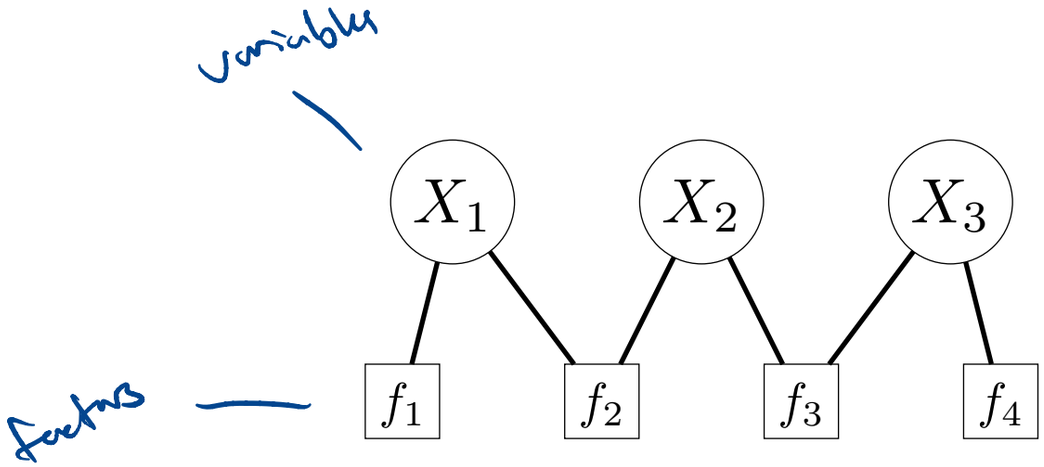
CSPs: dynamic ordering

- In the previous module, we spent some time with understanding CSPs from a modeling perspective.
- In this module, I will present an algorithm to perform inference (i.e., to solve a CSP) based on backtracking search.
- In particular, we will speed up vanilla backtracking search with dynamic ordering, where we prioritize which variables and values to process first.

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**:

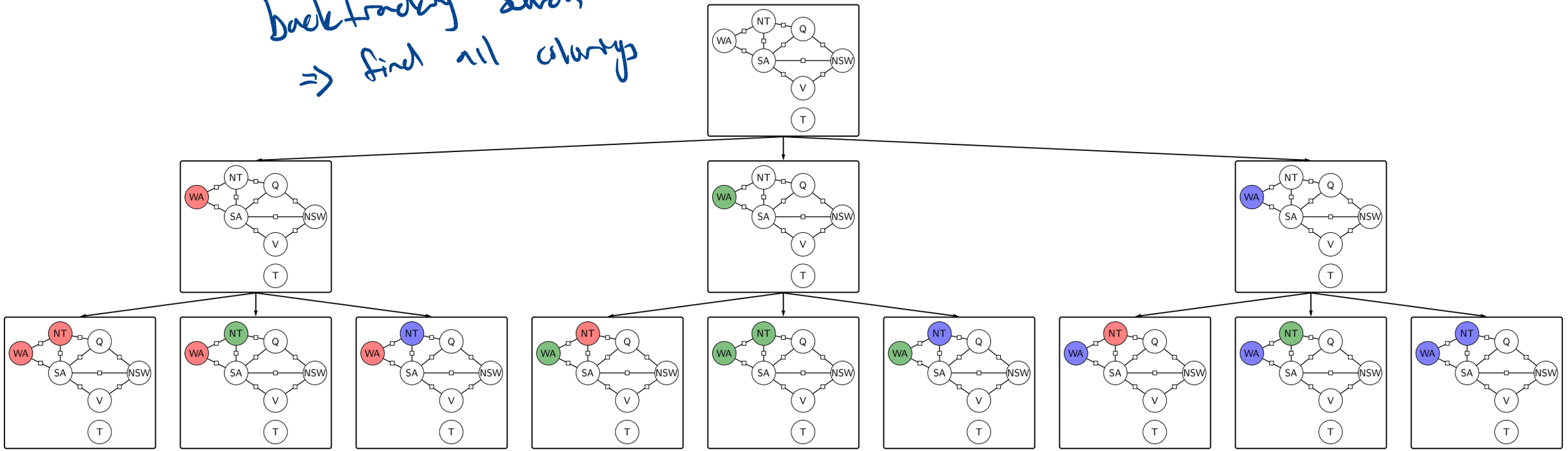
$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

Objective:

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

find $\vec{x}$ that maximizes Weight(x)

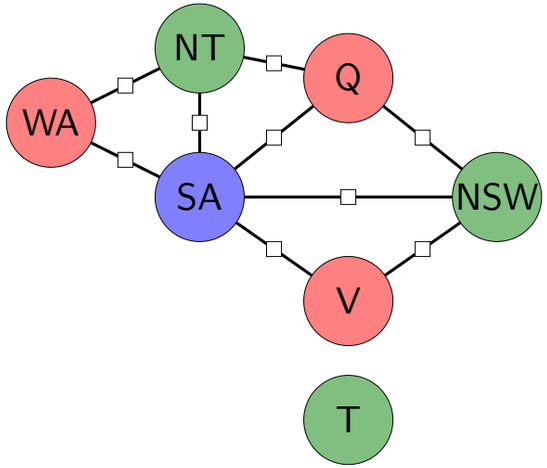
backtracking search
⇒ find all colorings



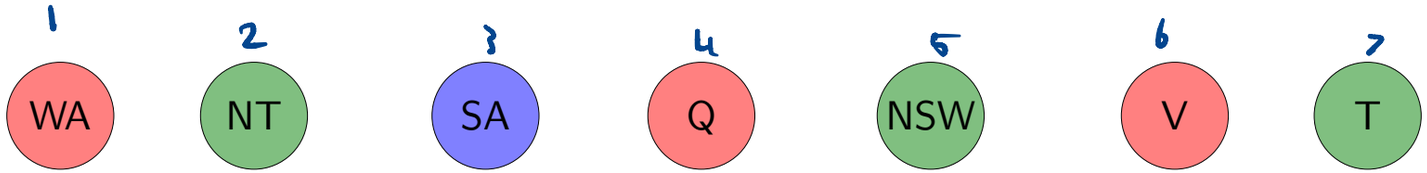
0 0 1

Partial assignment weights

Idea: compute weight of partial assignment as we go



assignment order



[WA ≠ NT]

