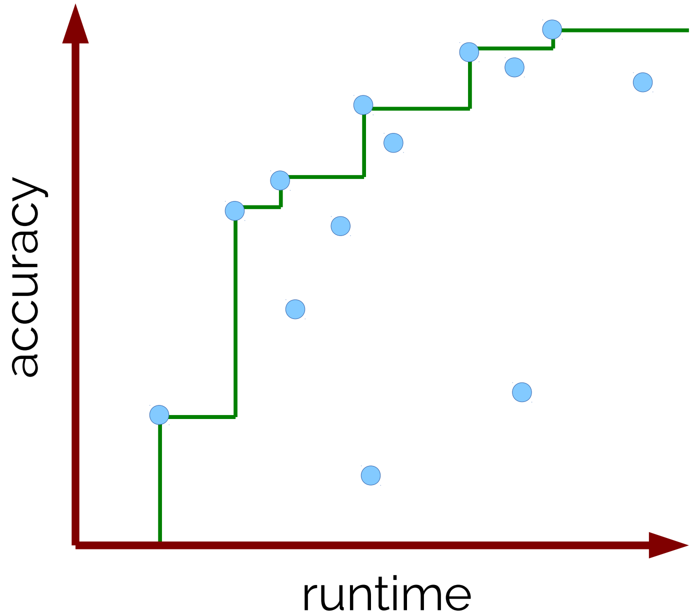
Learning to Prune: Exploring the Frontier of Fast & Accurate Inference

Tim Vieira & Jason Eisner
Johns Hopkins University

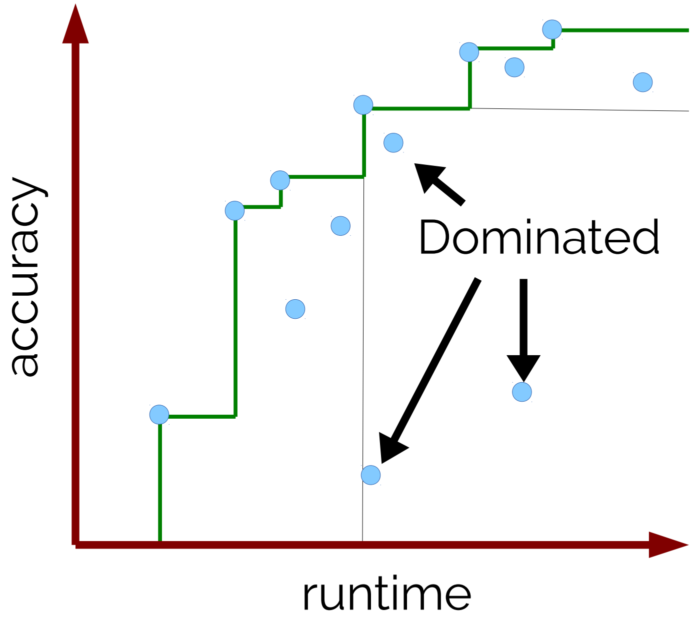
Learning to Prune:
Exploring the Frontier of Fast & Accurate Inference

Tim Vieira & Jason Eisner
Johns Hopkins University

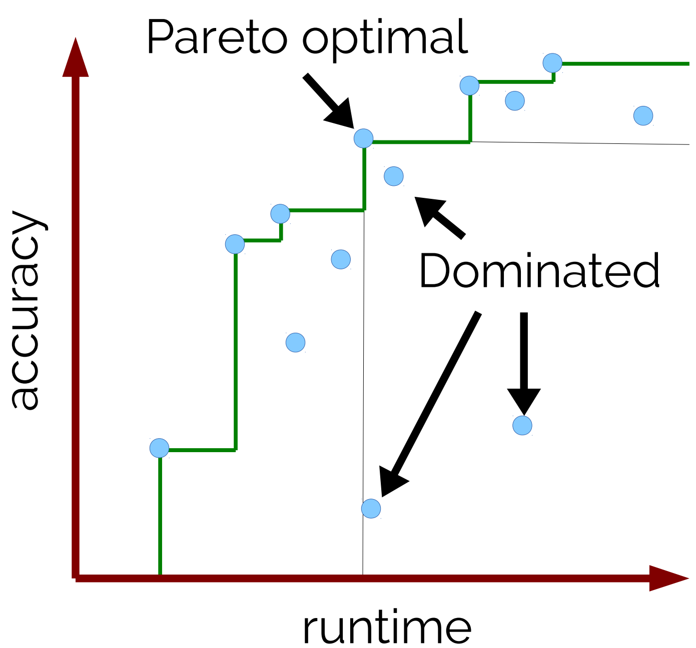
Exploring the Pareto Frontier of Fast & Accurate Inference



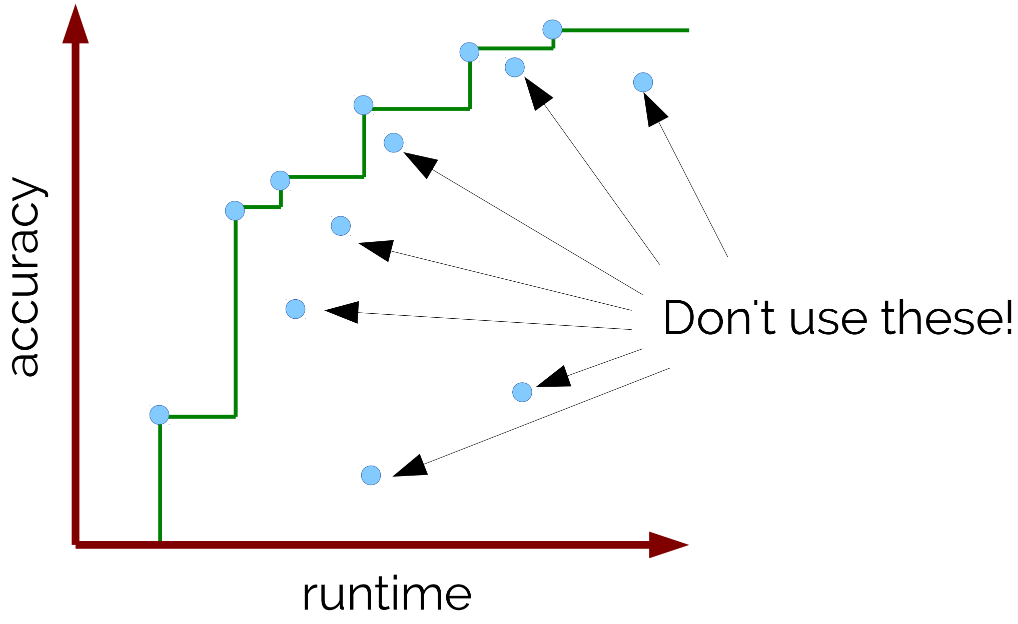
Exploring the Pareto Frontier of Fast & Accurate Inference



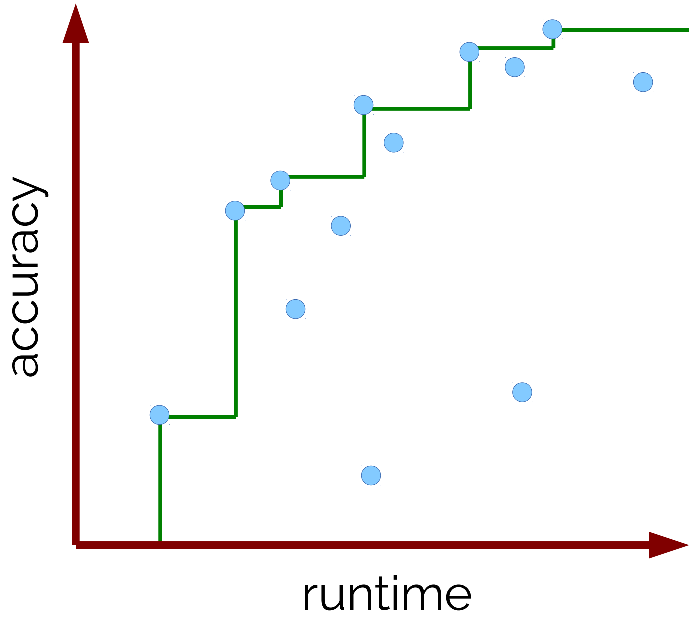
Pareto
Exploring the Frontier of Fast & Accurate Inference



Exploring the Pareto Frontier of Fast & Accurate Inference

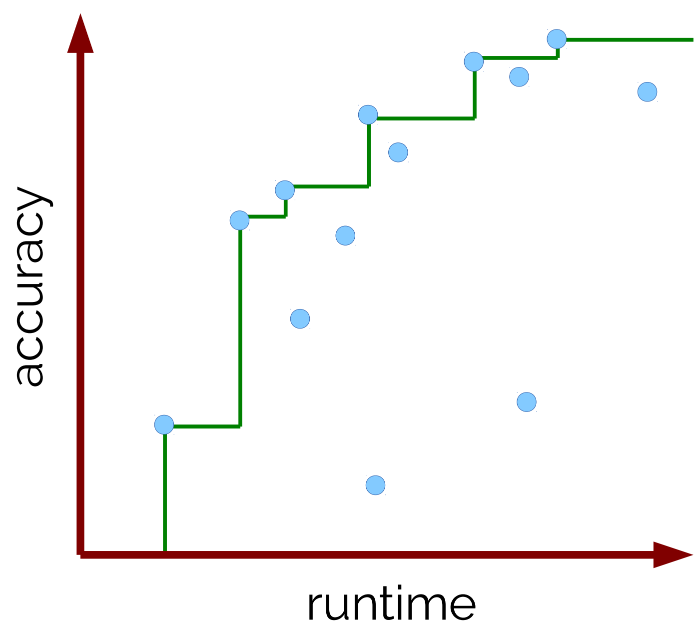


Exploring the Pareto Frontier of Fast & Accurate Inference



Search a space of approximate inference policies using machine learning

Exploring the Pareto Frontier of Fast & Accurate Inference



Search a space of approximate inference policies using machine learning

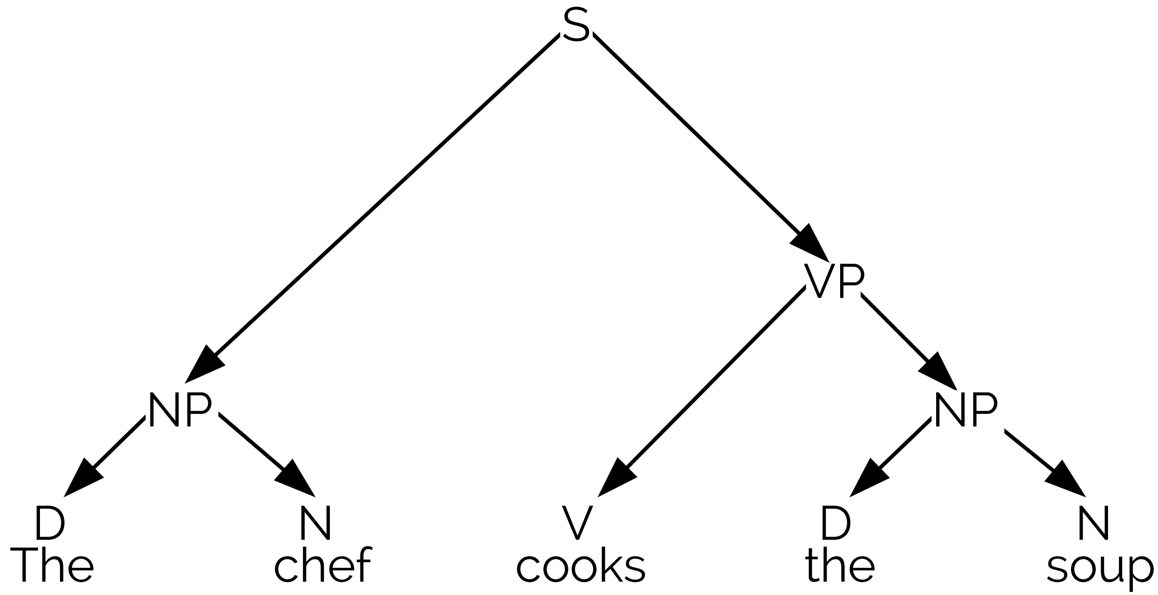
Want better algorithms? Use data, not just theory!

Outline

- Background
- Learning to prune
- Learning algorithm
- Making learning fast
  - Change propagation
  - Dynamic programming
- Results & conclusions

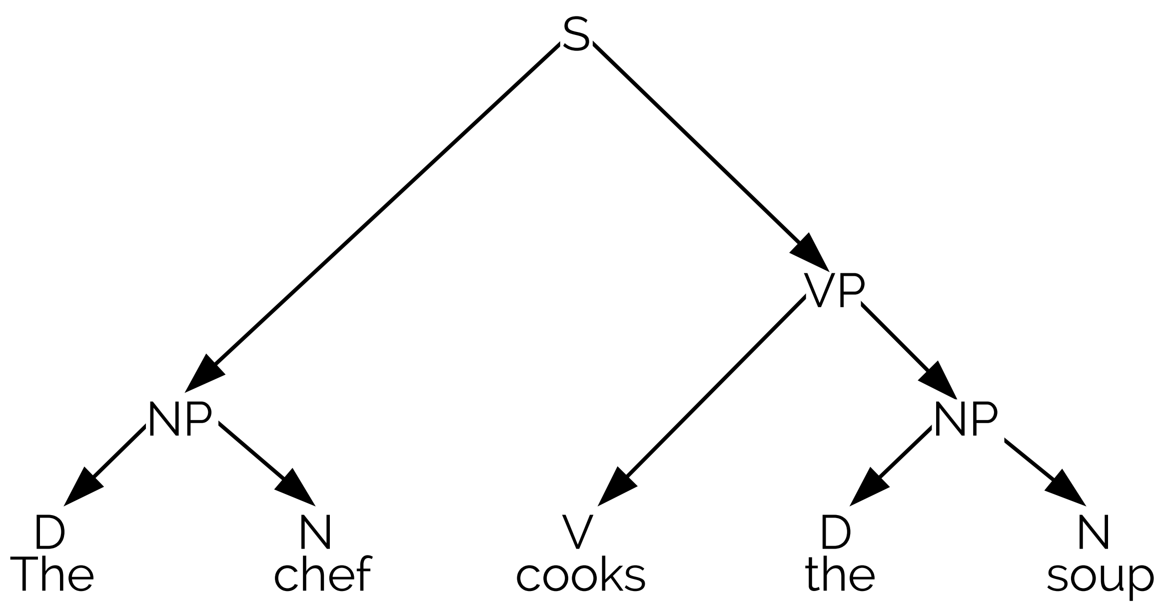
Parsing

"Diagramming sentences"



Parsing

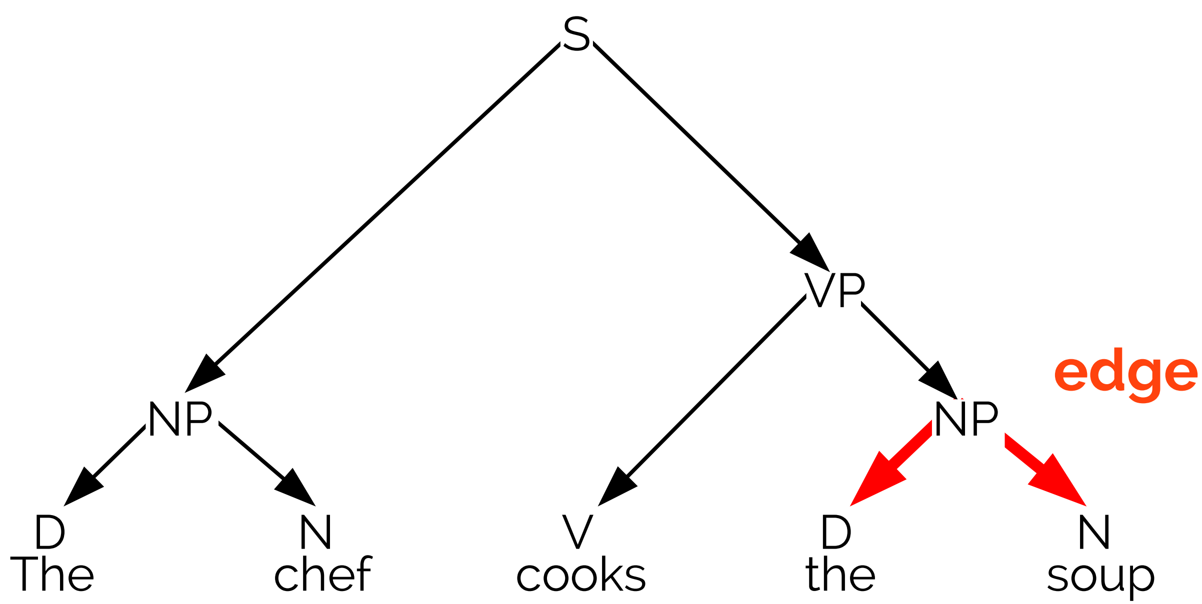
What is a good derivation for this sentence?



p(d) =

Parsing

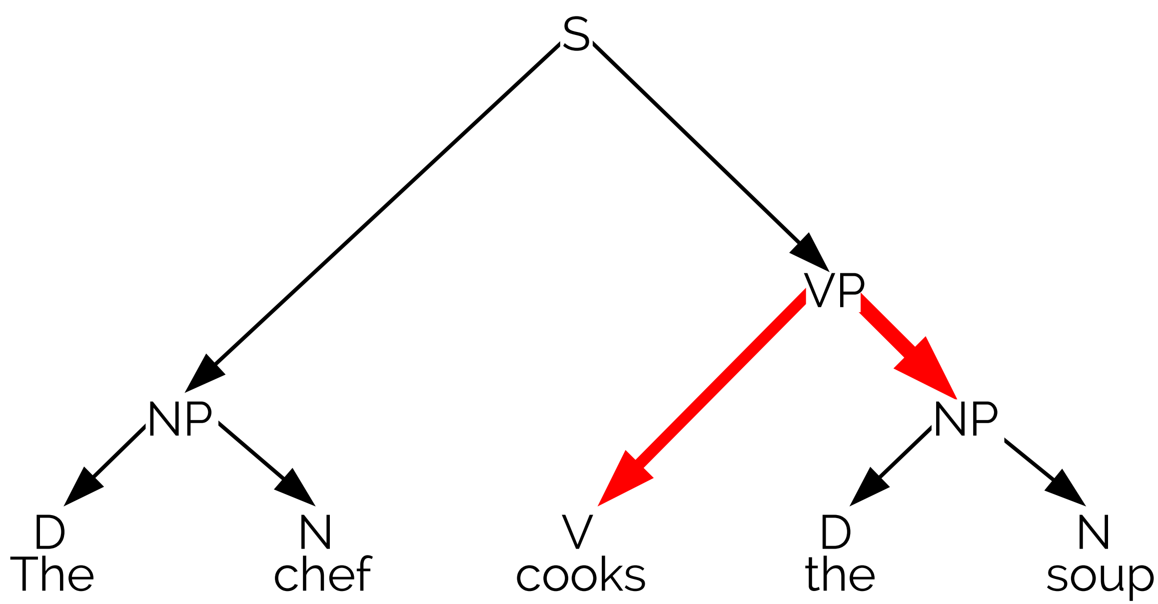
What is a good derivation for this sentence?



p(d) = g(NP → D N)

Parsing

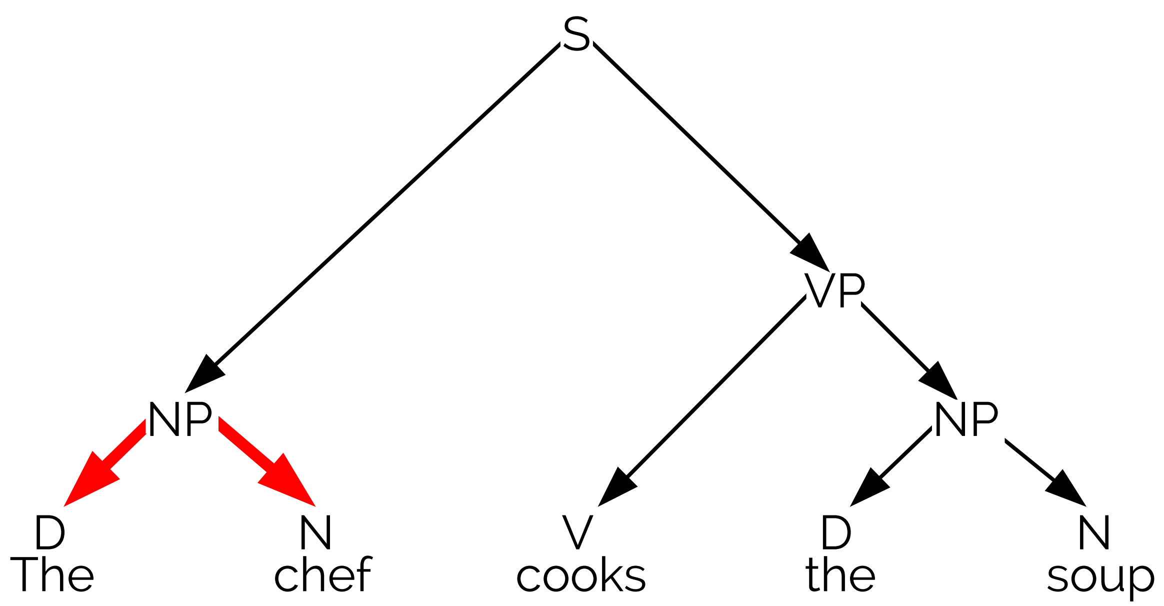
What is a good derivation for this sentence?



p(d) = g(NP → D N) * g(VP → V NP)

Parsing

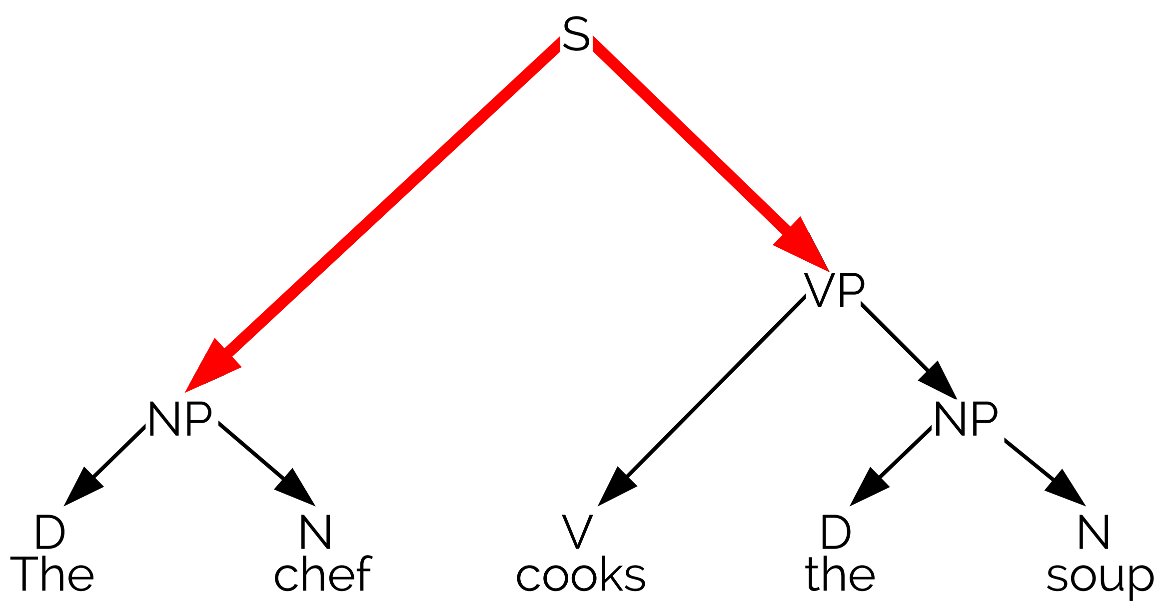
What is a good derivation for this sentence?



p(d) = g(NP → D N) * g(VP → V NP) * g(NP → D N)

Parsing

What is a good derivation for this sentence?

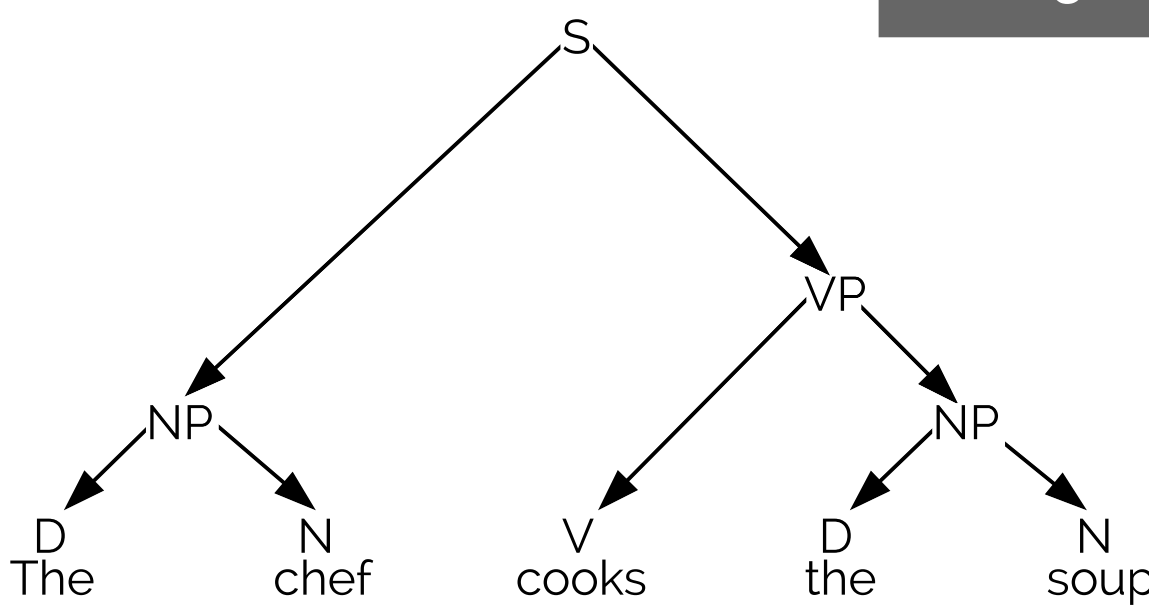


$p(d) = g(NP \rightarrow D\ N) * g(VP \rightarrow V\ NP) * g(NP \rightarrow D\ N) * g(S \rightarrow NP\ VP)$

Parsing

What is a good derivation for this sentence?

Scoring function is learned.



p(d) = g(NP → D N) * g(VP → V NP) * g(NP → D N) * g(S → NP VP)

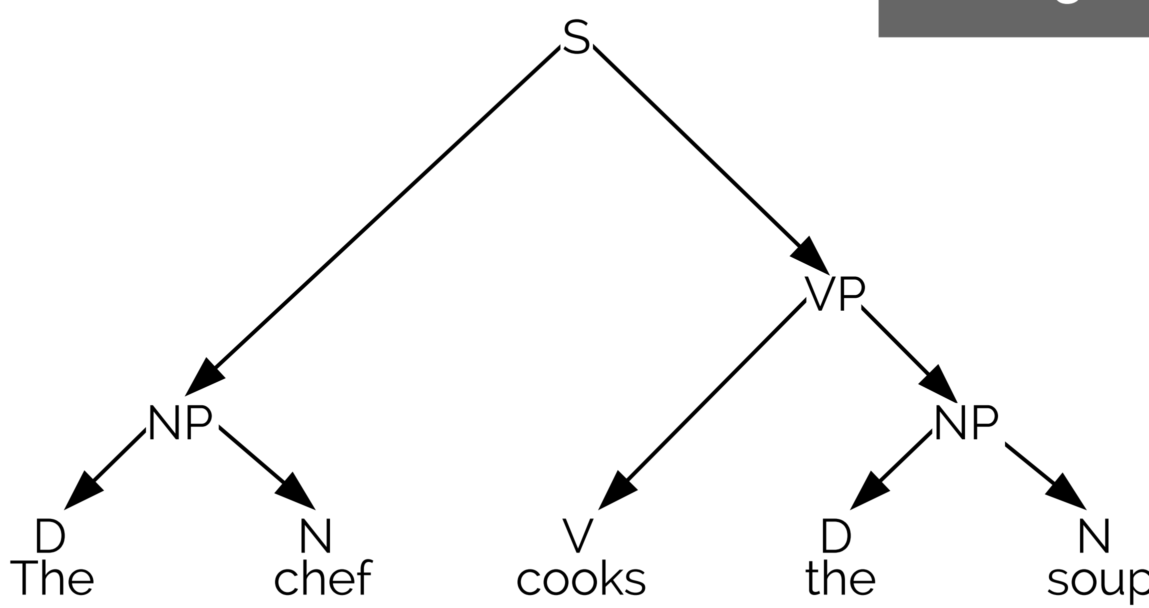
Parsing

What is a good derivation for this sentence?

Scoring function is learned.

Grammar

S → NP VP
VP → V NP
NP → D N
D → the
V → cooks
N → chef
N → soup



p(d) = g(NP → D N) * g(VP → V NP) * g(NP → D N) * g(S → NP VP)

Parsing

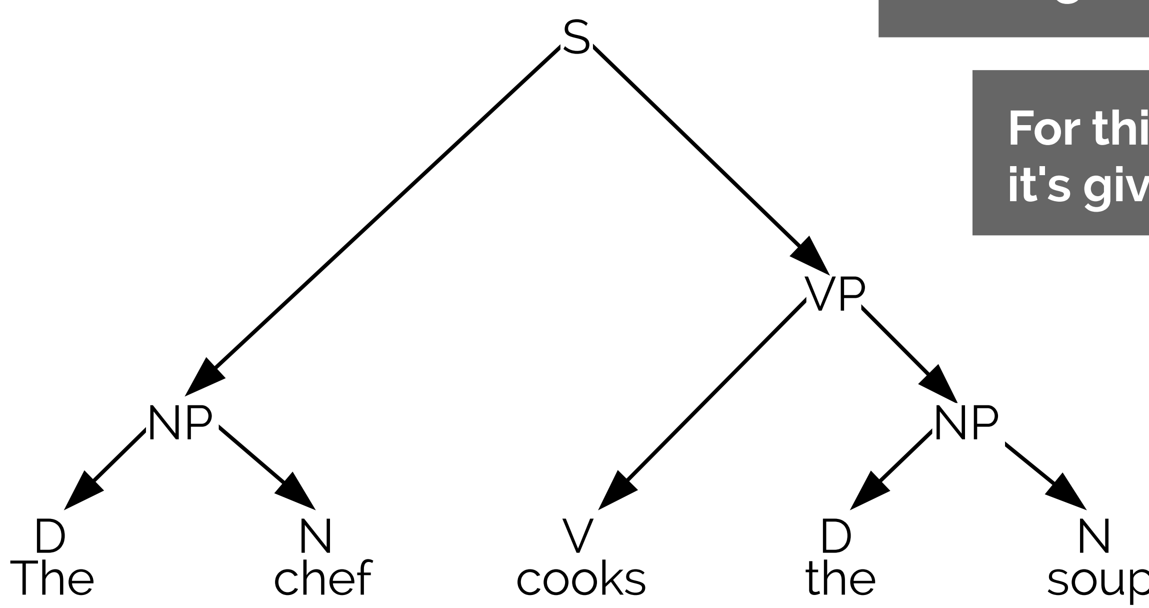
What is a good derivation for this sentence?

Scoring function is learned.

For this talk, I'll assume it's given.

