Is solver guidance redundant for strong SMT implementations?

An exploration of domain-specific vs general improvements applied to Z3's string theories.

Odysseas Machairas
Supervisor(s): Soham Chakraborty[1], Dennis Sprokholt[1]

[1] EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfillment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Odysseas Machairas
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Andy Zaidman

An electronic version of this thesis is available at http://repository.tudelft.nl/

Abstract: The Z3 SMT solver, a type of satisfiability solver used both in research and in industry, can give better performance by either improving the underlying implementation or using domain-specific guidance. We present a way to simulate domain-specific help automatically by reducing the search space based on the model solution, and we use it to compare two implementations of Z3's string solver – Z3STR3 (weaker) and Z3-NOODLER (stronger) – with and without domain-specific help. We find that Z3-NOODLER sees significantly less improvement than Z3STR3 because 1) the additional "helping" constraints are in fact occasionally counterproductive and 2) Z3-NOODLER solves equivalent problems much faster than Z3STR3, so the relative overhead of the additional constraints is more noticed.

1. Introduction

Even though SAT is the archetypal NP-complete problem [1], which in practice means no universally efficient solution exists, it is often necessary to find specific solutions in a reasonable amount of time. The SMT-LIB [2] standard expands SAT to types such as integers, bitvectors and, the topic of this paper, strings. The Z3 solver [3], which implements this standard, helps us to find satisfiable assignments to queries such as two numbers X and Y where $X + Y < 13$ and $X \cdot Y > 10$. SMT solvers are widely used: just Amazon alone does one billion daily SMT queries to verify the correctness of their AWS user policies [4], and they are employed in many other areas too [5–12].

As the use-cases become more complex and extensive, we naturally desire better performance. While optimizations and improvements may come from anywhere, one way to categorize them is between *general purpose* and *domain-specific*. General purpose improvements usually come in the form of stronger implementations (such as Z3-NOODLER [13] for Z3's strings). However, sometimes there is some exploitable property of a problem or a family of problems that you might want to take advantage of. Z3 provides the *tactics* mechanism — where the user can change some behavior of the solver or subdivide the problem — ostensibly for this reason (even though domain-specific knowledge can also be manifested as additional "refining" constraints, and tactics do not *have* to be domain-specific)

A question comes to mind when faced with these two approaches: **are there diminishing returns to using domain-specific guidance as the implementation strength increases?** That is, if an SMT implementation is particularly clever and efficient, it might be the case that guiding it is not very helpful, or even counterproductive. For instance, a human that tries to "guide" a strong chess engine will almost certainly make it perform worse, since chess engines are much better than humans. In a similar way, we explore to what extent guiding an SMT solver using domain-specific knowledge contributes more in overhead than in performance improvements.

1.1. Contributions

In this paper we describe the relationship between solver implementation strength and the effectiveness of domain-specific guidance. The main contributions of the research are:

1. An automatic procedure to simulate domain-specific knowledge on arbitrarily large datasets while providing similar amounts of help, based on a quantified reduction the search space.
2. Insight on how the performance of Z3 changes when using domain-specific guidance.
3. Understanding on how the strength of an underlying implementation affects said change in performance when using domain-specific guidance.
4. A benchmarking suite to test and compare these models, providing 100% reproducible builds of the compared implementations using Nix.

A motivating use-case for the findings presented in this paper is, for example, helping a researcher working on formal verification to make an informed decision about whether it is more worthwhile to invest time on understanding a problem versus in running or improving a solver.

2. Theoretical Background

This section provides a quick overview of how a generic SMT solver works and the differentiating features of each solver, as well as other relevant concepts and works referenced in the rest of the paper.

2.1. Anatomy of an SMT solver

The typical way that constraints over strings get resolved Z3 and other solvers is through DPLL(T) constraint propagation [3], [14]. It roughly consists in making a decision (i.e., assigning a concrete value) on one variable and propagating the implications of that decision on the rest of the constraints. For example, for two numbers X and Y and constraints $X + Y < 13$ and $X \cdot Y > 10$ a decision could be assigning $X = 10$ which propagates into $10 + Y < 13 \Rightarrow Y < 3$ and $10 \cdot Y > 10 \Rightarrow Y > 1$. Then, Y can be assigned as $Y = 2$ which successfully finds a satisfiable assignment for this instance. If at any point in the propagation a variable has no possible options, then the last decision is deemed infeasible.

In the case of strings, we can split decisions into the length and the string itself. For instance for a variable X we can make a decision that $|X| = 2$, do the necessary propagation to see if it is infeasible, and then make a decision on a 2-character variable.

Additionally, in practice, many constraints over strings have an analogous linear integer arithmetic (LIA) equation over their lengths. For example, concatenation of strings implies sum of their lengths, i.e.: $X = a \mathbin{+\!\!+} b \mathbin{+\!\!+} c \Rightarrow |X| = |a| + |b| + |c|$ (where $\mathbin{+\!\!+}$ indicates string concatenation and $| \cdot |$ indicates length).

We show a summarized diagram of the procedure for an SMT problem in Figure 1.

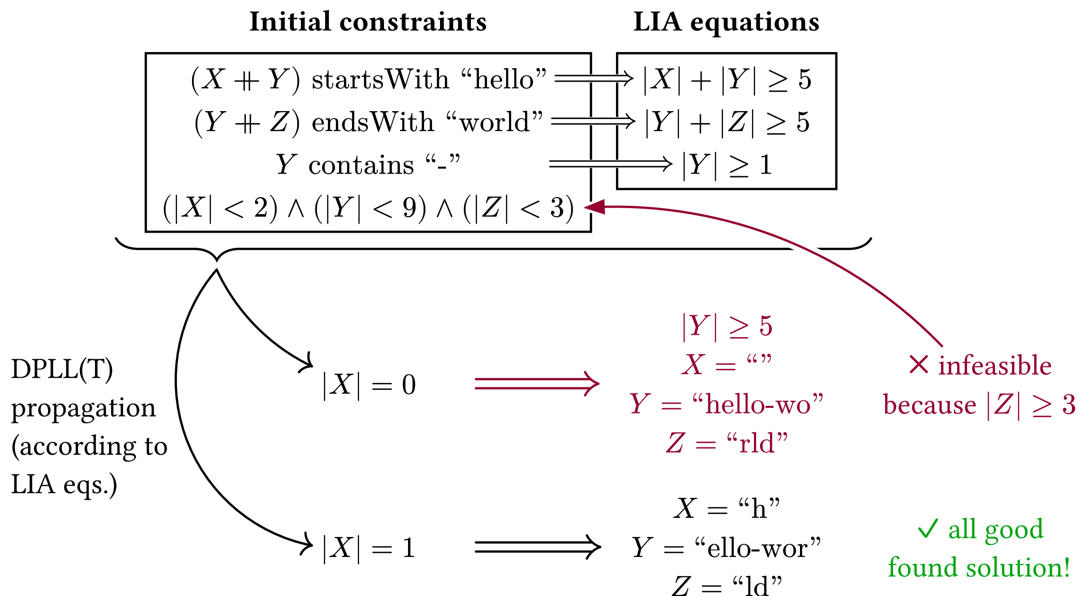

Figure 1: Generic string solving procedure. Initial constraints imply LIA equations. Lengths that are valid for LIA eqs. are checked and propagated (full propagation procedure not shown).

