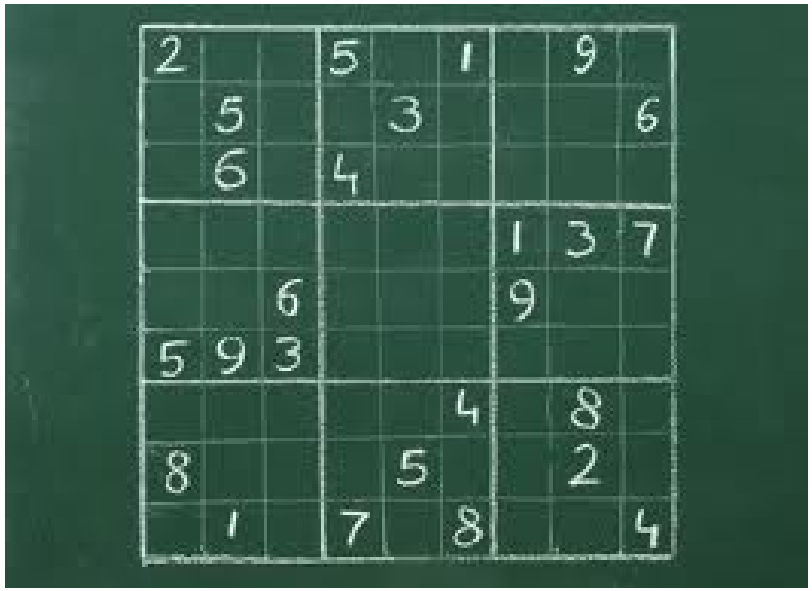
Lecture 11: CSPs I



Announcements

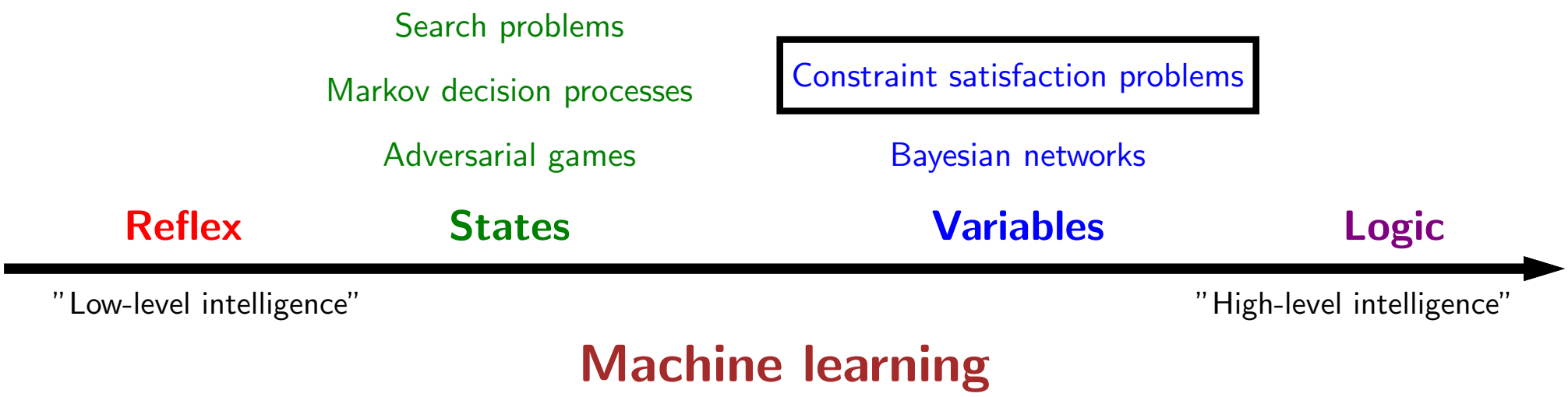
- **Exam**: Please see Piazza for the logistics and resources.
- Homework 5 (pacman) due tomorrow at **11:00pm**.
- Homework 6 (scheduling) is out.

answer in chat

Question

Find two neighboring countries, one that begins with an A and the other that speaks Hungarian.