Grammar

S → NP VP
VP → V NP
NP → D N
D → the
V → cooks
N → chef
N → soup



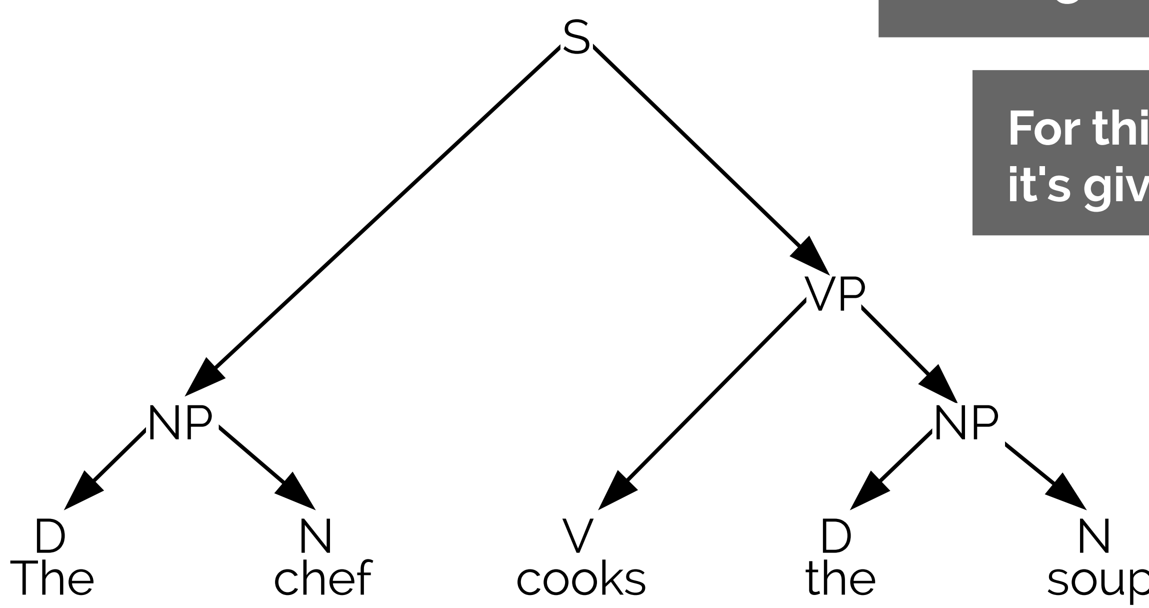
p(d) = g(NP → D N) * g(VP → V NP) * g(NP → D N) * g(S → NP VP)

Parsing

What is a good derivation for this sentence?

Grammar

S → NP VP
VP → V NP
NP → D N
D → the
V → cooks
N → chef
N → soup



Scoring function is learned.

For this talk, I'll assume it's given.

Written more generally, product of edge weights

$$p(d) = \prod_{e \in d} k_e$$

Parsing

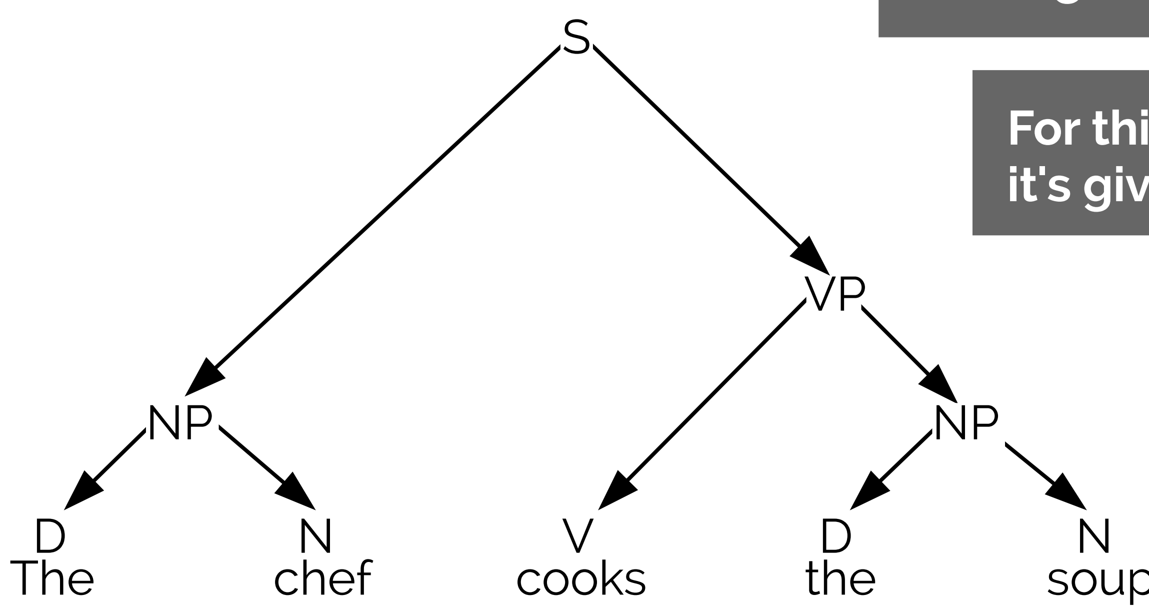
What is a good derivation for this sentence?

Scoring function is learned.

For this talk, I'll assume it's given.

Grammar

S → NP VP
VP → V NP
NP → D N
D → the
V → cooks
N → chef
N → soup

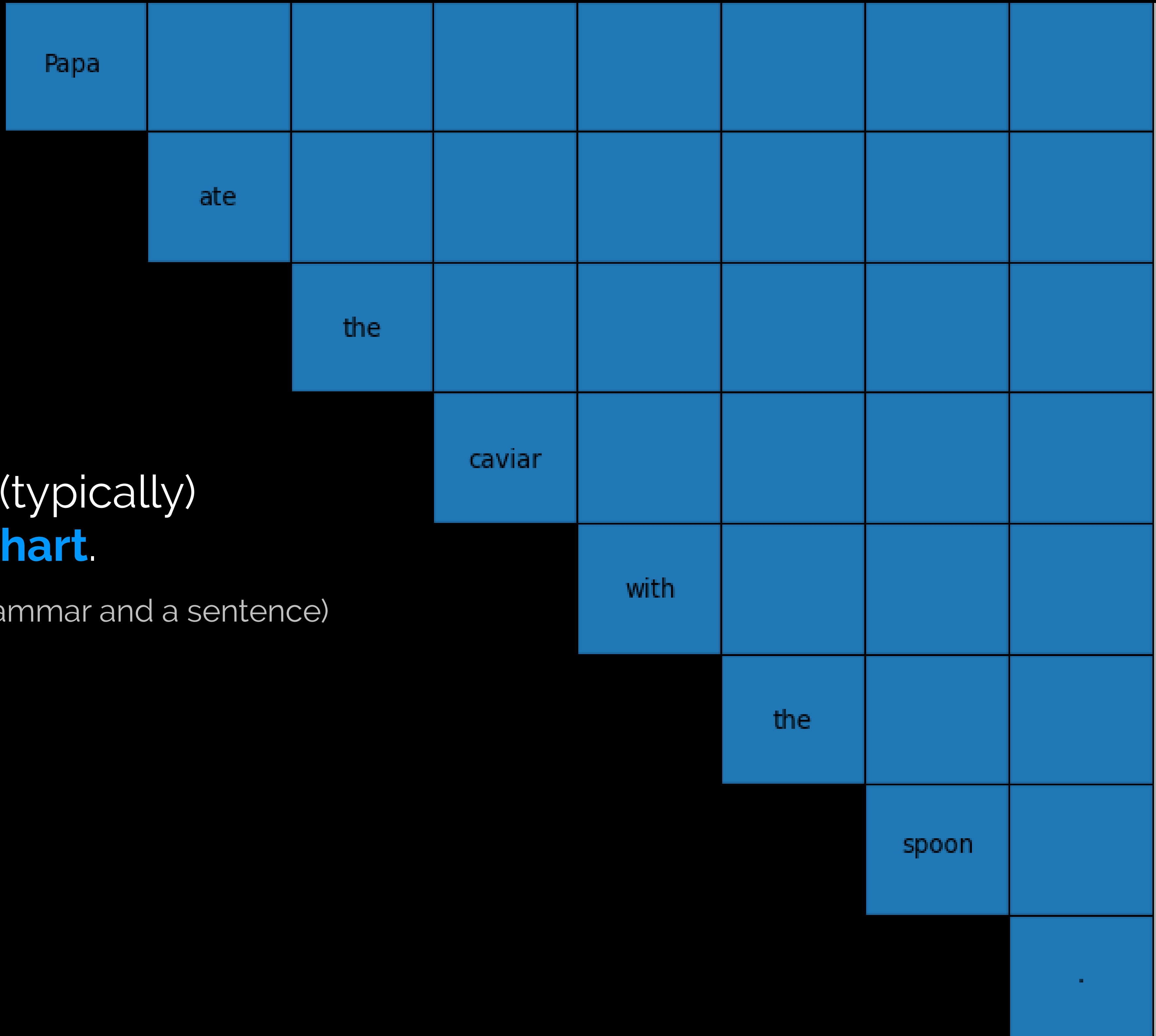


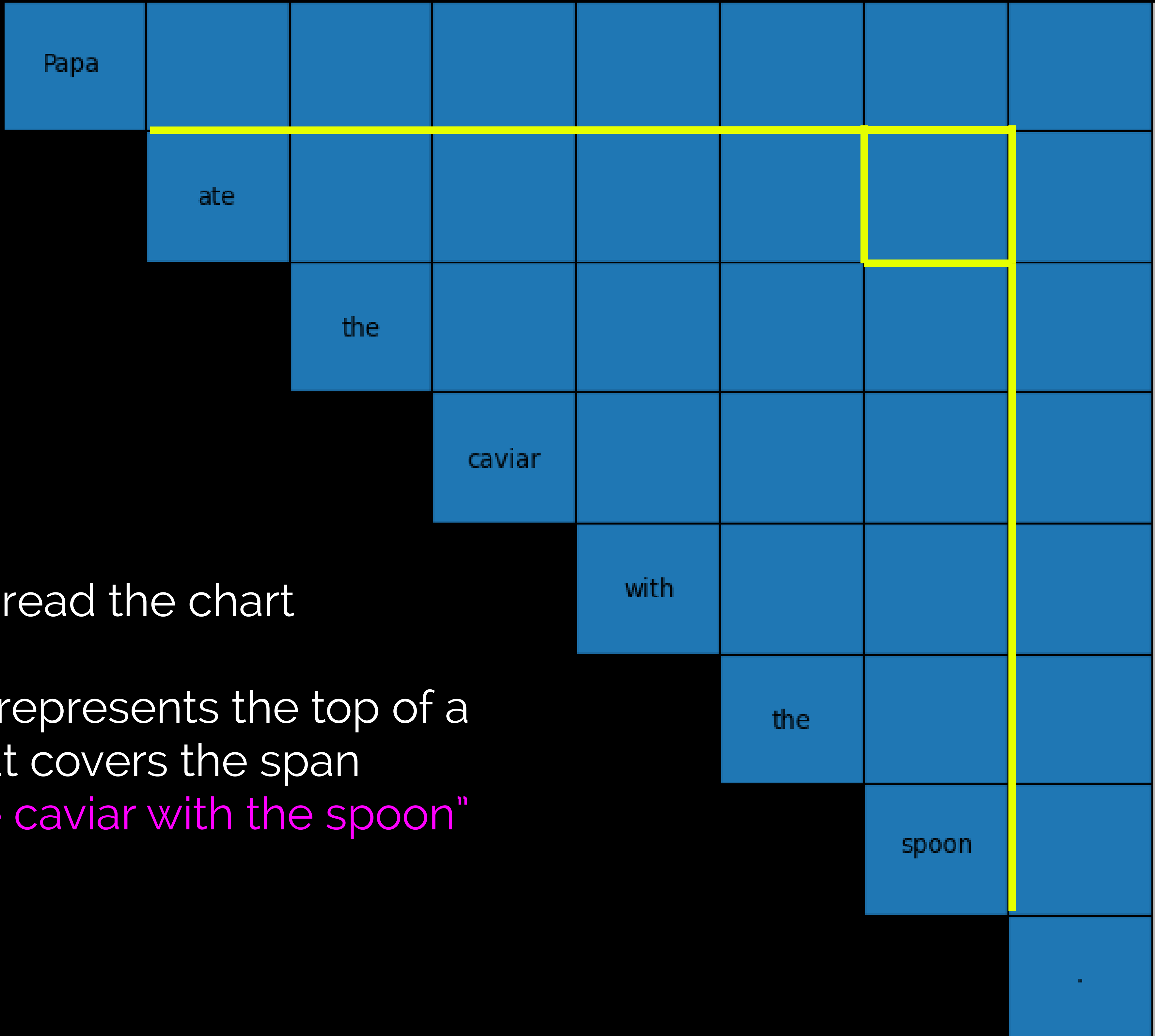
$$\underset{d \in D}{\operatorname{argmax}}\, p(d) = \underset{d \in D}{\operatorname{argmax}} \prod_{e \in d} k_e$$

How does a parser work?

Parsers (typically) fill in a **chart**.

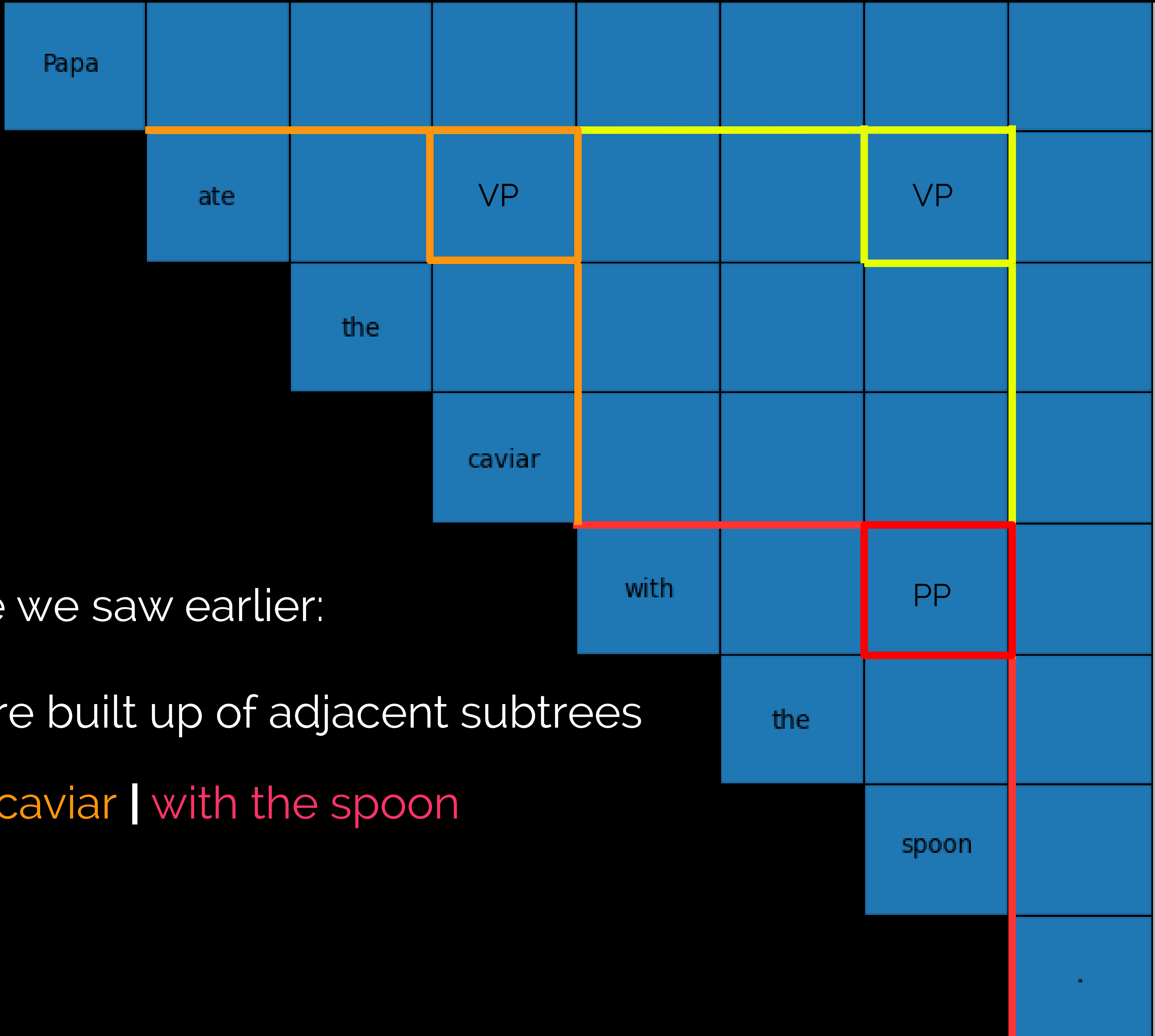
(Given a grammar and a sentence)





How to read the chart

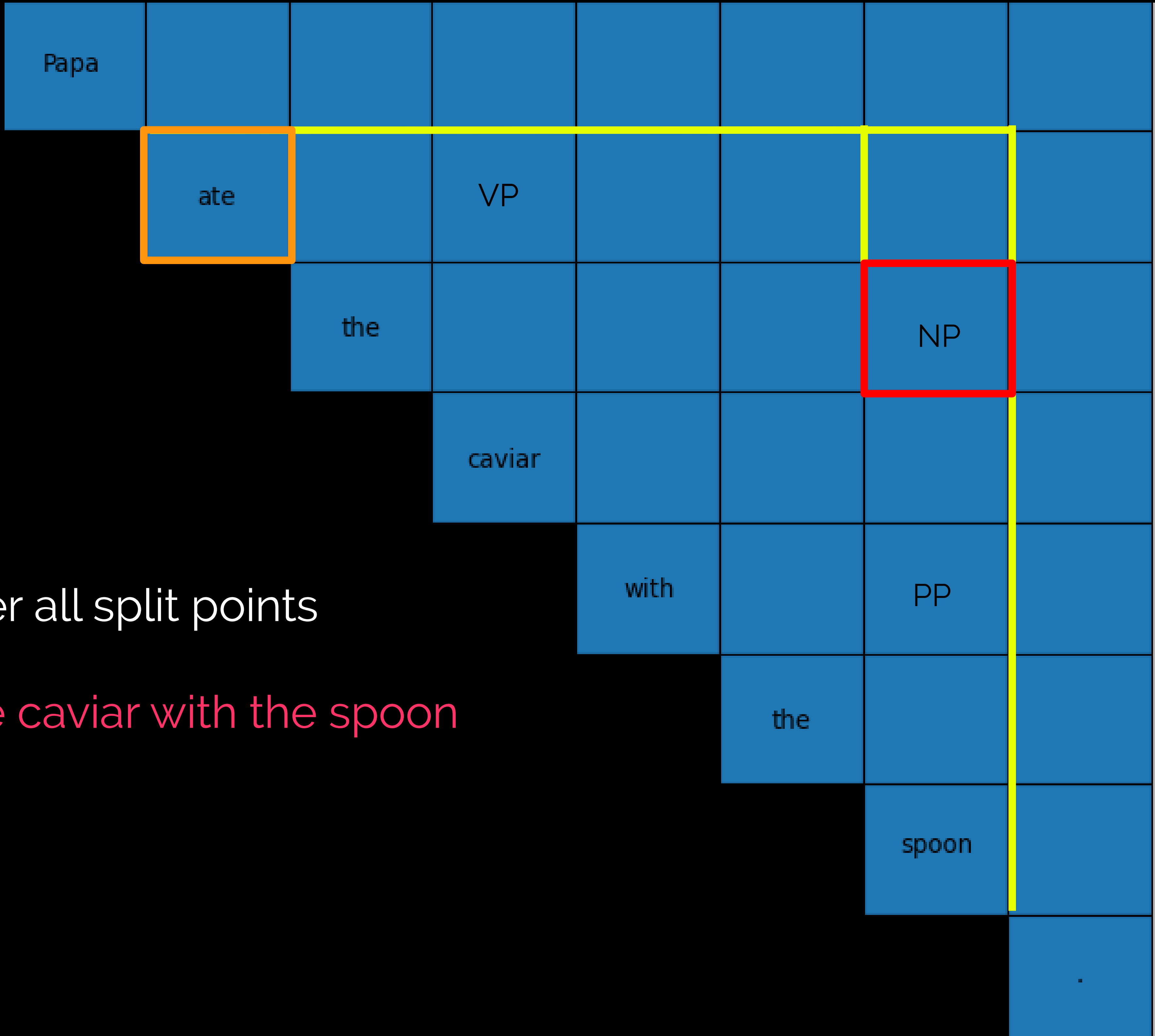
Yellow represents the top of a tree that covers the span "ate the caviar with the spoon"



Just like we saw earlier:

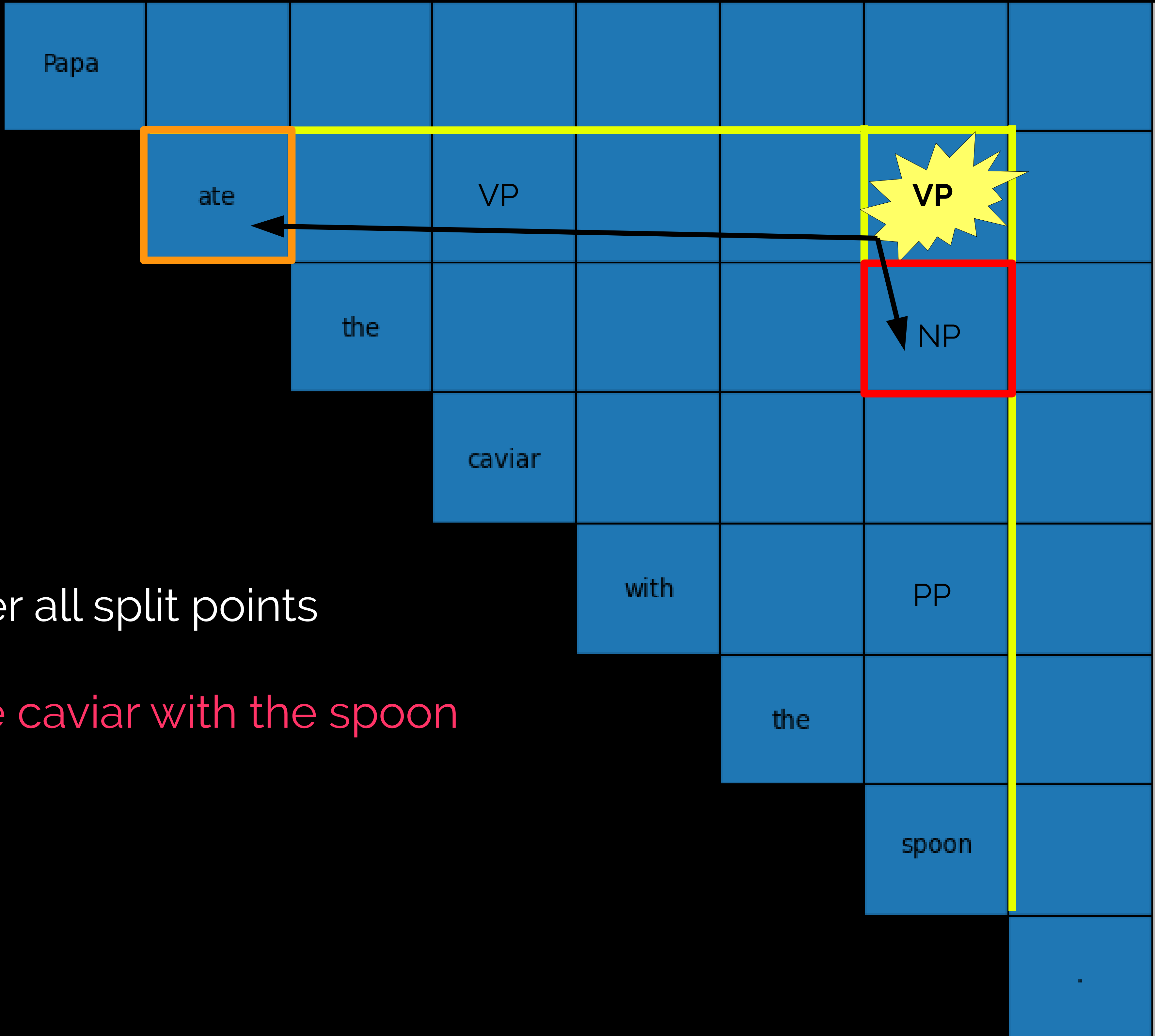
Trees are built up of adjacent subtrees

ate the caviar | with the spoon



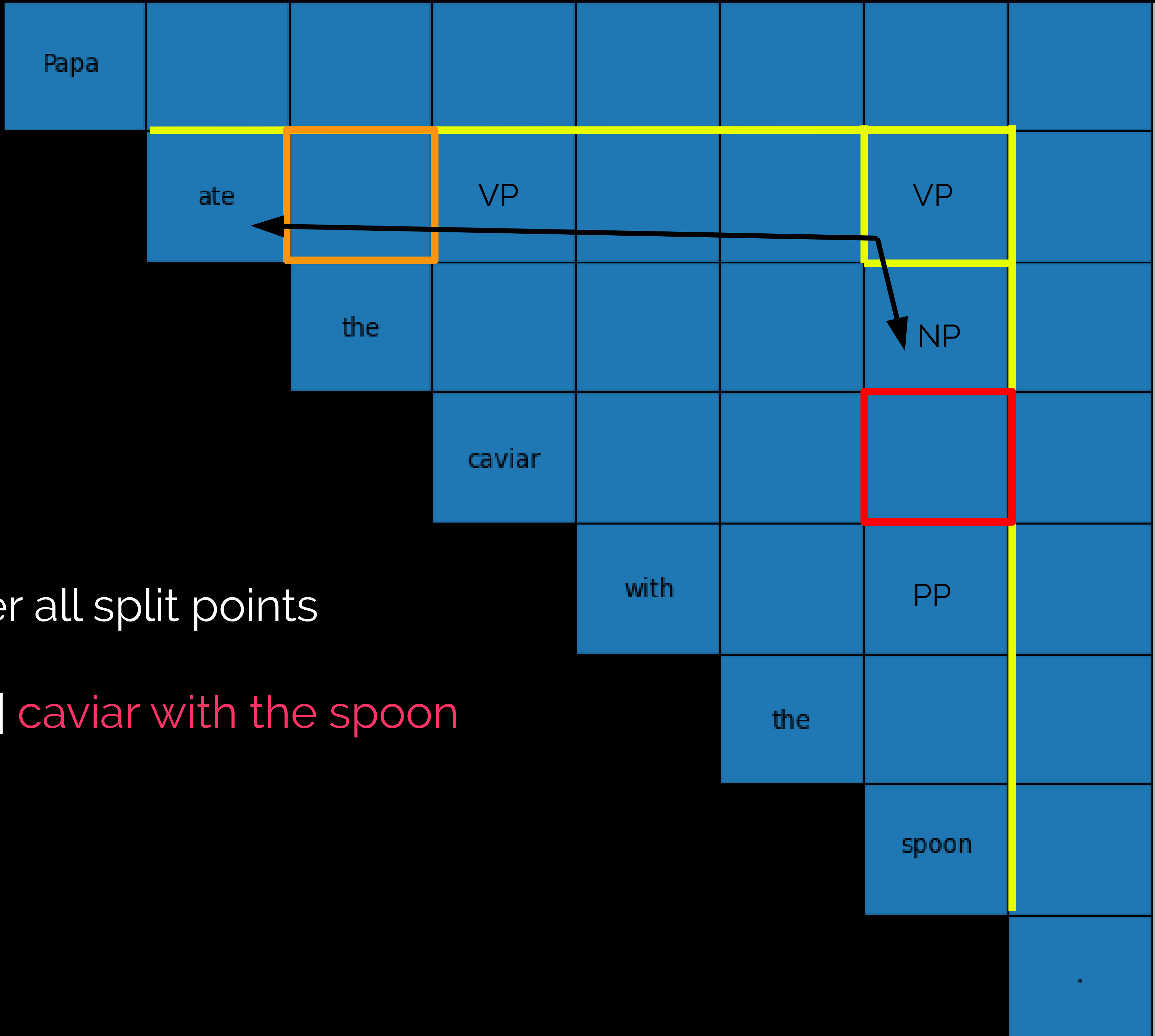
Consider all split points

ate | the caviar with the spoon



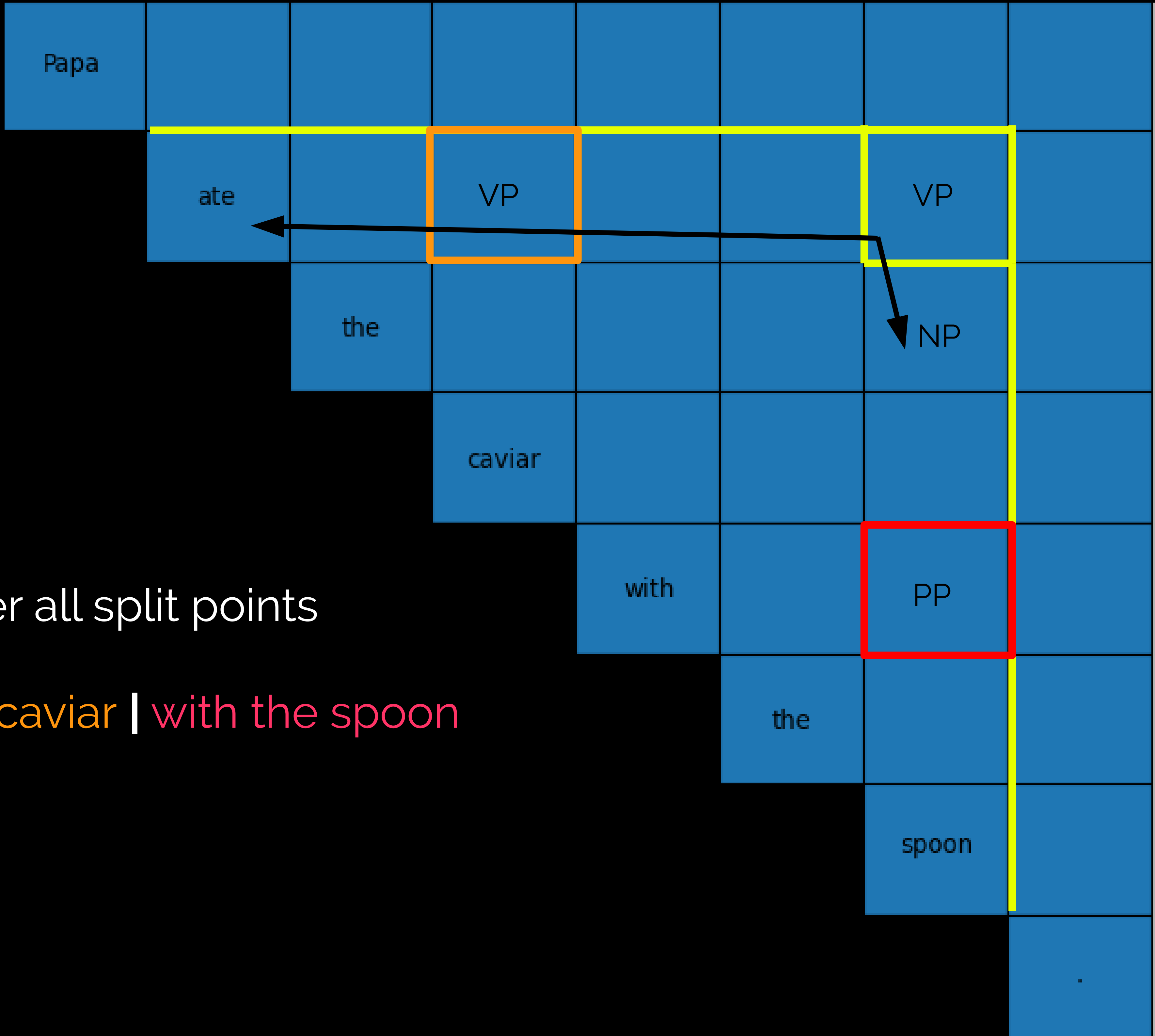
Consider all split points

ate | the caviar with the spoon



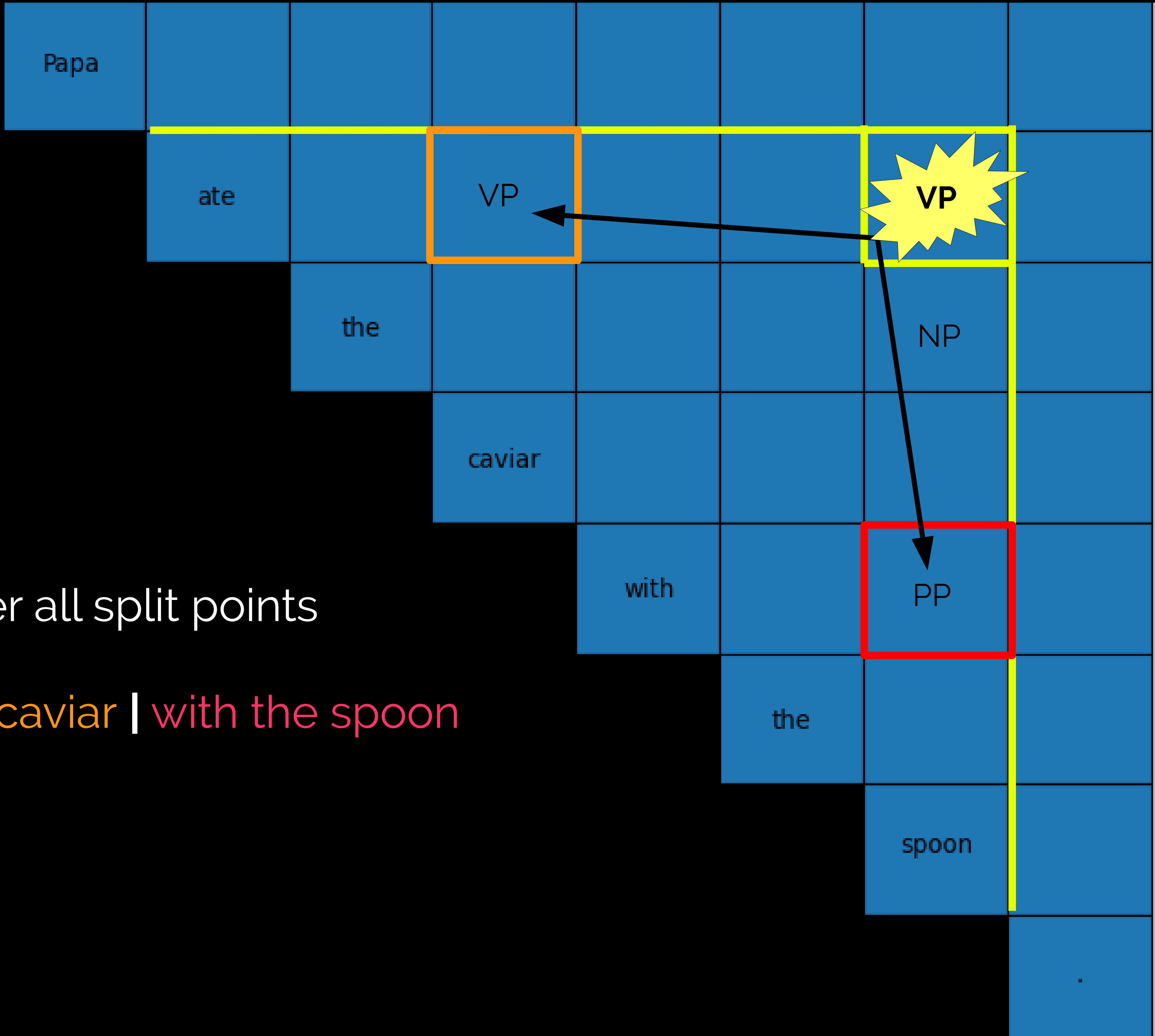
Consider all split points

ate the | caviar with the spoon



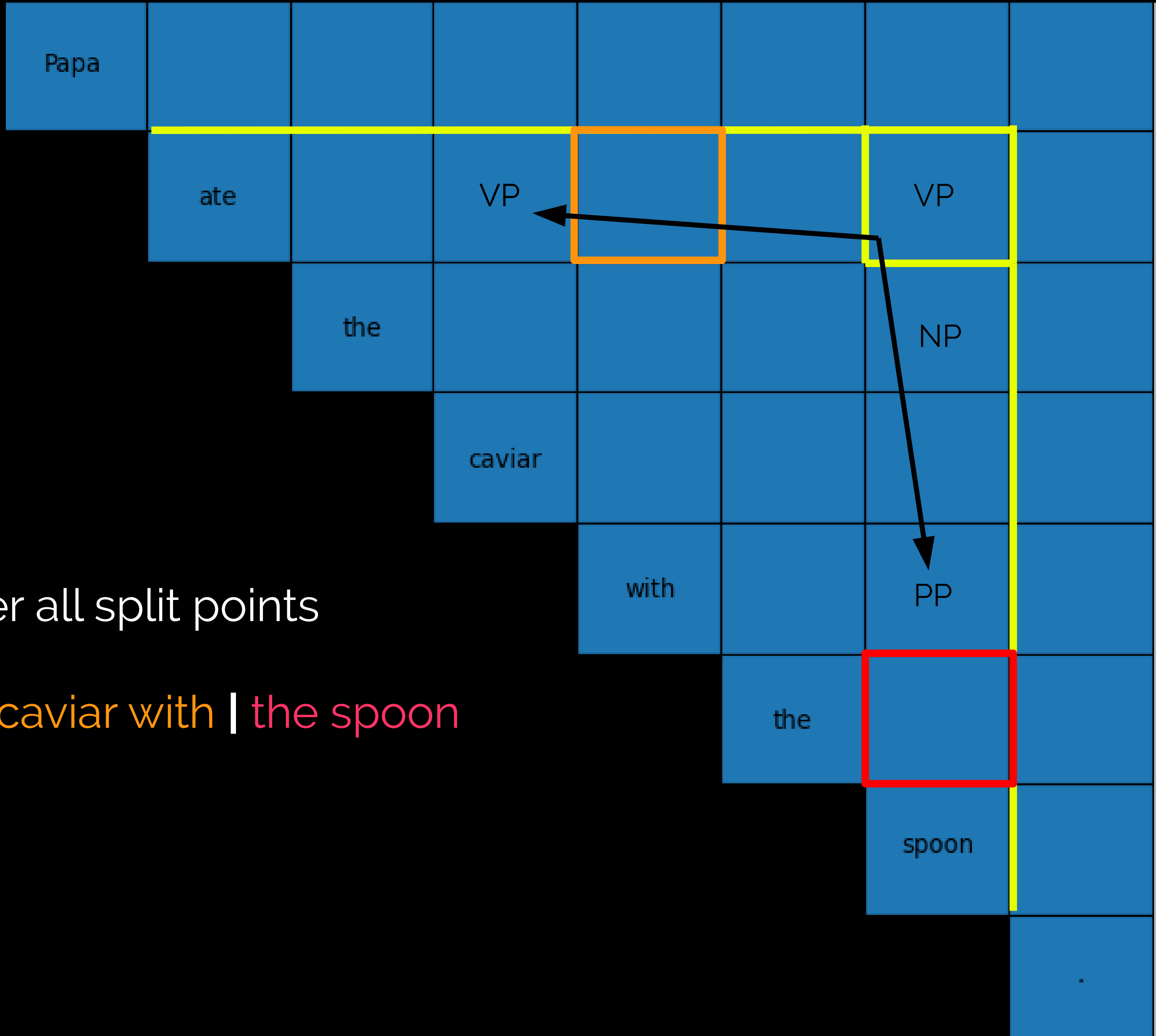
Consider all split points

ate the caviar | with the spoon



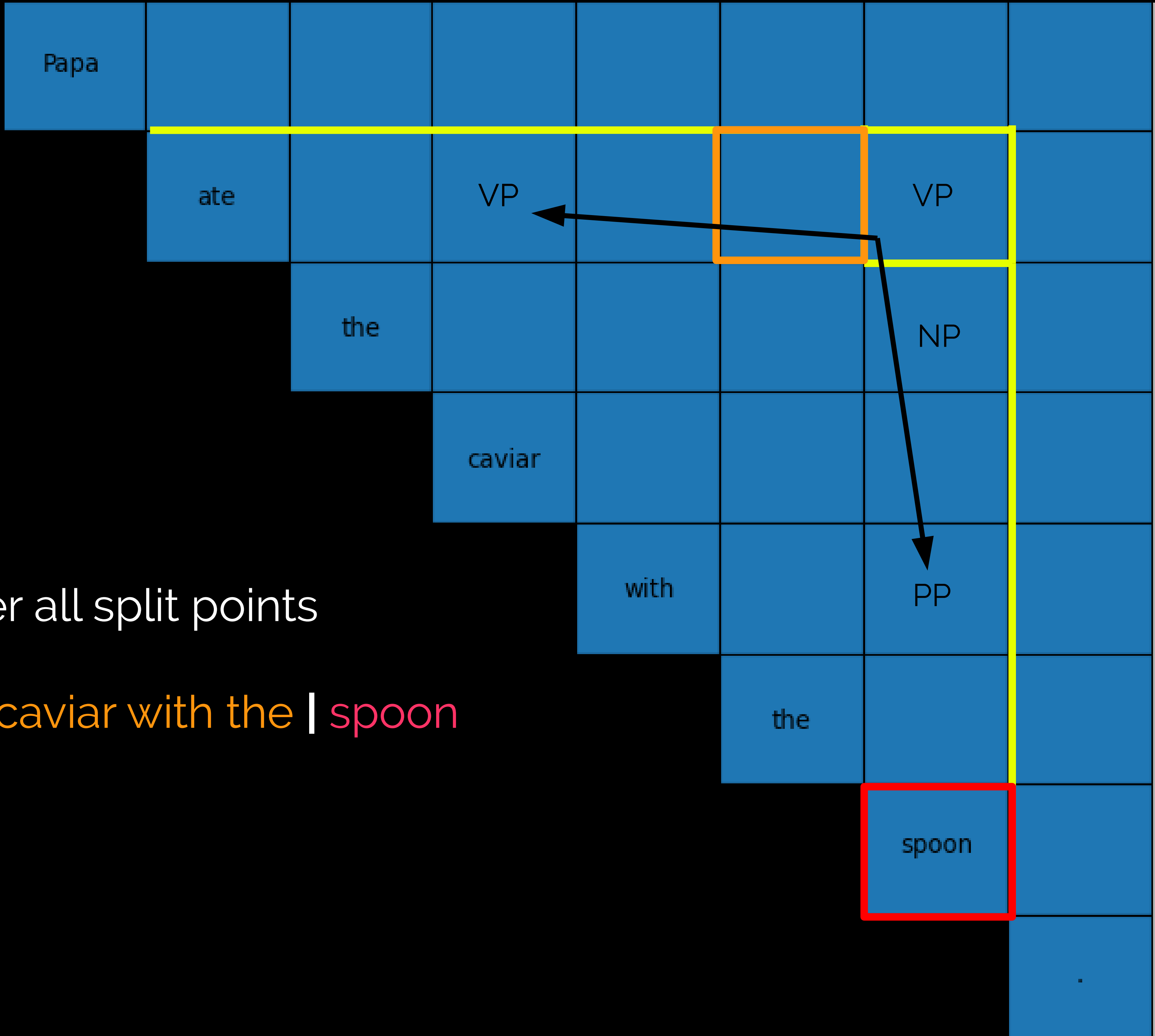
Consider all split points

ate the caviar | with the spoon



Consider all split points

ate the caviar with | the spoon



Consider all split points

ate the caviar with the | spoon

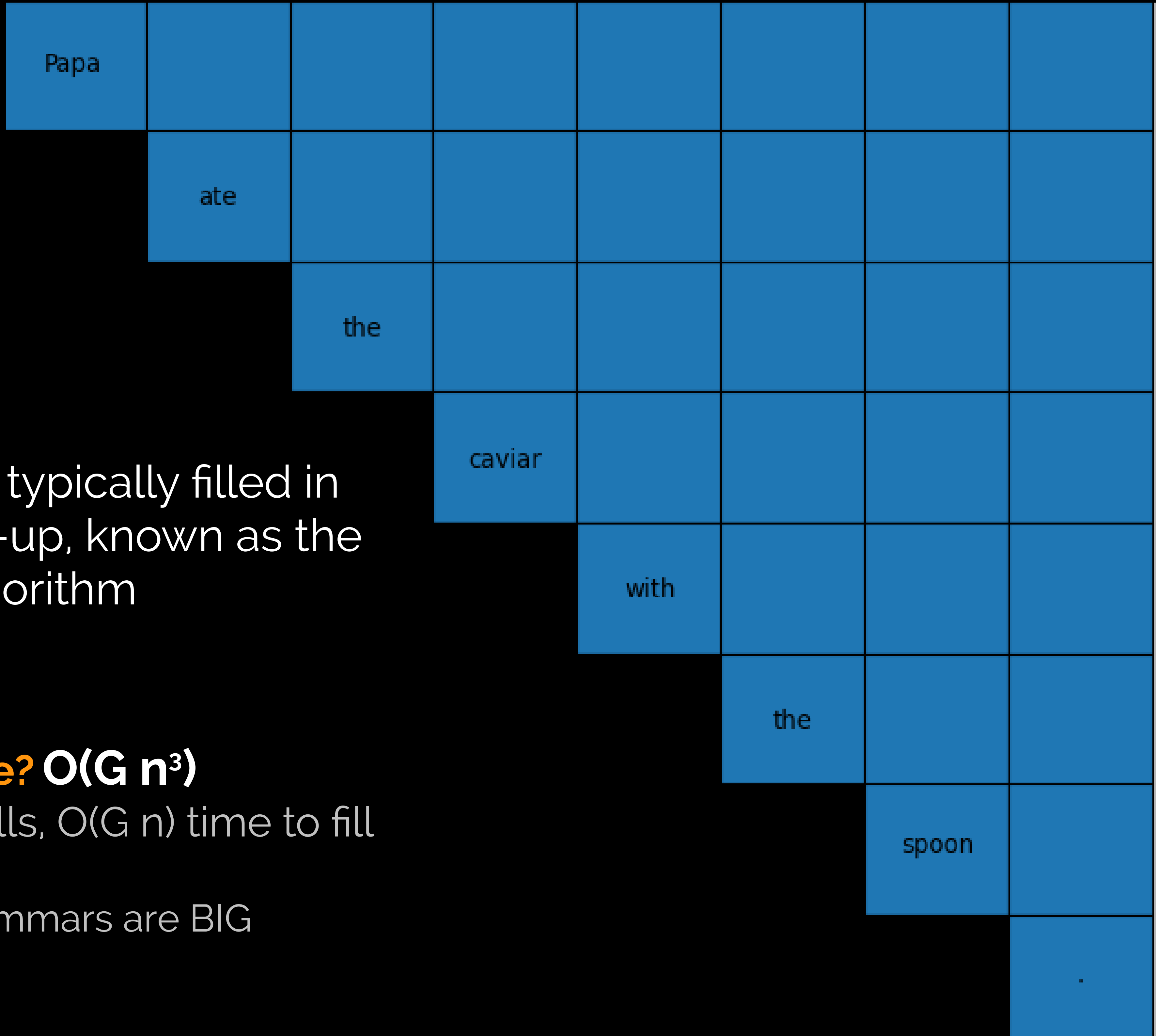


Chart is typically filled in bottom-up, known as the CKY algorithm

Runtime? O(G n^3)

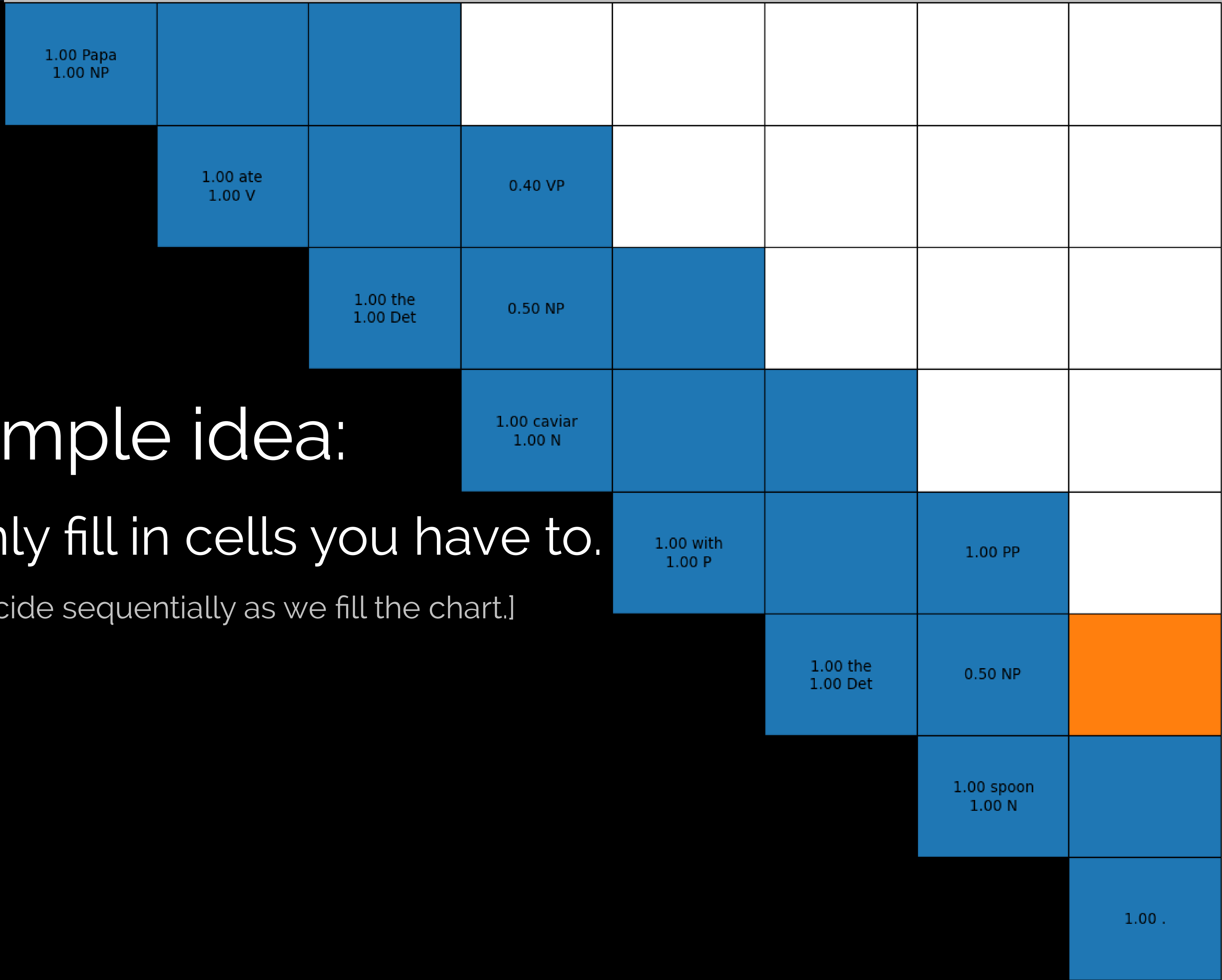
O(n^2) cells, O(G n) time to fill

Real grammars are BIG

Simple idea:

Only fill in cells you have to.

[Decide sequentially as we fill the chart.]



1.00 Papa 1.00 NP							
	1.00 ate 1.00 V		0.40 VP				
		1.00 the 1.00 Det	0.50 NP				
			1.00 caviar 1.00 N				
				1.00 with 1.00 P		1.00 PP	
					1.00 the 1.00 Det	0.50 NP	
						1.00 spoon 1.00 N	
							1.00 .

Simple idea:

Only fill in cells you have to.

Use a classifier

If $\vec{\theta}^\top f(x) < 0$ **then** skip

else fill it in

What features?

Span classification: Bodenstab (2012)

Simple idea:

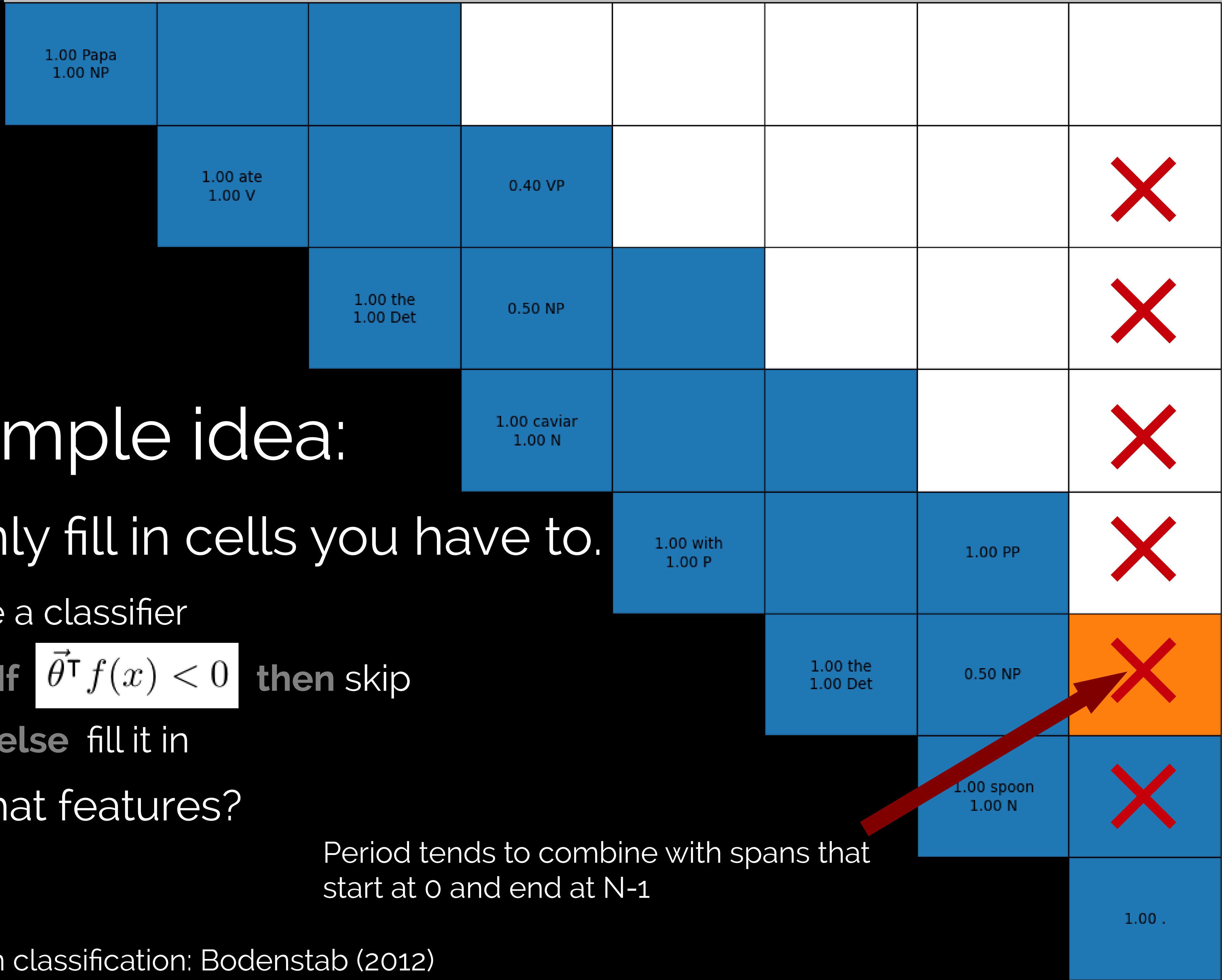
Only fill in cells you have to.

Use a classifier

If $\vec{\theta}^{\top} f(x) < 0$ **then** skip

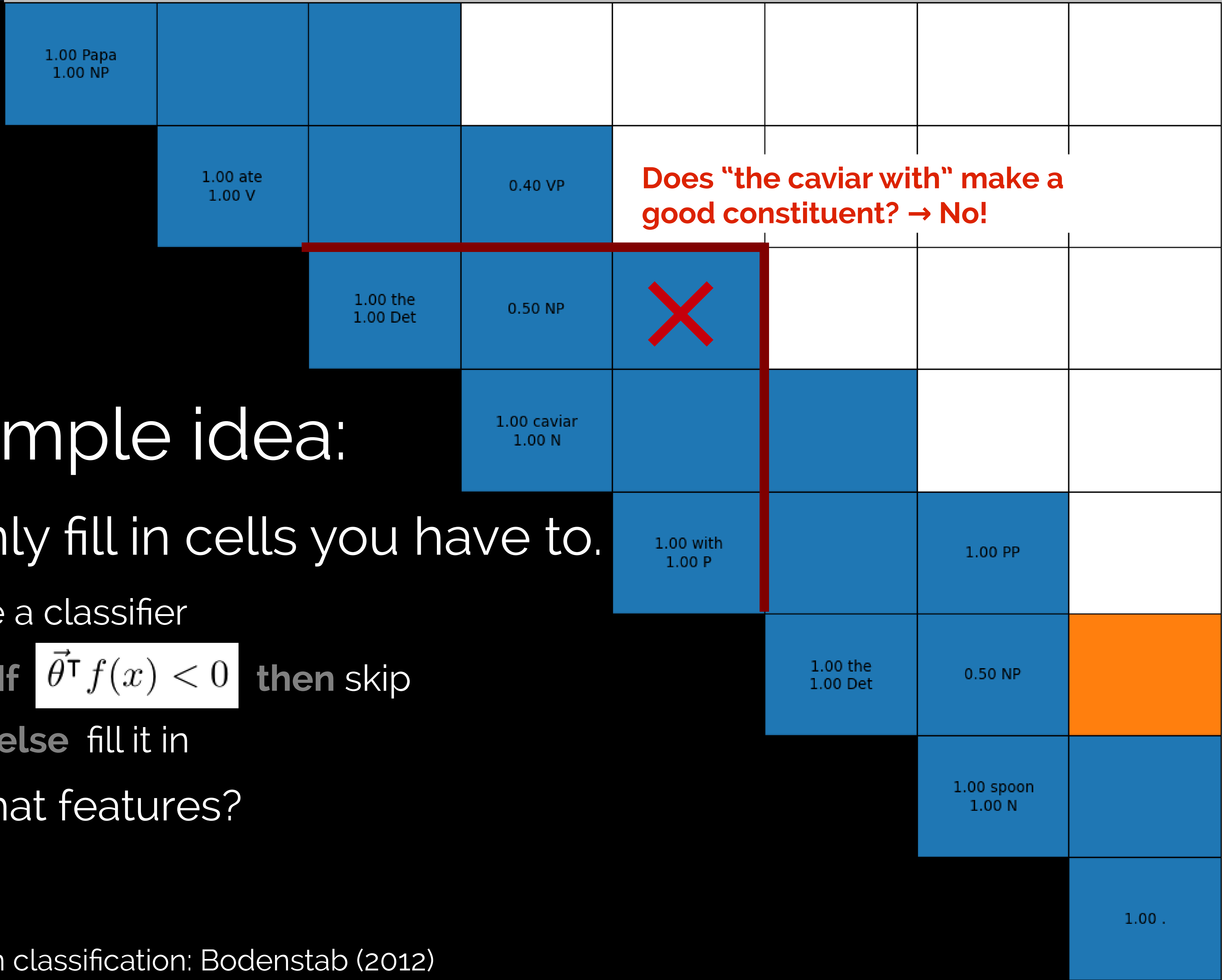
else fill it in

What features?



Period tends to combine with spans that start at 0 and end at N-1

Span classification: Bodenstab (2012)



Simple idea:

Only fill in cells you have to.

Use a classifier

If $\vec{\theta}^{\top} f(x) < 0$ **then** skip

else fill it in

What features?

Span classification: Bodenstab (2012)

How to train?

1.00 Papa 1.00 NP			1.00 S			1.00 S 1.00 S	1.00 S 1.00 S
	1.00 ate 1.00 V		0.40 VP			0.40 VP 0.10 VP	
		1.00 the 1.00 Det	0.50 NP			0.12 NP	
			1.00 caviar 1.00 N				
				1.00 with 1.00 P		1.00 PP	
					1.00 the 1.00 Det	0.50 NP	
						1.00 spoon 1.00 N	
							1.00 .

How to train?

Don't prune the **gold** parse

Everything else → negative example

(Don't prune width=1 or n, always want those.)

1.00 Papa 1.00 NP			1.00 S			1.00 S 1.00 S	1.00 S 1.00 S
	1.00 ate 1.00 V		0.40 VP			0.40 VP 0.10 VP	
		1.00 the 1.00 Det	0.50 NP			0.12 NP	
			1.00 caviar 1.00 N				
				1.00 with 1.00 P		1.00 PP	
					1.00 the 1.00 Det	0.50 NP	
						1.00 spoon 1.00 N	
							1.00 .

How to train?

Don't prune the **gold** parse

Everything else → negative example

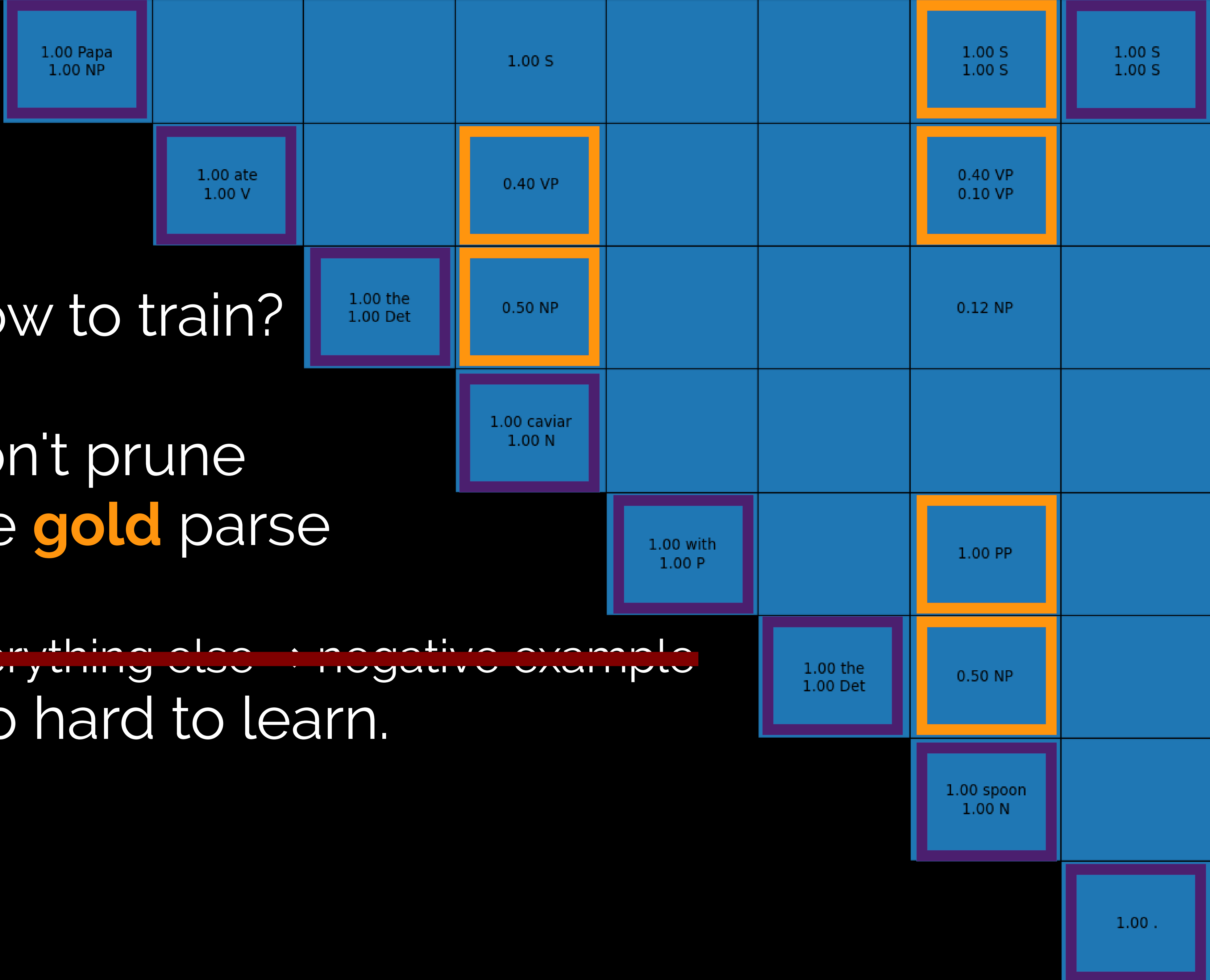
1.00 Papa 1.00 NP			1.00 S			1.00 S 1.00 S	1.00 S 1.00 S
	1.00 ate 1.00 V		0.40 VP			0.40 VP 0.10 VP	
		1.00 the 1.00 Det	0.50 NP			0.12 NP	
			1.00 caviar 1.00 N				
				1.00 with 1.00 P		1.00 PP	
					1.00 the 1.00 Det	0.50 NP	
						1.00 spoon 1.00 N	
							1.00 .

