Adaptive Restarts for Stochastic Synthesis

Jason R. Koenig
Stanford University
Stanford, CA, USA
jrkoenig@stanford.edu

Oded Padon
Stanford University
Stanford, CA, USA
VMware Research
Palo Alto, CA, USA
oded.padon@gmail.com

Alex Aiken
Stanford University
Stanford, CA, USA
aiken@cs.stanford.edu

Abstract

We consider the problem of program synthesis from input-output examples via stochastic search. We identify a robust feature of stochastic synthesis: The search often progresses through a series of discrete *plateaus*. We observe that the distribution of synthesis times is often heavy-tailed and analyze how these distributions arise. Based on these insights, we present an algorithm that speeds up synthesis by an order of magnitude over the naive algorithm currently used in practice. Our experimental results are obtained in part using a new program synthesis benchmark for superoptimization distilled from widely used production code.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Keywords: stochastic synthesis, restart strategies, superoptimization

ACM Reference Format:
Jason R. Koenig, Oded Padon, and Alex Aiken. 2021. Adaptive Restarts for Stochastic Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454071

1 Introduction

The problem of program synthesis by example is simple to state but difficult to solve generally. Given a set of *(input, output)* pairs, the *test cases*, the task is to synthesize a program that produces the specified output for each input. We are interested in *stochastic synthesis*, which conducts an explicit randomized search over programs. In each iteration of the search loop the current search state is a concrete program p, and the next state p' is chosen by making random changes to p in the hope of finding a p' that is closer to correct on the test cases, as measured by a decrease in a *cost function*. The program is repeatedly modified until it produces the correct output for every given input. Stochastic search is a popular synthesis technique because it is general, easy to implement, and is complementary to other synthesis approaches, working well in some situations where other techniques do not [1, 8, 12, 14, 18, 19]. This paper focuses on the foundations of stochastic synthesis: its characteristic behaviors, why those behaviors arise, and how to exploit these observations to design more efficient synthesis algorithms. We first discuss the behavior of stochastic synthesis (Section 4), and while these observations motivate our new algorithm (Section 5), they are not required to understand it.

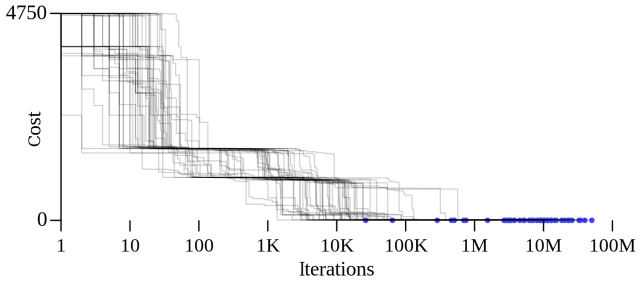


Figure 1. Plateau chart for an example drawn from our benchmark, showing the cost plotted against the logarithm of the number of iterations for a number of independent synthesis runs. Darker lines indicate more searches have those cost(s) for the relevant span of iterations. Dots indicate the successful end of a synthesis run.

Stochastic search is a Markovian process where the probability to transition between states depends on the test cases, distribution of random changes, cost function, and a parameter β that controls the probability of allowing a cost increase (Section 3). When we plot the cost over the course of many independent runs of the same synthesis problem in Figure 1, we see that the searches are dominated by *plateaus*, which we define as periods of a search that fluctuate around a fixed cost. We use a simplified model of stochastic search with a much smaller set of states to study plateaus, which allows us to visualize the search space (Section 4). We explain how the number and connectivity of the plateaus can give rise to different heavy-tailed distributions of search times, including *gamma* and *log-normal* distributions, as well as geometric distributions that are not heavy tailed. We explain how the

PLDI '21, June 20–25, 2021, Virtual, Canada

shape of the search space changes with β: a larger β increases the connectivity of the space and brings the search closer to a pure random walk, while a smaller β makes the search behave increasingly like greedy hill climbing. In a pure random walk, the cost function is ignored: all programs effectively have the same cost and the search time will be geometrically distributed. As β becomes smaller, the search increasingly prefers to maintain or decrease the cost.

For our purposes, a *heavy-tailed distribution* is one in which the mean value is much greater than the median. Thus, for a synthesis problem with a heavy-tailed distribution of search times, while some runs of synthesis may succeed relatively quickly, it is also possible for the search to "get lost" and take orders of magnitude longer. A natural approach to dealing with heavy-tailed searches is to restart from the beginning if a search takes longer than a certain cutoff, which can be fixed or vary according to some strategy (Section 5). We discuss existing strategies for varying when to restart from other domains including SAT solvers (Section 5.1). While restarting has a benefit if the distribution of search times is heavy-tailed, our experiments also show that when the distribution of search times is not heavy-tailed (e.g., is a geometric distribution), classical restart algorithms can actually harm performance.

Classical restart strategies assume the search is a black-box, i.e. no information about the state of the search is available other than whether it has finished. We develop an even more effective *adaptive restart* algorithm (Section 5.2) by relaxing this black-box assumption. This algorithm is inspired by the existing Luby restart strategy [15] and prioritizes search runs that have low cost, using the cost function as a proxy for identifying runs expected to finish quickly. Our experiments show this algorithm performs well for problems both with and without heavy-tailed distributions of synthesis times and that it significantly outperforms the classic Luby algorithm.

To evaluate these algorithms (Section 7), we present experiments using two separate benchmarks: the SyGuS Competition 2017 Programming by Example Bitvector problems and a new superoptimization benchmark collected from straight-line program fragments from the executables of a common Linux distribution (Section 6). We show the the search algorithms perform their best with different values for β, with the restart algorithms using generally lower β. Our adaptive algorithm synthesizes programs from 1.7x to more than 10x faster than the naive algorithm depending on the cost function and benchmark, and up to 5.5x faster than the Luby restart strategy. Some programs synthesized too rarely to reliably compute the expected time, but overall the adaptive algorithm synthesized 97% of the benchmark problems at least once in 50 trials. We give an analysis of why the remaining 3% of synthesis problems remain unsolved.

This work makes the following contributions:

1. We observe stochastic synthesis progresses via a series of *plateaus*. We analyze how plateaus arise with geometric and, when β is low, heavy-tailed log-normal distributions of search times.
2. We present new superoptimization synthesis problems collected from straight-line program fragments from the executables of a common Linux distribution.
3. We present a new algorithm that exploits both the wide variance of synthesis times and the extra information given by the cost function by focusing on the most promising searches. We show that this algorithm outperforms existing approaches on both benchmarks and across the range of possible values for β.

2 Related Work

Program synthesis is an old problem, dating back at least to the work of Waldinger [25] and Green [9]. Current work in program synthesis tends to fall into one of four categories:

1. Some approaches systematically enumerate programs until a program with the desired functionality is found [4, 16].
2. SMT-based methods express the synthesis problem as a system of constraints [23].
3. Stochastic synthesis was introduced in the STOKE project [19].
4. Some approaches use machine learning techniques (e.g. neural networks) to predict the structure of a solution program from input/output examples [3, 17].

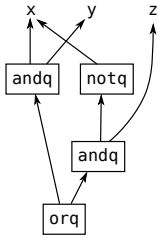
As discussed in Section 1, stochastic search has been used in multiple domains because it is general and straightforward to implement. Besides a variety of applications based directly on STOKE [6, 11, 20–22], stochastic search has been used in the Mimic project [12] and as the core search loop in [18]. Related techniques have been used in finding compiler bugs [14] and in synthesizing program repairs [8]. Alur et. al. [1] compare multiple approaches to program synthesis, including stochastic synthesis. None of these works has investigated the underlying nature of stochastic synthesis, or considered restart strategies for stochastic synthesis.

