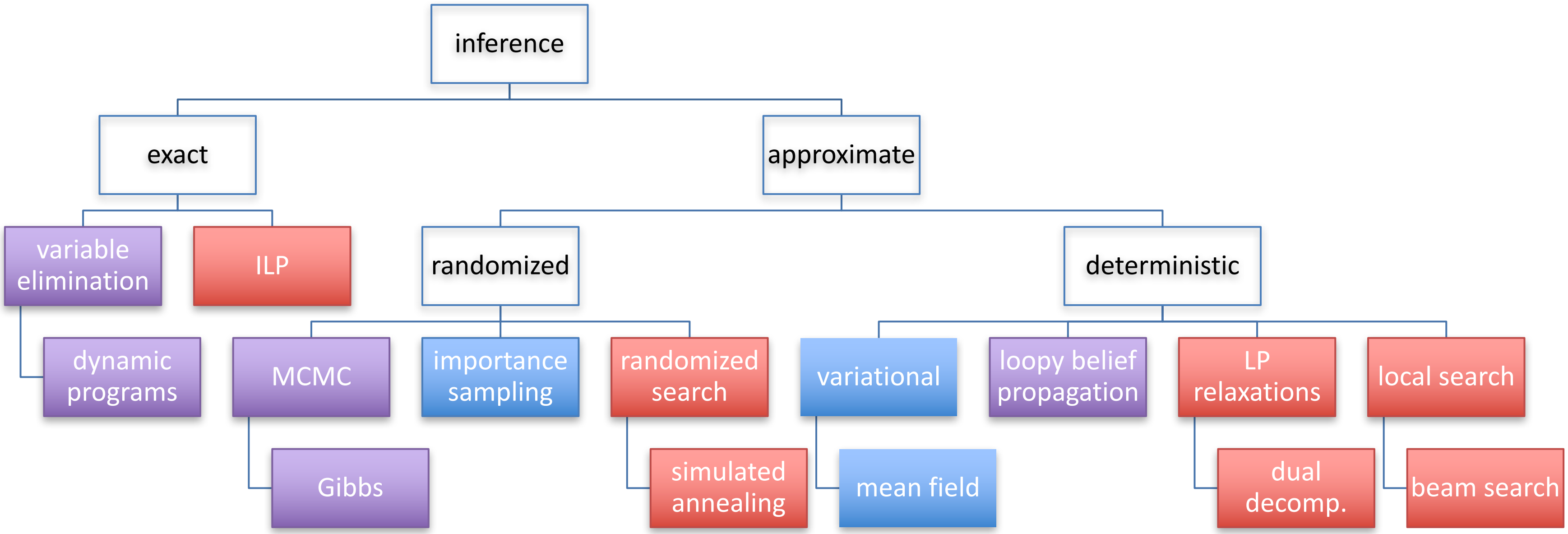
Approaches to Inference



red = hard inference
blue = soft inference
purple = both

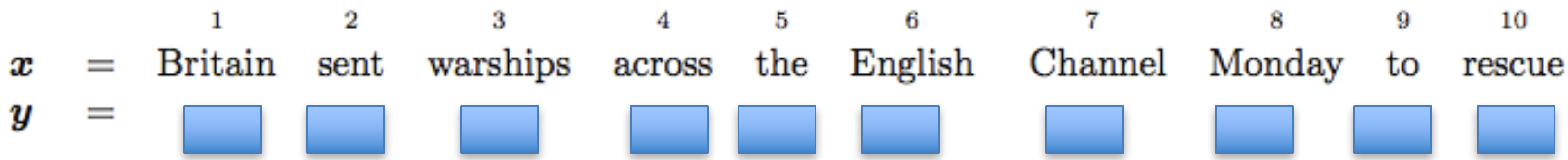
“Parts”

- Assume that feature function **g** breaks down into local parts.

$$\mathbf{g}(\boldsymbol{x}, \boldsymbol{y}) \quad = \quad \sum_{i=1}^{\#parts(\boldsymbol{x})} \mathbf{f}(\Pi_i(\boldsymbol{x}, \boldsymbol{y}))$$

- Each part has an alphabet of possible values.
  - Decoding is choosing values for all parts, with consistency constraints.
  - (In the graphical models view, a part is a clique.)

Example

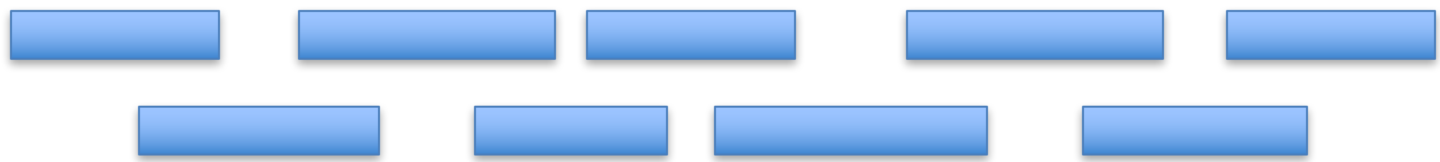


- One part per word, each is in {B, I, O}
- No features look at multiple parts
  - Fast inference
  - Not very expressive

