University of Alberta
Computing Science Department
XAI in Games - W20

Stochastic Search
Levi Lelis
levi.lelis@ualberta.ca

1 Stochastic Search

The problem of synthesizing scripts that satisfy a specification can be seen as an optimization problem. Let $G = (V, \Sigma, R, S)$ be a context free grammar defining a domain-specific language (DSL), where V is the set of non-terminal symbols, Σ is the set of terminal symbols, R are the rules in which one can transform non-terminals, and S is the initial symbol. Also, let $[\![G]\!]$ be the set of programs that can be synthesized with grammar G.

Given a set of training input-output pairs D, we can define an error function J for a program $p \in [\![G]\!]$ and D that counts the number of pairs for which the output produced by p for input x does not match with the true output y.

$$J(p, D) = \sum_{(x,y)\in D} [p(x) \neq y]$$

Then, the problem of synthesizing scripts can be formulated as the following problem,

$$\operatorname*{argmin}_{p\in[\![G]\!]} J(p, D) \tag{1}$$

We will see how the Metropolis-Hastings algorithm can be used for approximating a solution to Equation 1.

1.1 The Metropolis-Hastings Algorithm

The Metropolis-Hastings (MH) algorithm is able to produce samples from a probability density function by sampling from another distribution. The assumption for using MH and other Markov Chain Monte Carlo (MCMC) algorithms is that, although we can't sample from the target distribution, we can evaluate it.

Let's see an example. In this example we are interested in sampling from the following binomial distribution.

$$v(x) = 0.3\exp(-0.2x^2) + 0.7\exp(-0.2(x-10)^2)$$

We are able to evaluate the value of v for a given value of x, but we are unable to sample from v directly. In the MH algorithm we use a **proposal function** q from which we can obtain samples. We start with a sample $x_0 \sim q$ and then iteratively obtain new samples x_i conditioned to the sample x_{i-1}, $x_i \sim q(x_i|x_{i-1})$. The proposal function can be, for example, a Gaussian with mean x_{i-1}.

A sample x_i is probabilistically accepted by the algorithm. That is, MH accepts x_i according to a probability given by an **acceptance function**,

$$\alpha(x_i, x_{-i}) = \min\left\{1, \frac{v(x_i)q(x_{i-1}|x_i)}{v(x_{i-1})q(x_i|x_{i-1})}\right\} \tag{2}$$

Intuitively, larger values of v should be sampled more often than smaller values of v. The term $\frac{v(x_i)}{v(x_{i-1})}$ ensures a higher chance of accepting x_i if $v(x_i) > v(x_{i-1})$. The term $\frac{q(x_{i-1}|x_i)}{q(x_i|x_{i-1})}$ is to prevent the algorithm from getting stuck in some region of the space as it ensures that the algorithm has good chances of "backtracking", i.e., the term is maximized if chances of returning to x_{i-1} from x_i are higher than the probability of going to x_{i-1} from x_i. If the second term of the min operator is larger than 1, then MH accepts x_i.

Note that for some proposal functions are symmetric, i.e., $q(x_i|x_{i-1}) = q(x_{i-1}|x_i)$, thus simplifying the acceptance function to $\alpha(x_i, x_{-i}) = \min\left\{1, \frac{v(x_i)}{v(x_{i-1})}\right\}$. As an example, a Gaussian is symmetric.

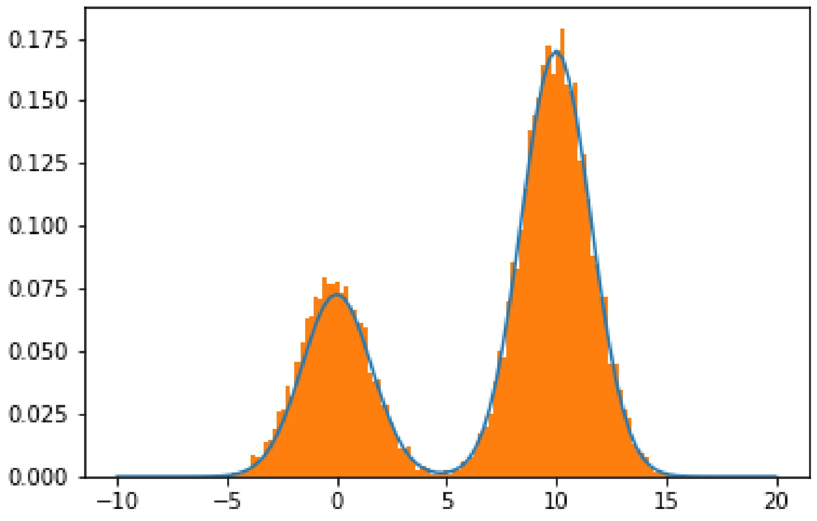
Here is an implementation of MH for our example, where the proposal function is a Gaussian.

```
samples = []
samples.append(np.random.normal(10, 1, 1)[0])

for i in range(0, 20000):
    x_i = samples[i]
    x_cand = np.random.normal(x_i, 1, 1)[0]
    accept = min(1, binomial(x_cand)/binomial(x_i))

    p = random.uniform(0, 1)
    if p < accept:
        samples.append(x_cand)
    else:
        samples.append(x_i)
```

Here is a plot with the binomial distribution and the samples collected by the MH implementation above.



