Lecture 4
Probabilistic Models and Stochastic Search

Nadia Polikarpova

Logistics

Project topics

- Once you have decided on the topic, put it on the Google sheet next to any of the team members
- If you haven’t decided, talk to me

Project proposals

- Due next Friday (Oct 20)
- Upload to the Proposals directory inside the shared Google folder
- Can be a Google Doc or a PDF
- File name must be “Team-N”, where N is your team ID

Announcement

Consider applying to the *Programming Languages Mentoring Workshop* (Jan 9, Los Angeles, CA)

https://popl18.sigplan.org/track/PLMW-POPL-2018

Enumerative search

Explores smaller programs before larger programs

- Small solution is likely to generalize
- Scales poorly with the size of the smallest solution

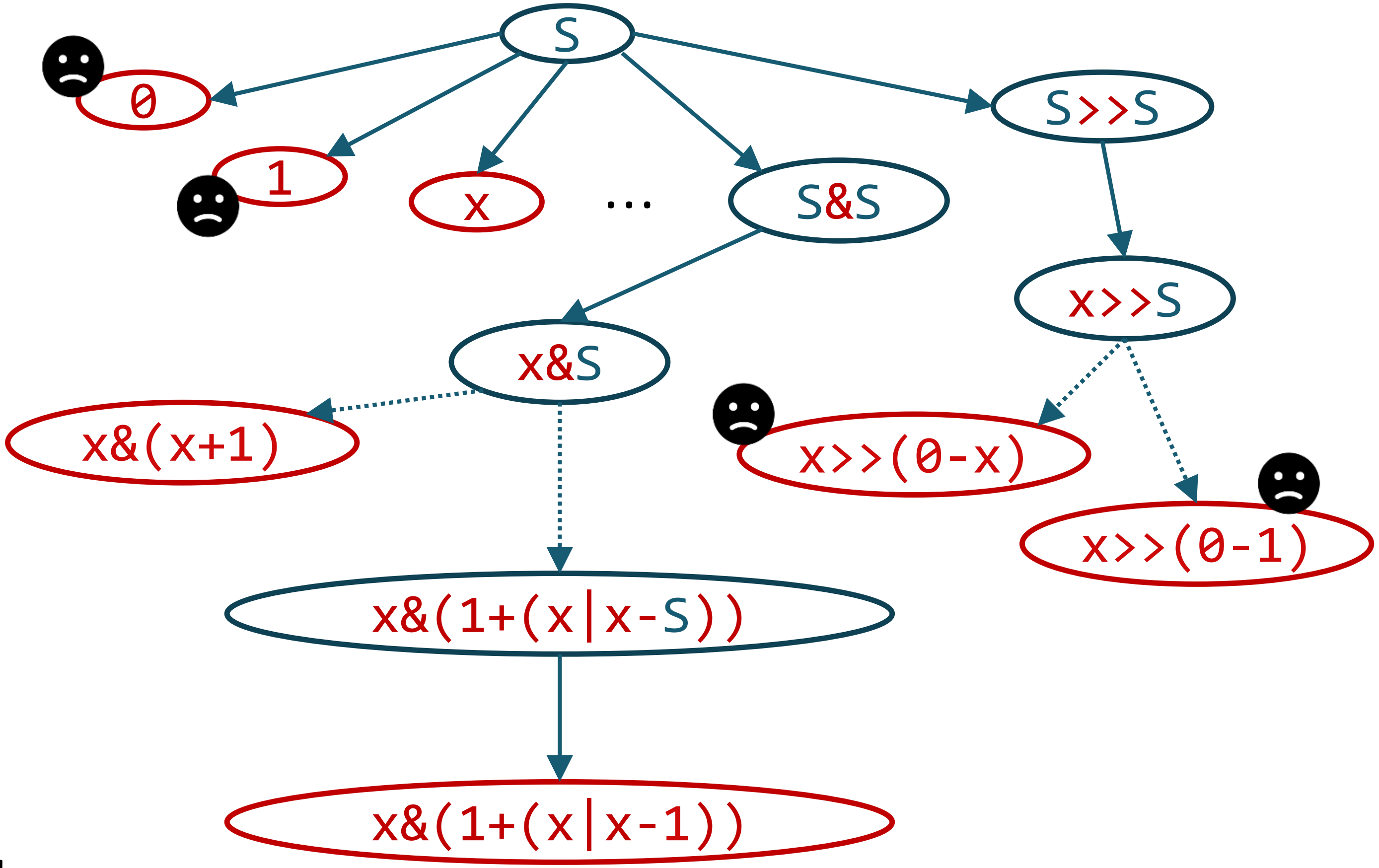
Top-down search (revisited)

Turn off the rightmost sequence of **1**s:

```
00101 → 00100
01010 → 01000
10110 → 10000
```

```
S ->  0 | 1 | x |
      S + S     |
      S - S     |
      S & S     |
      S | S     |
      S << S    |
      S >> S
```

Explores many unlikely programs!