Course plan


Search problems
Markov decision processes
Adversarial games
Constraint satisfaction problems
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

- We've finished our tour of machine learning and state-based models, which brings us to the midpoint of this course. Let's reflect a bit on what we've covered so far.

Paradigm

Modeling

Inference **Learning**

State-based models

[Modeling]

Framework	search problems	MDPs/games
Objective	minimum cost paths	maximum value policies

[Inference]

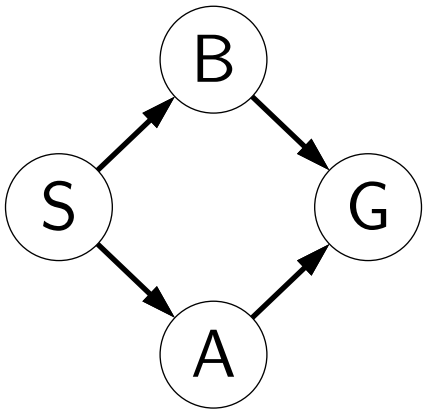
Tree-based	backtracking	minimax/expectimax
Graph-based	DP, UCS, A*	value/policy iteration

[Learning]

Methods	structured Perceptron	Q-learning, TD learning

- **Modeling**: In the context of state-based models, we seek to find minimum cost paths (for search problems) or maximum value policies (for MDPs and games).
- **Inference**: To compute these solutions, we can either work on the search/game tree or on the state graph. In the former case, we end up with recursive procedures which take exponential time but require very little memory (generally linear in the size of the solution). In the latter case, where we are fortunate to have few enough states to fit into memory, we can work directly on the graph, which can often yield an exponential savings in time.
- Given that we can find the optimal solution with respect to a fixed model, the final question is where this model actually comes from. **Learning** provides the answer: from data. You should think of machine learning as not just a way to do binary classification, but more as a way of life, which can be used to support a variety of different models.
- In the rest of the course, modeling, inference, and learning will continue to be the three pillars of all techniques we will develop.

State-based models: takeaway 1



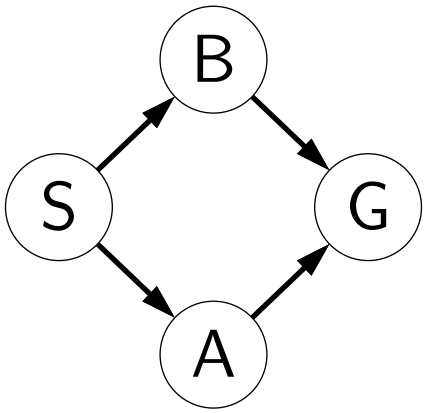
Key idea: specify locally, optimize globally

Modeling: specifies local interactions

Inference: find globally optimal solutions

- One high-level takeaway is the motto: specify locally, optimize globally. When we're building a search problem, we only need to specify how the states are connected through actions and what the local action costs are; we need not specify the long-term consequences of taking an action. It is the job of the inference to take all of this local information into account and produce globally optimal solutions (minimum cost paths).
- This separation is quite powerful in light of modeling and inference: having to worry only about local interactions makes modeling easier, but we still get the benefits of a globally optimal solution via inference which are constructed independent of the domain-specific details.
- We will see this local specification + global optimization pattern again in the context of variable-based models.