Example

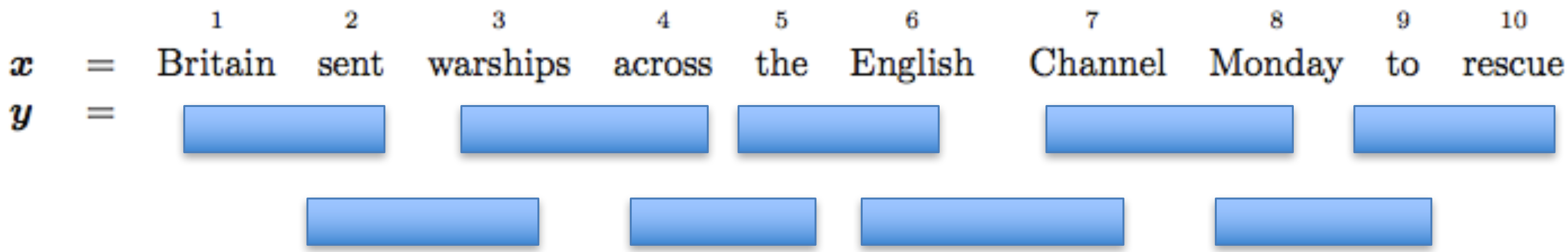
		1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$	=	Britain	sent	warships	across	the	English	Channel	Monday	to	rescue

$\boldsymbol{y}$ =



- One part per bigram, each is in {BB, BI, BO, IB, II, IO, OB, OO}
- Features and constraints can look at pairs
  - Slower inference
  - A bit more expressive

Geometric View



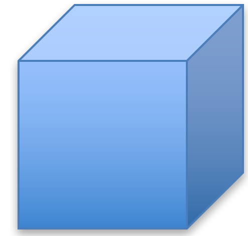
- Let $z_{i,\boldsymbol{\pi}}$ be 1 if part *i* takes value $\boldsymbol{\pi}$ and 0 otherwise.
- $\mathbf{z}$ is a vector in $\{0, 1\}^N$
  - N = total number of localized part values
  - Each $\mathbf{z}$ is a vertex of the unit cube

Score is Linear in $\mathbf{z}$

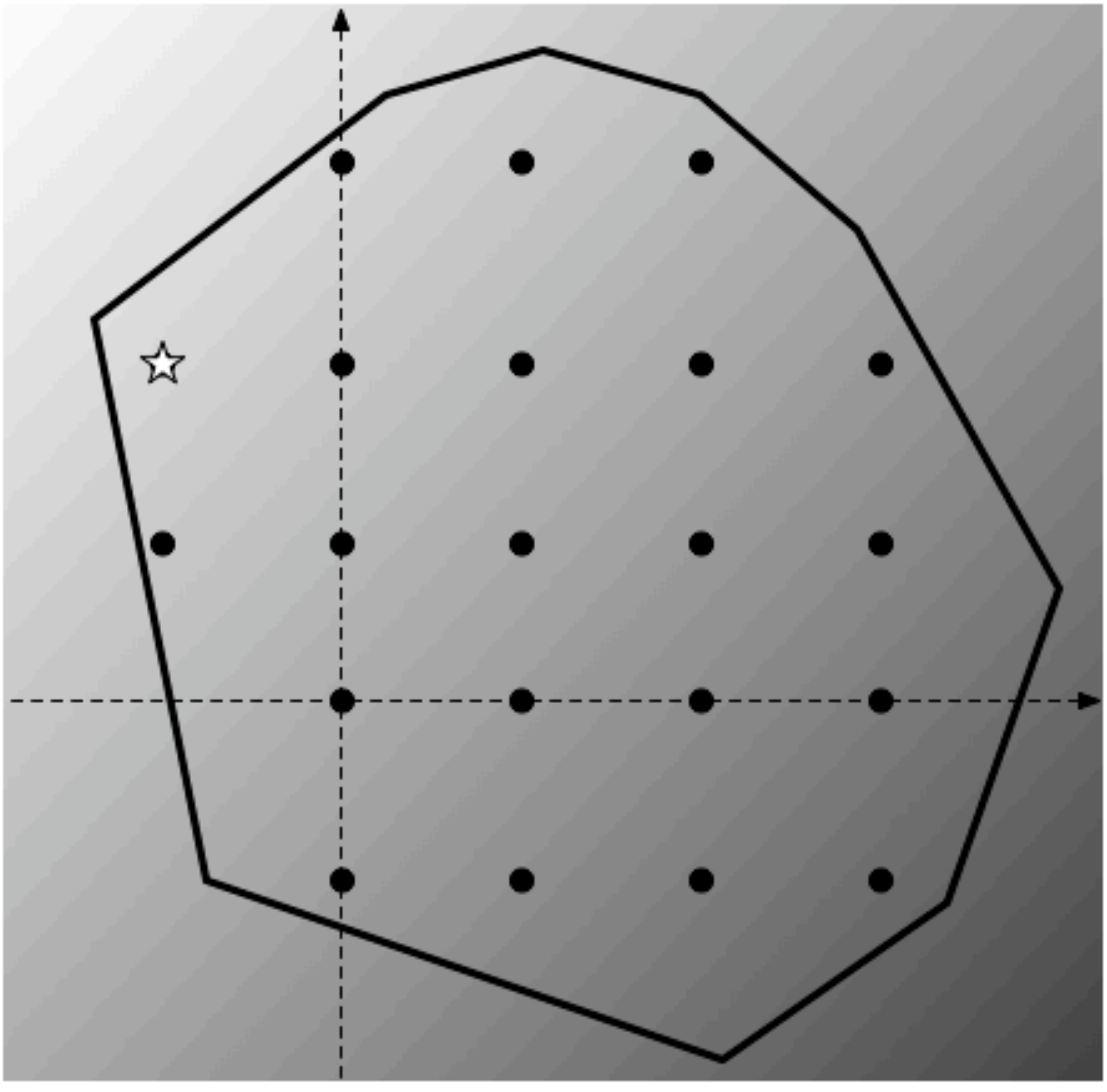
$$\begin{aligned}
\arg\max_{\boldsymbol{y}} \mathbf{w}^\top \mathbf{g}(\boldsymbol{x}, \boldsymbol{y}) &= \arg\max_{\boldsymbol{y}} \mathbf{w}^\top \sum_{i=1}^{\#parts(\boldsymbol{x})} \mathbf{f}(\Pi_i(\boldsymbol{x}, \boldsymbol{y})) \\
&= \arg\max_{\boldsymbol{y}} \mathbf{w}^\top \sum_{i=1}^{\#parts(\boldsymbol{x})} \sum_{\boldsymbol{\pi} \in \text{Values}(\Pi_i)} \mathbf{f}(\boldsymbol{\pi}) \mathbf{1}\{\Pi_i(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\pi}\} \\
&= \arg\max_{\mathbf{z} \in \mathcal{Z}_{\boldsymbol{x}}} \mathbf{w}^\top \sum_{i=1}^{\#parts(\boldsymbol{x})} \sum_{\boldsymbol{\pi} \in \text{Values}(\Pi_i)} \mathbf{f}(\boldsymbol{\pi}) z_{i,\boldsymbol{\pi}} \\
&= \arg\max_{\mathbf{z} \in \mathcal{Z}_{\boldsymbol{x}}} \mathbf{w}^\top \mathbf{F}_{\boldsymbol{x}} \mathbf{z} \\
&= \arg\max_{\mathbf{z} \in \mathcal{Z}_{\boldsymbol{x}}} \left(\mathbf{w}^\top \mathbf{F}_{\boldsymbol{x}}\right) \mathbf{z}
\end{aligned}$$