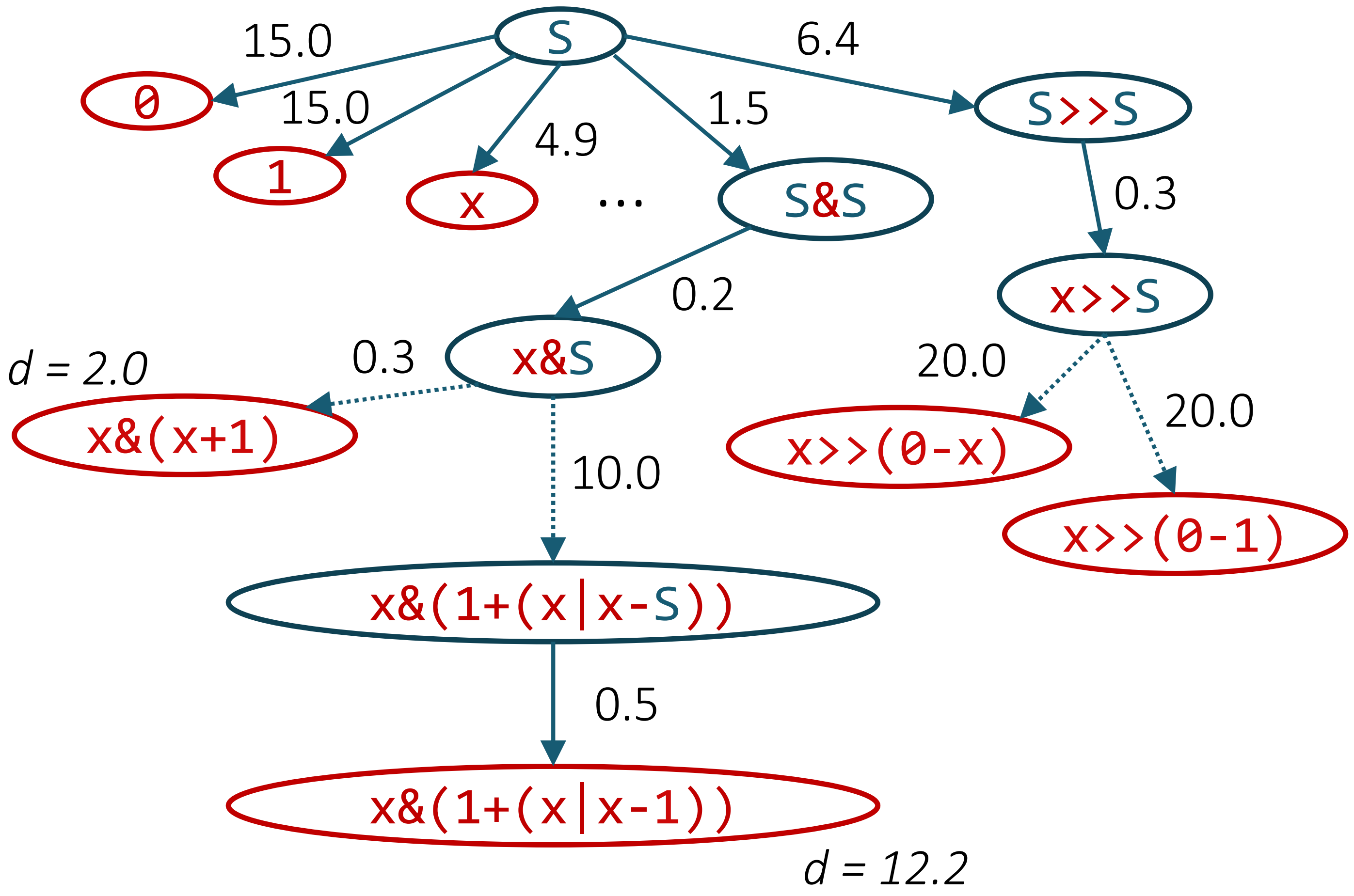
Weighted top-down search

Idea: explore programs in the order of likelihood, not size

1. Assign weights *w(e)* to edges such that *d(p)* < *d(p')* iff *p* is more likely than *p'*

$$d(p) = \sum_{e \in S \to p} w(e)$$

2. Use Dijkstra's algorithm to find closest leaves



Weighted top-down search (Dijkstra)

```
top-down(<T, N, R, S>, [i → o]) {
  P := [<S,0>]
  while (P != [])
    <p,d> := P.dequeue_min(d);
    if (ground(p) && p([i]) = [o])
      return p;
    P.enqueue(unroll(p,d));
}

unroll(p,d) {
  P’ := []
  N := leftmost nonterminal in p
  forall (N ::= rhs in R)
    P’ += <p[N -> rhs], d + w(rhs, p)>
  return P’;
}
```

