on the full suite of problems—on the other hand, there exists a collection of "Best possible answers collected so far for regex golf" (see Table 1) which shows, for each problem, the best solution. Results by human players are advertised on web forums by players themselves, often without providing any actual evidence of their results. On the other hand, there are players which do make some very good solutions publicly available, thereby simplifying the job of other players, which may either attempt to improve those solutions further or may use them for the corresponding problem instance while focusing their efforts on the remaining instances. In other words, the score obtained by a given player may actually result from efforts by multiple players. Finally, human players generally do not care to indicate the time they spent for generating a solution.

We collected several human players scores on the full suite from different web locations (including Reddit, Hacker News, Github) which we obtained by querying Google and Twitter with the search string "regex golf". Table 6 shows the 10 best scores we found, along with the total ideal score (i.e., the sum of ideal scores on the 16 problems), the best human score (i.e., the sum of the highest human player scores on the 16 problems) and the score of the three considered algorithms.

It can be seen that GP-RegexGolf would rank from 6th to 8th among human players (with $n_P = 1500$ and $n_P = 500$, respectively), whereas the scores of the other two algorithms are significantly lower than those of human players. In other words, leaving aside any caveat about how we gathered human scores, GP-RegexGolf is in the top ten of worldwide regex golf players.

6. CONCLUDING REMARKS

We have proposed and assessed experimentally an approach based on Genetic Programming for playing regex golf automatically, i.e., for generating automatically solutions to challenges which have recently become popular in the programmers' communities. The challenges consist in writing

Table 6: Total scores obtained by the 10 best humans and by the three algorithms.

	Player	Score
	Total ideal score	4320
	Best human score	4072
1	geniusleonid	4006
2	k_hanazuki	3785
3	bisqwit	3753
4	AlanDeSmet	3736
5	adamhiker	3693
	GP-RegexGolf ($n_P = 1500$)	3412
	GP-RegexGolf ($n_P = 1000$)	3201
6	adamschwartz	3181
7	flyingmeteor	3171
	GP-RegexGolf ($n_P = 500$)	3090
8	jpsim	3060
9	ItsIllak	2939
10	bg666	2683
	GP-RegexExtract	249
	Norvig-RegexGolf	−665

the shortest regular expression that matches all strings in a given list and does not match any string in another given list.

Our approach collects a score that is highly competitive against human players and improve significantly over a challenging baseline including a recently proposed algorithm tailored to this specific problem class and a recent proposal for automatic generation of regular expressions tailored to text extraction tasks. The time for generating a solution is in the order of tens of minutes and a prototype is available at `http://regex.inginf.units.it/golf`.

We think that our work shows how a GP-based approach running on modern IT machinery may deliver results, at least for this task, which are practically useful and can compete with humans.

7. REFERENCES

[1] D.F. Barrero, David Camacho, and M.D. R-Moreno. Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. *Data Mining and Multi-agent Integration*, pages 143–154, 2009.

[2] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio. Automatic generation of regular expressions from examples with genetic programming. In *International Conference on Genetic and evolutionary computation*, pages 1477–1478. ACM, 2012.

[3] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio. Automatic synthesis of regular expressions from examples. *Computer*, Early Access Online, 2013.

[4] Josh Bongard and Hod Lipson. Active coevolutionary learning of deterministic finite automata. *The Journal of Machine Learning Research*, 6:1651–1678, 2005.

[5] Falk Brauer, Robert Rieger, Adrian Mocan, and W.M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *ACM International Conference on Information and knowledge management*, pages 1285–1294. ACM, 2011.

[6] Ahmet Cetinkaya. Regular expression generation through grammatical evolution. In *International Conference on Genetic and evolutionary computation*, GECCO, pages 2643–2646, New York, NY, USA, 2007. ACM.

[7] Orlando Cicchello and Stefan C Kremer. Inducing grammars from sparse data sets: a survey of algorithms and results. *The Journal of Machine Learning Research*, 4:603–632, 2003.

[8] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182 –197, apr 2002.

[9] Jeffrey Friedl. *Mastering Regular Expressions.* O'Reilly Media, Inc., 2006.

[10] Pedro García, Manuel Vázquez de Parga, Gloria I. Álvarez, and José Ruiz. Universal automata and {nfa} learning. *Theoretical Computer Science*, 407(1–3):192 – 202, 2008.

[11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[12] Efim Kinber. Learning regular expressions from representative examples and membership queries. *Grammatical Inference: Theoretical Results and Applications*, pages 94–108, 2010.

[13] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and Ann Arbor. Regular Expression Learning for Information Extraction. *Computational Linguistics*, (October):21–30, 2008.

[14] Simon M Lucas and T Jeff Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1063–1074, 2005.

[15] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 187–95, 2013.

[16] Peter Norvig. xkcd 1313: Regex golf. `http://nbviewer.ipython.org/url/norvig.com/ipython/xkcd1313.ipynb`, January 2014.

[17] Wojciech Wieczorek. Induction of non-deterministic finite automata on supercomputers. *Journal of Machine Learning Research-Proceedings Track*, 21:237–242, 2012.

[18] Tianhao Wu and W.M. Pottenger. A semi-supervised active learning algorithm for information extraction from textual data. *Journal of the American Society for Information Science and Technology*, 56(3):258–271, 2005.