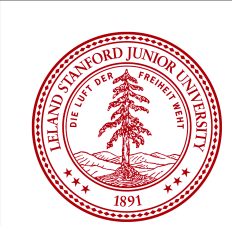
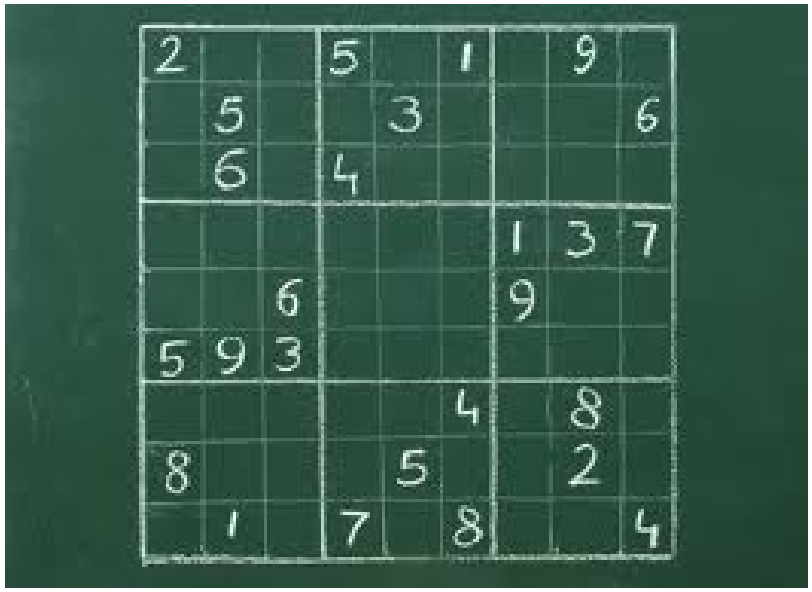
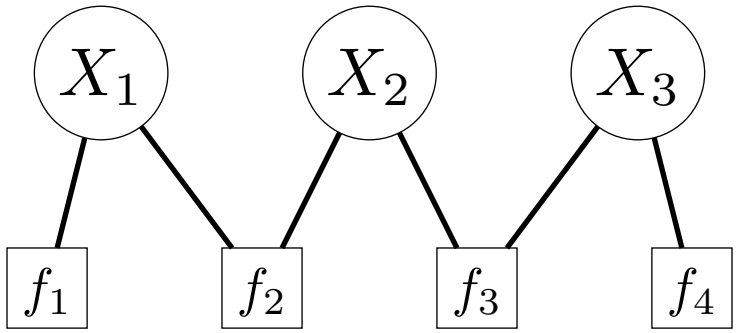
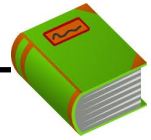


Lecture 12: CSPs II



Review: definition





Definition: factor graph

Variables:

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

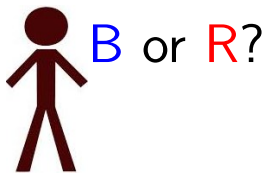
Factors:

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

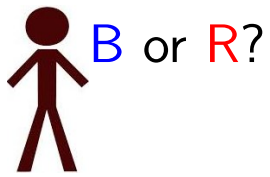
Scope of f_j: set of dependent variables

- Recall the definition of a factor graph: we have a set of variables $X_1, \ldots, X_n$ and a set of factors $f_1, \ldots, f_m$.
- Each factor f_j is a function that takes an assignment to the variables and returns a non-negative number $f_j(X)$ indicating how much that factor likes that assignment. A zero return value signifies a (hard) constraint that the assignment is to be avoided at all costs.
- Each factor f_j depends on only variables in its scope, which is usually a much smaller subset of the variables.
- Factor graphs are typically visualized graphically in a way that highlights the dependencies between variables and factors.

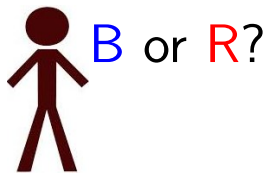
Factor graph (example)

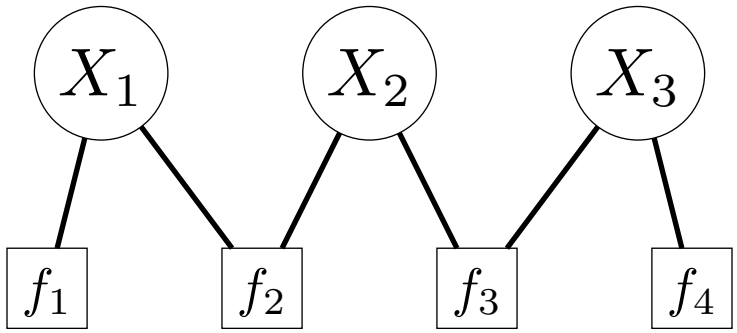


must agree



tend to agree





x_1	$f_1(x_1)$
R	0
B	1

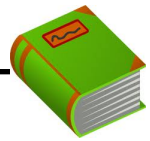
x_1	x_2	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

x_2	x_3	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

x_3	$f_4(x_3)$
R	2
B	1

$$f_2(x_1, x_2) = [x_1 = x_2] \qquad f_3(x_2, x_3) = [x_2 = x_3] + 2$$

Review: definition



Definition: assignment weight

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

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

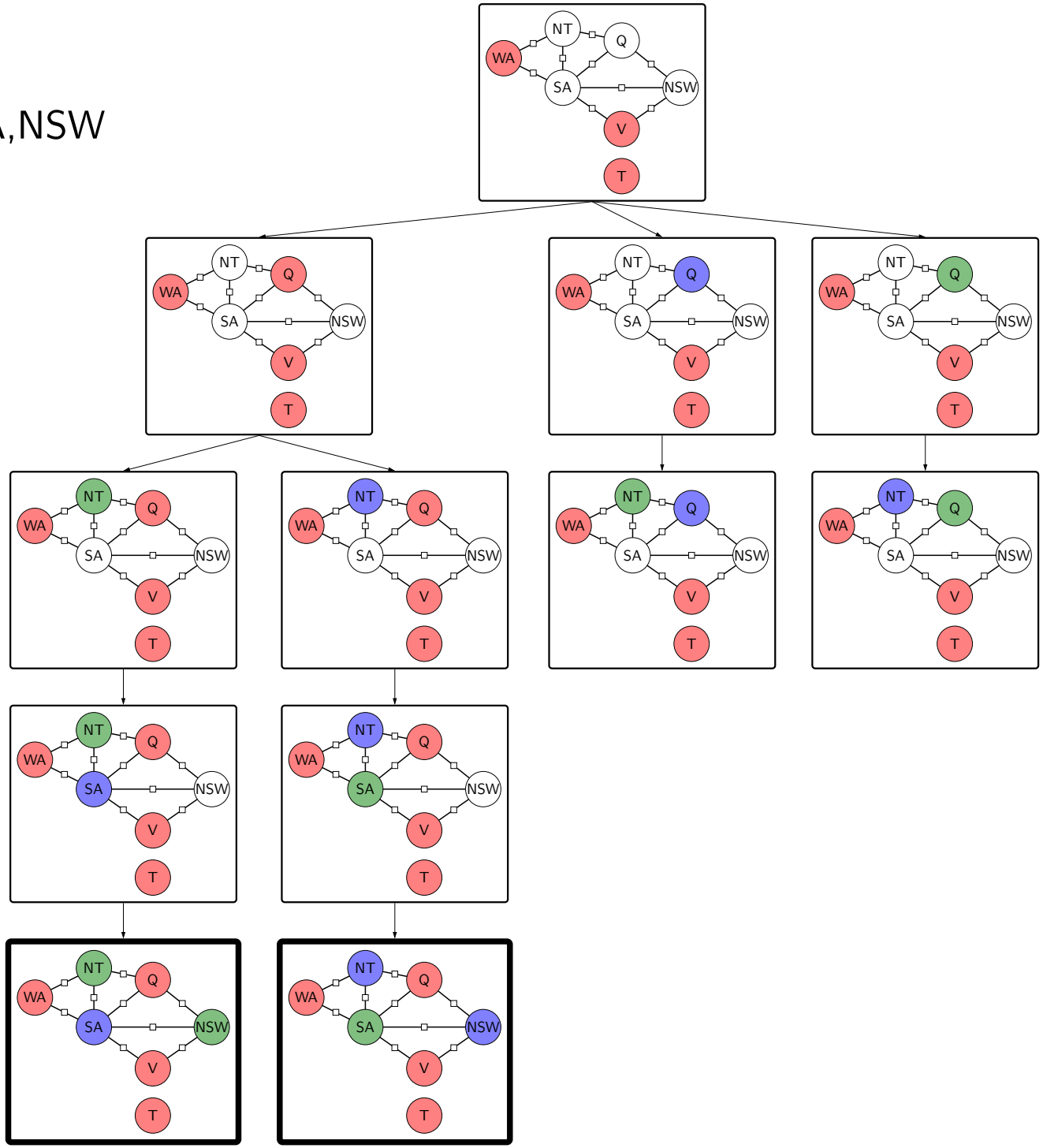
Objective: find a maximum weight assignment

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

- The weight of an assignment x is defined as the product of all the factors applied to x. Since it's a product, all factors have to unanimously like an assignment for the assignment to have high weight.
- Our objective is to find an assignment with the maximum weight (not to be confused with the weights in machine learning).

