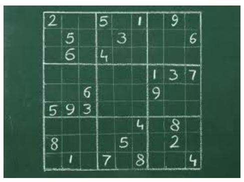
Constraint Satisfaction Problems (CSPs)



Roadmap

Modeling

Definitions

Examples

Backtracking (exact) search

Dynamic ordering

Arc consistency

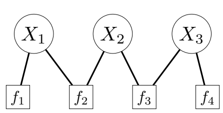
Approximate search

Beam search

Local search

- A quick reminder of the roadmap for the modules on CSPs. In the previous lecture, we defined constraint satisfaction problems and factor graphs formally, and gave a few examples of CSPs.
- Today, we will talk about backtracking search, which solves the problem exactly, though it takes exponential time in the worst case. To speed up search, we can take advantage of the fact that we can assign variables in any order to do dynamic ordering, where we heuristically figure out which variables to assign first. Arc consistency provides an lookahead algorithm called AC-3 to eagerly prune the search space, so that dynamic ordering can be more effective.
- Sometimes, you might not want to wait an exponential amount of time. If a crude solution suffices, one can apply approximate search algorithms. **Beam search** heuristically explores a small fraction of the exponentially-sized search tree, while **local search** takes an initial assignment and iteratively tries to improve it by changing one variable at a time.

Review: CSPs



Definition: factor graph

Variables:

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

Factors:

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

Definition: assignment weight

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

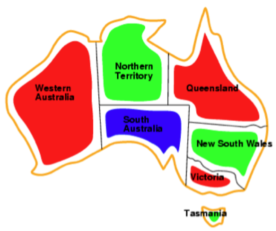
$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

Objective:

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

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

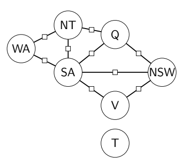
Map coloring



(one possible solution)

- Our running example will be the map coloring problem introduced in the previous lecture. How can we color the 7 states and territories of Australia with three colors so that no two neighboring provinces have the same color?

Example: map coloring



Variables:

$X = (\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T})$

$\text{Domain}_i \in \{\text{R}, \text{G}, \text{B}\}$

Factors:

$f_1(X) = [\text{WA} \neq \text{NT}]$

$f_2(X) = [\text{NT} \neq \text{Q}]$

...

- We represent the map coloring problem by a factor graph.
- For each province, we have a variable, whose domain is the three colors.
- We have one factor for each pair of neighboring provinces which returns 1 (okay) if the two colors are not equal and 0 otherwise.

Lecture

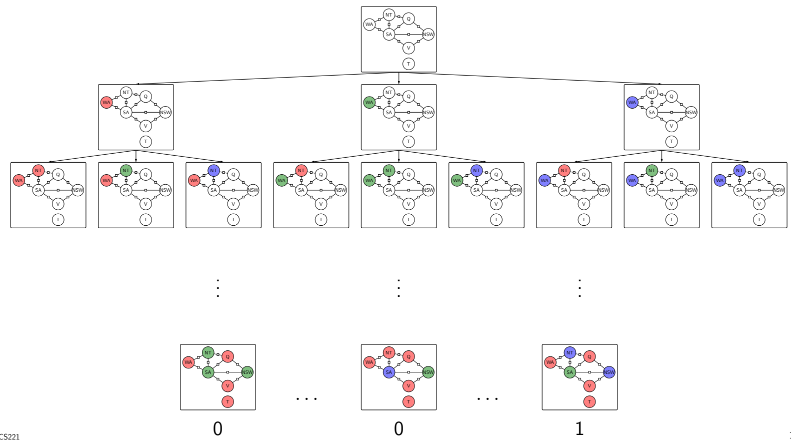
Dynamic Ordering

Arc Consistency

Beam Search

Local Search

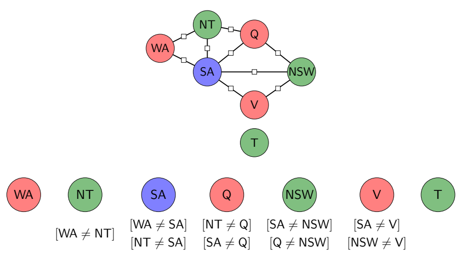
- In the previous module, we spent some time with understanding CSPs from a modeling perspective.
- In this module, I will present an algorithm to perform inference (i.e., to solve a CSP) based on backtracking search.
- In particular, we will speed up vanilla backtracking search with dynamic ordering,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



