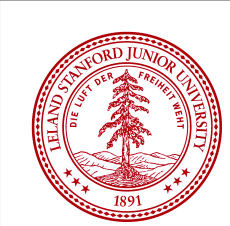
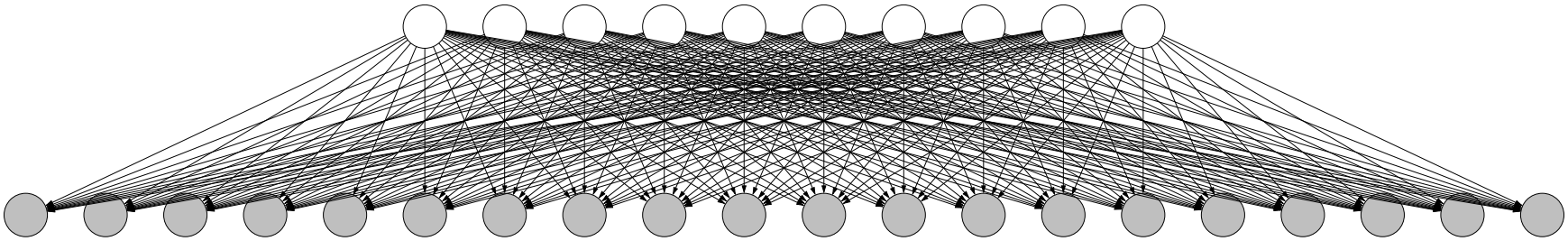
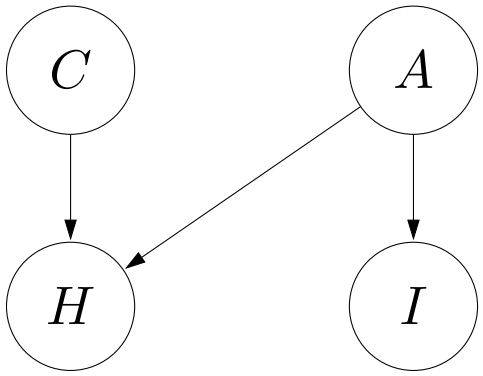


Lecture 14: Bayesian networks II



Review: Bayesian network



$$\mathbb{P}(C = c, A = a, H = h, I = i)$$
$$\stackrel{\text{def}}{=} p(c)p(a)p(h \mid c, a)p(i \mid a)$$

Definition: Bayesian network

Let $X = (X_1, \ldots, X_n)$ be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a joint distribution over X as a product of local conditional distributions, one for each node:

$$\mathbb{P}(X_1 = x_1, \ldots, X_n = x_n) = \prod_{i=1}^{n} p(x_i \mid x_{\text{Parents}(i)})$$

- Last time, we talked about Bayesian networks, which was a fun and convenient modeling framework. We posit a collection of variables that describe the state of the world, and then create a story on how the variables are generated (recall the probabilistic program interpretation).
- A Bayesian network specifies two parts: (i) a graph structure which governs the qualitative relationship between the variables, and (ii) local conditional distributions, which specify the quantitative relationship.
- Formally, a Bayesian network defines a **joint** probability distribution over many variables (e.g., $\mathbb{P}(C, A, H, I)$) via the **local** conditional distributions (e.g., $p(i \mid a)$). This joint distribution specifies all the information we know about how the world works.

Review: probabilistic inference

Input

Bayesian network: $\mathbb{P}(X_1 = x_1, \ldots, X_n = x_n)$

Evidence: $E = e$ where $E \subseteq X$ is subset of variables

Query: $Q \subseteq X$ is subset of variables

Output

$\mathbb{P}(Q = q \mid E = e)$ for all values q

Example: if coughing but no itchy eyes, have a cold?

$$\mathbb{P}(C \mid H = 1, I = 0)$$

- Think of the joint probability distribution defined by the Bayesian network as a guru. Probabilistic inference allows you to ask the guru anything: what is the probability of having a cold? What if I'm coughing? What if I don't have itchy eyes? In this lecture, we're going to build such a guru that can answer these queries efficiently.

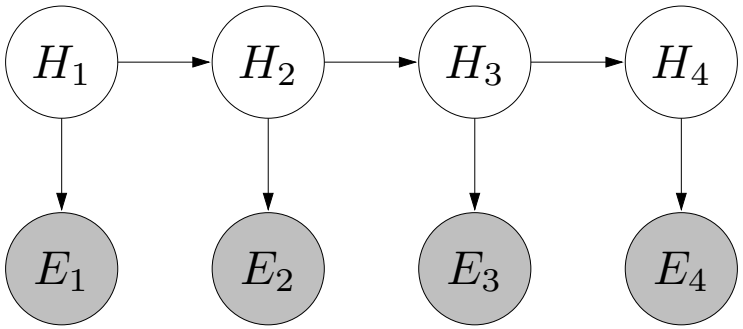
Roadmap

Forward-backward

Gibbs sampling

Particle filtering

Object tracking



Problem: object tracking

$H_i \in \{1, \ldots, K\}$: location of object at time step i

$E_i \in \{1, \ldots, K\}$: sensor reading at time step i

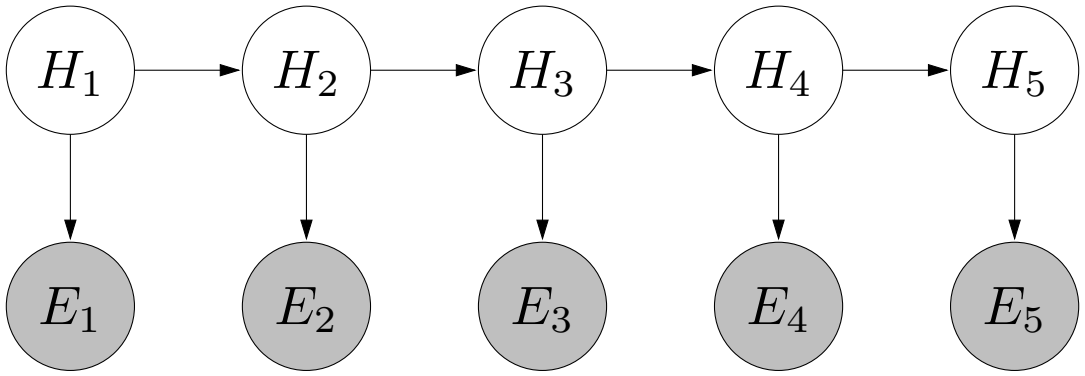
Start: $p(h_1)$: uniform over all locations