[WA ≠ SA] [NT ≠ SA]

[NT ≠ Q] [SA ≠ Q]

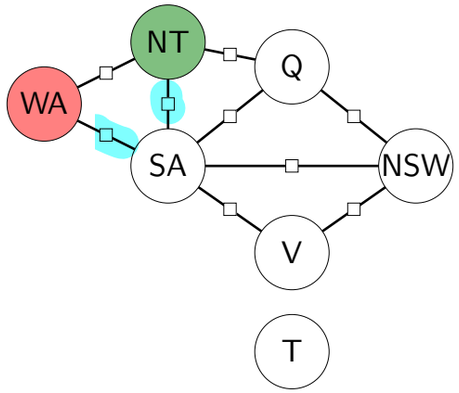
[SA ≠ NSW] [Q ≠ NSW]

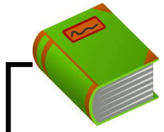
[SA ≠ V] [NSW ≠ V]

can evaluate this

Dependent factors

- Partial assignment (e.g., $x = \{\text{WA} : \text{R}, \text{NT} : \text{G}\}$)





Definition: dependent factors

Let $D(x, X_i)$ be set of factors depending on X_i and x but not on unassigned variables.

$$D(\{\text{WA} : \text{R}, \text{NT} : \text{G}\}, \text{SA}) = \{[\text{WA} \neq \text{SA}], [\text{NT} \neq \text{SA}]\}$$

x X_i [factors after we set X_i

Backtracking search

Algorithm: backtracking search

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

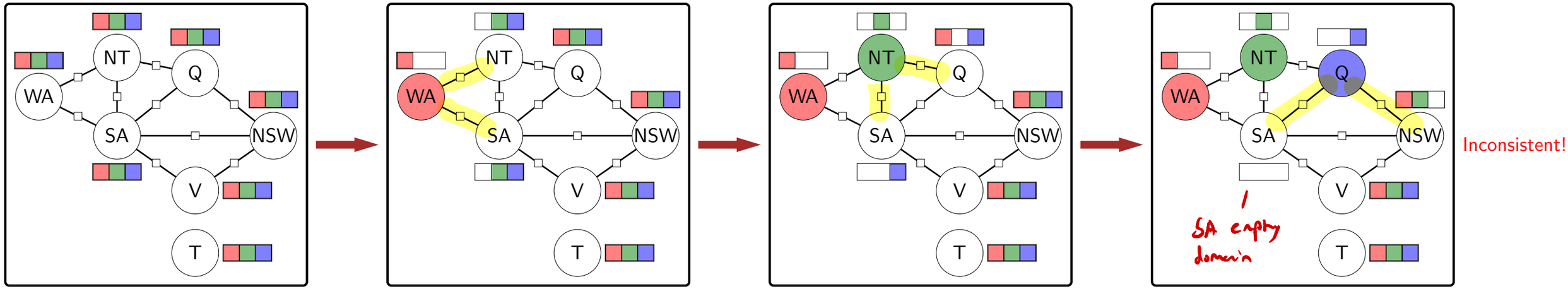
- If x is complete assignment: update best and return — leaf node
- Choose unassigned **VARIABLE** X_i
- Order **VALUES** Domain_i of chosen X_i
- For each value v in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$ — compute weight update
  - If $\delta = 0$: continue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

- Now we are ready to present the full backtracking search, which is a recursive procedure that takes in a partial assignment x, its weight w, and the domains of all the variables $\text{Domains} = (\text{Domain}_1, \ldots, \text{Domain}_n)$.
- If the assignment x is complete (all variables are assigned), then we update our statistics based on what we're trying to compute: We can increment the total number of assignments seen so far, check to see if x is better than the current best assignment that we've seen so far (based on w), etc. (For CSPs where all the weights are 0 or 1, we can stop as soon as we find one consistent assignment, just as in DFS for search problems.)
- Otherwise, we choose an **unassigned variable** X_i. Given the choice of X_i, we choose an **ordering of the values** of that variable X_i. Next, we iterate through all the values $v \in \text{Domain}_i$ in that order. For each value v, we compute δ, which is the product of the dependent factors $D(x, X_i)$; recall this is the multiplicative change in weight from assignment x to the new assignment $x \cup \{X_i : v\}$. If $\delta = 0$, that means a constraint is violated, and we can ignore this partial assignment completely, because multiplying more factors later on cannot make the weight non-zero.
- We then perform **lookahead**, removing values from the domains Domains to produce $\text{Domains}'$. This is not required (we can just use $\text{Domains}' = \text{Domains}$), but it can make our algorithm run faster. (We'll see one type of lookahead in the next slide.)
- Finally, we recurse on the new partial assignment $x \cup \{X_i : v\}$, the new weight $w\delta$, and the new domain $\text{Domains}'$.
- If we choose an unassigned variable according to an arbitrary fixed ordering, order the values arbitrarily, and do not perform lookahead, we get the basic tree search algorithm that we would have used if we were thinking in terms of a search problem. We will next start to improve the efficiency by exploiting properties of the CSP.