- Our starting point is **backtracking search**, where each node represents a **partial assignment** of values to a subset of the variables, and each child node represents an extension of the partial assignment.
- The leaves of the search tree represent complete assignments.
- We can simply explore the whole search tree, compute the weight of each complete assignment (leaf), and keep track of the maximum weight assignment.
- However, we will show that incrementally computing the weight along the way can be much more efficient.

Partial assignment weights

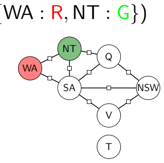
Idea: compute weight of partial assignment as we go



- Recall that the weight of an assignment is the product of all the factors.
- We define the weight of a partial assignment to be the product of all the factors that we can evaluate, namely those whose scope includes only assigned variables.
- For example, if only WA and NT are assigned, the weight is just value of the single factor between them.
- When we assign a new variable a value, the weight of the new extended assignment is the old weight times all the factors that depend on the new variable and only previously assigned variables.

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.

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

- Formally, we will use $D(x, X_i)$ to denote this set of these factors, which we will call **dependent factors**.
- For example, if we assign SA, then $D(x, \text{SA})$ contains two factors: the one between SA and WA and the one between SA and NT.

Backtracking search

Algorithm: backtracking search

Backtrack(x, w, Domains):

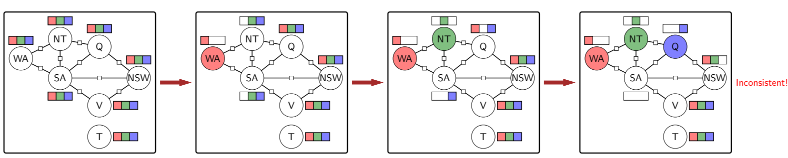
- If x is complete assignment: update best and return
- Choose unassigned **VARIABLE** X_i
- Order **VALUES** Domain$_i$ of chosen X_i
- For each value v in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If $\delta = 0$: continue
  - Domains$'$ $\leftarrow$ Domains via **LOOKAHEAD**
  - If any Domains$'_i$ is empty: continue
  - Backtrack($x \cup \{X_i : v\}, w\delta$, 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 Domains = (Domain$_1, \dots,$ 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 $v \in$ 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 Domains$'$. This is not required (we can just use Domains$'$ = 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 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

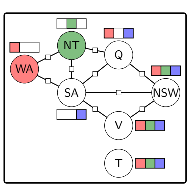
Key idea: forward checking (one-step lookahead)

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



- First, we will look at **forward checking**, which is a way to perform a one-step lookahead.
- For each variable, we visualize its domain, which is the set of values that that variable could still take on given the partial assignment.
- As soon as we assign a variable (e.g., WA = R), we can pre-emptively remove inconsistent values from the domains of neighboring variables (i.e., those that share a factor).
- If we get to a point where some variable has an empty domain, then we can stop and backtrack immediately, since there's no possible way to assign a value to that variable which is consistent with the previous partial assignment.
- In this example, after Q is assigned blue, we remove inconsistent values (blue) from SA's domain, emptying it. At this point, we need not even recurse further, since there's no way to extend the current assignment. We would then instead try assigning Q to red.
- Note that what is being visualized here is just one path down the search tree.
- Later, we will look at AC-3, which will allow us to lookahead even more to eliminate more values from the domains.

Choosing an unassigned variable



Which variable to assign next?

Key idea: most constrained variable

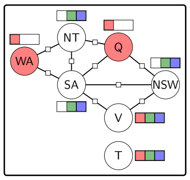
Choose variable that has the smallest domain.