Transition $p(h_i \mid h_{i-1})$: uniform over adjacent loc.

Emission $p(e_i \mid h_i)$: uniform over adjacent loc.

- So far, we have computed various ad-hoc probabilistic queries on ad-hoc Bayesian networks by hand. We will now switch gears and focus on the popular hidden Markov model (HMM), and show how the forward-backward algorithm can be used to compute typical queries of interest.
- As motivation, consider the problem of tracking an object. The probabilistic story is as follows: An object starts at H_1 uniformly drawn over all possible locations. Then at each time step thereafter, it **transitions** to an adjacent location with equal probability. For example, if $H_2 = 3$, then $H_3 \in \{2, 4\}$ with equal probability. At each time step, we obtain a sensor reading E_i which is also uniform over locations adjacent to H_i.

Hidden Markov model



$$\mathbb{P}(H = h, E = e) = \underbrace{p(h_1)}_{\text{start}} \prod_{i=2}^{n} \underbrace{p(h_i \mid h_{i-1})}_{\text{transition}} \prod_{i=1}^{n} \underbrace{p(e_i \mid h_i)}_{\text{emission}}$$

Query (**filtering**):

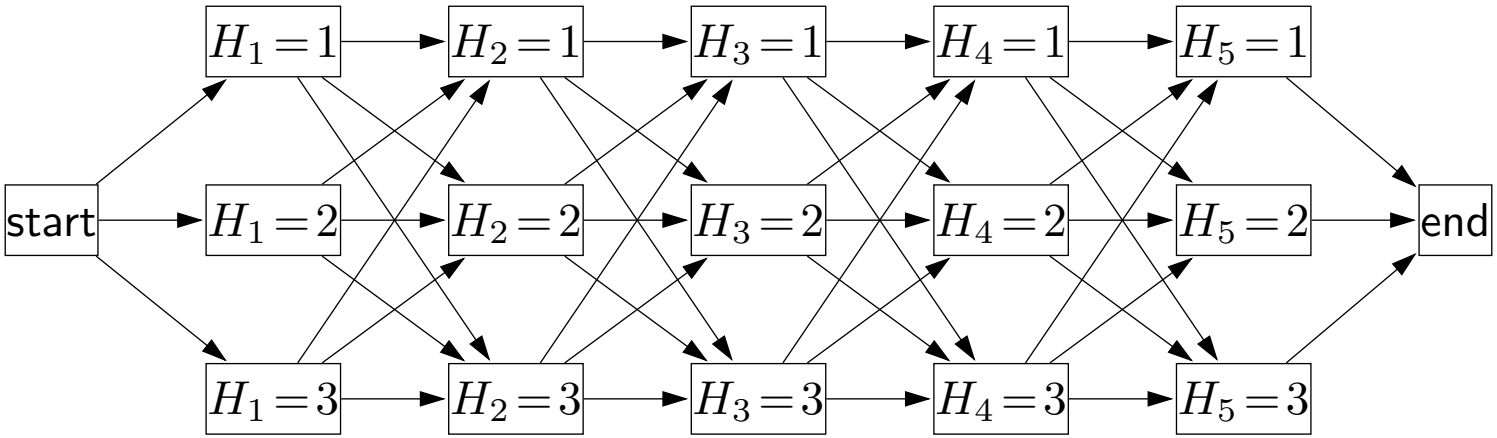
$$\mathbb{P}(H_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3)$$

Query (**smoothing**):

$$\mathbb{P}(H_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3, E_4 = e_4, E_5 = e_5)$$

- In principle, you could ask any type of query on an HMM, but there are two common ones: filtering and smoothing.
- Filtering asks for the distribution of some hidden variable H_i conditioned on only the evidence up until that point. This is useful when you're doing real-time object tracking, and you can't see the future.
- Smoothing asks for the distribution of some hidden variable H_i conditioned on all the evidence, including the future. This is useful when you have collected all the data and want to retroactively go and figure out what H_i was.