P now stores candidates (nodes) together with their distances

Dequeue the node with the shortest distance from the root

Distance to a new node: add the *w(e)*

Weighted top-down search (A*)

```
top-down(<T, N, R, S>, [i → o]) {
  P := [<S,0,h(S)>]
  while (P != [])
    <p,d,h> := P.dequeue_min(d + h);
    if (ground(p) && p([i]) = [o])
      return p;
    P.enqueue(unroll(p,d));
}

unroll(p,d) {
  P' := []
  N := leftmost nonterminal in p
  forall (N ::= rhs in R)
    P' += <p[N -> rhs], d + w(rhs, p),
                         h(p[N -> rhs])>
  return P';
}
```

Dijkstra: explores a lot of intermediate nodes that don't lead to any cheap leaves

A*: introduce heuristic function *h(p)* that estimates how close we are to the closest leaf

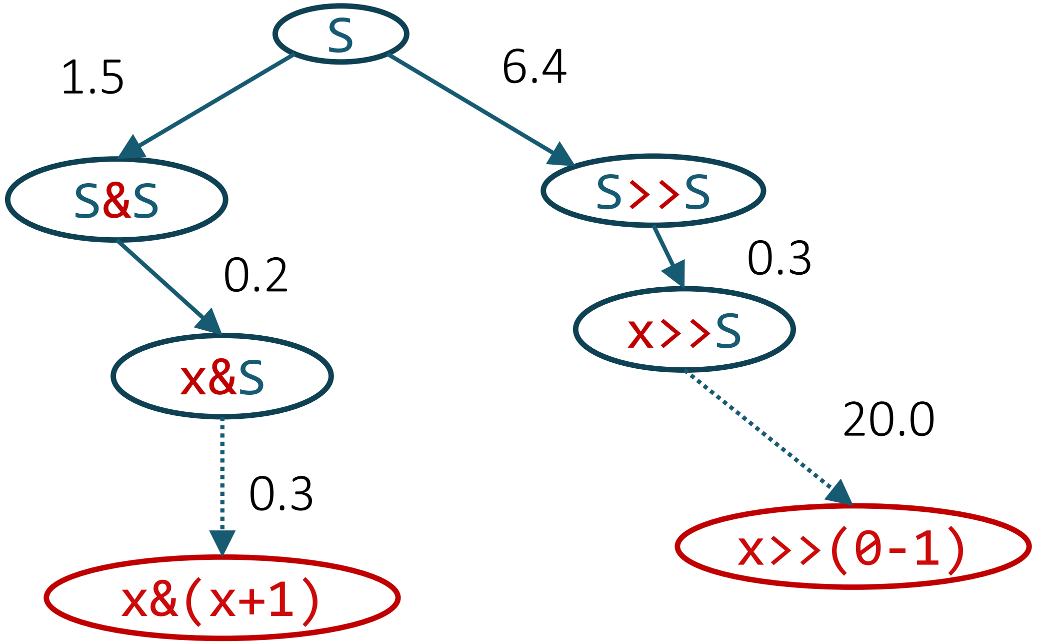
So, where does this come from?