2.2. String solver implementations

The two implementations compared in this paper are Z3str3 and Z3-Noodler:

- Z3str3 [15] is the official upstream solver used by Z3 since 2017. It generally resembles the standard approach for string solving explained in the previous section, apart from relatively incremental improvements.
- Z3-Noodler [13, 16] is the state of the art in string solving and the winner of SMT-COMP 2024 [17]. It employs many novel techniques, such as comparing the NFAs that result from regular expressions directly. This, along with other optimizations, are deviations from the standard solving approach, and thus from the mental model of constraint propagation, which makes it harder to predict whether guidance from the user will ultimately be useful.

2.3. *Sat* vs *unsat*

For each problem, SMT solvers can return one of three results: satisfiable (*sat*), unsatisfiable (*unsat*) or unknown. Unknown gets returned when the solver cannot find a solution, typically because of a timeout, otherwise returning either *sat* or *unsat*.

While these two cases may look similar, they are fundamentally different. Namely, finding *sat* is NP-complete, finding *unsat* is **co**NP-complete [18]. On *sat* cases Z3 returns a model solution (i.e., $X =$ "hello", $Y =$ "world") while on *unsat* you can get an unsatisfiability core. The helpfulness of the *unsat core* given by Z3 has no guarantees (in fact, just returning the original problem is perfectly valid), and in practice Z3 seems to struggle with *unsat*. As we will explain in Section 3.1, in this paper we only deal with the *sat* case.

2.4. Tactics vs domain-specific solver guidance

It is important to note that tactics and domain-specific solver guidance are not interchangeable in general. Even though the existence of tactics is motivated by the ability to model domain-specific knowledge, one can use tactics in a general-purpose way; and, conversely, one can add domain-specific knowledge without using tactics.

An example of the "general purpose tactics" is *portfolio solving* [19], where the solver tries various tactics in parallel and retrieves the first one, but in fact the contract of a tactic is wide enough to allow one to implement a whole different general-purpose SMT solver such as Cvc5 [20] itself as a "tactic".

And, on the other hand, when modeling a problem, there is often a minimal set of constraints that has the sought-for satisfiability result, but you can add more constraints that reduce/refine the search space (while keeping the same result). Symmetry-breaking constraints are a particularly common example, but they can be arbitrarily complex. The presented research uses this latter approach, which seems to be more common for string solving regardless.

2.5. Other related work

It is common practice to include benchmarks when introducing a new approach for a solver. Z3str3 and Z3-Noodler are no exception. Z3str3's benchmarks are relatively brief [15, pp. 57-58], while Z3-Noodler's are very extensive, analyzing and explaining each problem set used [13, pp. 27-30]. The results presented in the paper are somewhere between these two in terms of detail.

There is less research on domain-specific guidance for SMT solvers. In [21] they use a Monte Carlo Tree Search approach to synthesize strategies, and papers like [22] and [23] present machine-learning-based approaches to implement tactics, all of which carry out benchmarks. However, these papers present *general-purpose* tactics, since their goal is to find a solution faster automatically given the problem. This differs from the topic of this paper, which is simulating what happens when the same solver has additional information about the problem.

Finally, there is research on more general guidance for computer proofs, which overlaps in terms of use-cases with SMT solving. For example, in [24] they successfully apply guidance to an automated proving technique known as equality saturation. Most of the research in this area, however, does not have an analogue of different underlying "strength" levels for the computerized proving itself.

3. Methodology

The main goal of the presented research is to compare the difference in performance for various Z3 string solver implementations. To do that we need to run an experiment, for which we need:

1. A dataset of SMT2 problems about strings to benchmark the performance of each solver.
2. Domain-specific knowledge for each problem, or a simulation thereof.
3. A program to carry out the benchmarks in a reproducible way.

To start, in terms of datasets, we do not have to look any further than the SMT-LIB dataset for strings, which provides over 100k varied test cases for strings. Specifically, we use `QF_S`, `QF_SLIA` and `QF_SNIA`, non-incremental 2024 edition [25] (`QF` indicates quantifier-free; `S`, strings; and `LIA` and `NIA`, linear/non-linear integer arithmetic, respectively).

The "creation" of domain-specific knowledge and the runner implementation are more complicated and thus constitute the rest of this section.

3.1. Automatic simulation of domain-specific knowledge

Domain-specific knowledge should help the solver find a solution faster. One could study a few problems in detail to find some specific optimization, but it very quickly becomes intractable at the scales needed to have a chance of being representative of what happens "in general".

Thus, we present an automatic procedure to simulate domain-specific knowledge. At a high level, it works by taking the solution (which can be obtained by running the solver normally) and using it to give "hints" to the solver. The "hints" are, concretely, six types of constraints we can add to a variable, given a solution. The additional constrains help a typical constraint propagator since they cut off earlier decisions that are ultimately infeasible.

However, the constraint types are fundamentally different so we need to quantify how much help each constraint gives. We do this by calculating the log increase of the probability of guessing the solution given said constraint. Then, we can choose a value for help and add constraints until we reach the desired value.

The resulting procedure (shown in Algorithm 1, at the end of the section) gives sensible results. E.g., if we take the string "hello world" and constrain it to end with "rld" then we expect the help to be somewhere around $\frac{3}{11} \approx 27.2\%$ (since we revealed 3/11 characters), and indeed the help comes out to 26.7%[1]. The rest of this section explains the motivation behind this approach and its implementation in more detail.

3.1.1. Constraints and quantifying help

There are many constraints for strings but, generally, given a solution for a variable X there are six straightforward, standalone types of constraints one might add, listed in Table 1 (for concise referencing later, we give each constraint a symbol to represent them).

However, applying these constraints can be tricky. One could choose a random valid value for either a substring of s, ∂s, or a length ℓ, and add the constraints as given, but the help

[1] The full 27.2% help would be given if we already knew the length, but this constraint only reveals that the length is greater or equal to 3, therefore the actual help is smaller since length is still unknown.

Table 1: Constraints that can be added to a standalone variable X given a solution s, a substring of the solution ∂s and fresh variables Y and Z

Constraint name	**Symbol**	**Equation**	**Parameter**
Length greater than	$\geq$	$\lvert X\rvert \geq \ell$	ℓ where $\ell \leq \lvert s\rvert$
Length less than	$<$	$\lvert X\rvert < \ell$	ℓ where $\ell > \lvert s\rvert$
Length equals	$=$	$\lvert X\rvert = \lvert s\rvert$	None
Prefix of (i.e., starts with)	$\vdash$	$X = \partial s \mathbin{+\!\!+} Y$	$\lvert\partial s\rvert$
Suffix of (i.e., ends with)	$\dashv$	$X = Y \mathbin{+\!\!+} \partial s$	$\lvert\partial s\rvert$
Substring (i.e., contains)	$\perp$	$X = Y \mathbin{+\!\!+} \partial s \mathbin{+\!\!+} Z$	$\lvert\partial s\rvert$

that each constraint provides is very different. For example, the prefix, suffix, and substring constraints imply a "length greater than" constraint since if a variable has to contain a substring ∂s then the length has to be at least $|\partial s|$, that is, $|X| > |\partial s|$.