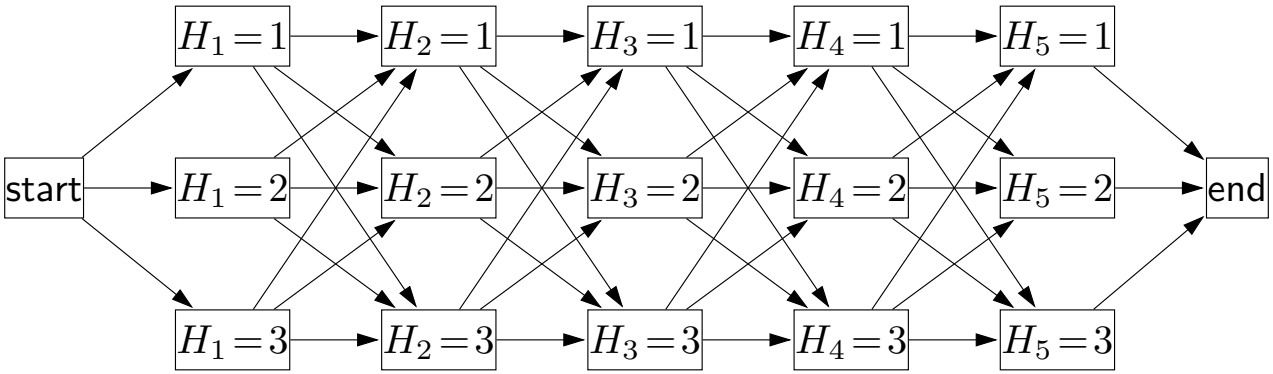
Lattice representation



- Edge $\boxed{\text{start}} \Rightarrow \boxed{H_1 = h_1}$ has weight $p(h_1)p(e_1 \mid h_1)$
- Edge $\boxed{H_{i-1} = h_{i-1}} \Rightarrow \boxed{H_i = h_i}$ has weight $p(h_i \mid h_{i-1})p(e_i \mid h_i)$
- Each path from $\boxed{\text{start}}$ to $\boxed{\text{end}}$ is an assignment with weight equal to the product of node/edge weights

- Now let's actually compute these queries. We will do smoothing first. Filtering is a special case: if we're asking for H_i given $E_1, \ldots, E_i$, then we can marginalize out the future, reducing the problem to a smaller HMM.
- A useful way to think about inference is returning to state-based models. Consider a graph with a start node, an end node, and a node for each assignment of a value to a variable $H_i = v$. The nodes are arranged in a lattice, where each column corresponds to one variable H_i and each row corresponds to a particular value v. Each path from the start to the end corresponds exactly to a complete assignment to the nodes.
- Note that in the reduction from a variable-based model to a state-based model, we have committed to an ordering of the variables.
- Each edge has a weight (a single number) determined by the local conditional probabilities (more generally, the factors in a factor graph). For each edge into $\boxed{H_i = h_i}$, we multiply by the transition probability into h_i and emission probability $p(e_i \mid h_i)$. This defines a weight for each path (assignment) in the graph equal to the joint probability $P(H = h, E = e)$.
- Note that the lattice contains $O(Kn)$ nodes and $O(K^2 n)$ edges, where n is the number of variables and K is the number of values in the domain of each variable.

Lattice representation



Forward: $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1}) w(h_{i-1}, h_i)$

sum of weights of paths from start to $H_i = h_i$

Backward: $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1}) w(h_i, h_{i+1})$

sum of weights of paths from $H_i = h_i$ to end

Define $S_i(h_i) = F_i(h_i) B_i(h_i)$:

sum of weights of paths from start to end through $H_i = h_i$

- The point of bringing back the search-based view is that we can cast the probability queries we care about in terms of sums over paths, and effectively use dynamic programming.
- First, define the forward message $F_i(v)$ to be the sum of the weights over all paths from the start node to $\boxed{H_i = v}$. This can be defined recursively: any path that goes $\boxed{H_i = h_i}$ will have to go through some $\boxed{H_{i-1} = h_{i-1}}$, so we can so over all possible values of h_{i-1}.
- Analogously, let the backward message $B_i(v)$ be the sum of the weights over all paths from $\boxed{H_i = v}$ to the end node.
- Finally, define $S_i(v)$ to be the sum of the weights over all paths from the start node to the end node that pass through the intermediate node $\boxed{X_i = v}$. This quantity is just the product of the weights of paths going into $\boxed{H_i = h_i}$ ($F_i(h_i)$) and those leaving it ($B_i(h_i)$).

Lattice representation

Smoothing queries (marginals):

$$\mathbb{P}(H_i = h_i \mid E = e) \propto S_i(h_i)$$

Algorithm: forward-backward algorithm

Compute $F_1, F_2, \ldots, F_L$

Compute $B_L, B_{L-1}, \ldots, B_1$

Compute S_i for each i and normalize

Running time: $O(LK^2)$

- Let us go back to the smoothing queries: $\mathbb{P}(H_i = h_i \mid E = e)$. This is just gotten by normalizing S_i.
- The algorithm is thus as follows: for each node $\boxed{H_i = h_i}$, we compute three numbers: $F_i(h_i), B_i(h_i), S_i(h_i)$. First, we sweep forward to compute all the F_i's recursively. At the same time, we sweep backward to compute all the B_i's recursively. Then we compute S_i by pointwise multiplication.
- Implementation note: we technically can normalize S_i to get $\mathbb{P}(H_i \mid E = e)$ at the very end but it's useful to normalize F_i and B_i at each step to avoid underflow. In addition, normalization of the forward messages yields $\mathbb{P}(H_i = v \mid E_1 = e_1, \ldots, E_i = e_i) \propto F_i(v)$.

Summary

- Lattice representation: paths are assignments (think state-based models)
- Dynamic programming: compute sums efficiently
- Forward-backward algorithm: share intermediate computations across different queries

Roadmap

Forward-backward

Gibbs sampling

Particle filtering

Particle-based approximation

$$\mathbb{P}(X_1, X_2, X_3)$$

Key idea: particles

Use a small set of assignments (particles) to represent a large probability distribution.

x_1	x_2	x_3	$\mathbb{P}(X_1 = x_1, X_2 = x_2, X_3 = x_3)$
0	0	0	0.18
0	0	1	0.02
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0.08
1	1	1	0.72
0.8	**0.8**	**0.74**	**(true marginals)**

	x_1	x_2	x_3
Sample 1	0	0	0
Sample 2	1	1	1
Sample 3	1	1	1
Estimated marginals	**0.67**	**0.67**	**0.67**

- The central idea to both of Gibbs sampling and particle filtering is the use of particles (just a fancy word for complete assignment, usually generated stochastically) to represent a probability distribution.
- Rather than storing the probability of every single assignment, we have a set of assignments, some of which can occur multiple times (which implicitly represents a higher probability). Maintaining this small set of assignments will allows us to answer queries more quickly, but at a cost: our answers will now be approximate instead of exact.
- From a set of particles, we can compute approximate marginals (or any query we want) by simply computing the fraction of assignments that satisfy the desired condition. Here, marginalization is easy because we're explicitly enumerating full assignments.
- Once we have a set of particles, we can compute all the queries we want with it. So now how do we actually generate the particles?