not really equal; need to transform back to get **y**

Polyhedra

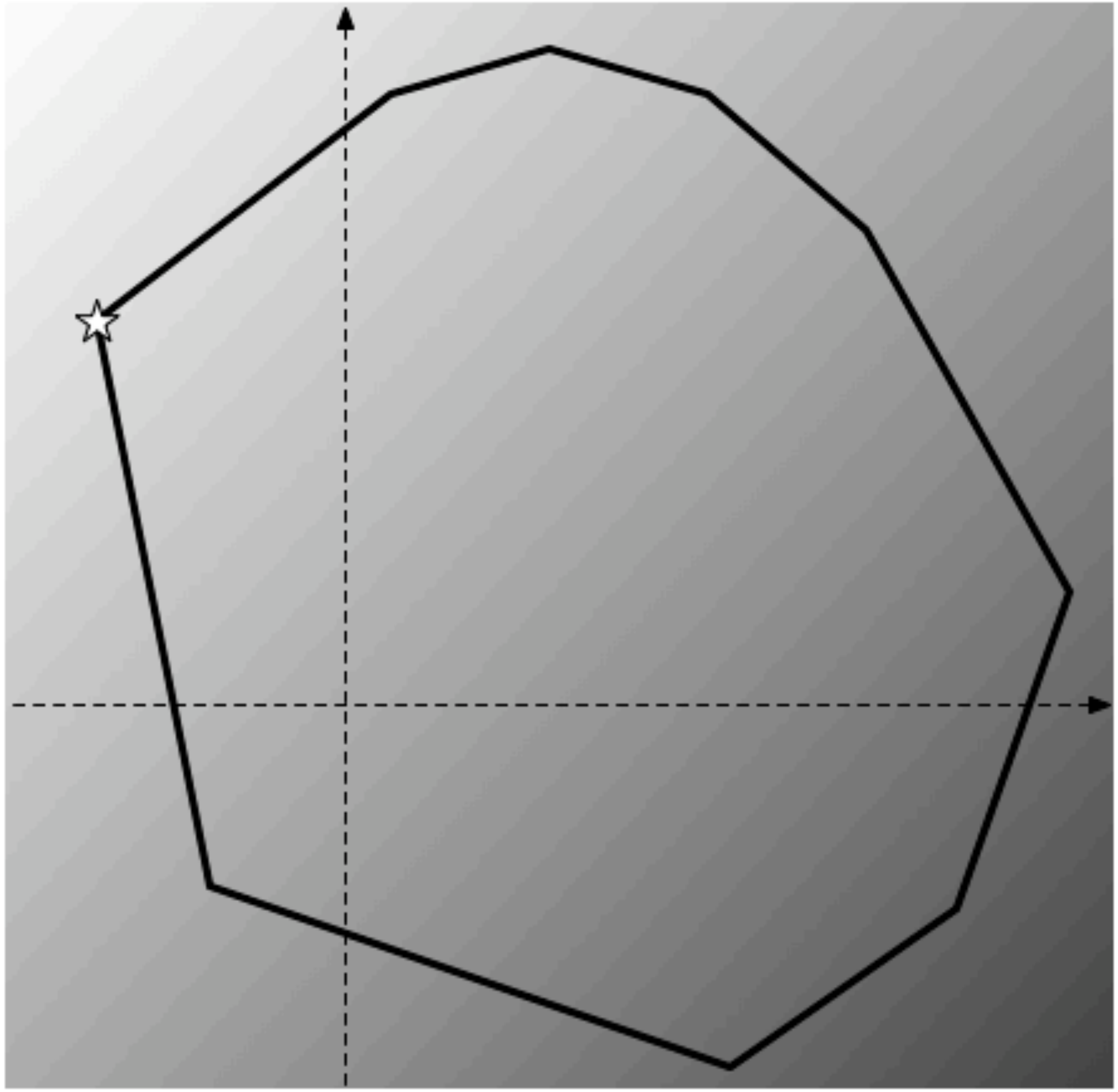


- Not all vertices of the *N*-dimensional unit cube satisfy the constraints.
  - E.g., can't have $z_{1,\mathrm{BI}} = 1$ and $z_{2,\mathrm{BI}} = 1$
- Sometimes we can write down a small (polynomial number) of linear constraints on **z**.
- Result: linear objective, linear constraints, integer constraints …



Integer Linear Programming

- Very easy to add new constraints and non-local features.
- Many decoding problems have been mapped to ILP (sequence labeling, parsing, …), but it's *not* always trivial.
- NP-hard in general.
  - But there are packages that often work well in practice (e.g., CPLEX)
  - Specialized algorithms in some cases
  - LP relaxation for approximate solutions



Remark

- Graphical models assumed a probabilistic interpretation
  - Though they are not always learned using a probabilistic interpretation!