Search

WA,V,T,Q,NT,SA,NSW



Review: backtracking search

Vanilla version:

$$O(|\text{Domain}|^n) \text{ time}$$

Lookahead: forward checking, AC-3

$$O(|\text{Domain}|^n) \text{ time}$$

Dynamic ordering: most constrained variable, least constrained value

$$O(|\text{Domain}|^n) \text{ time}$$

Note: these pruning techniques useful only for constraints

- Last time, we talked about backtracking search as a way to find maximum weight assignments. In the worst case, without any further assumptions on the factor graph, this requires exponential time. We can be more clever and employ lookahead and dynamic ordering, which in some cases can dramatically improve running time, but in the worst case, it's still exponential.
- Also, these heuristics are only helpful when there are hard constraints, which allow us to prune away values in the domains of variables which definitely are not compatible with the current assignment.
- What if all the factors were strictly positive? None of the pruning techniques we encountered would be useful at all. Thus we need new techniques.

Example: object tracking

Setup: sensors (e.g., camera) provide noisy information about location of an object (e.g., video frames)

Goal: infer object's true location

- As motivation, we will consider object (e.g., person) tracking, an important task in computer vision.
- 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?

Modeling object tracking

Problem: object tracking

Noisy sensors report positions: $0, 2, 2$.

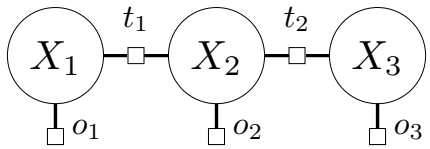
Objects don't move very fast.

What path did the object take?

[whiteboard: trajectories over time]

Person tracking solution

Factor graph (chain-structured):



- Variables X_i: location 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

[demo: create factor graph]

- 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 locations 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 locations X_i are compatible with the noisy information. Note that these numbers are just numbers, not necessarily probabilities; later we'll see how probabilities fit in to factor graphs.
- Having modeled the problem as a factor graph, we can now ask for the maximum weight assignment for that factor graph, which would give 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 we smooth out the trajectory, assuming that the first sensor reading was inaccurate.
- Next we will develop algorithms for finding a maximum weight assignment in a factor graph. These algorithms will be overkill for solving this simple tracking problem, but it will nonetheless be useful for illustrative purposes.

