1 High-level overview

We want to sample programs from a description length prior. Let x be a program. Sample from

$$P(x) = \frac{2^{-|x|}}{Z} \tag{1}$$

Assume $|x| \leq n$ always, where n is the number of SAT variables encoding the structure of the program.

The approach will draw from two techniques: (1) constraint-solving techniques for program synthesis; and (2) uniform solution sampling using XOR constraints.

The two most obvious approaches to this problem don't work in practice:

- Prefix codes: Write down a prefix code for the space of allowed programs. Uniformly sample values of all n SAT variables encoding the program structure; then the probability of sampling x is proportional to $2^{n-|x|}$.
- Embed & Project: Introduce n auxiliary variables, A, and add constraints $|x| > j \implies A_{j+1}$. Uniformly sample solutions to (x, A). This is the technique introduced in "Embed and Project" (Stefano Ermon, NIPS 2013)

The problem with these approaches is that they dramatically increase the number of satisfyable solutions to the underlying SAT formula by a factor of $O(2^n)$, and so require at least about n random constraints; for large n, the solver scales exponentially with the number of random constraints (empirically observed for even CryptoMiniSAT). The body of work on sampling with random constraints has previously restricted itself to cases where the number of satisfiable solutions is relatively small (< 600 in the original Gomes et al. NIPS paper; on the order of tens of thousands in later work, by the authors of UniGen).

Our idea is to use auxiliary variables as in "Embed and Project," but reduce the dimensionality of the embedding by only hashing the first α bits of A. If we put $\alpha = n$ then our samples come from P; otherwise they come from a different distribution Q:

$$Q(x) \propto 2^{-\min(\alpha,|x|)} \tag{2}$$

We can correct for this proposal distribution by accepting a sample x from Q with probability $A(|x|, \alpha) = 2^{\min(0,\alpha-|x|)}$. So we accept any sample whose description length is bounded by α and accept longer samples with probability $2^{\alpha-|x|}$. This satisfies $Q(x)A(|x|, \alpha) \propto P(x)$.

The only reason that this works in the general case is because of the assumption that $|x| \leq n$, which allows us to make Q arbitrarily close to P (and thus arbitrarily decrease the rejection rate), but at the expense of needing more random constraints. The rest of this document analyzes the trade-off between increasing α and increasing K (the number of random constraints), and then presents some experimental results.

2 Algorithmic details

2.1 The algorithm

Let X be a set of programs satisfying some specification, such as agreeing with certain input-output pairs. We want to sample from

$$P(x \in X) = \frac{2^{-|x|}}{Z} \tag{3}$$

where $Z = \sum_{x \in X} 2^{-|x|}$.

Let α be the dimensionality of the embedding. The set E of programs in the embedded space is

$$E = \{(x, A) : x \in X, |x| > j \implies A_{j+1} \text{ when } j < \alpha\} \tag{4}$$

In each iteration of the algorithm we sample a random projection from $n+\alpha$ to K variables, setting it equal to a random K dimensional vector. Let $h(x, A)$ be true if $(x, A) \in E$ satisfies this random constraint. Let $S = \{e \in E : h(e)\}$. If S is empty, reject this attempt at sampling. Otherwise draw uniformly from S to obtain (x, A), and reject this attempt at sampling with probability $1 - A(|x|, \alpha) = 1 - 2^{\min(0, \alpha - |x|)}$.

The embedding introduces many symmetries (lots of e's that have the same x), and we can break this symmetry by only enumerating unique values of x. For a given x, the number of $(x, A) \in S$ can be calculated without further calls to the solver: it is $2^{\alpha - |x| - \text{rank}(H_{>n+|x|})}$ where H is the matrix used for the random hash.

2.2 Algorithm analysis

First, what is the probability that a sample from Q is accepted? Call this $P(\text{accepted})$:

$$\begin{aligned}
P(\text{accepted}) &= \sum_x Q(x) A(|x|, \alpha) && (5)\\
&= \sum_x \frac{2^{-|x|}}{\sum_{x'} Q(x')} && (6)\\
&= \frac{Z}{\sum_{|x|>\alpha} 2^{-\alpha} + \sum_{|x| \le \alpha} 2^{-|x|}} && (7)\\
&= \frac{Z}{Z - Z_\alpha + 2^{-\alpha} C_\alpha} && (8)\\
&= \frac{1}{1 + 2^{-\alpha} C_\alpha / Z - P(|x| > \alpha)} && (9)
\end{aligned}$$

where $C_\alpha = |\{x \in X : |x| > \alpha\}|$ and $Z_\alpha = \sum_{|x|>\alpha} 2^{-|x|}$.