- The polytope view is agnostic about how you interpret the weights.
  - It only says that the decoding problem is an ILP.

3. Weighted Parsing

Grammars

- Grammars are often associated with natural language parsing, but they are extremely powerful for imposing constraints.
- We can add weights to them.
  - HMMs are a kind of weighted regular grammar (closely connected to WFSAs)
  - PCFGs are a kind of weighted CFG
  - Many, many more.
- Weighted parsing: find the maximum-weighted derivation for a string **x**.

Decoding as Weighted Parsing

- Every valid **y** is a grammatical derivation (parse) for **x**.
  - HMM: sequence of "grammatical" states is one allowed by the transition table.
- Augment parsing algorithms with weights and find the best parse.

The Viterbi algorithm is an instance of recognition by a weighted grammar!

BIO Tagging as a CFG

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl@{\qquad}lcl}
N_{\Rightarrow} & \rightarrow & B\ R_B & R_B & \rightarrow & B\ R_B & R_I & \rightarrow & B\ R_B & R_O & \rightarrow & B\ R_B \\
N_{\Rightarrow} & \rightarrow & O\ R_O & R_B & \rightarrow & O\ R_O & R_I & \rightarrow & O\ R_O & R_O & \rightarrow & O\ R_O \\
 & & & R_B & \rightarrow & I\ R_I & R_I & \rightarrow & I\ R_I & & & \\
 & & & R_B & \rightarrow & \epsilon & R_I & \rightarrow & \epsilon & R_O & \rightarrow & \epsilon
\end{array}
$$

$$
\forall x \in \Sigma, \qquad B \rightarrow x \qquad I \rightarrow x \qquad O \rightarrow x
$$

- Weighted (or probabilistic) CKY is a dynamic programming algorithm very similar in structure to classical CKY.

4. Paths and Hyperpaths

Best Path

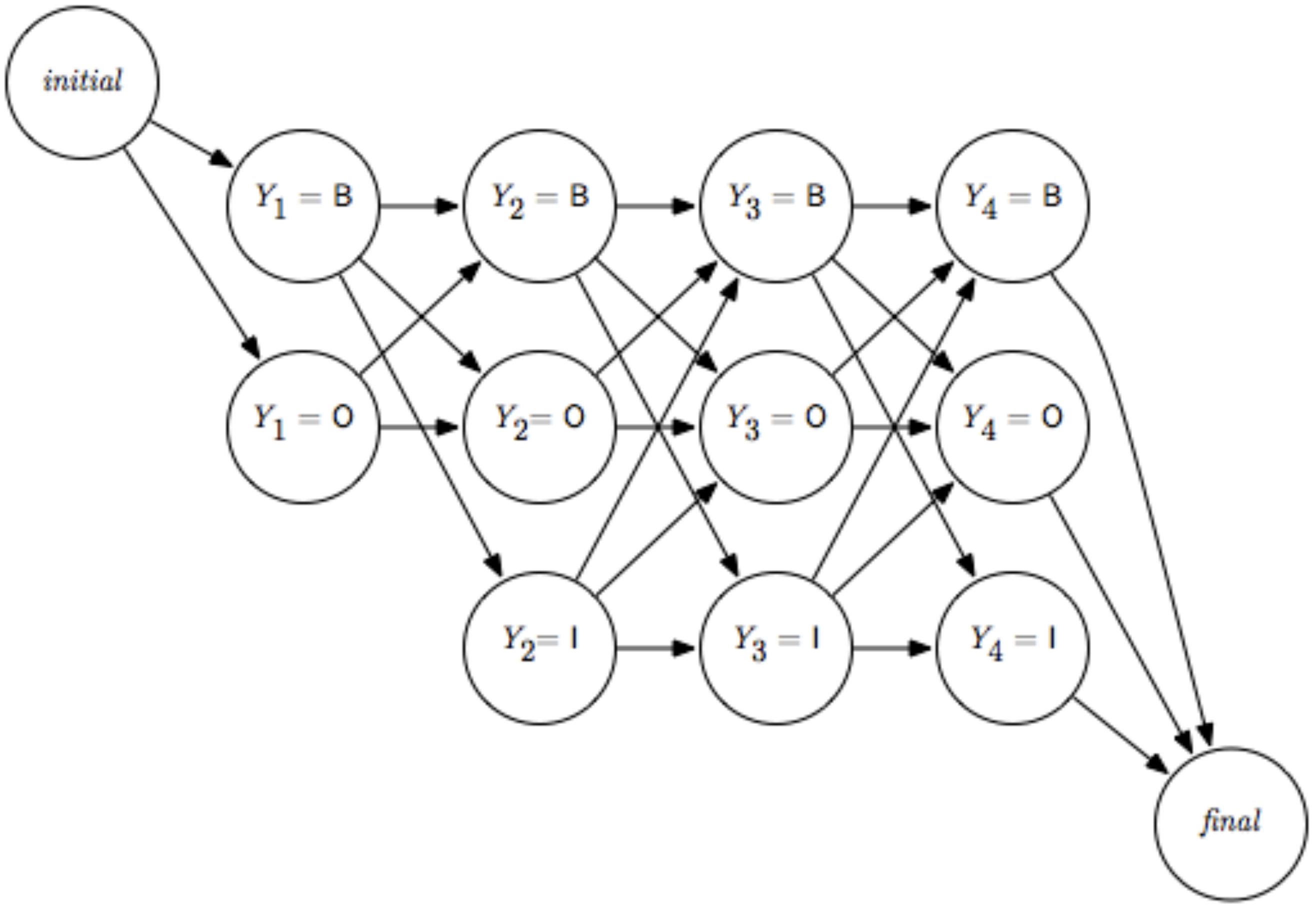
- General idea: take ***x*** and build a graph.
- Score of a path factors into the edges.