How to train?

Don't prune the **gold** parse

~~Everything else → negative example~~

too hard to learn.

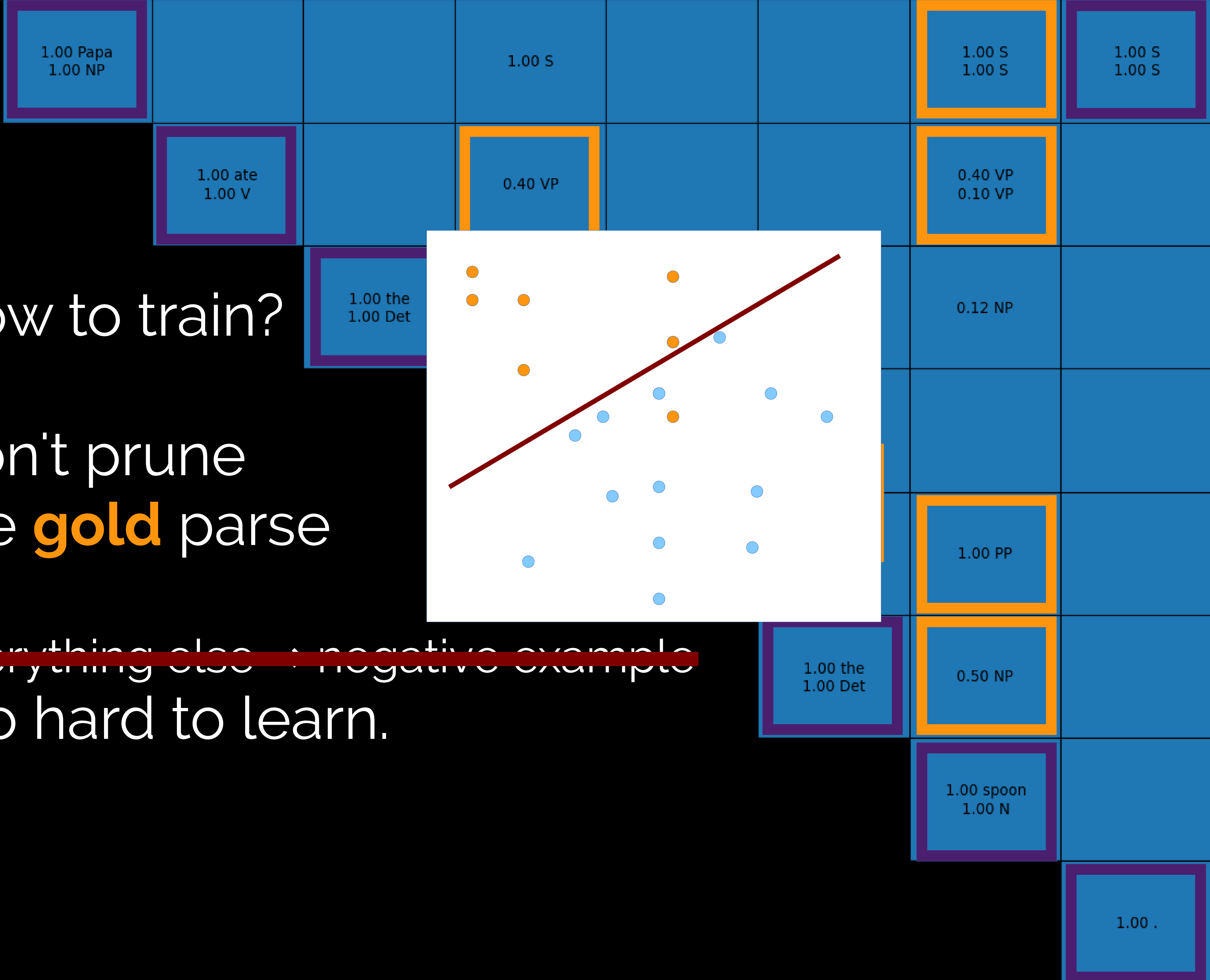


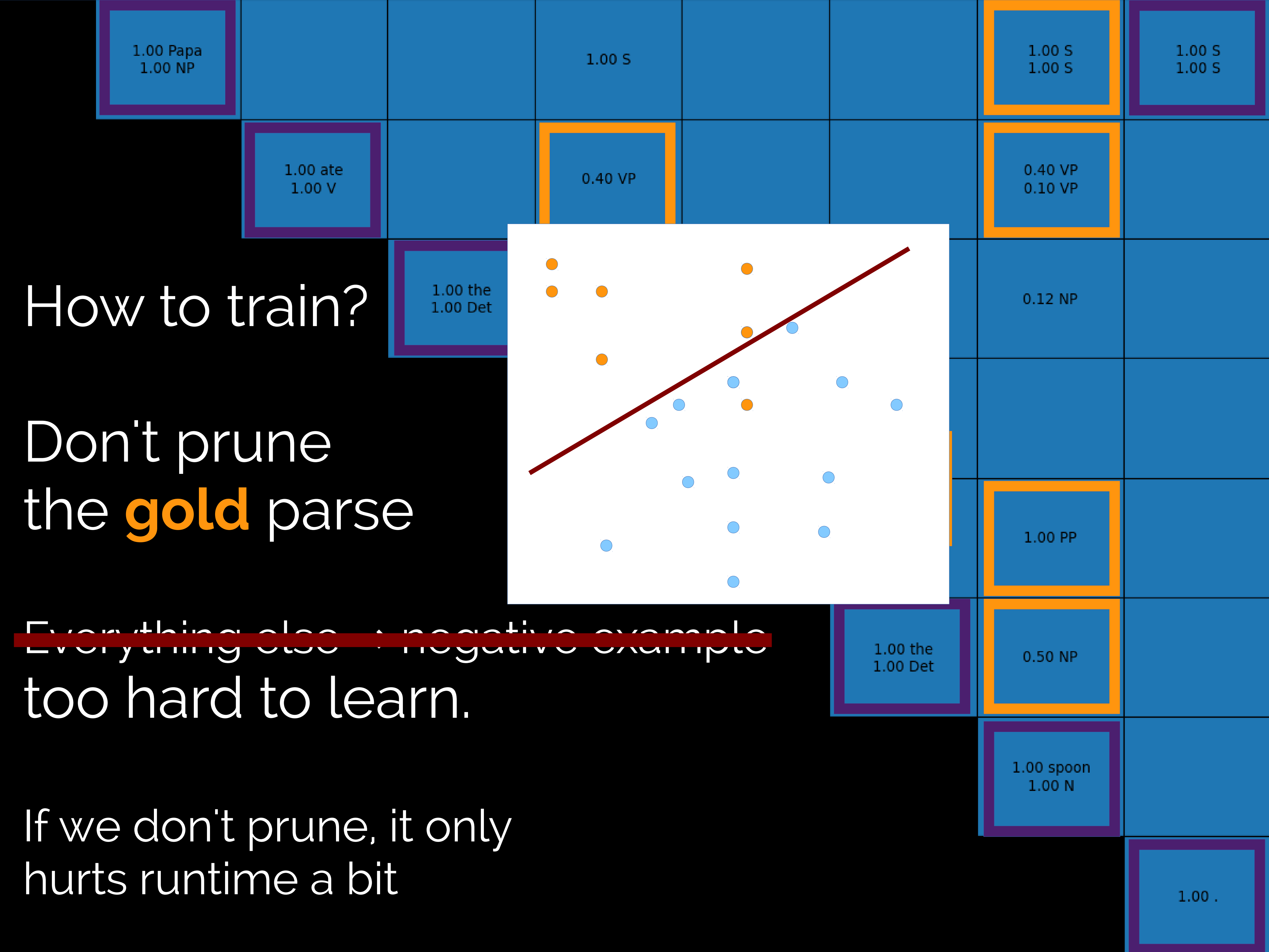
How to train?

Don't prune the **gold** parse

~~Everything else → negative example~~

too hard to learn.

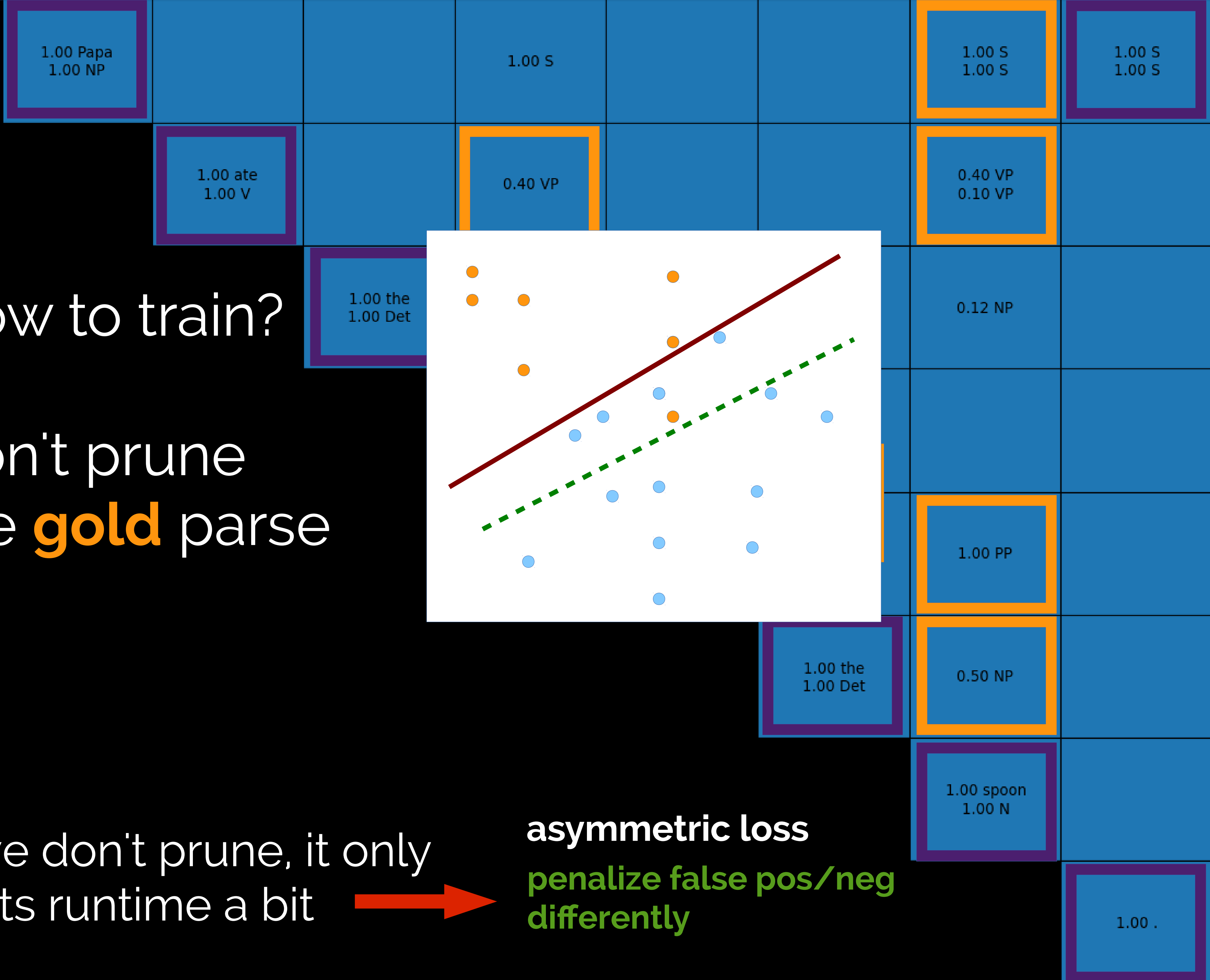




How to train?
Don't prune
the **gold** parse
~~Everything else → negative example~~
too hard to learn.
If we don't prune, it only
hurts runtime a bit
1.00 Papa
1.00 NP
1.00 S
1.00 S
1.00 S
1.00 S
1.00 S
1.00 ate
1.00 V
0.40 VP
0.40 VP
0.10 VP
1.00 the
1.00 Det
0.12 NP
1.00 PP
1.00 the
1.00 Det
0.50 NP
1.00 spoon
1.00 N
1.00 .

How to train?

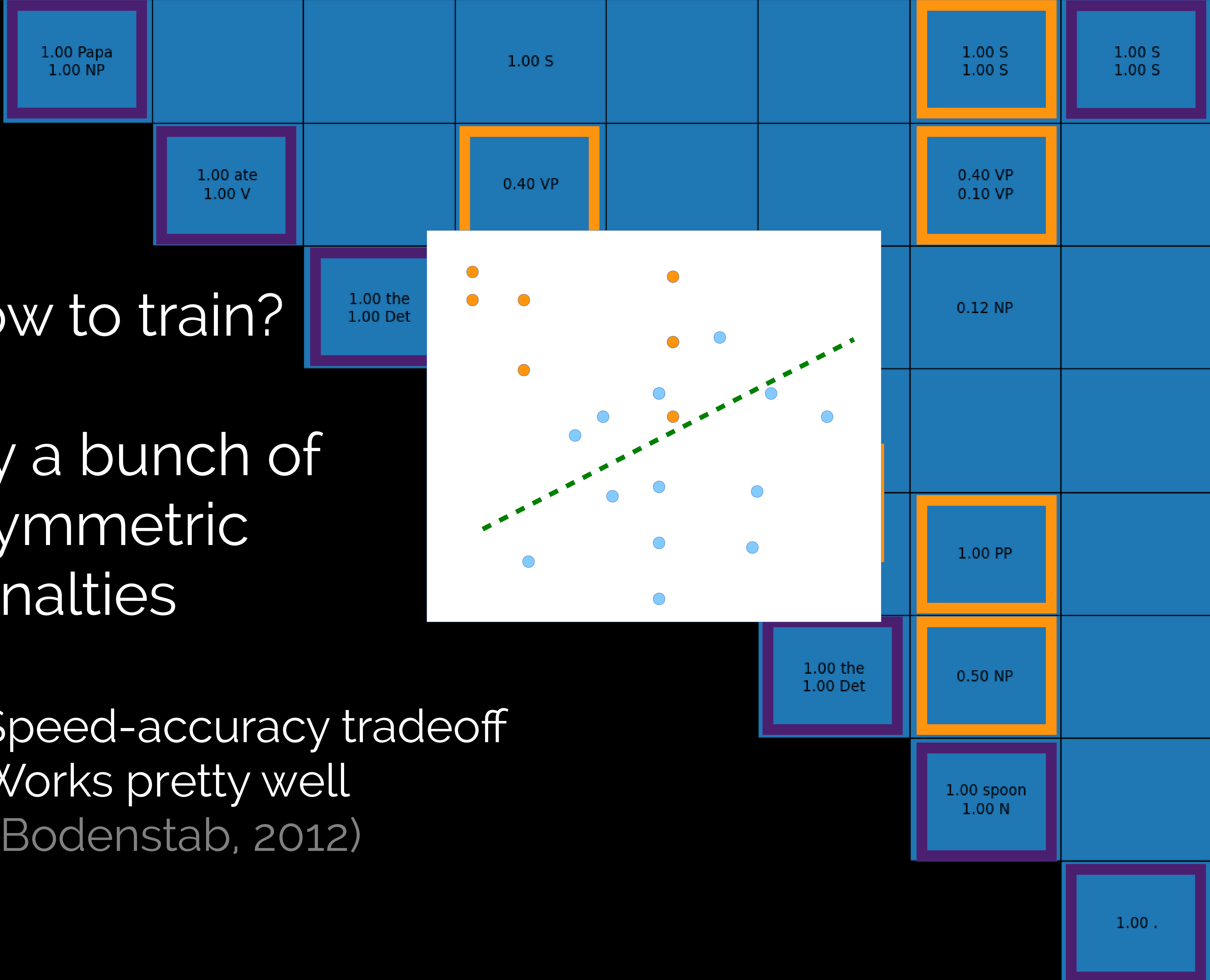
Don't prune the gold parse



If we don't prune, it only hurts runtime a bit

asymmetric loss

penalize false pos/neg differently



How to train?

Try a bunch of asymmetric penalties

→ Speed-accuracy tradeoff

→ Works pretty well
(Bodenstab, 2012)

What's missing?

What's missing?

- Unmodeled interactions

  Fails to capture end-to-end performance.

What's missing?

- Unmodeled interactions

  Fails to capture end-to-end performance.

- Requires labeling
  - Doesn't consider other "good" parses
  - What's the best labeling to use?

What's missing?

- Unmodeled interactions

  Fails to capture end-to-end performance.

- Requires labeling
  - Doesn't consider other "good" parses
  - What's the best labeling to use?

- Limits expressiveness of features

  Doesn't support dynamic features, e.g., looking the parse chart.

Our Approach

Global objective

reward = accuracy - λ runtime

Global objective

$$\text{reward} = \text{accuracy} - \lambda \ \text{runtime}$$

Accuracy and runtime
of the **end-to-end** system

Global objective

$$\text{reward} = \text{accuracy} - \lambda \ \text{runtime}$$

Accuracy and runtime
of the **end-to-end** system

Preferably
deterministic
functions
(not seconds)

Global objective

Tradeoff parameter
(exchange rate)

$$\text{reward} = \text{accuracy} - \lambda \ \text{runtime}$$

Preferably
deterministic
functions
(not seconds)

Accuracy and runtime
of the **end-to-end** system

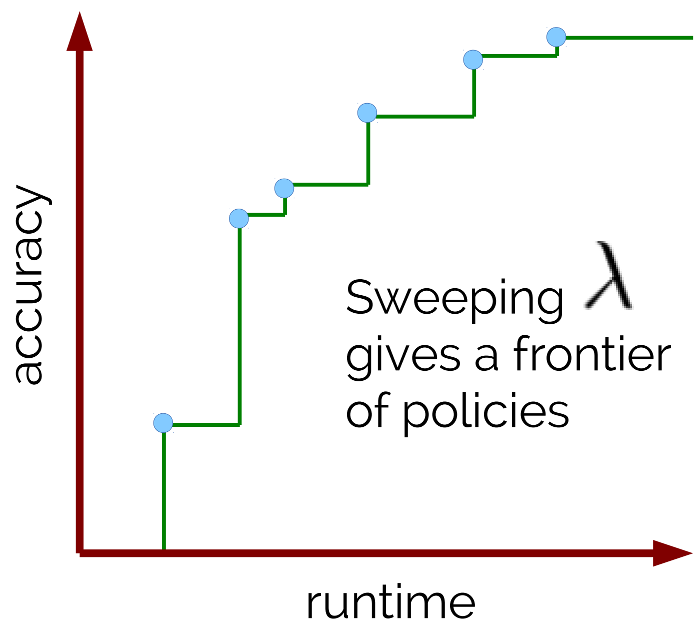
Global objective

Tradeoff parameter
(exchange rate)

$$\text{reward} = \text{accuracy} - \lambda\ \text{runtime}$$

Preferably
deterministic
functions
(not seconds)

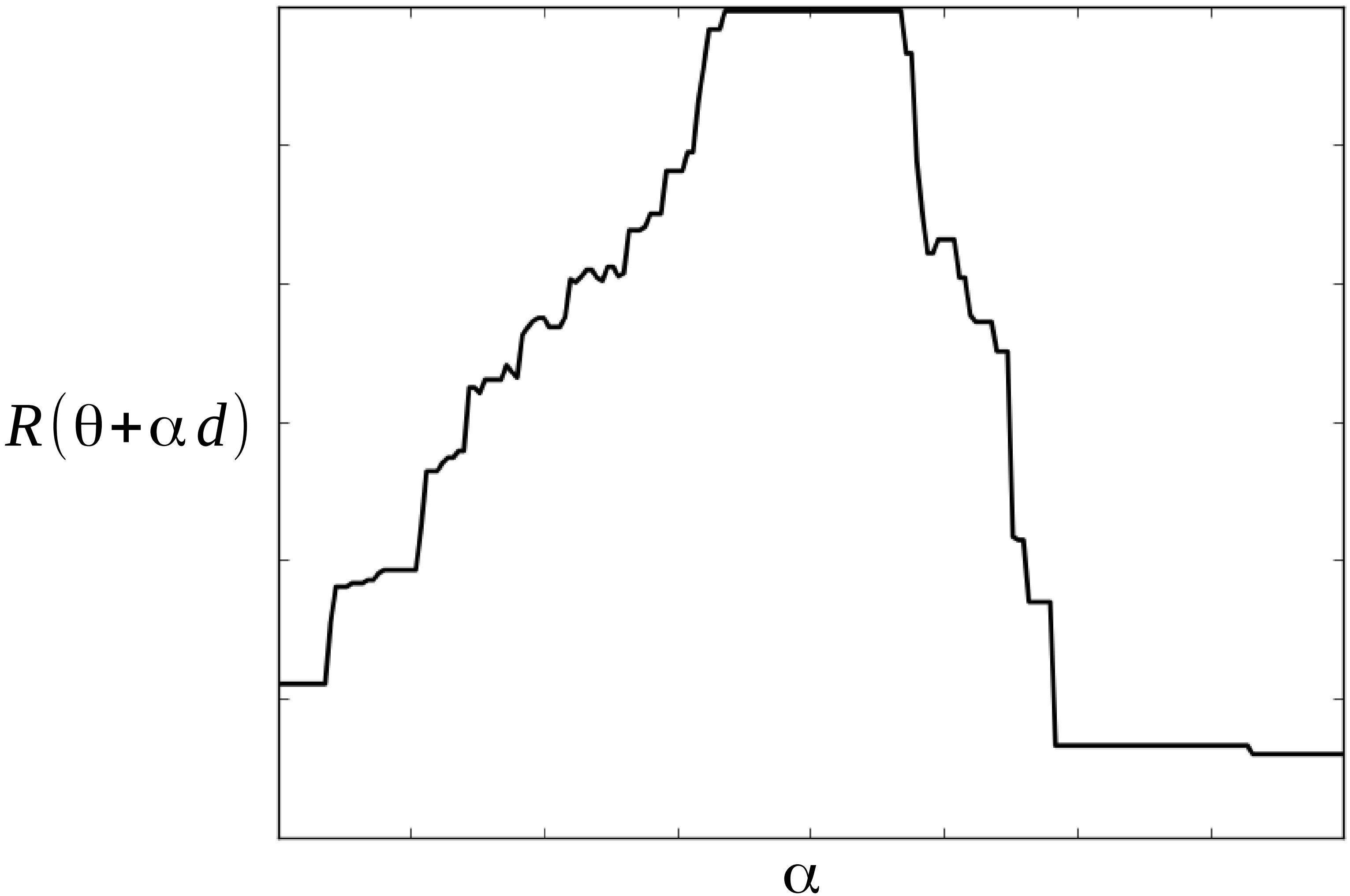
Accuracy and runtime
of the **end-to-end** system



How to train?

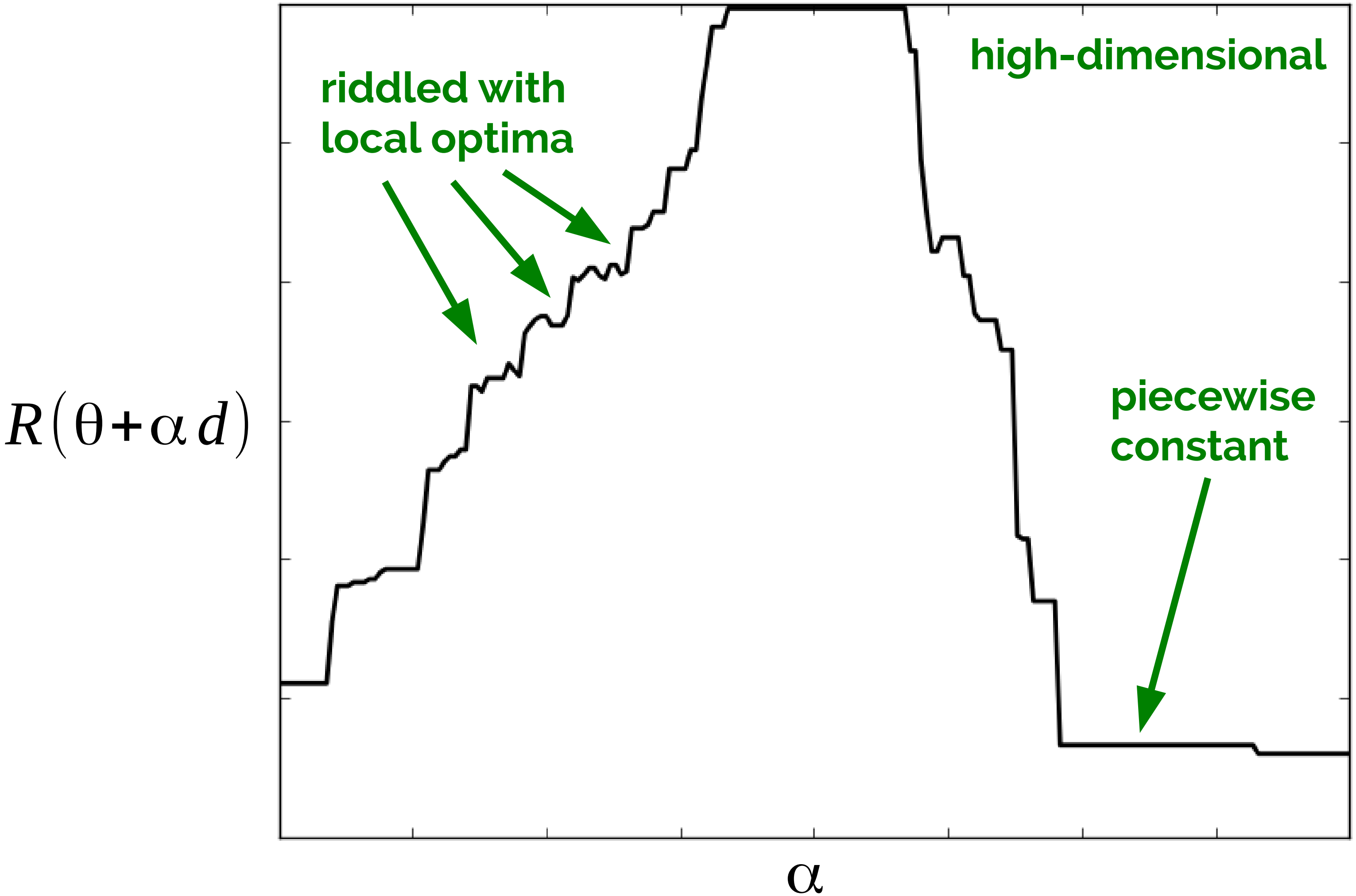
Hard to optimize

Cross-section of reward along a random direction, d



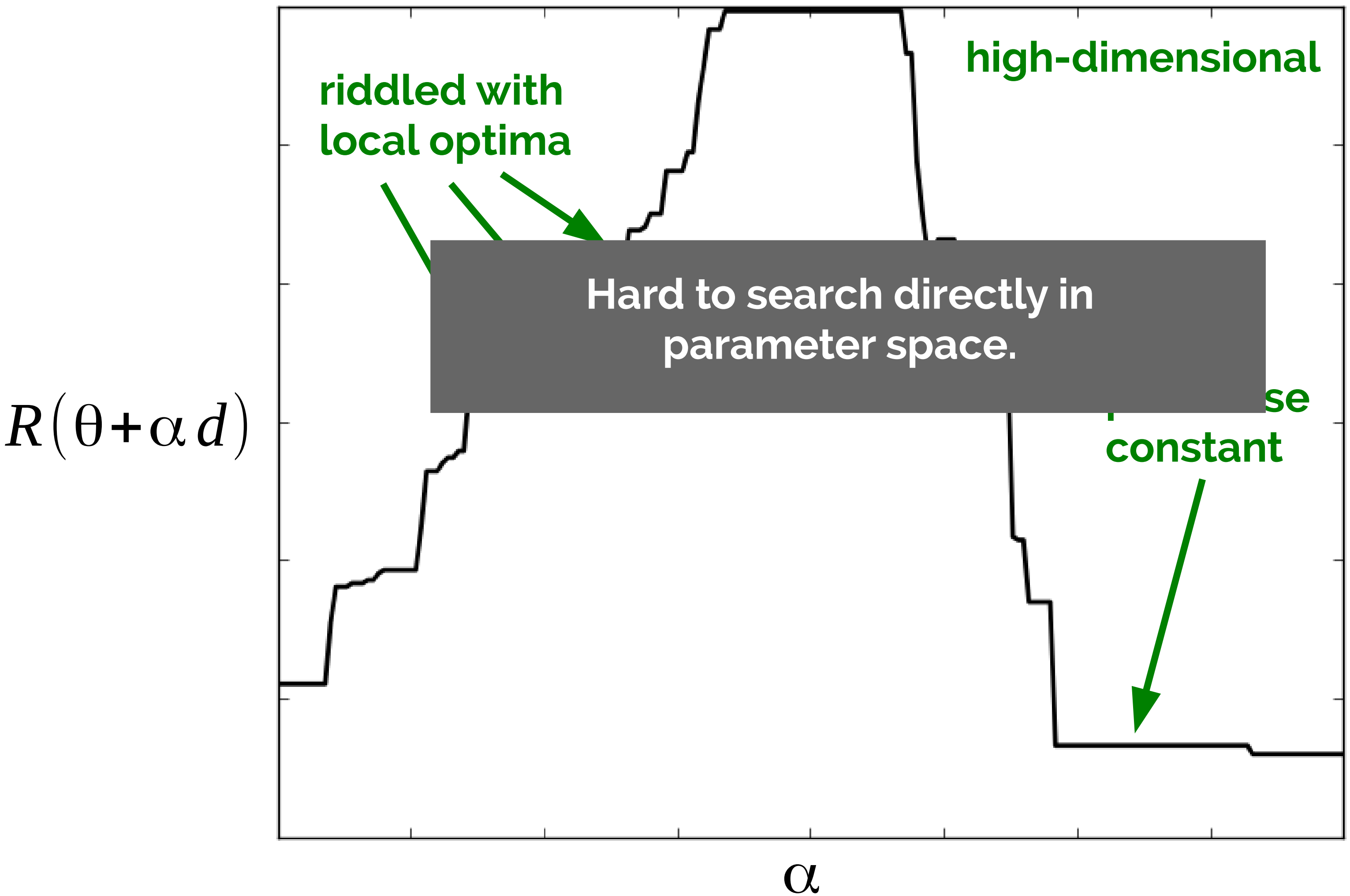
Hard to optimize

Cross-section of reward along a random direction, d



Hard to optimize

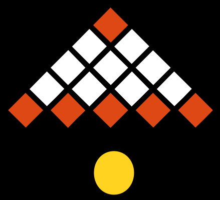
Cross-section of reward along a random direction, d



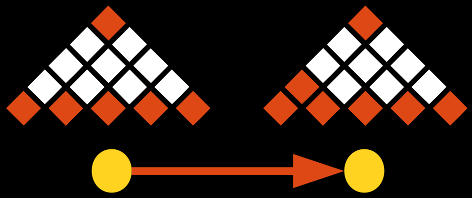
Hard to search directly in parameter space.