State-based models: takeaway 2



Key idea: state

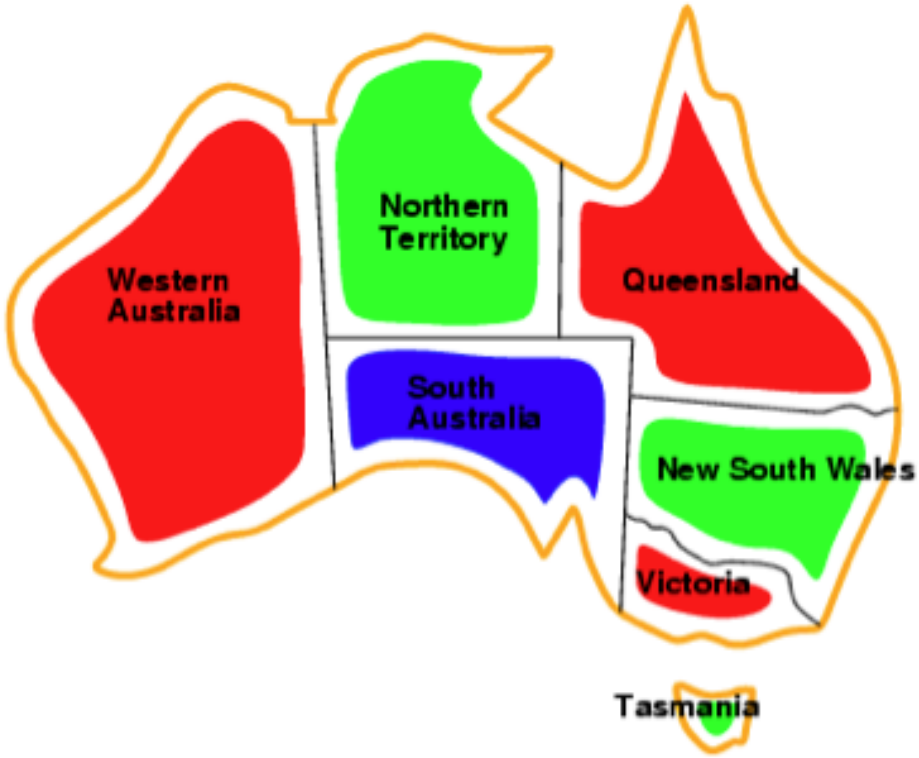
A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Mindset: move through states (nodes) via actions (edges)

- The second high-level takeaway which is core to state-based models is the notion of **state**. The state, which summarizes previous actions, is one of the key tools that allows us to manage the exponential search problems frequently encountered in AI. We will see the notion of state appear again in the context of conditional independence in variable-based models.
- With states, we were in the mindset of thinking about taking a sequence of actions (where order is important) to reach a goal. However, in some tasks, order is irrelevant. In these cases, maybe search isn't the best way to model the task. Let's see an example.

Question: how can we color each of the 7 provinces {red,green,blue} so that no two neighboring provinces have the same color?

