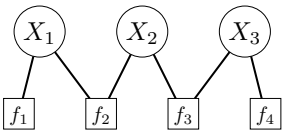
Lecture 5.2: CSPs II

Review: definition



Definition: factor graph

Variables:

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

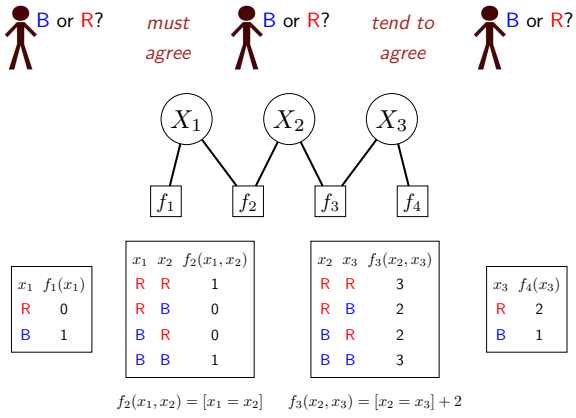
Factors:

$f_1, \ldots, f_m$, 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)



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

$f_2(x_1, x_2) = [x_1 = x_2]$

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

$f_3(x_2, x_3) = [x_2 = x_3] + 2$

x_3	$f_4(x_3)$
R	2
B	1

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

Roadmap

Modeling

Arc consistency

Beam search

Local search

Example: LSAT question

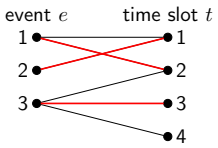
Three sculptures (A, B, C) are to be exhibited in rooms 1, 2 of an art gallery.

The exhibition must satisfy the following conditions:

- Sculptures A and B cannot be in the same room.
- Sculptures B and C must be in the same room.
- Room 2 can only hold one sculpture.

[demo]

Example: event scheduling (section)



Setup:

- Have E events and T time slots
- Each event e must be put in **exactly one** time slot
- Each time slot t can have **at most one** event
- Event e allowed in time slot t only if $(e, t) \in A$

- Consider a simple scheduling problem, where we have E events that we want to schedule into T time slots. There are three families of requirements: (i) every event must be scheduled into a time slot; (ii) every time slot can have at most one event (zero is possible); and (iii) we are given a fixed set A of (event, time slot) pairs which are allowed.
- There are in general multiple ways to cast a problem as a CSP. In section, you will see two reasonable ways to do it.

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

- Next, 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

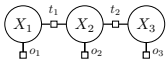
Problem: object tracking

Noisy sensors report positions: $0, 2, 2$.
Objects don't move very fast.
What path did the object take?

[whiteboard: trajectories over time]

Object 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.

Example: relation extraction

Motivation: build a question-answering system

Which US presidents played the guitar?

Prerequisite: learn knowledge by reading the web

Systems:

[NELL (CMU)] [OpenIE (UW)]

- Finally, let's look at a different problem. Some background which is unrelated to CSPs: A major area of research in natural language processing is **relation extraction**, the task of building systems that can process the enormous amount of unstructured text on the web, and populate a structured knowledge base, so that we can answer complex questions by querying the knowledge base.

Example: relation extraction

Input (hundreds of millions of web pages):

Barack Obama is the 44th and current President of the United States...

Output (database of relations):

```
EmployedBy(BarackObama, UnitedStates)
Profession(BarackObama, President)
...
```


Example: relation extraction

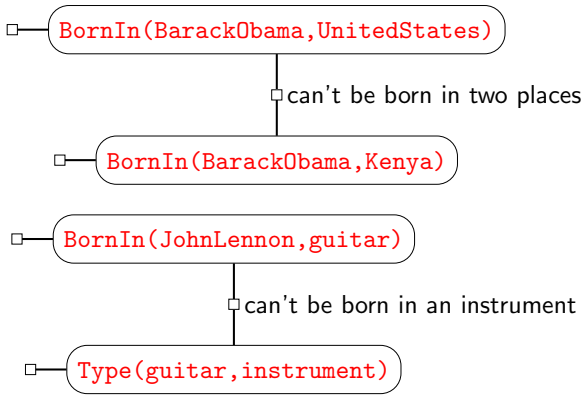
Typical predictions of classifiers:

`BornIn(BarackObama,UnitedStates)`	0.9
`BornIn(BarackObama,Kenya)`	0.6
`BornIn(JohnLennon,guitar)`	0.7
`Type(guitar,instrument)`	0.9
...	

How do reconcile conflicting predictions?