$$\arg\max_{\boldsymbol{y}} \mathbf{w}^\top \mathbf{g}(\boldsymbol{x}, \boldsymbol{y}) \quad = \quad \arg\max_{\boldsymbol{y}} \mathbf{w}^\top \sum_{e \in \text{Edges}} \mathbf{f}(e) \mathbf{1}\{e \text{ is crossed by } \boldsymbol{y}\text{'s path}\}$$

- Decoding is finding the *best* path.

The Viterbi algorithm is an instance of finding a best path!

“Lattice” View of Viterbi



A Generic Best Path Algorithm

- Input: directed graph $G = (V, E)$, cost : $E \rightarrow \mathbb{R}$, start vertex v_0
- Output: $d : V \rightarrow \mathbb{R}$ (shortest path function) and back pointers $b : V \rightarrow V$

for all $v \in V \setminus \{v_0\}$, $d(v) := \infty$ and $b(v) := \varnothing$

set $d(v_0) = 0$

while d has not converged:

    pick an arbitrary edge (u, v)

    if d(u) + cost(u, v) < d(v):

        d(v) := d(u) + cost(u, v)

        b(v) := u

Ordering Updates

- Naïve ways of choosing edges will lead to cyclic updating and gross inefficiency!
- Before considering various ways of doing it, let's consider how the Viterbi algorithm is essentially solving the same problem.

Viterbi Algorithm (In the Style of A Best Path Algorithm)

- Input:
  - directed graph $G = (V, E)$ where
    each vertex $v = (q, t)$, $q \in Q \cup \{\varnothing\}$, $t \in \{-1, 0, 1, \ldots, n\}$
    and each edge $(u, v) = ((q, t), (q', t + 1))$
  - cost : $E \to \mathbb{R}$, defined by
    $\text{cost}((q, t), (q', t + 1)) = -\log \gamma(q' \mid q) - \log \eta(s_{t+1} \mid q) - \log (1 - \xi(q))$
    $\text{cost}((q, n - 1), (q', n)) = -\log \gamma(q' \mid q) - \log \eta(s_{t+1} \mid q) - \log \xi(q')$
    $\text{cost}((\varnothing, -1), (q, 0)) = -\log \pi(q)$
  - fixed start vertex $v_0 = (\varnothing, -1)$
- Output: $d : V \to \mathbb{R}$ (shortest path function) and back pointers $b : V \to V$

for all $v \in V \setminus \{v_0\}$, $d(v) := \infty$ and $b(v) := \varnothing$

set $d(v_0) = 0$

perform a topological sort on V

~~while d has not converged:~~ for each v in top-sort order:

    ~~pick an arbitrary edge (u, v)~~

    for each $(u, v) \in E$:

      if $d(u) + \text{cost}(u, v) < d(v)$:

          $d(v) := d(u) + \text{cost}(u, v)$

          $b(v) := u$

    // d(v) and b(v) are now known

The Viterbi Trick

- From a “best path” perspective, Viterbi is:
  - defining the vertices and edges to have special structure (state/time step)
  - assigning costs based on HMM weights and the specific input string $s_1 \ldots s_n$
  - ordering the edges cleverly to make things efficient
- Note also: Viterbi's graph has no cycles.

Another Variant: "Forward" Updating

- After topological sort, can also choose all edges going *out of* current node.

for all $v \in V \setminus \{v_0\}$, $d(v) := \infty$ and $b(v) := \varnothing$

set $d(v_0) = 0$

perform a topological sort on V

for each u in top-sort order:

    for each $(u, v) \in E$:

      if $d(u) + cost(u, v) < d(v)$:

        $d(v) := d(u) + cost(u, v)$

        $b(v) := u$

Memoized Recursion

- Input: directed graph $G = (V, E)$, cost : $E \rightarrow \mathbb{R}$, start vertex v_0, target vertex v_t
- Output: $d : V \rightarrow \mathbb{R}$ (shortest path function) and back pointers $b : V \rightarrow V$

for all $v \in V \setminus \{v_0\}$, $d(v) := \varnothing$ and $b(v) := \varnothing$
set $d(v_0) = 0$
memoize(v_t)

memoize(v):
    // guaranteed to return best-cost path score for v
    if $d(v) = \varnothing$:
      $d(v) := \infty$
      for each $(u, v) \in E$:
            if memoize(u) + cost(u, v) < d(v):
                  d(v) := d(u) + cost(u, v)
                  b(v) := u
    return d(v)

A Generic Best Path Algorithm

- Input: directed graph $G = (V, E)$, $cost : E \rightarrow \mathbb{R}$, start vertex v_0
- Output: $d : V \rightarrow \mathbb{R}$ (shortest path function) and back pointers $b : V \rightarrow V$

for all $v \in V \setminus \{v_0\}$, $d(v) := \infty$ and $b(v) := \varnothing$
set $d(v_0) = 0$
while d has not converged:
    pick an arbitrary edge (u, v)
    if $d(u) + cost(u, v) < d(v)$:
        $d(v) := d(u) + cost(u, v)$
        $b(v) := u$

