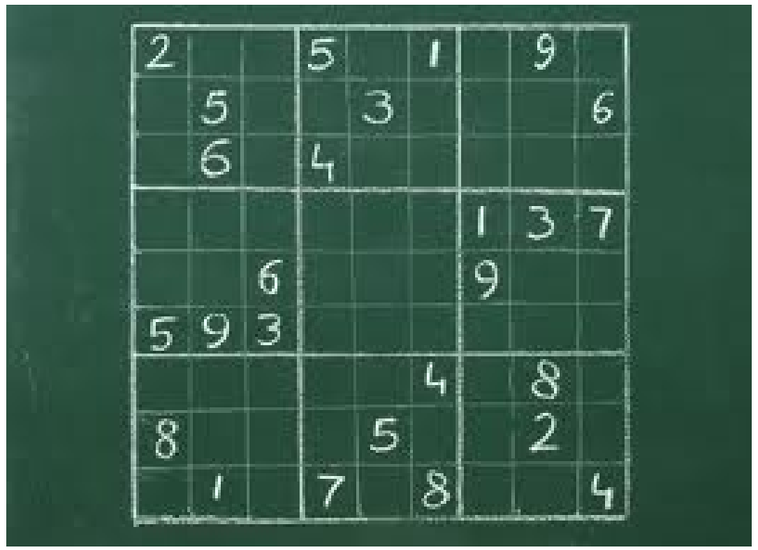
CSPs, search, and markov networks



- In this lecture, we will discuss beam search, a simple heuristic algorithm for find an approximate maximum weight assignments efficiently without incurring the full cost of backtracking search.

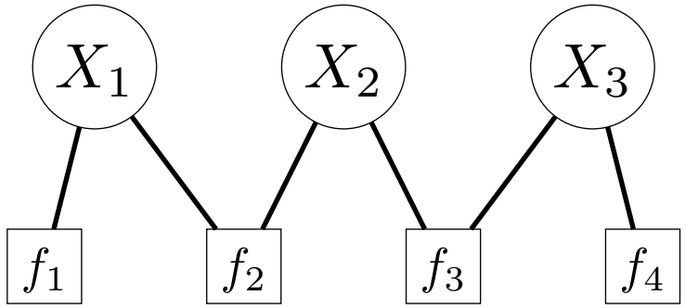
Announcements

- Section **315pm-445pm, Huang 018**

- Helpful as you prepare for exams

- announcements before we get started
- Sections are at 315 - 445pm, and may be helpful as you start to prep for exams

Review: CSPs



Definition: factor graph

Variables:

$X = (X_1, \ldots, X_n)$, where $X_i \in \text{Domain}_i$

Factors:

$f_1, \ldots, f_m$, with each $f_j(X) \geq 0$

Definition: assignment weight

Each **assignment** $x = (x_1, \ldots, x_n)$ has a **weight**:

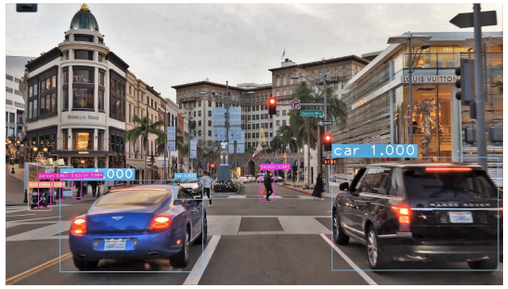
$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

Objective:

$$\arg\max_x \text{Weight}(x)$$

- Recall that a constraint satisfaction problem is defined by a factor graph, where we have a set of variables and a set of factors. Each assignment of values to variables has a weight, and the objective is to find the assignment with the maximum weight.

Example: object tracking

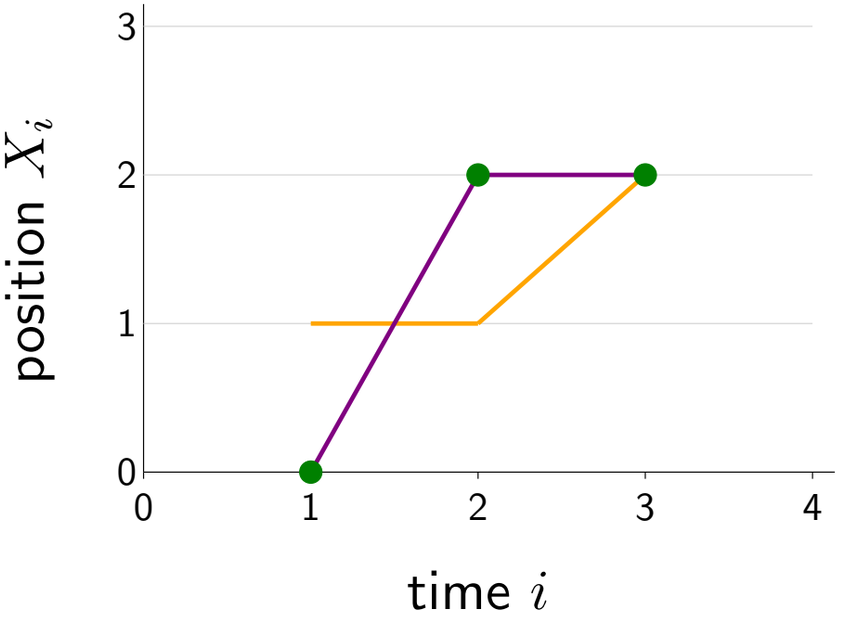


Problem: object tracking

(O) Noisy sensors report positions: $0, 2, 2$.

(T) Objects can't teleport.

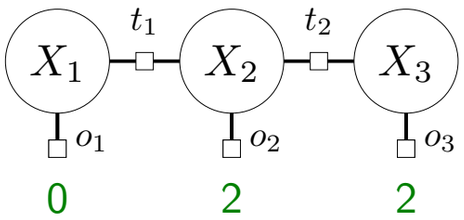
What trajectory did the object take?



- In this example, consider the problem of object tracking. For instance, for autonomous driving, objects such as cars and pedestrians must be tracked to know where not to drive.
- Here, at each discrete time step i, we are given some noisy information about where the object might be. For example, this noisy information could be the video frame at time step i. The goal is to answer the question: what trajectory did the object take?
- To simplify, suppose we consider an object moving in 1D and we have a sensor that tells us an approximate position at each time step. We observe 0, 2, 2 from this sensor.

Example: object tracking CSP

Factor graph:



x_1	$o_1(x_1)$
0	2
1	1
2	0

x_2	$o_2(x_2)$
0	0
1	1
2	2

x_3	$o_3(x_3)$
0	0
1	1
2	2

$\lvert x_i - x_{i+1} \rvert$	$t_i(x_i, x_{i+1})$
0	2
1	1
2	0

[demo]

- Variables $X_i \in \{0, 1, 2\}$: position of object at time i
- Observation factors $o_i(x_i)$: noisy information compatible with position
- Transition factors $t_i(x_i, x_{i+1})$: object positions can't change too much

- Let's try to model this problem. Always start by defining the variables: these are the quantities which we don't know. In this case, it's the positions of the object at each time step: $X_1, X_2, X_3 \in \{0, 1, 2\}$.
- Now let's think about the factors, which need to capture two things. First, transition factors make sure physics isn't violated (e.g., object positions can't change too much). Second, observation factors make sure the hypothesized positions X_i are compatible with the noisy information. Note that these numbers returned by the factors are just numbers, not necessarily probabilities.
- Having modeled the problem as a factor graph, we can now ask for the maximum weight assignment, which gives us the most likely trajectory for the object.
- Click on the the [track] demo to see the definition of this factor graph as well as the maximum weight assignment, which is [1, 2, 2]. Note that this trajectory is a smoothed version of the observations, which assumes that the first sensor reading was inaccurate.