2.1 Prior Datasets

Most synthesis benchmarks, such as [10] or the evaluation dataset used for Morpheus [7] are relatively small, with 25-80 examples, because they are manually curated. We use the larger set of 600 bit manipulation problems from the SyGuS competition [2]; these are all the problems in the SyGuS benchmark that use input-output pairs and so these are all the SyGuS problems amenable to stochastic synthesis. Our new superoptimization benchmark, discussed in Section 6, is automatically generated, selecting a set of 1000 straight-line code sequences that are representative of hundreds of thousands of code sequences found in a release of the Ubuntu

```
Inputs: x = %rdi, y = %rcx, z = %rax
  andq %rdi, %rcx
  notq %rdi
  andq %rdi, %rax
  orq %rax, %rcx
Output: %rcx
```

(a)



(b)

Figure 2. Example x86 program and equivalent graph

Linux distribution. With a combined total of 1600 benchmark programs, our experiments represent one of the largest synthesis studies to date.

2.2 Restart Strategies

When the distribution of stochastic synthesis times is heavy-tailed, there is value in periodically restarting the search. Because the time to synthesize is a random variable, stochastic synthesis methods are *Las Vegas* algorithms, which have been analyzed in the general case in [15]. Restart strategies are standard in SAT solvers [13]. Solvers usually preserve some information (learned clauses, etc.) between restarts, while we consider restarts as entirely independent searches.

3 Synthesis Algorithm

We present a stochastic synthesis algorithm designed to be representative of existing algorithms while allowing us to consider synthesis problems derived from real code.

3.1 Program Representation

Programs are rooted, directed, acyclic dataflow graphs, where nodes are *instructions*, *constants*, or *inputs* and edges represent the use by the source node of a value produced by the destination node, as in Figure 2b. The root node is the program's result.

Motivated by our desire to represent low-level machine code programs, intermediate values and constants are 64-bit integers, and we define operations like `orq` and `addq` for bitwise OR and integer addition. Each instruction node has an opcode specifying a deterministic operation of fixed arity with no side effects.[1] We do not allow dead code, i.e. all nodes must be connected to the root node. This representation is chosen because it is general, simple to understand, and avoids unnecessarily making the order of operations significant.

An example of a program fragment (here written in x86 assembly) that we might want to synthesize along with its graph equivalent is given in Figure 2, where `x`, `y`, and `z` are the inputs. We can also represent this program as an expression:

```
1  p ← zero_program
2  n ← 0
3  c ← cost_of(p)
4  while n < N ∧ c > 0 do
5      p′ ← propose_change(p)
6      if is_valid(p′) then
7          c′ ← cost_of(p′)
8          if c′ ≤ c − β ln(random(0, 1)) then
9              p, c ← p′, c′
10         n ← n + 1
```

Figure 3. The main loop of stochastic synthesis.

`orq(andq(x, y), andq(notq(x), z))`,[2] which we use as a convenient textual notation.

3.2 Stochastic Synthesis Algorithm

A *synthesis problem* is a set of input-output pairs, or test cases, specifying the behavior of a program we wish to synthesize. For the purposes of this paper, we consider any program that matches the input-output specification to be a solution.

A *stochastic synthesis algorithm* in the style of [19] maintains a current program p, and in each iteration proposes a possible new program p' via one of several kinds of *moves*, evaluates the new *cost* and *accepts* p' if the cost has decreased or only increased by at most a small random margin. Pseudocode is given in Figure 3. If p' is accepted, it becomes the current program, otherwise p is retained as the current program, and the process repeats until the cost is zero. The initial program is the constant zero. The ability to sometimes accept programs that are worse than the current state is needed to escape local minima of the search.

The cost function measures progress towards a solution of a given synthesis problem q. All of the cost functions we consider will be zero exactly when the candidate output equals the testcase output. In addition to the cost function introduced in [19], Hamming, we also consider two additional cost functions for our experiments in Section 7:

1. **Hamming.** Total number of incorrect bits across all test cases, i.e., the Hamming weight of the XOR of the desired results and the candidate output.
2. **Incorrect test cases.** The cost is the number of test cases that are not entirely correct, i.e. that differ in at least one bit from the correct value. This cost function can avoid artifacts caused by the Hamming cost but produces less signal for the synthesis algorithm.
3. **Log-difference.** For each testcase, we interpret the candidate output and desired result as 64-bit signed integers a and b. If they differ, the cost for that testcase

[1] Operations such as division and modulus that would trap at runtime for undefined results instead produce zero.

[2] We can express sharing of intermediate nodes via variables as in `a = notq(x); addq(a, a)`.

is $1+\log_2(|a-b|)$; otherwise the cost is zero. This cost function is most useful when the output is numeric.

Each move makes a small syntactic change to the current program, such as redirecting an edge or altering a node's operation. For the baseline algorithm, there are three moves:

1. **Instruction.** Pick a random argument to an operation, or the slot for the root node. Generate a random opcode, and fill its arguments with random existing nodes (without creating cycles) or random constants. Point the selected argument to the new instruction.
2. **Opcode.** Pick a random instruction node. Replace its opcode with a random opcode of the same arity.
3. **Operand.** Pick a random argument or the root slot. Pick a random node that does not create a cycle and point the argument to that node.

Instruction moves introduce new components and opcode and operand moves re-wire existing parts of the graph. These operations are standard and their analogues appear in every approach to using stochastic search. Each move selects among valid options for each choice (opcodes, operand slots, etc.) with uniform probability.

The distribution of $-\beta \cdot \ln(\text{random}(0,1))$ is exponential and is motivated by MCMC theory [19].[3] The probability of accepting a new program just 3 bits higher in cost has a probability of only about 5% per iteration with $\beta = 1$, and at 6 bits the probability falls to 0.25%. If a new program has the same or a lower cost, then the new program is always accepted. The parameter β controls the permissiveness of the search: a larger β results in accepting more cost increasing moves, while a smaller β brings the search closer to only allowing moves that preserve or decrease the cost. The parameter β is expressed in the units of the cost function (such as bits, number of test cases, numerical difference, etc.) and thus must be tuned separately for each cost function. Because the cost function depends on the number of test cases, we normalize β to that of a synthesis problem with 100 test cases:

$$\beta' = \beta \frac{|\text{test cases}|}{100}$$

Finally, we place a maximum size limit of 16 nodes on programs, rejecting any change that would bring the count above this number. This limit prevents the time to evaluate the program on the test cases, which is the dominant cost in the synthesis algorithm, from growing arbitrarily. In our experiments the search achieves a mean of 339K iterations per second of the loop in Figure 3 on a single CPU core. In our discussions and evaluation we use iteration count as a measure to compare algorithms independently of the speed of the underlying hardware.

[3]The assumptions of ergodicity and reciprocity required to apply the theory are not met by our system.

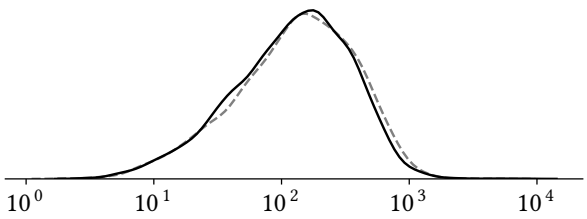


Figure 4. Synthesis times for `or(shl(x),x)`, measured (solid) against predicted (dashed). Note the apparent left skew.

3.3 Markov Representation

For a given synthesis problem, stochastic search is described by a Markov chain where states are programs p_i. The probability of moving to a new state is the product of the chance of proposing that state and the chance of accepting it, which for $p_i \neq p_j$ is:

$$\Pr[p_i \to p_j] = \Pr[p_i \text{ prop. } p_j] \cdot \Pr_{X \sim \text{Exp}(\beta)}[C(p_i) \leq C(p_j) + X]$$

The remaining probability is the chance of a self-loop, i.e. the probability that the search remains in the current state. Note the structure of the Markov chain depends on the move proposal distribution, the cost function, the specific test cases used, and β. Any zero cost state is absorbing because the search finishes when that state is reached.