This example: SA (has only one value)

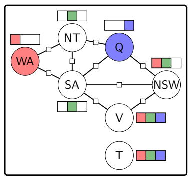
- Now let us look at the problem of choosing an unassigned variable.
- Intuitively, we want to choose the variable which is most constrained, that is, the variable whose domain has the fewest number of remaining valid values (based on forward checking), because those variables yield smaller branching factors.
- Note that this is just a heuristic.

Ordering values of a selected variable

What values to try for Q?



$2 + 2 + 2 = 6$ consistent values



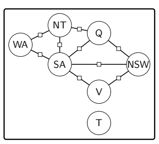
$1 + 1 + 2 = 4$ consistent values

Key idea: least constrained value

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

- Once we've selected an unassigned variable X_i, we need to figure out which order to try the different values in. The principle we will follow is to first try values which are less constrained.
- There are several ways we can think about measuring how constrained a variable is, but for the sake of concreteness, here is the heuristic we'll use: just count the number of values in the domains of all neighboring variables (those that share a factor with X_i).
- If we color Q red, then we have 2 valid values for NT, 2 for SA, and 2 for NSW. If we color Q blue, then we have only 1 for NT, 1 for SA, and 2 for NSW. Therefore, red is preferable (6 total valid values versus 4).
- The intuition is that we want values which impose the fewest number of constraints on the neighbors, so that we are more likely to find a consistent assignment.

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

- The most constrained variable and the least constrained value heuristics might seem at odds with each other, but this is only a superficial difference.
- An assignment requires setting **every** variable, whereas for each variable we only need to choose **some** value.
- Therefore, for variables, we want to try to detect failures as early as possible; we'll have to confront those variables sooner or later anyway).
- For values, we want to steer away from possible failures because we might not have to consider those other values if we find a happy path.

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)

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

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

- Most constrained variable is useful for finding maximum weight assignments as long as there are some factors which are constraints (return 0 or 1). This is because we only save work if we can prune away assignments with zero weight, and this only happens with violated constraints (weight 0).
- On the other hand, least constrained value only makes sense if all the factors are constraints. In general, ordering the values makes sense if we're going to just find the first consistent assignment. If there are any non-constraint factors, then we need to look at all consistent assignments to see which one has the maximum weight. Analogy: think about when depth-first search is guaranteed to find the minimum cost path.

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.

Lecture

Dynamic Ordering

Arc Consistency

Beam Search

Local Search

- In this module, we will introduce the notion of arc consistency, which will lead us to a lookahead algorithm called AC-3 for pruning domains and speeding up backtracking search.

Arc consistency: example

Example: numbers

Before enforcing arc consistency on X_i:

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

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

Factor: $[X_i + X_j = 4]$

After enforcing arc consistency on X_i:

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

X_i	1	2	3	4	5
X_j	1	2			

- To build up to AC-3, we need to introduce the idea of arc consistency.
- 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

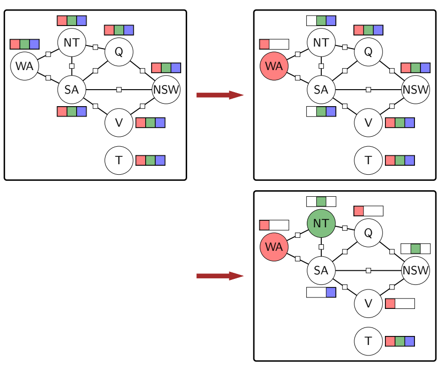
A variable X_i is **arc consistent** with respect to X_j if for each $x_i \in \text{Domain}_i$, there exists $x_j \in \text{Domain}_j$ such that $f(\{X_i : x_i, X_j : x_j\}) \neq 0$ for all factors f whose scope contains X_i and X_j.

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.

- Formally, a variable X_i is arc consistent with respect to X_j if every value in the domain of X_i has some potential partner in the domain of X_j.
- Enforcing arc consistency just makes it so.