Second, what is the probability that we get a sample from Q? Let mc be the model count (members of E that survived the hashing constraint). Then

$E[mc] = \mu = 2^{-K}|E|$ and Variance$[mc] = \sigma^2 = |E|2^{-K}(1-2^{-K})$. We fail to get a sample if $mc < 1$, equivalently, $mc \leq \epsilon$ for $0 < \epsilon < \min(\mu, 1)$.

$$\begin{aligned} P(mc \leq \epsilon) &< P(|mc - \mu| \geq \mu - \epsilon) && (10)\\ &\leq \frac{\sigma^2}{(\mu-\epsilon)^2}, \text{ Chebyshev 's inequality} && (11)\\ &\leq \frac{\mu}{(\mu-\epsilon)^2}. && (12)\end{aligned}$$

Put these together to bound the probability of failing to get a sample:

$$\begin{aligned} P(\text{fail to get sample}) &= P(mc=0) + P(mc>0)(1 - P(\text{accepted})) && (13)\\ &< \frac{\mu}{(\mu-\epsilon)^2} + 1 - \frac{1}{1 + 2^{-\alpha}C_\alpha/Z - P(|x| > \alpha)} && (14)\\ P(\text{get sample}) &> \frac{1}{1 + 2^{-\alpha}C_\alpha/Z - P(|x| > \alpha)} - \frac{\mu}{(\mu-\epsilon)^2} && (15)\\ & && (16)\end{aligned}$$

Now that we have a bound on the probability of getting a sample (and thus abound on the expected number of attempts needed to get a sample), we want to analyze how much work we expect to do on each attempt at sampling. A reasonable proxy for this is the number of invocations made to the solver, which is just the number of programs in E that survived the constraints (*not* the number of surviving solutions). Let x be a program. If $|x| > \alpha$ then the probability that it survives is just 2^{-K}. Otherwise it's more complicated. Let $|x| \leq \alpha$, H be the random matrix and b the random vector. The hashing constraint reads

$$b_i = \sum_{j=1}^{n} H_{ij}x_j + \sum_{j=n+1}^{n+|x|} H_{ij}A_{j-n} + \sum_{j=n+|x|+1}^{n+\alpha} H_{ij}A_{j-n} \quad (17)$$

The first two sums can be folded into b_i, giving

$$b'_i = \sum_{j=n+|x|+1}^{n+\alpha} H_{ij}A_{j-n} \quad (18)$$

So x survives if $b' \in \text{range}(H_{j \geq n+|x|+1})$. The matrix $H_{j \geq n+|x|+1}$ is K by $\alpha - |x|$; call it G.

$$\begin{aligned} P(x \text{ survives}) &= \sum_{r=0}^{\min(K,\alpha-|x|)} P(b' \in \text{range}(G)|\text{rank}(G)=r)P(\text{rank}(G)=r) && (19)\\ &= \sum_r 2^{r-K}2^{-K(\alpha-|x|)} \prod_{0 \leq i < r} \frac{(2^K - 2^i)(2^{\alpha-|x|} - 2^i)}{2^r - 2^i} && (20)\\ &< \min(1, 2^{\alpha-|x|-K}) && (21)\end{aligned}$$

where the last inequality is not derived from the former inequality but from applying the union bound to the original statement.

What is the total expected work to get one sample? Let $s_i(x)$ be an indicator variable for whether x survives the ith sampling attempt, and f_i be an indicator variable for whether the ith sampling attempt failed. the number of solver invocations is

$$\begin{aligned} & 1 + \sum_x s_1(x) \\ + \quad & f_1 + \sum_x s_2(x) f_1 \\ + \quad & f_1 f_2 + \sum_x s_3(x) f_1 f_2 \\ + \quad & \cdots \end{aligned}$$

which has the expectation

$$\frac{1 + \sum_x P(x \text{ survives})}{P(\text{get sample})} \tag{22}$$

We can upper bound this quantity using the previously bounded quantities. The only approximation comes in here:

$$p(\text{no survivors}) < \frac{\mu}{(\mu - \epsilon)^2} \tag{23}$$

Thus by enumerating all of the solutions to a program synthesis problem we can compute fairly tight bounds on the performance of the proposed algorithm, as the next section details.