Dijkstra's Algorithm

- Input: directed graph $G = (V, E)$, $cost : E \rightarrow \mathbb{R}_{\geq 0}$ (important!), start vertex v_0
- Output: $d : V \rightarrow \mathbb{R}$ (shortest path function) and back pointers $b : V \rightarrow V$

```
for all v ∈ V \ {v0}, d(v) := ∞ and b(v) := ∅
set d(v0) = 0
Q := priority queue on V ordered by d (lower cost = higher priority)
~~while d has not converged:~~  while Q is not empty:
    ~~pick an arbitrary edge (u, v)~~
    u := extract-min(Q)
    for each (u, v) ∈ E:
      if d(u) + cost(u, v) < d(v):
            d(v) := d(u) + cost(u, v)
            b(v) := u
            update v's priority in Q
```

A* Algorithm

- Input: directed graph G = (V, E), cost : E → $\mathbb{R}_{\geq 0}$, start vertex v_0, target vertex v_t, heuristic h : V → $\mathbb{R}_{\geq 0}$ such that h(v) ≤ best-cost(v, v_t)
- Output: d : V → $\mathbb{R}$ (shortest path function) and back pointers b : V → V

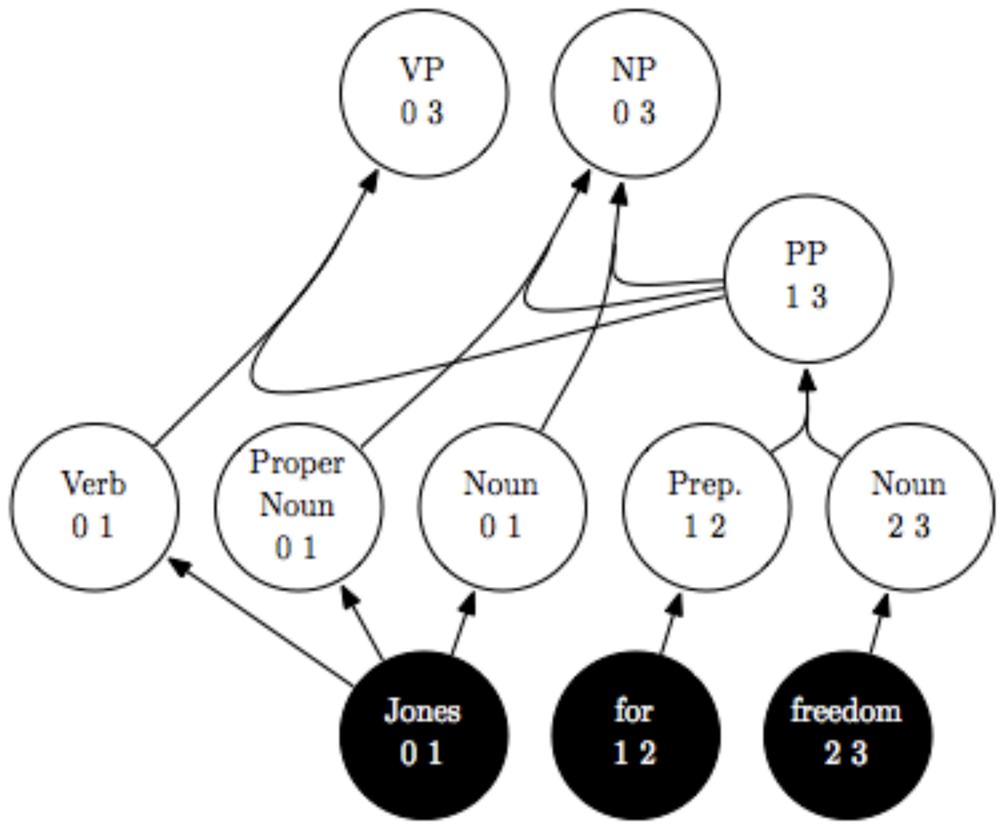
```
for all v ∈ V \ {v0}, d(v) := ∞ and b(v) := ∅
set d(v0) = 0
Q := priority queue on V ordered by d + h (lower cost = higher priority)
while Q is not empty:
    u := extract-min(Q)
    for each (u, v) ∈ E:
      if d(u) + cost(u, v) < d(v):
          d(v) := d(u) + cost(u, v)
          b(v) := u
          update v's priority in Q
```

Minimum Cost Hyperpath

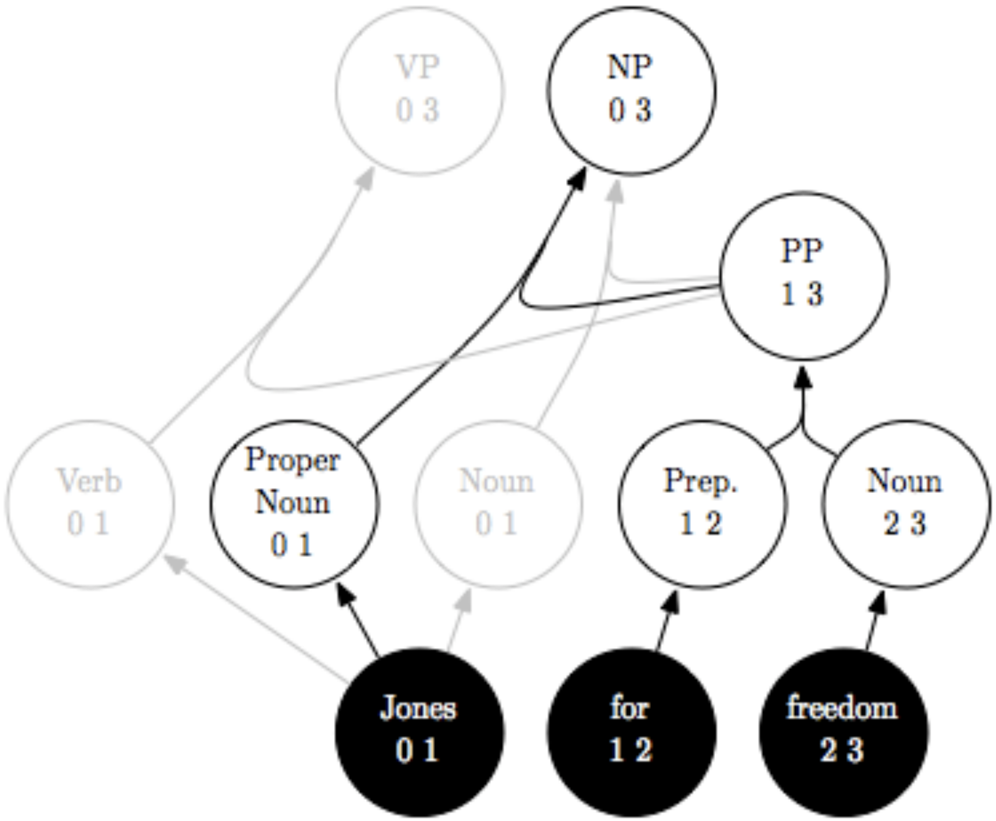
- General idea: take ***x*** and build a hypergraph.
- Score of a hyperpath factors into the hyperedges.
- Decoding is finding the best *hyperpath*.