AC-3 (example)



- The AC-3 algorithm simply enforces arc consistency until no domains change. Let's walk through this example.
- We start with the empty assignment.
- Suppose we assign WA to R. Then we enforce arc consistency on the neighbors of WA. Those domains change, so we enforce arc consistency on their neighbors, but nothing changes. Note that at this point AC-3 produces the same output as forward checking.
- We recurse and suppose we now pick NT and assign it G. Then we enforce arc consistency on the neighbors of NT (which is forward checking), SA, Q, NSW.
- AC-3 converges at this point, but note how much progress we've made: we have have nailed down the assignment for all the variables (except T)!
- Importantly, though we've touched all the variables, we are still at the NT node in the search tree. We've just paved the way, so that when we recurse, there's only one value to try for SA, Q, NSW, and V!

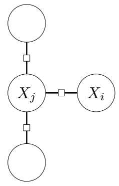
AC-3

Forward checking: when assign $X_j : x_j$, set 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

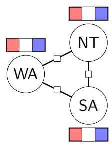
$S \leftarrow \{X_j\}$.
While S is non-empty:
  Remove any X_j from S.
  For all neighbors X_i of X_j:
    Enforce arc consistency on X_i w.r.t. X_j.
    If Domain$_i$ changed, add X_i to S.



- 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 $D = \max_{1 \le i \le n} |\text{Domain}_i|$ 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 (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...

- The previous example showed that AC-3 essentially solves the CSP, but this is too good to be true in general.
- Here is an example of a simplified factor graph, where running AC-3 won't prune any of the domains. While the domains are not empty, there is no solution.
- AC-3 really only spot checks the domains locally. Locally, everything looks fine, even though there's no global solution (impossible to assign two colors to 3 provinces).
- 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.

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!

Lecture

Dynamic Ordering

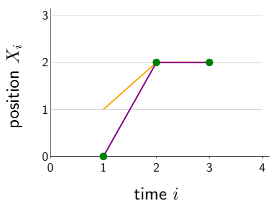
Arc Consistency

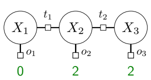
Beam Search

Local 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.

Example: object tracking





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

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

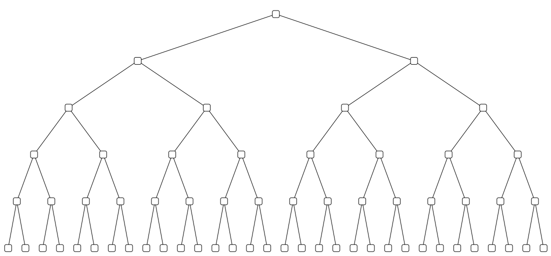
x_3	$o_3(x_3)$
0	0
1	1
2	2

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

[demo]

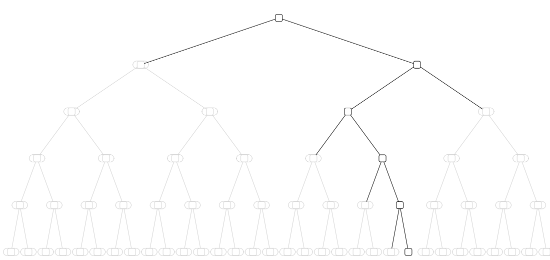
- Recall the object tracking example in which we observe noisy sensor readings 0, 2, 2.
- We have observation factors o_i that encourage the position X_i and the corresponding sensor reading to be nearby.
- We also have transition factors t_i that encourage the positions X_i and X_{i+1} to be nearby.

Backtracking search



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

Greedy search



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

Greedy search

Algorithm: greedy search

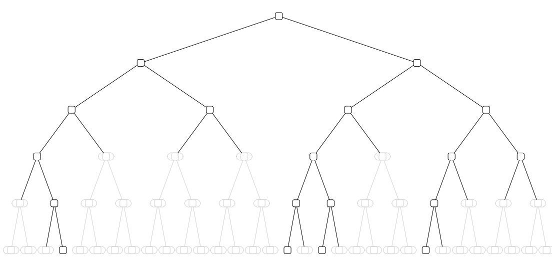
Partial assignment $x \leftarrow \{\}$
For each $i = 1, \ldots, n$:
  Extend:
    Compute weight of each $x_v = x \cup \{X_i : v\}$
  Prune:
    $x \leftarrow x_v$ with highest weight

Not guaranteed to find maximum weight assignment!

