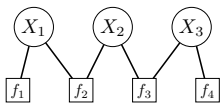
CSPs: dynamic ordering

- 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

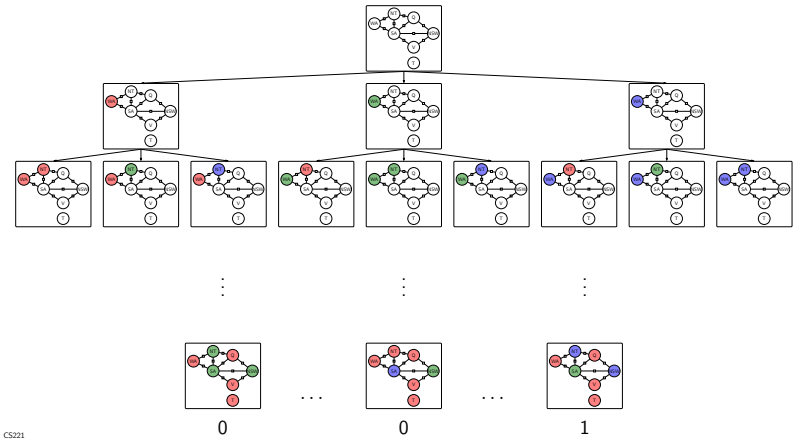
Each **assignment** $x = (x_1, \ldots, x_n)$ has a **weight**:

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

Objective:

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

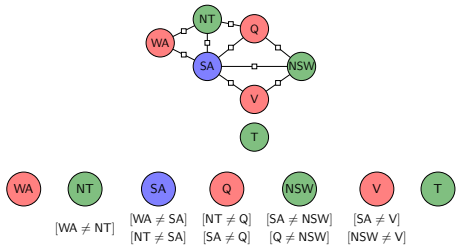
- Recall that a constraint satisfaction problem is defined by a factor graph, where we have a set of variables and a set of factors. Each assignment of values to variables has a weight, and the objective is to find the assignment with the maximum weight.



- 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

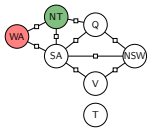
Idea: compute weight of partial assignment as we go



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

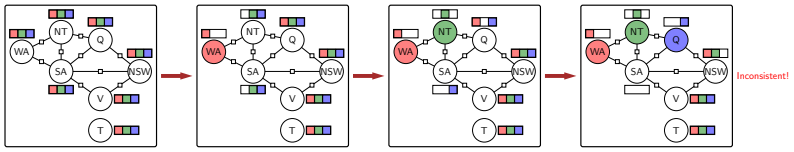
- If x is complete assignment: update best and return
- Choose unassigned **VARIABLE** X_i
- Order **VALUES** Domain_i of chosen X_i
- For each value v in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If $\delta = 0$: continue
  - $\text{Domains}' \leftarrow$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

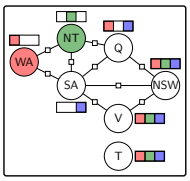
Key idea: forward checking (one-step lookahead)

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



- 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

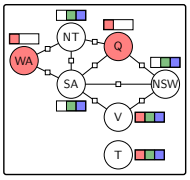
Choose variable that has the smallest domain.

This example: SA (has only one value)

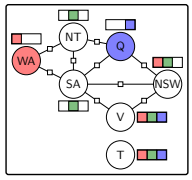
- Now let us look at the problem of choosing an unassigned variable.
- Intuitively, we want to choose the variable which is most constrained, that is, the variable whose domain has the fewest number of remaining valid values (based on forward checking), because those variables yield smaller branching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



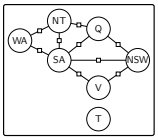
$1 + 1 + 2 = 4$ consistent values

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)

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

- Least constrained value: useful when **all** factors are constraints (all assignment weights are 1 or 0)

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

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

- 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
  - $\text{Domains}' \leftarrow$ Domains via **LOOKAHEAD** (forward checking)
  - If any $\text{Domains}'_i$ is empty: continue
  - Backtrack$(x \cup \{X_i : v\}, w\delta, \text{Domains}')$

- 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.