Roadmap

Beam search

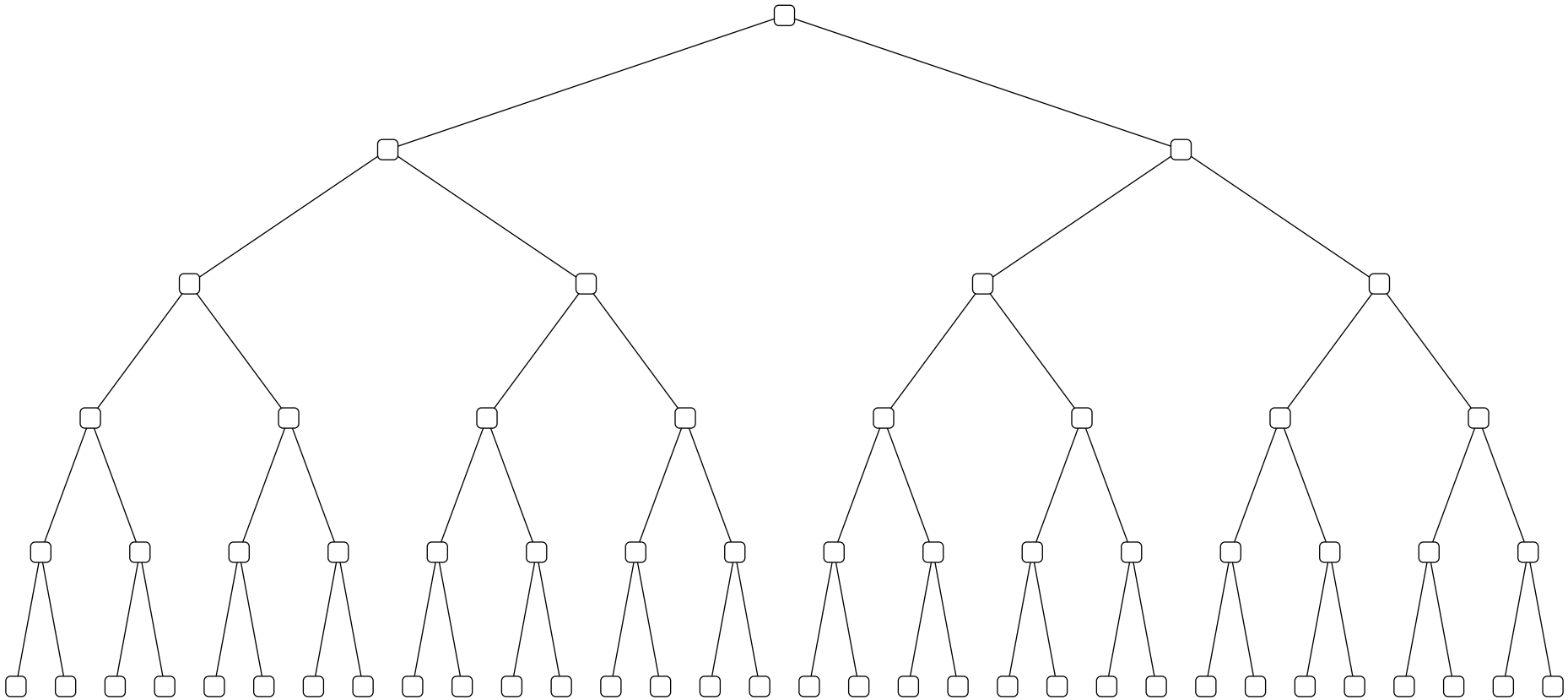
Local search

Conditioning

Elimination

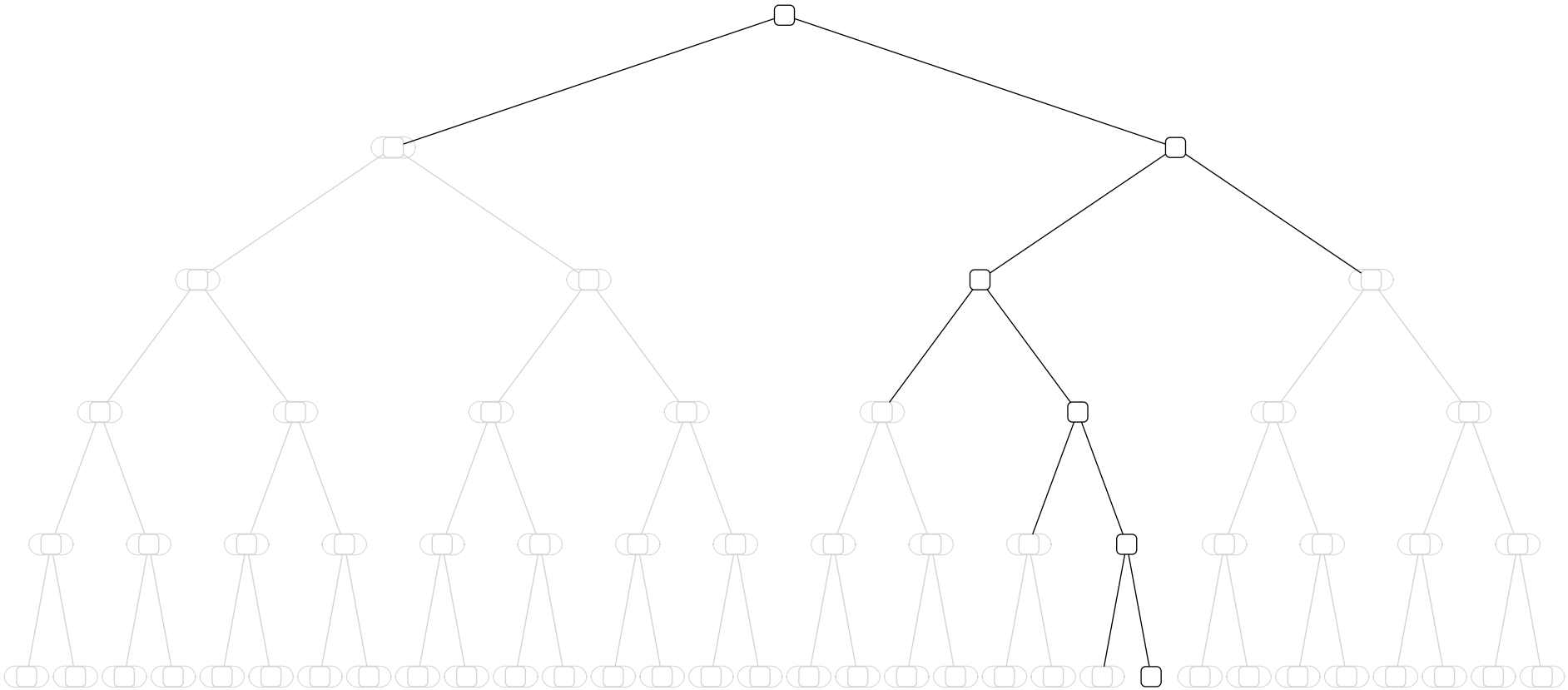
- In this lecture, we will discuss alternative ways to find maximum weight assignments efficiently without incurring the full cost of backtracking search.
- The first two methods (beam search and local search) are approximate algorithms. We give up guarantees of finding the exact maximum weight assignment, but they can still work well in practice.
- Then, we will look at the fact that we have a factor **graph**, and show that for graph structures, we can get exponentially faster algorithms, analogous to the savings we obtained by using dynamic programming in search problems. We will introduce two factor graph operations: conditioning and elimination.

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! In greedy search, we're just going to stick with our guns, marching down one thin slice of the search tree, and never looking back.

Greedy search

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

Algorithm: greedy search

Partial assignment $x \leftarrow \{\}$

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

Extend:

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

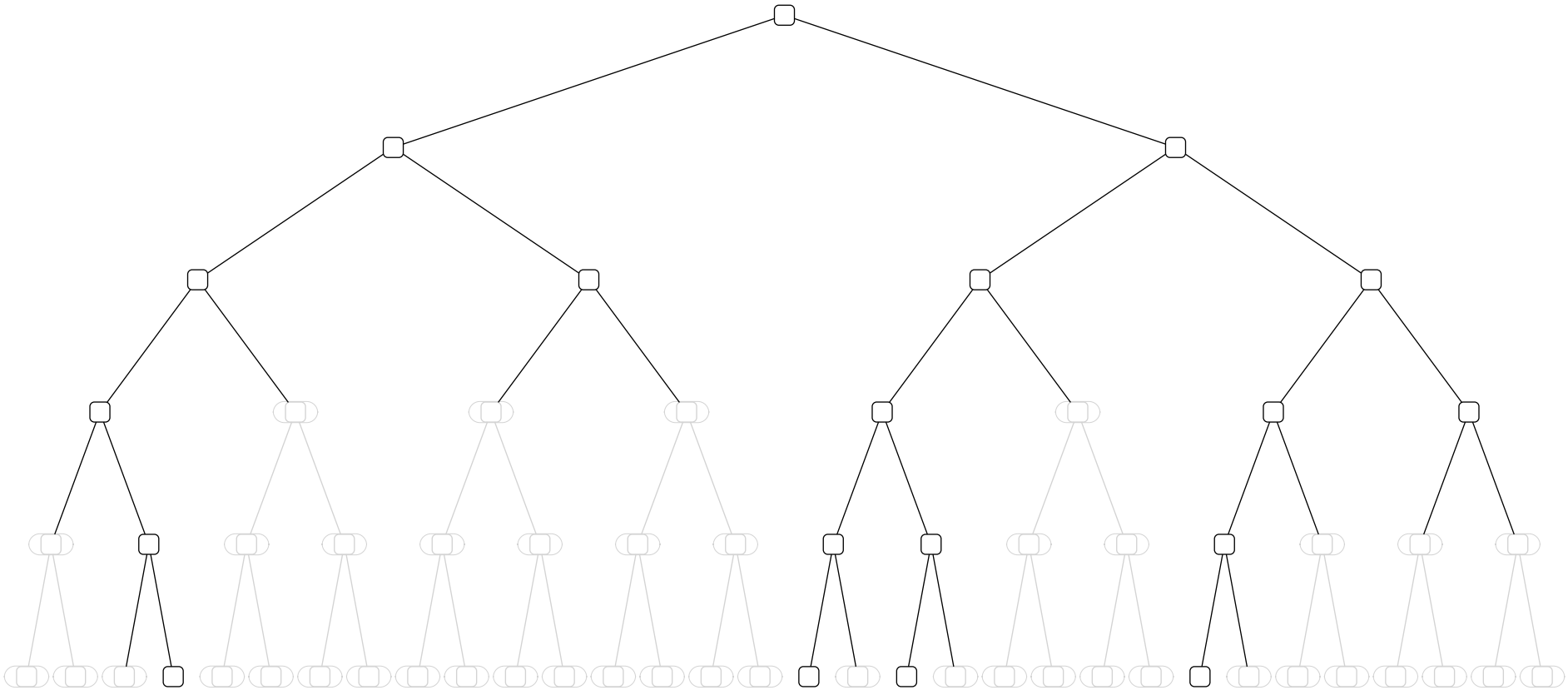
Prune:

$x \leftarrow x_v$ with highest weight

Not guaranteed to find optimal assignment!

- Specifically, we assume we have a fixed ordering of the variables. As in backtracking search, we maintain a partial assignment x and its weight, which we denote $w(x)$. 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 $K = 4$

Beam search

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

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 optimal assignment!

- 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).
- The beam search algorithm maintains a candidate list 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 sort C' by decreasing weight, and keep only the top K elements.
- Beam search also has no guarantees of finding the maximum weight assignment, but it generally works better than greedy at the cost of an increase in running time.
- In the demo, we can see that with a beam size of $K = 3$, we are able to find the globally optimal solution.

Beam search properties

- Running time: $O(n(Kb)\log(Kb))$ with branching factor $b = |\text{Domain}|$, beam size K
- Beam size K controls tradeoff between efficiency and accuracy
  - $K = 1$ is greedy ($O(nb)$ time)
  - $K = \infty$ is BFS tree search ($O(b^n)$ time)
- Analogy: backtracking search : DFS :: BFS : beam search (pruned)

- Beam search offers a nice way to tradeoff efficiency and accuracy and is used quite commonly in practice. If you want speed and don't need extremely high accuracy, use greedy ($K = 1$). The running time is $O(nb)$, since for each of the n variables, we need to consider b possible values in the domain.
- If you want high accuracy, then you need to pay by increasing K. For each of the n variables, we keep track of K candidates, which gets extended to Kb when forming C'. Sorting these Kb candidates by score requires $Kb \log(Kb)$ time.
- With large enough K (no pruning), beam search is just doing a BFS traversal rather than a DFS traversal of the search tree, which takes $O(b^n)$ time. Note that K doesn't enter in to the expression because the number of candidates is bounded by the total number, which is $O(b^n)$. Technically, we could write the running time of beam search as $O(\min\{b^n, n(Kb)\log(Kb)\})$, but for small K and large n, b^n will be much larger, so it can be ignored.
- For moderate values of K, beam search is a kind of pruned BFS, where we use the factors that we've seen so far to decide which branches of the tree are worth exploring.
- In summary, beam search takes a broader view of the search tree, allowing us to compare partial assignments in very different parts of the tree, something that backtracking search cannot do.