Roadmap

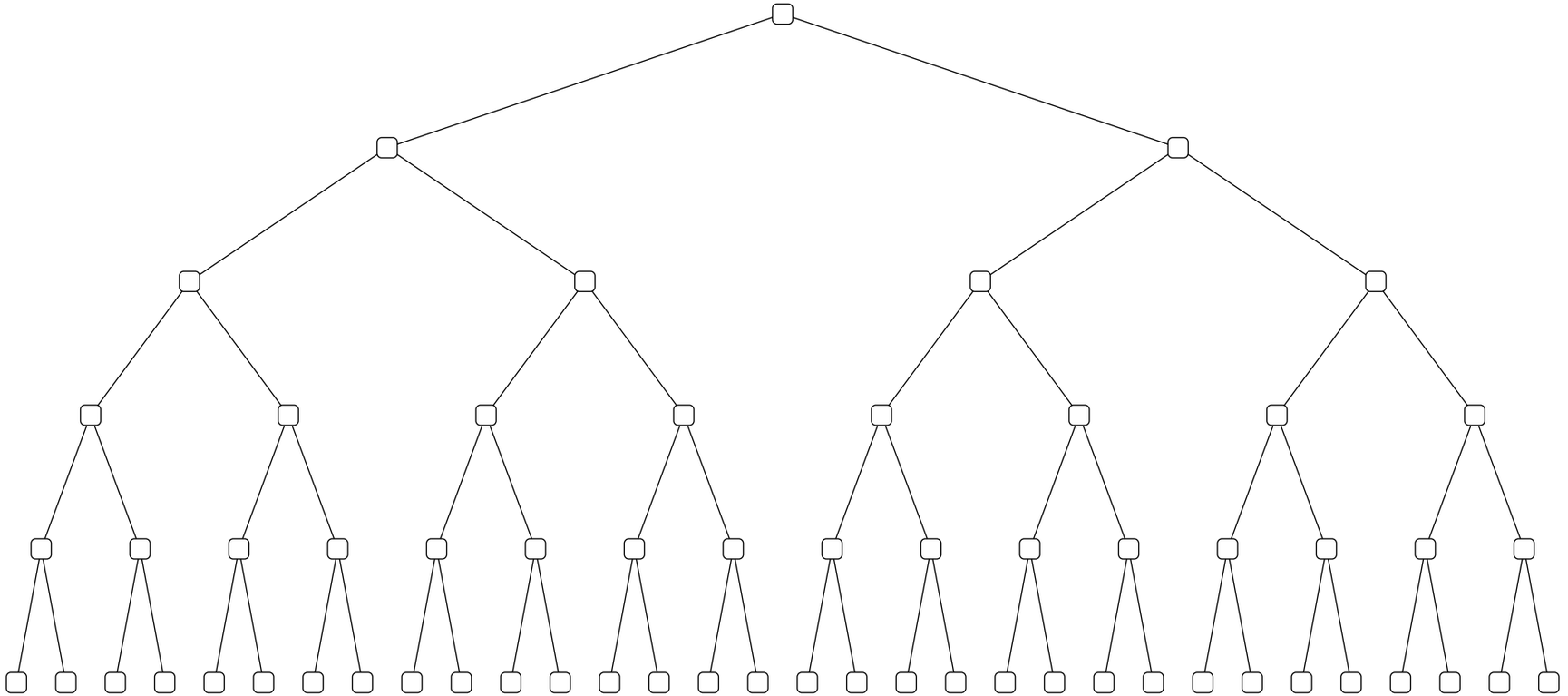
Beam search (c4.1.3)

Local search

Gibbs sampling (c14.5.2)

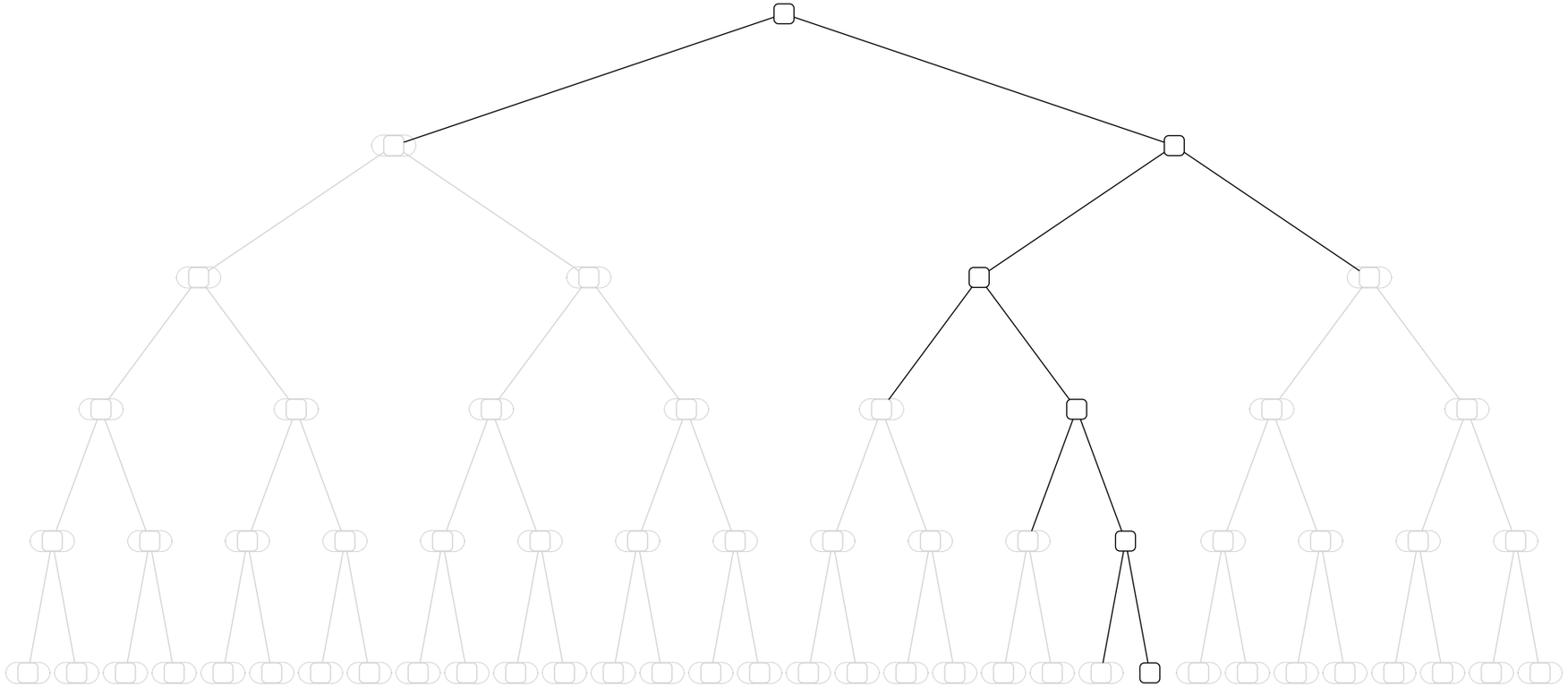
Cond. Independence (c14.2.2)

Backtracking search



- Backtracking search in the worst case performs an exhaustive DFS of the entire search tree, which can take a very very long time. How do we avoid this?

Greedy search



- One option is to simply not backtrack!
- Pictorially, at each point in the search tree, we choose the option that seems myopically best, and march down one thin slice of the search tree, never looking back.

Greedy search

Algorithm: greedy search

Partial assignment $x \leftarrow \{\}$

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

Extend:

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

Prune:

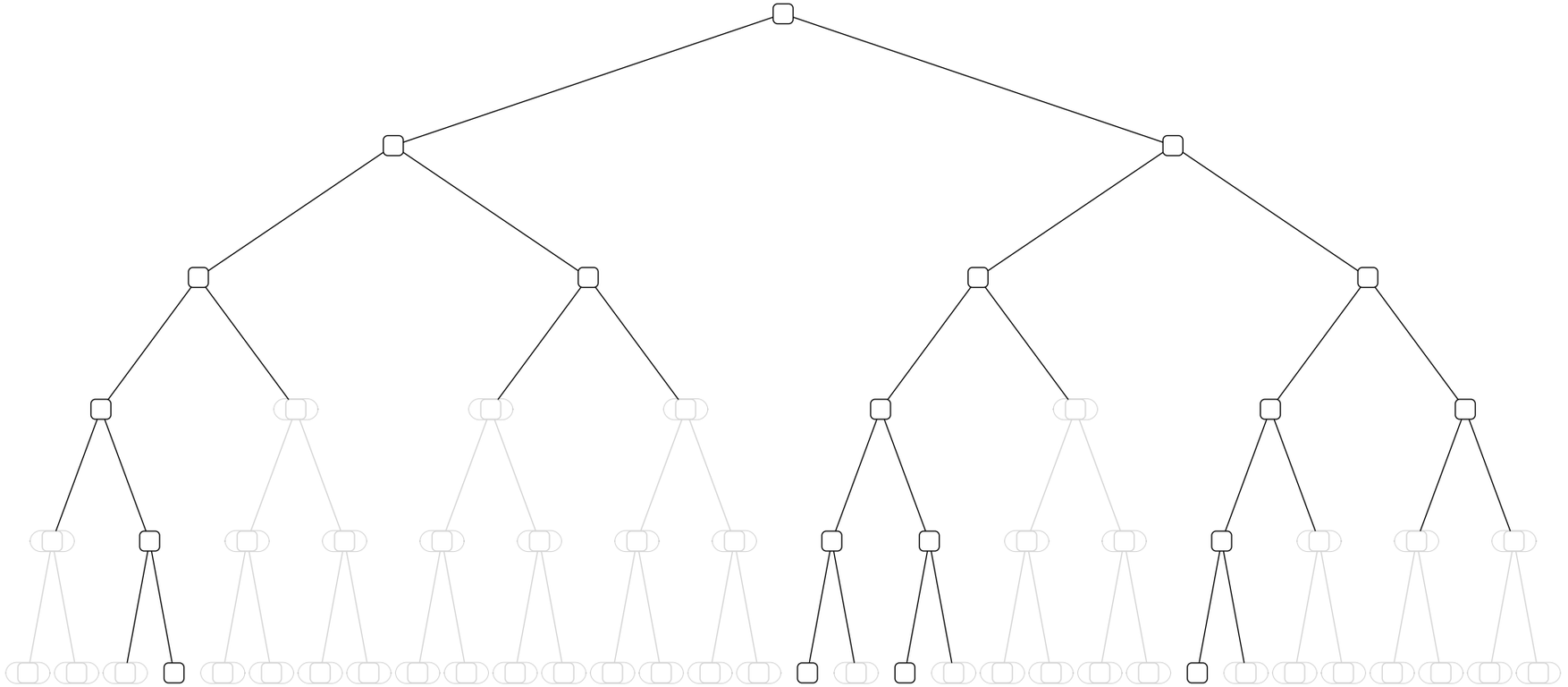
$x \leftarrow x_v$ with highest weight

Not guaranteed to find maximum weight assignment!

[demo: `beamSearch({K:1})`]

- Specifically, we assume we have a fixed ordering of the variables. As in backtracking search, we maintain a partial assignment x and its weight. We consider extending x to include $X_i : v$ for all possible values $v \in \text{Domain}_i$. Then instead of recursing on all possible values of v, we just commit to the best one according to the weight of the new partial assignment $x \cup \{X_i : v\}$.
- It's important to realize that "best" here is only with respect to the weight of the partial assignment $x \cup \{X_i : v\}$. The greedy algorithm is by no means guaranteed to find the globally optimal solution. Nonetheless, it is incredibly fast and sometimes good enough.
- In the demo, you'll notice that greedy search produces a suboptimal solution.

Beam search



Beam size $K = 4$

- The problem with greedy is that it's too myopic. So a natural solution is to keep track of more than just the single best partial assignment at each level of the search tree. This is exactly **beam search**, which keeps track of (at most) K candidates (K is called the beam size). It's important to remember that these candidates are not guaranteed to be the K best at each level (otherwise greedy would be optimal).

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:

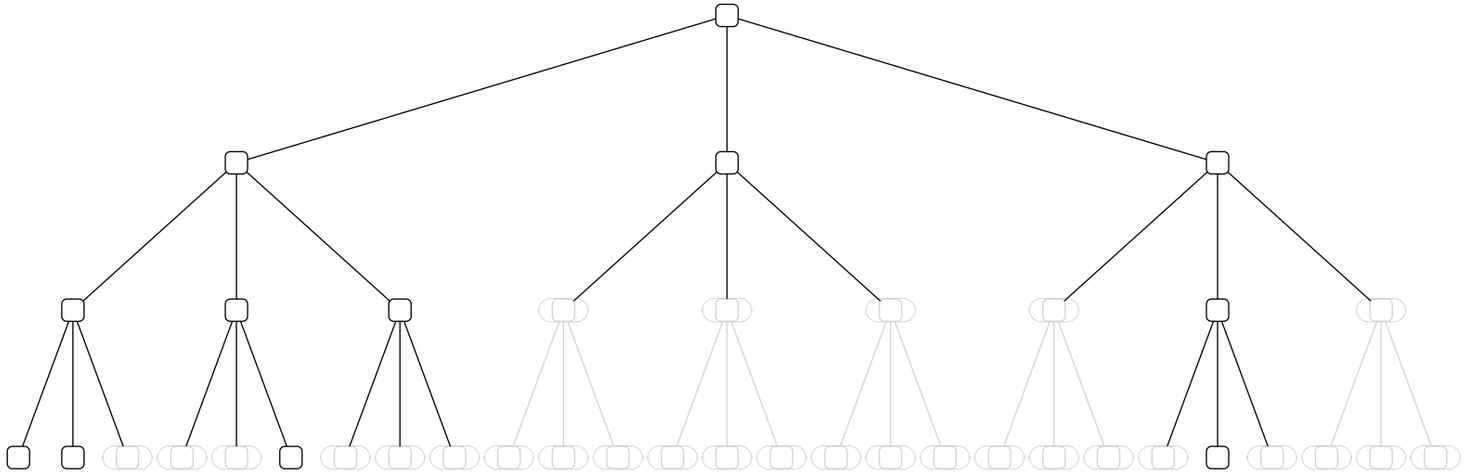
$C \leftarrow K$ elements of C' with highest weights

Not guaranteed to find maximum weight assignment!

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

- The beam search algorithm maintains set of candidates C and iterates through all the variables, just as in greedy.
- It extends each candidate partial assignment $x \in C$ with every possible $X_i : v$. This produces a new candidate list C'.
- We compute the weight for each new candidate in C' and then keep the K elements with the largest weight.
- Like greedy, beam search also has no guarantees of finding the maximum weight assignment, but it generally works better than greedy.
- In the demo, let's examine the object tracking CSP from before. Let us run beam search with $K = 3$. We start with the empty assignment, and extend to the three candidates for X_1. These are pruned down to $K = 3$ (so nothing happens). Then we extend each of the partial assignments to all the possible values of X_2, compute each of their weights, and then we prune down to the $K = 3$ candidates with the largest weight. Then the same with X_3. Finally, we see that the highest weight (full) assignment is $\{X_1 : 1, X_2 : 2, X_3 : 2\}$, which has a weight of 8. In this case, we got lucky and ended up with the globally optimal solution, but remember that beam search is in general not guaranteed to find the maximum weight assignment.

Time complexity



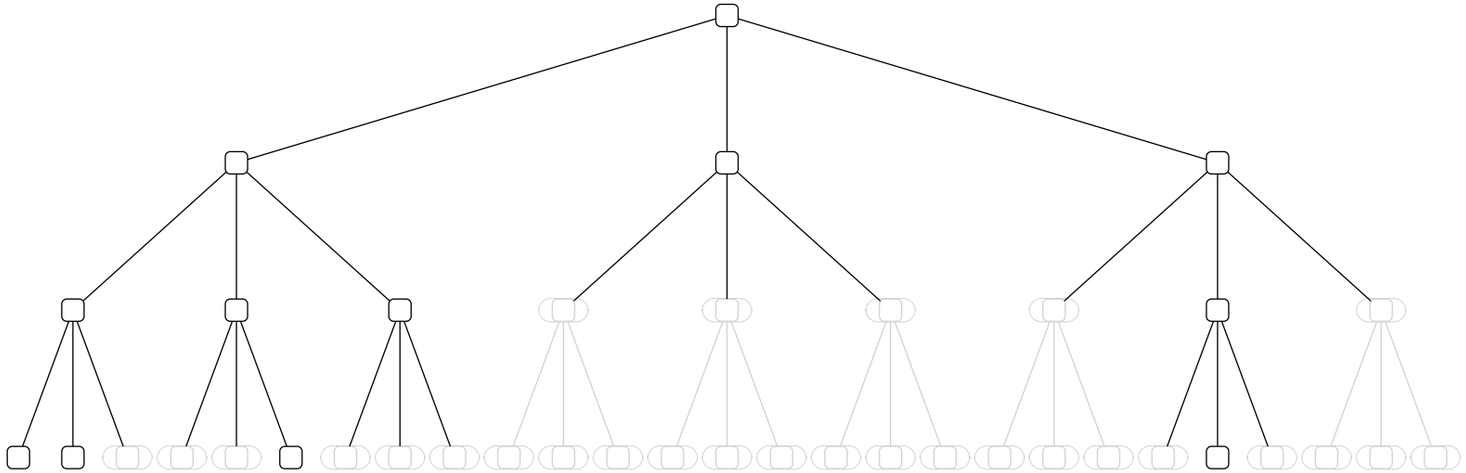
n variables (depth)

Branching factor $b = |\text{Domain}_i|$ → Time: $O(nKb \log K)$

Beam size K

- The advantage of beam search is that the time complexity is very predictable.
- Suppose we have a CSP with n variables; this is the depth of the search tree. Each variable X_i can take on $|\text{Domain}_i|$ values, and for simplicity, assume all these values are b, which is the branching factor of the search tree. Finally, we have the beam size K, which is the number of paths down the search tree we're entertaining.
- For each of the n levels, we have to iterate over K candidates, extend each one by b. The time it takes to select the K largest elements from a list of Kb elements is $Kb \log K$ by using a heap.