- This connection was elucidated by Klein and Manning (2002).

Parsing as a Hypergraph

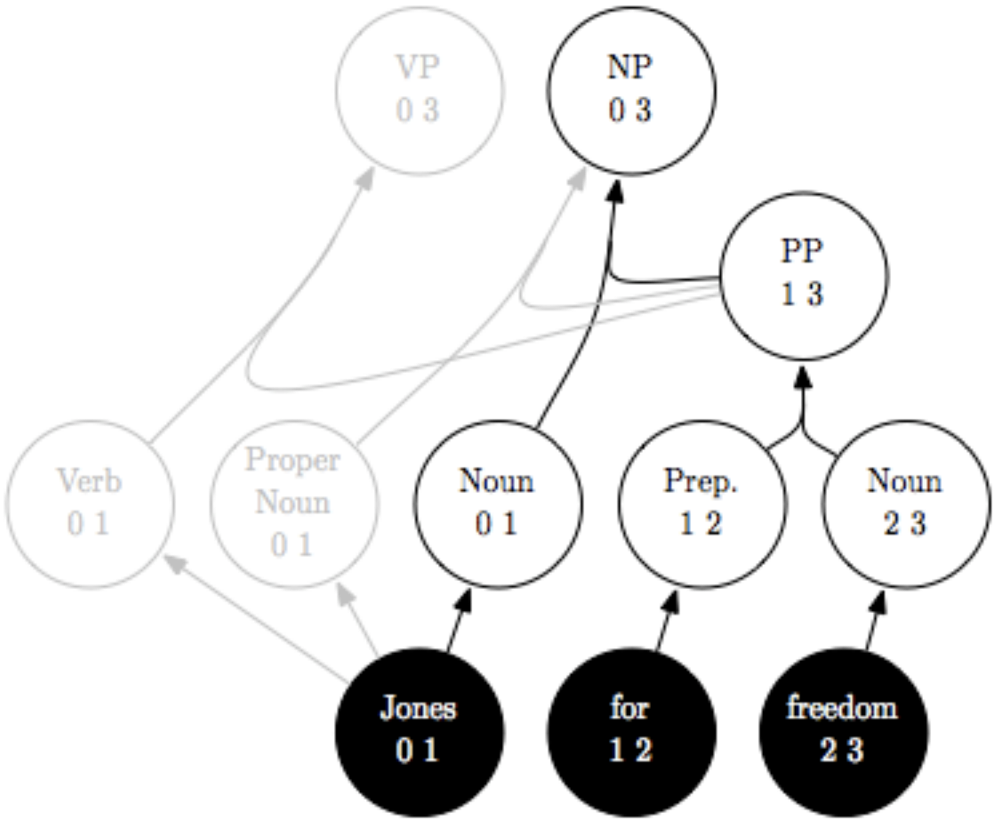


Parsing as a Hypergraph



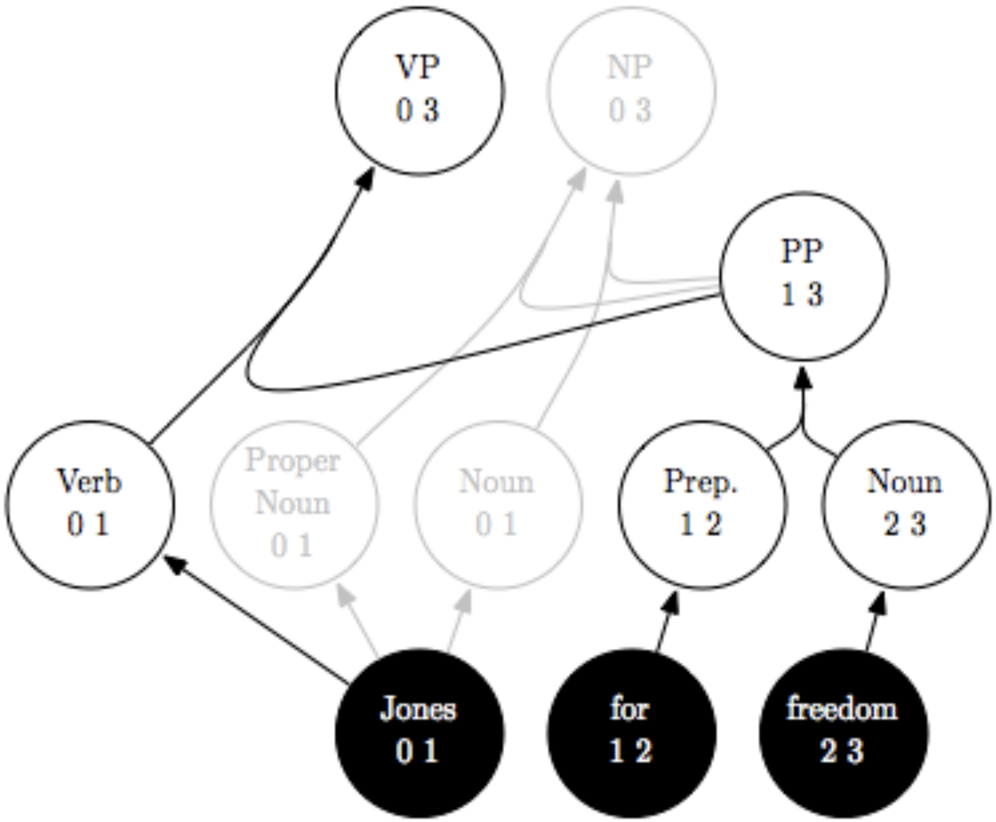
cf. "Dean for democracy"