4 Behavior of Stochastic Synthesis

To investigate the plateau structure observed in Figure 1 and motivate the algorithm we present in Section 5, we first build a simplified model of stochastic synthesis that allows us to visualize the search space. We reduce the set of operations to: `and`, `or`, `xor`, `not`, `shr`, `shl`, `zero`, and `ones`. The bitwise operations behave as expected, while `shr` and `shl` shift by one bit and shift in zero on either end. The constants `zero` and `ones` are all zero and all one bits respectively. We also add a fourth move type that canonicalizes programs, which further reduces the search space:

4. **Redundancy.** Merge a random pair of instruction nodes that have the same output values for a randomly chosen subset of test cases, by redirecting incoming edges from one node to the other.

This set of operations is complex enough to have interesting behavior while being simple enough to analyze fully for the synthesis of the program: `or(shl(x), x)`. We start the search in the `zero` state. We can now identify where the search spends its time because there are not many frequently visited states. We run many synthesis trials and track the 35 most frequently visited states. We build the transition matrix of this Markov chain by sampling the probability of transitioning from one state to another, conditioned on staying within this set of *popular* states. The imprecision of ignoring

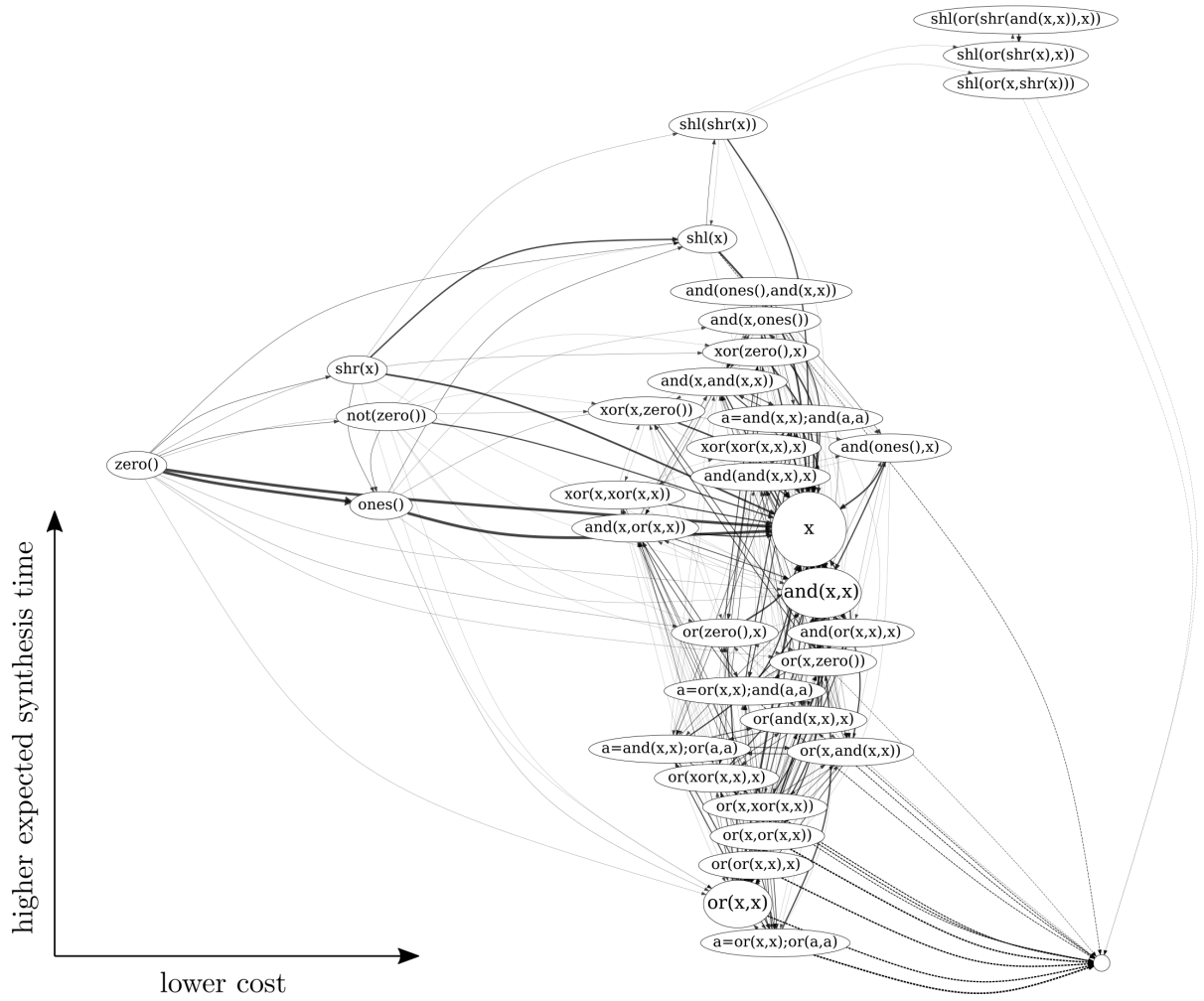


Figure 5. The state transition diagram for `or(shl(x), x)`. Nodes are states, with transitions between them according to weight. The search starts on the leftmost node and ends on the node in the lower right corner.

rarer states is small because their aggregate probability is low, something that is not true for real search spaces.

We sample the traversal time of this Markov chain and compare this distribution to that of the original synthesis problem in Figure 4. We see that the distributions agree closely, and also note that the distribution appears to be skewed to the left when plotted logarithmically.

A diagram of this search space is given in Figure 5. We show a node for each popular state. Edge thickness is proportional to the frequency the edge is traversed; edges leading to the final state are dotted. Nodes are scaled to show their significance and labeled with the programs that they represent. Nodes are positioned to show two important quantities: the cost, decreasing left to right, and the empirically determined expected time for the search to complete from that state, plotted from low on the bottom to high on top. The final node is thus at the lower right corner. These locations are not precise because drawing the diagram to scale would result in the nodes overlapping.

4.1 Plateaus

There are several interesting observations from Figure 5, which we later generalize to realistic stochastic synthesis. First, there is a strongly connected component in the center of the diagram around `x` and the transition probabilities ensure the search spends most of its time in this component. There are many different ways to write the identity program, including `x`, `or(x, x)`, `or(x, zero())`, `and(x, x)`, etc. Due to the redundancy move, these have frequently taken transitions back to the `x` node. All of these highly connected nodes have the same cost. They have similar expected times to reach the final state, but the states that involve an `or` have slightly smaller times than those that lack an `or`, because the `or` state typically has a transition directly to the goal state by replacing one argument with `shl(x)`. The other states must

use two moves: one to introduce `or` and another to produce `shl`. However, this difference is negligible relative to the time to reach the final state from each of the two groups.

The key property is the existence of a strongly connected component of the search space where all the states have equal, or nearly equal, costs; for brevity we call such regions *strong components*. It is easy to see that strong components give rise to plateaus. First, because of the large number of ways to write a functionally equivalent program in different ways, strong components are always present in stochastic synthesis problems. When a strong component has few exits (transitions out with a significant drop in cost) relative to the number of transitions connecting states of the component, then the time to transition among the states of the component will be small compared to the time to leave the component, resulting in a plateau. The cost function provides no guidance in finding an exit from the plateau as the search becomes a random walk among states of nearly equal cost. Because the time to transition between states in the plateau is small relative to the time spent on the whole plateau, we can ignore the internal structure and consider the probability of leaving as a constant p. Thus we can summarize the plateau as one node in the Markov chain with a self loop with probability $1 - p$. Under this approximation the time to leave a plateau is a geometric random variable.