Gibbs sampling

Algorithm: Gibbs sampling

Initialize x to a random complete assignment

Loop through $i = 1, \ldots, n$ until convergence:

Compute weight of $x \cup \{X_i : v\}$ for each v

Choose $x \cup \{X_i : v\}$ with probability prop. to weight

Gibbs sampling (probabilistic interpretation)

Loop through $i = 1, \ldots, n$ until convergence:

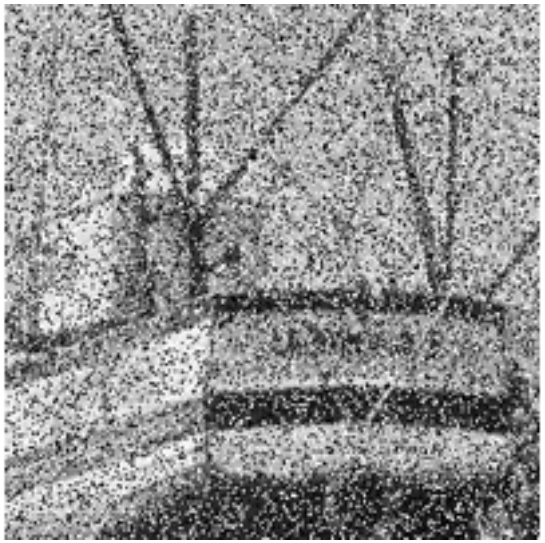
Set $X_i = v$ with prob. $\mathbb{P}(X_i = v \mid X_{-i} = x_{-i})$

(notation: $X_{-i} = X \backslash \{X_i\}$)

[demo]

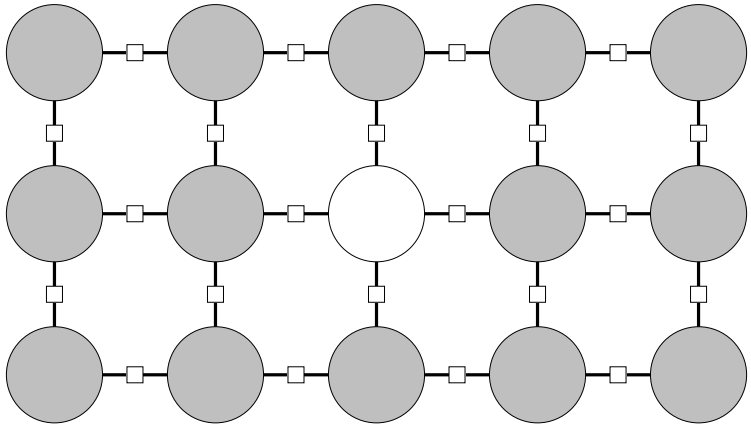
- Recall that Gibbs sampling proceeds by going through each variable X_i, considering all the possible assignments of X_i with some $v \in \text{Domain}_i$, computing the weight of the resulting assignment $x \cup \{X_i : v\}$, and choosing an assignment with probability proportional to the weight.
- We first introduced Gibbs sampling in the context of local search to get out of local optima. Now, we will use it for its original purpose, which is to draw samples from a probability distribution. (In particular, our goal is to draw from the joint distribution over $X_1, X_2, \ldots, X_n$.) To do this, we need to define a probability distribution given an arbitrary factor graph. Recall that a general factor graph defines a $\text{Weight}(x)$, which is the product of all its factors. We can simply normalize the weight to get a distribution: $\mathbb{P}(X = x) \propto \text{Weight}(x)$. Then, Gibbs sampling is exactly sampling according to the conditional distribution $\mathbb{P}(X_i = v \mid X_{-i} = x_{-i})$.
- Note that the convergence criteria is on the *distribution* of the sample values, not the values themselves.
- Advanced: Gibbs sampling is an instance of a Markov Chain Monte Carlo (MCMC) algorithm which generates a sequence of particles $X^{(1)}, X^{(2)}, X^{(3)}, \ldots$. A Markov chain is irreducible if there is positive probability of getting from any assignment to any other assignment (now the probabilities are over the random choices of the sampler). When the Gibbs sampler is irreducible, then in the limit as $t \to \infty$, the distribution of $X^{(t)}$ converges to the true distribution $\mathbb{P}(X)$. MCMC is a very rich topic which we will not talk about very much here.

Application: image denoising



Application: image denoising

Example: image denoising



- $X_i \in \{0, 1\}$ is pixel value in location i
- Subset of pixels are observed

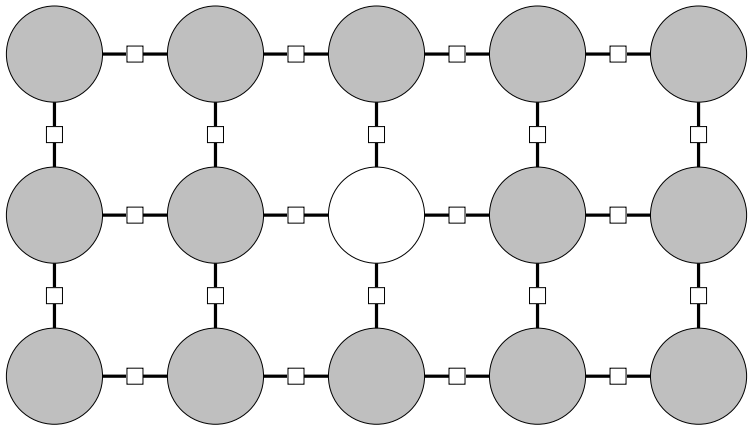
$o_i(x_i) = [x_i = \text{observed value at } i]$

