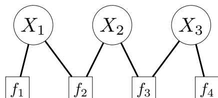
CSPs: beam search

- 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:

$X = (X_1, \dots, X_n)$, 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**:

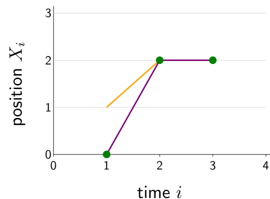
$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

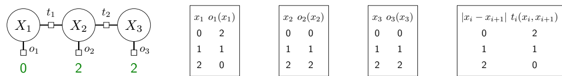
Objective:

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

- 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.

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

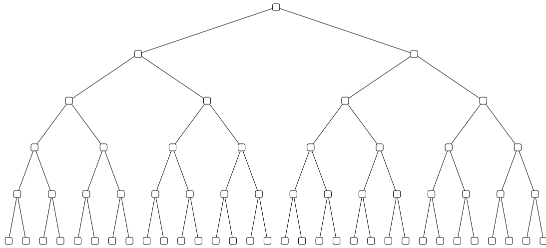
x_3	$o_3(x_3)$
0	0
1	1
2	2

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

[demo]

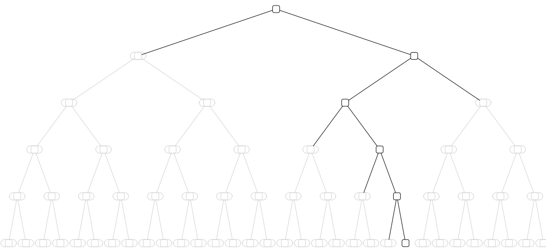
- Recall the object tracking example in which we observe noisy sensor readings 0, 2, 2.
- We have observation factors o_i that encourage the position X_i and the corresponding sensor reading to be nearby.
- We also have transition factors t_i that encourage the positions X_i and X_{i+1} to be nearby.

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

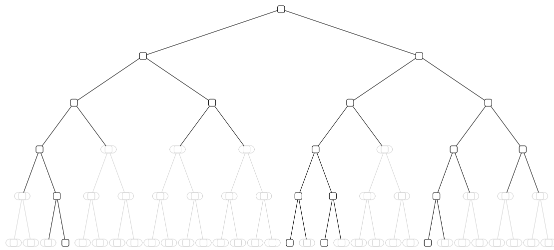
Partial assignment $x \leftarrow \{\}$
For each $i = 1, \ldots, n$:
Extend:
Compute weight of each $x_v = x \cup \{X_i : v\}$
Prune:
$x \leftarrow x_v$ with highest weight

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

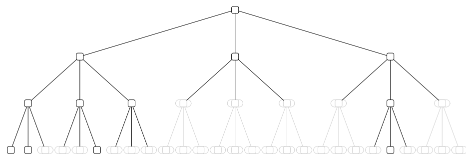
Initialize $C \leftarrow [\{\}]$
For each $i = 1, \ldots, n$:
Extend:
$C' \leftarrow \{x \cup \{X_i : v\} : x \in C, v \in \text{Domain}_i\}$
Prune:
$C \leftarrow K$ elements of C' with highest weights

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.

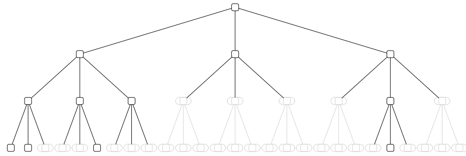
Time complexity



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

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

Summary



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

Backtracking search : DFS :: beam search : pruned BFS

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