Another observation is that there are nodes in the upper right corner, corresponding to a different strong component with low cost but much higher expected remaining synthesis time than the initial state or the `x` component. These states produce almost the correct output except they have an extra right shift followed by a left shift that unnecessarily loses information in the lowest bit. Intuitively, these states represent a situation where the cost function is unhelpful, because while the output is almost correct substantial changes are required to reach a correct program. These states show one way a search can get stuck: the paths out of these states are rare as indicated by a high expected time to synthesize starting from that state. The low cost relative to the main component means that the search is unlikely to return to the main plateau and must find another path to the final state.

We can now explain the left skewed distribution from Figure 4. For this example, the probability of reaching the upper strong component at all from the start state is very small, so these states do not have a large influence on the expected time to synthesize. The search reaches the central plateau very quickly relative to the time spent there, and so synthesis time is dominated by the time to leave the central plateau. This is a geometric distribution, which is a good fit for the shape of Figure 4 and explains the left skew.

A final key point is that the connectivity of the search space is strongly influenced by the value of β. When β is very large the cost function becomes irrelevant, all states become part of the same strong component, and the search time will therefore be geometrically distributed. To see this, observe that at any point the search can return to the start state in one step by proposing an instruction move with `zero()` for the root slot. In general it is desirable to set β much lower, where the cost function provides guidance and much faster search times.

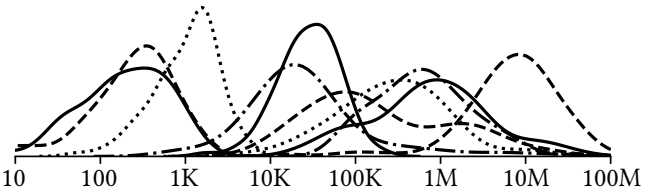


Figure 6. Distributions of finishing times for ten random examples from our benchmark, showing the prevalence of log-normal-like distributions.

4.2 Multiple Plateaus

A Markov chain with one dominant strong component is described by a geometric distribution, but stochastic synthesis problems can have more complicated structure.

Some searches travel along a *path* with more than one plateau. For a path of plateaus in sequence, the distribution of search times is equal to a sum of geometric variables, one for each plateau. If the one of the geometrics dominates, then the overall distribution will still be close to a geometric, but if the geometric variables are all approximately the same the result will be what is known as a *gamma* distribution. When there is only a single path restarts are actually detrimental, as any progress along the single path to the goal is sacrificed and must be redone from the beginning.

We have already seen how a synthesis problem can have more than one path, where each path has very different expected times and probability of occurrence. If the distribution is a mixture of multiple alternate paths through the search space, then it can have virtually any shape. However, we might expect that if many paths have uncorrelated means and variances, the mixture distribution converges to a normal or bell-curve like distribution. When the means of individual paths vary over orders of magnitude, the mixture will converge to a log-normal, as seen in Figure 6. We expect this variance in means because small absolute changes in a typically small probability p of leaving a strong component has a large multiplicative effect on the mean time to leave ($\sim 1/p$).

Intuitively, restarts can help when there are both short and long paths to the goal, as a restart while on a long path has some chance to take a short path instead. Figure 7 gives an example of a plateau chart of a heavy-tailed synthesis problem with multiple paths that is a mixture of a tight distribution and a much higher variance distribution. Note the skew of finishing times to the right: the median run

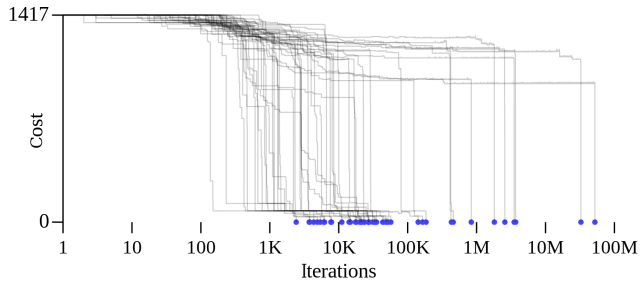


Figure 7. Plateau chart for a benchmark example.

completes orders of magnitude faster than the average time over all synthesis runs.

Just because the search space is complex with multiple paths does not imply that search times will be heavy-tailed. Recall Figure 1, where there are three distinct plateaus. This example is actually best fit by a geometric distribution, because the final plateau dominates the runtime with a much higher time to traverse than the others (extending from 10K to 10M iterations).

5 Improving Stochastic Synthesis with Restart Strategies

As we saw in the previous section, heavy-tailed distributions imply most synthesis runs are much shorter than a tail of searches that take orders of magnitude longer. For heavy-tailed search problems it therefore makes sense to abandon a search and begin a fresh one if we think the current search may be long-running, which is called *restarting*. The baseline *naive* algorithm is one that never restarts (i.e. it runs one search until it completes or times out), and thus may get stuck in a long search. We discuss existing restart strategies and then develop an algorithm that goes beyond classic restart strategies by exploiting the information given by the cost function.

5.1 Restart Strategies

The question for restarts is *when* should the search be restarted? As observed in [15], if no information about the search is available, this question can be answered by a (possibly infinite) sequence of cutoffs called a *restart strategy*. For a particular synthesis problem there is a fixed distribution D of finishing times which depends on the structure of the Markov chain, and the optimal strategy for that distribution is to always use a particular fixed cutoff $t*$, which induces an expected time to complete of T^*.

Calculating the optimal cutoff for a given synthesis problem requires knowing the distribution of synthesis times. Alternatively, we can create a restart strategy that varies the cutoff each time the search is restarted. In [15], a strategy based on the recursively defined Luby sequence $\langle 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \dots \rangle$ is given. Any prefix of this sequence causes the search to spend approximately equal total search time on an exponentially increasing set of cutoffs. This algorithm runs in expected time $O(T^* \ln(T^*))$, and no algorithm can do better for arbitrary distributions. However, given prior information about the class of distributions, it may be possible to design better strategies. The design of such strategies has previously been considered in SAT solving, where the search for a satisfying assignment can become stuck by early decisions. These proposed strategies include exponentially increasing cutoffs $t_0 z^k$ for $k = 0, 1, 2, \dots$, and an inner-outer geometric strategy where $k = 0, 1, 0, 1, 2, 0, 1, 2, 3, \dots$ [5].

5.2 Adaptive Algorithm

Restarts can be used to avoid long searches and consequently speed up stochastic synthesis, but these approaches assume that no information about a search is available except whether it has completed. If we exploit a program's cost as an estimate of how close the search is to succeeding, we can do better than the best restart algorithms. Instead of restarting searches, we run searches in parallel but focus search time on the searches with low cost. Spending some iterations on searches that are not the current best is beneficial because the best cost search may be stuck in a long plateau that other searches might avoid (recall Section 4). As a heuristic for distributing iterations between the searches of varying costs, we use the Luby sequence. We develop the algorithm in two steps: first we show how the Luby restart strategy works and can be modified to run searches in parallel, and then we show how to prioritize searches with low cost.

The Luby sequence can be defined as the limit of the sequence recurrence $L_0 = \langle 1 \rangle$ and $L_i = L_{i-1} \mid\mid L_{i-1} \mid\mid \langle 2^i \rangle$. The full Luby sequence is then the limit L_∞, which is well defined because every L_i is a prefix of L_j for $j > i$. This definition can also be seen as defining a series of trees as shown in Fig 8(a) and (c), for L_2 and L_3 respectively. The sequence is then recovered as the depth-first post-order traversal of such trees. Whenever the root of one tree is reached, the next tree can be constructed by duplicating the tree and attaching both under a new root with double the label of the old root. The classic Luby restart algorithm is in effect traversing this series of trees and running a search for each node with an iteration limit equal to the base limit t_0 times the node's label, stopping if a search finishes and otherwise moving to the next node.