Lookahead: forward checking

Key idea: forward checking (one-step lookahead)

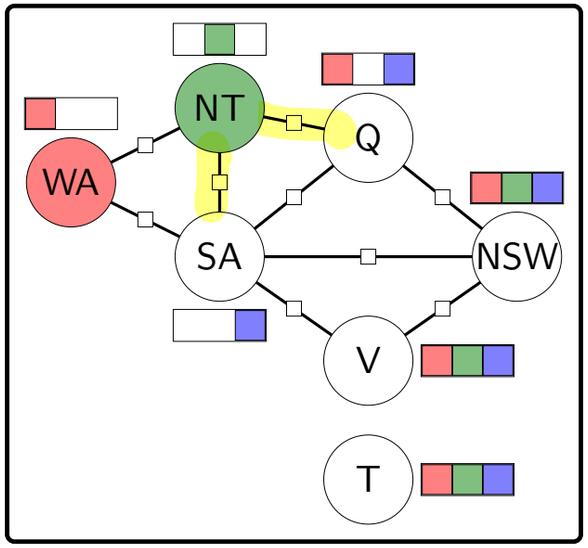
- After assigning a variable X_i, eliminate inconsistent values from the domains of X_i's neighbors.
- If any domain becomes empty, return.



Visualize domains at all

after assign and, remove inconsistent values in domains

Choosing an unassigned variable



Which variable to assign next?

Key idea: most constrained variable

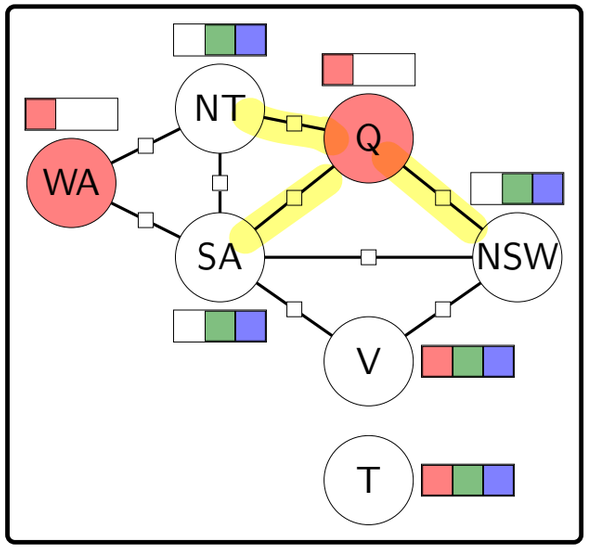
Choose variable that has the smallest domain.

This example: SA (has only one value)

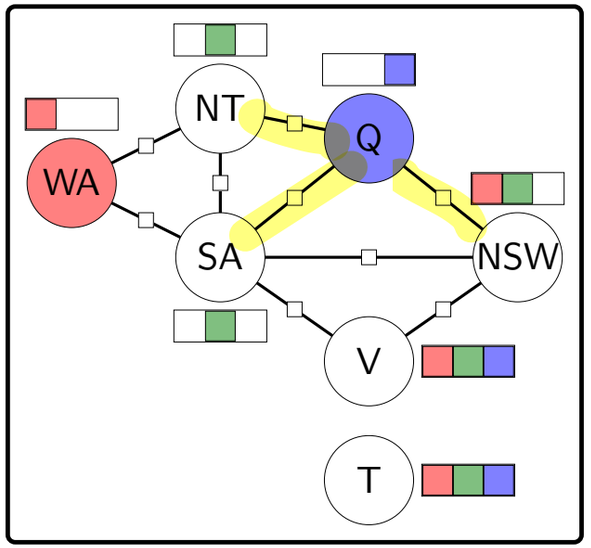
↑ just a heuristic

Ordering values of a selected variable

What values to try for Q?



$2 + 2 + 2 = 6$ consistent values



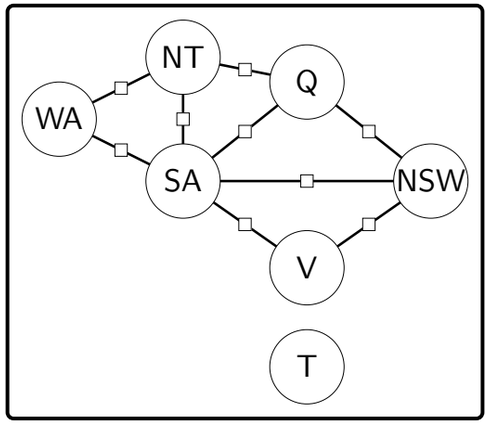
$1 + 1 + 2 = 4$ consistent values

=> try red first

Key idea: least constrained value

Order values of selected X_i by decreasing number of consistent values of neighboring variables.

When to fail?



Most constrained variable (MCV):

- Must assign **every** variable
- If going to fail, fail early $\Rightarrow$ more pruning

Least constrained value (LCV):

- Need to choose **some** value
- Choose value that is most likely to lead to solution

When do these heuristics help?

- Most constrained variable: useful when **some** factors are constraints (can prune assignments with weight 0)

$$[x_1 = x_2] \qquad\qquad [x_2 \neq x_3] + 2$$

- Least constrained value: useful when **all** factors are constraints (all assignment weights are 1 or 0) → if all constraints, just have to find nonzero weight assignment

$$[x_1 = x_2] \qquad\qquad [x_2 \neq x_3]$$

- Forward checking: needed to prune domains to make heuristics useful!

→ e.g. Pruning domains of colors

Summary

Algorithm: backtracking search

Backtrack$(x, w,$ Domains$)$:

- If x is complete assignment: update best and return
- Choose unassigned **VARIABLE** X_i (MCV)
- Order **VALUES** Domain$_i$ of chosen X_i (LCV)
- 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
  - Domains$'$ $\leftarrow$ Domains via **LOOKAHEAD** (forward checking)
  - If any Domains$'_i$ is empty: continue
  - Backtrack$(x \cup \{X_i : v\}, w\delta,$ Domains$')$

- In conclusion, we have presented backtracking search for finding the maximum weight assignment in a CSP, with some bells and whistles.
- Given a partial assignment, we first choose an unassigned variable X_i. For this, we use the most constrained variable (MCV) heuristic, which chooses the variable with the smallest domain.
- Next we order the values of X_i using the least constrained value (LCV) heuristic, which chooses the value that constrains the neighbors of X_i the least.
- We multiply all the new factors to get δ.
- Then we perform lookahead (forward checking) to prune down the domains, so that MCV and LCV can work on the latest information.
- Finally, we recurse with the new partial assignment.
- All of these heuristics aren't guaranteed to speed up backtracking search, but can often make a big difference in practice.