Roadmap

Beam search

Local search

Conditioning

Elimination

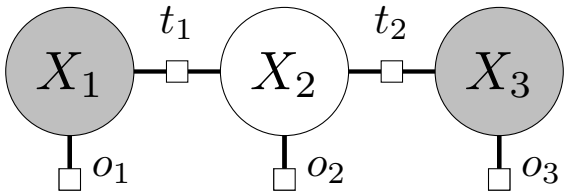
Local search

Backtracking/beam search: extend partial assignments

Local search: modify complete assignments

- So far, both backtracking and beam search 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.
- Recall that one of the motivations for moving to variable-based models is that we wanted to downplay the role of ordering. **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.

Iterated conditional modes (ICM)



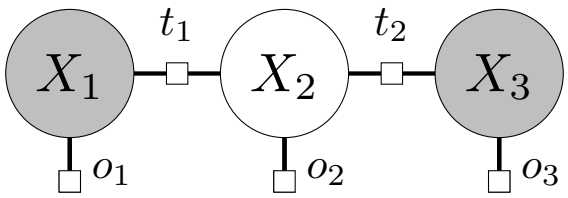
Current assignment: $(0, 0, 1)$; how to improve?

(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: $(0, 1, 1)$

- Consider a complete assignment $(0, 0, 1)$. 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 possibility, we can compute the weight, and just take the highest scoring option. This results in a new assignment $(0, 1, 1)$ with a higher weight (4 rather than 0).

Iterated conditional modes (ICM)



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.

- If we write down the weights of the various new assignments $x \cup \{X_2 : v\}$, we will notice that all the factors return the same value except the ones that depend on X_2.
- Therefore, we only need to compute the product of these relevant factors and take the maximum value. 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)

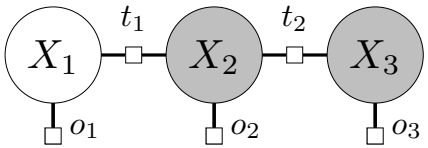
Algorithm: iterated conditional modes (ICM)

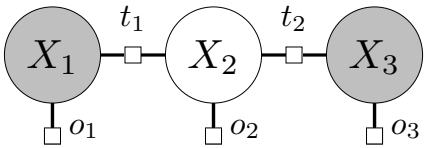
Initialize x to a random complete assignment

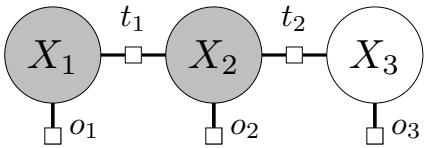
Loop through $i = 1, \ldots, n$ until convergence:

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

$x \leftarrow x_v$ with highest weight





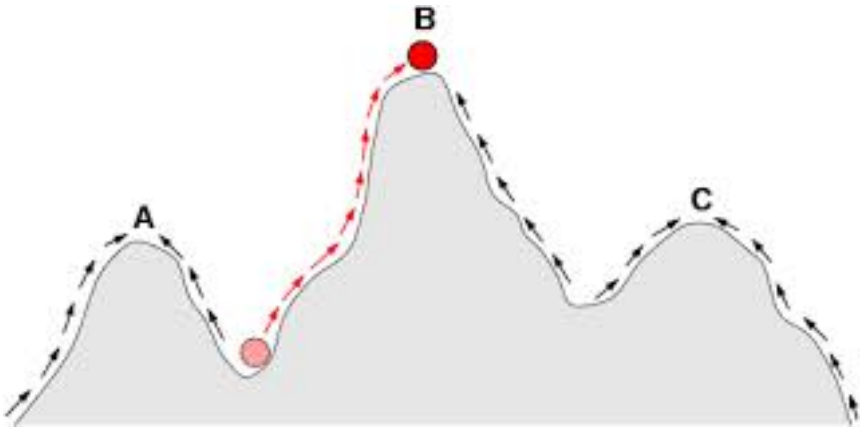


- Now we can state our first algorithm, ICM, which is the local search analogy of the greedy algorithm we described earlier. The idea is simple: we start with a random complete assignment. We repeatedly loop through all the variables X_i. On variable X_i, we consider all possible ways of re-assigning 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.

Iterated conditional modes (ICM)

[demo: `iteratedConditionalModes()`]

- 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 ICM will increase the weight of the assignments monotonically and converges, but it will get stuck in local optima, where there is a better assignment elsewhere, but all the one variable changes result in a lower weight assignment.
- 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.
- Connection: these local optima are an example of a Nash equilibrium (for collaborative games), where no unilateral change can improve utility.
- Note that in the demo, ICM gets stuck in a local optimum with weight 4 rather than the global optimum's 8.

Gibbs sampling

Sometimes, need to go downhill to go uphill...

Key idea: randomness

Sample an assignment with probability proportional to its weight.

Example: Gibbs sampling

$\text{Weight}(x \cup \{X_2 : 0\}) = 1$ 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

- In reinforcement learning, we also had a problem where if we explore by using a greedy policy (always choosing the best action according to our current estimate of the Q function), then we were doomed to get stuck. There, we used **randomness** via epsilon-greedy to get out of local optima.
- Here, we will do the same, but using a slightly more sophisticated form of randomness. The idea is **Gibbs sampling**, a method originally designed for using Markov chains to sample from a distribution over assignments. We will return to that original use later, but for now, we are going to repurpose it for the problem of finding the maximum weight assignment.

Gibbs sampling

[demo: `gibbsSampling()`]

Algorithm: Gibbs sampling

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

Choose $x \leftarrow x_v$ with probability prop. to its weight

Can escape from local optima (not always easy though)!

- The form of the algorithm is identical to ICM. The only difference is that rather than taking the assignment $x \cup \{X_i : v\}$ with the maximum weight, we choose the assignment with probability proportional to its weight.
- In this way, even if an assignment has lower weight, we won't completely rule it out, but just choose it with lower probability. Of course if an assignment has zero weight, we will choose it with probability zero (which is to say, never).
- Randomness is not a panacea and often Gibbs sampling can get ensnarled in local optima just as much as ICM. In theory, under the assumption that we could move from the initial assignment and the maximum weight assignment with non-zero probability, Gibbs sampling will move there eventually (but it could take exponential time in the worst case).
- Advanced: just as beam search is greedy search with K candidates instead of 1, we could extend ICM and Gibbs sampling to work with more candidates. This leads us to the area of particle swarm optimization, which includes genetic algorithms, which is beyond the scope of this course.

cs221.stanford.edu/q

Question

Which of the following algorithms are guaranteed to find the maximum weight assignment (select all that apply)?

- backtracking search
- greedy search
- beam search
- Iterated Conditional Modes
- Gibbs sampling

Summary so far

Algorithms for max-weight assignments in factor graphs:

(1) Extend partial assignments:

- Backtracking search: exact, exponential time
- Beam search: approximate, linear time

(2) Modify complete assignments:

- Iterated conditional modes: approximate, deterministic
- Gibbs sampling: approximate, randomized

Roadmap

Beam search

Local search

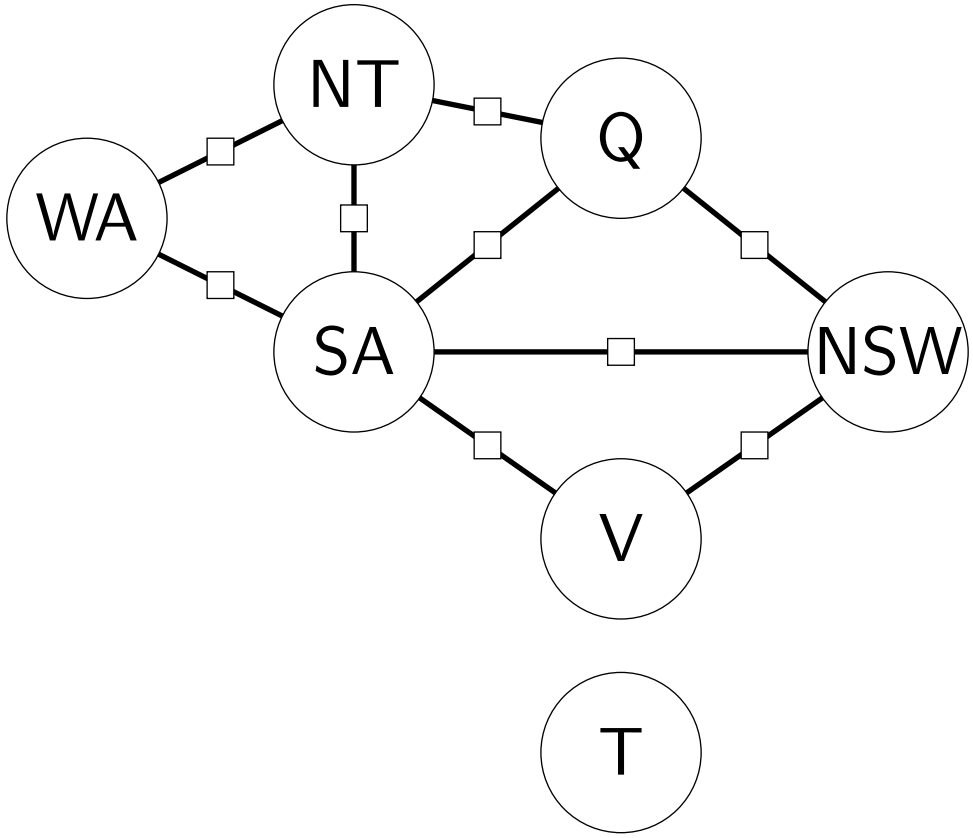
Conditioning

Elimination

Motivation

Key idea: graph

Leverage graph properties to derive efficient algorithms which are exact.