Assigning weights to edges

$$d(p) = \sum_{e \in S \to p} w(e)$$

$$2^{-d(p)} = \prod_{e \in S \to p} 2^{-w(e)}$$

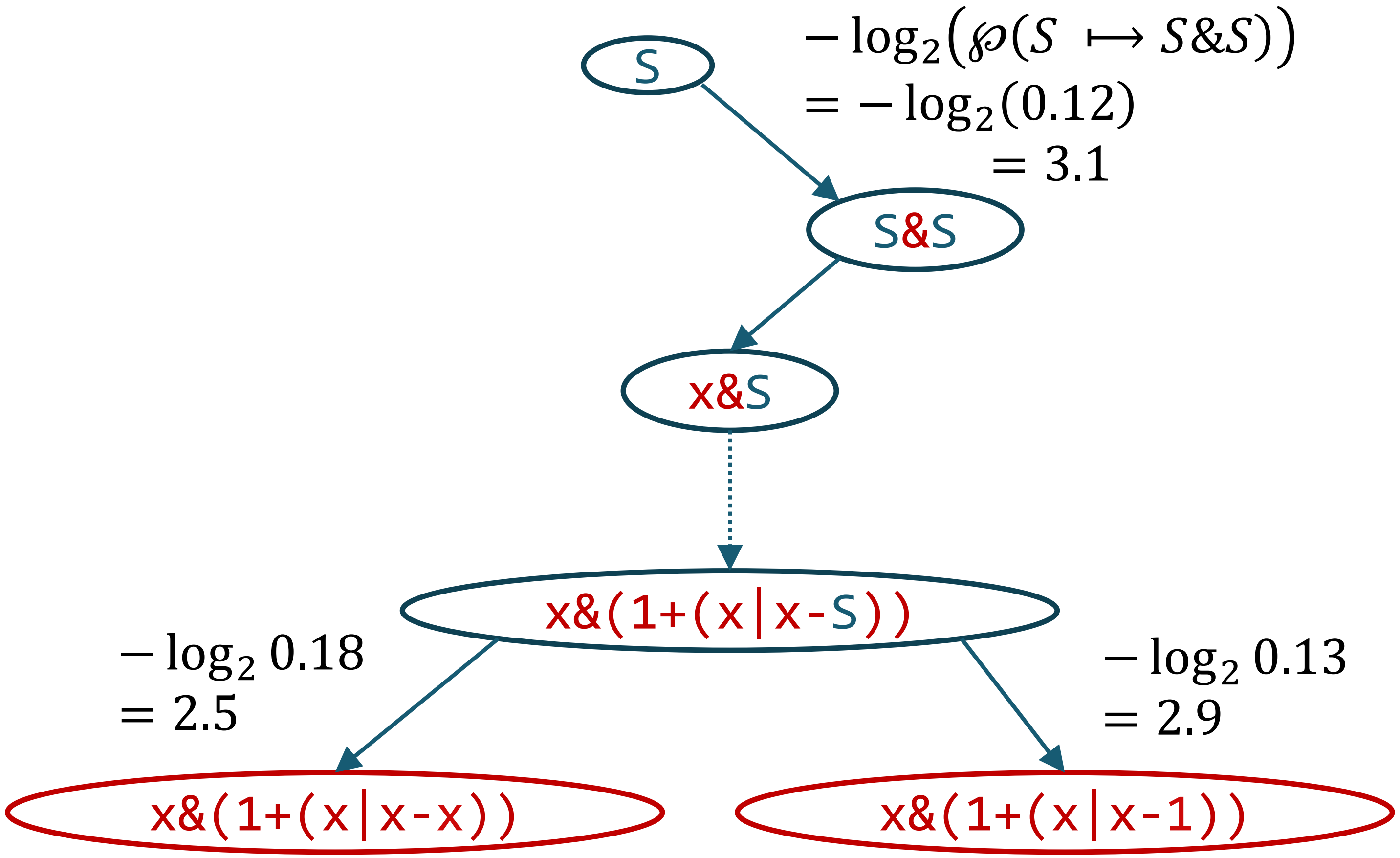
$$\wp(p) = \prod_{e \in S \to p} \wp(e)$$



So, we should decide what is the probability of taking each edge $\wp(e)$ and then set $w(e) = -\log_2 \wp(e)$

Probabilistic CFG (PCFG)

Rule	$\wp$
`S -> 0`	0.13
`S -> 1`	0.13
`S -> x`	0.18
`S -> S + S`	0.11
`S -> S - S`	0.11
`S -> S & S`	0.12
`S -> S | S`	0.12
`S -> S << S`	0.05
`S -> S >> S`	0.05