CSPs: arc consistency

Arc consistency: example

Example: numbers

Before enforcing arc consistency on X_i:

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

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

Factor: $[X_i + X_j = 4]$

After enforcing arc consistency on X_i:

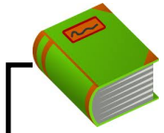
$X_i \in \text{Domain}_i = \{2, 3\}$

X_i	~~1~~	2	3	~~4~~	~~5~~
X_j	1	2			

$\forall n \in X_i$ remove n if does not exist $m \in X_j$ that satisfy factor

- 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



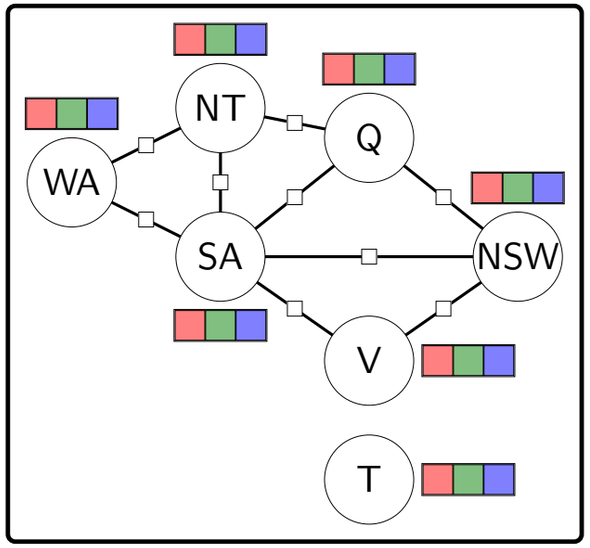
Definition: arc consistency

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.

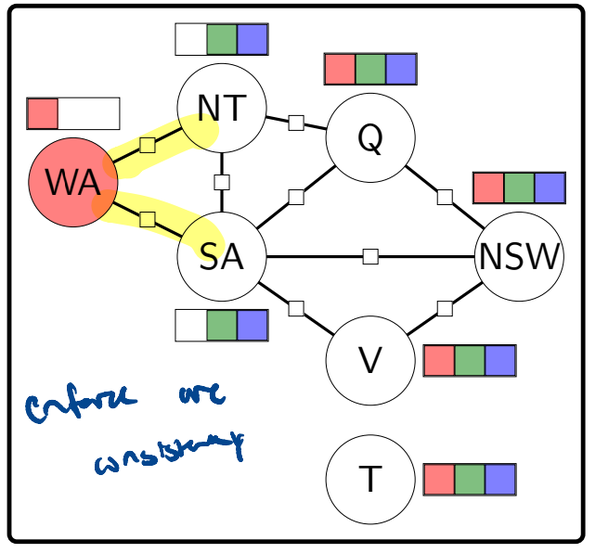
Algorithm: enforce arc consistency

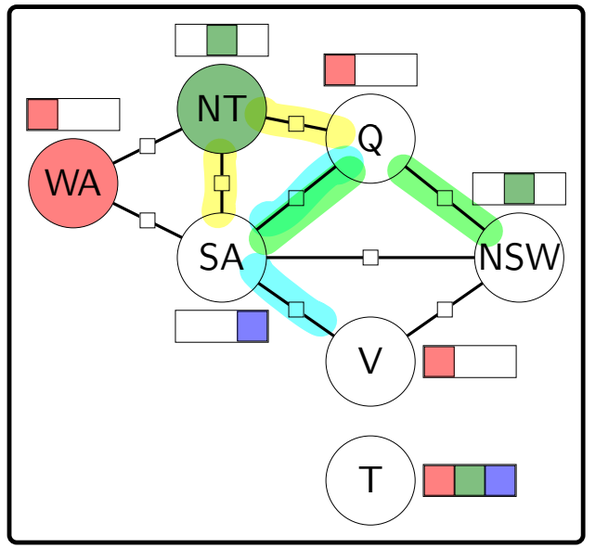
EnforceArcConsistency(X_i, X_j): Remove values from Domain_i to make X_i arc consistent with respect to X_j.

AC-3 (example)



set WA = R





enforce AC on NT

then enforce AC on Q

then enforce AC on SA

then enforce AC on NSW

etc

* just removes domains so later backtracking is faster

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

set of variables to process

Algorithm: AC-3

$S \leftarrow \{X_j\}$.

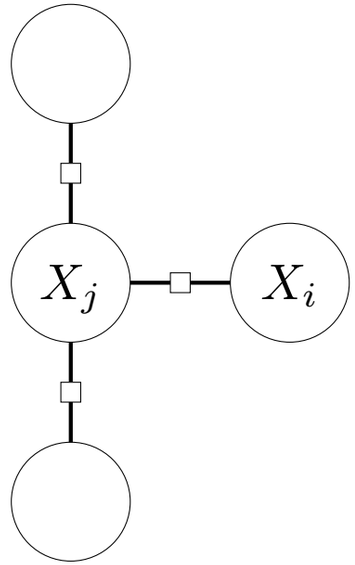
While S is non-empty:

  Remove any X_j from S.

  For all neighbors X_i of X_j:

    Enforce arc consistency on X_i w.r.t. X_j.

    If Domain_i changed, add X_i to S.