In practice we want to sample without enumerating all of the solutions. Notice that $\mu = |E|2^{-K} > (|S| - 1 + 2^{\alpha - L})2^{-K}$ where $L = \min_x |x|$. So if K is much less than $\log(|S| - 1 + 2^{\alpha - L})$ then $1/\mu$ is close to zero. More precisely, introduce a parameter $\delta > 0$,

$$\begin{aligned} K + \delta &= \log(|S| - 1 + 2^{\alpha - L}) && (24) \\ 2^{\delta} &= 2^{-K}(|S| - 1 + 2^{\alpha - L}) < \mu && (25) \\ 2^{-\delta} &> 1/\mu && (26) \end{aligned}$$

From here on out assume that $K + \delta = \log(|S| - 1 + 2^{\alpha - L})$ (look at the experimental results: you only get good sampling when you are in a region that falls below a diagonal line on the heat maps. This diagonal line corresponds to this constraint) and plug the above result into the upper bound:

$$\begin{aligned} E[\text{invocations}] &= \frac{1 + \sum_x P(x \text{ survives})}{P(\text{get sample})} && (27) \\ &< \frac{1 + \sum_x P(x \text{ survives})}{\frac{1}{1 + 2^{-\alpha} C_\alpha / Z - P(|x| > \alpha)} - 2^{-\delta}} && (28) \end{aligned}$$

For large enough α, the probability of a sample from Q being accepted has a simple bound. Assume from here on out that $\alpha = \log|S| + L + \gamma$ for $\gamma > 0$:

$$\begin{aligned} 1/P(\text{accepted}) &= 2^{-\alpha} C_\alpha / Z + P(|x| \leq \alpha) && (29)\\ &< 2^{-\alpha}|S|2^L + 1 && (30)\\ &= 2^{-\log|S| - L - \gamma}|S|2^L + 1 && (31)\\ &= 1 + 2^{-\gamma} && (32) \end{aligned}$$

So the probability of acceptance approaches 1 exponentially quickly once α gets big enough. Looking at the heat maps in the experimental results, we see that we don't get good samplers until $\alpha - L$ crosses some threshold that increases as the number of possible programs increases. This qualitative feature of the heat maps is reflected in the above analysis.

The last piece of the approximation is to show that K cannot be too small compared to alpha or else the number of programs enumerated in each round of sampling grows two large.

$$\sum_x P(s_x) = \sum_{\alpha - K \leq |x|} \max(2^{-K}, 2^{\alpha - K - |x|}) + C_{<\alpha - K} \quad (33)$$

Suppose that we hold $\alpha - K$ constant, so that we are restricting ourselves to a diagonal line on the heat maps. Then the above equation is decreasing in K, so as we walk down the diagonal we get increasingly better samplers (and walking all the way down corresponds to prior work). What if we don't restrict ourselves to a diagonal? Consider the case of moving K while keeping α constant. Number of survivors grows like $2^{-K} \propto 2^{\delta}$, number of iterations grows like $1/(1 - 2^{-\delta}(1 + 2^{-\gamma}))$. let $w = 2^{\delta}$, then our loss function looks like $w^2/(w - (1 + 2^{-\gamma}))$, which achieves its minimum at $\delta = 1 + \log(1 + 2^{-\gamma})$. So for large γ we expect the optimal δ to be close to 1. Looking at the heat maps it seems that a better value is 2, but that 1 is pretty close.

Let's simplify things a little bit:

$$\begin{aligned} \alpha &= \log|S| + L + \gamma && (34)\\ K &= \alpha - L - \delta && (35)\\ &= \log|S| + \gamma - \delta && (36) \end{aligned}$$

With these new definitions of the parameters,

$$\begin{aligned} E[tt] =& \frac{1 + \sum_{\alpha - K \leq |x|} \max(2^{-K}, 2^{\alpha - K - |x|}) + C_{<\alpha - K}}{1 - 2^{-\delta}(1 + 2^{-\gamma})}(1 + 2^{-\gamma}) && (37)\\ =& \frac{1 + 2^{\delta}\sum_{|x| \geq L + \delta} \max(2^{-\gamma}/|S|, 2^{L - |x|}) + C_{<L+\delta}}{1 - 2^{-\delta}(1 + 2^{-\gamma})}(1 + 2^{-\gamma}) && (38)\\ & && (39) \end{aligned}$$

which is much clearer.