The samples (in orange), follows very closely the binomial distribution we wanted to sample from.

1.2 Application of the Metropolis-Hastings to Program Synthesis

How can we apply the MH algorithm to program synthesis (i.e., solve Equation 1)? The MH algorithm is also suitable to optimization problems as the value x that maximizes the function will be sampled more often than any other value of the target function v. In program synthesis we define v as a score function for programs in $[\![G]\!]$. A score function used in the program synthesis literature is the following.

$$C(p, D) = e^{-\beta J(p,D)}$$

where β is a constant usually set to 0.5. If the number of errors $J(p, D)$ is large, then the score C is small. Conversely, if $J(p, D)$ is small, then the score C is large.

Here is the general idea for using MH for synthesizing programs.

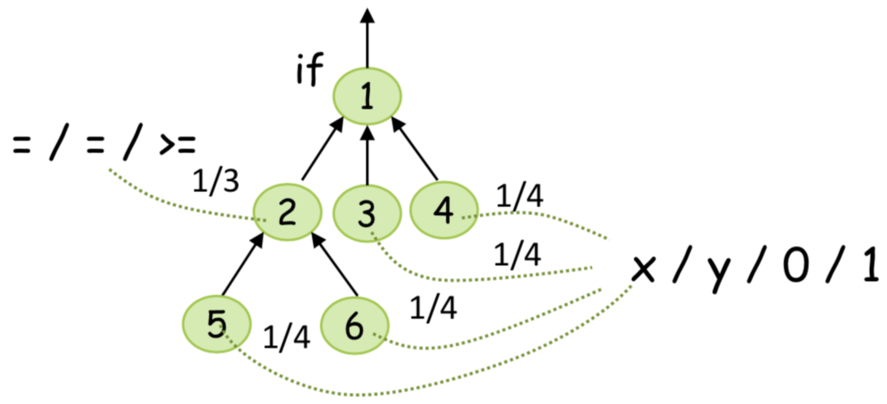
1. We will use the MH algorithm to sample programs p according to function C.
2. The proposal function is given by a tree representation of programs in the grammar G. We initially sample a program p_0 of fixed size (e.g., 6 symbols) and iteratively sample other programs p_i by editing one node of a tree defining current program p_{i-1}.
3. Program p_i is accepted by probability $\min\{1, C(p_i, D)/C(p_{i-1}, D)\}$.

1.2.1 Example

Let's consider an example, which is modified from an example from the paper 'Program Synthesis' by S. Gulwani, O. Polozov, and R. Singh. The grammar below allows simple operations.

$$\begin{aligned} S &\rightarrow x \mid y \mid 0 \mid 1 \mid (S + S) \mid \text{if } B \text{ then } S \text{ else } S \\ B &\rightarrow (S \leq S) \mid (S == S) \mid (S \geq S) \end{aligned}$$

The specification is given by the following set of training input-outputs $D = \{[(2, 1), 2], [(-1, 1), 1], [(-3, 3), 3]\}$ that were drawn from a max function. The synthesizer constructs trees representing all possible programs of a given size n. Here we assume $n = 6$, which gives the following tree.



The root of the tree specifies an if-clause with three children: S symbol for the true case, operator, S symbol for the false case. The operator has two children with one symbol S each. We also show in the figure the possible values that each node in the tree can assume. In total there are $4 \times 4 \times 4 \times 4 \times 3 = 768$ possible programs that can be generated from the tree.

The tree defines the proposal function and the initial program p_0 can be sampled by randomly sampling values for the nodes. Let's assume that $p_0 = if(y \leq 0, y, x)$. Then, $J(p_0, D) = 2$ and $C(p_0, D) = e^{-0.5\times 2} = 0.37$.

The next program to be sampled can be $p_1 = if(x \leq 0, y, x)$ which is obtained by randomly switching the value of node 5 in the tree. Here we have that $J(p_1, D) = 0$, $C(p_1, D) = e^{-0.5\times 0} = 1$, and $\alpha(p_1, p_0) = \min\{1, 2.70\} = 1$ so the algorithm accepts the new program, which is a solution to the problem.

A problem with the approach just described is that one doesn't know a priori the size of the program that will solve the problem. One strategy is to allow with some probability (e.g. 1%) that the size n of the program is increased.

1.3 Simulated Annealing

The MH algorithm searches by sampling programs according to function C. Although MH will sample programs with larger C values with higher probability, it will still sample programs with small C values. A small modification in the MH algorithm allows it to focus its search around programs with larger C value.

Instead of using the acceptance function $\min\{1, \frac{C(p_i, D)}{C(p_{i-1}, D}\}$, we use the following function,

$$\min\left\{1, \frac{C^{\frac{1}{T_i}}(p_i, D)}{C^{\frac{1}{T_i}}(p_{i-1}, D}\right\}$$

where T_i is a temperature parameter at iteration i. The temperature is reduced according to a schedule. If T_i is small, then it is much less likely that the algorithm will accept a program p_{i-1} that has lower score than p_i. This algorithm is known as Simulated Annealing. In the beginning of search, with large temperatures, the algorithm accepts more easily programs with lower scores. This allows the algorithm to explore the space. Eventually the search becomes more focused on more promising parts of the space.