Summary



- Beam size K controls tradeoff between efficiency and accuracy
  - $K = 1$ is greedy search ($O(nb)$ time)
  - $K = \infty$ is BFS ($O(b^n)$ time)

Backtracking search : DFS :: beam search : pruned BFS

- In summary, we have presented a simple heuristic for approximating maximum weight assignments, beam search.
- Beam search offers a nice way to tradeoff efficiency and accuracy and is used quite commonly in practice, especially on naturally sequential problems like optimizing over sentences (sequences of words) or trajectories (sequences of positions).
- If you want speed and don't need extremely high accuracy, use $K = 1$, which recovers the greedy algorithm. The running time is $O(nb)$, since for each of the n variables, we need to consider b possible values in the domain.
- With large enough K (no pruning), beam search is just doing a BFS traversal of the search tree (whereas backtracking search performs a DFS traversal), which takes $O(b^n)$ time.
- To draw a connection between perhaps more familiar tree search algorithms, think of backtracking search as performing a DFS of the search tree.
- Beam search is like a pruned version of BFS, where we are still exploring the tree layer by layer, but we use the factors that we've seen so far to aggressively cut out the branches of the tree that are not worth exploring further.

Roadmap

Beam search (c4.1.3)

Local search

Gibbs sampling (c14.5.2)

Cond. Independence (c14.2.2)

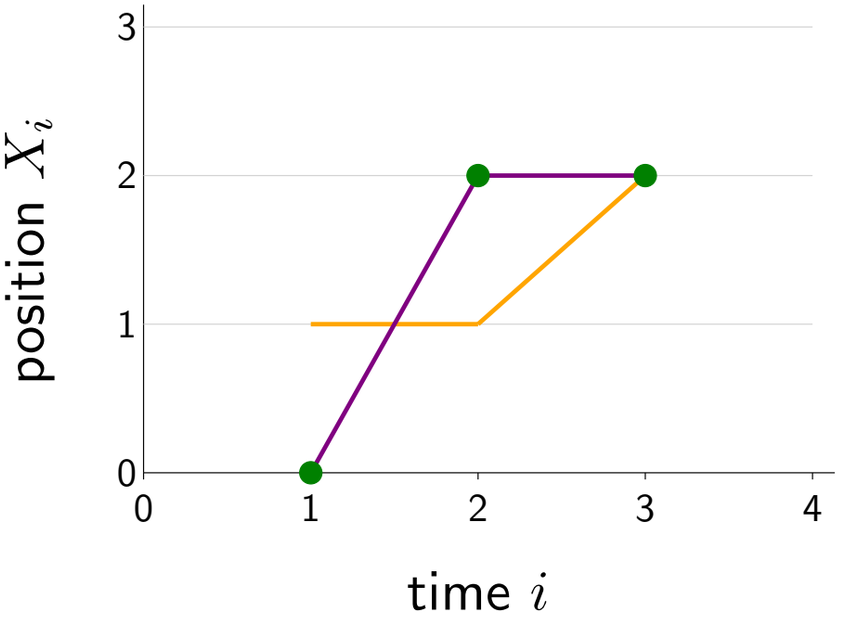
Search strategies

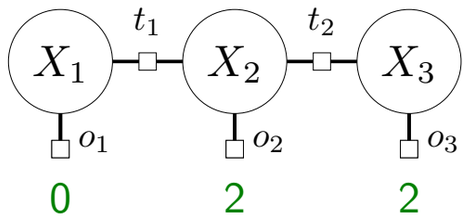
Backtracking/beam search: extend partial assignments

Local search: modify complete assignments

- So far, we've seen both backtracking and beam search. These search algorithms build up a partial assignment incrementally, and are structured around an ordering of the variables (even if it's dynamically chosen). With backtracking search, we can't just go back and change the value of a variable much higher in the tree due to new information; we have to wait until the backtracking takes us back up, in which case we lose all the information about the more recent variables. With beam search, we can't even go back at all.
- **Local search** (i.e., hill climbing) provides us with additional flexibility. Instead of building up partial assignments, we work with a complete assignment and make repairs by changing one variable at a time.

Example: object tracking





x_1	$o_1(x_1)$
0	2
1	1
2	0

x_2	$o_2(x_2)$
0	0
1	1
2	2

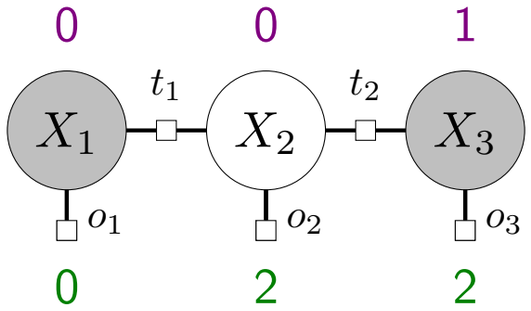
x_3	$o_3(x_3)$
0	0
1	1
2	2

$\lvert x_i - x_{i+1} \rvert$	$t_i(x_i, x_{i+1})$
0	2
1	1
2	0

[demo]

- Recall the object tracking example in which we observe noisy sensor readings 0, 2, 2.
- We have observation factors o_i that encourage the position X_i and the corresponding sensor reading to be nearby.
- We also have transition factors t_i that encourage the positions X_i and X_{i+1} to be nearby.

One small step



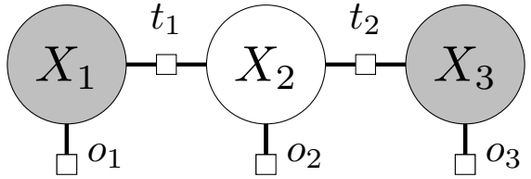
Old assignment: $(0, 0, 1)$; how to improve?

(x_1, v, x_3)	weight
$(0, 0, 1)$	$2 \cdot 2 \cdot 0 \cdot 1 \cdot 1 = 0$
$(0, 1, 1)$	$2 \cdot 1 \cdot 1 \cdot 2 \cdot 1 = 4$
$(0, 2, 1)$	$2 \cdot 0 \cdot 2 \cdot 1 \cdot 1 = 0$

New assignment: $(0, 1, 1)$

- Suppose we have a complete assignment $(0, 0, 1)$, perhaps randomly generated. This complete assignment has weight 0.
- Can we make a local change to the assignment to improve the weight? Let's just try setting x_2 to a new value v.
- For each possible value v, we compute the weight of the resulting assignment from setting $x_2 : v$.
- We then just take the v that produces the maximum weight.
- This results in a new assignment $(0, 1, 1)$ with a higher weight (4 rather than 0).
- This is one step of ICM, and one can now take another variable and try to change its value to improve the weight of the complete assignment.

Exploiting locality



Weight of new assignment (x_1, v, x_3):

$$o_1(x_1)t_1(x_1, v)o_2(v)t_2(v, x_3)o_3(x_3)$$

Key idea: locality

When evaluating possible re-assignments to X_i, only need to consider the factors that depend on X_i.

- There is one optimization we can make. If we write down the weight of a new assignment $x \cup \{X_2 : v\}$, we will notice that all the factors return the same value as before except the ones that depend on X_2.
- Therefore, we only need to compute the product of these relevant factors and take the maximum weight. Because we only need to look at the factors that touch the variable we're modifying, this can be a big saving if the total number of factors is much larger.