We can solve this by quantifying how much each constraint "reduces the search space" and adjusting the parameters accordingly. Since the search space is infinite, we consider a guesser that chooses a length from an exponential distribution $\mathrm{Exp}(\lambda)$ and then chooses a random substring of that length, which has C possibilities for each character. The help that a constraint gives is a measure of the reduction of the expected number of guesses when using said constraint or, equivalently, the **increase in probability of guessing correctly**. Specifically, we define the help h^* given by a constraint with symbol $*$ to be:

$$h^* = -\frac{\ln(p^*) - \ln(p)}{\ln(p)} \qquad 1.$$

Where p is the probability of guessing the solution without help and p^* with help.

This scheme is the simplest metric that gives meaningful results for all the listed constraints. The help calculation is done in log space because it maps multiplications in probability calculations to additive changes in help (otherwise revealing a single character to the solver would correspond to 99.9995% help just by itself).

The derivation for the help used by each constraint is not too complicated, but wordy, so it is available in full in Appendix A. In Table 2 we show the results.

One might expect at first that λ and C would cancel out, but this is not the case. In particular, when λ is larger, the guesses are generally lower, so a "length greater than" constraint is more informative than if λ was small (see Figure 2). Similarly, for C, the more possible characters there are, the more informative each character you reveal is.

3.1.2. Applying constraints given a solution

We now need to apply the constraints. To do that, we first need concrete values for C and λ. We can calculate the value of C easily according to the SMT-LIB specification of strings [26],

Table 2: Relationships between constraint parameters and help they provide, given the probability of guessing the string p_s and of guessing the length p_ℓ.

Constraint	**Help, given parameters**	**Parameters, given help**
Length greater than	$h^{\geq} = \frac{\lambda\ell}{\ln(p_s)}$	$\ell = \frac{h^{\geq}\ln(p_s)}{\lambda}$
Length less than	$h^{<} = \frac{\ln(1 - e^{-\lambda\ell})}{\ln(p_s)}$	$\ell = \frac{\ln(1 - e^{h^{<}} p_s)}{-\lambda}$
Length equals	$\frac{\ln(p_\ell)}{\ln(p_s)}$	None
Prefix (starts with)	$h^{\vdash} = \lvert\partial s\rvert \frac{\ln(C) + \lambda}{-\ln(p_s)}$	$\lvert\partial s\rvert = h^{\vdash}\frac{-\ln(p_s)}{\ln(C) + \lambda}$
Suffix (ends with)	$h^{\dashv} = \lvert\partial s\rvert \frac{\ln(C) + \lambda}{-\ln(p_s)}$	$\lvert\partial s\rvert = h^{\dashv}\frac{-\ln(p_s)}{\ln(C) + \lambda}$
Substring (contains)	$h^{\perp} = \frac{\lvert\partial s\rvert \ln(C) - \ln(\lvert s\rvert - \lvert\partial s\rvert + 1)}{-\ln(p_s)}$	Transcendental eq., find by binary search

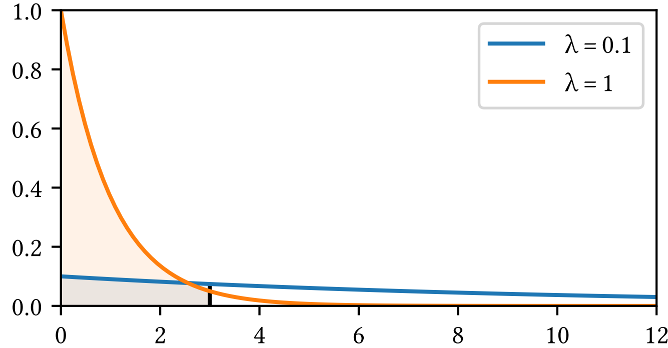

Figure 2: Probability density of $\text{Exp}(\lambda)$ constraint with a smaller (blue) vs bigger (orange) value of λ. Shaded area indicates the search space removed by a "greater than" constraint

which results in $C = 3 \cdot 16^4 = 196\,608$. As for λ, the choice is more arbitrary, but a sensible option is to choose the value that maximizes finding the solution, since the help is relative to that. The value turns out to be $\lambda = \ln\left(\frac{|s|+1}{|s|}\right)$ (per Appendix A.1). This value should provide a representative enough range of lengths the solver is considering, and it definitely provides a representative range of lengths for the specific solution we are hinting at.

The final piece to the puzzle is considering that lengths are discrete, so the constraints cannot give arbitrary help (e.g., if a solution is 3 characters long, you can only give a 1, 2 or 3 character prefix/suffix). We handle this by undershooting the help and adding different constraints until the actual help is at a very small distance ε to the target (in our case,

$\varepsilon = 0.01$) or until we run out of unique constraints (repeated constraints make each other redundant and thus misrepresent the help calculation). This is done in Lines 5, 9, 10 and 12 of Algorithm 1.

```
1  Given help h and n variables with solutions strings s_1, s_2, ..., s_n:
2  │ Split h into n uniformly distributed ranges (h_1, h_2, ..., h_n).
3  │ For each n:
4  │ │ Let r = h_n (help given so far)
5  │ │ While r > ε
6  │ │ │ Let c be a random non-visited constraint (if none exist, break).
7  │ │ │ Let λ = ln(|s_n| / (|s_n| + 1))
8  │ │ │ Calculate constraint parameter p of c (either ℓ or |∂s|) according to Table 2.
9  │ │ │ Calculate help h′ given by ⌊p⌋ according to Table 2 (undershoot help)
10 │ │ │ If h′ > r, continue
11 │ │ │ Add constraint c with parameter ⌊p⌋
12 └ └ └ Subtract h′ from r
```

Algorithm 1: Generation of domain-specific constraints.

As explained in Section 2.3, the *unsat* case is different enough from the *sat* case that we left it outside the scope of this paper. In terms of giving help, we can see the difference in the sense that it is trivial to give 100% help in the *sat* case (just give the solution), while for the *unsat* case you would need some kind of "for all" proof. This case needs to be handled with additional care, so we leave it as possible future work.

3.2. Benchmark runner and reproducibility

We have written a tool, in Rust, that can run all the benchmarks automatically and store them in an SQLite3 database. As illustrated in Figure 3, the runner first collects the problems into the databasem then, it computes their solutions and whether they are *sat* and, finally, we generate the tactics according to Algorithm 1 for each implementation to be benchmarked on every problem with different levels of help (in our case 0 and 0.9), multiple times (as many as time budget allows, in our case between 3 to 5). We only benchmark on problems that both Z3str3 and Z3-Noodler were able to find *sat* solutions for, to prevent unfairness.