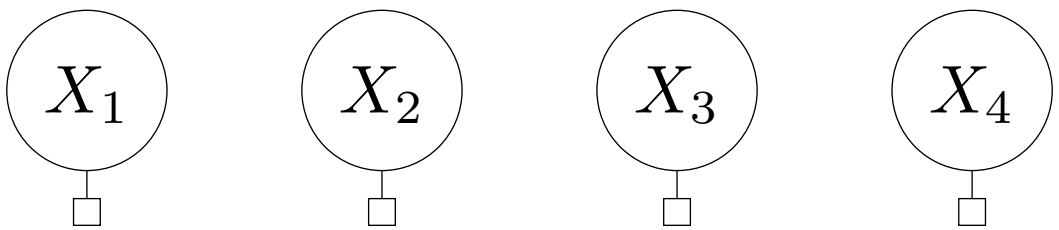


- The goal in the second part of the lecture is to take advantage of the fact that we have a factor **graph**. We will see how exploiting the graph properties can lead us to more efficient algorithms as well as a deeper understanding of the structure of our problem.

Motivation

Backtracking search:

exponential time in number of variables n

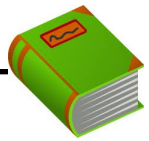


Efficient algorithm:

maximize each variable separately

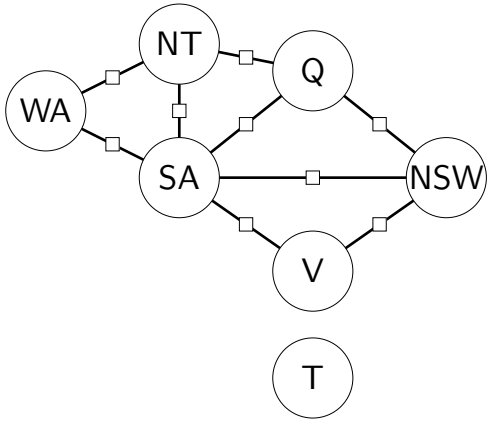
- Recall that backtracking search takes time exponential in the number of variables n. While various heuristics can have dramatic speedups in practice, it is not clear how to characterize those improvements rigorously.
- As a motivating example, consider a fully disconnected factor graph. (Imagine n 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

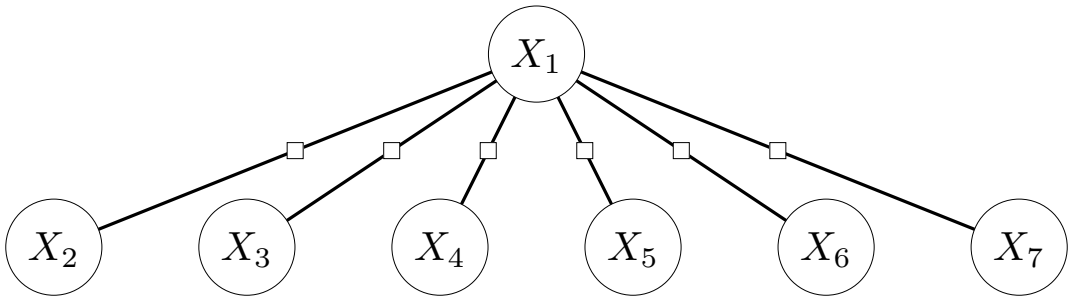
- Let A and B be a partitioning of variables X.
- We say A and B are **independent** if there are no edges between A and B.
- In symbols: $A \perp\!\!\!\perp B$.



$\{\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}\}$ 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

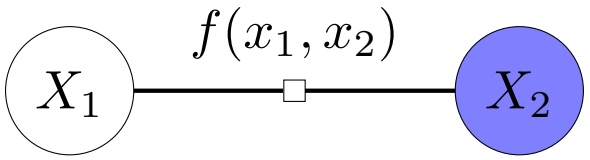


No variables are independent of each other, but feels close...

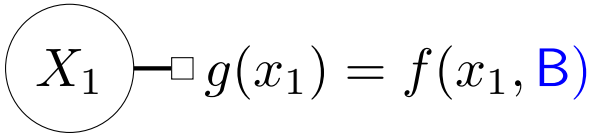
- When all the variables are independent, finding the maximum weight assignment is easily solvable in time linear in n, the number of variables. However, this is not a very interesting factor graph, because the whole point of a factor graph is to model dependencies (preferences and constraints) between 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



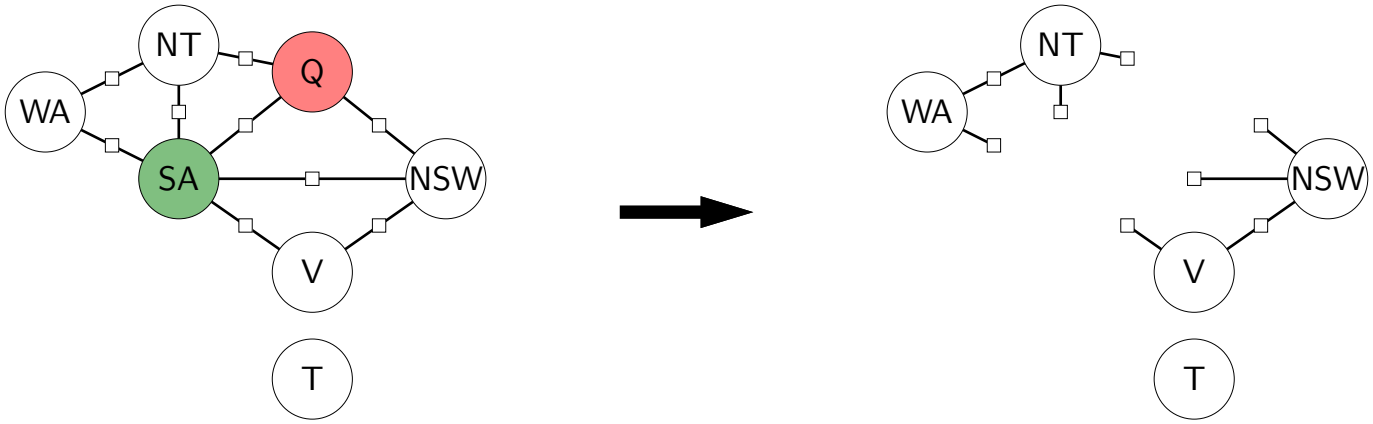
x_1	$g(x_1)$
R	7
B	2

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.