There is another way to construct the same sequence, which runs searches in parallel rather than sequentially. We repeatedly traverse the tree in depth-first post-order, doubling the labels of all existing nodes and adding a new pair of 1-labeled leaves to each existing leaf node. We show a snapshot of this process in Fig 8(b), where the current point is marked with a $*$. Rather than duplicating the tree and adding a new root, we make the old tree the top layers of the new one, with only the leaves as newly added nodes. To

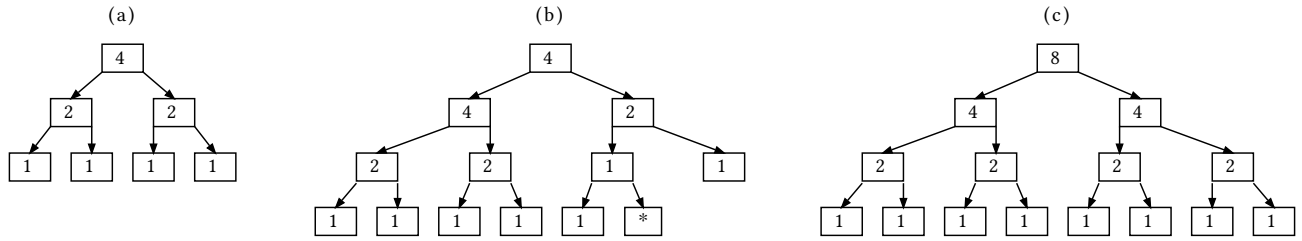


Figure 8. The adaptive search algorithm at the (a) beginning of one doubling, (b) middle of doubling execution, and (c) beginning of next doubling. Node labels are the number of iterations search has run assuming no swaps have occurred.

```
1  n ← ⊥
2  while not successful do
3      n ← double(n, ⊥)
4  Function double(n, p):
5      if n = ⊥ then
6          s ← new search
7          run s for t_0 iterations
8          return
             {search : s, height : 0, left : ⊥, right : ⊥}
9      n.left ← double(n.left, n)
10     n.right ← double(n.right, n)
11     run n.search for t_0 · 2^{n.height} iterations
12     n.height ← n.height + 1
13     if p ≠ ⊥ ∧ cost(n.search) < cost(p.search) then
14         swap n.search ↔ p.search
15     return n
```

Figure 9. Pseudocode for adaptive search algorithm.

turn this into a restart algorithm, we associate a search state with each node, and each time we double a node's label l we run the search for an additional $l \cdot t_0$ iterations. Because the labels are powers of 2, the cumulative runtime is the same as if this node had been visited by the sequential algorithm. Note that unlike the sequential method, we must retain partially executed searches which can increase memory usage. After n doublings, we have run the same number of searches with the same distribution of runtimes as the sequential algorithm, so this parallel algorithm has the same guarantees on expected total runtime.

To construct the *adaptive search algorithm*, we drop the black-box assumption on the search state. In the context of stochastic synthesis, the cost is a reasonable proxy to determine which searches are likely to complete quickly. Nodes closer to the root run more iterations, so it makes sense to ensure that the most promising runs, i.e. the lowest cost ones, are near the root. The adaptive search algorithm is the same as the parallel Luby algorithm except whenever we finish visiting a non-root node, we swap it with its parent

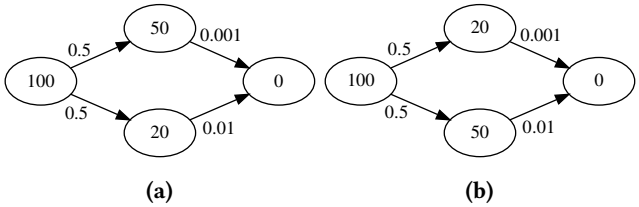


Figure 10. Example of Markov chains for which adaptive search works (a) well and (b) poorly. Nodes are labeled by cost and the non-self loop probabilities are given on each transition edge.

if the parent has a higher cost. Pseudocode for the adaptive search algorithm is given in Figure 9. We traverse the tree in depth first post-order, which means that if a search achieves a sufficiently low cost it may move up multiple levels of the tree in a single doubling. Because of the swapping, the label of a node does not necessarily indicate how many iterations a search has run so far, only how many future iterations it will be allocated. Because this algorithm exploits the costs of individual searches, it can potentially do better than the theoretically optimal Luby restart sequence. We evaluate the adaptive algorithm experimentally in Section 7, but here we give examples of when this algorithm performs well or poorly compared to classic Luby.

5.2.1 Model Markov Chains. Two Markov chains representing models of stochastic searches are given in Fig 10. Here we show the transition probability on each edge, and label the nodes with their costs, with the leftmost node as the initial state. For this model, all transitions except self-loops are depicted, i.e. there is zero probability of the system transitioning from a middle node to the start or between middle nodes. The model is symmetric except for the costs and probability of finishing from each of the middle nodes. In example (a) the costs align with the probabilities: the lower cost state is 10 times more likely to reach the goal state than the high cost state. In (b), the opposite is the case, in that low cost does not predict which state is "closer" to the goal. The situation in (b) is similar to the upper right

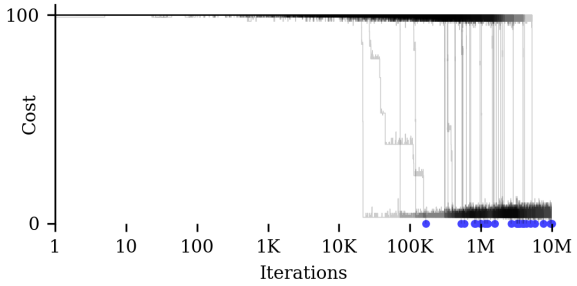


Figure 11. Example plateau charts for an example using the incorrect test cases cost function and $\beta = 1$.

plateau from Fig 5, where the low cost did not predict a short time to synthesize. These differences make all the difference for the adaptive algorithm: while simulations of the classic Luby restart algorithm perform the same on both models, the adaptive algorithm is 31% faster than Luby on (a) and 46% slower on (b). Thus we expect the adaptive algorithm to do well when there is a correlation between the cost of a partial search and its time to finish. If that correlation is reversed, then the adaptive search will spend more time on searches that will take longer. Section 7 shows that the adaptive algorithm is over twice as fast as the classical Luby algorithm on benchmarks, but this speedup is not a limit on the difference in performance.

5.2.2 Effect of β on Search Space. As another example of when adaptive restarts are affected by the structure of the search space, we give the plateau charts for an example from our benchmark using the incorrect test cases cost function and $\beta = 1$ in Figure 11. The relatively high value of β means the search frequently returns to a higher cost, so frequently that the individual moves blur together. A high value of β increases the connectivity of the search space, making the search closer to a random walk, and the overall time to synthesize is dominated by a geometric distribution for the time to leave this initial plateau. Because the geometric distribution is not heavy tailed, restarts are actually detrimental in this example, because the restarts spend more time on the initial plateau at cost 100 instead of the minimum cost plateau that can reach the solution. The naive algorithm, which does not restart, actually outperforms the classic Luby algorithm, as the Luby algorithm spends time on short searches which do not reach the lower plateau before the next restart. Our adaptive restart algorithm has the same performance as the naive algorithm, because it can detect searches on the lower plateau and allocate almost all search time to those searches.