We have taken care to achieve the highest standard of reproducibility. Namely, to generate random numbers for Line 2 and Line 6 of Algorithm 1 we used the `wyrand` crate[2], which is a deterministic and portable RNG; which we seed with a hash of the problem's content. We also provide Nix[3] derivations that are deterministic and allow for any user to trivially

[2] https://crates.io/crates/wyrand

[3] https://nixos.org

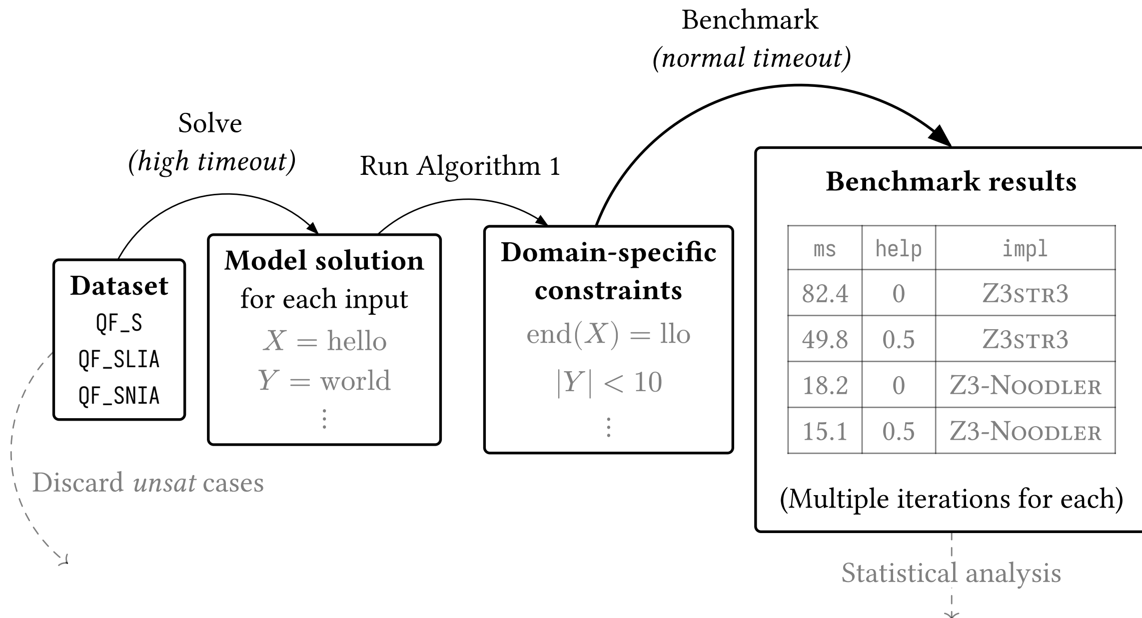

Figure 3: Experiment's pipeline diagram. Gray text indicates example data.

run the programs since, in practice, it is often a hassle to compile and integrate alternative implementations. All in all, any person can easily do the same exact experiments given the source code, which is available on GitHub[4] and an independent forge[5].

The results were ran on a M1 Max CPU with 8GB of RAM. To ensure consistency, apart from running each case multiple times, the laptop was plugged in with no other user programs running.

4. Responsible Research

Before discussing the results, we want to spend a few paragraphs on a somewhat unorthodox reflection (which a busy reader can skip) about making responsible research. For, despite most of the "usual suspects" of unethical investigation are not present in this paper (there is no Machine Learning, no training dataset, no massive use of computing/natural resources, no direct consequences in highly critical infrastructure, etc.) there are considerations we can take about the effect of the results presented in the paper.

4.1. Integrity

As explained in Section 3.2, careful design and using Nix allows for practically trivial, 100% accurate reproduction of the testing infrastructure. That does not imply that the results themselves are 100% reproducible, since the specifics of runtime and number of executed instructions depend on the underlying architecture of the machine. However, any person with access to a Linux or MacOS machine (with either x86 or ARM architecture) can easily run the benchmarking suite and verify the result themselves. Even if the results are not identical,

[4]https://github.com/odilf/smt-guidance-experiments

[5]https://git.odilf.com/odilf/smt-guidance-experiments

we expect them to reach the same conclusions (in fact, finding that some computing platform does not follow the results found in this paper would be highly insightful in of it itself).

Reproducibility is important both for the integrity of the results (we have no room for lying, since anyone can verify them), but also to democratize the advancement knowledge (in an, admittedly, small way). We allow and encourage any interested person to use our runner, or modify it to find new knowledge. We do not require any payment or any special institutional access. The only necessary tools (a computer and an initial internet connection) are by necessity, not by choice.

4.2. The bigger picture

The presented research is relatively self contained, so it is hard to find problem within. However, we can think about the bigger picture. For example, the first line of the paper is an anecdote about how Amazon does one billion SMT queries a day [4]. If we help to improve Z3, we might be indirectly helping Amazon, which is arguably considered widely unethical. So is the presented research unethical too by association?

We would say that the answer is, generally, no — for two reasons. Firstly, the SMT queries at Amazon are very far removed from the specifics of the unethical things they do, so any help we give them in that regard is minuscule. And secondly, as we said before, the code is free, open source, and very easy to use, so any entity in privileged position (such as Amazon) has no more advantage than a university researcher or a hobbyist in their bedroom.

5. Results and discussion

The principal statistic from the experiments is the geometric mean of the speedup of the average runtime to solve a problem with vs without help. We weigh the mean by the original runtime (i.e., without help), which is more representative of performance in practice (since speedups in bigger cases are more relevant). We show both the weighted and unweighted mean in Table 3.

Table 3: Mean speedups of average runtime with vs without help, weighted by the original runtime (without help) and equally.

	Mean speedup (weighted)	Mean speedup (unweighted)
Z3str3	$\mathbf{3.392} \pm 2.43$	$\mathbf{0.879} \pm 2.43$
Z3-Noodler	$\mathbf{0.282} \pm 4.64$	$\mathbf{0.294} \pm 4.64$

The results are clear: Z3-Noodler sees *significantly* less performance improvement than Z3str3 when given domain-specific guidance in the form of search space reduction.

In the rest of the section we explain in more detail the two primary reasons we see these results; namely, the **inherent slowdowns** of Z3-Noodler and the sense in which it is **too good to improve**.

5.1. Inherent slowdowns

There are some cases where the additional constraints that reduce the search space actively harmed performance. This happened more often with Z3-Noodler. We show the runtime differences on Figure 4, where the upper red zone indicates slowdowns. Z3-Noodler, on the right, has a vertical line at around 1ms of original time and, in general, we see Z3-Noodler going a lot further and more consistently into the red (i.e., slowdown) region. This is also reflected in the mean difference in runtime, which is, in fact, positive on average for Z3-Noodler (i.e., problems took longer to run).