2.3 KL divergence of the proposal distribution

$$D(P||Q) = \sum_x P(x) \log P(x)/Q(x) \tag{40}$$

$$= \sum_{|x|\leq\alpha} P(x) \log \frac{2^{-|x|} Z_Q}{2^{-|x|} Z} + \sum_{|x|>\alpha} P(x) \log \frac{2^{-|x|} Z_Q}{2^{-\alpha} Z} \tag{41}$$

$$= \log(Z_Q/Z) + \sum_{|x|>\alpha} P(x)(\alpha - |x|) \tag{42}$$

$$< \log(Z_Q/Z) \tag{43}$$

$$= \log \frac{Z - Z_\alpha + C_\alpha 2^{-\alpha}}{Z} \tag{44}$$

$$= \log(1 - P(|x| > \alpha) + C_\alpha 2^{-\alpha}/Z) \tag{45}$$

$$< \log(1 + |S| 2^L 2^{-\alpha}) \tag{46}$$

$$= \log(1 + 2^{-\gamma}) \tag{47}$$

$$< \frac{2^{-\gamma}}{\ln 2} \tag{48}$$

$$\max_x |P(x) - Q(x)| < \frac{2^{-\gamma/2}}{\sqrt{\ln 4}}, \text{ Pinsker's inequality} \tag{49}$$

and so the kl divergence goes to zero exponentially quickly in γ, as does the total variation distance. This suggests an alternative algorithm in which we never reject samples from Q but instead specify a bound on the KL divergence which in turn gives us a bound on γ. This tightens the upper bound on expected run time given in the previous section, and makes the math simpler.

2.4 Fluctuations around Q

We don't sample exactly from Q but from a related distribution Q', because these XOR constraints introduce small fluctuations around the true distribution. Let x temporarily refer to an assignment both of the program and auxiliary variables, and let $\mu = 2^{-K}(N-1) + 1$ the average number of survivors given

that x survives:

$$p_1(x) = 2^{-K} \sum_i \frac{1}{i} P(mc = i | x \text{ survives}) \tag{50}$$

$$= 2^{-K} E[\frac{1}{mc} | x \text{ survives}] \tag{51}$$

$$> 2^{-K} \frac{1}{E[mc | x \text{ survives}]}, \text{ Jensen's inequality} \tag{52}$$

$$= 2^{-K} 1/\mu \tag{53}$$

$$= \frac{2^{-K}}{2^{-K}(N-1)+1} \tag{54}$$

$$= \frac{1}{N-1+2^K} \tag{55}$$

$$= \frac{1}{N} \times \frac{1}{1+2^K/N-1/N} \tag{56}$$

$$> \frac{1}{N} \times \frac{1}{1+2^K/N} \tag{57}$$

$$D(Q||Q') = \sum_x q(x) \log \frac{q(x)}{q'(x)} \tag{58}$$

$$= \sum_x q(x) \log \frac{|A_x|(1/N)}{\sum_{A_x} p_1(x, A_x)/P(mc > 0)} \tag{59}$$

$$< \log P(mc > 0) + \sum_x q(x) \log \frac{|A_x|(1/N)}{\sum_{A_x}(1/N)/(1+2^K/N)} \tag{60}$$

$$= \log P(mc > 0) + \sum_x q(x) \log \frac{|A_x|}{|A_x|/(1+2^K/N)} \tag{61}$$

$$= \log P(mc > 0) + \sum_x q(x) \log(1+2^K/N) \tag{62}$$

$$= \log P(mc > 0) + \log(1+2^K/N) \tag{63}$$

$$< 2^K/N \tag{64}$$

$$< 2^K/(|S| - 1 + 2^{\alpha - L}) \tag{65}$$

$$= 2^{-\delta}. \tag{66}$$

2.4.1 Fluctuations around P

Of course we actually care about $D(p||q_K)$, where q_K is the distribution sampled by the algorithm. The previous two sections were warm-ups for this calculation. Let x be a program. Then $q_K(x) = p_1(x)A(|x|, \alpha)/W$ (where W is the