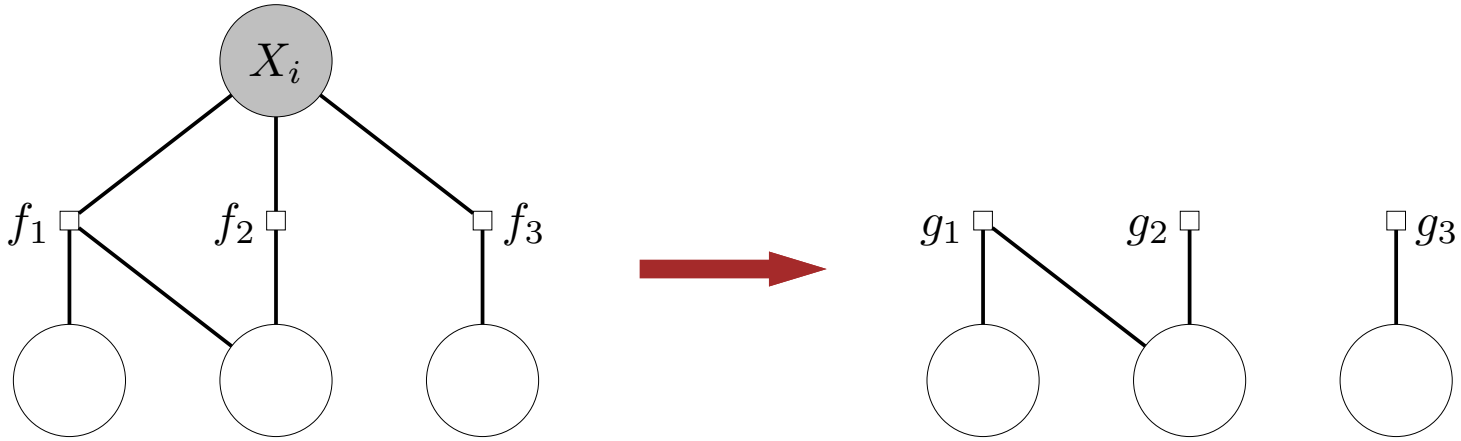
New factors:

[NT ≠ R]
[NSW ≠ R]

[WA ≠ G]
[NT ≠ G]
[NSW ≠ G]
[V ≠ G]

Conditioning: general

Graphically: remove edges from X_i to dependent factors

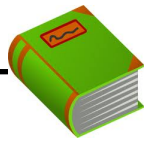


Definition: conditioning

- 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



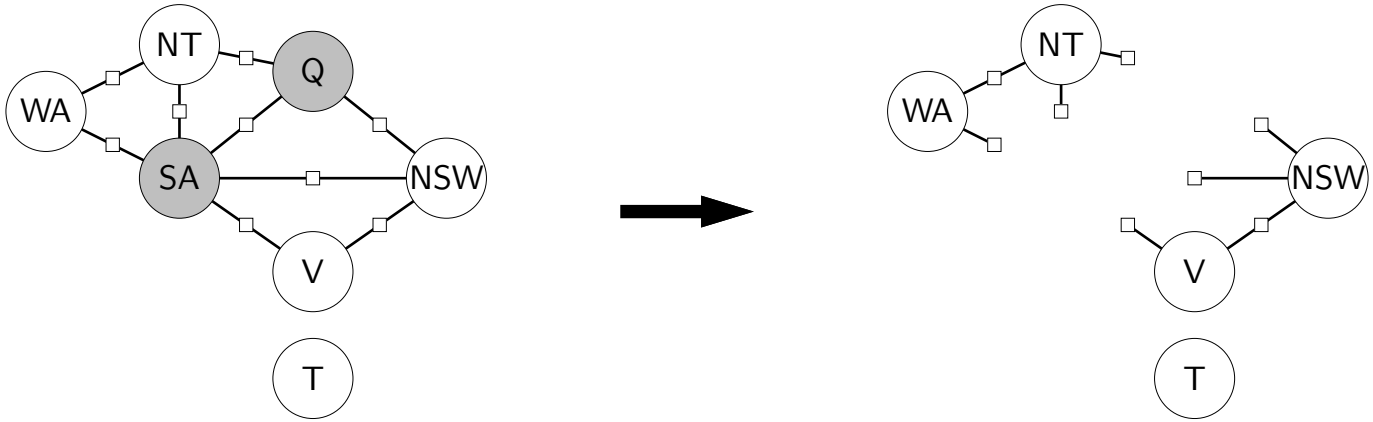
Definition: conditional independence

- 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



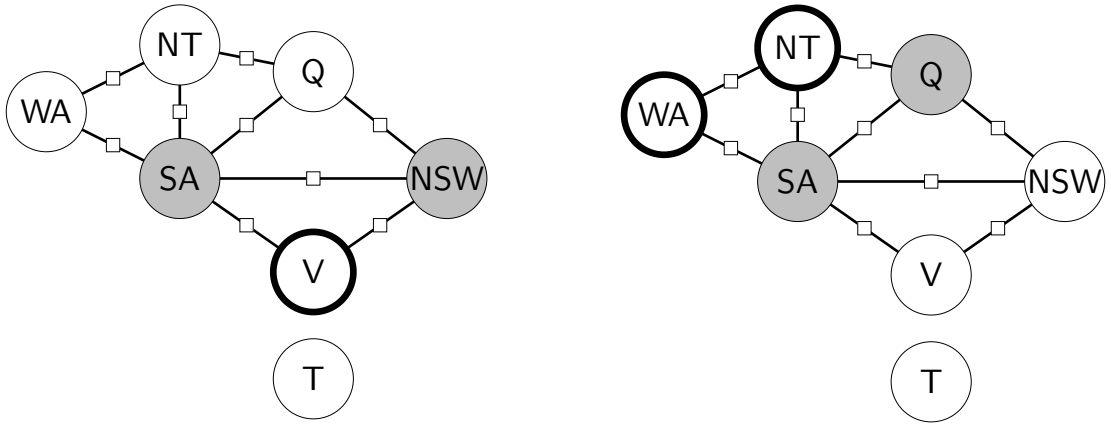
Conditional independence assertion:

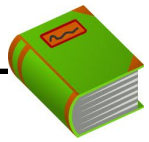
$$\{\text{WA}, \text{NT}\} \perp\!\!\!\perp \{\text{V}, \text{NSW}, \text{T}\} \mid \{\text{SA}, \text{Q}\}$$

- 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?



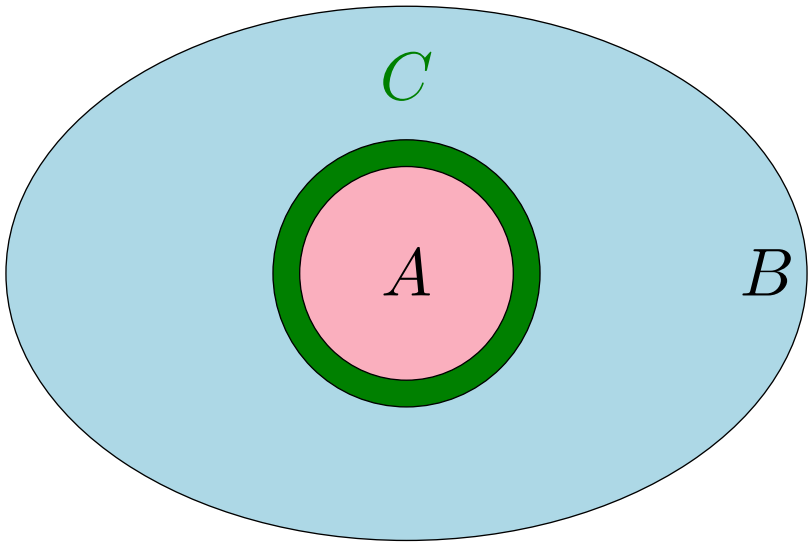


Definition: Markov blanket

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

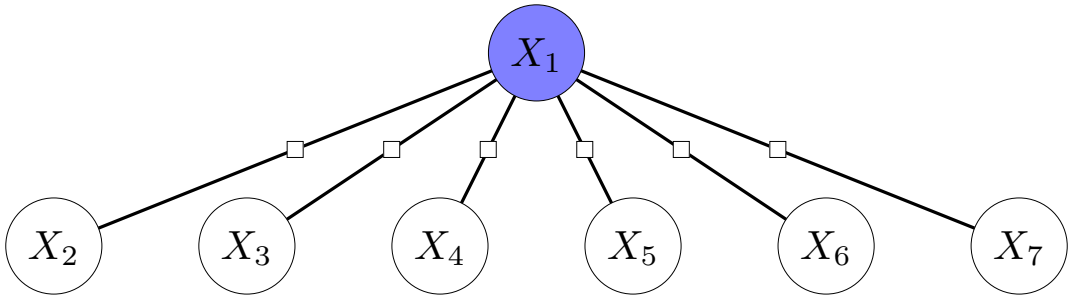
- Suppose we wanted to disconnect a subset of variables $A \subset X$ from the rest of the graph. What is the smallest set of variables C that we need to condition on to make A and the rest of the graph ($B = X \backslash (A \cup C)$) conditionally independent.
- It's intuitive that the answer is simply 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 so far

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 woule 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.