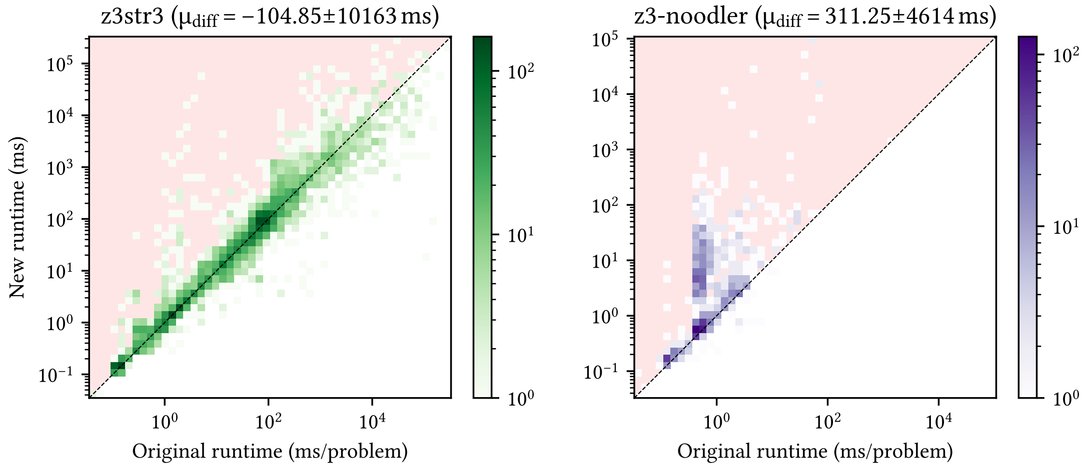

Figure 4: Heatmap comparison of runtime with and without help. Shaded area indicates slowdowns, diagonal line corresponds to no change in performance. μ_{diff} is the mean of the difference of the runtimes.

To highlight one particular example where a constraint has surprising consequences, instance 5942 of dataset “automatark-lu” has solution $X =$ “`000-\ddddd-\ddddd-\ddd↵`”, which Z3 finds instantly. The solution is 23 characters long and clearly contains many “`d`” characters. Logically, adding either a constraint that the length is less than 24 or that the solution contains a “`d`” does not really impact performance. However, if one adds both constraints, Z3 timeouts. That is, adding two reasonable constraints collapses the solver, but adding either independently does not. We have not investigated the underlying cause, but it goes to show that the performance impact of adding constraints can be very unpredictable.

5.2. Too fast to improve

The second reason why Z3-Noodler sees less improvement than Z3str3 is because there is more speedup the bigger a test case is, and, conversely, the additional constraints end up

adding mostly overhead to small, fast cases. The truth of the matter is, then, that Z3-Noodler is so efficient that it rarely reaches high runtimes.

We can see clearly how the speedup increases in Figure 5. There is a lot of noise at the start but there is a clear trend upwards for Z3str3. For Z3-Noodler, all datapoints are clumped closed to zero so we cannot observe any trend. This is unfortunate, but it does reflect what would happen in practice; that is, that Z3-Noodler has less of a chance to get help from one's guidance, since the runs end a lot faster.

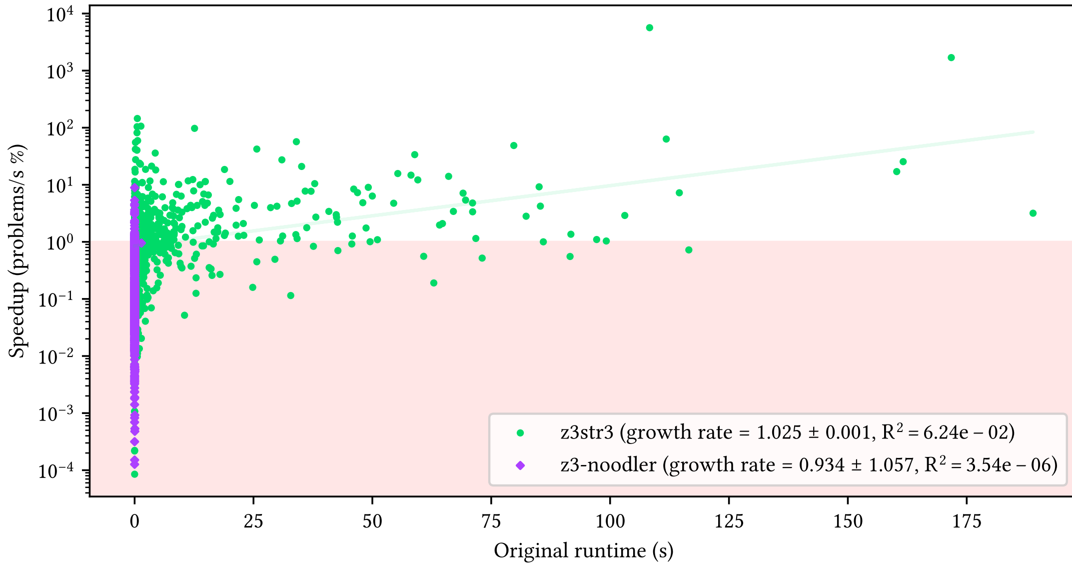

Figure 5: Speedup vs original time for Z3str3 and Z3-Noodler. Shaded area indicates slowdown. (density at the start not accurately represented)

But, to be clear, the overall slowdown is not *only* because Z3-Noodler completes faster. Across basically every range, Z3-Noodler is slowed down significantly more than Z3str3. For instance, between 1-100ms the speedup is $\mathbf{4.396} \pm 2.57$ for Z3str3 vs $\mathbf{0.097} \pm 5.6$ for Z3-Noodler and even in the range 0-1ms, the speedups are $\mathbf{0.691} \pm 1.91$ for Z3str3 vs $\mathbf{0.255} \pm 4.38$ for Z3-Noodler. This happens across almost all ranges, which strongly indicates, again, that Z3-Noodler being fast is not singlehandedly the reason why the speedup is worse, and that seemingly harmless (or even seemingly helpful) constraints genuinely do more relative harm to performance in Z3-Noodler than Z3str3.

6. Conclusions

Answering the titular question: yes, there do seem to be diminishing returns for domain specific guidance as the implementation strength increases, at least in the case of Z3-Noodler as compared to Z3str3. A simple "restrict the search space" approach has inconsistent results: it does speed up some cases, but it slows down others. Small, fast cases are slowed more often

than not by additional constraints. As the problems get more complex, the average speedup tends to increase. And, while for Z3str3 there is a point where there are consistent significant speedups, for Z3-Noodler almost all ranges on average are slowed down.

In other words, Z3-Noodler demonstrates that reducing the search space, although arguably being conventional wisdom and a sound strategy for simple constraint propagation, is not necessarily helpful. Extrapolating this point, we can take away that, even in the simpler cases, one needs to have deep knowledge of how the solver works to be confident that domain-specific help actually improves performance.

We obtained these results using a procedure, introduced in this paper, to quantify the help given by constraints on strings, and how to apply them given a solution. This way we were able to benchmark the performance of Z3str3 and Z3-Noodler with and without help on thousands of cases. The implementations were packaged using Nix to aid in reproducibility, and we provide a Rust tool to run the benchmarks. We strove to make it relatively easy to modify, in order to help future researchers doing a similar investigations.

6.1. Limitations and future work

In this regard, there are many potential developments for the presented research, of which we highlight the most important ones in the rest of this section.