In general, a high β increases the likelihood that the search space is dominated by a geometric distribution, where traditional restart algorithms can incur a penalty but our adaptive restart algorithm is unaffected. Conversely, lowering β can increase the ability of the search to stay in a particular plateau.

```
*addl %r14d, %ebp
 pxor %xmm1, %xmm1
*addl %ebp, %eax
 movsd 0x2f251(%rip), %xmm2
*leal (%rax,%rax,4), %edx
 leal (%r14,%r14,4), %eax
 movsd 0x2f24a(%rip), %xmm0
 shll $0x3, %eax
*shll $0x3, %edx
```

Figure 12. Example showing identification of dataflow-related instructions for output register %edx.

If that plateau represents meaningful progress towards a solution, then this focus is desirable. However, if the plateau is on a very long path, such as the upper-right plateau of Figure 5, then requiring the search to find a lower cost state to exit the plateau can cause the search to take significantly longer. A primary advantage of our adaptive restart strategy is the ability to use restarts with search prioritization to gain the benefits of a low β without paying the cost of the search getting stuck on long paths through the search space.

6 Superoptimization Synthesis Benchmark

Because superoptimization is one of the standard applications for stochastic search, we wanted to test our algorithm on a large set of superoptimization problems drawn from real code in addition to the existing SyGuS benchmark. We constructed a new benchmark of synthesis problems from dataflow-related subsequences of the basic blocks of binaries. "Dataflow-related" means that the instructions selected all influence the result, i.e. we remove instructions that are irrelevant to computing a particular output. For example, the basic block in Figure 12 has the set of dataflow instructions for the %edx register highlighted. We focus on binaries because large amounts of production binary code are readily available, allowing us to automate the construction of a comprehensive benchmark.

The source for our synthesis problems is the x86_64 ELF binaries from the default packages of the Ubuntu 16.04 operating system. For each basic block in each function in the distribution, we consider each register r that is live-out at the end of the block and compute the backwards slice of all instructions that contribute to r's value. Any memory reads included in the slice are replaced by moves from registers that are not otherwise used. If the resulting code fragment had at least two non-trivial instructions, we retained it as a potential synthesis problem.

There are a number limitations to this process of scraping synthesis problems from binaries:

- We do not generate problems with memory operations, which simplifies the calculation of dataflow and is consistent with much of the state of the art of program synthesis at the assembly level.

- Our synthesis problems are all straight line code—no branches or loops—which is again largely consistent with the current state of the art. Excluding control flow makes the automatic generation of high coverage test cases more reliable. We also excluded programs containing the conditional move instruction (`cmov`).
- Our scraping process is imperfect and at each stage potential synthesis problems are lost. For example, some basic blocks contain instructions our disassembler does not support.

6.1 Generating the Standard Benchmark

Following the procedure outlined above for every basic block in the Ubuntu 16.04 distribution produces 187,077 program fragments. Many of these program fragments are closely related, such as variations on $ax + b$. To generate a set of problems with distinct behaviors, we first group problems by their *instruction signature*, i.e. the sequence of instructions ignoring registers and arguments. Additionally we ignore simple data-movement instructions such as `movq` and `movl`. We sampled one example from each instruction signature equivalence class, yielding 9,719 synthesis problems ranging in size from 2 to 15 instructions, with about 74% being 3-6 instructions.

One remaining problem is that some of the programs in this set could be unsynthesizable, meaning that no program in our program syntax could express the functionality (e.g., due to instructions that our reduced language lacks). For each program fragment, we attempted to synthesize each prefix of length n given the solution to synthesizing the prefix of length $n-1$ as a starting point, effectively synthesizing the problem one instruction at a time. This process eliminated the few percent of the dataset that are not likely to be expressible with the instructions we have implemented. We took a random sample of 1000 synthesizable programs as our standard benchmark. This benchmark is available in our artifact.[4]

Finally, we also generated test cases for each synthesis problem. We generated inputs that include important corner cases (0, 1, -1), uniformly random bit patterns, and bit patterns with high and low Hamming weight.

7 Evaluation

The main goal of our evaluation is to compare the performance of the naive, Luby, and adaptive restart algorithms using multiple cost functions on the SyGuS and superoptimization benchmarks. To perform this comparison we first need to (1) ensure the algorithms are compared fairly by finding the best β for each individually, and (2) establish a way to assign a score for an individual problem in the presence of time outs. To address (1), we run many synthesis trials with each cost function and algorithm while varying β, and observe how frequently each combination is successful as a function of β. For this experiment, we are most concerned with comparing each algorithm against itself as β changes. From this data, we select the best β for each combination of algorithm, cost function, and benchmark.

To address (2), given synthesis times from a set of trials where some may have timed out, we define a way to calculate an estimate of the mean time to synthesize by adding a penalty for timed out trials to the sample mean (Section 7.2). With this definition in hand, we perform an experiment by running many trials of each algorithm and cost function on all benchmark problems and compute a corrected mean time for each. We then compare the performance of the algorithms by summarizing this data in several ways, which shows our adaptive restart algorithm is superior. Finally, we discuss qualitatively why some programs failed to synthesize.

7.1 Effect of β

To fairly compare the restart strategies, we need to tune the parameter β for each algorithm. As we discussed in Section 5.2.2, β affects the shape of the search space, and we expect this to have an effect on the performance of the synthesis algorithms. We ran 10 synthesis trials on a randomly selected 10% of each benchmark, with varying β and with each algorithm and cost function. To compare performance, we take the overall fraction of trials (across all synthesis problems for that benchmark) that completed within 100M iterations. The results are plotted in Figure 13.

There are three trends to note in these plots: (1) all algorithms perform similarly for high beta with Luby sometimes showing a penalty for restarting, (2) the naive algorithm usually has a local minimum before getting worse with smaller β, and (3) the adaptive algorithm is almost always the best algorithm. The fact that the algorithms are similar for high β is consistent with the search becoming a random walk, relatively uninformed by the cost function. We can also see that for high β in SyGuS with Hamming and log-difference, the Luby algorithm does slightly worse than naive, which is explained by the observations in Section 5.2.2. Except for SyGuS with incorrect test cases, the naive algorithm has a minimum in each plot before getting worse as β is decreased. This is explained by the search more easily being caught in local minima, which the naive algorithm has no way to escape. The two restart algorithms also exhibit a slight increase as β drops below their optimal point, but this increase is much less pronounced. Finally we note that the adaptive algorithm is almost always the best algorithm. The adaptive restart algorithm is thus a good choice if the value of β cannot be tuned in advance, as would be the case with novel synthesis problems.

From these results, we can tune β by computing the minimum for each algorithm, cost function, and benchmark. We give these results in Table 1, and use these values for the comparisons in the rest of our evaluation.

[4] Available through the ACM Digital Library.

