Ambiguous Probability of Random String Optimization

by Sven Nilsen, 2019

In this paper I argue that random string optimization is not sufficient for general problem solving.

With infinite computing capacity, it would be possible to solve problems by generating random strings and evaluate them to computationally mine desired knowledge. Random strings combined with infinite computing capacity can be used to construct a general problem solver[1]. The reason this kind of general problem solver is not practical, is because in most cases random strings fail to meet criteria. Infinite computing capacity is required to overcome the low probability of some string satisfying the criteria.

A natural question to ask: Can a practical general problem solver be constructed by reducing the number of cases where random strings fail to meet criteria?

Reformulation: Random string optimization means that a generator is optimized for outputting strings that succeed to satisfy some criteria with high probability.

At first sight, it seems that this reformulation of the problem is sound. It turns out that this is not the case. The reason is that measuring the probability of success is ambiguous over diversity. With infinite computing capacity, the diversity of successful strings are preserved, but this is not necessarily true for bounded computing capacity. A full specification of criteria is practically impossible.

A practical general problem solver can be constructed by random string optimization, if the measured probability of success is not extremely ambiguous over the diversity. This is because generators of low diversity are easier to construct and therefore much more probable under general optimization. Low diversity generators alone makes general problem solving impossible. One can think about diversity as a more detailed specification of success, but which is usually impractical to compute.

Definition:

x : [probability_success] p ∧ [diversity] v — Talking about generator `x`

generator : () → str — Non-deterministic generator
probability_success : generator → prob
diversity : generator → prob

probability_success <=> \(g) = $\sum s_i : \exists g \wedge success \{ (\exists_p g)(s_i) \} / \sum s_i : \exists g \{ (\exists_p g)(s_i) \}$
diversity := \(g) = $1 - sqrt(1/2 \sum s_i : str \{ (dd(s_i) - sg(g)(s_i))^2 \})$

success : str → bool — Successful string
sg : generator → str → prob — Successfully generated string
dd : str → prob — Desired distribution

sg := \(g) = \(s) = if $(\exists g)(s) \wedge success(s) \{ (\exists_p g)(s) \}$ else { 0 }

The expression `$s_i : \exists g \wedge$ success` means “successful strings generated by `g`”.
The expression `$(\exists_p g)(s_i)$` means the probability that `g` outputs `$s_i$`.

Here, `diversity` uses the definition of partial diversity[2] and measures the match of an unknown desired probability distribution `dd` with successfully generated strings `sg`. `$(\exists g)(s)$` means that the existential path[3] (co-domain) of `g` contains `s`. If the diversity is close to zero, the generator returns few strings or just a single output string. If the diversity is one, the generator is perfect.

The definition of `probability_success` is more general than it needs to be in this case.
Since the trivial path[4] `$\forall g$` of `g` is `\(_ : ()) = true`, one can replace `$\sum s_i : \exists g \{ (\exists_p g)(s_i) \}$` with `1`.
However, the definition is nice to keep for easier generalization to generators that receive inputs.

Probabilistic existential paths[5] `$\exists_p g$` are assuming finite types, while `str` is infinite. The notation here assumes that `dd` assigns non-zero probability to a finite set of strings, while zero probability is assigned for the rest of strings. It is also assumed that the generator outputs a finite set of strings. In practice, this works for infinite sets too when probabilities are estimated by sampling.

For every `x` generator with high probability of success, there *usually* exists another `y` generator with same or similar probability of success, but with higher output diversity.

Formally, the following is *almost* true:

$$\forall\ x\ \{\ \exists\ y\ \{\ \text{probability\_success}(x) \sim= \text{probability\_success}(y) \wedge \text{diversity}(x) < \text{diversity}(y)\ \}\ \}$$

This holds for *most* probability distributions of diversity and for *most* measured probability of success.

Therefore, *most* generators with high probability of success operate in environments where the measured probability of success is ambiguous over diversity.

Considering that low diversity generators are easier to construct with high success probability, it might mean that probability of success is extremely ambiguous over diversity in most cases. As a result, an optimizer not programmed to approximate diversity will fail to generate solvers for general problems.

One way to solve this is to build semantics of diversity directly into the `success` function.
A string might be successful if lots of other strings, that are successful, gets generated by the generator.

Or, one could try heuristics that require the generator to output a certain amount of diversity irrespective of whether the strings are successful or not. By selecting generators under this criteria by high probability of success, the generators selected will likely output a somewhat diverse set of strings.

References:

[1] “Solver”
Wikipedia
https://en.wikipedia.org/wiki/Solver

[2] “Partial Diversity”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/partial-diversity.pdf

[3] “Existential Paths”
Sven Nilsen, 2017
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/existential-paths.pdf

[4] “Constrained Functions”
Sven Nilsen, 2017
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/constrained-functions.pdf

[5] “Probabilistic Existential Paths”
Sven Nilsen, 2017
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/probabilistic-existential-paths.pdf