Firstly, one clear limitation was time needed to run larger scale tests. However, at a fundamental level, since equivalent problems take a lot longer to run with Z3str3 than Z3-Noodler, to compare problems that take a long time for Z3-Noodler means that they will take a *very* long time for Z3str3, so it is hard to avoid this pitfall.

Another limitation is the lack of possibility to communicate “soft constraints” to the solver; that is, constraints that the solver is free to use or not use. If this mechanism were implemented then solvers might be smart enough to ignore the constraints if they would add too much overhead, and we would hope to see less slowdowns overall, especially for Z3-Noodler. This should likely be prototyped in standalone solvers, since it is a big ask to put it on the SMT-LIB standard, but it has potential to be useful.

Of course, one could carry the same experiments with different SMT theories and solver implementations. Depending on what theories and solvers are used, our implementation can be reused to different extents. For instance, comparing two Z3 implementations for a different theory can be done by just changing the way additional bounds are generated (which is fairly easy) and packaging the different implementations. One particularly interesting project for potential future benchmarking is Z3alpha [21], which finds good tactics using Monte Carlo methods, so it would be interesting to see how it compares to tailor-made tactics for specific problems. However, that would require a fundamentally different methodology from the one shown in this paper.

Finally, as explained in Section 3.1, the *unsat* case is theoretically tougher than the *sat* case; but the hope is that, in practice, most cases of *unsat* problems are relatively simple and there should be a procedure somewhat similar to Algorithm 1 (but probably not identical) that provides meaningful results. It might even be worthwhile to apply the exact same approach blindly to see what happens, but finding the theoretical justification would be best.

Bibliography

[1] R. M. Karp, “Reducibility among Combinatorial Problems,” *Complexity of Computer Computations*. Springer US, Boston, MA, pp. 85–103, 1972. doi: 10.1007/978-1-4684-2001-2_9.

[2] C. Barrett, A. Stump, C. Tinelli, and others, “The Smt-Lib Standard: Version 2.0,” in *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, UK)*, 2010, p. 14.

[3] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24.

[4] N. Rungta, “A Billion SMT Queries a Day (Invited Paper),” *Computer Aided Verification*, vol. 13371. Springer International Publishing, Cham, pp. 3–18, 2022. doi: 10.1007/978-3-031-13185-1_1.

[5] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A Symbolic Execution Framework for JavaScript,” in *2010 IEEE Symposium on Security and Privacy*, Oakland, CA, USA: IEEE, 2010, pp. 513–528. doi: 10.1109/SP.2010.38.

[6] M. Emmi, R. Majumdar, and K. Sen, “Dynamic Test Input Generation for Database Applications,” in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London United Kingdom: ACM, Jul. 2007, pp. 151–162. doi: 10.1145/1273463.1273484.

[7] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: A Solver for String Constraints,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, Chicago IL USA: ACM, Jul. 2009, pp. 105–116. doi: 10.1145/1572272.1572286.

[8] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg Russia: ACM, Aug. 2013, pp. 488–498. doi: 10.1145/2491411.2491447.

[9] J. Backes *et al.*, “Semantic-Based Automated Reasoning for AWS Access Policies Using SMT,” in *2018 Formal Methods in Computer Aided Design (FMCAD)*, Austin, TX: IEEE, Oct. 2018, pp. 1–9. doi: 10.23919/FMCAD.2018.8602994.

[10] G. Wassermann and Z. Su, “Sound and Precise Analysis of Web Applications for Injection Vulnerabilities,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego California USA: ACM, Jun. 2007, pp. 32–41. doi: 10.1145/1250734.1250739.

[11] G. Redelinghuys, W. Visser, and J. Geldenhuys, “Symbolic Execution of Programs with Strings,” in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, Pretoria South Africa: ACM, Oct. 2012, pp. 139–148. doi: 10.1145/2389836.2389853.

[12] L. D’Antoni and M. Veanes, “The Power of Symbolic Automata and Transducers,” *Computer Aided Verification*, vol. 10426. Springer International Publishing, Cham, pp. 47–67, 2017. doi: 10.1007/978-3-319-63387-9_3.

[13] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, “Z3-Noodler: An Automata-based String Solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 14570. Springer Nature Switzerland, Cham, pp. 24–33, 2024. doi: 10.1007/978-3-031-57246-3_2.

[14] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “DPLL(T): Fast Decision Procedures,” *Computer Aided Verification*, vol. 3114. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 175–188, 2004. doi: 10.1007/978-3-540-27813-9_14.

[15] M. Berzish, V. Ganesh, and Y. Zheng, “Z3str3: A String Solver with Theory-aware Heuristics,” in *2017 Formal Methods in Computer Aided Design (FMCAD)*, Vienna: IEEE, Oct. 2017, pp. 55–59. doi: 10.23919/FMCAD.2017.8102241.

[16] V. Havlena, L. Holík, O. Lengál, and J. Síč, “Cooking String-Integer Conversions with Noodles,” *LIPIcs, Volume 305, SAT 2024*, vol. 305, pp. 1–19, 2024, doi: 10.4230/LIPICS.SAT.2024.14.

[17] “Results | SMT-COMP 2024.” Accessed: Jun. 22, 2025. [Online]. Available: https://smt-comp.github.io/2024/results/qf_strings-single-query/

[18] T. Gur, “Complexity Theory Lecture 8: coNP,” [Online]. Available: https://www.cl.cam.ac.uk/teaching/2324/Complexity/lecture8.pdf

[19] C. M. Wintersteiger, Y. Hamadi, and L. De Moura, “A Concurrent Portfolio Approach to SMT Solving,” *Computer Aided Verification*, vol. 5643. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 715–720, 2009. doi: 10.1007/978-3-642-02658-4_60.

[20] H. Barbosa *et al.*, “Cvc5: A Versatile and Industrial-Strength SMT Solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 13243. Springer International Publishing, Cham, pp. 415–442, 2022. doi: 10.1007/978-3-030-99524-9_24.

[21] Z. Lu, S. Siemer, P. Jha, J. Day, F. Manea, and V. Ganesh, “Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis.” Accessed: Jun. 02, 2025. [Online]. Available: https://arxiv.org/abs/2401.17159

[22] X. Zhang, B. Li, and S. Cai, “Deep Combination of CDCL(T) and Local Search for Satisfiability Modulo Non-Linear Integer Arithmetic Theory,” in *Proceedings of the IEEE/*

ACM 46th International Conference on Software Engineering, Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. doi: 10.1145/3597503.3639105.

[23] X. Liu, H. Liu, X. Yi, and J. Wang, "LLM-Enhanced Theorem Proving with Term Explanation and Tactic Parameter Repair?," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, Macau China: ACM, Jul. 2024, pp. 21–30. doi: 10.1145/3671016.3674823.

[24] T. Kœhler, A. Goens, S. Bhat, T. Grosser, P. Trinder, and M. Steuwer, "Guided Equality Saturation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 1727–1758, Jan. 2024, doi: 10.1145/3632900.

[25] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, *SMT-LIB Release 2024 (Non-Incremental Benchmarks)*. (Apr. 24, 2024). Zenodo. doi: 10.5281/ZENODO.11061097.