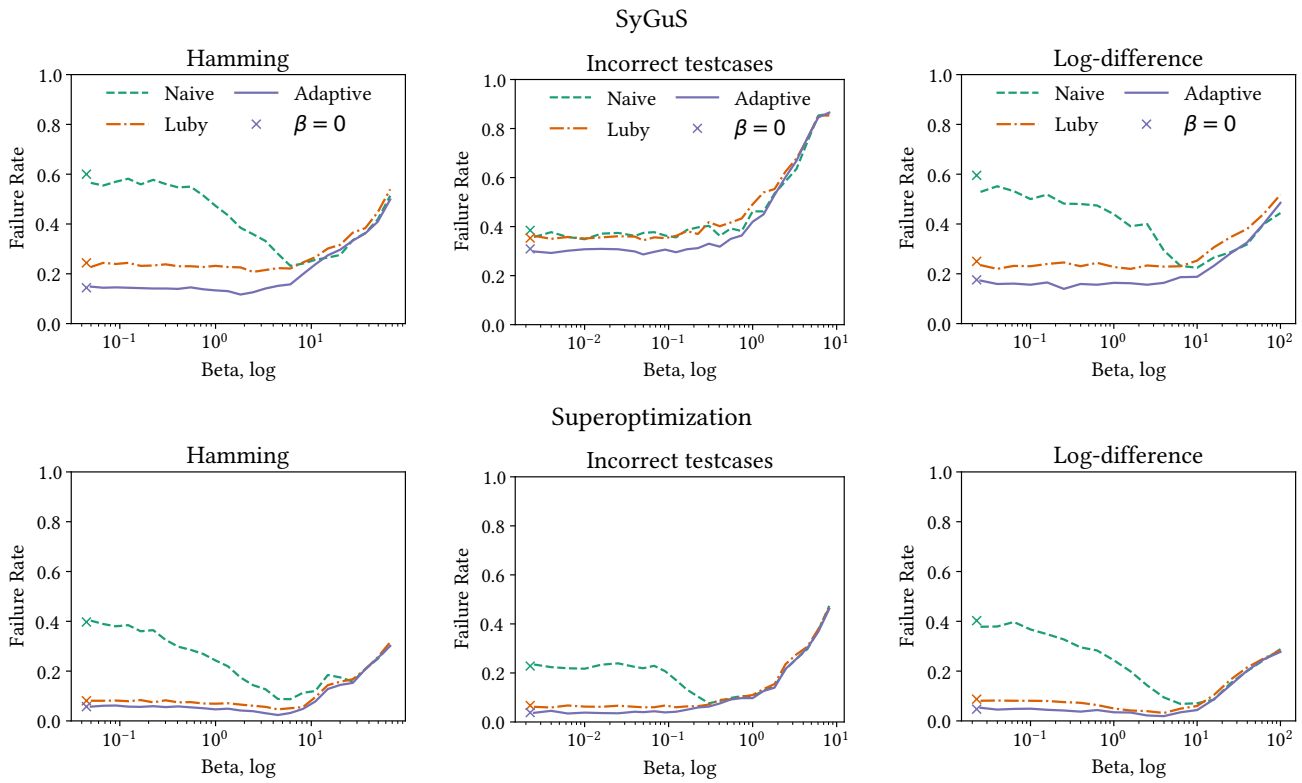


Figure 13. Effect of beta on the performance of the various algorithms. The beta parameter is plotted in logarithmic space, and has a different range for incorrect test cases due to the different scale of that cost function. The y-axis is failure rate, and as such lower curves are better. A "×" gives the performance of $\beta = 0$, which cannot be plotted directly in logarithmic space.

Table 1. Optimal β for various algorithms, benchmarks, and cost functions. These values are the minimums of the curves in Figure 13.

Cost	Benchmark	Naive β	Luby β	Adapt. β
Hamming	SyGuS	6.05	2.46	1.82
Hamming	Superopt.	6.05	4.48	4.48
Inc. tests	SyGuS	0.01	0.0498	0.0498
Inc. tests	Superopt.	0.301	0.004	0.006
Log-diff.	SyGuS	10.0	0.0398	0.251
Log-diff.	Superopt.	6.31	3.98	3.98

7.2 Calculating Mean Time to Synthesize

We run 50 trials for each algorithm with a limit of 100 million iterations. To allow us to estimate the mean time to synthesize even when some trials time out, we add a penalty $P = (\frac{1}{p_s} - 1)C$ to the mean of the successful trials where p_s is the empirical probability of a successful trial. This estimate of the mean is equivalent to a meta-restart strategy that always completely resets the algorithm after C iterations, because the penalty is the expected number of iterations spent in failed meta-trials. Because this penalty estimate is noisy when the number of successes is small, means beyond C iterations are increasingly unreliable.

This cutoff and penalty strategy creates an artifact in the comparison chart: if the optimal cutoff t^* for a problem happens to be around C, then the naive algorithm will actually be optimal among all black-box restart algorithms. In a large benchmark some problem(s) will coincidentally have a t^* close to C, and the estimated mean underestimates of the true cost of the naive algorithm. We see this when the naive and Luby lines converge near C (e.g., Figure 14).

7.3 Algorithm Comparisons and Discussion

To compare the algorithms at their optimal β in more detail we use *cactus plots*, which compare the mean time to synthesize (y-axis) of the individually nth fastest synthesis problem (x-axis) for each algorithm. In these plots, vertical lines correspond to a particular ordinal rank, but not necessarily the same underlying synthesis problem for each algorithm. Horizontal lines allow comparing the fraction of the dataset that an algorithm can solve within a limit. These plots gracefully handle each algorithm timing out on different problems, which would otherwise prevent computing a speedup for those examples. This style of plot has been used

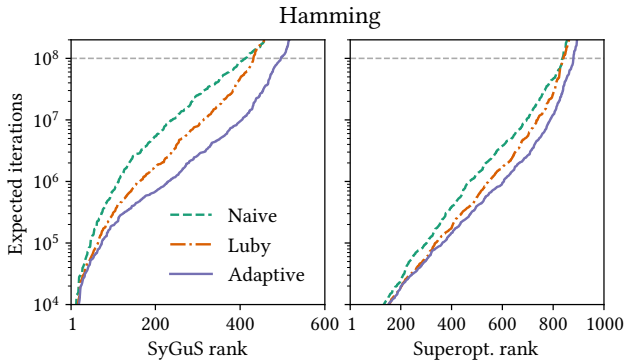


Figure 14. Cactus plot for the Hamming cost function, log axis. The horizontal dashed line is the 100M iteration cutoff.

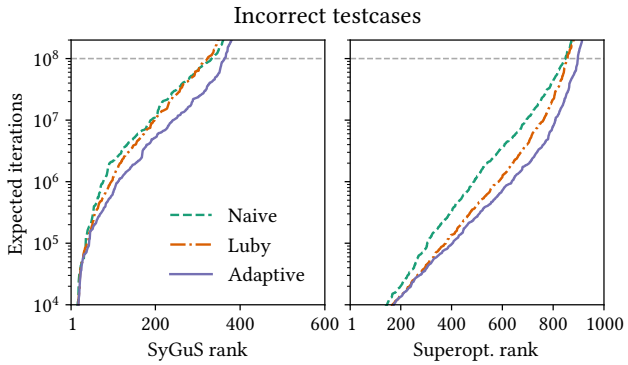


Figure 15. Cactus plot for the incorrect test cases cost function.

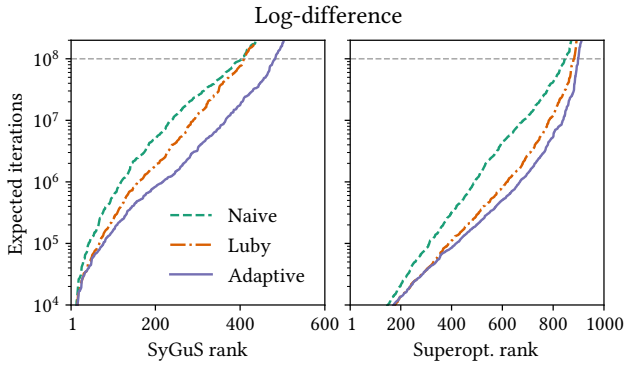


Figure 16. Cactus plot for the log-difference cost function.

in e.g. theorem proving competitions to compare solvers that may time out on different problems [24]. We give the results for the cost functions in Figures 14, 15, and 16.

The results for the Hamming cost function are given in Figure 14. The Luby algorithm generally outperforms naive, and the adaptive restart outperforms both. Because these plots are in logarithmic space, small gaps represent large speedups.

Table 2. Speedups at various ordinal ranks. Each speedup is the geometric mean of a small window to reduce noise, and (-) indicates that time outs prevent calculating a ratio.