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

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

Beam search



Beam size $K = 4$

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

Beam search

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

Algorithm: beam search

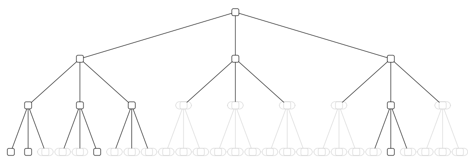
Initialize $C \leftarrow [\{\}]$
For each $i = 1, \ldots, n$:
  Extend:
    $C' \leftarrow \{x \cup \{X_i : v\} : x \in C, v \in \text{Domain}_i\}$
  Prune:
    $C \leftarrow K$ elements of C' with highest weights

Not guaranteed to find maximum weight assignment!

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

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

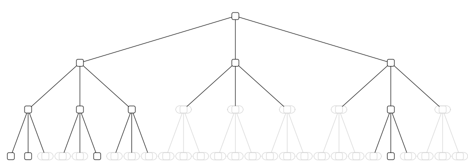
Time complexity



n variables (depth)
Branching factor $b = |\text{Domain}_i|$ → Time: $O(nKb)$
Beam size K

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

Summary



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

Backtracking search $\simeq$ DFS ; Beam search $\simeq$ 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.

Lecture

Dynamic Ordering

Arc Consistency

Beam Search

Local Search

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

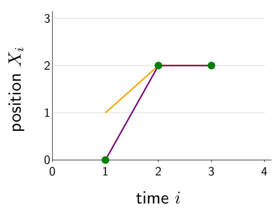
Search strategies

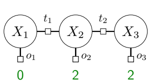
Backtracking/beam search: extend partial assignments

Local search: modify complete assignments

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

Example: object tracking





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

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

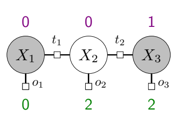
x_3	$o_3(x_3)$
0	0
1	1
2	2

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

[demo]

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

One small step



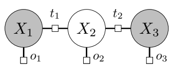
Old assignment: $(0, 0, 1)$; how to improve?

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

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

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

Exploiting locality



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

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

Key idea: locality

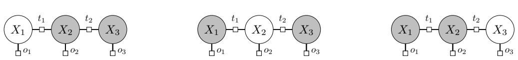
When evaluating possible re-assignments to X_i, only need to consider the factors that depend on X_i.

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

Iterated conditional modes (ICM)

Algorithm: iterated conditional modes (ICM)

Initialize x to a random complete assignment
Loop through $i = 1, \ldots, n$ until convergence:
    Compute weight of $x_v = x \cup \{X_i : v\}$ for each v
    $x \leftarrow x_v$ with highest weight

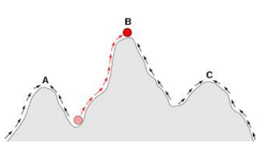


[demo: `iteratedConditionalModes()`]

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

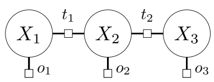
Convergence properties

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



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

Summary

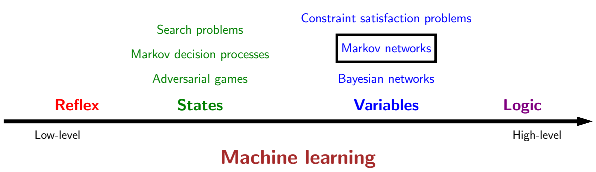


Algorithm	Strategy	Optimality	Time complexity
Backtracking search	extend partial assignments	exact	exponential
Beam search	extend partial assignments	approximate	linear
Local search (ICM)	modify complete assignments	approximate	linear*

* time to do $O(1)$ passes

- 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.

Course plan



- A quick reminder of where we are in the course. We just completed our discussion of CSPs, the first of our variable-based models.
- Markov networks are the second type of variable-based model, which will connect factor graphs with **probability** and serve as a stepping stone on the way to Bayesian networks. That will be the topic for discussion next week.