Parsing as a Hypergraph



Forced to work on his thesis, sunshine streaming in the window, Mike experienced a …

Parsing as a Hypergraph



Forced to work on his thesis, sunshine streaming in the window, Mike began to …

Why Hypergraphs?

- Useful, compact encoding of the hypothesis space.
  - Build hypothesis space using local features, maybe do some filtering.
  - Pass it off to another module for more fine-grained scoring with richer or more expensive features.

5. Weighted Logic Programming

Logic Programming

- Start with a set of axioms and a set of inference rules.

$$\forall A, C, \qquad \text{ancestor}(A, C) \Leftarrow \text{parent}(A, C)$$
$$\forall A, C, \qquad \text{ancestor}(A, C) \Leftarrow \bigvee_B \text{ancestor}(A, B) \wedge \text{parent}(B, C)$$

- The goal is to prove a specific theorem, goal.
- Many approaches, but we assume a *deductive* approach.
  - Start with axioms, iteratively produce more theorems.

$$\begin{array}{rc}
& \textsf{label-bigram}(\text{“B”}, \text{“B”}) \\
& \textsf{label-bigram}(\text{“B”}, \text{“I”}) \\
& \textsf{label-bigram}(\text{“B”}, \text{“O”}) \\
& \textsf{label-bigram}(\text{“I”}, \text{“B”}) \\
& \textsf{label-bigram}(\text{“I”}, \text{“I”}) \\
& \textsf{label-bigram}(\text{“I”}, \text{“O”}) \\
& \textsf{label-bigram}(\text{“O”}, \text{“B”}) \\
& \textsf{label-bigram}(\text{“O”}, \text{“O”}) \\
\forall x \in \Sigma, & \textsf{labeled-word}(x, \text{“B”}) \\
\forall x \in \Sigma, & \textsf{labeled-word}(x, \text{“I”}) \\
\forall x \in \Sigma, & \textsf{labeled-word}(x, \text{“O”})
\end{array}$$

$$\begin{array}{rcl}
\forall \ell \in \Lambda, \ \ \textsf{v}(\ell, 1) & = & \textsf{labeled-word}(x_1, \ell) \\
\forall \ell \in \Lambda, \ \ \textsf{v}(\ell, i) & = & \displaystyle\bigvee_{\ell' \in \Lambda} \textsf{v}(\ell', i-1) \wedge \textsf{label-bigram}(\ell', \ell) \wedge \textsf{labeled-word}(x_i, \ell) \\
\textsf{goal} & = & \displaystyle\bigvee_{\ell \in \Lambda} \textsf{v}(\ell, n)
\end{array}$$

Weighted Logic Programming

- Twist: axioms have weights.
- Want the proof of goal with the best score:

$$\arg\max_{\boldsymbol{y}} \mathbf{w}^\top \mathbf{g}(\boldsymbol{x}, \boldsymbol{y}) = \arg\max_{\boldsymbol{y}} \mathbf{w}^\top \sum_{a \in \text{Axioms}} \mathbf{f}(a) \mathit{freq}(a; \boldsymbol{y})$$

- Note that axioms can be used more than once in a proof (**y**).

Whence WLP?

- Shieber, Schabes, and Pereira (1995): many parsing algorithms can be understood in the same deductive logic framework.
- Goodman (1999): add weights in a semiring, get many useful NLP algorithms.
- Eisner, Goldlust, and Smith (2004, 2005): semiring-generic algorithms, Dyna.

Dynamic Programming

- Most views (exception is polytopes) can be understood as DP algorithms.
  - The low-level *procedures* we use are often DP.
  - Even DP is too high-level to know the best way to implement.
- Break a problem into slightly smaller problems with **optimal substructure**.
  - Best path to v depends on best paths to all u such that $(u,v) \in E$.
- **Overlapping subproblems**: each subproblem gets used repeatedly, and there aren't too many of them.

Dynamic Programming

- Three main strategies for DP:
  - Viterbi, Levenshtein edit distance, CKY: predefined, “clever” ordering.
  - Memoization
  - Agenda (Dijkstra’s algorithm, A*)
- Things to remember in general:
  - The hypergraph may too big to represent explicitly; exhaustive calculation may be too expensive.
  - The hypergraph may or may not have properties that make “clever” orderings possible.
  - DP does *not* imply polynomial time and space! Most common approximations when the desired state space is too big: beam search, cube pruning, agendas with early stopping, ...

Summary

- Decoding is the general problem of choosing a complex structure.
  - Linguistic analysis, machine translation, speech recognition, …
  - Statistical models are usually involved (not necessarily probabilistic).
- No perfect general view, but much can be gained through a combination of views.
- First question: can I solve it exactly with DP?