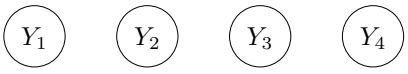
- State-of-the-art methods typically use machine learning, casting it as a classification problem. However, relation extraction is a very difficult problem, and even the best systems today often fail, producing nonsensical facts.
- A key observation is that these classification decisions are not independent, and we have some prior knowledge on how they should be related. For example, you can't be born in two places, and you also can't be born in an instrument (not usually, anyway).

Example: relation extraction



General framework

Classification decisions are generally related:



- Unary factors: local classifiers (provide evidence)

$$\exp(\mathbf{w} \cdot \phi(x_i) Y_i)$$

- Binary factors: enforce that outputs are consistent

$$[Y_i \text{ consistent with } Y_j]$$

- To operationalize this intuition, we can leverage factor graphs. Think about each of the classification decisions as a variable, which can take on 1 or 0 (assume binary classification for now).
- We have a unary factor which specifies the contribution of the classifier. Recall that linear classifiers return a score $\mathbf{w} \cdot \phi(x_i)$, which is a real number. Factors must be non-negative, so it's typical to exponentiate the score.
- We can add binary factors between pairs of classification decisions which are related in some way (e.g., $[\text{BornIn(BarackObama,UnitedStates)} + \text{BornIn(BarackObama,Kenya)} \leq 1]$). The factors do not have to be hard constraints, but rather general preferences that encode soft preferences (e.g., returning weight 0.01 instead of 0).
- Once we have a CSP, we can ask for the maximum weight assignment, which takes into account all the information available and reasons about it globally.

Roadmap

Modeling

Arc consistency

Beam search

Local search

Review: backtracking search

Algorithm: backtracking search

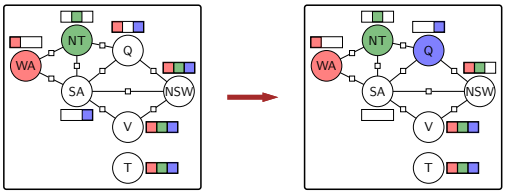
Backtrack$(x, w, \text{Domains})$:

- If x is complete assignment: update best and return
- Choose unassigned **VARIABLE** X_i
- Order **VALUES** Domain_i of chosen X_i
- For each value v in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If $\delta = 0$: continue
  - $\text{Domains}' \leftarrow$ Domains via **LOOKAHEAD**
  - Backtrack$(x \cup \{X_i : v\}, w\delta, \text{Domains}')$

Review: forward checking

Key idea: forward checking (one-step lookahead)

- After assigning a variable X_i, eliminate inconsistent values from the domains of X_i's neighbors.
- If any domain becomes empty, don't recurse.
- When unassign X_i, restore neighbors' domains.



Arc consistency

Idea: eliminate values from domains $\Rightarrow$ reduce branching

Example: numbers

Before enforcing arc consistency on X_i:

$X_i \in \text{Domain}_i = \{1, 2, 3, 4, 5\}$

$X_j \in \text{Domain}_j = \{1, 2\}$

$f_1(X) = [X_i + X_j = 4]$

After enforcing arc consistency on X_i:

$X_i \in \text{Domain}_i = \{2, 3\}$

- Now let us return to the issue of using lookahead to eliminate values from domains of unassigned variables. One motivation is that smaller domains lead to smaller branching factors, which makes search faster.
- A second motivation is that since the domain sizes are used in the context of the dynamic ordering heuristics (most constrained variable and least constrained value), we can hope to choose better orderings with domains that more accurately reflect what values are actually possible.
- We've already seen forward checking as a simple way of using lookahead to prune the domains of unassigned variables. Shortly, we will introduce AC-3, which is forward checking without brakes. To build up to that, we need to introduce the idea of arc consistency.
- The idea behind enforcing arc consistency is to look at all the factors that involve just two variables X_i and X_j and rule out any values in the domain of X_i which are obviously bad without even looking at other variables.
- To enforce arc consistency on X_i with respect to X_j, we go through each of the values in the domain of X_i and remove it if there is no value in the domain of X_j that is consistent with X_i. For example, $X_i = 4$ is ruled out because no value $X_j \in \{1, 2, 3, 4, 5\}$ satisfies $X_i + X_j = 4$.

Arc consistency

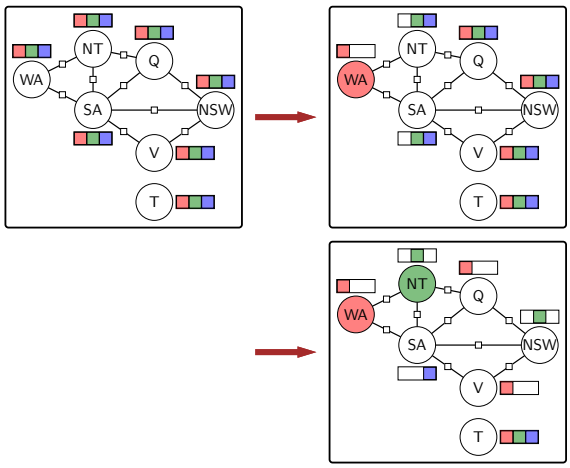
Definition: arc consistency

A variable X_i is **arc consistent** with respect to X_j if for each $x_i \in \text{Domain}_i$, there exists $x_j \in \text{Domain}_j$ such that $f(\{X_i : x_i, X_j : x_j\}) \neq 0$ for all factors f whose scope contains X_i and X_j.