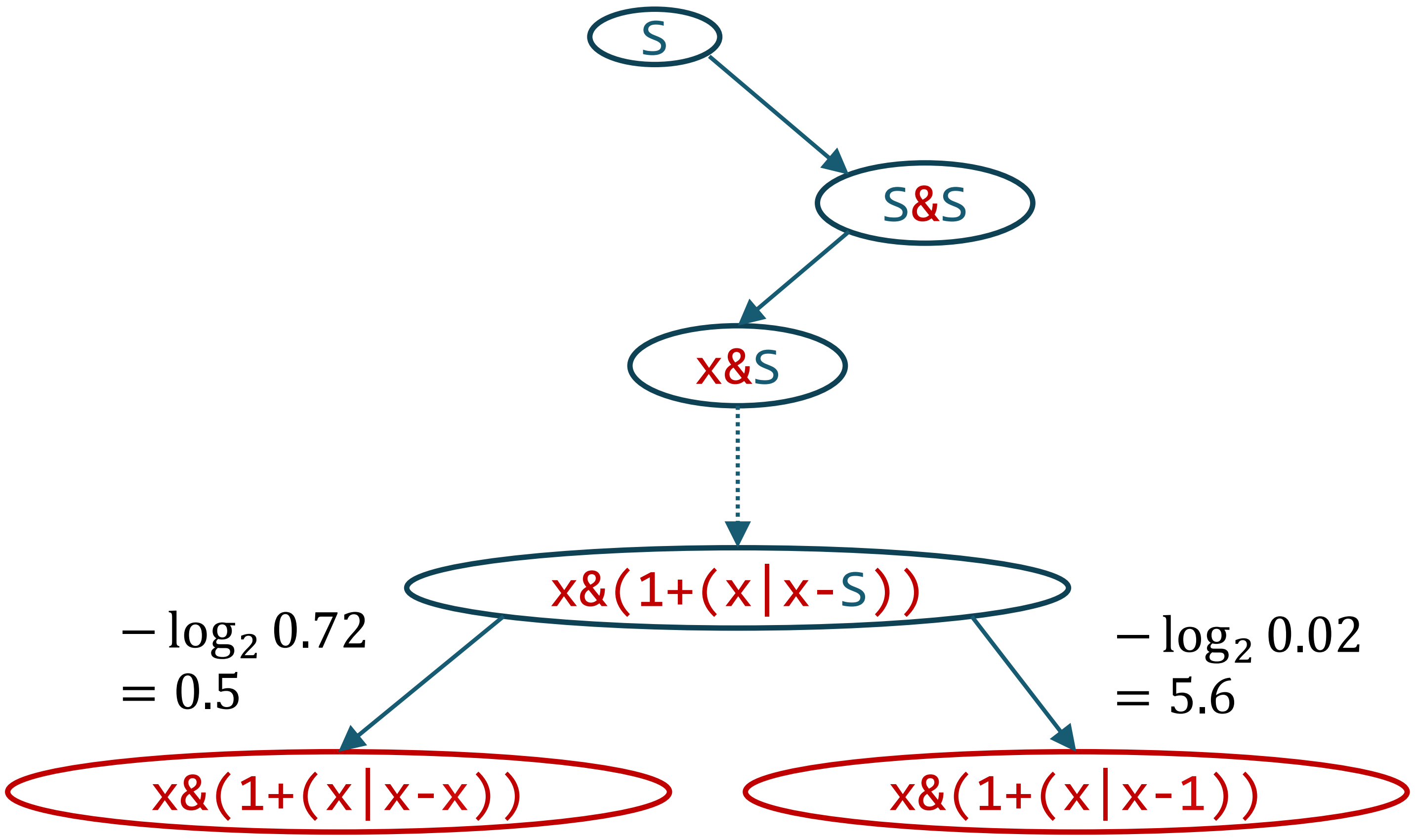


Probabilistic Higher-Order Grammar (PHOG)

[Bielik, Raychev, Vechev ‘16]

```
N[context] -> rhs
```

		$\wp$
`S[x,-] ->`	`1`	0.72
`S[x,-] ->`	`x`	0.02
`S[x,-] ->`	`S + S`	0.12
`S[x,-] ->`	`S - S`	0.12
`...`		
`S[1,+] ->`	`1`	0.26
`S[1,+] ->`	`x`	0.25
`S[1,+] ->`	`S + S`	0.19
`S[1,+] ->`	`S - S`	0.08



Learning PHOGs

[Bielik, Raychev, Vechev ‘16]

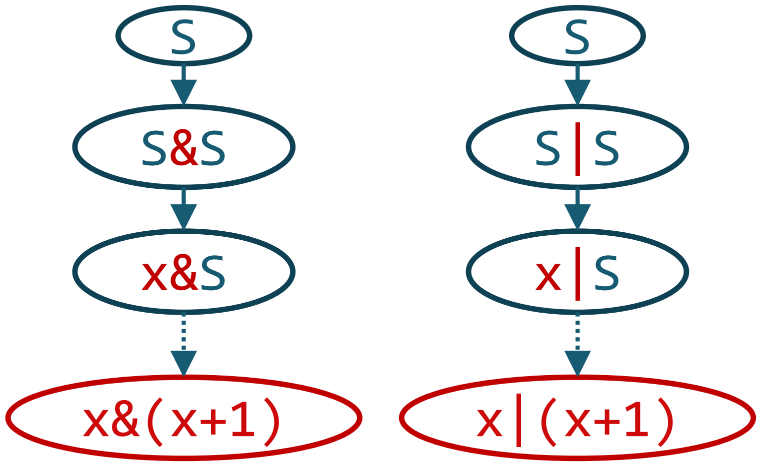
CFG +

Corpus

```
x&(x+1)
x|(x-1)
x
x&(x+x)
x&(1+(x|x-1))
...
```

parse →

ASTs / Paths



…

learn → `context`, ℘

PHOGs useful for:

- code completion
- deobfuscation
- programming language translation
- statistical bug detection

Probabilistic models: overview

Learn natural programs

Learn solutions for particular problem

- useful for MOOCs

Learn mapping from spec to code

- or features of code

sk_p

[Pu et al. '16]

Program corrections for MOOCs

Treats programs as text

- Modulo concrete variable names etc.
- Uses the skipgram model to predict which statement is most likely to occur between the two

Features

- Can repair syntax errors

Limitations

- Needs all algorithmically distinct solutions to appear in the training set

DeepCoder

[Balog et al. ‘17]

Answer to *neural programming*: neural nets that write programs

Predicts likely components from IO examples:

```
[-17 -3 4 11 0 -5 -9 13 6 6 -8 11]
→ [-12 -20 -32 -36 -68]
```

`*4`	(1.0)	`filter`	(1.0)
`>0`	(1.0)	`sort`	(1.0)
`map`	(1.0)	`reverse`	(0.7)

Features

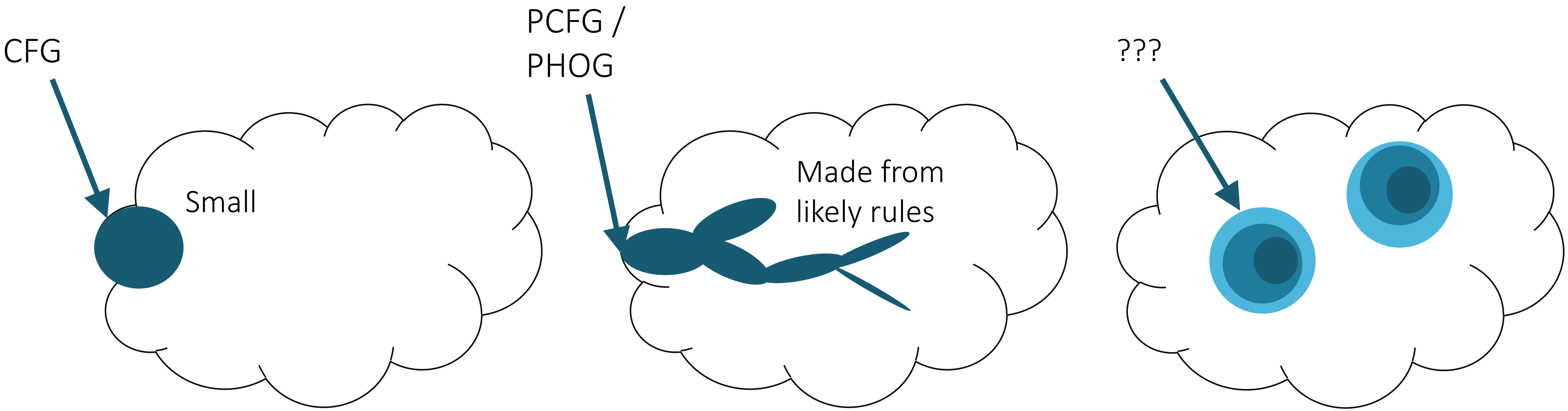
- Can be combined with any enumerative search
- Significant speedups for a small list DSL