Approximate policy iteration

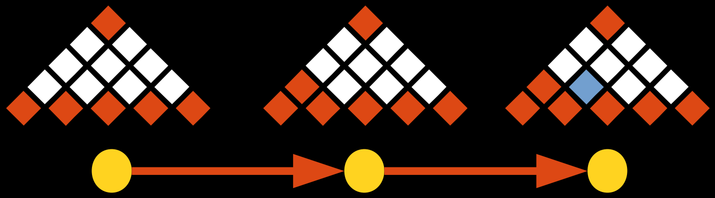
Approximate policy iteration



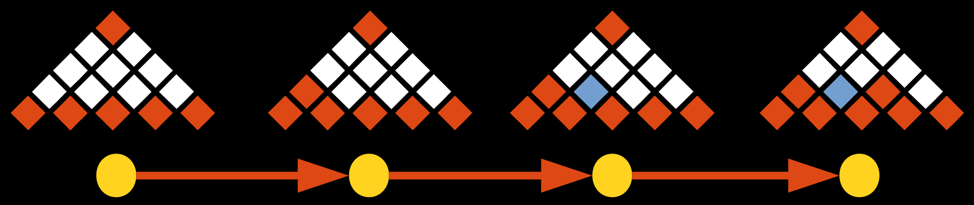
Approximate policy iteration



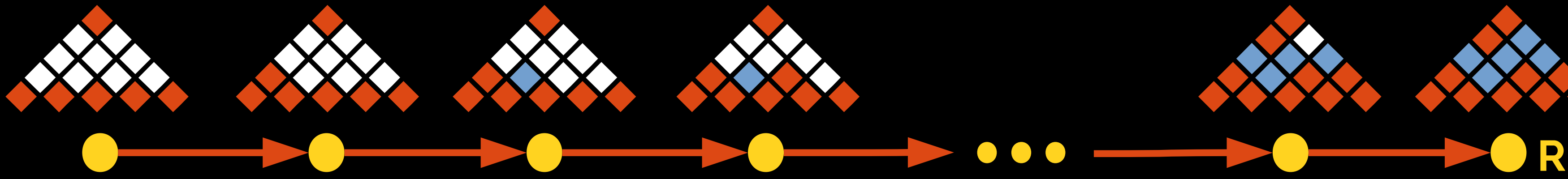
Approximate policy iteration



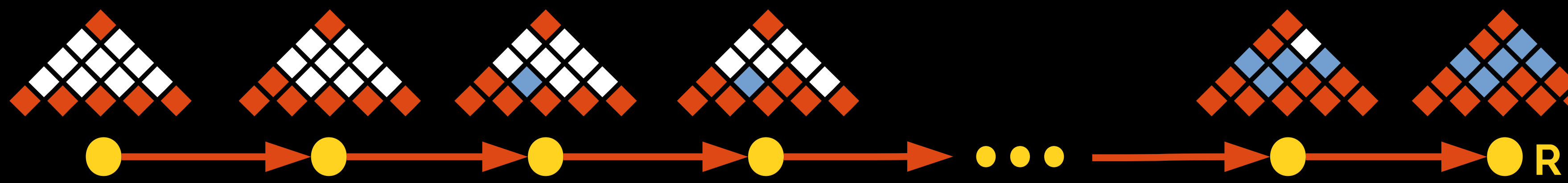
Approximate policy iteration



Approximate policy iteration

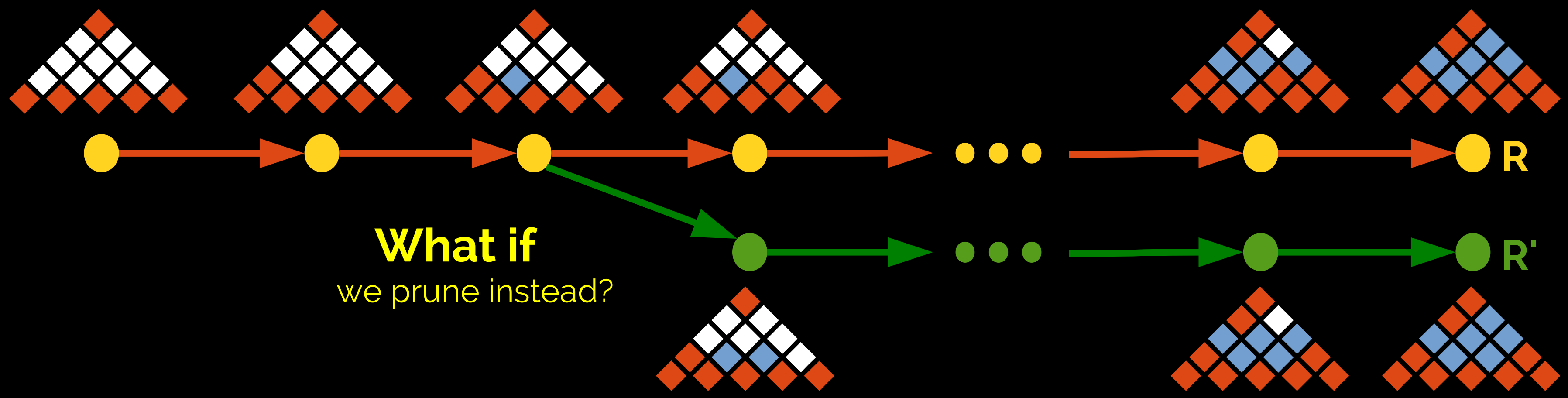


Approximate policy iteration

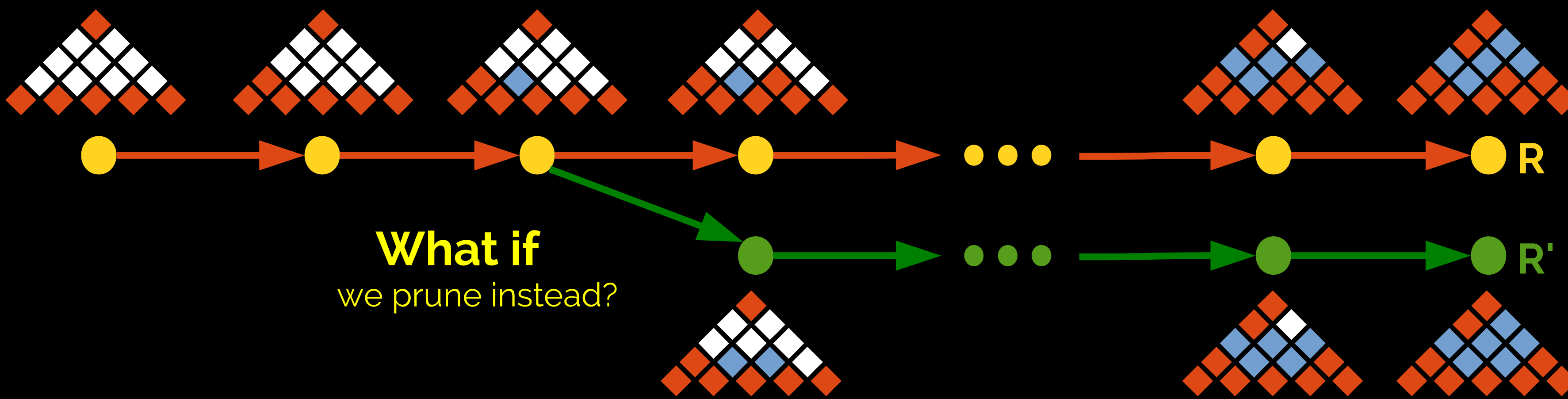


What if
we prune instead?

Approximate policy iteration

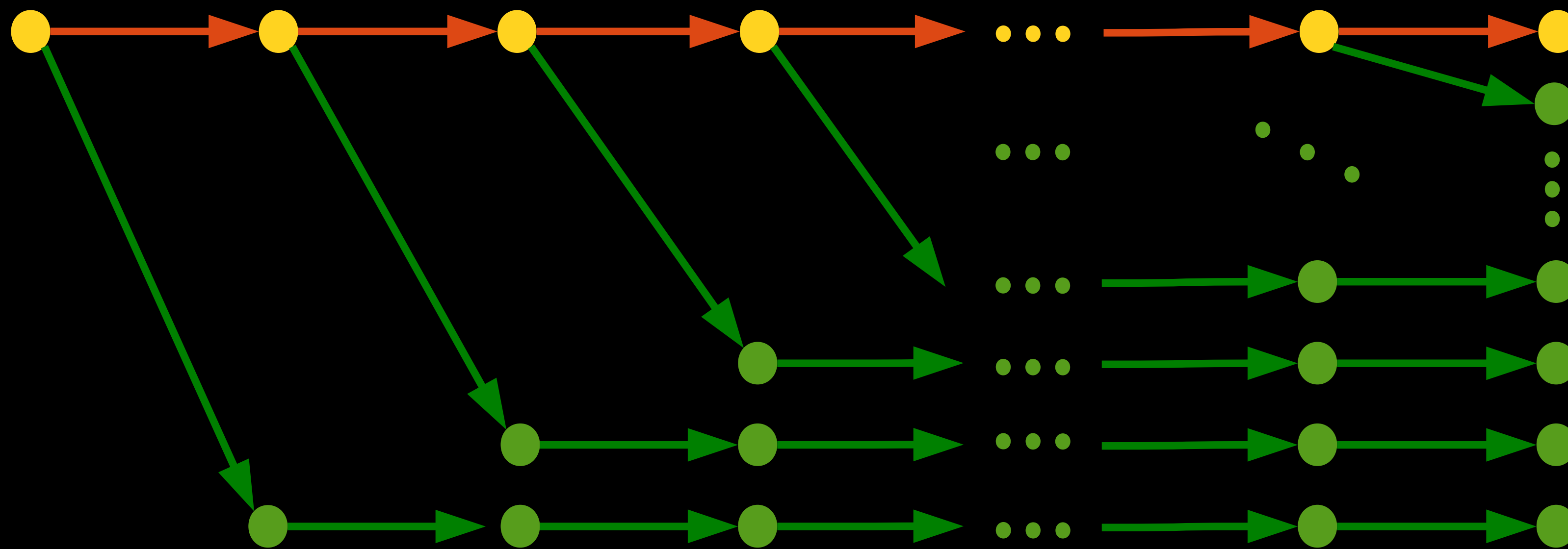


Approximate policy iteration



Classification example:
label: better action **R>R'**
weight: **|R – R'|**

Approximate policy iteration

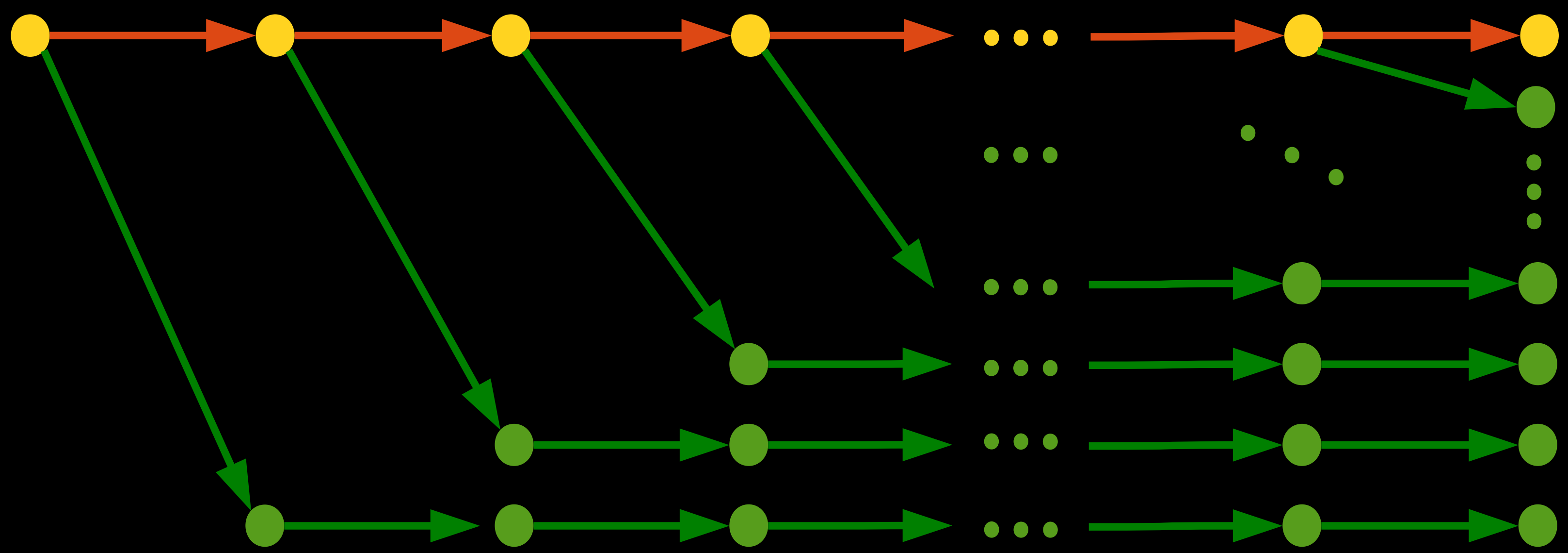


Classification example:
label: better action **R>R'**
weight: **|R – R'|**

Try alternative actions at each position

Update policy

Approximate policy iteration



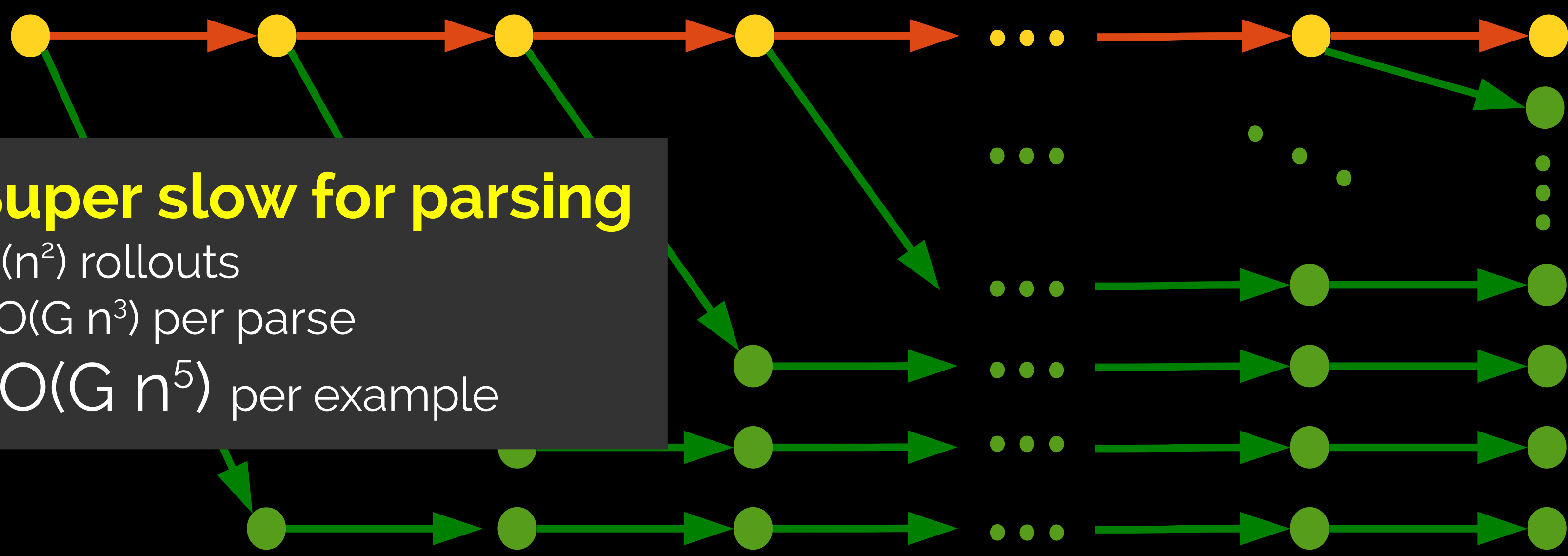
Classification example:
label: better action **R>R'**
weight: **|R – R'|**

Try alternative actions at each position

Update policy

SEARN (Daumé+,2009)
LOLS (Chang+,2015)

Approximate policy iteration



Super slow for parsing

$O(n^2)$ rollouts

* $O(G\ n^3)$ per parse

= $O(G\ n^5)$ per example

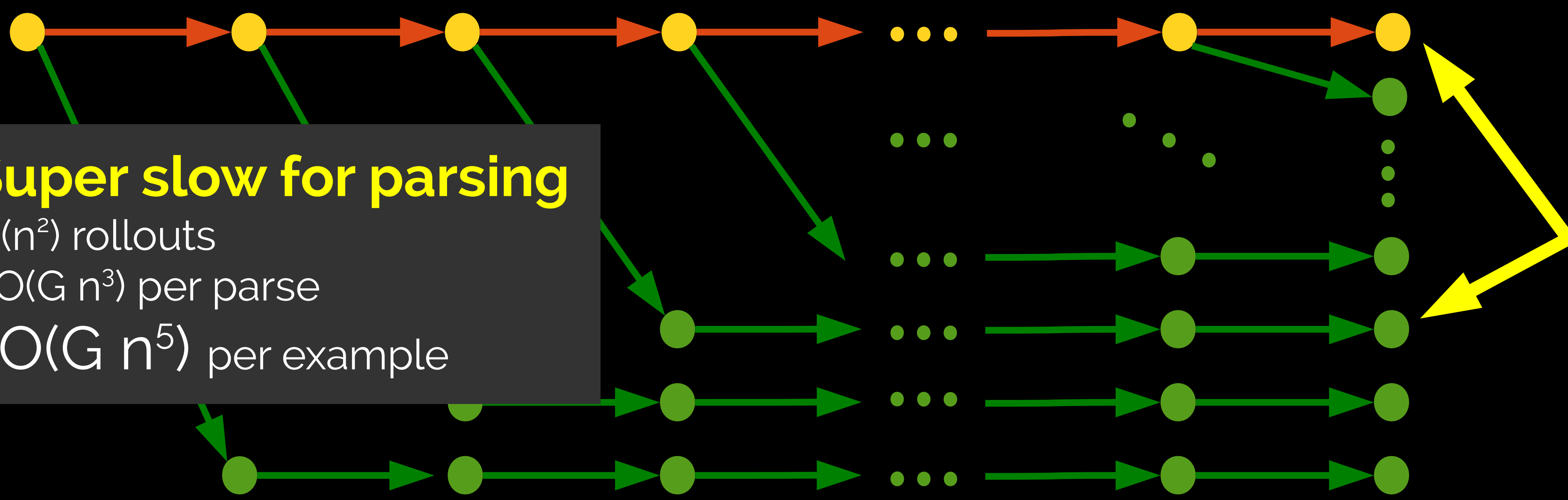
Classification example:
label: better action **R>R'**
weight: **|R – R'|**

Try alternative actions at each position

Update policy

SEARN (Daumé+,2009)
LOLS (Chang+,2015)

Approximate policy iteration



Super slow for parsing

$O(n^2)$ rollouts

* $O(G\,n^3)$ per parse

= $O(G\,n^5)$ per example

Try alternative actions at each position

Charts are so similar!
Can we reuse work?

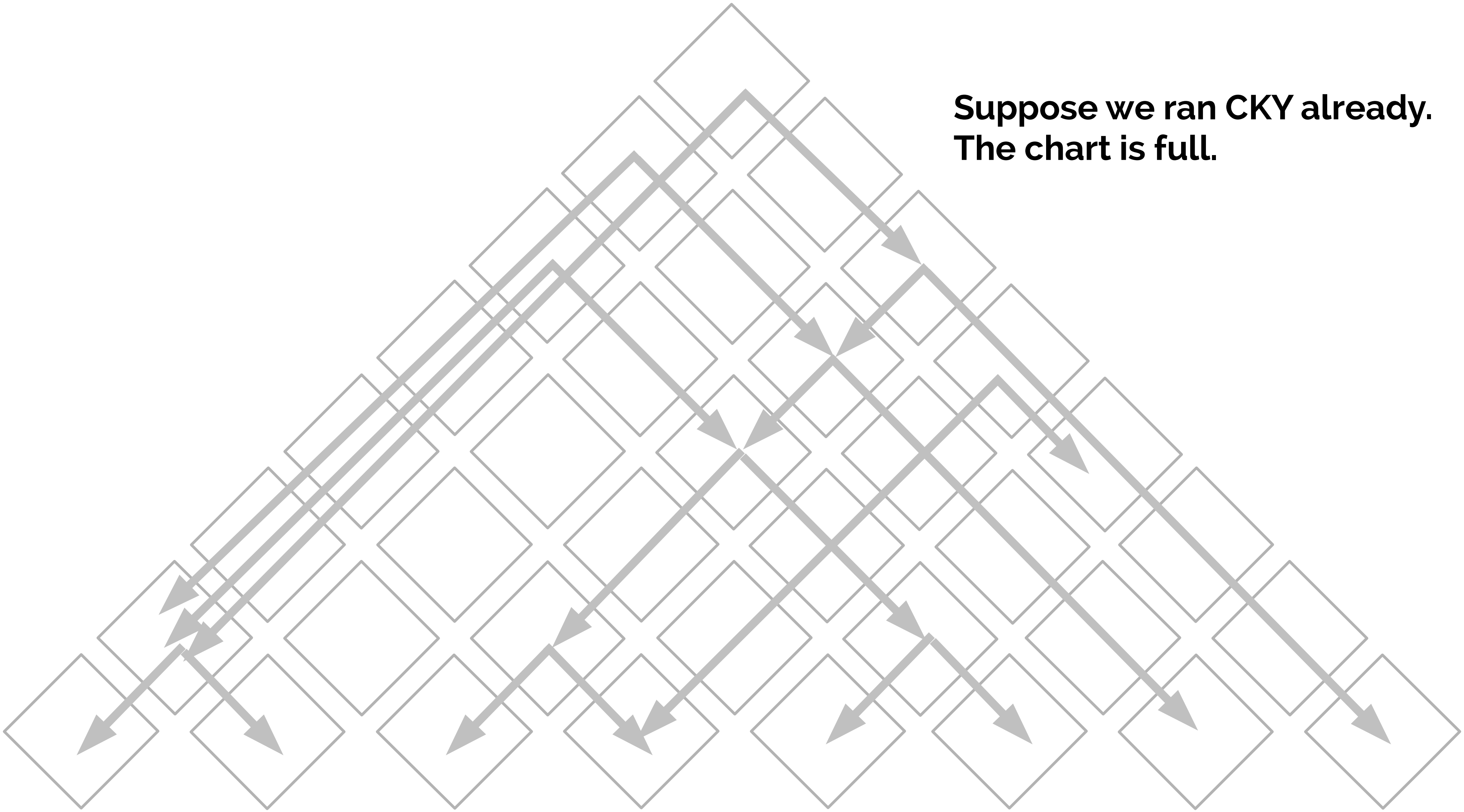
SEARN (Daumé+,2009)
LOLS (Chang+,2015)

Making learning fast

Changeprop

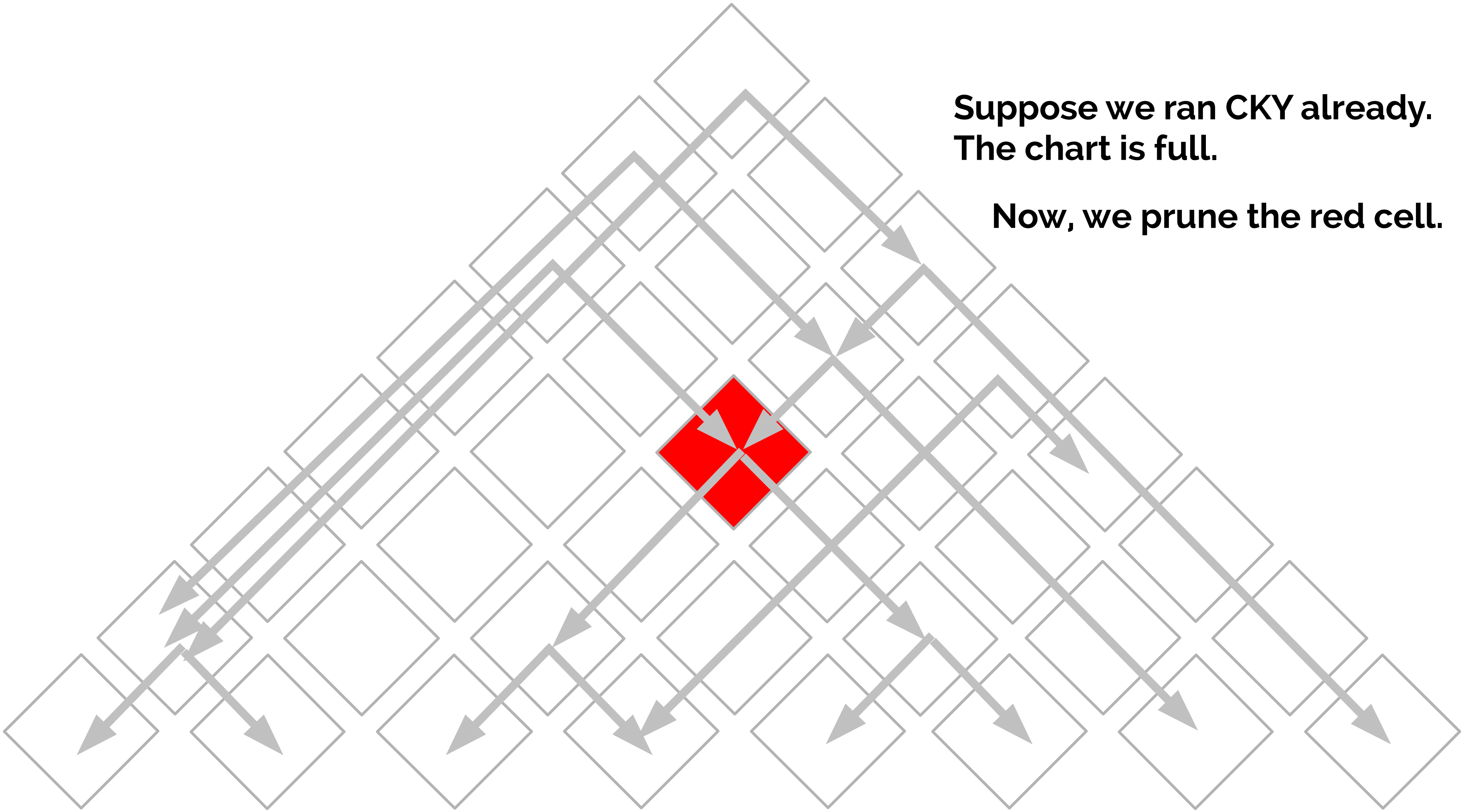
Like a Makefile for parse charts...

Changeprop



Suppose we ran CKY already.
The chart is full.

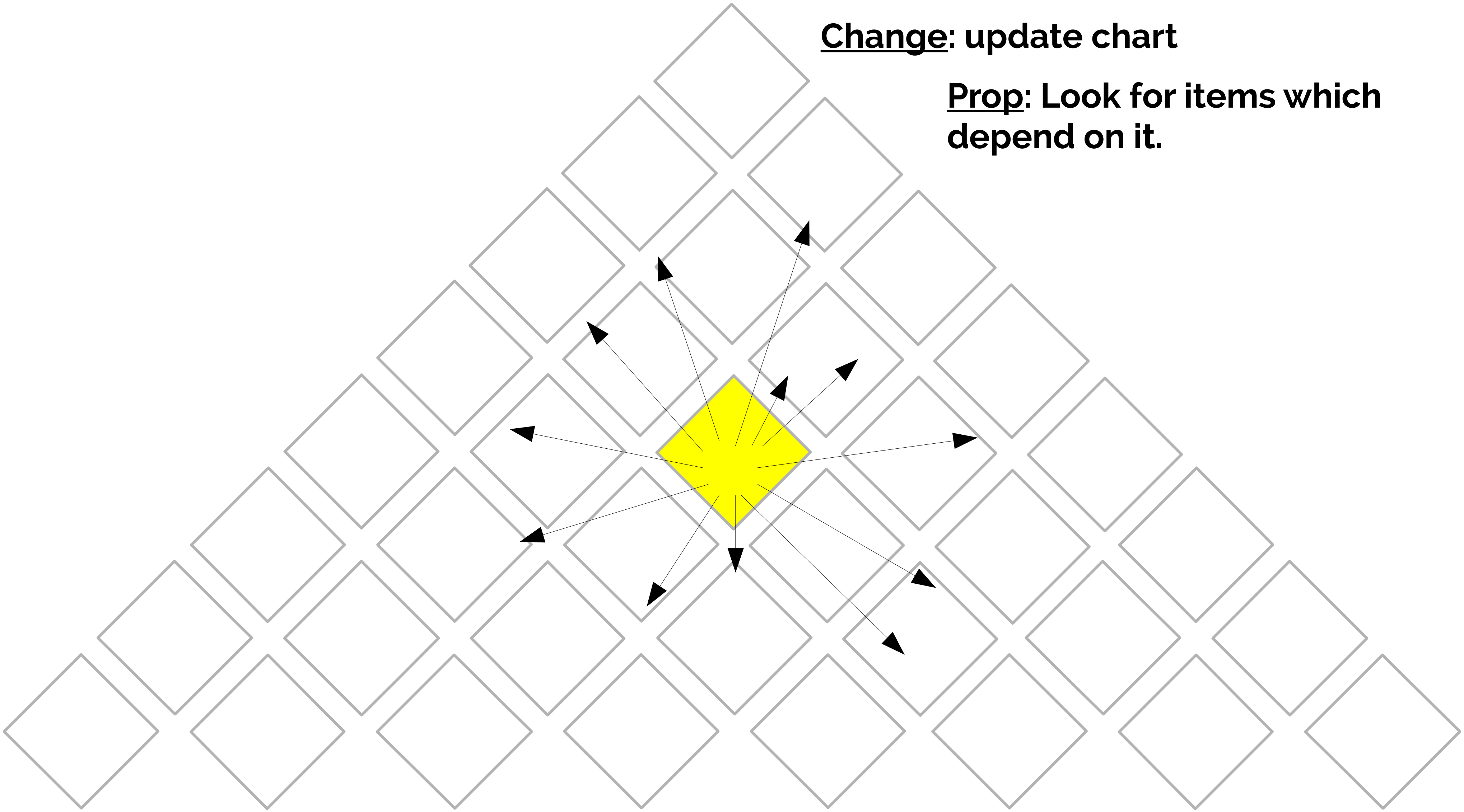
Changeprop



Suppose we ran CKY already.
The chart is full.