Algorithm: enforce arc consistency

EnforceArcConsistency(X_i, X_j): Remove values from Domain_i to make X_i arc consistent with respect to X_j.

AC-3 (example)



AC-3

Forward checking: when assign $X_j : x_j$, set $\text{Domain}_j = \{x_j\}$ and enforce arc consistency on all neighbors X_i with respect to X_j

AC-3: repeatedly enforce arc consistency on all variables

Algorithm: AC-3

Add X_j to set.

While set is non-empty:

- Remove any X_k from set.
- For all neighbors X_l of X_k:
  - Enforce arc consistency on X_l w.r.t. X_k.
  - If Domain_l changed, add X_l to set.

AC-3 runtime

Let D be maximum domain size, E be number of edges

- EnforceArcConsistency(X_i, X_j) takes at most $O(D^2)$ time.
- For every edge (X_i, X_j), EnforceArcConsistency called at most $2D$ times.

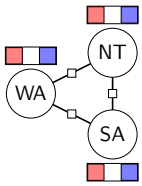
AC-3: $O(ED^3)$ in worst case (often better)

Overall: polynomial time work at each step, might save exponential time by pruning search tree

- In fact, we already saw a limited version of arc consistency. In forward checking, when we assign a variable X_i to a value, we are actually enforcing arc consistency on the neighbors of X_i with respect to X_i.
- Why stop there? AC-3 doesn't. In AC-3, we start by enforcing arc consistency on the neighbors of X_i (forward checking). But then, if the domains of any neighbor X_j changes, then we enforce arc consistency on the neighbors of X_j, etc.
- In the example, after we assign WA : R, performing AC-3 is the same as forward checking. But after the assignment NT : G, AC-3 goes wild and eliminates all but one value from each of the variables on the mainland.
- Note that unlike BFS graph search, a variable could get added to the set multiple times because its domain can get updated more than once. More specifically, we might enforce arc consistency on (X_i, X_j) up to D times in the worst case, where $D = \max_{1 \le i \le n} |\text{Domain}_i|$ is the size of the largest domain. There are at most m different pairs (X_i, X_j) and each call to enforce arc consistency takes $O(D^2)$ time. Therefore, the running time of this algorithm is $O(ED^3)$ in the very worst case where E is the number of edges (usually, it's much better than this).

Limitations of AC-3

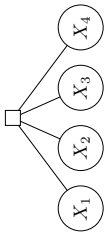
- Ideally, if no solutions, AC-3 would remove all values from a domain
- AC-3 isn't always effective:



- No consistent assignments, but AC-3 doesn't detect a problem!
- Intuition: if we look locally at the graph, nothing blatantly wrong...

- In the best case, if there is no way to consistently assign values all the variables, then running AC-3 will detect that there is no solution by emptying out a domain. However, this is not always the case, as the example above shows. Locally, everything looks fine, even though there's no global solution.
- Advanced: We could generalize arc consistency to fix this problem. Instead of looking at every 2 variables and the factors between them, we could look at every subset of k variables, and check that there's a way to consistently assign values to all k, taking into account all the factors involving those k variables. However, there is a substantial cost to doing this (the running time is exponential in k in the worst case), so generally arc consistency ($k = 2$) is good enough.

N-ary constraints (section)



Variables: $X_1, X_2, X_3, X_4 \in \{0, 1\}$

Factor: $[X_1 \vee X_2 \vee X_3 \vee X_4]$

Examples:

Weight($\{X_1 : 0, X_2 : 0, X_3 : 0, X_4 : 0\}$) $= 0$

Weight($\{X_1 : 0, X_2 : 1, X_3 : 0, X_4 : 0\}$) $= 1$

What if inference only takes unary/binary factors?

Can convert higher-arity factors into lower-arity ones!

- When we are modeling with factor graphs, we would like the factors to have any arity (depend on any number of variables). This allows us to, for example, require that at least one of the provinces is colored red.
- However, from an algorithms and implementation perspective, it is often useful to just think about unary and binary factors. For example, arc consistency is defined with respect to binary factors.
- It appears that there is a tradeoff between modeling expressivity and algorithmic efficiency, but this is actual not a real tradeoff, since we can reduce the general arity case to the unary-binary case.
- Consider the simple problem: given n variables $X_1, \ldots, X_n$, where each $X_i \in \{0, 1\}$, impose the requirement that at least one $X_i = 1$. The case of $n = 4$ is shown in the slide.
- In section, next week, we will see how to do this converstion.