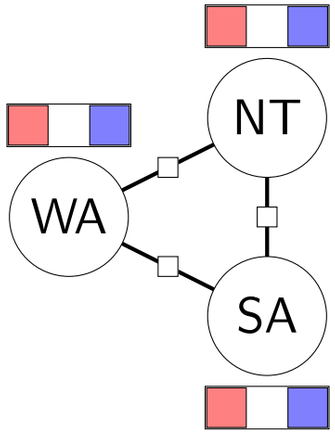


kind of like BFS, but can visit node multiple times

- 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.
- 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 $D = \max_{1 \leq i \leq n} |\text{Domain}_i|$ 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 (though usually, it's much better than this).

Limitations of AC-3

- 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...

Summary

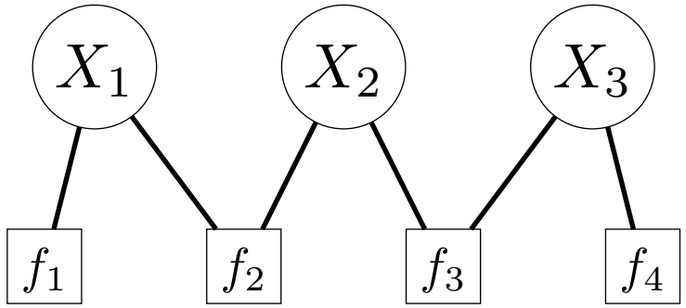
- Enforcing arc consistency: make domains consistent with factors
- Forward checking: enforces arc consistency on neighbors
- AC-3: enforces arc consistency on neighbors and their neighbors, etc.
- Lookahead very important for backtracking search!

- In summary, we presented the idea of enforcing arc consistency, which prunes domains based on information from a neighboring variable.
- After assigning a variable, forward checking enforces arc consistency on its neighbors, while AC-3 does it to the neighbors of neighbors, etc.
- Recall that AC-3 (or any lookahead algorithm) is used in the context of backtracking search to reduce the branching factor and keeps the dynamic ordering heuristics accurate. It can make a big difference!

CSPs: beam search

- In this module, 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.

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

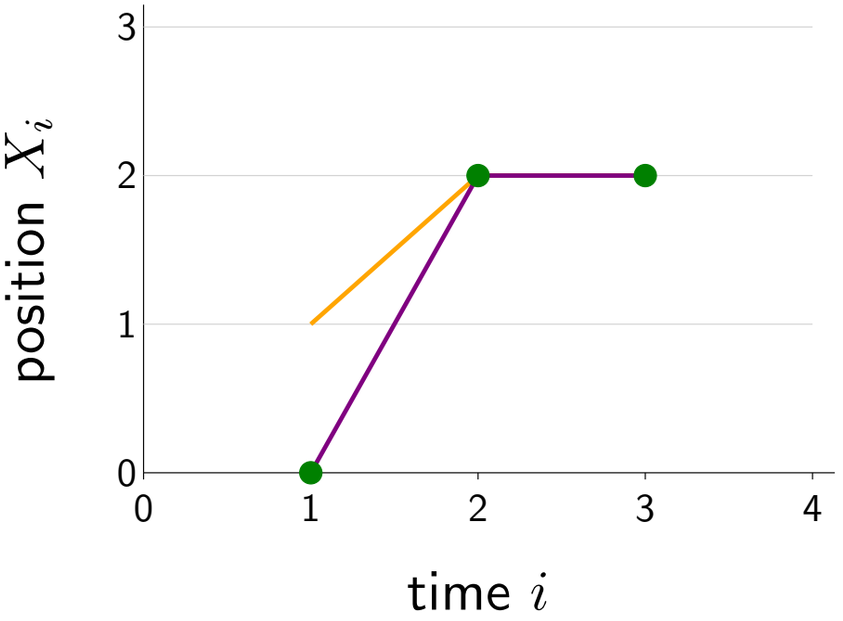
Each **assignment** $x = (x_1, \ldots, x_n)$ has a **weight**:

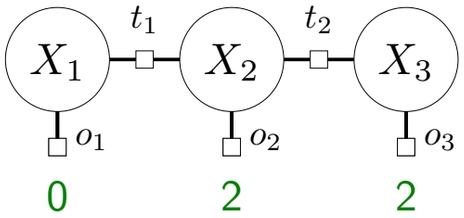
$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

Objective:

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

Example: object tracking





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

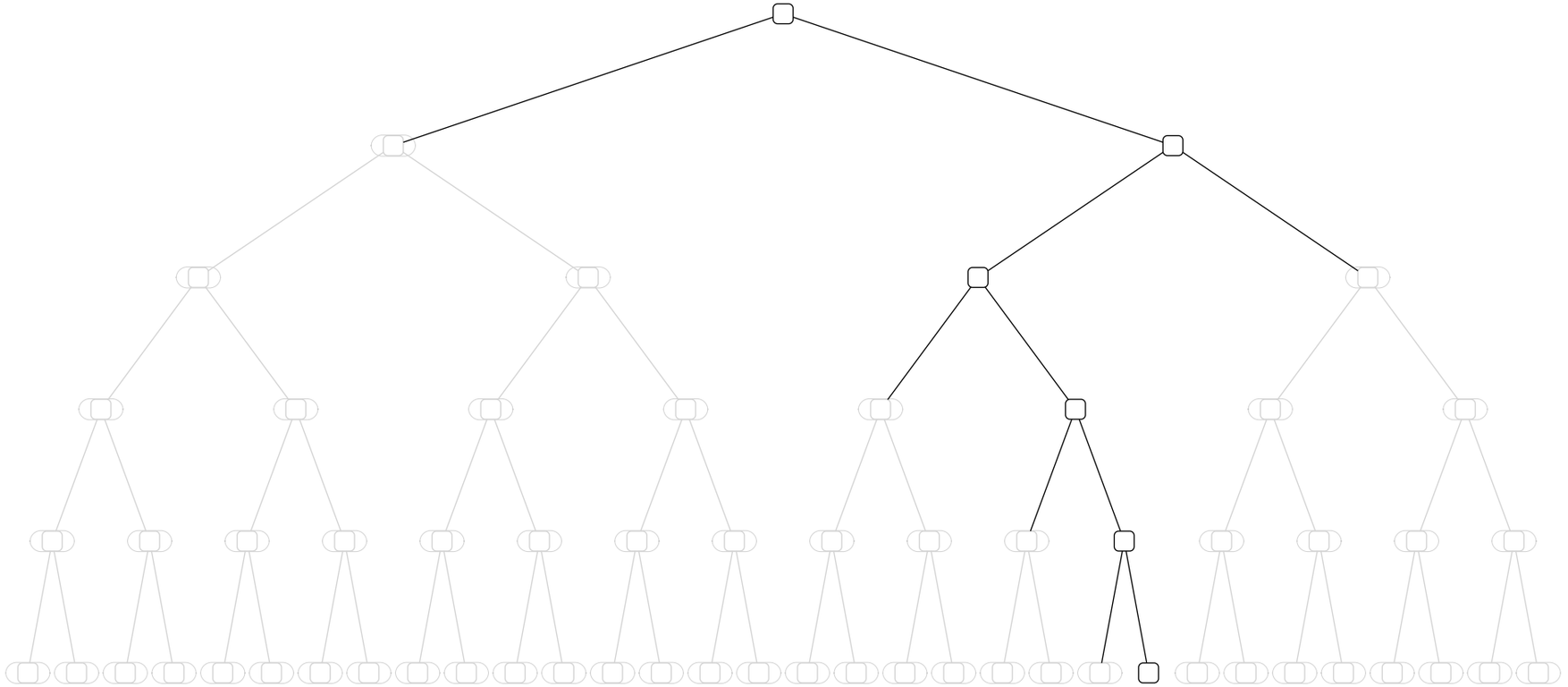
x_2	$o_2(x_2)$
0	0
1	1
2	2

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

- 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.

Greedy search



- don't backtrack, just choose option w/ highest weight of partial assignment

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

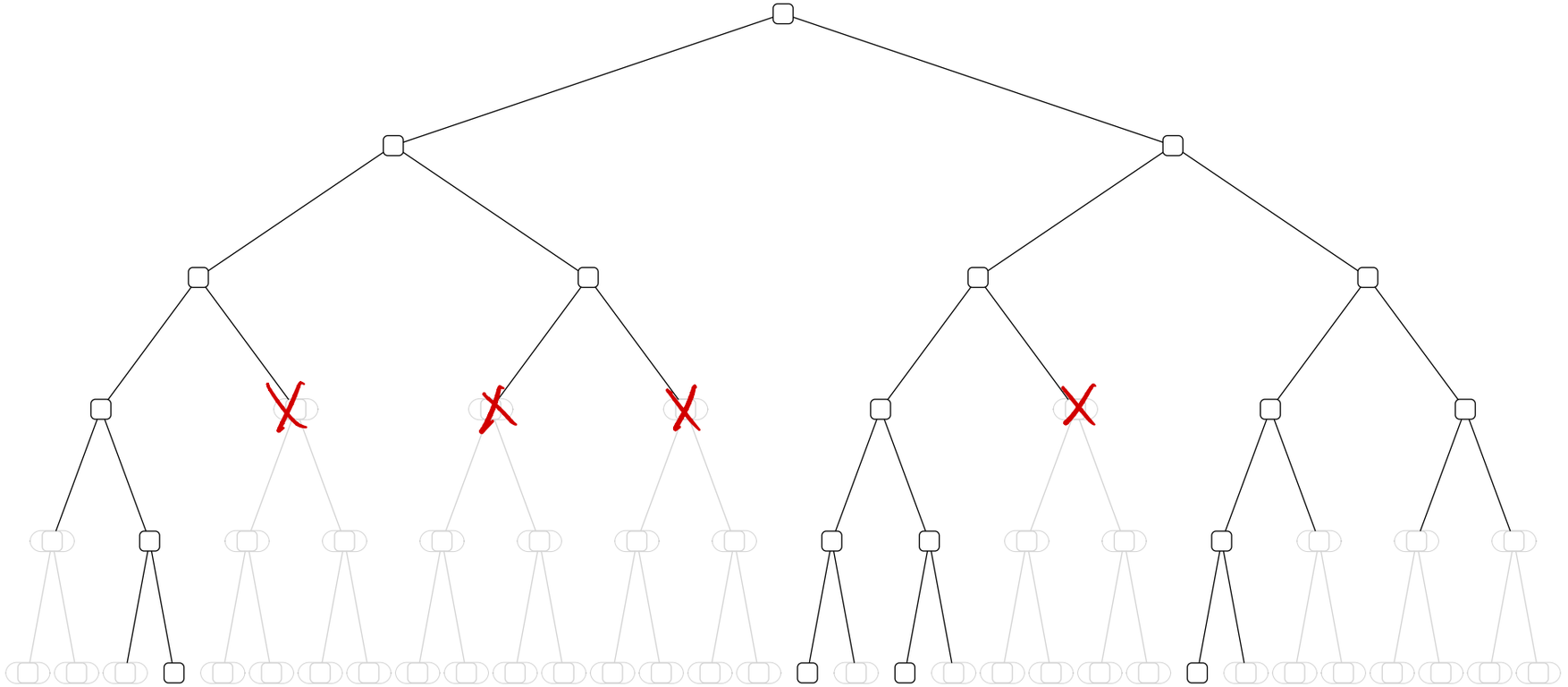
list of candidates

Not guaranteed to find maximum weight assignment!