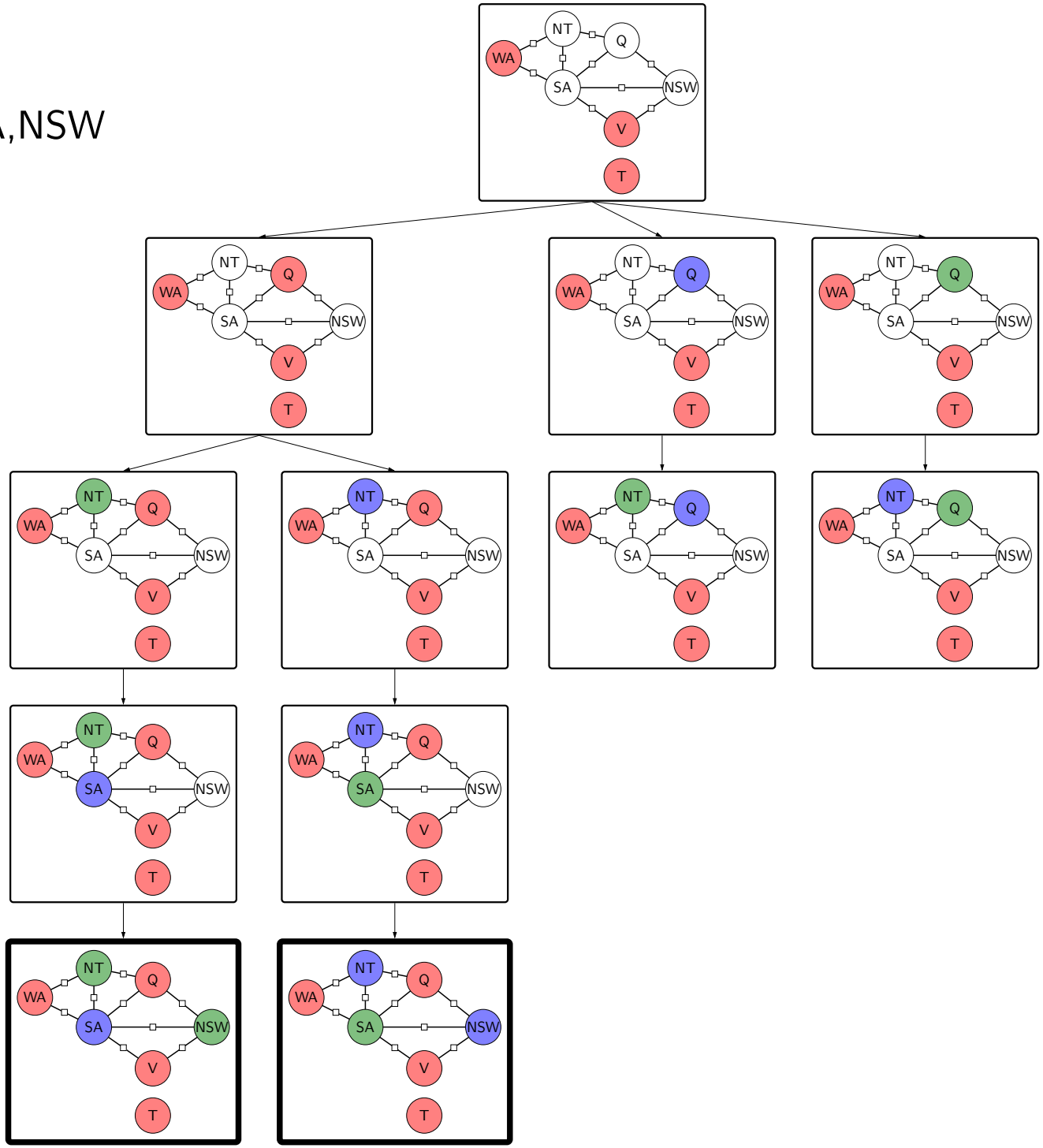
Map coloring



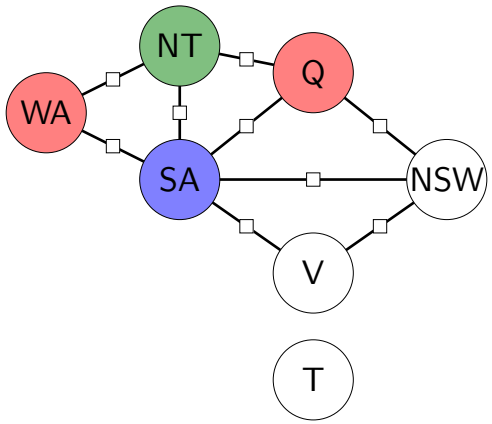
(one possible solution)

Search

WA,V,T,Q,NT,SA,NSW



As a search problem



- State: partial assignment of colors to provinces
- Action: assign next uncolored province a compatible color

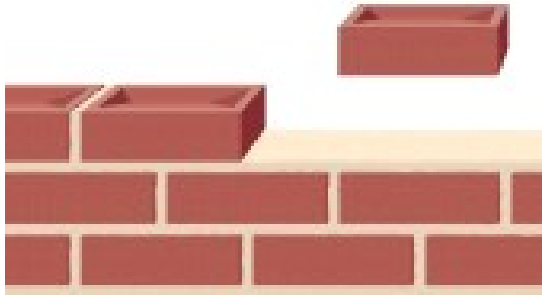
What's missing? There's more problem structure!

- Variable ordering doesn't affect correctness
- Variables are interdependent in a local way

- We can certainly use search to find an assignment of colors to the provinces of Australia. Let's fix an arbitrary ordering of the provinces. Each state contains an assignment of colors to a subset of the provinces (a **partial assignment**), and each action chooses a color for the next unassigned province as long as the color isn't already assigned to one of its neighbors. In this way, all the leaves of the search tree are solutions (18 of them). (In the slide, in the interest of space, we've only shown the subtree rooted at a partial assignment to 3 variables.)
- This is a fine way to solve this problem, and in general, it shows how powerful search is: we don't actually need any new machinery to solve this problem. But the question is: can we do better?
- First, the particular search tree that we drew had several dead ends; it would be better if we could detect these earlier. We will see in this lecture that the fact that **the order in which we assign variables doesn't matter for correctness** gives us the flexibility to dynamically choose a better ordering of the variables. That, with a bit of lookahead will allow us to dramatically improve the efficiency over naive tree search.
- Second, it's clear that Tasmania's color can be any of the three colors regardless of the colors on the mainland. This is an instance of **independence**, and next time, we'll see how to exploit these observations systematically.