[26] "SMT-LIB The Satisfiability Modulo Theories Library." Accessed: Jun. 17, 2025. [Online]. Available: https://smt-lib.org/theories-UnicodeStrings.shtml

Appendix

A. Derivations of help quantification

To carry out the mathematical derivation of exactly how much help each constraint needs (as shown in Table 2), we first define the guesser with more rigor. Namely, we say that the length ℓ of the guessed string follows an exponential distribution with parameter λ such that $\ell \sim \mathrm{Exp}(\lambda)$ with probability density function $f(x) = \lambda e^{-\lambda x}$. The chance of, given a solution s, guessing the correct string length is

$$\begin{aligned} p_\ell &= P(|s| \leq L < |s| + 1) \\ &= \int_{|s|}^{|s|\ +1} f(x)dx \\ &= e^{-\lambda|s|} - e^{-\lambda(|s|\ +1)} \\ &= e^{-\lambda|s|}\left(1 - e^{-\lambda}\right) \end{aligned} \quad (2.)$$

Note that this is equivalent to a geometric distribution $\ell \sim \mathrm{Geom}(p)$ with $p = 1 - e^{-\lambda}$.

and

$$\begin{aligned} \ln(p_\ell) &= \ln\left(e^{-\lambda|s|}\left(1 - e^{-\lambda}\right)\right) \\ &= \ln\left(e^{-\lambda|s|}\right) + \ln\left(\left(1 - e^{-\lambda}\right)\right) \\ &= -\lambda|s| + \ln\left(1 - e^{-\lambda}\right) \end{aligned} \quad (3.)$$

The probability of guessing the correct characters given the length is just $p_c = C^{-|s|}$ so $\ln(p_c) = -|s|\ln(C)$, where C is the number of distinct characters you can choose. The probability of guessing a solution s is then $p_s = p_c \cdot p_\ell$ and so

$$\begin{aligned} \ln(p_s) &= \ln(p_c) + \ln(p_\ell) \\ &= -|s|\ln(C) - \lambda|s| + \ln\left(1 - e^{-\lambda}\right) \end{aligned} \quad (4.)$$

.

Help is defined as

$$\begin{aligned} h^* &= -\frac{\ln\left(\frac{p^*}{p}\right)}{\ln(p)} \\ &= \frac{\ln(p^*) - \ln(p)}{-\ln(p)} \end{aligned} \quad (5.)$$

A.1. Value for λ that maximizes p_ℓ

This value is used to determine the specific value of λ when running the benchmarks. To calculate it, we first need to find the local extrema by $\frac{dp_\ell}{d\lambda} = 0$:

$$\begin{aligned}
&\frac{d}{d\lambda}\left(e^{-\lambda|s|} - e^{-\lambda(|s|\,+1)}\right) \\
&\quad = -|s|\ e^{-\lambda|s|} - \left(-(|s|+1)e^{-\lambda(|s|\,+1)}\right) \\
&\quad = (|s|+1)e^{-\lambda(|s|\,+1)} - |s|\ e^{-\lambda|s|} = 0 \\
&\quad \Rightarrow e^{-\lambda|s|}\left((|s|+1)e^{-\lambda} - |s|\right) = 0 \\
&\quad \Rightarrow \begin{cases} e^{-\lambda|s|} = 0 \Rightarrow \lambda = \infty \\ \quad (|s|+1)e^{-\lambda} - |s| = 0 \\ \Rightarrow e^{-\lambda} = \frac{|s|}{|s|\,+1} \\ \Rightarrow -\lambda = \ln|s| - \ln(|s|+1) \\ \Rightarrow \lambda = \ln(|s|+1) - \ln|s| \end{cases}
\end{aligned} \tag{6.}$$

The point at $\lambda = \infty$ is 0, so it is a minimum. The maximum, i.e., the value of λ that maximizes the probability of guessing a length $|s|$ is $\lambda = \ln\left(\frac{|s|\,+1}{|s|}\right)$

A.2. Length constraints

A.2.1. Length greater than

p_c for length greater than, $p_c^{\geq}$, stays the same (i.e., $p_c^{\geq} = p_c$). $p_\ell^{\geq}$ is different. Namely, the probability distribution $f^{\geq(x)}$ should be 0 before ℓ, since now the string cannot be smaller than ℓ. Then,

$$f^{\geq(x)} = \begin{cases} 0 & \text{if } x < \ell \\ cf(x) & \text{if } x \geq \ell \end{cases} \tag{7.}$$

for some normalization constant c. We can find c by normalizing $\int_0^\infty f^{\geq(x)} dx = 1$

$$\begin{aligned}
\int_0^\infty f^{\geq(x)} dx &= \int_0^\ell 0 dx + c\int_\ell^\infty \lambda e^{-\lambda x} dx \\
&= c\left(-e^{-\lambda\infty} - \left(-e^{-\lambda\ell}\right)\right) \\
&= ce^{-\lambda\ell} = 1 \\
&\Rightarrow c = e^{\lambda\ell}
\end{aligned} \tag{8.}$$

which gives $c = e^{\lambda\ell}$. Therefore, the probability is of guessing the length is now:

$$p_\ell^{\geq} = \int_{|s|}^{|s|\,+1} f^{\geq(x)} dx = e^{\lambda\ell}\int_{|s|}^{|s|\,+1} f(x) dx = p_\ell \cdot e^{\lambda\ell} \tag{9.}$$

And the log of the probability is:

$$\ln\left(p_\ell^{\geq}\right) = \ln(p_\ell) + \lambda\ell \tag{10.}$$

And since $p_c^{\geq} = p_c$, then the overall form is

$$\ln(p_s^{\geq}) = \ln(p_s) + \lambda\ell \tag{11.}$$

So the help is:

$$\begin{aligned} h^{\geq} &= 1 - \frac{\ln(p_s) + \lambda\ell}{\ln(p_s)} \\ &= 1 - 1 - \frac{\lambda\ell}{\ln(p_s)} \\ &= \frac{\lambda\ell}{\ln(p_s)} \end{aligned} \tag{12.}$$

The value of $\ln(p_s)$ can be calculated as shown in Equation 4 does not depend on ℓ (nor $h^{\geq}$).