Now, we prune the red cell.

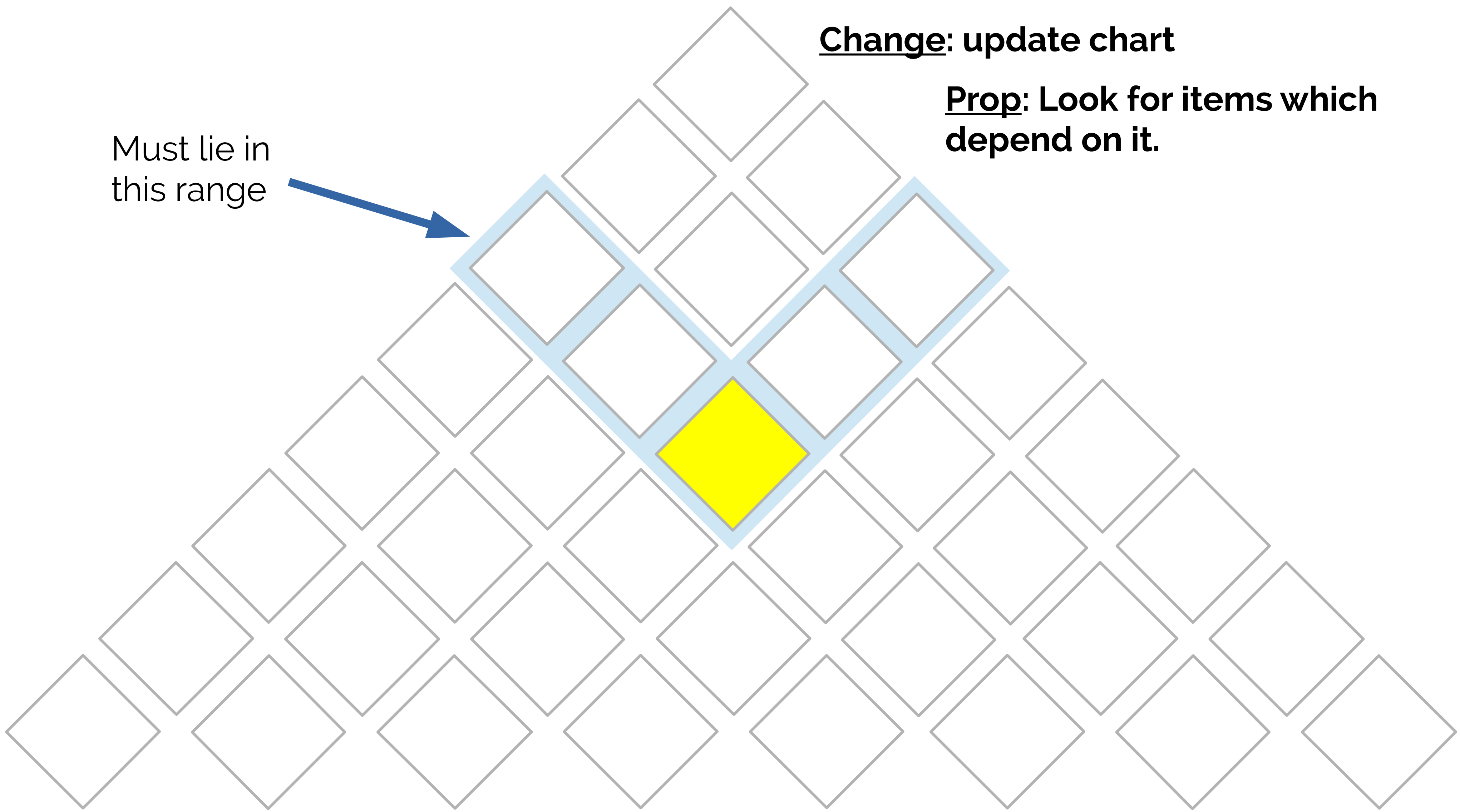
Changeprop



<u>Change</u>: update chart

<u>Prop</u>: Look for items which depend on it.

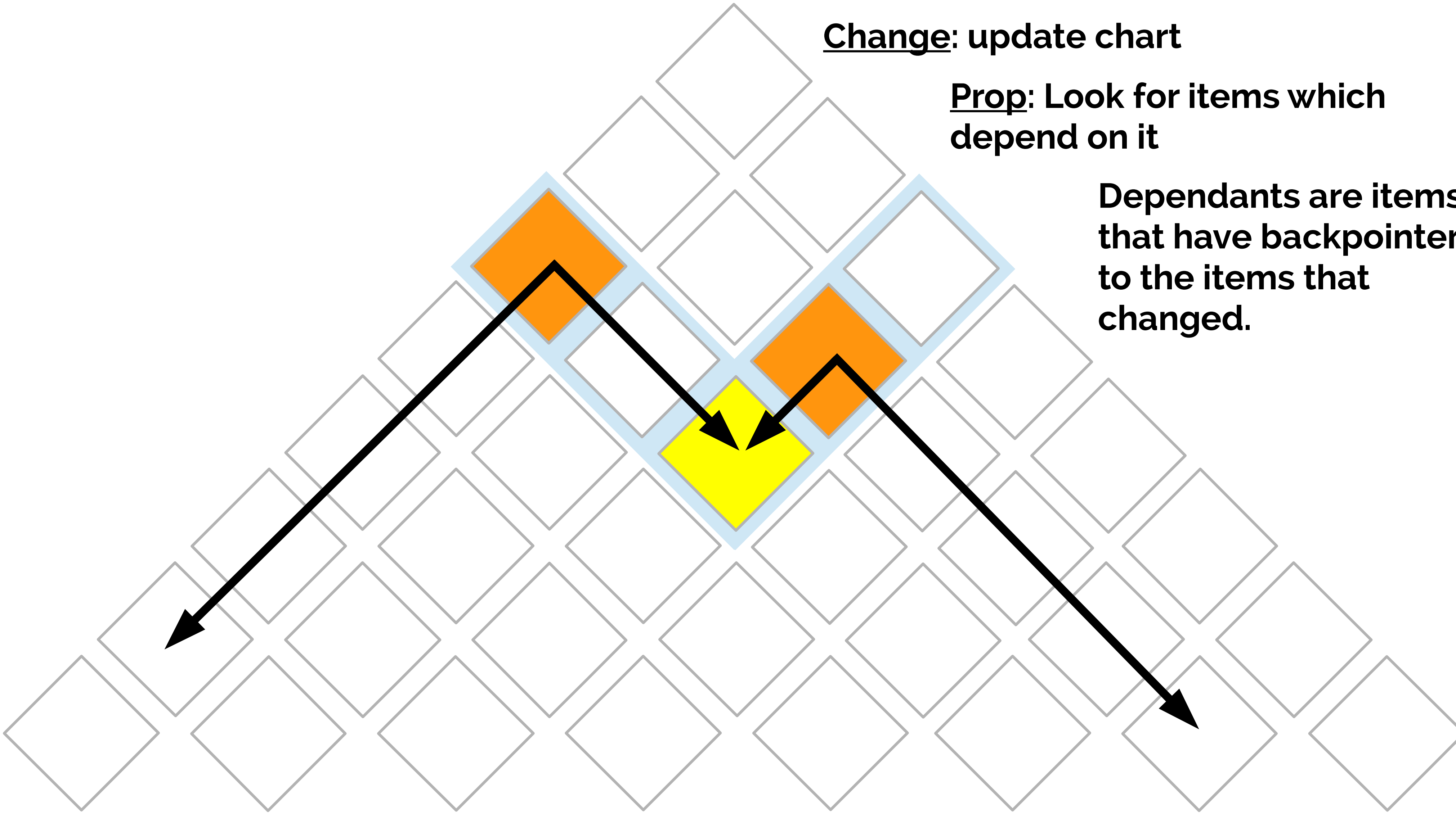
Changeprop



Change: update chart

Prop: Look for items which depend on it.

Changeprop

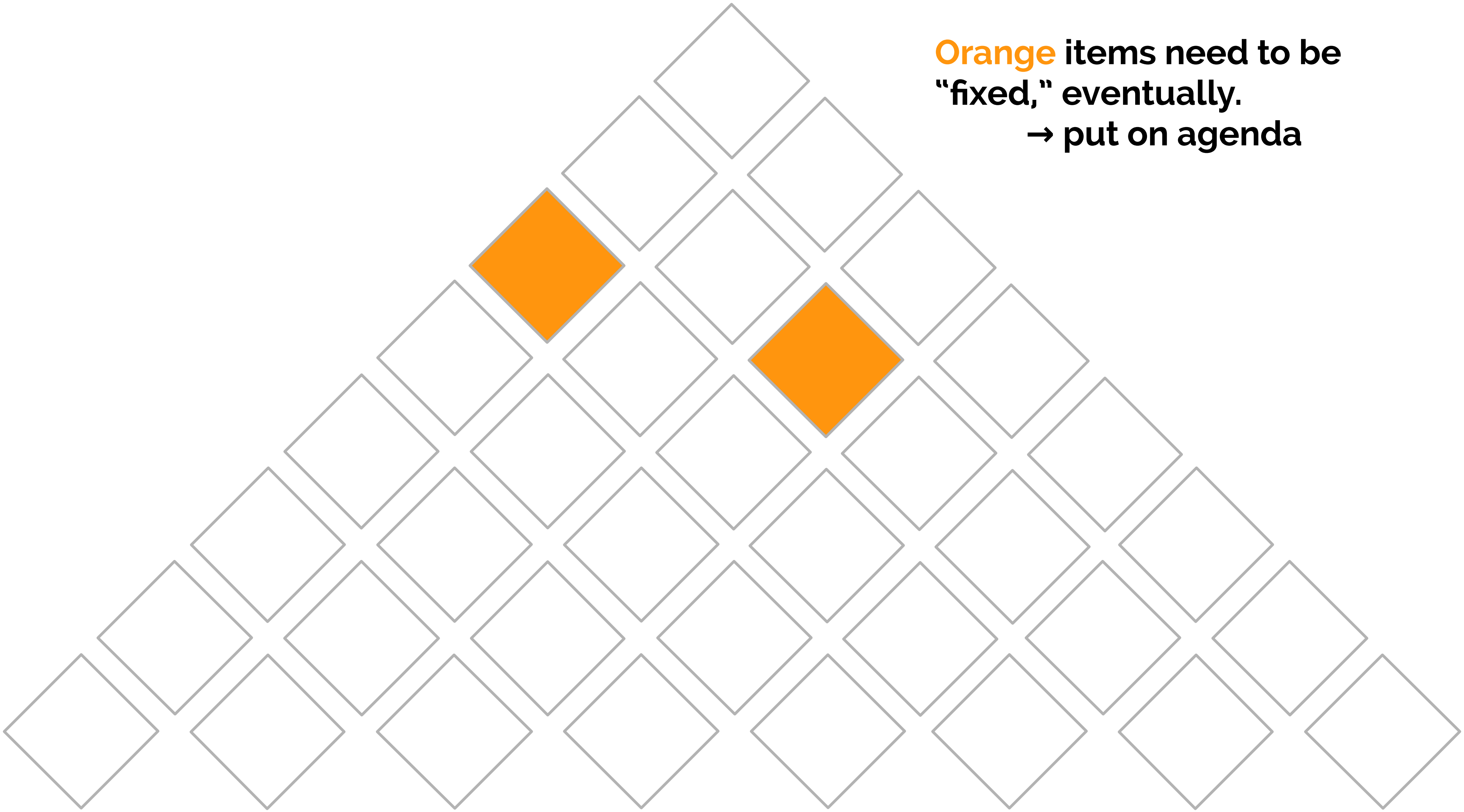


Change: update chart

Prop: Look for items which depend on it

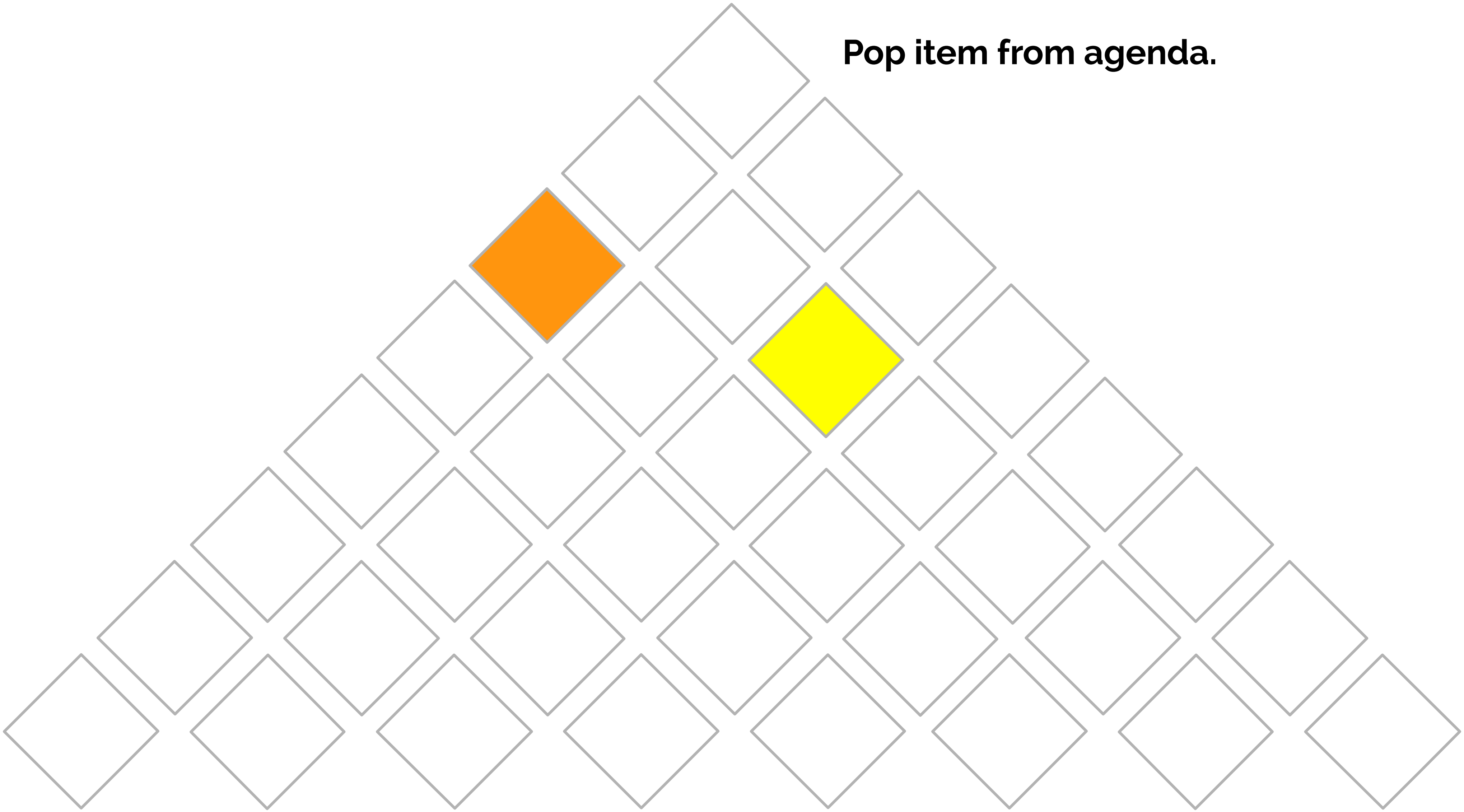
Dependants are items that have backpointers to the items that changed.

Changeprop



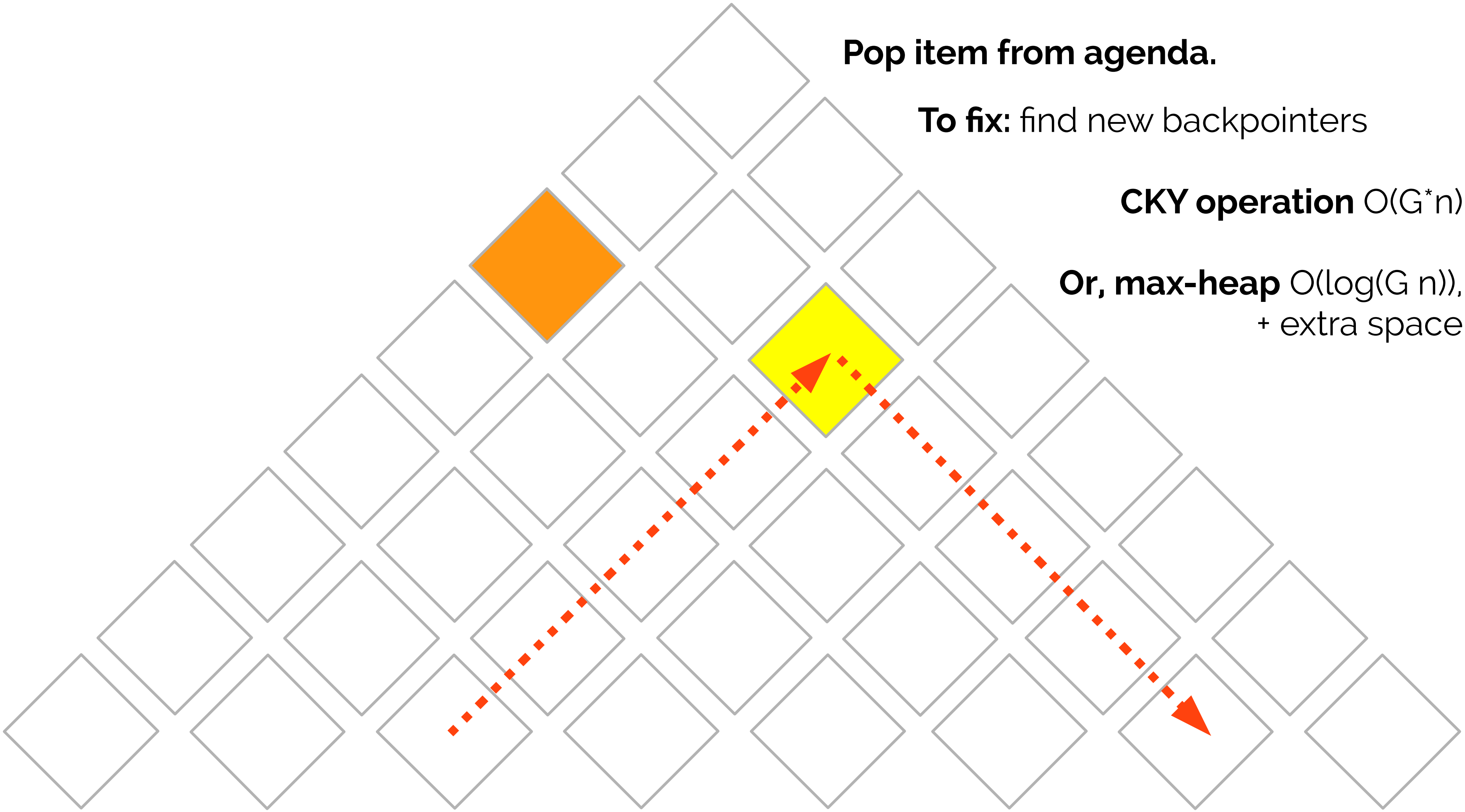
Orange items need to be "fixed," eventually.
→ put on agenda

Changeprop



Pop item from agenda.

Changeprop



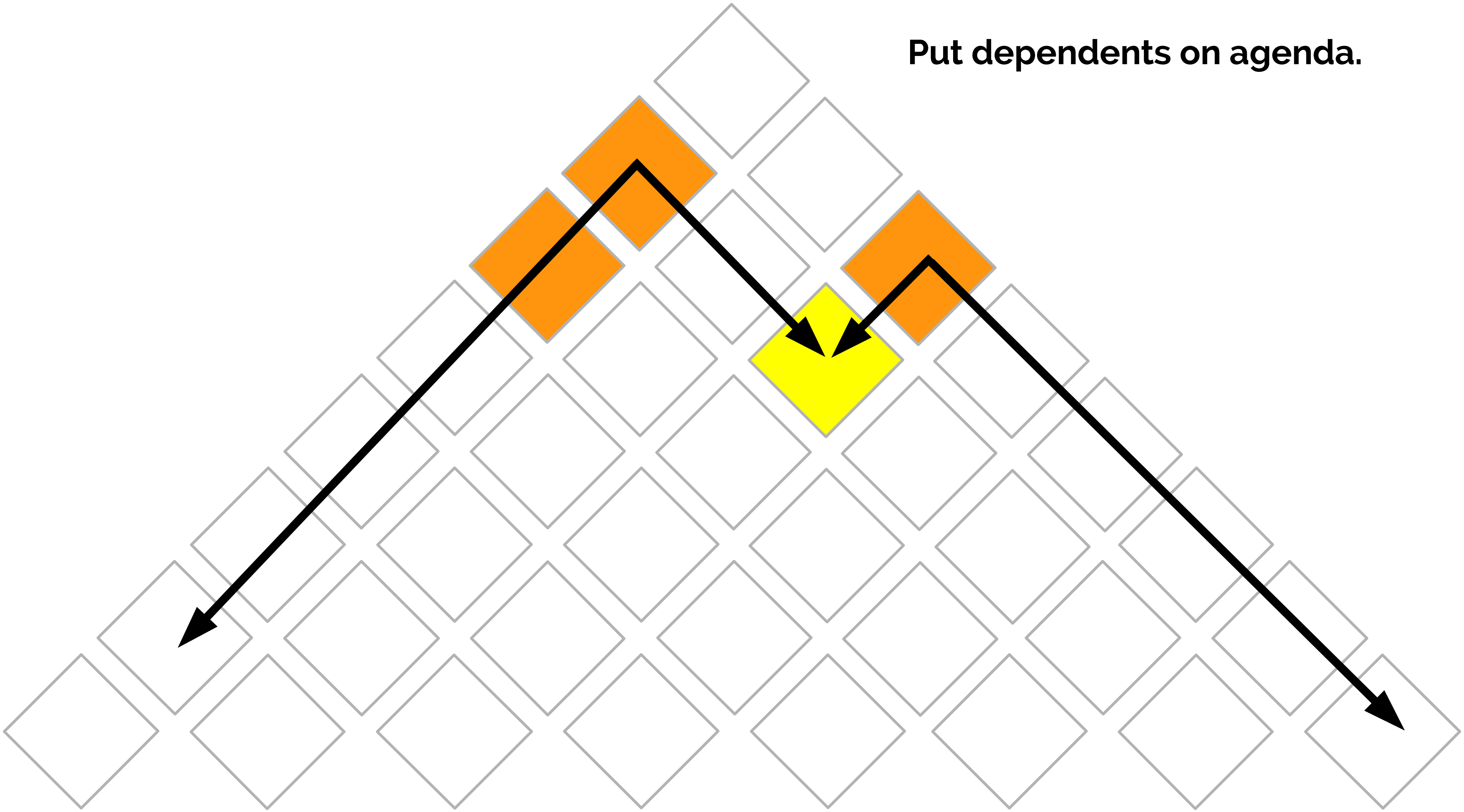
Pop item from agenda.

To fix: find new backpointers

CKY operation O(G*n)

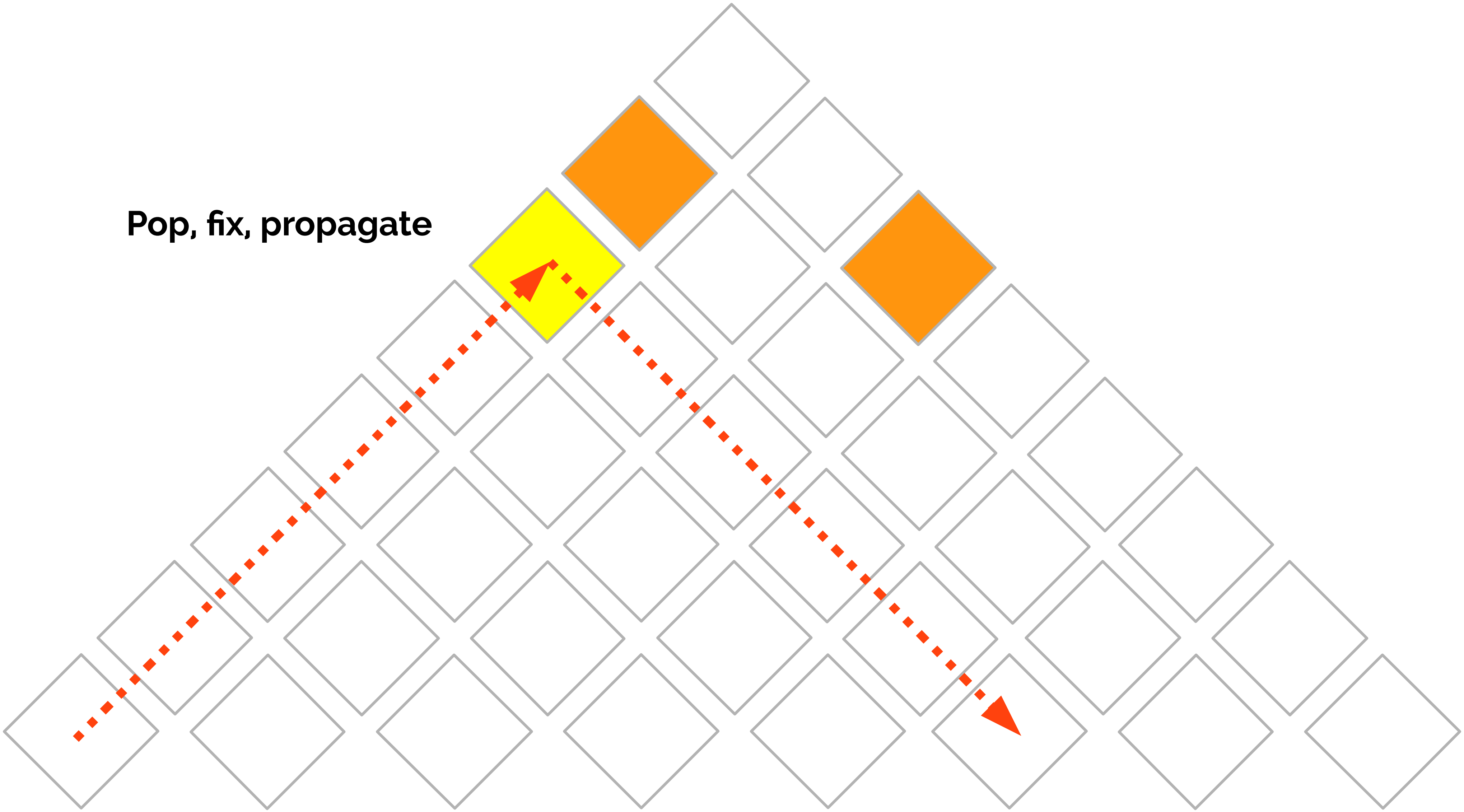
Or, max-heap O(log(G n)),
+ extra space

Changeprop

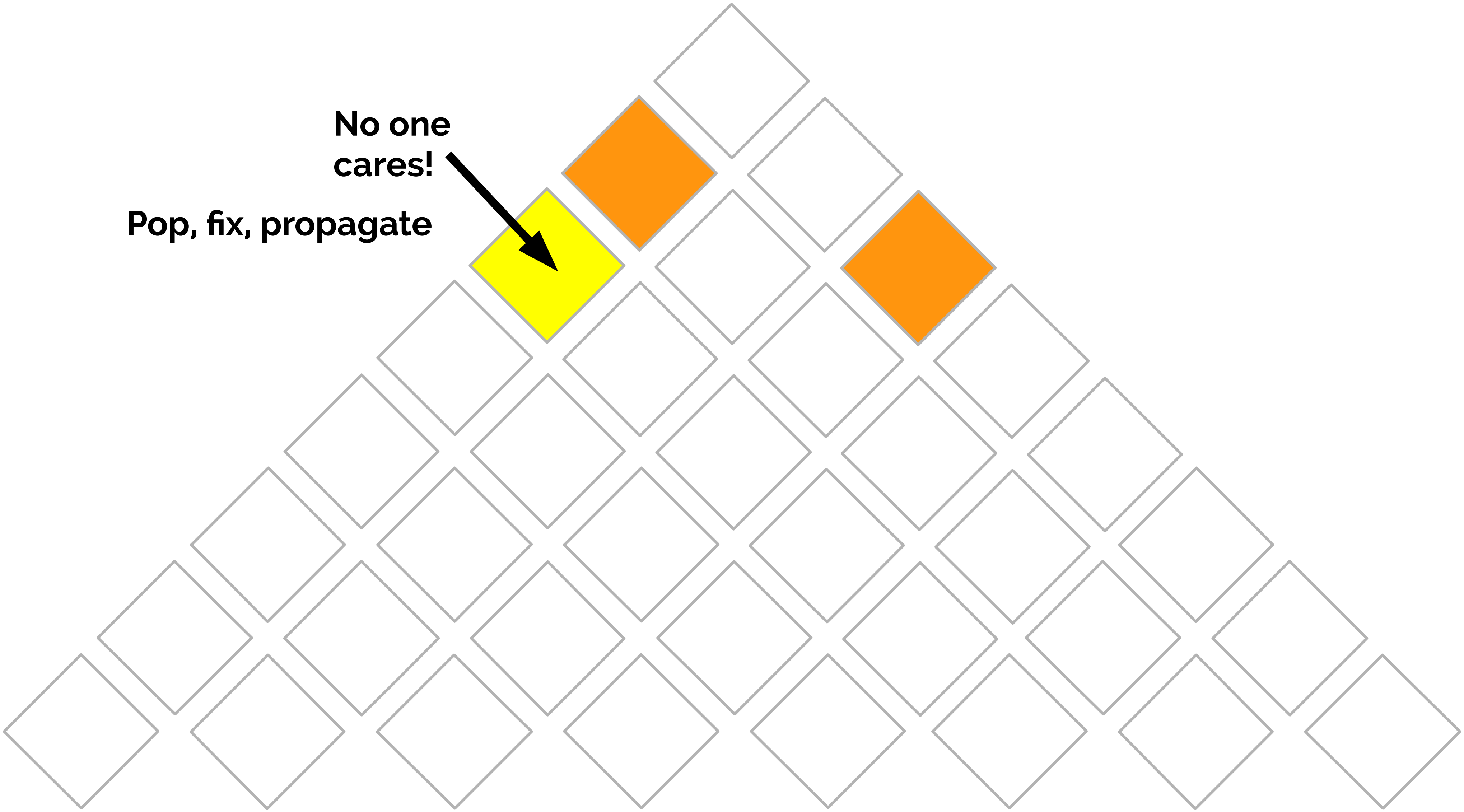


Put dependents on agenda.