Limitations

- Unclear whether it scales to larger DSLs or more complex data structures

Stochastic search

Search space



Enumerative search

Weighted
enumerative search

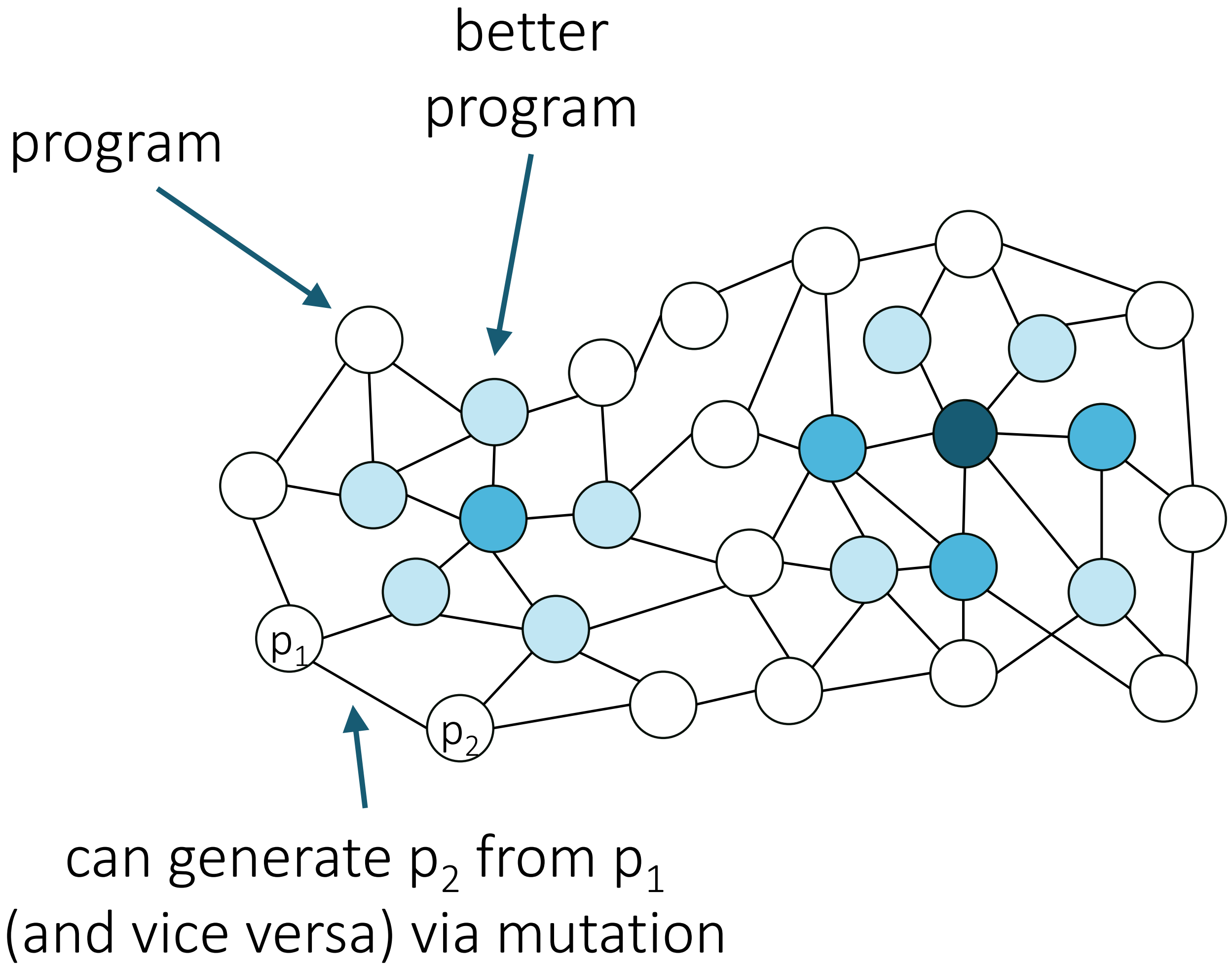
MCMC sampling!

Search by hill climbing

To find the best program:

```
p := random()
while (true) {
  p' := mutate(p);
  if (cost(p') < cost(p))
    p := p';
}
```

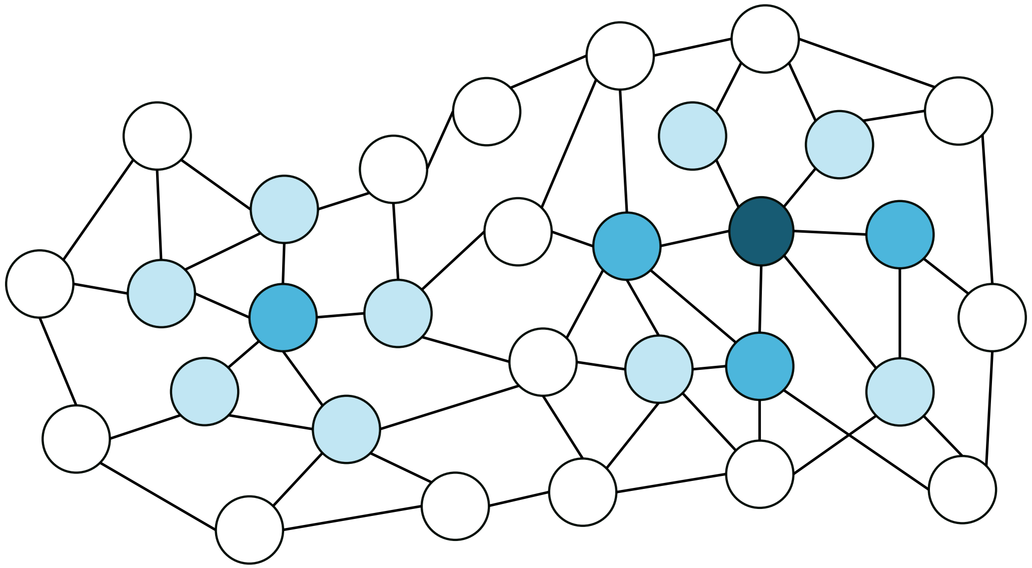
Will never get to from p_1!