normalizing constant) so:

$$D(p||q_K) = \sum_x p(x) \log \frac{p(x)}{q_K(x)} \tag{67}$$

$$= \sum_x p(x) \log \frac{A(x)q(x)}{\sum_y A(y)q(y)} \frac{\sum_y A(y)q_k(y)}{A(x)q_k(x)}, \text{ as } q(x)A(x) \propto p(x) \tag{68}$$

$$= \log \frac{\sum_y A(y)q_k(y)}{\sum_y A(y)q(y)} + \sum_x p(x) \log \frac{q(x)P(mc > 0)}{p_1(x)} \tag{69}$$

$$< \log \frac{\sum_y A(y)q_k(y)}{\sum_y A(y)q(y)} + \log \frac{1}{c} + \log P(mc > 0) \tag{70}$$

$$< \log \frac{\sum_{y \neq x_*} A(y)c'q(y) + A(x_*)(1 - \sum_{y \neq x_*} c'q(y))}{\sum_y A(y)q(y)} + \log \frac{P(mc > 0)}{c} \tag{71}$$

$$= \log \frac{\sum_{y \neq x_*} A(y)c'q(y) + 1 - \sum_{y \neq x_*} c'q(y)}{\sum_y A(y)q(y)} + \log \frac{P(mc > 0)}{c} \tag{72}$$

$$= \log \frac{1 + c' \sum_{y \neq x_*} q(y)(A(y) - 1)}{\sum_y A(y)q(y)} + \log \frac{P(mc > 0)}{c} \tag{73}$$

$$= \log \frac{1 + c' \sum_y q(y)(A(y) - 1)}{\sum_y A(y)q(y)} + \log \frac{P(mc > 0)}{c} \tag{74}$$

$$= \log \frac{1 + c'(\sum_y q(y)A(y) - 1)}{\sum_y A(y)q(y)} + \log \frac{P(mc > 0)}{c} \tag{75}$$

$$= \log \left(c' + \frac{1 - c'}{P(\text{accept})} \right) + \log \frac{P(mc > 0)}{c}, \text{ where } c' = \frac{c}{P(mc > 0)} \tag{76}$$

$$= \log \left(1 + \frac{1 - c'}{P(\text{accept})} \times \frac{P(mc > 0)}{c} \right) \tag{77}$$

$$= \log \left(1 + \frac{(1 + 2^K/N - 1/N)P(mc > 0) - 1}{P(\text{accept from } Q)} \right) \tag{78}$$

$$< \log \left(1 + \frac{2^K/N}{P(\text{accept from } Q)} \right) \tag{79}$$

$$< \log \left(1 + 2^{-\delta}(1 + 2^{-\gamma})\right) < 2^{-\delta} + 2^{-\delta-\gamma} < 2^{-\delta+1} \tag{80}$$

And so the KL divergence decreases exponentially in δ, and can be closely bounded above after enumerating all of the programs by equation 78. It is easier to show that a bound with the similar asymptotics (specifically, $O(2^{-\min(\delta,\gamma)})$ holds whenever we don't perform the rejection, which is interesting, and suggests that the rejection sampling isn't buying us very much. So the amount of work we have to do grows exponentially as we move away from the diagonal, and the KL divergence decays exponentially as we move away from the diagonal. This suggests going as close to the diagonal as the bound on the KL will allow.

2.5 Entropy in the program distribution

$$H[P] = \sum_x 1/Z 2^{-|x|} \log Z 2^{|x|} \tag{81}$$
$$= \sum_x 1/Z 2^{-x} (\log Z + x) \tag{82}$$
$$= \log Z + \sum_x P(x)|x| \tag{83}$$
$$= \log Z + E_P[|x|] \tag{84}$$
$$\tag{85}$$

Use the Markov inequality to relate this to C_α:

$$P(|x| > l) < E_p[|x|]/l = (H[P] - \log Z)/l \tag{86}$$
$$lP(|x| > l) + \log Z < H[p] \tag{87}$$
$$Z < 2^{H[P] - lP(|x|>l)} \tag{88}$$

Let $l > 0$:

$$C_{<l} = \sum_{|x|<l} 1 \tag{89}$$
$$= \sum 2^l 2^{-l} \tag{90}$$
$$< Z 2^l \sum_{|x|>l} P(x) \tag{91}$$
$$= Z 2^l P(|x| > l) \tag{92}$$
$$< 2^H 2^{lP(|x|<l)} P(|x| < l) \tag{93}$$
$$< 2^{H+l} \tag{94}$$

Which thus far seems to be useless, probably because the Markov inequality is super loose in practice.