		SyGuS Rank		Superopt. Rank	
Cost func.	Algo.	300th	400th	500th	700th
Hamming	Naive	11.07	10.63	3.63	4.50
	Luby	3.31	5.57	1.45	2.27
	Adaptive	1	1	1	1
Inc. tests	Naive	2.99	1.69	4.86	6.35
	Luby	3.01	-	1.59	2.32
	Adaptive	1	1	1	1
Log-diff.	Naive	8.28	5.84	6.35	9.74
	Luby	3.93	4.64	1.48	2.09
	Adaptive	1	1	1	1

Table 3. Fraction of problems unsynthesized within 100M expected iterations.

Cost function	Algorithm	SyGuS	Superopt.
Hamming	Naive	31.3%	16.3%
	Luby	28.2%	16.0%
	Adaptive	17.3%	12.1%
Incorrect test cases	Naive	44.7%	15.2%
	Luby	46.3%	14.5%
	Adaptive	39.5%	10.3%
Log-difference	Naive	32.8%	15.3%
	Luby	31.8%	11.7%
	Adaptive	19.7%	10.3%

We can see that for easier examples (the left of the plots), there is not very much difference between the algorithms. We also note that the naive and Luby curves intersecting near the cutoff of 100M iterations is an artifact of the penalty method for computing means as noted previously.

The incorrect test cases cost function has a much smaller difference between the naive and Luby algorithms. This is due to the low signal from the cost function: because the cost function only gives one bit of information per test case, and the SyGuS benchmark has a low average number of test cases, there are not many distinct values of the cost function, and thus the problems often only have a single dominant plateau. We see that for the superoptimization benchmark, where almost all examples have 100 test cases, the curves are very similar to Hamming. These results reinforce the intuition that the cost function needs to have enough signal to guide the search in order to effectively synthesize programs.

The log-difference cost function shows the same general trend as the other two, with more separation between the restart algorithms and the naive algorithm.

We can summarize the results of these experiments in Table 2, which gives the speedup of the adaptive algorithm over the other algorithms at a selection of ordinal ranks. Each

number is calculated as the ratio of the curves at a specific vertical lines in the previous charts, and thus compares the speedup of the nth fastest synthesis problem for one algorithm against another. We note that the adaptive algorithm is always the fastest, and usually about twice as fast as the classic Luby algorithm and up to an order of magnitude faster than the naive algorithm.

We can summarize these results in a different way by comparing the fraction of the benchmark unsolved within an expected 100M iterations. This corresponds to determining where the curves cross the dashed lines in the figures. Table 3 lets us compare cost functions; observe for example that the incorrect test cases and log-difference cost functions are best for the superoptimization benchmark and the Hamming function is best for the SyGuS benchmark. We also see that the adaptive restart algorithm is always the most successful algorithm, which mirrors our results from the β comparison.

7.4 Analysis of Unsynthesized Problems

A noticeable fraction of programs do not synthesize with an expected time less than 100M iterations. For the SyGuS benchmark, our search considers a significantly different set of operations than those required for the benchmark. For the superoptimization benchmark all programs are potentially synthesizable by construction, but using the adaptive restart algorithm, there were 28 synthesis problems that did not synthesize in the 150 trials from all three cost functions, representing 2.8% of the 1,000 problems from the superoptimization benchmark (i.e., 97.2% did synthesize at least once).

We manually reviewed these programs and classified why they never synthesized. The primary issues were non-trivial constants (16) and using a large number of shifts (7); the 5 remaining programs failed for other reasons. Constants are difficult to guess and may require generating something close to the final result to have a good chance of synthesizing. Shifts cause problems because all of the cost functions are not smooth when shifts are involved. Not being smooth with respect to the syntax of the program is not specific to the these cost functions: all low-complexity cost functions must have some programs for which they perform poorly.

8 Conclusion

We have analyzed the behavior of stochastic synthesis, showing that searches proceed from plateau to plateau, and these plateaus dictate the distribution of search times. We also presented a new superoptimization benchmark of low level program fragments. In analyzing algorithms for exploiting heavy-tailed distributions of search times, we found the strengths of the classic Luby algorithm could be improved by using the cost of the search in our adaptive restart algorithm. We also showed that to gain advantage from this adaptive restart algorithm, the β parameter must be tuned to ensure the search is not a random walk, and we also found that the adaptive restart algorithm works well for a wide range of choices of β in our benchmarks. Given the performance improvements over the current standard algorithm and existing black-box restart strategies, and the simplicity of the adaptive algorithm, we suggest that the adaptive algorithm should be employed whenever stochastic search is used.

Acknowledgments

This work was supported by NSF grants CCF-1160904 and CCF-1409813 as well as a grant of cloud credits from Amazon Web Services.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017 (EPTCS, Vol. 260)*, Dana Fisman and Swen Jacobs (Eds.). 97–115. https://doi.org/10.4204/EPTCS.260.9

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

[4] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[5] Armin Biere. 2008. PicoSAT Essentials. *J. Satisf. Boolean Model. Comput.* 4, 2-4 (2008), 75–97. https://doi.org/10.3233/sat190039

[6] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 313–326. https://doi.org/10.1145/3093337.3037754

[7] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.* 52, 6 (June 2017), 422–436. https://doi.org/10.1145/3140587.3062351

[8] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Québec, Canada) *(GECCO '09)*. ACM, New York, NY, USA, 947–954. https://doi.org/10.1145/1569901.1570031

[9] C. Green. 1969. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 219–239. http://dl.acm.org/citation.cfm?id=1624562.1624585

[10] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[11] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the X86-64 Instruction Set. *SIGPLAN Not.* 51, 6 (June 2016), 237–250. https://doi.org/10.1145/

2980983.2908121

[12] S. Heule, M. Sridharan, and S. Chandra. 2015. Mimic: Computing Models for Opaque Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 710–720. https://doi.org/10.1145/2786805.2786875

[13] J. Huang. 2007. The Effect of Restarts on the Efficiency of Clause Learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence* (Hyderabad, India) *(IJCAI'07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2318–2323. http://dl.acm.org/citation.cfm?id=1625275.1625649

[14] V. Le, C. Sun, and Z. Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[15] M. Luby, A. Sinclair, and D. Zuckerman. 1993. Optimal speedup of Las Vegas algorithms. In *[1993] The 2nd Israel Symposium on Theory and Computing Systems*. 128–133. https://doi.org/10.1109/ISTCS.1993.253477

[16] H. Massalin. 1987. Superoptimizer: A Look at the Smallest Program. *SIGPLAN Not.* 22, 10 (oct 1987), 122–126. https://doi.org/10.1145/36205.36194

[17] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJ0JwFcex

[18] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. 2016. Scaling Up Superoptimization. *SIGPLAN Not.* 51, 4 (March 2016), 297–310. https://doi.org/10.1145/2954679.2872387

[19] E. Schkufza, R. Sharma, and A. Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (March 2013), 305–316. https://doi.org/10.1145/2499368.2451150

[20] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-Point Programs with Tunable Precision. *SIGPLAN Not.* 49, 6 (June 2014), 53–64. https://doi.org/10.1145/2666356.2594302

[21] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (Jan. 2016), 114–122. https://doi.org/10.1145/2863701

[22] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. *SIGPLAN Not.* 50, 10 (Oct. 2015), 147–162. https://doi.org/10.1145/2858965.2814278

[23] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGPLAN Not.* 41, 11 (Oct. 2006), 404–415. https://doi.org/10.1145/1168918.1168907

[24] Geoffrey Sutcliffe. 2016. The 8th IJCAR automated theorem proving system competition - CASC-J8. *AI Communications* 29, 5 (2016), 607–619. https://doi.org/10.3233/AIC-160709

[25] R. Waldinger and R. Lee. 1969. PROW: A Step Toward Automatic Program Writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 241–252. http://dl.acm.org/citation.cfm?id=1624562.1624586