Iterated conditional modes (ICM)

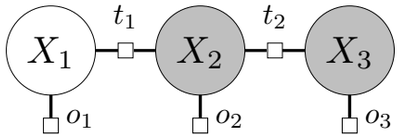
Algorithm: iterated conditional modes (ICM)

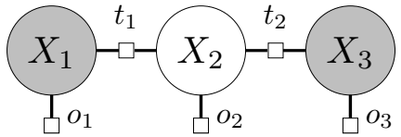
Initialize x to a random complete assignment

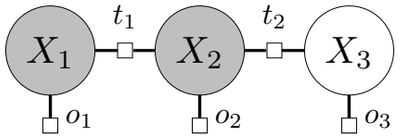
Loop through $i = 1, \ldots, n$ until convergence:

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

$x \leftarrow x_v$ with highest weight





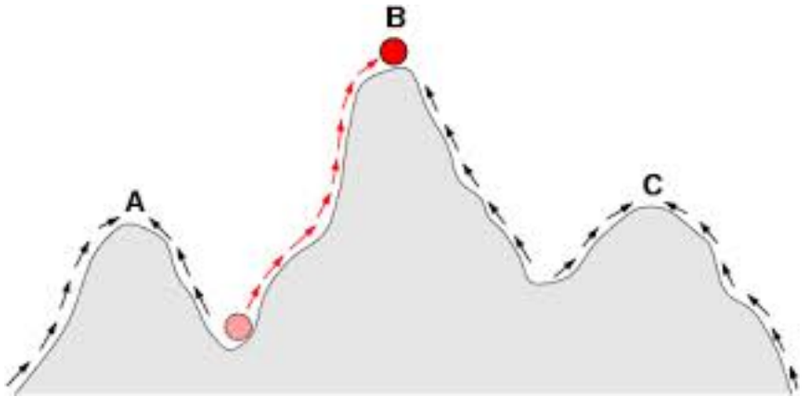


[demo: `iteratedConditionalModes()`]

- Now we can state our first algorithm, ICM. The idea is simple: we start with a random complete assignment. We repeatedly loop through all the variables X_i.
- On variable X_i, we consider all possible ways of re-assigning it $X_i : v$ for $v \in \text{Domain}_i$, and choose the new assignment that has the highest weight.
- Graphically, we represent each step of the algorithm by having shaded nodes for the variables which are fixed and unshaded for the single variable which is being re-assigned.
- Note that in the demo, ICM gets stuck in a local optimum with weight 4 rather than the global optimal weight of 8.

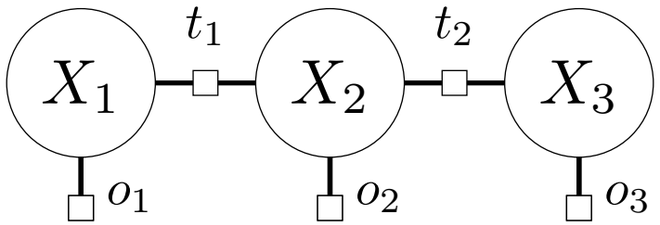
Convergence properties

- Weight(x) increases or stays the same each iteration
- Converges in a finite number of iterations
- Can get stuck in **local optima**
- Not guaranteed to find optimal assignment!



- Note that each step of ICM cannot decrease the weight because we can always stick with the old assignment.
- ICM terminates when we stop increasing the weight, which will happen eventually since there are a finite number of assignments and therefore possible weights we can increase to.
- However, ICM can get stuck in local optima, where there is a assignment with larger weight elsewhere, but no one-variable change increases the weight.
- Connection: this hill-climbing is called coordinate-wise ascent. We already saw an instance of coordinate-wise ascent in the K-means algorithm which would alternate between fixing the centroids and optimizing the object with respect to the cluster assignments, and fixing the cluster assignments and optimizing the centroids. Recall that K-means also suffered from local optima issues.
- There are two ways to mitigate local optima. One is to change multiple variables at once. Another is to inject randomness, which we'll see later with Gibbs sampling.

Summary



Algorithm	Strategy	Optimality	Time complexity
Backtracking search	extend partial assignments	exact	exponential
Beam search	extend partial assignments	approximate	linear
Local search (ICM)	modify complete assignments	approximate	linear

- This concludes our presentation of a local search algorithm, Iterated Conditional Modes (ICM).
- Let us summarize all the search algorithms for finding maximum weight assignment CSPs that we have encountered.
- Backtracking search starts with an empty assignment and incrementally build up partial assignments. It produces exact (optimal) solutions and requires exponential time (although heuristics such as dynamic ordering and AC-3 help).
- Beam search also extends partial assignments. It takes linear time in the number of variables, but yields approximate solutions.
- In this lecture, we've considered an alternative strategy, local search, which works directly with complete assignments and tries to improve them one variable at a time. If we always choose the value that maximizes the weight, we get ICM, which has the same characteristics as beam search: approximate but fast.

Roadmap

Beam search (c4.1.3)

Local search

Gibbs sampling (c14.5.2)

Cond. Independence (c14.2.2)

Gibbs sampling

Algorithm: Gibbs sampling

Initialize x to a random complete assignment

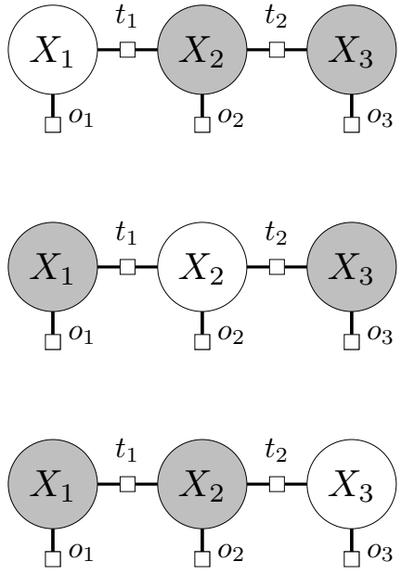
Loop through $i = 1, \ldots, n$ until convergence:

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

(X_{-i} denotes all variables except X_i)

Increment $\text{count}_i(x_i)$

Estimate $\hat{\mathbb{P}}(X_i = x_i) = \frac{\text{count}_i(x_i)}{\sum_v \text{count}_i(v)}$



Example: sampling one variable

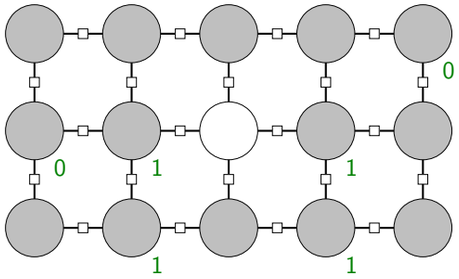
$\text{Weight}(x \cup \{X_2 : 0\}) = 1$	prob. 0.2
$\text{Weight}(x \cup \{X_2 : 1\}) = 2$	prob. 0.4
$\text{Weight}(x \cup \{X_2 : 2\}) = 2$	prob. 0.4

[demo]

- Now we present Gibbs sampling, a simple algorithm for approximately computing marginal probabilities. The algorithm follows the template of local search, where we change one variable at a time, but unlike Iterated Conditional Modes (ICM), Gibbs sampling is a randomized algorithm.
- Gibbs sampling proceeds by going through each variable X_i, considering all the possible assignments of X_i with some $v \in \text{Domain}_i$, and setting $X_i = v$ with probability equal to the conditional probability of $X_i = v$ given everything else.
- To perform this step, we can rewrite this expression using laws of probability: $\mathbb{P}(X_i = v \mid X_{-i} = x_{-i}) = \frac{\text{Weight}(x \cup \{X_i : v\})}{Z \mathbb{P}(X_{-i} = x_{-i})}$, where the denominator is a new normalization constant. We don't need to compute it directly. Instead, we first compute the weight of $x \cup \{X_i : v\}$ for each v, and then normalize to get a distribution. Finally we sample a v according to that distribution.
- Along the way, for each variable X_i that we're interested in tracking, we keep a counter $\text{count}_i(v)$ of how many times we've seen $X_i = v$. These counts can be normalized at any time to produce an estimate $\hat{\mathbb{P}}(X_i = x_i)$ of the marginal probability.

Application: image denoising

Example: image denoising



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

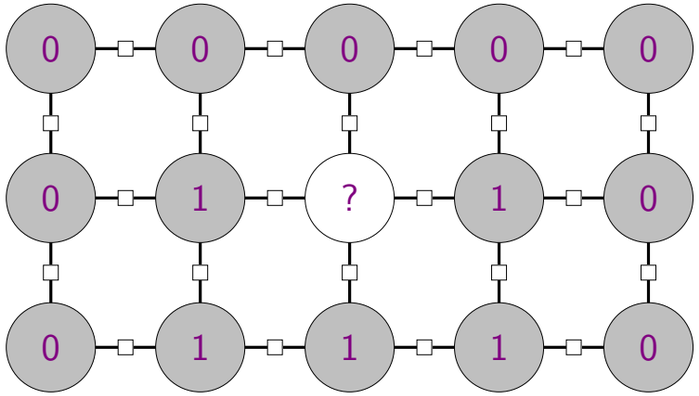
$o_i(x_i) = [x_i =$ observed value at $i]$

- Neighboring pixels more likely to be same than different

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

- Let's apply Gibbs sampling to the image denoising application.
- Recall that we have a grid of pixels, a subset of which are observed, and we wish to fill in the remaining pixels.
- The unknown pixels are represented by a variable X_i for each pixel i. We have observation factors can constrain the observed pixels, and transition factors that encourage neighboring pixels to agree.