- Neighboring pixels more likely to be same than different

$t_{ij}(x_i, x_j) = [x_i = x_j] + 1$

- Factor graphs (sometimes referred to as Markov random fields) play an important role in computer vision applications. Here we take a look at a very simple image denoising application and construct a factor graph.
- We assume that we have observed some fraction of the pixels in an image, and we wish to recover the pixels which have been removed. Our simple factor graph has two types of factors.
- First, $o_i(x_i) = 0$ if the pixel at i is observed and x_i does not match it. This is analogous to the emission probabilities in an HMM, but here, the factor is deterministic (0 or 1).
- Second, $t_{ij}(x_i, x_j)$ exists for every pair of neighboring pixels, and encourages them to agree (both be 0 or both be 1). Weight 2 is given to those pairs which are the same and 1 if the pair is different. This is analogous to the transition probabilities in an HMM.

Application: image denoising



Example: image denoising

If neighbors are $1, 1, 1, 0$ and X_i not observed:

$$\mathbb{P}(X_i = 1 \mid X_{-i} = x_{-i}) = \frac{2 \cdot 2 \cdot 2 \cdot 1}{2 \cdot 2 \cdot 2 \cdot 1 + 1 \cdot 1 \cdot 1 \cdot 2} = 0.8$$

If neighbors are $0, 1, 0, 1$ and X_i not observed:

$$\mathbb{P}(X_i = 1 \mid X_{-i} = x_{-i}) = \frac{1 \cdot 2 \cdot 1 \cdot 2}{1 \cdot 2 \cdot 1 \cdot 2 + 2 \cdot 1 \cdot 2 \cdot 1} = 0.5$$

[whiteboard]

- Let us compute the Gibbs sampling update. We go through each pixel X_i and try to update its value. Specifically, we condition on $X_{-i} = x_{-i}$ being the current value.
- To compute the conditional $\mathbb{P}(X_i = x_i \mid X_{-i} = x_{-i})$, we only need to multiply in the factors that are touching X_i, since conditioning disconnects the graph completely. Many of the factors simply can be ignored.
- For example, suppose we are updating variable X_i, which has neighbors X_j, X_k, X_l, X_m. Suppose we condition on $X_j = 1, X_k = 1, X_l = 0, X_m = 1$. Then $\mathbb{P}(X_i = x_i \mid X_{-i} = x_{-i}) \propto t_{ij}(x_i, 1)t_{ik}(x_i, 1)t_{il}(x_i, 0)t_{im}(x_m, 1)o_i(x_i)$. We can compute the RHS for both $x_i = 1$ and $x_i = 0$, and then normalize to get the conditional probability 0.8.
- Intuitively, the neighbors are all trying to pull X_i towards their values, and 0.8 reflects the fact that the pull towards 1 is stronger.

Gibbs sampling: demo

[see web version]

- Try playing with the demo by modifying the settings to get a feeling for what Gibbs sampling is doing. Each iteration corresponds to resampling each pixel (variable).
- When you hit ctrl-enter for the first time, red and black correspond to 1 and 0, and white corresponds to unobserved.
- `showMarginals` allows you to either view the particles produced or the marginals estimated from the particles (this gives you a smoother probability estimate of what the pixel values are).
- If you increase `missingFrac`, the problem becomes harder.
- If you set `coherenceFactor` to 1, this is equivalent to turning off the edge factors.
- If you set `icm` to true, we will use local search rather than Gibbs sampling, which produces very bad solutions.
- In summary, Gibbs sampling is an algorithm that can be applied to any factor graph. It performs a guided random walk in the space of all possible assignments. Under some mild conditions, Gibbs sampling is guaranteed to converge to a sample from the true distribution $\mathbb{P}(X = x) \propto \text{Weight}(x)$ but this might take until the heat death of the universe for complex models. Nonetheless, Gibbs sampling is widely used due to its simplicity and can work reasonably well.

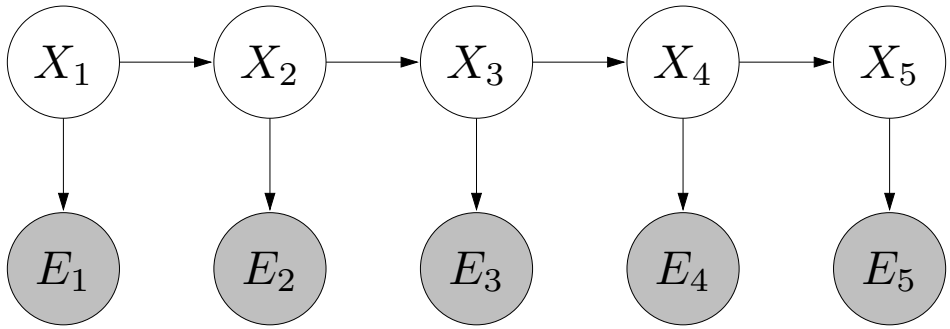
Roadmap

Forward-backward

Gibbs sampling

Particle filtering

Hidden Markov models



$$\mathbb{P}(X = x, E = e) = \underbrace{p(x_1)}_{\text{start}} \prod_{i=2}^{n} \underbrace{p(x_i \mid x_{i-1})}_{\text{transition}} \prod_{i=1}^{n} \underbrace{p(e_i \mid x_i)}_{\text{emission}}$$

Query (**filtering**):

$$\mathbb{P}(X_1 \mid E_1 = e_1)$$

$$\mathbb{P}(X_2 \mid E_1 = e_1, E_2 = e_2)$$

$$\mathbb{P}(X_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3)$$

- Now we turn our attention to particle filtering. Gibbs sampling is the probabilistic analog of local search methods such as ICM, and particle filtering is the probabilistic analog of partial search such as beam search.
- Although particle filtering applies to general factor graphs, we will present them for hidden Markov models for concreteness.
- As the name suggests, we will use particle filtering for answering filtering queries.