Changeprop



Changeprop



Changeprop

And so on...
Until agenda is empty

Changeprop


And so on...
Until agenda is empty
Evaluate reward on converged chart

Changeprop

And so on…
Until agenda is empty

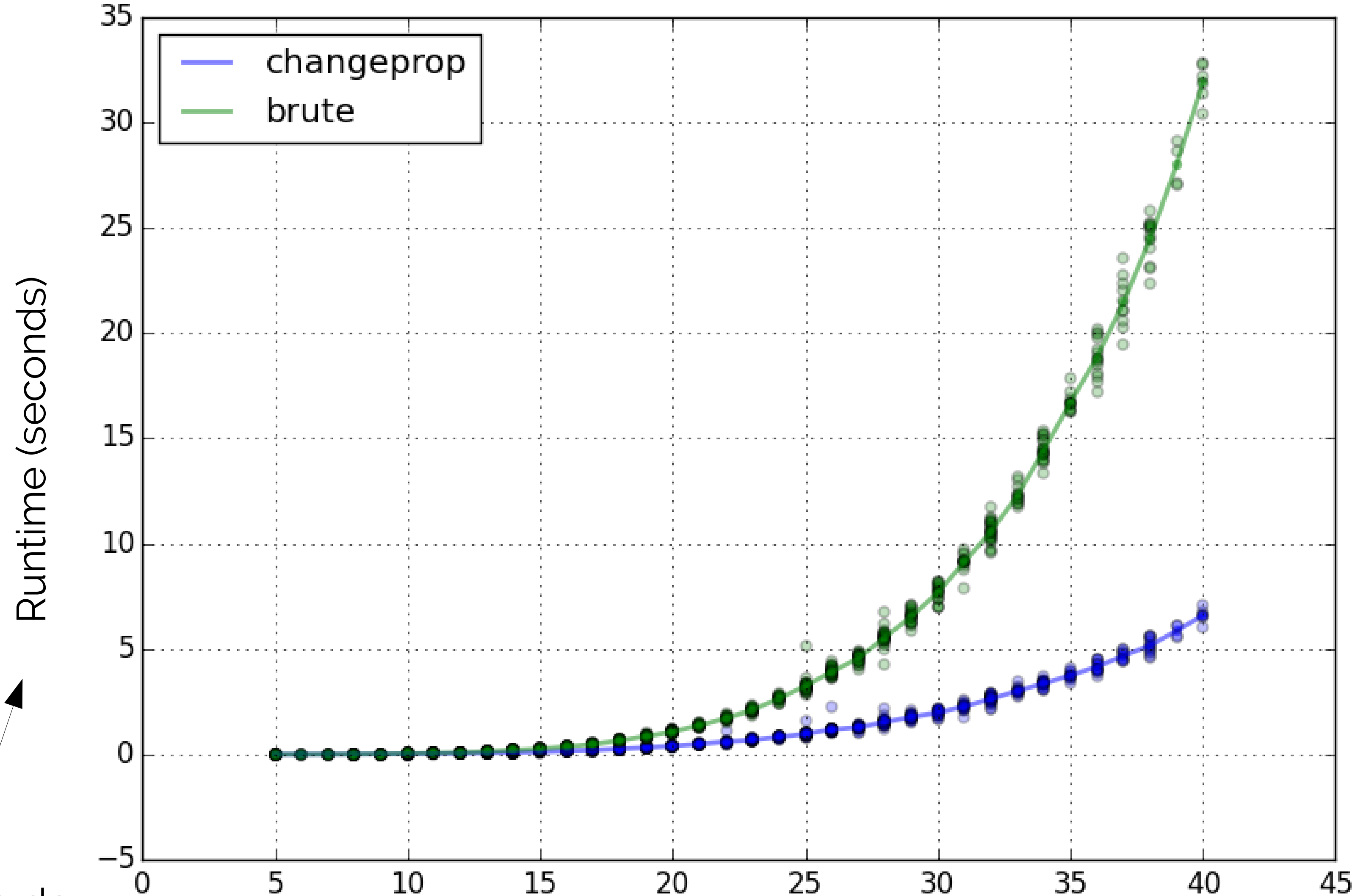
Evaluate reward on converged chart

What about unprune?

Changeprop

- Like a Makefile for parse charts → fast because changes are sparse.
- Leverages structure of underlying computation so that we can efficiently propagate updates.
- Not an asymptotic speed up, but works very well in practice.

Empirical comparison



Time to do all rollouts.

1K sentences, 5 < len <= 40, not much pruning

Dynamic program

Dynamic program

Rollouts are a lot like a gradient...

$$\frac{\Delta r}{\Delta \pi_j}$$

Dynamic program

Expected reward:

$$r\left(\operatorname*{argmax}_{d \in D} p(d)\right) \approx \sum_{d \in D} r(d)\, p(d)$$

Dynamic program

Expected reward:

$$r\left(\underset{d \in D}{\operatorname{argmax}}\, p(d)\right) \approx \sum_{d \in D} r(d)\, p(d)$$

Sample a tree from the pruned forest instead of argmax

Dynamic program

Expected reward:

$$r\left(\underset{d\in D}{\operatorname{argmax}}\, p(d)\right)\approx\sum_{d\in D} r(d)\,p(d)$$

Differentiable: If we change one edge's value a little bit. The gradient tells us how reward changes.

$$r(\vec{k}+\varepsilon\cdot\vec{1}_e)\approx r(\vec{k})+\varepsilon\cdot\frac{\partial r}{\partial k_e}$$

Sample a tree from the pruned forest instead of argmax

Dynamic program

Expected reward:

$$r\left(\underset{d\in D}{\operatorname{argmax}}\, p(d)\right)\approx\sum_{d\in D} r(d)\,p(d)$$

Differentiable: If we change one edge's value a little bit. The gradient tells us how reward changes.

$$r\left(\vec{k}+\varepsilon\cdot\vec{1}_e\right)\approx r\left(\vec{k}\right)+\varepsilon\cdot\frac{\partial r}{\partial k_e}$$

Sample a tree from the pruned forest instead of argmax

Inaccurate for large ε

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum_{d \in D} r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum_{d \in D} r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Now, what happens if we change just one edge?

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Multi-linear (by example)

$$f(x, y, z) = xyz$$

Now, what happens if we change just one edge?

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Multi-linear (by example)

$$f(x, y, z) = xyz$$

$$f(x+\varepsilon, y, z)$$

Now, what happens if we change just one edge?

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Multi-linear (by example)

$$f(x, y, z) = xyz$$

$$f(x+\varepsilon, y, z) = (x+\varepsilon)\, yz$$

Now, what happens if we change just one edge?

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum r(d)\, \bar{p}(d)$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Now, what happens if we change just one edge?

Multi-linear (by example)

$$f(x, y, z) = xyz$$

$$\begin{aligned} f(x+\varepsilon, y, z) &= (x+\varepsilon)\, yz \\ &= xyz + \varepsilon\, yz \\ &= f(x, y, z) + \varepsilon \frac{\partial f}{\partial x} \end{aligned}$$

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$\bar{r} = \sum_{d \in D} r(d)\, \bar{p}(d)$$
$$= \sum_{d \in D} r(d) \prod_{e \in d} k_e$$

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Now, what happens if we change just one edge?

Multi-linear function of edge weights!

Dynamic program

Expected reward:

$$r = \sum_{d \in D} r(d)\, p(d)$$

Break-up into numerator and denominator.

$$Z = \sum_{d \in D} \prod_{e \in d} k_e$$

Now, what happens if we change just one edge?

Taylor expansion works!

$$\bar{r}(\vec{k} + \varepsilon \cdot \vec{1}_e) = \bar{r}(\vec{k}) + \varepsilon \cdot \frac{\partial \bar{r}}{\partial k_e}$$

Same for Z

Quotient of separate expansions gives us exact expected reward for any perturbation.

Dynamic program

Loose ends

- Efficiently computing gradients

- Pruning affects more than one edge at a time.

- Want one-best, not expected

Dynamic program

Loose ends

- Efficiently computing gradients

Fast algorithms for decomposable rewards, e.g., Second-order inside-outside algorithm (Li & Eisner,09)

$$r(d) = \sum_{e \in d} r_e$$

- Pruning affects more than one edge at a time.

- Want one-best, not expected

Dynamic program

Loose ends

- Efficiently computing gradients

Fast algorithms for decomposable rewards, e.g., Second-order inside-outside algorithm (Li & Eisner,09)

$$r(d) = \sum_{e \in d} r_e$$

- Pruning affects more than one edge at a time.

Need to maintain multi-linearity → pruning factors appear at most once per derivation.

- Want one-best, not expected

Dynamic program

Loose ends

- Efficiently computing gradients

Fast algorithms for decomposable rewards, e.g., Second-order inside-outside algorithm (Li & Eisner,09)

$$r(d) = \sum_{e \in d} r_e$$

- Pruning affects more than one edge at a time.

Need to maintain multi-linearity → pruning factors appear at most once per derivation.

- Want one-best, not expected

Annealing – a general trick for interpolating between expectation and maximization.

$$\lim_{\gamma \to \infty} \frac{1}{Z_\gamma} \sum_{d \in D} r(d)\, p(d)^\gamma = r\left(\operatorname*{argmax}_{d \in D} p(d)\right)$$

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\, n^3 + n^2)$ algorithm.

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\ n^3 + n^2)$ algorithm.
- Fast and exact for decomposable reward functions

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\,n^3 + n^2)$ algorithm.

- Fast and exact for decomposable reward functions

  - Accuracy :-)

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\ n^3 + n^2)$ algorithm.

- Fast and exact for decomposable reward functions

  - Accuracy :-)

  - Runtime :-(

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\ n^3 + n^2)$ algorithm.
- Fast and exact for decomposable reward functions
  - Accuracy :-)
  - Runtime :-(
    Boolean version of changeprop for runtime is super fast.

Dynamic program

- How it works: a carefully applied Taylor expansion gives us an $O(G\, n^3 + n^2)$ algorithm.

- Fast and exact for decomposable reward functions

  – Accuracy :-)

  – Runtime :-(
  Boolean version of changeprop for runtime is super fast

Sorry, no benchmark plot, yet.

Experiments

TODO

- See paper for updated experimental results.

Conclusions

- Modeling end-to-end performance leads to better policies

Conclusions

- Modeling end-to-end performance leads to better policies
- LOLS works pretty well for training

Conclusions

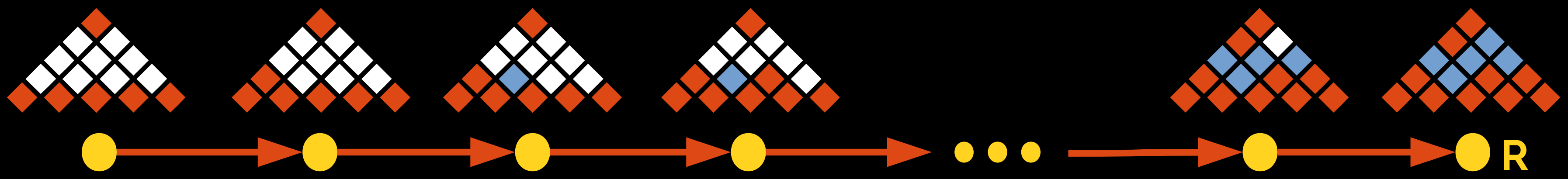
- Modeling end-to-end performance leads to better policies
- LOLS works pretty well for training
- Training under end-to-end objective requires running inference millions of times.

Conclusions

- Modeling end-to-end performance leads to better policies
- LOLS works pretty well for training
- Training under end-to-end objective requires running inference millions of times.

    We presented efficient algorithms for repeated inference: change propagation and dynamic programming.

Thanks!



Twitter: @xtimv
Blog: http://timvieira.github.io/blog

Backup slides

Runtime with Jacobian

$$\text{runtime}(\pi + \varepsilon \mathbf{1}_j) = \sum_x \mathbf{1}\left[\beta_x(\pi) + \varepsilon \cdot \frac{\partial \beta_x(\pi)}{\partial \pi_j} > 0\right]$$

General trick: apply expansion before nonlinearity.

Needs T Jacobian-vector products → asymptotically slower than running inference T times.

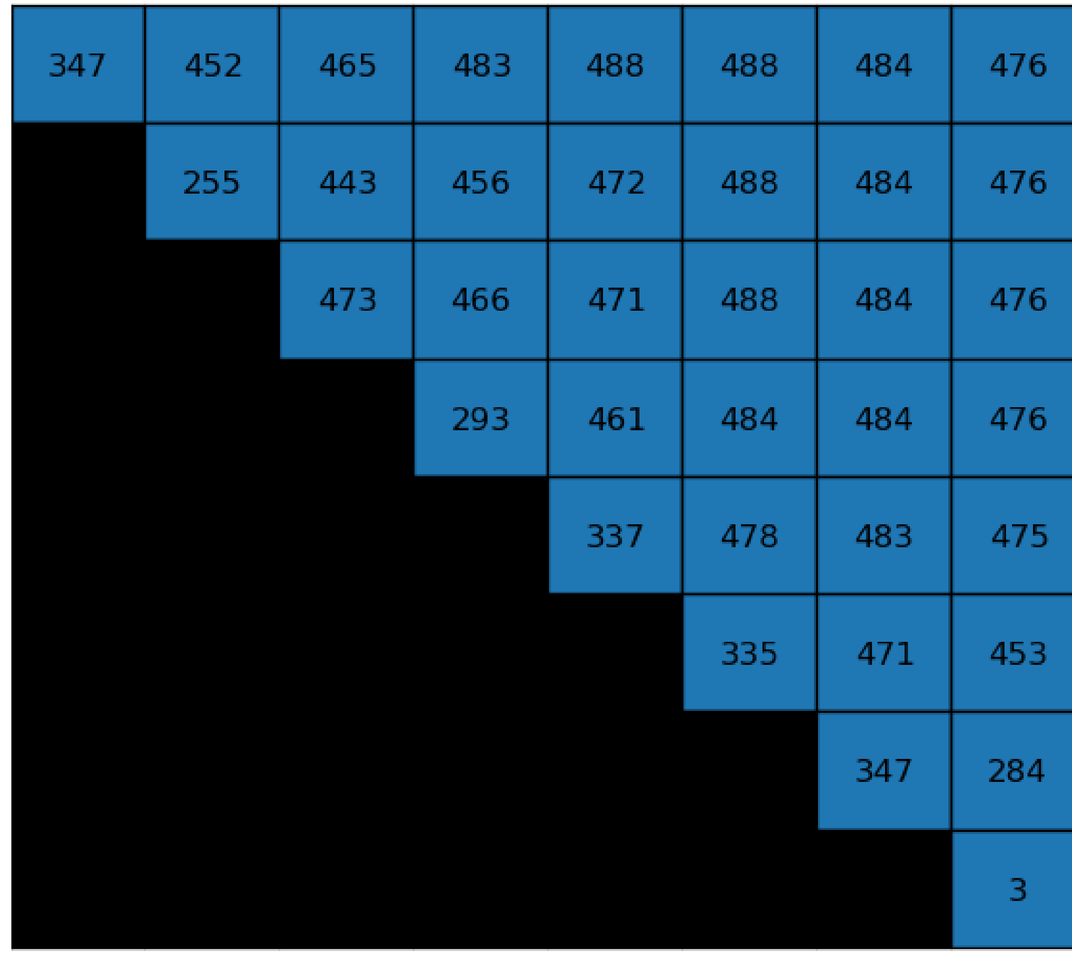
Just want Booleans in the end → changeprop will be faster.

Runtime?

O(G*n^3)

O(n^2) cells
O(G*n) time to fill each

#symbols in each cell of a realistic grammar



Runtime?

O(G*n^3)

O(n^2) cells
O(G*n) time to fill each

How many grammar lookups
to fill this cell

255*484

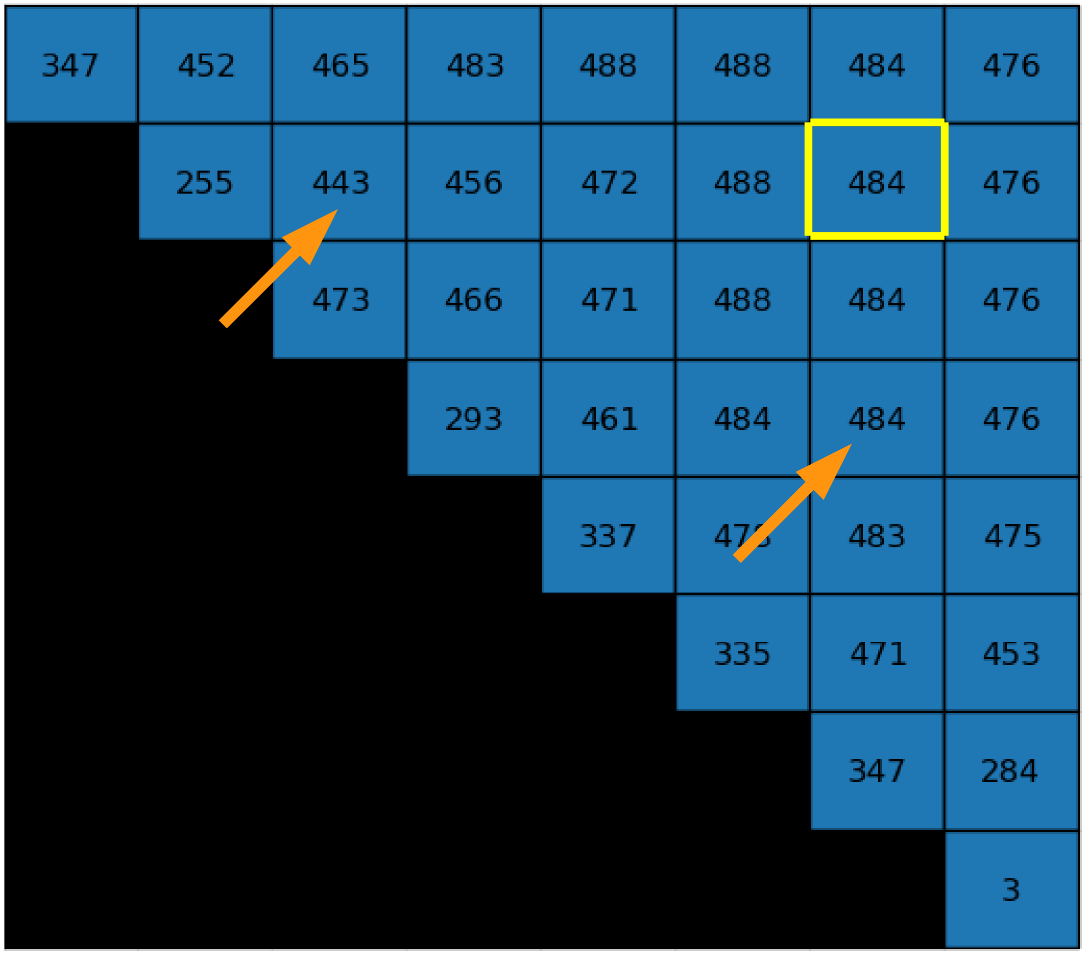
#symbols in each cell of a realistic grammar

347	452	465	483	488	488	484	476
	255	443	456	472	488	484	476
		473	466	471	488	484	476
			293	461	484	484	476
				337	478	483	475
					335	471	453
						347	284
							3

Runtime?

#symbols in each cell of a realistic grammar

O(G*n^3)



O(n^2) cells
O(G*n) time to fill each

How many grammar lookups
to fill this cell

255*484
+ 443*484

Runtime?

#symbols in each cell of a realistic grammar

O(G*n^3)

347	452	465	483	488	488	484	476
	255	443	456	472	488	484	476
		473	466	471	488	484	476
			293	461	484	484	476
				337	478	483	475
					335	471	453
						347	284
							3

O(n^2) cells
O(G*n) time to fill each

How many grammar lookups to fill this cell

255*484
+ 443*484
+ 456*483

Runtime?

#symbols in each cell of a realistic grammar

347	452	465	483	488	488	484	476
	255	443	456	472	488	484	476
		473	466	471	488	484	476
			293	461	484	484	476
				337	478	483	475
					335	471	453
						347	284
							3

O(G*n^3)

O(n^2) cells
O(G*n) time to fill each

How many grammar lookups to fill this cell

255*484
+ 443*484
+ 456*483
+ 472*471

Runtime?

#symbols in each cell of a realistic grammar

$O(G^*n^\wedge 3)$

$O(n^\wedge 2)$ cells
$O(G^*n)$ time to fill each

How many grammar lookups to fill this cell

347	452	465	483	488	488	484	476
	255	443	456	472	488	$\boxed{484}$	476
		473	466	471	488	484	476
			293	461	484	484	476
				337	478	483	475
					335	471	453
						347	284
							3

255*484
+ 443*484
+ 456*483
+ 472*471
+ 488*347

Runtime?

#symbols in each cell of a realistic grammar

$O(G*n^3)$

$O(n^2)$ cells
$O(G*n)$ time to fill each

How many grammar lookups to fill this cell

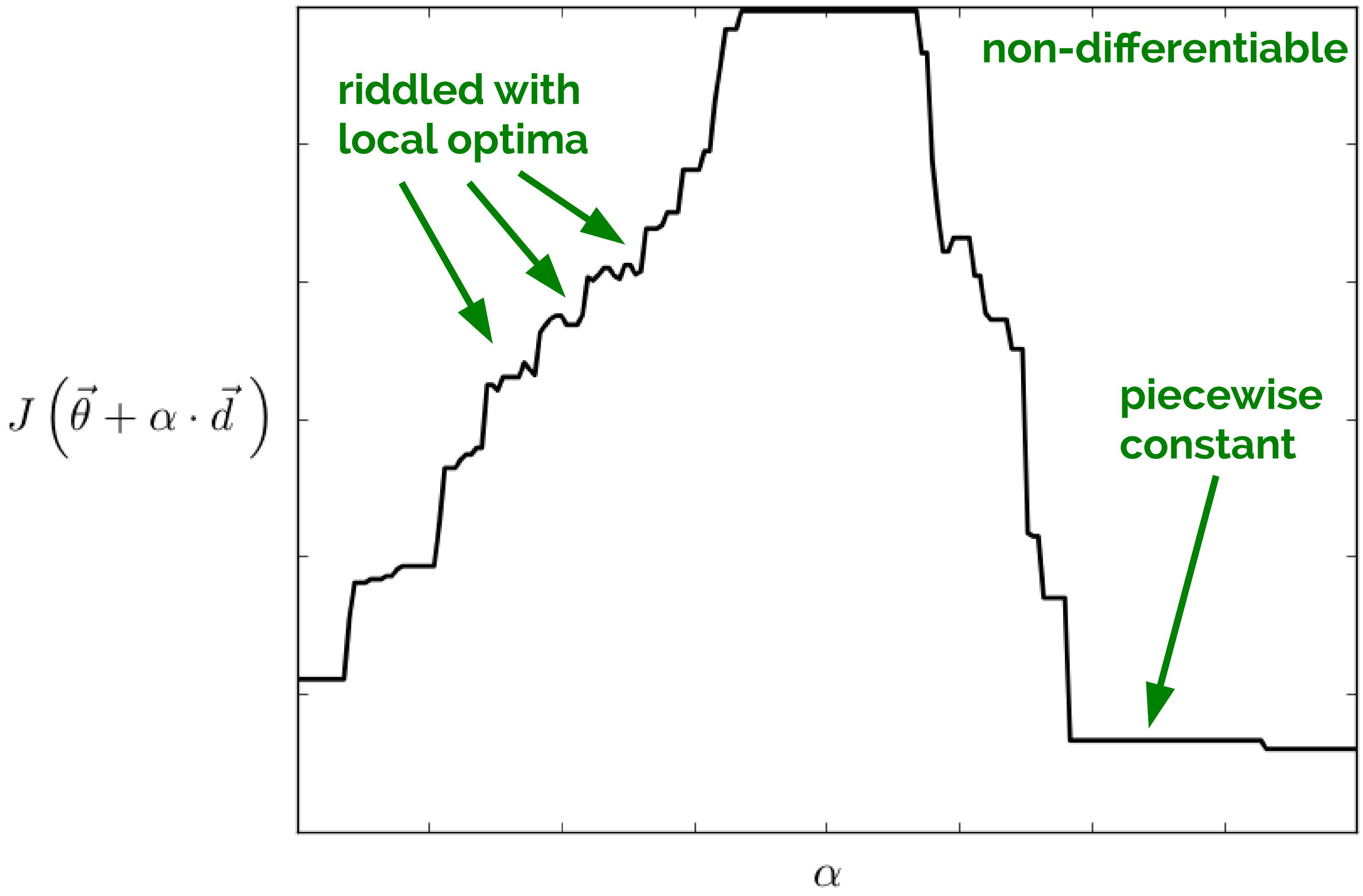
347	452	465	483	488	488	484	476
	255	443	456	472	488	484	476
		473	466	471	488	484	476
			293	461	484	484	476
				337	478	483	475
					335	471	453
						347	284
							3

$$
\begin{aligned}
&255*484 \\
&+ 443*484 \\
&+ 456*483 \\
&+ 472*471 \\
&+ 488*347 \\
&= 949{,}728
\end{aligned}
$$

Almost a million lookups for one cell!

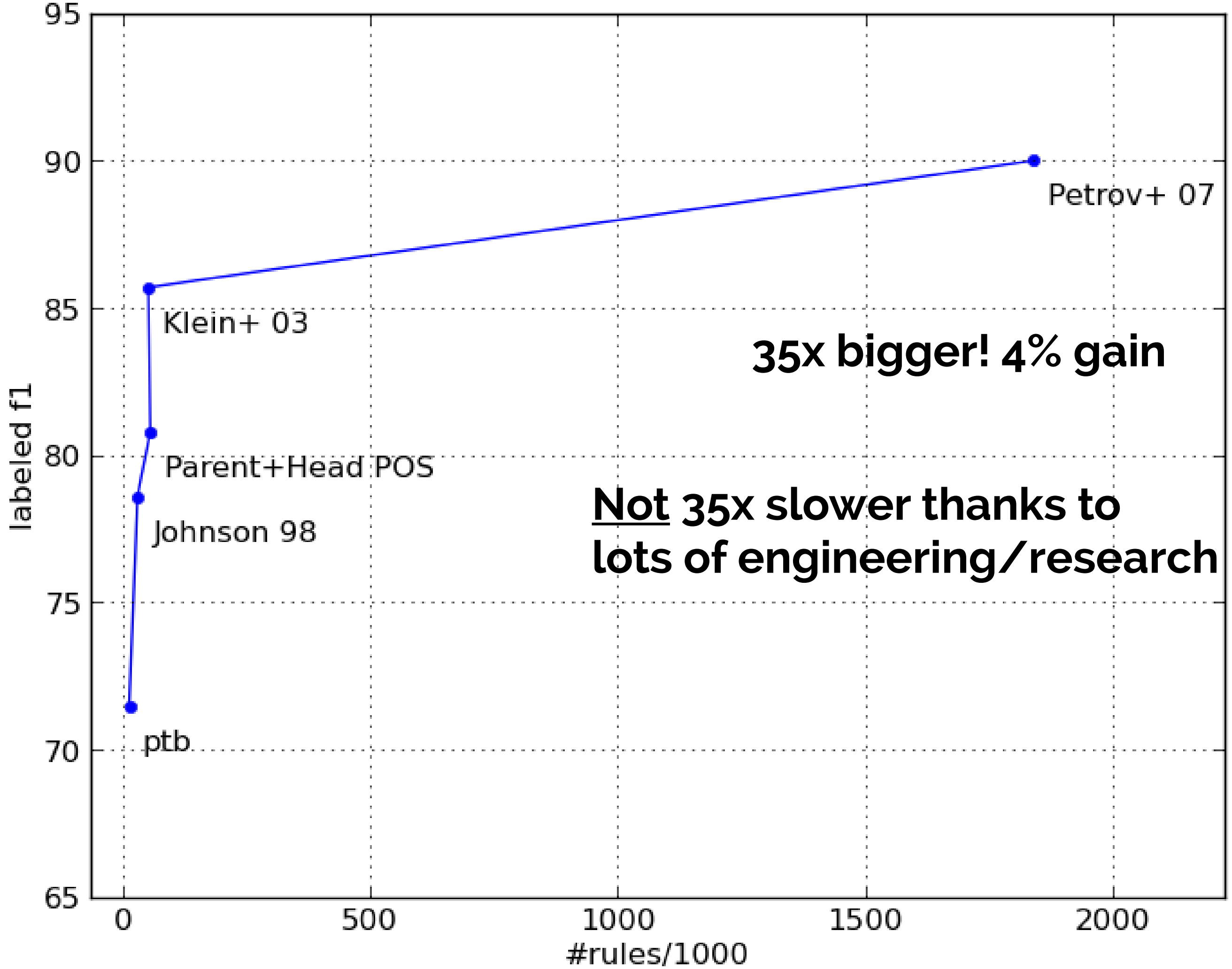
Tricky to optimize

Cross-section of objective, J, along a random direction, d



Pruning is a discrete decision

Diminishing returns

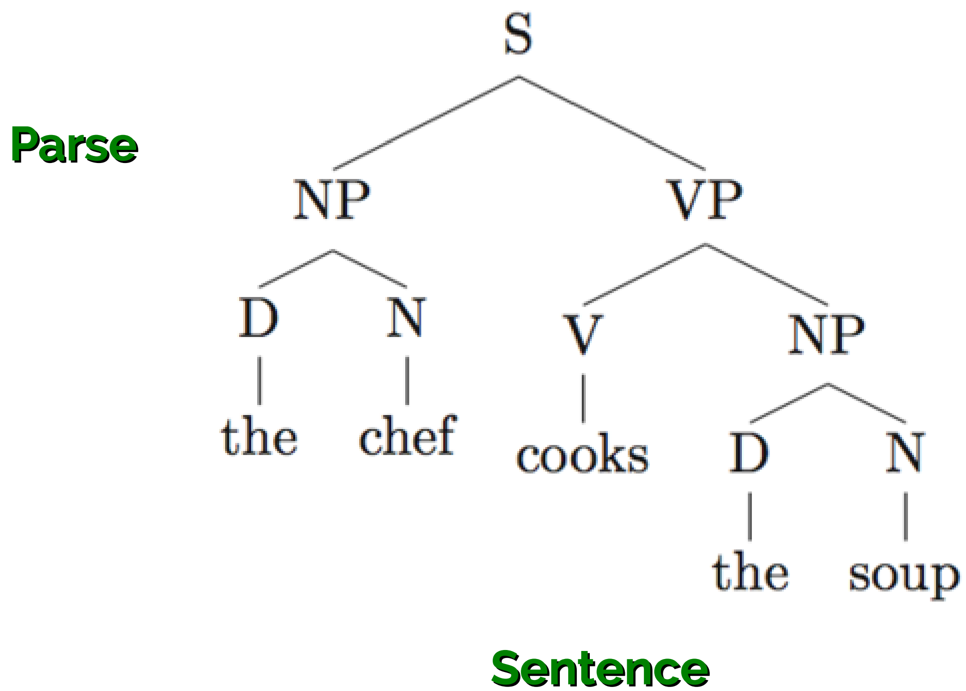


35x bigger! 4% gain

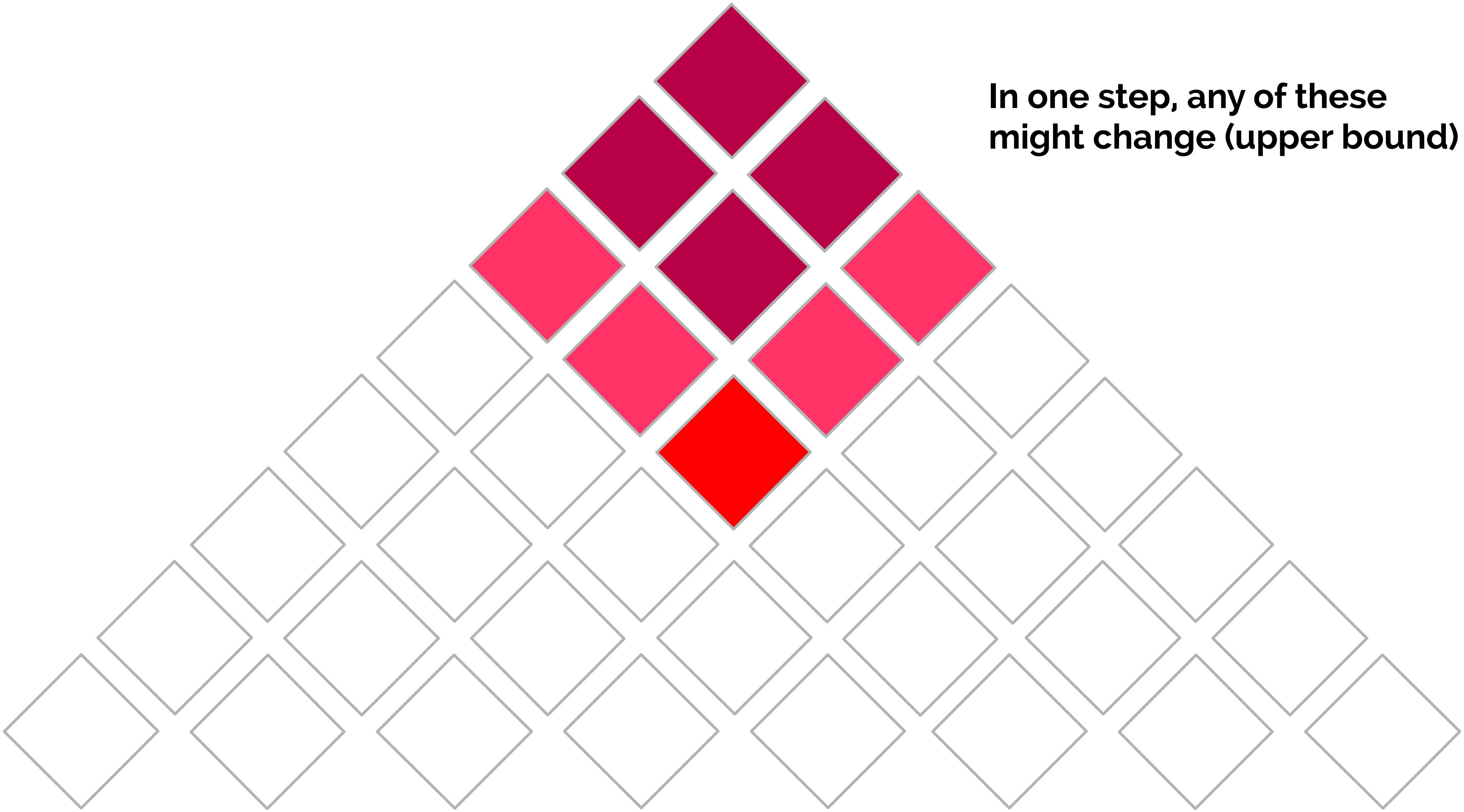
<u>Not</u> 35x slower thanks to lots of engineering/research