Variable-based models

A new framework...



Key idea: variables

- Solutions to problems $\Rightarrow$ assignments to variables (modeling).
- Decisions about variable ordering, etc. chosen by inference.

- With that motivation in mind, we now embark on our journey into variable-based models. Variable-based models is an umbrella term that includes constraint satisfaction problems (CSPs), Markov networks, Bayesian networks, hidden Markov models (HMMs), conditional random fields (CRFs), etc., which we'll get to later in the course. The term graphical models can be used interchangeably with variable-based models, and the term probabilistic graphical models (PGMs) generally encompasses both Markov networks (also called undirected graphical models) and Bayesian networks (directed graphical models).
- The unifying theme is the idea of thinking about solutions to problems as assignments of values to variables (this is the modeling part). All the details about how to find the assignment (in particular, which variables to try first) are delegated to inference. So the advantage of using variable-based models over state-based models is that it's making the algorithms do more of the work, freeing up more time for modeling.
- An apt analogy is programming languages. Solving a problem directly by implementing an ad-hoc program is like using assembly language. Solving a problem using state-based models is like using C. Solving a problem using variable-based models is like using Python. By moving to a higher language, you might forgo some amount of ability to optimize manually, but the advantage is that (i) you can think at a higher level and (ii) there are more opportunities for optimizing automatically.
- Once the different modeling frameworks become second nature, it is almost as if they are invisible. It's like when you master a language, you can "think" in it without constantly referring to the framework.

Roadmap

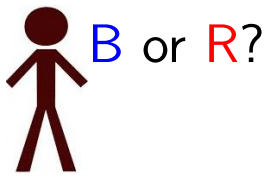
Factor graphs

Dynamic ordering

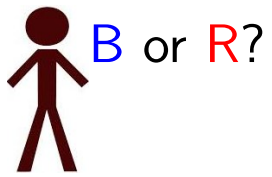
Arc consistency

Modeling

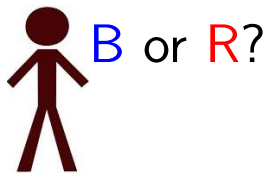
Factor graph (example)

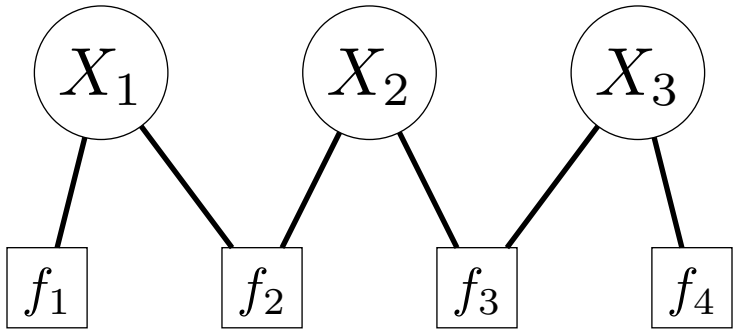


must agree



tend to agree





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

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

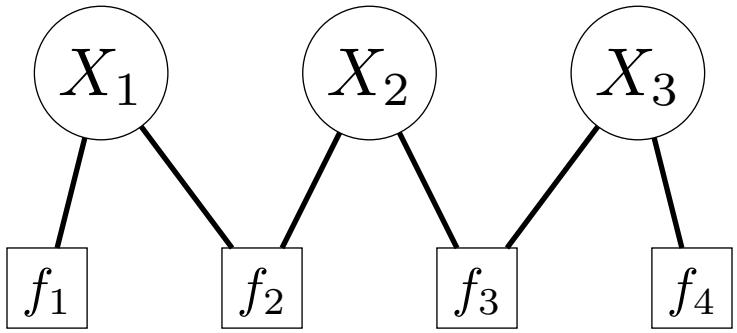
x_3	$f_4(x_3)$
R	2
B	1

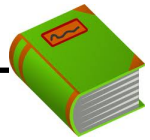
$$f_2(x_1, x_2) = [x_1 = x_2] \qquad f_3(x_2, x_3) = [x_2 = x_3] + 2$$