Gibbs sampling for image denoising



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

Scan through image and update each pixel given rest:

v	weight	$\mathbb{P}(X_i = v \mid X_{-i} = x_{-i})$
0	$2 \cdot 1 \cdot 1 \cdot 1$	0.2
1	$1 \cdot 2 \cdot 2 \cdot 2$	0.8

- Let us compute the Gibbs sampling update. We go through each pixel X_i and try to update its value.
- For the given example, we consider both values 0 and 1, and multiply exactly the transition factors that depend on that value. Assume there are no observation factors here.
- The factor returns 2 if the pixel values agree and 1 if they disagree.
- We then normalize the weights to form a distribution and then sample v.
- Intuitively, the neighbors are all trying to pull $X_{(3,2)}$ towards their values, and 0.8 reflects the fact that the pull towards 1 is stronger.

Image denoising demo

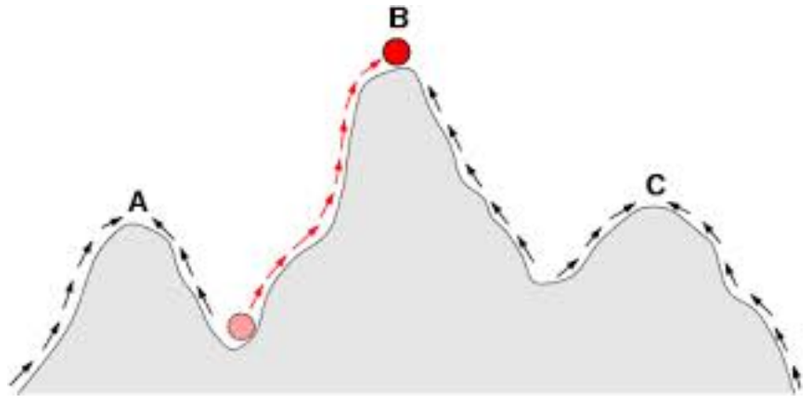
[see web version]

- Let's actually play around with Gibbs sampling for image denoising in the browser.
- 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 assignments produced or the marginals estimated from the particles (this gives you a smoother probability estimate of what the pixel values are).
- If you decrease `missingFrac` to 0.3, the problem becomes easier, and the reconstruction looks pretty good.
- If you set `coherenceFactor` to 10, then there will be coupling between neighboring variables, and you'll see sharper lines, although the reconstruction is not perfect.
- If you set `icm` to true, we will use local search rather than Gibbs sampling, which produces very bad solutions.

Search versus sampling

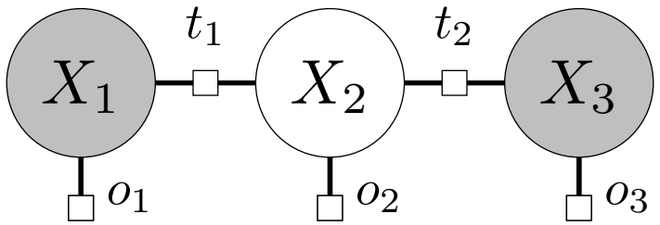
Iterated Conditional Modes	Gibbs sampling
maximum weight assignment	marginal probabilities
choose best value	sample a value
converges to local optimum	marginals converge to correct answer*

*under technical conditions (sufficient condition: all weights positive), but could take exponential time



- It is instructive to compare Gibbs sampling with its cousin, Iterated Conditional Modes (ICM). Both iteratively go through the variables and tries to update each one of them holding the others fixed.
- Recall that the goals are different: ICM tries to find the maximum weight assignment while Gibbs sampling is trying to compute marginal probabilities.
- Accordingly, ICM will choose the value for a variable X_i with the highest weight, whereas Gibbs sampling will use the weights to form a distribution to sample from.
- ICM converges to local optimum, an assignment that can't be improved on. Note that Gibbs sampling is stochastic so in some sense never converges. However, the estimates of hte marginal probabilities do in fact converge under some technical assumptions. The simplest sufficient condition if all weights are positive, but it also suffices that the probability of Gibbs sampling going between any two assignments is positive. A major caveat is that the time it takes to converge can be exponential in the number of variables.
- 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.

Summary



- Objective: compute marginal probabilities $\mathbb{P}(X_i = x_i)$
- Gibbs sampling: sample one variable at a time, count visitations
- More generally: Markov chain Monte Carlo (MCMC) powerful toolkit of randomized procedures

- In summary, we are trying to compute the marginal probabilities of a Markov network.
- Gibbs sampling allows us to do this by exploring all the assignments randomly, but very carefully controlled probabilities, so that the visitation frequencies of various values converge to the right answer.
- Gibbs sampling is part of a beautiful and rich set of tools for using randomness to do inference on Markov networks, which I encourage you to check out.

Roadmap

Beam search (c4.1.3)

Local search

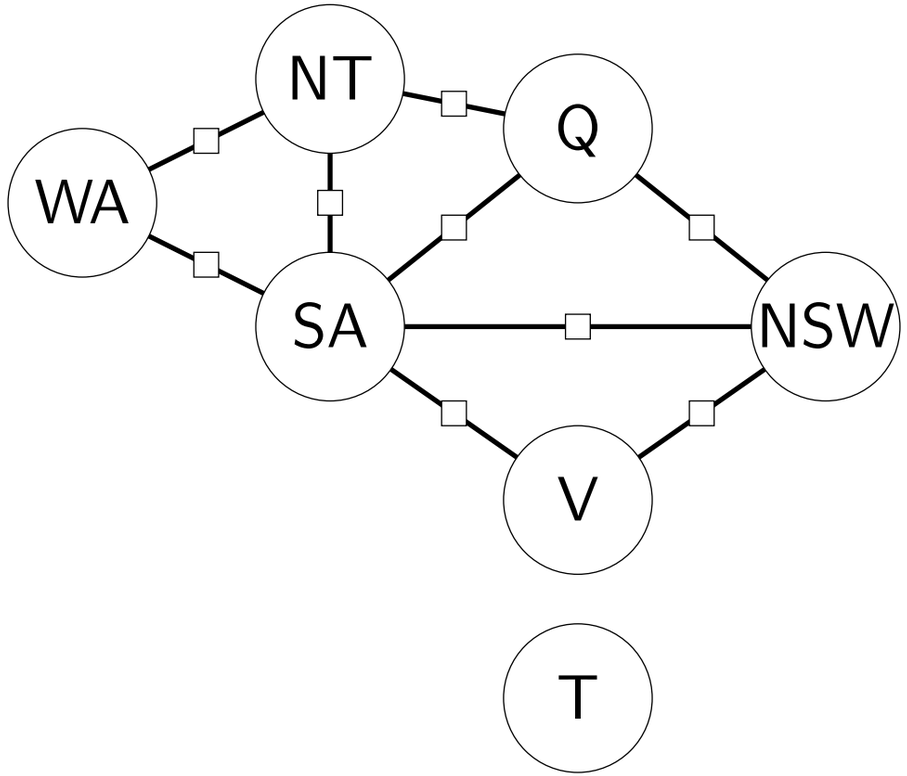
Gibbs sampling (c14.5.2)

Cond. Independence (c14.2.2)

Motivation

Key idea: graph

Leverage graph properties to derive efficient algorithms which are exact.

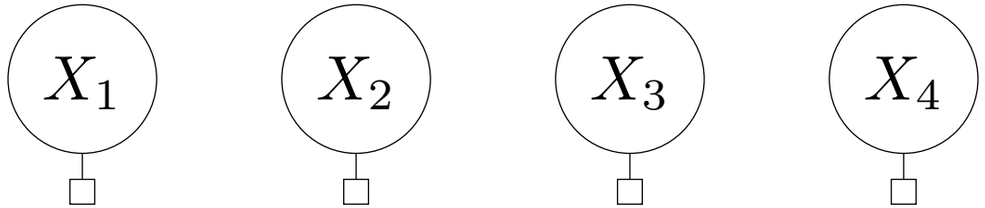


- The goal is to take advantage of the fact that we have a factor **graph**. We will see how exploiting the graph properties can lead us to more efficient algorithms as well as a deeper understanding of the structure of our problem.