Summary

- Basic template: backtracking search on partial assignments
- Dynamic ordering: most constrained variable (fail early), least constrained value (try to succeed)
- Lookahead: forward checking (enforces arc consistency on neighbors), AC-3 (enforces arc consistency on neighbors and their neighbors, etc.)

Roadmap

Modeling

Arc consistency

Beam search

Local search

Review: backtracking search

Vanilla version:

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

Lookahead: forward checking, AC-3

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

Dynamic ordering: most constrained variable, least constrained value

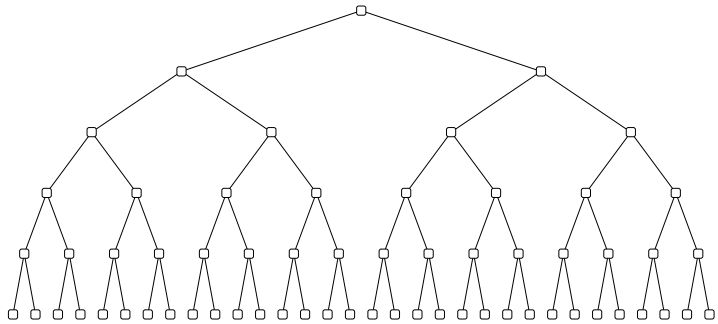
$O(|\text{Domain}|^n)$ time

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.

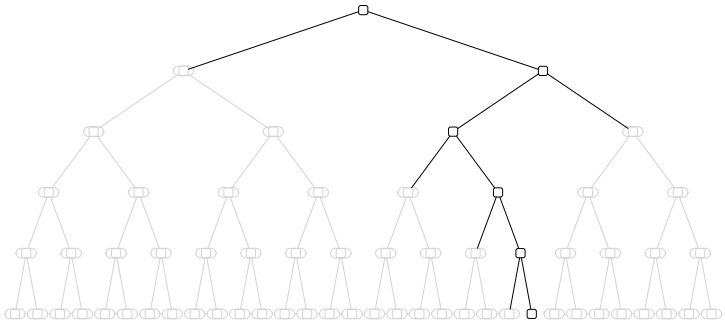
- In this lecture, we will discuss alternative ways to find maximum weight assignments efficiently without incurring the full cost of backtracking search.
- We will look at two **approximate** search algorithms: beam search and local search. We give up guarantees of finding the exact maximum weight assignment, but they can still work well in practice.

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 $x_v = x \cup \{X_i : v\}$

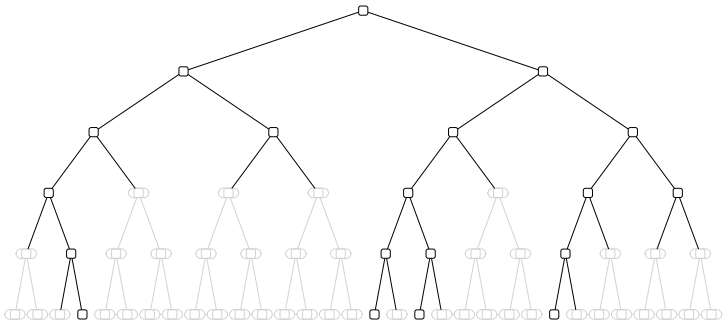
Prune:

$x \leftarrow x_v$ 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

Modeling

Arc consistency

Beam search

Local search

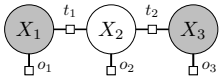
Local search

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)



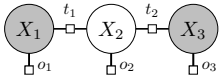
Current assignment: $(0, 0, 1)$; how to improve?

(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

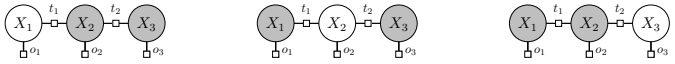
When evaluating possible re-assignments to X_i, only need to consider the factors that depend on X_i.

- 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

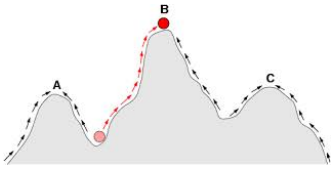


- Now we can state our first algorithm, ICM, which is the local search analogy of the greedy algorithm we described earlier. The idea is simple: we start with a random complete assignment. We repeatedly loop through all the variables X_i. On variable X_i, we consider all possible ways of re-assigning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

Summary

- Modeling: lots of possibilities!
- AC-3: forward checking without brakes
- 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
- Next time: structural properties of factor graphs