And, solving for ℓ we get $\ell = \frac{h^{\geq}\ln(p_s)}{\lambda}$

A.2.2. Length less than

This constraint uses a similar logic to "length greater than". Namely, we make the probability density 0 at points after ℓ, so

$$f^{<}(x) = \begin{cases} 0 & \text{if } x \geq \ell \\ cf(x) & \text{if } x < \ell \end{cases} \tag{13.}$$

Carrying out the normalization integral $\int_0^\infty f^{<}(x)dx = 1$, we get that

$$\begin{aligned} \int_0^\infty f^{<}(x)dx &= c\int_0^\ell \lambda e^{-\lambda x}dx + \int_\ell^\infty 0dx \\ &= c\left(-e^{-\lambda\ell} - \left(-e^{-\lambda 0}\right)\right) \\ &= c\left(1 - e^{-\lambda\ell}\right) = 1 \\ &\Rightarrow c = \frac{1}{1 - e^{\lambda\ell}} \end{aligned} \tag{14.}$$

$$p_\ell^{<} = \int_{|s|}^{|s|+1} f^{<}(x)dx = \frac{1}{1 - e^{-\lambda\ell}}\int_{|s|}^{|s|+1} f(x)dx = \frac{p_\ell}{1 - e^{-\lambda\ell}} \tag{15.}$$

And

$$\ln(p_s^{<}) = \ln(p_s) - \ln\left(1 - e^{-\lambda\ell}\right) \tag{16.}$$

So the help is

$$h^{<} = \frac{\ln(p_s) - \ln\left(1 - e^{-\lambda\ell}\right) - \ln(p_s)}{-\ln(p_s)} = \frac{\ln\left(1 - e^{-\lambda\ell}\right)}{\ln(p_s)} \qquad 17.$$

Solving for ℓ here is a bit more cumbersome, but not too complicated.

$$\begin{aligned} h^{<} &= \frac{\ln\left(1 - e^{-\lambda\ell}\right)}{\ln(p_s)} \\ h^{<} \ln(p_s) &= \ln\left(1 - e^{-\lambda\ell}\right) \\ e^{h^{<}} p_s &= 1 - e^{-\lambda\ell} \\ e^{-\lambda\ell} &= 1 - e^{h^{<}} p_s \\ -\lambda\ell &= \ln\left(1 - e^{h^{<}} p_s\right) \\ \ell &= \frac{\ln\left(1 - e^{h^{<}} p_s\right)}{-\lambda} \end{aligned} \qquad 18.$$

A.2.3. Length equals

This constraint does not have a parameter. Since we know the length, $p_\ell^{=} = 1$ so $p_s^{=} = p_c = \frac{p_s}{p_\ell}$ so

$$h^{=} = \frac{\ln(p_s) - \ln(p_\ell) - \ln(p_s)}{\ln(p_s)} = \frac{\ln(p_\ell)}{\ln(p_s)} \qquad 19.$$

A.3. Substring constraints

A.3.1. Prefix of and suffix of

These two are identical in terms of help calculation, the only thing that changes is whether you choose the first characters or the last ones. We use x^* as a way of saying any quantity x for either the “prefix of” or “suffix of” constraint (i.e., h^*, p_s^*, etc.).

Given a substring ∂s, there is an implied constraint that $|X| \geq |\partial s|$, so $p_\ell^{\vdash} = p_\ell^{\dashv} = p_\ell^{\geq}$.

In this constraints, p_c^* does now change. Namely, without the constraint we have $p_c = C^{-|s|}$ (a $\frac{1}{C}$ chance of guessing each character in $|s|$). And $p_c^* = C^{|\partial s| - |s|}$, since we have $|\partial s|$ less characters to guess. The log of p_c^* is

$$\ln(p_c^*) = (|\partial s| - |s|) \ln(C) = \ln(p_c) + |\partial s| \ln(C) \qquad 20.$$

and the help then is:

$$
\begin{aligned}
h^{\vdash} = h^{\dashv} = h^{*} &= 1 - \frac{\ln(p_s^*)}{\ln(p_s)} \\
&= 1 - \frac{\ln(p_s) + |\partial s| \ln(C) + \lambda |\partial s|}{\ln(p_s)} \\
&= 1 - 1 - \frac{|\partial s| \ln(C)}{\ln(p_s)} \\
&= |\partial s| \; \frac{\ln(C) + \lambda}{-\ln(p_s)}
\end{aligned}
\qquad 21.
$$

And, converselty, $|\partial s| = h^* \frac{-\ln(p_s)}{\ln(C)+\lambda}$.

A.3.2. Substring

Contains is similar except that $p_c^{\perp}$ is not $C^{|\partial s| - |s|}$ since we remove $|\partial s|$ characters, but we have to guess where the substring is placed. Therefore, we need to divide by the amount of slots left, which is $|s| - |\partial s| + 1$ (adding one because we need to include both edges). So, $p_c^{\perp} = \frac{C^{|\partial s| - |s|}}{|s| - |\partial s| + 1}$. We have to be careful since $|s| - |\partial s| + 1$ can be greater than $C^{|\partial s|}$ (e.g., $C = 2$, $|s| = 1000$, $|\partial s| = 2$, we get $C^{|\partial s|} = 2^2 = 4$ and $|s| - |\partial s| + 1 = 999$), which means the probability of guessing would *decrease*. A solver should ignore this help if given, so we have to clamp the value to be at least 0. Therefore, $p_c^{\perp} = \max\left(\frac{C^{|\partial s| - |s|}}{|s| - |\partial s| + 1}, 0\right)$

We can calculate the log of $p_s^{\perp}$ by using the fact that, for nonnegative x and y, $\ln(\max(x, y)) = \max(\ln(x), \ln(y))$.

$$
\ln(p_s^{\perp}) = \max(\ln(p_s) + |\partial s| \ln(C) - \ln(|s| - |\partial s| + 1), 0) \qquad 22.
$$

so the help is

$$
h^{\perp} = \frac{|\partial s| \ln(C) - \ln(|s| - |\partial s| + 1)}{-\ln(p_s)} \qquad 23.
$$

This is a transcendental equation that cannot be rearranged to solve for $|\partial s|$, but the value can be found for any given instance using binary search, since this function is monotonically increasing with respect to $|\partial s|$. To spell it out, you can set a lower bound to 0, upper bound to $|s|$, calculate the help given by the average of the lower and upper bounds, and update the bounds accordingly: if the help is greater, set the upper bound to the current guess, otherwise set the lower bound to the current.

B. Use of LLMs

This bachelor thesis is human-made. Large language models (in particular, Claude Sonnet 4) were occasionally used in this project for writing the code to make plots. A few of the queries used where:

- "I am generating this scatter plot in a log-log scale:

  ```
  a, b = compare_col("z3", 0.0, 0.9, "runtime")
  plt.scatter(a, b / a)
  ```

Can you help me convert it to a heatmap (making the squares look equal in the log-log scale)?"

- "I'm getting ValueError: data type <class 'numpy.object_'> not inexact when doing a t-test on some data I have in a polars dataframe. What is wrong?"
- "I have some bins with data and I want to plot the mean and asymmetric error bars. How can I do that in Python?"
- "How can I show a shaded triangle on a heatmap I'm drawing using plt.colorbar(im, ax=ax)?"

A few other queries were used for miscellaneous tasks, such as:

- "What indexes do i need to add to sqlite to make this query run faster? select count(1) from problem join solution as s1 on s1.problem_id = problem.id join solution as s2 on s2.problem_id = problem.id where s1.implementation = "z3str3" and s1.sat = 'sat' and s2.implementation = "z3-noodler" and s2.sat = 'sat'?"
- "I'm getting ValueError: data type <class 'numpy.object_'> not inexact when doing a t-test on some data I have in a polars dataframe. What is wrong?"
- "How can I do Rust escape_unicode on a Cow<str> and not mutate the string if no escaping is necessary?"

Generally, these were used as a starting point, and all results were checked/improved by a human (i.e., me).