Parsing

"diagramming sentences"

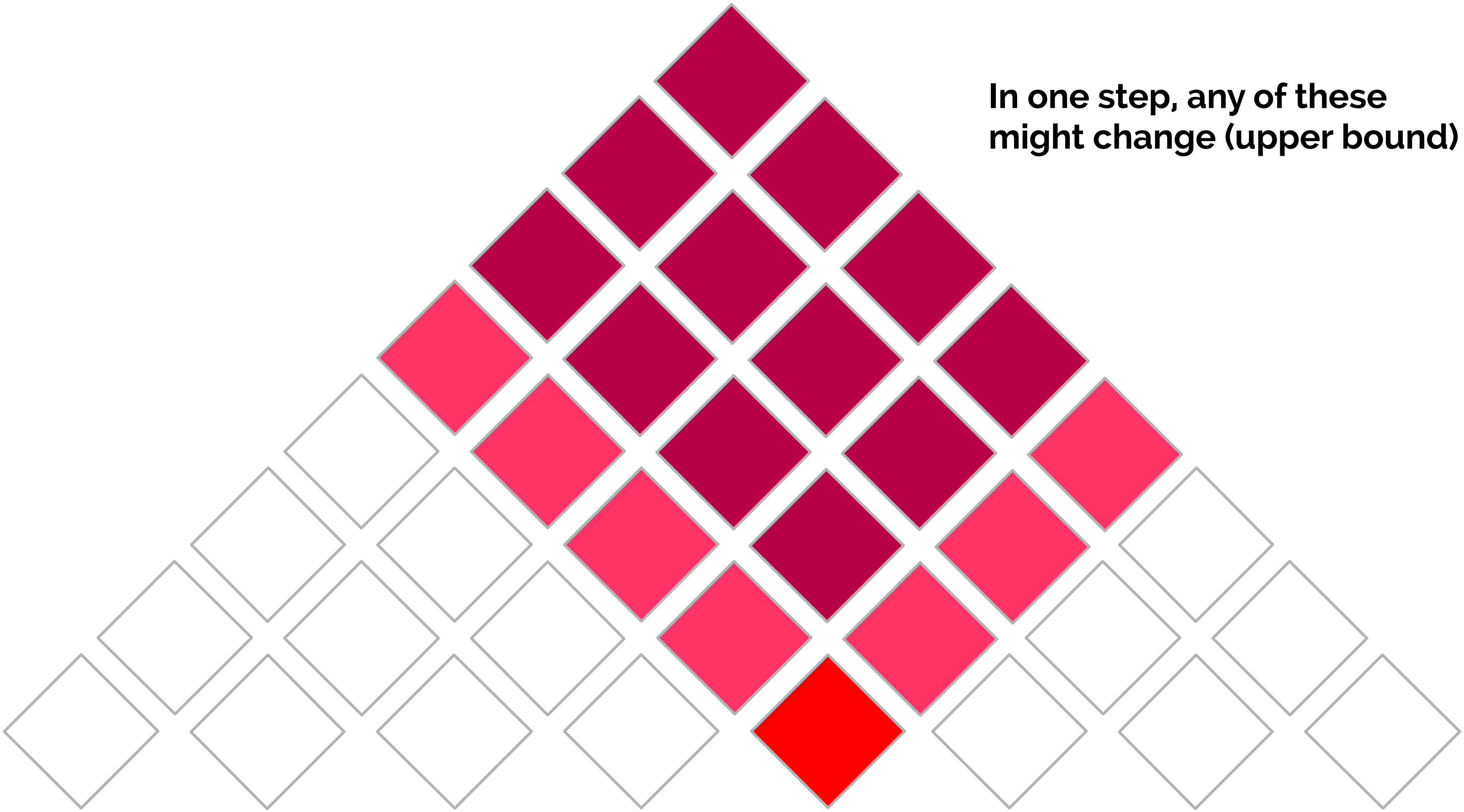


Changeprop



In one step, any of these might change (upper bound)

Changeprop



In one step, any of these might change (upper bound)

Dynamic program (cheat sheet)

Suppose final reward is expected reward instead of one-best

$$r = \bar{r} / Z \qquad Z = \sum_d \prod_{e \in d} k_e$$

(1) To start: what if we change just one edge?

$$\bar{r} = \sum_d r(d) p(d) = \sum_d r(d) \prod_{e \in d} k_e$$

(2) Multi-linear functions of single edge weights

(3) Multi-linear (example): **f(x,y,z) = xyz** not jointly linear, but is linear in x, y or z, separately (i.e., hold others fixed).

(4) No edge appears twice in a given derivation. Note: r(d) can't depend on edge weights.

(5) Compute change with Taylor expansion:

$$\bar{r}(\vec{k} + \varepsilon \cdot \vec{1}_e) = \bar{r}(\vec{k}) + \varepsilon \cdot \frac{\partial \bar{r}}{\partial k_e}$$

Similarly for Z.

(7) Take quotient

(8) Need additive reward to efficiently compute. Use second-order inside-outside or backprop to get gradients.

→ All T=O(n^2) rollouts for the cost of one!

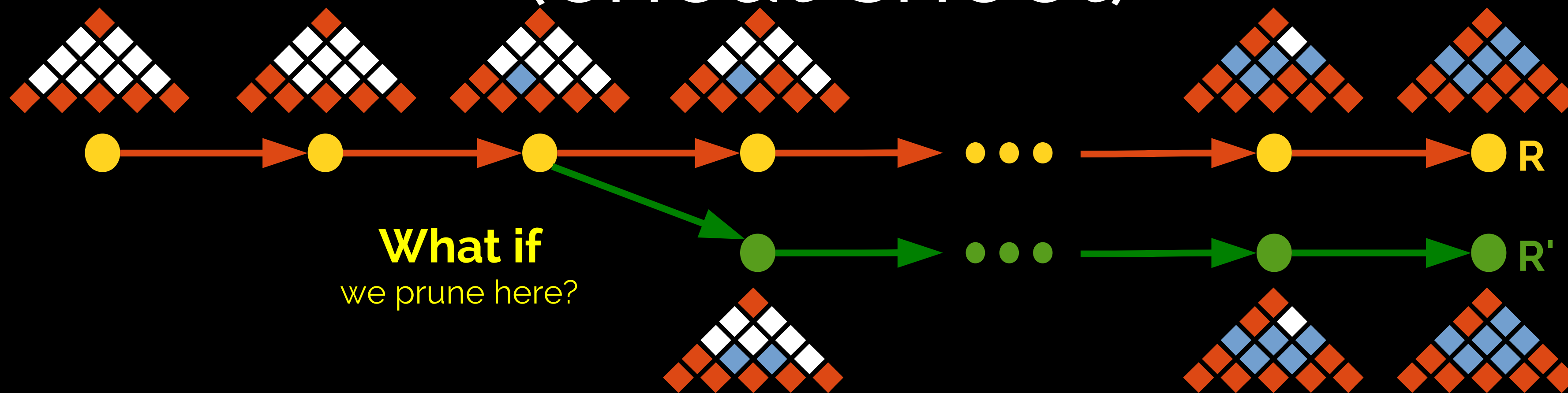
Tweaks:

(9) Pruning changes multiple edges.

(10) Use annealing to recover one-best

Similar trick for runtime requires Jacobian. Not very efficient

Approximate policy iteration (cheat sheet)



Algorithm 2 SEARN algorithm for learning a sequential prediction policy.

$\pi_1 = \text{EXPERT}$
for $i \leftarrow 1$ to iterations **do**
E
  for $j \leftarrow 1$ to minibatch **do**
    sample sentence x_j from training data
    $s_1 \cdots s_T = \text{ROLL-IN}$ with current policy π_i on x_j
    for $s \leftarrow s_1 \cdots s_T$ **do**
      for $a \leftarrow A(s)$ **do**
        $Q_{\pi_i}(s, a) = \text{ROLL-OUT}$ with π_i starting at s
M
  $\pi' \leftarrow \text{TRAIN}(Q_{\pi_i})$
  $\pi_{i+1} \leftarrow (1 - \beta)\pi_i + \beta\pi'$

Dependence on expert decreases quickly with iterations. (relative performance; exploration)

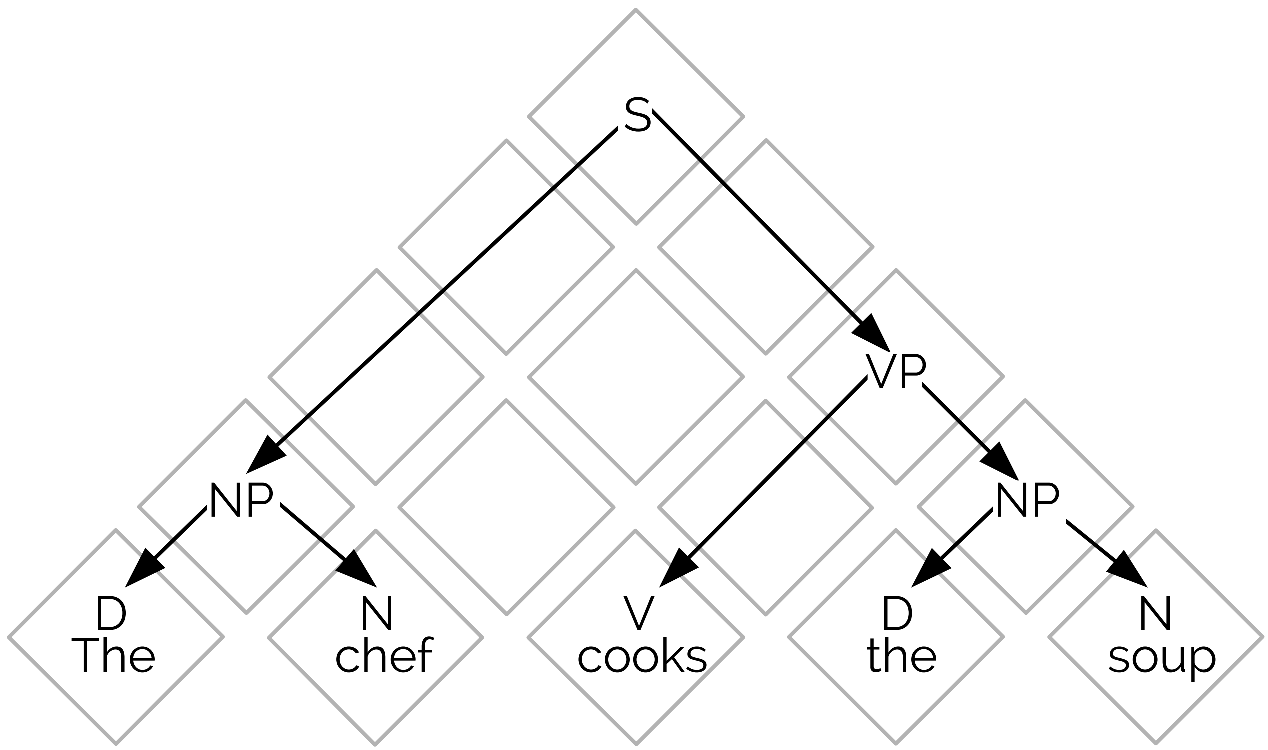
Learn "ensemble" of policies (relaxation)

Like a "gradient" → d R / d a

Move along the "gradient", within policy family ("linear min. oracle")

Keeps policy from changing "too much"

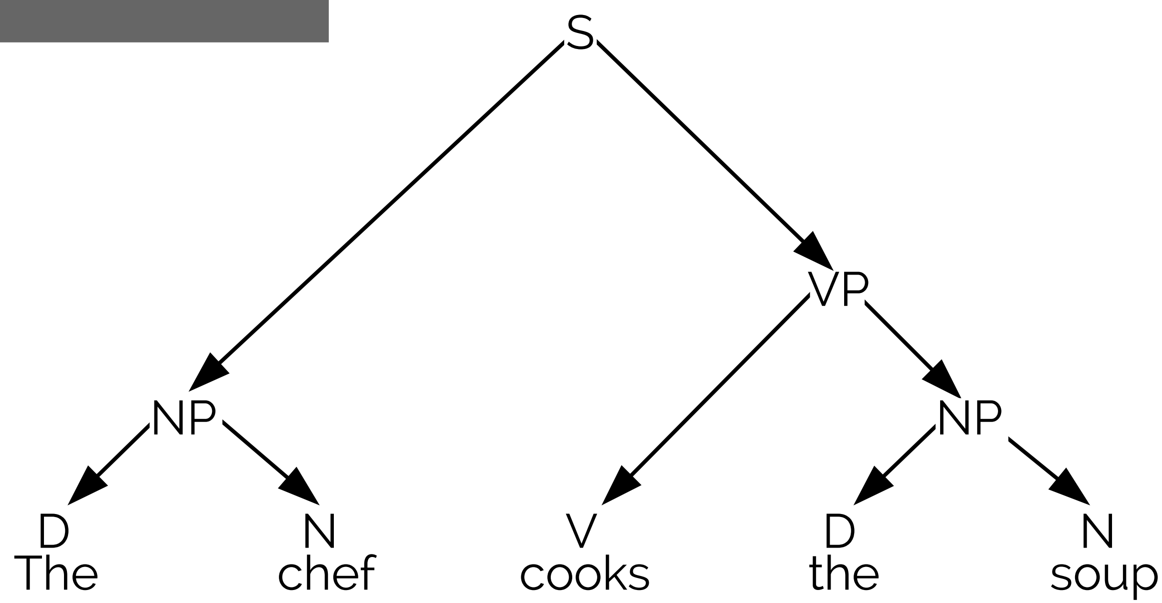
Parsing



Parsing

“Diagramming sentences”

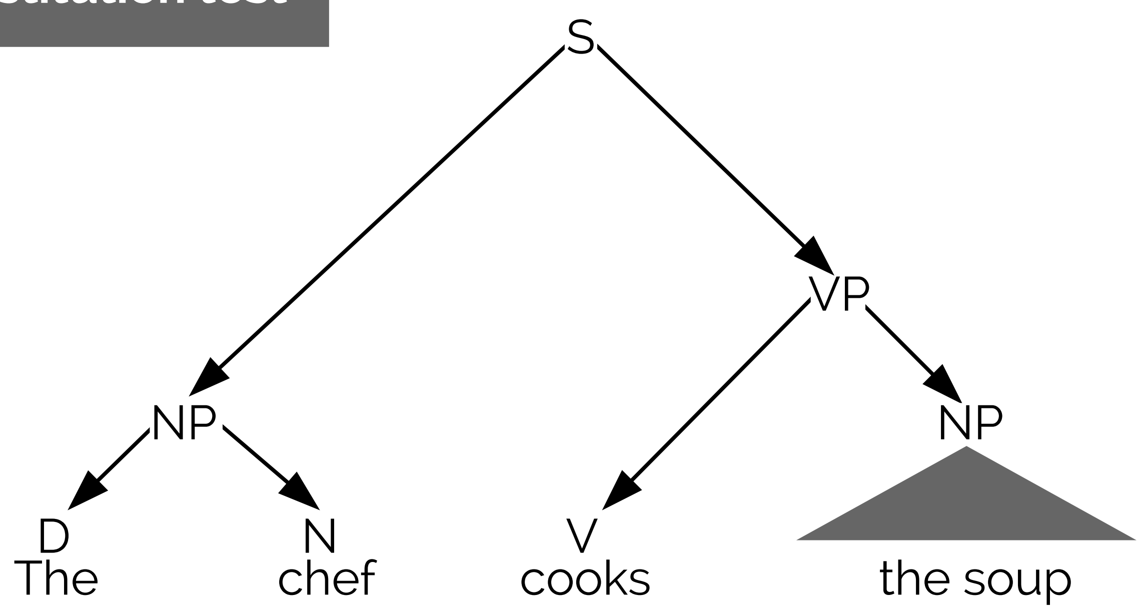
Intuition: the substitution test



Parsing

“Diagramming sentences”

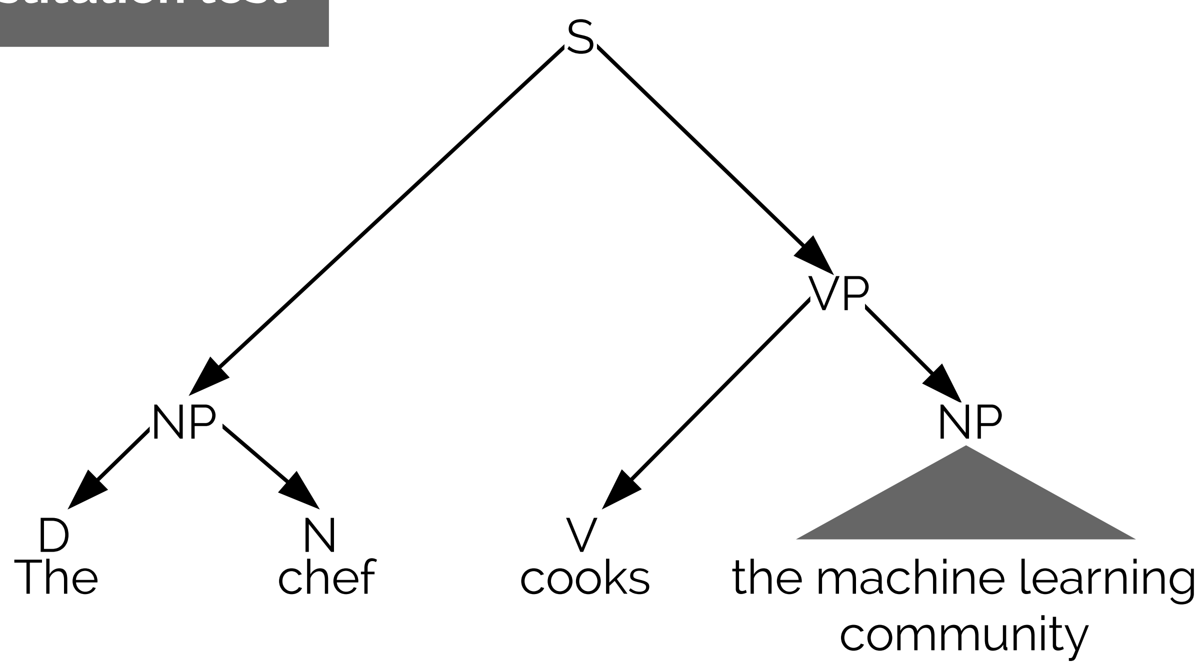
Intuition: the substitution test



Parsing

"Diagramming sentences"

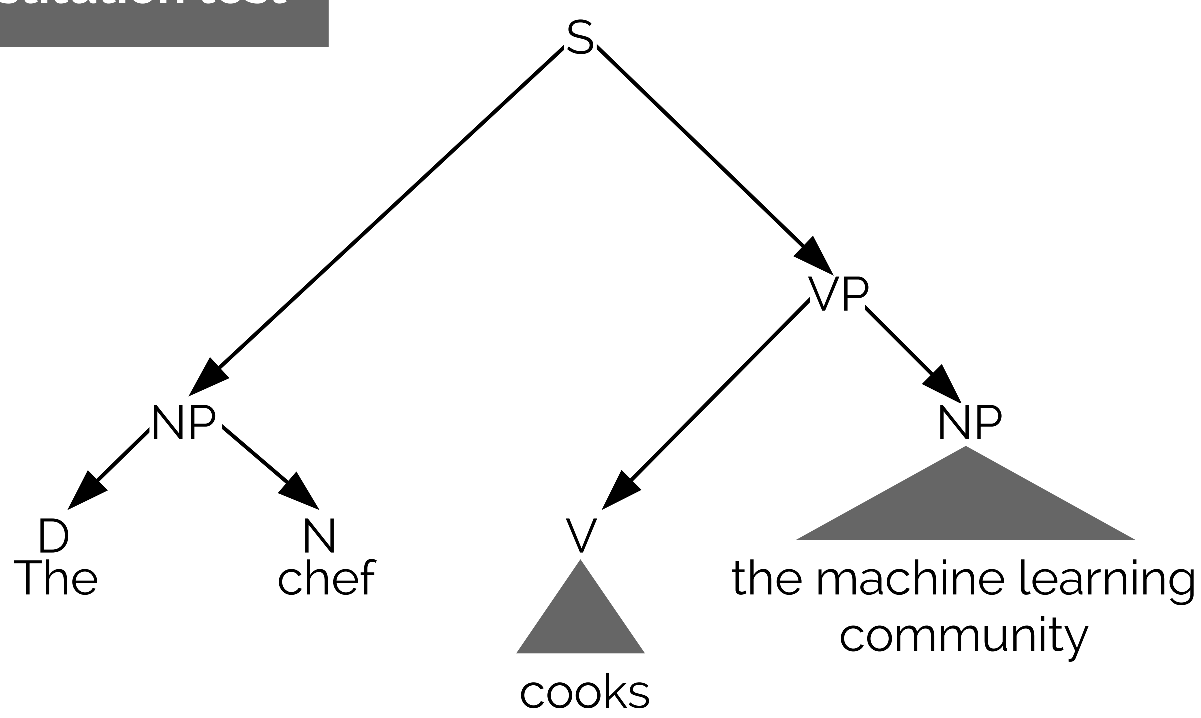
Intuition: the substitution test



Parsing

"Diagramming sentences"

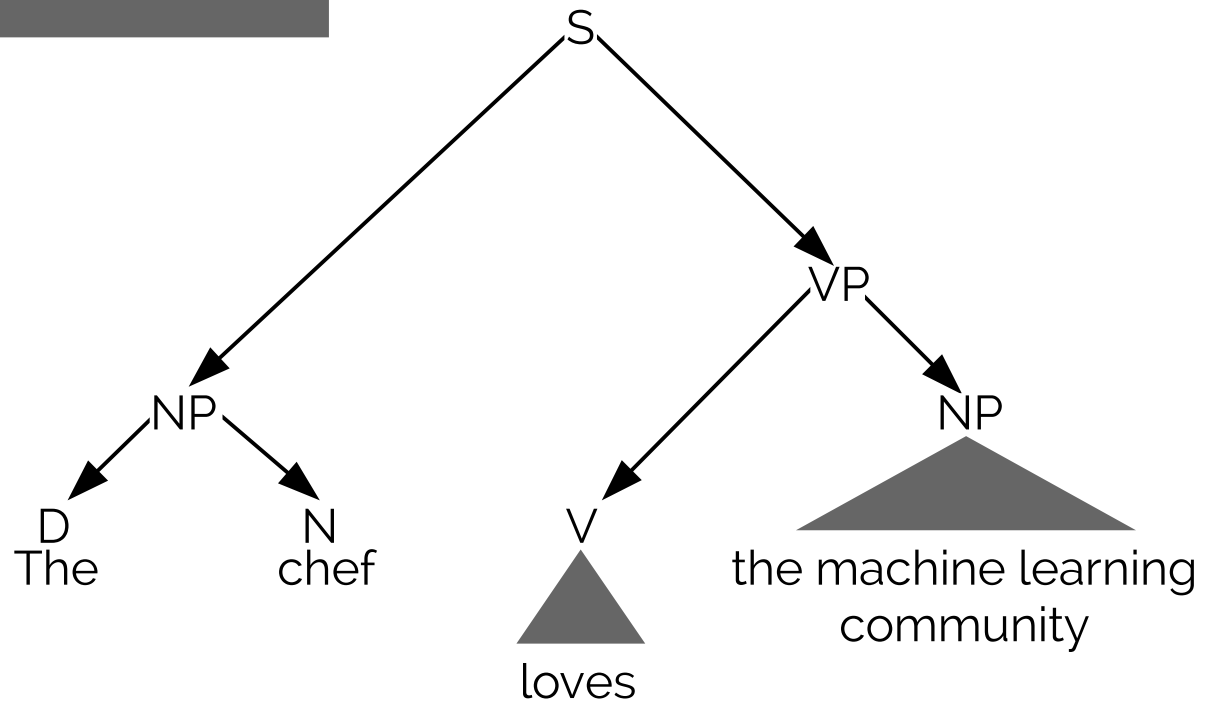
Intuition: the substitution test



Parsing

“Diagramming sentences”

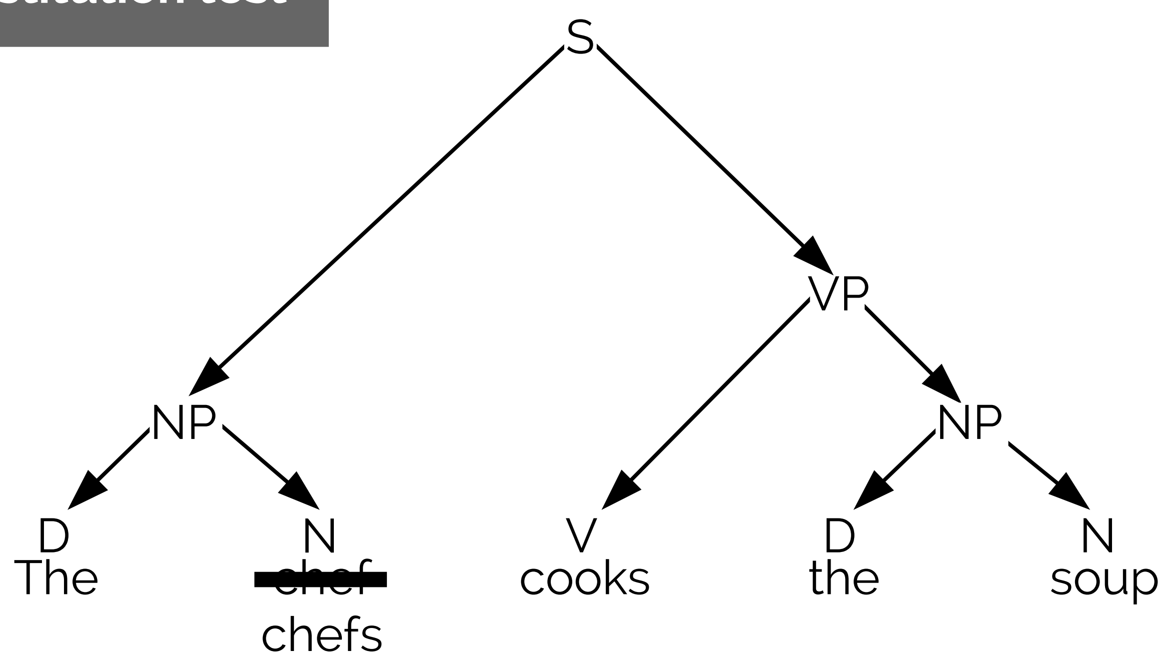
Intuition: the substitution test



Parsing

“Diagramming sentences”

Intuition: the substitution test

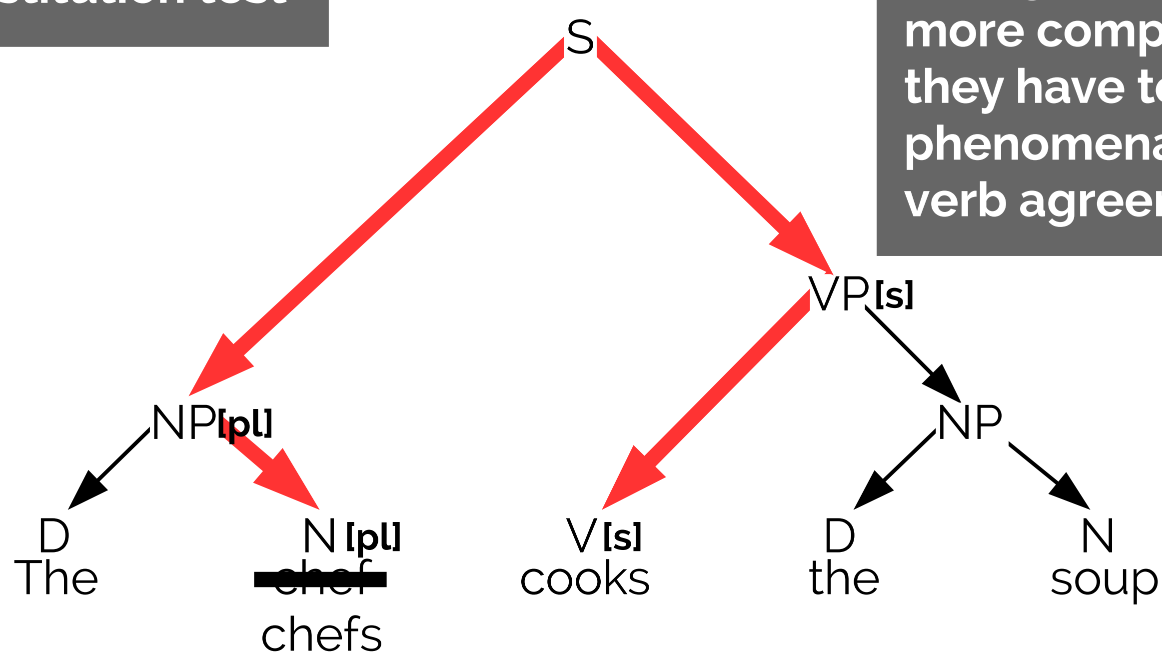


Parsing

What is a good derivation for this sentence?

Intuition: the substitution test

Real grammars are much more complex because they have to capture lots of phenomena (e.g., subject-verb agreement)



Approximate policy iteration

Intuition: It's easier to navigate the space of decision sequences (pruning masks) than parameters.

1. Guess a decision sequence

2. Given a decision sequence, we can approximately map to parameters, which produce that sequence by solving a classification problem.

3. Improve guess by looking a nearby sequences.

4. Repeat 2 and 3