Review: beam search

Idea: keep $\leq K$ **candidate list** C of partial assignments

Algorithm: beam search

Initialize $C \leftarrow [\{\}]$

For each $i = 1, \ldots, n$:

Extend:

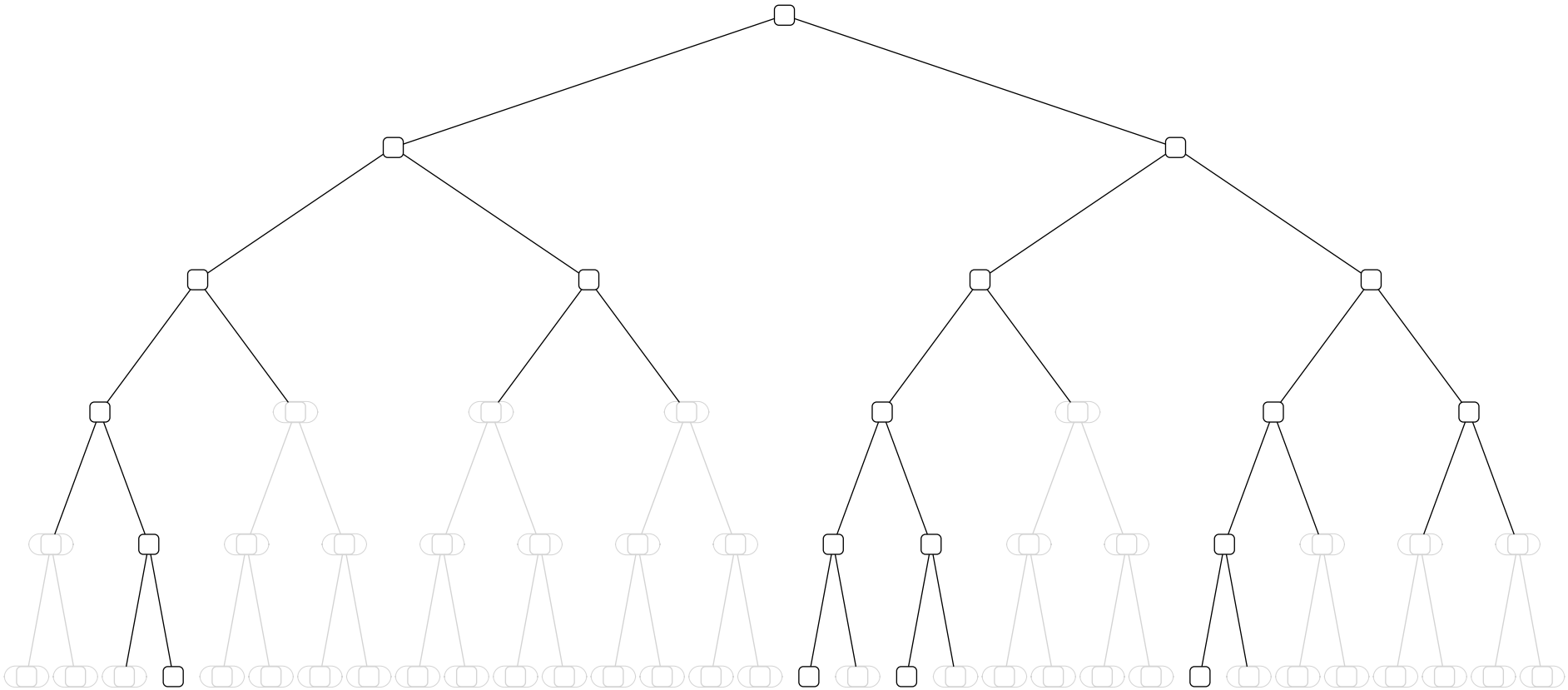
$C' \leftarrow \{x \cup \{X_i : v\} : x \in C, v \in \text{Domain}_i\}$

Prune:

$C \leftarrow K$ elements of C' with highest weights

[demo: `beamSearch({K:3})`]

Review: beam search



Beam size $K = 4$

- Recall that beam search effectively does a pruned BFS of the search tree of partial assignments, where at each level, we keep track of the K partial assignments with the highest weight.
- There are two phases. In the first phase, we **extend** all the existing candidates C to all possible assignments to X_i; this results in $K = |\text{Domain}_i|$ candidates C'. These C' are sorted by weight and **pruned** by taking the top K.

Beam search

End result:

- Candidate list C is set of particles
- Use C to compute marginals

Problems:

- Extend: slow because requires considering every possible value for X_i
- Prune: greedily taking best K doesn't provide diversity

Solution (3 steps): **propose, weight, resample**

- Beam search does generate a set of particles, but there are two problems.
- First, it can be slow if Domain_i is large because we have to try every single value. Perhaps we can be smarter about which values to try.
- Second, we are greedily taking the top K candidates, which can be too myopic. Can we somehow encourage more diversity?
- Particle filtering addresses both of these problems. There are three steps: propose, which extends the current parital assignment, and weight/resample, which redistributes resources on the particles based on evidence.