3 Experimental results

3.1 List manipulation

This is a domain of recursive list manipulation programs designed to demonstrate the ability of program sampling to synthesize algorithms. There are two goals here: (1) show that description length priors can give reasonable algorithm implementations from examples, and (2) show that by maintaining a full posterior over programs we get better predictive accuracy than just picking the most likely program.

Primitives include car, cdr, append, $\geq$, $\leq$, 0, nil, increment, decrement, length, filter, and a list singleton constructor. The sketch constrains the space

by (1) specifying that there is exactly one base case guarded by one conditional, and (2) specifying that the return value is a concatenation of lists which may be optionally recursed upon. This is expressive enough to encode sorting, reversing, indexing, counting, and searching through lists.

3.1.1 Sorting

Given a these few examples, can we learn a posterior over sorting programs? In particular, does sampling help us in this domain?

```
f([]) = [], f([5]) = [5], f([7,9]) = [7,9], f([2,7]) = [2,7],
f([5,4,3]) = [3,4,5], f([5,3,4]) = [3,4,5]
```

I sampled 10 programs from the posterior using $K = 13$, $\alpha - L = 11$ and noticed the following program within the posterior:

```
(if ((eq 0) (length a))
    nil
    (append (recur (filter (gt (car a)) (cdr a)))
            (list (car a))
            (recur (filter (lt (car a)) (cdr a)))))
```

Which is a recursive implementation of quicksort. Interestingly this is not the most likely program, which only uses two conditional branches and effectively only recursives on the smaller list elements (intentionally omitted from the training set is an example that would force it to recursive in this case). So if you do map inference you get a hard decision that fails for many lists, but if you sample sorting algorithms and then marginalize over the program you get a soft decision that includes the correct sorted list.

Generating the samples took 40 minutes, compared to around a day to enumerate all 13000 solutions. This is a factor of 1–2 orders of magnitude better, but much less than what you would expect by just looking at the number of solver invocations (50 for sampling, which is 2–3 orders of magnitude fewer than enumeration). Empirically it seems that the first solver invocation takes a very long time, and subsequent invocations take on the order of seconds.

I picked these values of K and α by looking at Figure 3.1.1, which plots upper bounds upon the run time and KL divergence.

3.2 Flashfill

This is a domain of toy flash fill problems.

3.2.1 Small solution set regime

Problem:

```
Tom and Jerry      ---> Tom
Jack and Jill      ---> Jack
```

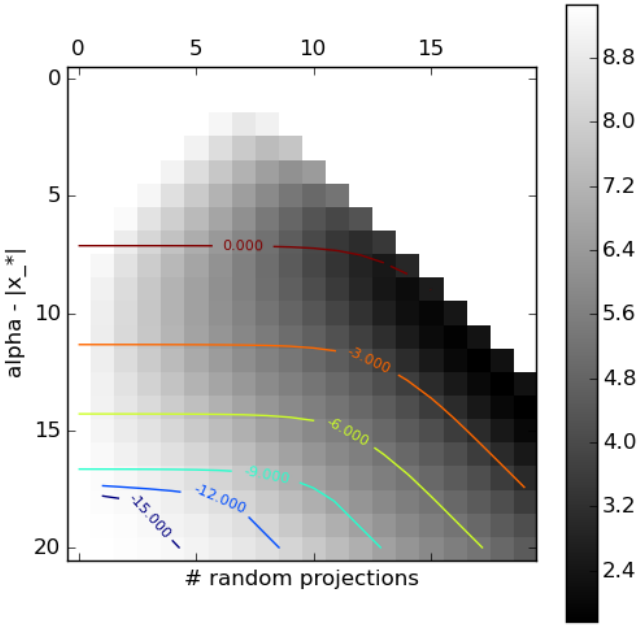


Figure 1: Grayscale: log expected solver invocations. Colored contours: upper bound on the log KL divergence.