Roadmap

Beam search

Local search

Conditioning

Elimination

Conditioning versus elimination

Conditioning:

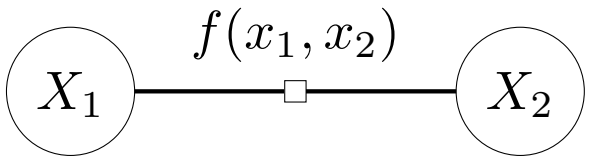
- Removes X_i from the graph
- Add factors that use fixed value of X_i

Elimination (max):

- Removes X_i from the graph
- Add factors that maximize over all values of X_i

- Now we discuss the second important factor graph transformation: **elimination**. Conditioning was great at breaking the factor graph apart but required a fixed value on which to condition. If we don't know what the value should be, we just have to try all of them.
- Elimination (more precisely, max elimination) also removes variables from the graph, but actually chooses the best value for the eliminated variable X_i. But how do we talk about the best value? The answer is that we compute the best one for all possible assignments to the Markov blanket of X_i. The result of this computation can be stored as a new factor.

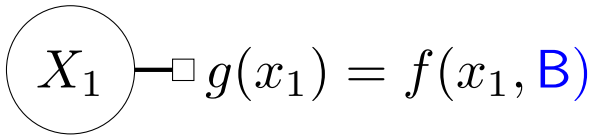
Conditioning versus elimination



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

Conditioning:

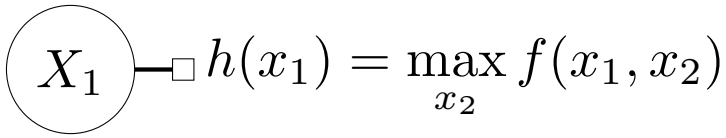
consider **one** value ($X_2 = \text{B}$)



x_1	$g(x_1)$
R	7
B	2

Elimination:

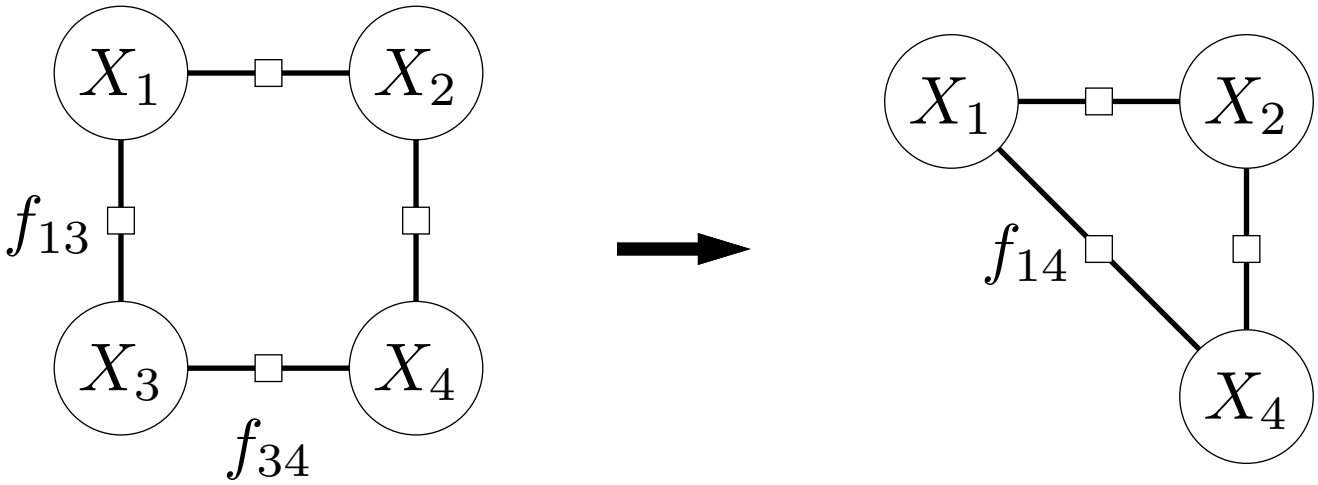
consider **all** values



x_1	$h(x_1)$
R	7
B	3

- If we eliminate X_2 in this simple example, we produce a factor graph with the same structure as what we got for conditioning (but in general, this is not the case), but a different factor.
- In conditioning, the new factor produced $g(x_1)$ was just f evaluated with $x_2 = \mathsf{B}$. In elimination, the new factor produced $h(x_1)$ is the maximum value of f over all possible values of x_2.

Elimination: example



$$f_{14}(x_1, x_4) = \max_{x_3}[f_{13}(x_1, x_3) f_{34}(x_3, x_4)]$$

(maximum weight of assignment to X_3 given X_1, X_4)

$\max_{x_3}$

x_1	x_3	$f_{13}(x_1, x_3)$
R	R	4
R	B	1
B	R	1
B	B	4

$\cdot$

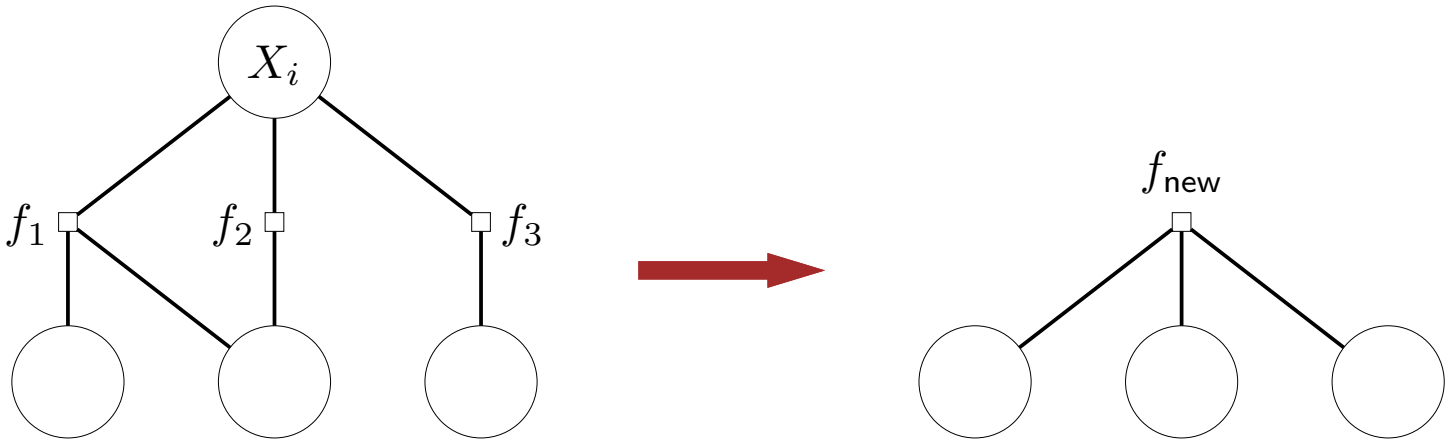
x_3	x_4	$f_{34}(x_3, x_4)$
R	R	1
R	B	2
B	R	2
B	B	1

$=$

x_1	x_4	$f_{14}(x_1, x_4)$
R	R	$\max(4 \cdot 1, 1 \cdot 2) = 4$
R	B	$\max(4 \cdot 2, 1 \cdot 1) = 8$
B	R	$\max(1 \cdot 1, 4 \cdot 2) = 8$
B	B	$\max(1 \cdot 2, 4 \cdot 1) = 4$

- Now let us look at a more complex example. Suppose we want to eliminate X_3. Now we have two factors f_{13} and f_{34} that depend on X_3.
- Again, recall that we should think of elimination as solving the maximum weight assignment problem over X_3 conditioned on the Markov blanket $\{X_1, X_4\}$.
- The result of this computation is stored in the new factor $f_{14}(x_1, x_4)$, which depends on the Markov blanket.