Motivation

Backtracking search:

exponential time in number of variables n

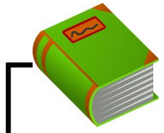


Efficient algorithm:

maximize each variable separately

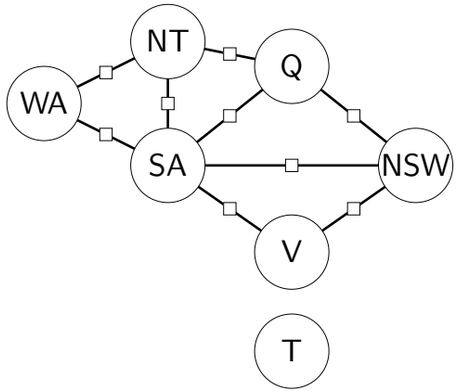
- Recall that backtracking search takes time exponential in the number of variables n. While various heuristics can have dramatic speedups in practice, it is not clear how to characterize those improvements rigorously.
- As a motivating example, consider a fully disconnected factor graph. (Imagine n people trying to vote red or blue, but they don't talk to each other.) It's clear that to get the maximum weight assignment, we can just choose the value of each variable that maximizes its own unary factor without worrying about other variables.

Independence



Definition: independence

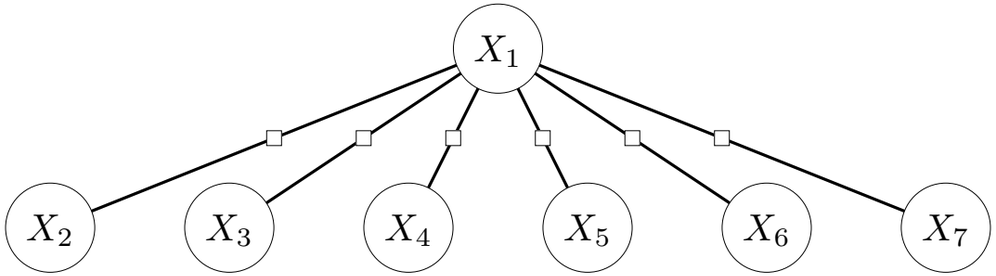
- Let A and B be a partitioning of variables X.
- We say A and B are **independent** if there are no edges between A and B.
- In symbols: $A \perp\!\!\!\perp B$.



$\{\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}\}$ and $\{\text{T}\}$ are independent.

- Let us formalize this intuition with the notion of **independence**. It turns out that this notion of independence is deeply related to the notion of independence in probability, as we will see in due time.
- Note that we are defining independence purely in terms of the graph structure, which will be important later once we start operating on the graph using two transformations: conditioning and elimination.

Non-independence

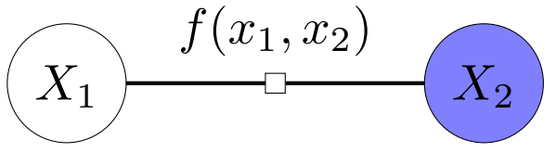


No variables are independent of each other, but feels close...

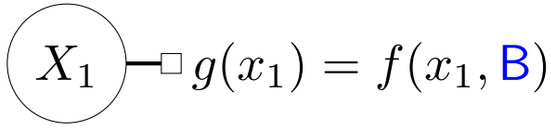
- When all the variables are independent, finding the maximum weight assignment is easily solvable in time linear in n, the number of variables. However, this is not a very interesting factor graph, because the whole point of a factor graph is to model dependencies (preferences and constraints) between variables.
- Consider the tree-structured factor graph, which corresponds to $n-1$ people talking only through a leader. Nothing is independent here, but intuitively, this graph should be pretty close to independent.

Conditioning

Goal: try to disconnect the graph



x_1	x_2	$f(x_1, x_2)$
R	R	1
R	B	7
B	R	3
B	B	2



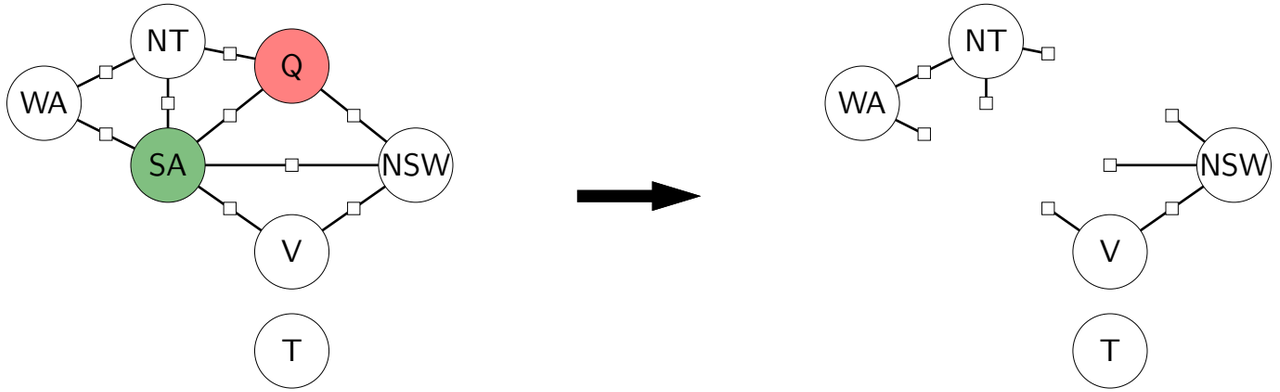
x_1	$g(x_1)$
R	7
B	2

Condition on $X_2 = \text{B}$: remove X_2, f and add g

Conditioning: example

Example: map coloring

Condition on Q = R and SA = G.

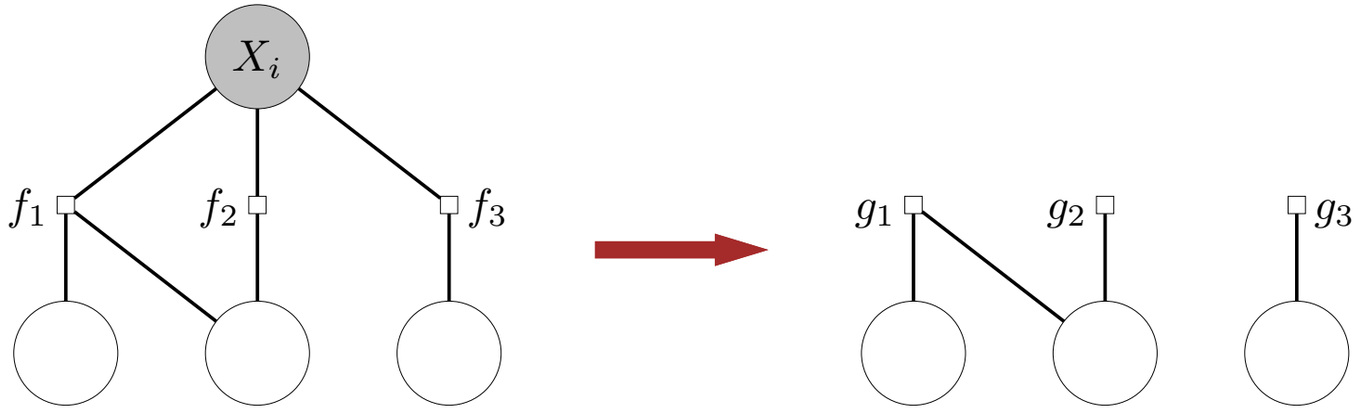


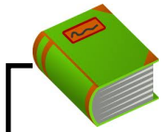
New factors:

[NT ≠ R]
[NSW ≠ R]
[WA ≠ G]
[NT ≠ G]
[NSW ≠ G]
[V ≠ G]

Conditioning: general

Graphically: remove edges from X_i to dependent factors



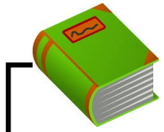


Definition: conditioning

- To **condition** on a variable $X_i = v$, consider all factors $f_1, \ldots, f_k$ that depend on X_i.
- Remove X_i and $f_1, \ldots, f_k$.
- Add $g_j(x) = f_j(x \cup \{X_i : v\})$ for $j = 1, \ldots, k$.

- In general, factor graphs are not going to have many partitions which are independent (we got lucky with Tasmania, Australia). But perhaps we can transform the graph to make variables independent. This is the idea of **conditioning**: when we condition on a variable $X_i = v$, this is simply saying that we're just going to clamp the value of X_i to v.
- We can understand conditioning in terms of a graph transformation. For each factor f_j that depends on X_i, we create a new factor g_j. The new factor depends on the scope of f_j excluding X_i; when called on x, it just invokes f_j with $x \cup \{X_i : v\}$. Think of g_j as a partial evaluation of f_j in functional programming. The transformed factor graph will have each g_j in place of the f_j and also not have X_i.