Step 1: propose

Old particles: $\approx \mathbb{P}(X_1, X_2 \mid E_1 = 0, E_2 = 1)$

[0, 1]
[1, 0]

Key idea: proposal distribution

For each old particle (x_1, x_2), sample $X_3 \sim p(x_3 \mid x_2)$.

New particles: $\approx \mathbb{P}(X_1, X_2, X_3 \mid E_1 = 0, E_2 = 1)$

[0, 1, 1]
[1, 0, 0]

- Suppose we have a set of particles that approximates the filtering distribution over X_1, X_2. The first step is to extend each current partial assignment (particle) from $x_{1:i-1} = (x_1, \ldots, x_{i-1})$ to $x_{1:i} = (x_1, \ldots, x_i)$.
- To do this, we simply go through each particle and extend it stocastically, using the transition probability $p(x_i \mid x_{i-1})$ to sample a new value of X_i.
- (For concreteness, think of what will happen if $p(x_i \mid x_{i-1}) = 0.8$ if $x_i = x_{i-1}$ and 0.2 otherwise.)
- We can think of advancing each particle according to the dynamics of the HMM. These particles approximate the probability of X_1, X_2, X_3, but still conditioned on the same evidence.

Step 2: weight

Old particles: $\approx \mathbb{P}(X_1, X_2, X_3 \mid E_1 = 0, E_2 = 1)$

[0, 1, 1]

[1, 0, 0]

Key idea: weighting

For each old particle (x_1, x_2, x_3), weight it by $w(x_1, x_2, x_3) = p(e_3 \mid x_3)$.

New particles:

$$\approx \mathbb{P}(X_1, X_2, X_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$$

[0, 1, 1] (0.8)

[1, 0, 0] (0.4)

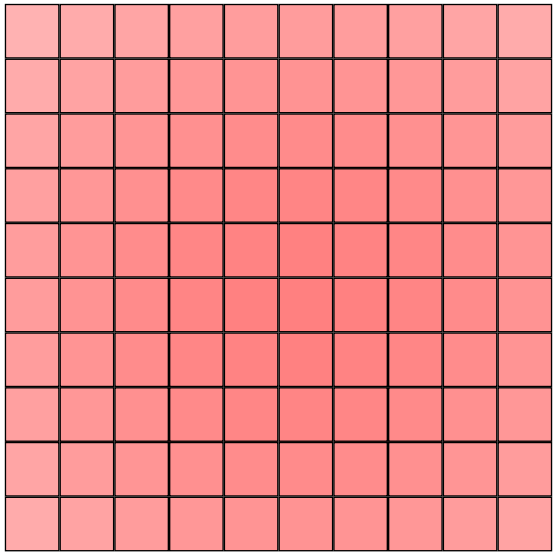
- Having generated a set of K candidates, we need to now take into account the new evidence $E_i = e_i$. This is a deterministic step that simply weights each particle by the probability of generating $E_i = e_i$, which is the emission probability $p(e_i \mid x_i)$.
- Intuitively, the proposal was just a guess about where the object will be X_3. To get a more realistic picture, we condition on $E_3 = 1$. Supposing that $p(e_i = 1 \mid x_i = 1) = 0.8$ and $p(e_i = 1 \mid x_i = 0) = 0.4$, we then get the weights 0.8 and 0.4. Note that these weights do not sum to 1.

Step 3: resample

Question: given weighted particles, which to choose?

Tricky situation:

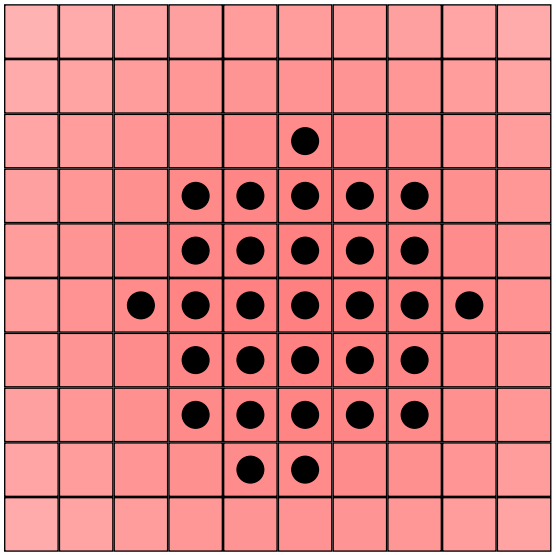
- Target distribution close to uniform
- Fewer particles than locations



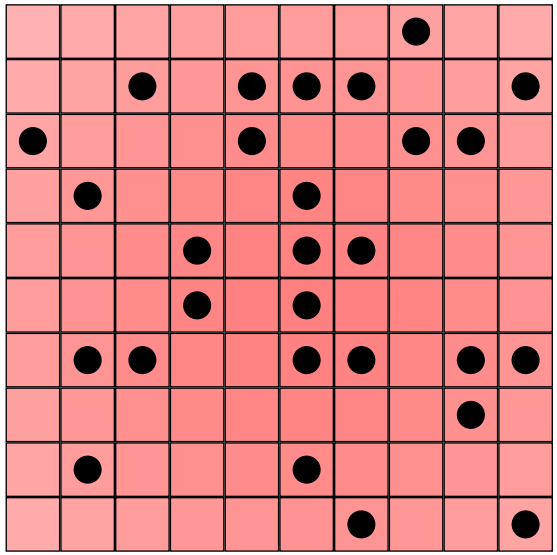
- Having proposed extensions to the particles and computed a weight for each particle, we now come to the question of which particles to keep.
- Intuitively, if a particle has very small weight, then we might want to prune it away. On the other hand, if a particle has high weight, maybe we should dedicate more resources to it.
- As a motivating example, consider an almost uniform distribution over a set of locations, and trying to represent this distribution with fewer particles than locations. This is a tough situation to be in.