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

but very fast

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:

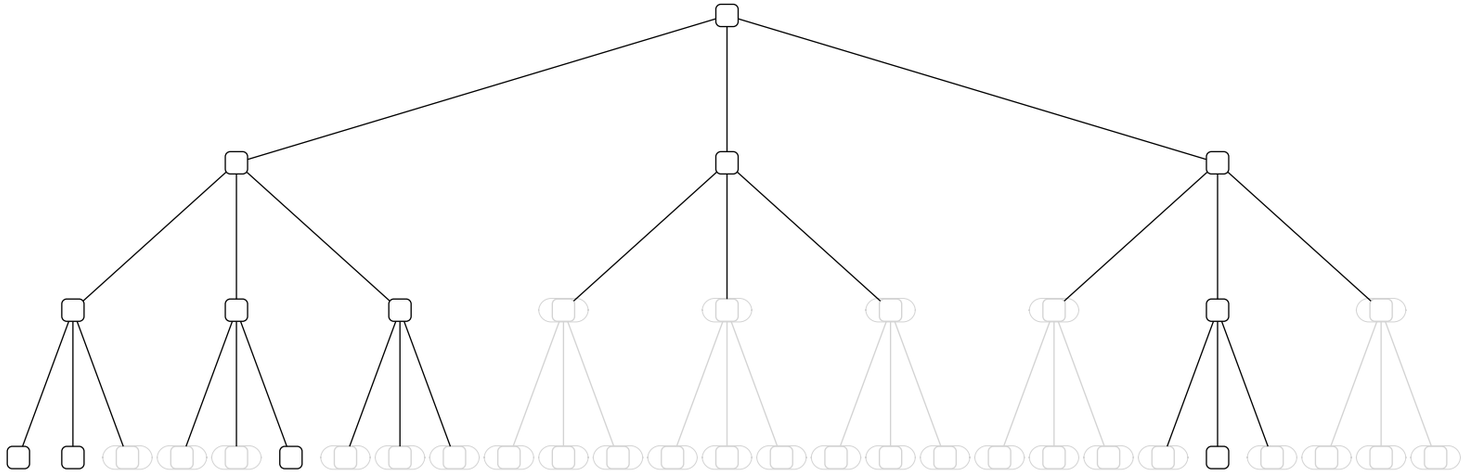
$C \leftarrow K$ elements of C' with highest weights

extend all candidates

Not guaranteed to find maximum weight assignment!

- 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.

Time complexity



n variables (depth)

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

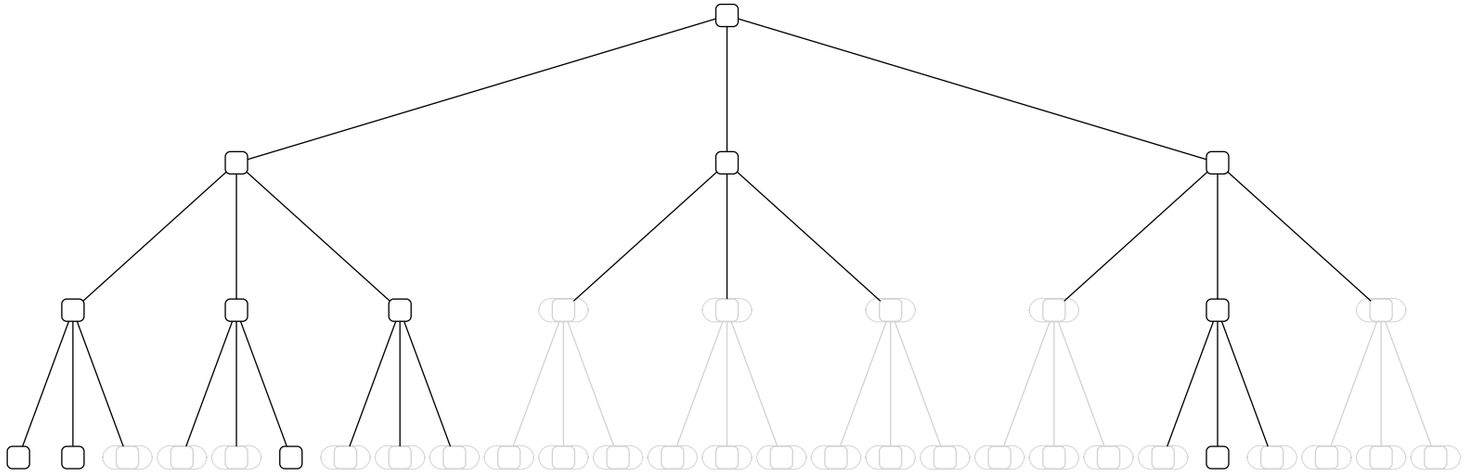
Beam size K

at each depth, select top K candidates

CSP has n variables

each variable has b possible values

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)

\ just an approximation

Backtracking search : DFS :: beam search : pruned BFS

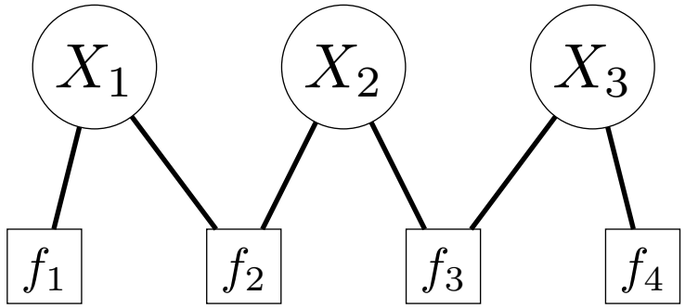
- In summary, we have presented a simple heuristic for approximating maximum weight assignments, beam search.
- Beam search offers a nice way to tradeoff efficiency and accuracy and is used quite commonly in practice, especially on naturally sequential problems like optimizing over sentences (sequences 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.

* pruning based on factors

CSPs: local search

- In this module, I will talk about local search, a strategy for approximately computing the maximum weight assignment in a CSP.