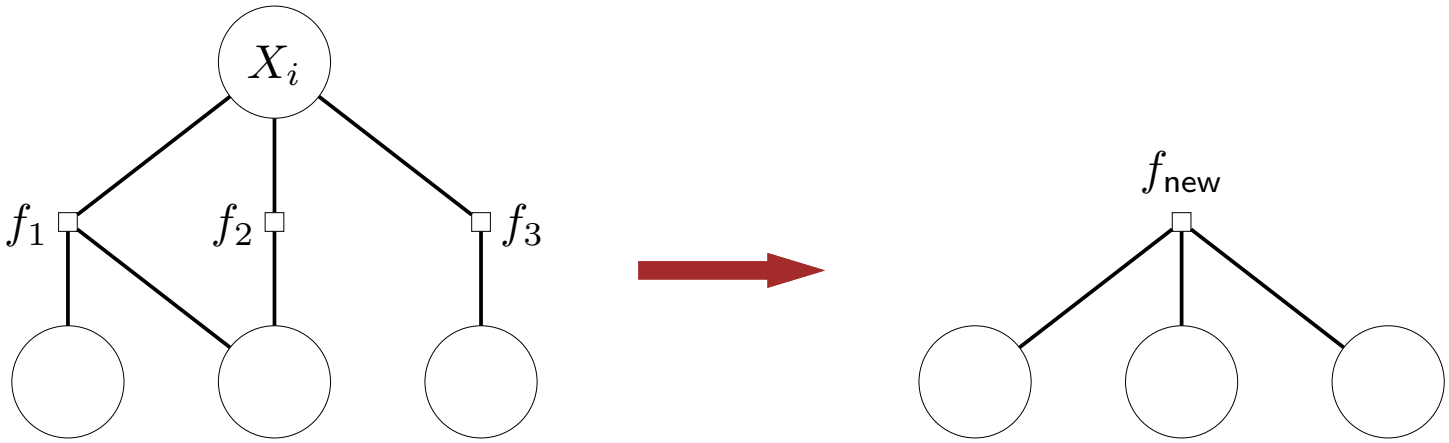
Elimination: general



Definition: elimination

- To **eliminate** a variable X_i, consider all factors $f_1, \ldots, f_k$ that depend on X_i.
- Remove X_i and $f_1, \ldots, f_k$.
- Add $f_{\text{new}}(x) = \max_{x_i} \prod_{j=1}^{k} f_j(x)$

Elimination: general



$$f_{\text{new}}(x) = \max_{x_i} \prod_{j=1}^{k} f_j(x)$$

- Solves a mini-problem over X_i conditioned on its Markov blanket!
- Scope of f_{new} is MarkovBlanket(X_i)

- In general, to eliminate a variable X_i, we look at all factors which depend on it, just like in conditioning. We then remove those factors $f_1, \ldots, f_k$ and X_i, just as in conditioning. Where elimination differs is that it produces a single factor which depends on the Markov blanket rather than a new factor for each f_j.
- Note that eliminating a variable X_i is much more costly than conditioning, and will produce a new factor which can have quite high arity (if X_i depends on many other variables).
- But the good news is that once a variable X_i is eliminated, we don't have to revisit it again. If we have an assignment to the Markov blanket of X_i, then the new factor gives us the weight of the best assignment to X_i, which is stored in the new factor.
- If for every new factor f_{new}, we store for each input, not only the value of the max, but also the argmax, then we can quickly recover the best assignment to X_i.

cs221.stanford.edu/q

Question

Suppose we have a star-shaped factor graph. Which of the following is true (select all that apply)?

Conditioning on the hub produces unary factors.

Eliminating the hub produces unary factors.

Variable elimination algorithm

Algorithm: variable elimination

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

    Eliminate X_i (produces new factor $f_{\text{new},i}$).

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

    Set X_i to the maximizing value in $f_{\text{new},i}$.

[demo: `query(''); maxVariableElimination()`]

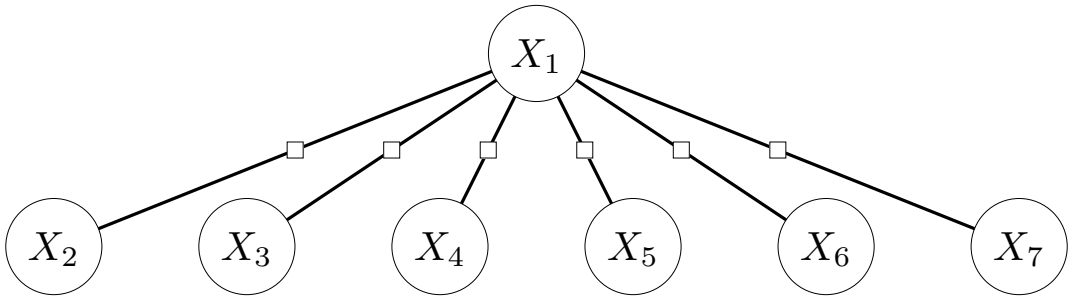
Let max-arity be the maximum arity of any $f_{\text{new},i}$.

Running time: $O(n \cdot |\text{Domain}|^{\text{max-arity}+1})$

- We can turn elimination directly into an actual algorithm for computing the maximum weight assignment by just repeating it until we are left with one variable. This is called the **variable elimination** algorithm.
- The running time of this algorithm is exponential in the maximum arity of the factor produced along the way in variable elimination. The arity in the worst case is $n - 1$, but in the best case it could be a lot better, as we will see.

Variable ordering

What's the maximum arity?



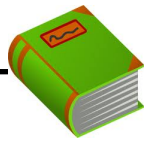
If eliminate leaves first, all factors have arity 1 (good)

If eliminate root first, get giant factor have arity 6 (bad)

Degree heuristic: eliminate variables with the fewest neighbors

- The arity of the factors produced during variable elimination depends on the ordering of the variables. In this extreme example, the difference is between 1 and 6.
- A useful heuristic is to eliminate variables with the smallest Markov blanket. In this example, the heuristic would eliminate the leaves and we'd only end up with factors with arity 1.

Treewidth



Definition: treewidth

The **treewidth** of a factor graph is the maximum arity of any factor created by variable elimination with the **best** variable ordering.

[whiteboard]

- Treewidth of a chain is 1.
- Treewidth of a tree is 1.
- Treewidth of simple cycle is 2.
- Treewidth of $m \times n$ grid is $\min(m, n)$.

- If we use the best ordering, the arity of the largest factor produced is known as the **treewidth**, a very important property in graph theory. Computing the treewidth in general is NP-complete, and verifying that treewidth is k is exponential in k (but linear in the number of nodes).
- However, in practice, it's useful to remember a few examples.
- The treewidth of a chain is 1, by just eliminating all the variables left to right.
- The treewidth of a tree is also 1 by eliminating the variables from the leaves first.
- The treewidth of a simple cycle is 2. By symmetry, we can pick any variable on the cycle; eliminating it creates a factor that connects its two neighbors.
- The treewidth of an $m \times n$ grid is more complex. Without loss of generality, assume that $m \leq n$. One can eliminate the variables by going along the columns left to right and processing the variables from the top row to the bottom row. Verify that when eliminating variable X_{ij} (in the i-th row and the j-th column), its Markov blanket is all the variables in column $j+1$ and row $\leq i$ as well as all the variables in column j but in row $> i$.
- Note that even if we don't know the exact treewidth, having an upper bound gives us a handle on the running time of variable elimination.

Summary

- Beam search: follows the most promising branches of search tree based on myopic information (think pruned BFS search)
- Local search: can freely re-assign variables; use randomness to get out of local optima
- Conditioning: break up a factor graph into smaller pieces (divide and conquer); can use in backtracking
- Elimination: solve a small subproblem conditioned on its Markov blanket

- Last lecture, we focused on algorithms that worked in the backtracking search framework. This lecture explores two classes of methods for efficienty finding the maximum weight assignment in a factor graph.
- The first class of methods are approximate methods. **Beam search**, like backtracking search, builds up partial assignments, but extends multiple partial assignments over the same subset of variables at once, and heuristically keeping the ones that seem most promising so far. It is quite possible that the actual maximum weight assignment will "fall off the beam". **Local search**, in contrast, works with complete assignments, modifying the value of one variable at time. In both beam and local search, one considers one variable at a time, and we only need to look at the factors touching that one variable.
- The second class of methods are exact methods that rely on (conditional) **independence** structure of the graph, in particular, that the graph is weakly connected, for example, a chain or a tree. We approached this methods by thinking about two graph operations, conditioning and elimination. **Conditioning** sets the value of a variable, and breaks up any factors that touch that variable. **Elimination** maximimizes over a variable, but since the maximum value depends on the Markov blanket, this maximization is encoded in a new factor that depends on all the variables in the Markov blanket. The variable elimination computes the maximum weight assignment by applying elimination to each variable sequentially.