Step 3: resample

K with highest weight



K sampled from distribution



Intuition: top K assignments not representative.

Maybe random samples will be more representative...

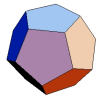
- Beam search, which would choose the K locations with the highest weight, would clump all the particles near the mode. This is risky, because we have no support out farther from the center, where there is actually substantial probability.
- However, if we sample from the distribution which is proportional to the weights, then we can hedge our bets and get a more representative set of particles which cover the space more evenly.

Step 3: resample

Key idea: resampling

Given a distribution $\mathbb{P}(A = a)$ with n possible values, draw a sample K times.

Intuition: redistribute particles to more promising areas



Example: resampling

a	$\mathbb{P}(A = a)$
a1	0.70
a2	0.20
a3	0.05
a4	0.05

sample 1	a1
sample 2	a2
sample 3	a1
sample 4	a1

- After proposing and weighting, we end up with a set of samples $x_{1:i}$, each with some weight $w(x_{1:i})$. Intuitively, if $w(x_{1:i})$ is really small, then it might not be worth keeping that particle around.
- Resampling allows us to put (possibly multiple) particles on high weight particles. In the example above, we don't sample a3 and a4 because they have low probability of being sampled.

Step 3: resample

Old particles:

$$\approx \mathbb{P}(X_1, X_2, X_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$$

[0, 1, 1] (0.8) $\Rightarrow 2/3$
[1, 0, 0] (0.4) $\Rightarrow 1/3$

New particles:

$$\approx \mathbb{P}(X_1, X_2, X_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$$

[0, 1, 1]
[0, 1, 1]

- In our example, we normalize the particle weights to form a distribution over particles. Then we draw $K = 2$ independent samples from that distribution. Higher weight particles will get more samples.

Particle filtering

Algorithm: particle filtering

Initialize $C \leftarrow [\{\}]$

For each $i = 1, \ldots, n$:

Propose (extend):

$$C' \leftarrow \{(x_{1:i-1}, x_i) : x_{1:i-1} \in C, x_i \sim p(x_i \mid x_{i-1})\}$$

Reweight:

Compute weights $w(x_{1:i}) = p(e_i \mid x_i)$ for $x \in C'$

Resample (prune):

$C \leftarrow K$ elements drawn independently from $\propto w(x_{1:i})$

[demo: `particleFiltering({K:100})`]

- The final algorithm here is very similar to beam search. We go through all the variables $X_1, \ldots, X_n$.
- For each candidate $x_{i-1} \in C$, we propose x_i according to the transition distribution $p(x_i \mid x_{i-1})$.
- We then weight this particle using $w(x_{1:i}) = p(e_i \mid x_i)$.
- Finally, we select K particles from $\propto w(x_{1:i})$ by sampling K times independently.

Particle filtering: implementation

- If only care about last X_i, collapse all particles with same X_i (think elimination)

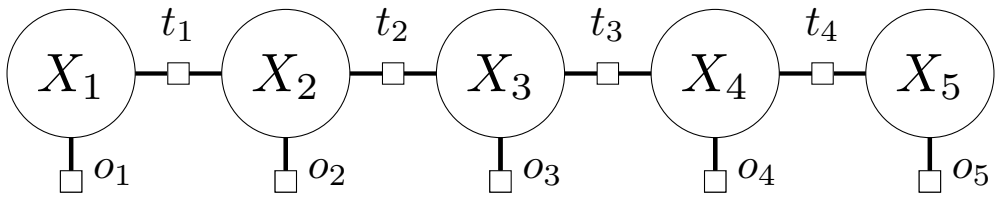
$$001 \Rightarrow 1$$
$$101 \Rightarrow 1$$
$$010 \Rightarrow 0$$
$$010 \Rightarrow 0$$
$$110 \Rightarrow 0$$

- If many particles are the same, can just store counts

$$\begin{matrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{matrix} \Rightarrow \begin{matrix} 1:2 \\ 0:3 \end{matrix}$$

- In particle filtering as it is currently defined, each particle is an entire trajectory in the context of object tracking (assignment to all the variables).
- Often in tracking applications, we only care about the last location X_i, and the HMM is such that the future $(X_{i+1}, \ldots, X_n)$ is conditionally independent of $X_1, \ldots, X_{i-1}$ given X_i. Therefore, we often just store the value of X_i rather than its entire ancestry.
- When we only keep track of the X_i, we might have many particles that have the same value, so it can be useful to store just the counts of each value rather than having duplicates.

Application: tracking



Example: tracking

- X_i: position of object at i
- Transitions: $t_i(x_i, x_{i+1}) = [x_i \text{ near } x_{i+1}]$
- Observations: $o_i(x_i) =$ sensor reading...

Particle filtering demo

[see web version]

- Consider a tracking application where an object is moving around in a grid and we are trying to figure out its location $X_i \in \{1, \ldots, \text{grid-width}\} \times \{1, \ldots, \text{grid-height}\}$.
- The transition factors say that from one time step to the next, the object is equally likely to have moved north, south, east, west, or stayed put.
- Each observation is a location on the grid (a yellow dot). The observation factor is a user-defined function which depends on the vertical and horizontal distance.
- Play around with the demo to get a sense of how particle filtering works, especially the different observation factors.

Probabilistic inference

Model (Bayesian network or factor graph):

$$\mathbb{P}(X = x) = \prod_{i=1}^{n} p(x_i \mid x_{\mathsf{Parents}(i)})$$

Probabilistic inference:

$$\mathbb{P}(Q \mid E = e)$$

Algorithms:

- Forward-backward: chain-structured (HMMs), exact
- Gibbs sampling, particle filtering: general, approximate

Next time: learning