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:

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

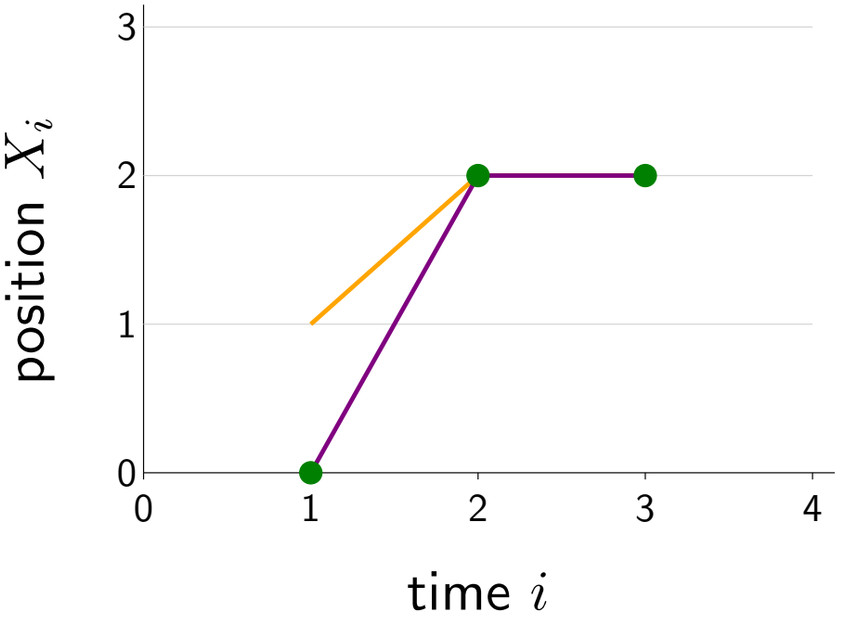
Search strategies

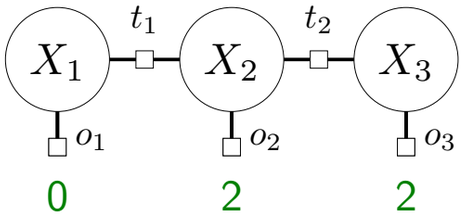
Backtracking/beam search: extend partial assignments

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

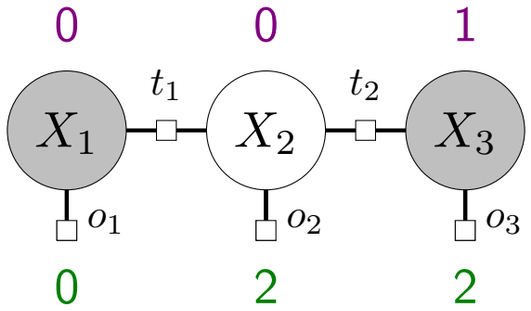
x_2	$o_2(x_2)$
0	0
1	1
2	2

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

- 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



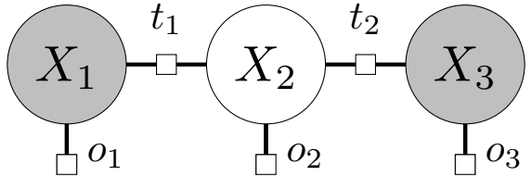
Old assignment: $(0, 0, 1)$; how to improve?

(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: $(0, 1, 1)$

- 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.

Iterated conditional modes (ICM)

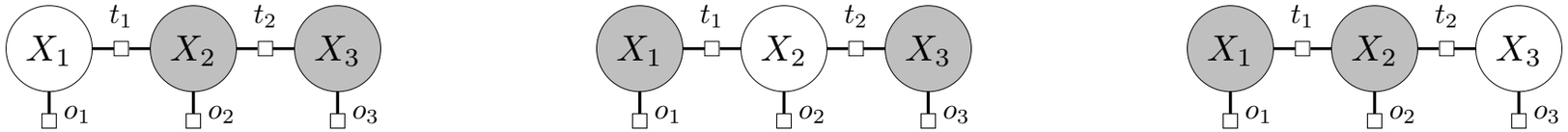
Algorithm: iterated conditional modes (ICM)

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

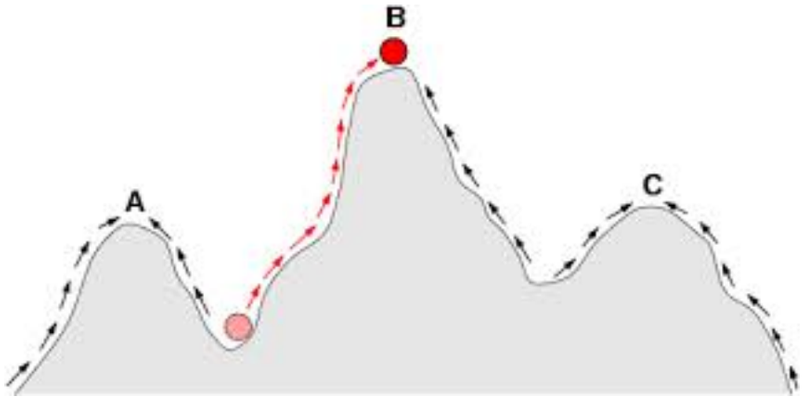
$x \leftarrow x_v$ with highest weight



- Now we can state our first algorithm, ICM. 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.

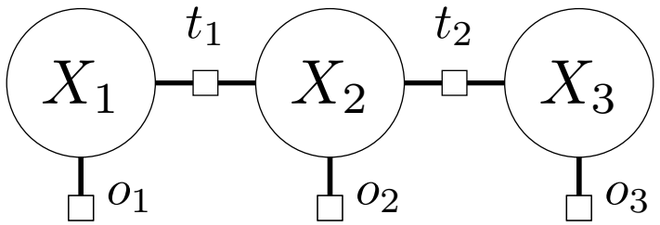
Convergence properties

- 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 module, 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.