MCMC sampling

Avoid getting stuck in local minima:

```
p := random()
while (true) {
  p' := mutate(p);
  if (random(A(p,p'))
    p := p';
}
```



$$A(p \to p') = \min(1, e^{-\beta * C(p')/C(p)})$$

MCMC sampling

Why did we pick this A?

$$A(p \to p') = \min(1, e^{-\beta * C(p')/C(p)})$$

The theory of Markov chains tells us that in the limit we will be sampling with the probability proportional to

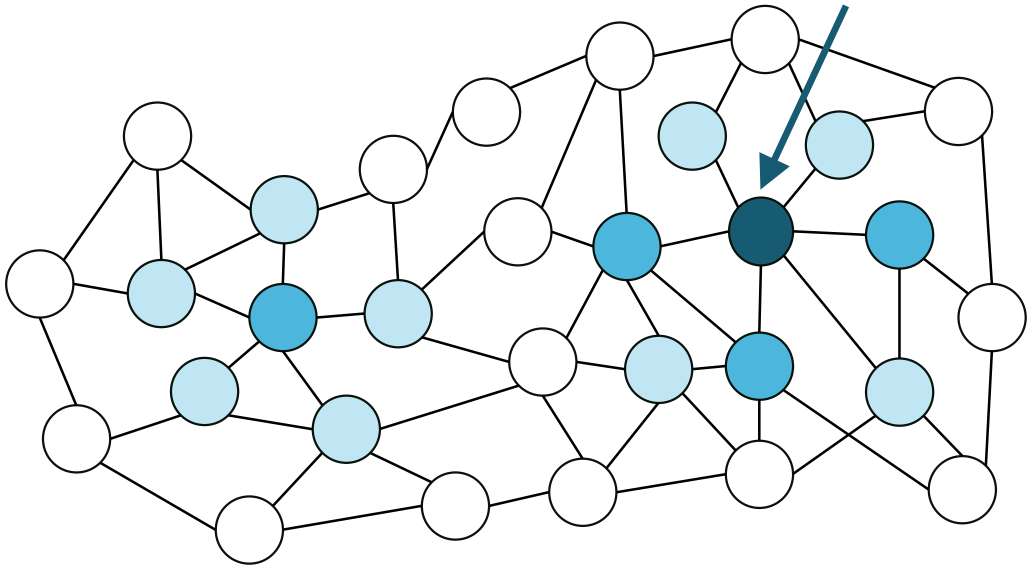
$$e^{-\beta * C(p)}$$

MCMC for superoptimization

[Schkufza, Sharma, Aiken ‘13]

```
.L0:
movq rsi, r9
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rdx, r9
imulq rsi, rdx
imulq rsi, rcx
addq rdx, rax
jae .L2
movabsq 0x100000000, rdx
addq rdx, rcx
.L2:
movq rax, rsi
movq rax, rdx
shrq 32, rsi
salq 32, rdx
addq rsi, rcx
addq r9, rdx
adcq 0, rcx
addq r8, rdx
adcq 0, rcx
addq rdi, rdx
adcq 0, rcx
movq rcx, r8
movq rdx, rdi
```

```
.L0:
shlq 32, rcx
movl edx, edx
xorq rdx, rcx
movq rcx, rax
mulq rsi
addq r8, rdi
adcq 0, rdx
addq rdi, rax
adcq 0, rdx
movq rdx, r8
movq rax, rdi
```



Cost function

$$C_s(p) = \mathrm{eq}_s(p) + \mathrm{perf}(p)$$

source program

penalty for wrong results

penalty for being slow

$$\mathrm{eq}_s(p)=\sum_{t \in Tests} \mathrm{reg}_s(p,t) + \mathrm{mem}_s(p,t) + \mathrm{err}(p,t)$$

# of different bits in registers/memory

# of segfaults etc

when $\mathrm{eq}_s(p) = 0$, use a symbolic validator

Cost function

$$C_s(p) = \mathrm{eq}_s(p) + \mathrm{perf}(p)$$

source program

penalty for wrong results

penalty for being slow

$$\mathrm{perf}(p) = \sum_{i \in \mathrm{instr}(p)} \mathrm{latency}(i)$$

Stochastic search: discussion

Hill climbing can explore larger spaces

Limitations?

- only applicable when there is a cost function that faithfully approximates correctness
- Counterexample: round to next power of two

Other examples of making programs incrementally “more correct”?

- Condition abduction!