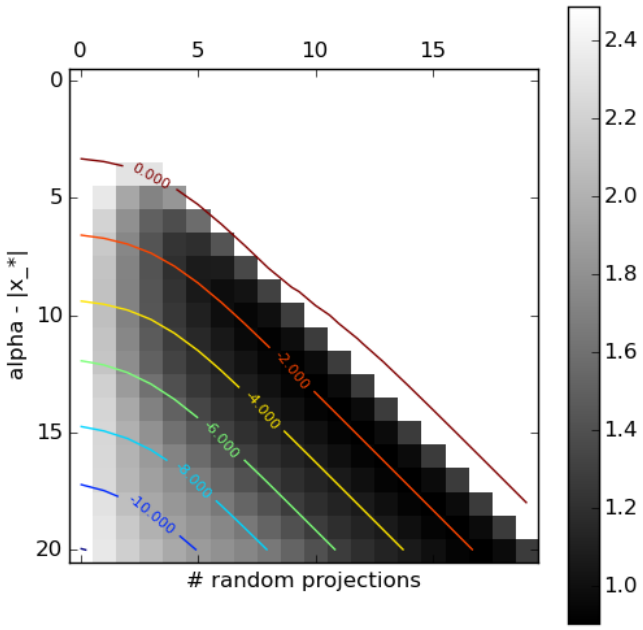


Figure 2: Grayscale: log expected solver invocations. Colored contours: upper bound on the log KL divergence.

There are 12 satisfying solutions to this problem within the program space. Thus we are trying to beat enumeration, which would take 13 calls to the solver. Figure 3.2.1 shows a heat map of the upper bound upon the numeber of solver invocations as a function of α and K. What if we sample at the beginning of the dark region ($\alpha = 27, K = 5$)? Below is the output of the sampler:

```
|s| = 1  log_2(z) = -19.0  1/p = 524288.0  shortest = 19 bits
Implicitly enumerated 8 satisfying solutions
Samples:
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
total time =  7.25420975685
```

So we were able to get away with only 5 random constraints. What if we go into the lighter region at $\alpha = 26$,$K = 2$? We observe that there are more programs enumerated and that some of them are rejected:

```
|s| = 5  log_2(z) = -18.955264207  1/p = 508280.094574  shortest = 19 bits
```

```
Implicitly enumerated 36 satisfying solutions
Samples:
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '", "'U'"],0)) 24
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,1)++SubString(1,2)++SubString(2,pos([],["' '"],0)) 42
Rejected.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '", "'U'"],0)) 24
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
total time =  7.45482897758
```

Lastly, does this scheme for reducing the number of constraints you need help for this problem? Put $\alpha = 52$ and $K = 31$.

```
|s| = 1  log_2(z) = -19.0  1/p = 524288.0  shortest = 19 bits
Implicitly enumerated 8 satisfying solutions
Samples:
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
```

```
Length bounded by alpha, so accepted.
SubString(0,pos([],["' '"],0)) 19
Length bounded by alpha, so accepted.
total time =  155.674251795
```

So without picking a small value of K we are actually slower than just enumerating all of the programs. Empirically the solver time seems to be blowing up around $K = 31$, thus we have good reason to not set α to n, which is about 150 bits for this program space.

3.2.2 Large solution set regime

If we remove the second training example, the correct program becomes highly ambiguous:

```
Tom and Jerry      -> Tom
```

There are 55434 possible programs in the space, and once we apply the embedding this number grows exponentially in α. It takes about 2300 seconds to enumerate these programs, but we can sample them much more efficiently with $K = 15$, $\alpha = 32$:

```
|s| = 7  log_2(z) = -11.9887727446  1/p = 4064.24806201  shortest = 12 bits
Implicitly enumerated 38 satisfying solutions
Samples:
SubString(0,3) 12
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
Const["'U'", 25, 23] 19
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
Const["'U'", 25, 23] 19
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
SubString(0,3) 12
Length bounded by alpha, so accepted.
total time =  17.3803441525
```

Notice that this is on a log scale due to the very large number of solutions.