Conditional independence



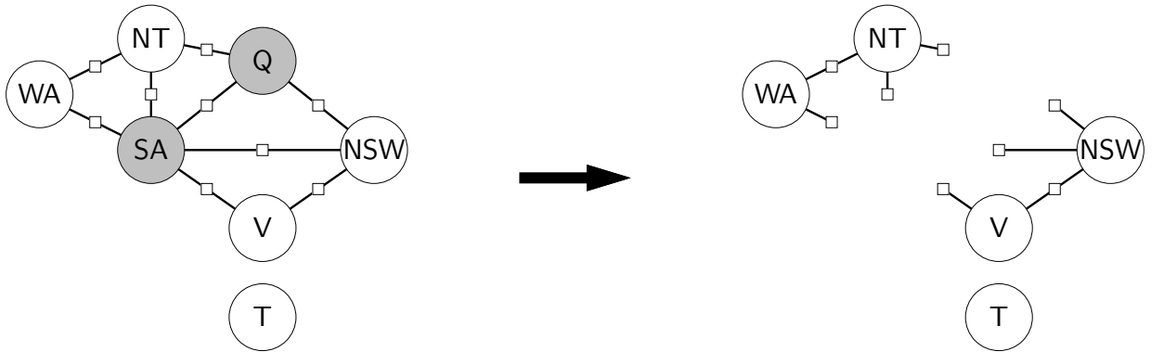
Definition: conditional independence

- Let A, B, C be a partitioning of the variables.
- We say A and B are **conditionally independent** given C if conditioning on C produces a graph in which A and B are independent.
- In symbols: $A \perp\!\!\!\perp B \mid C$.

Equivalently: every path from A to B goes through C.

Conditional independence

Example: map coloring



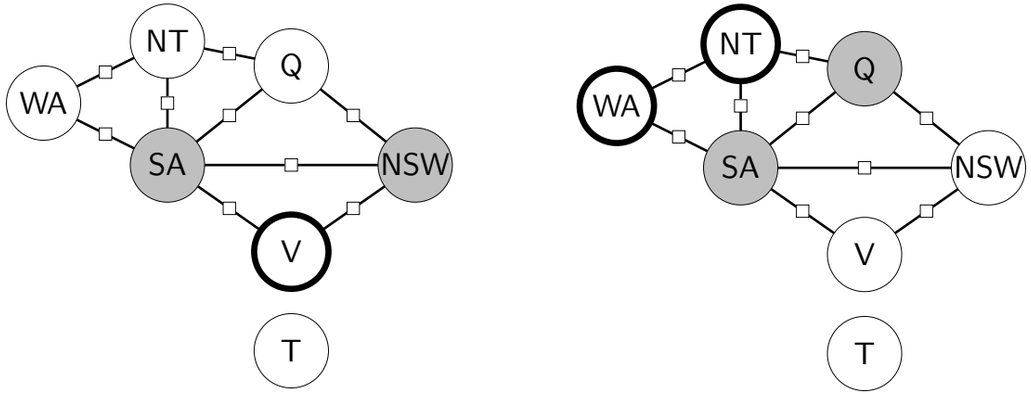
Conditional independence assertion:

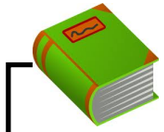
$$\{\mathrm{WA}, \mathrm{NT}\} \perp\!\!\!\perp \{\mathrm{V}, \mathrm{NSW}, \mathrm{T}\} \mid \{\mathrm{SA}, \mathrm{Q}\}$$

- With conditioning in hand, we can define **conditional independence**, perhaps the most important property in factor graphs.
- Graphically, if we can find a subset of the variables $C \subset X$ that disconnects the rest of the variables into A and B, then we say that A and B are conditionally independent given C.
- Later, we'll see how this definition relates to the definition of conditional independence in probability.

Markov blanket

How can we separate an arbitrary set of nodes from everything else?



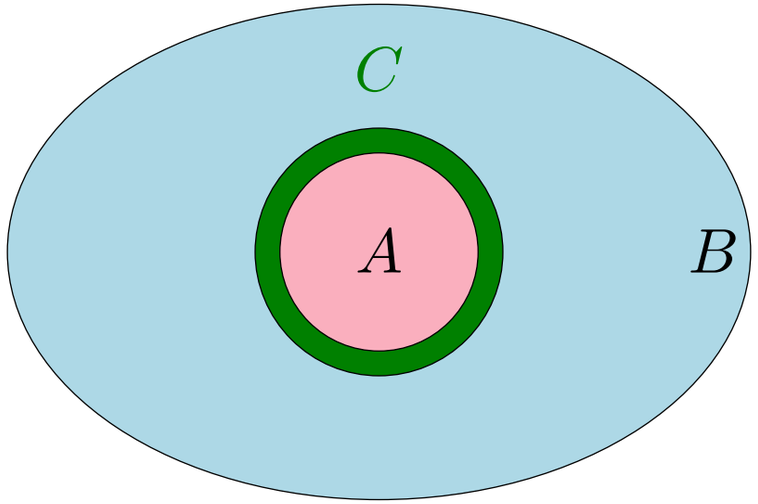


Definition: Markov blanket

Let $A \subseteq X$ be a subset of variables.

Define MarkovBlanket(A) be the neighbors of A that are not in A.

Markov blanket



Proposition: conditional independence

Let $C = \text{MarkovBlanket}(A)$.

Let B be $X \backslash (A \cup C)$.

Then $A \perp\!\!\!\perp B \mid C$.

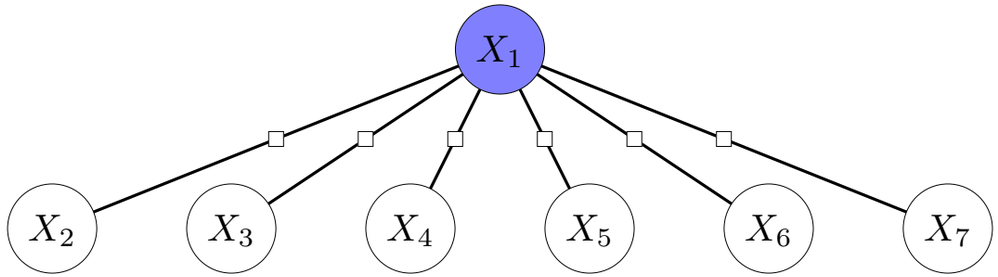
- Suppose we wanted to disconnect a subset of variables $A \subset X$ from the rest of the graph. What is the smallest set of variables C that we need to condition on to make A and the rest of the graph ($B = X \backslash (A \cup C)$) conditionally independent.
- It's intuitive that the answer is simply all the neighbors of A (those that share a common factor) which are not in A. This concept is useful enough that it has a special name: **Markov blanket**.
- Intuitively, the smaller the Markov blanket, the easier the factor graph is to deal with.

Using conditional independence

For each value $v =$ R, G, B:

Condition on $X_1 = v$.

Find the maximum weight assignment (easy).



R 3

G 6

B 1

maximum weight is 6

- Now that we understand conditional independence, how is it useful?
- First, this formalizes the fact that if someone tells you the value of a variable, you can condition on that variable, thus potentially breaking down the problem into simpler pieces.
- If we are not told the value of a variable, we can simply try to condition on all possible values of that variable, and solve the remaining problem using any method. If conditioning breaks up the factor graph into small pieces, then solving the problem becomes easier.
- In this example, conditioning on $X_1 = v$ results in a fully disconnected graph, the maximum weight assignment for which can be computed in time linear in the number of variables.

Summary

Independence: when sets of variables A and B are disconnected; can solve separately.

Conditioning: assign variable to value, replaces binary factors with unary factors

Conditional independence: when C blocks paths between A and B

Markov blanket: what to condition on to make A conditionally independent of the rest.

- **Independence** is the key property that allows us to solve subproblems in parallel. It is worth noting that the savings is huge — exponential, not linear. Suppose the factor graph has two disconnected variables, each taking on m values. Then backtracking search would take m^2 time, whereas solving each subproblem separately would take $2m$ time.
- However, the factor graph isn't always disconnected (which would be uninteresting). In these cases, we can **condition** on particular values of a variable. Doing so potentially disconnects the factor graph into pieces, which can be again solved in parallel.
- Factor graphs are interesting because every variable can still influence every other variable, but finding the maximum weight assignment is efficient if there are small bottlenecks that we can condition on.