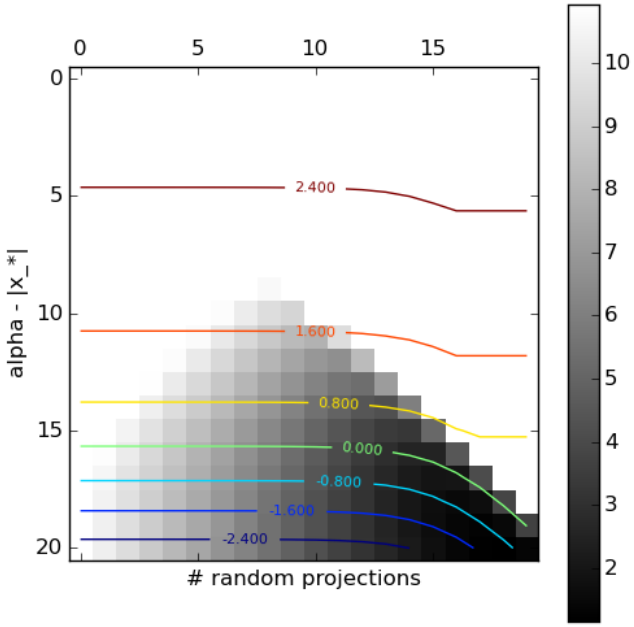


Figure 3: Grayscale: log expected solver invocations. Colored contours: upper bound on the log KL divergence.

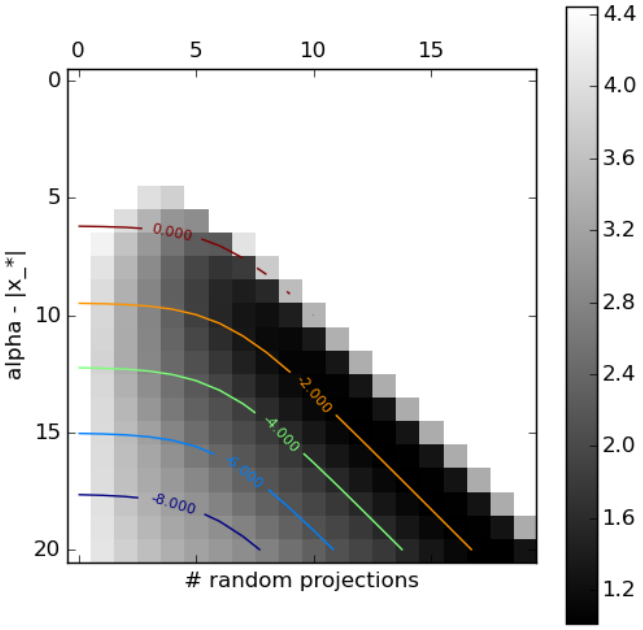


Figure 4: Grayscale: log expected solver invocations. Colored contours: upper bound on the log KL divergence.

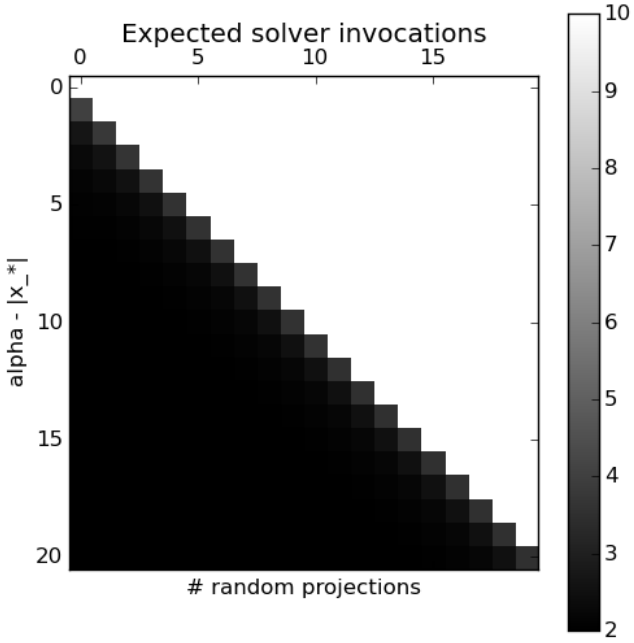
3.2.3 High description length programs

One benefit of using a solver is that it is sometimes easy to find programs even if they have a high description length, which can be caused by having some long constant in them. For example, consider the following training data:

```
Eyal Dechter      -> Hi Eyal!
Rishabh Singh     -> Hi Rishabh!
```

With only the first example, the performance curve looks like:

With two examples, there is only one satisfying solution (the program is uniquely determined).


Expected solver invocations
0
5
10
15
0
5
10
15
20
alpha - |x_*|
random projections
10
9
8
7
6
5
4
3
2