[demo]

- The most important concept for the next three weeks will be that of a **factor graph**. But before we define it formally, let us consider a simple example.
- Suppose there are three people, each of which will vote for a color, red or blue. We know that Person 1 is leaning pretty set on blue, and Person 3 is leaning red. Person 1 and Person 2 must have the same color, while Person 2 and Person 3 would weakly prefer to have the same color.
- We can model this as a factor graph consisting of three **variables**, X_1, X_2, X_3, each of which must be assigned red (R) or blue (B).
- We encode each of the constraints/preferences as a **factor**, which assigns a non-negative number based on the assignment to a subset of the variables. We can either describe the factor as an explicit table, or via a function (e.g., $[x_1 = x_2]$).

Factor graph





Definition: factor graph

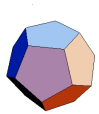
Variables:

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

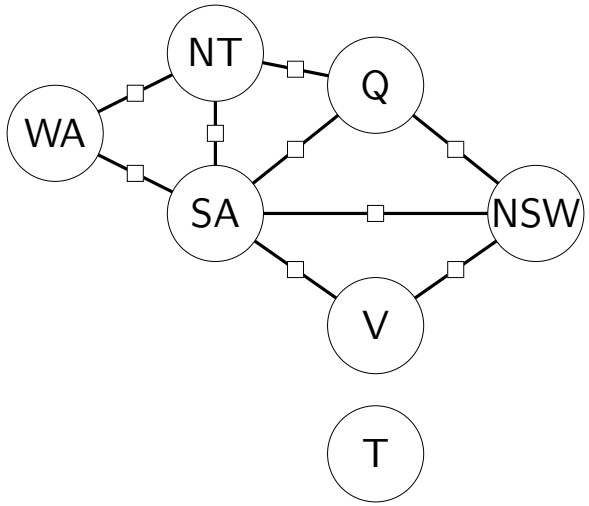
Factors:

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

- Now we proceed to the general definition. a factor graph consists of a set of variables and a set of factors: (i) n variables $X_1, \ldots, X_n$, which are represented as circular nodes in the graphical notation; and (ii) m factors (also known as potentials) $f_1, \ldots, f_m$, which are represented as square nodes in the graphical notation.
- Each variable X_i can take on values in its **domain** Domain_i. Each factor f_j is a function that takes an assignment x to all the variables and returns a non-negative number representing how good that assignment is (from the factor's point of view). Usually, each factor will depend only on a small subset of the variables.



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:

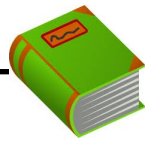
$$f_1(X) = [\text{WA} \neq \text{NT}]$$

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

...

- Notation: we use $[\text{condition}]$ to represent the indicator function which is equal to 1 if the condition is true and 0 if not. Normally, this is written $\mathbf{1}[\text{condition}]$, but we drop the $\mathbf{1}$ for succinctness.

Factors



Definition: scope and arity

Scope of a factor f_j: set of variables it depends on.

Arity of f_j is the number of variables in the scope.

Unary factors (arity 1); **Binary** factors (arity 2).

Example: map coloring

- Scope of $f_1(X) = [\text{WA} \neq \text{NT}]$ is $\{\text{WA}, \text{NT}\}$
- f_1 is a binary factor

- The key aspect that makes factor graphs useful is that each factor f_j only depends on a subset of variables, called the **scope**. The arity of the factors is generally small (think 1 or 2).

Assignment weights (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

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

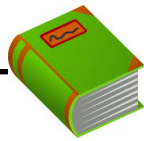
x_3	$f_4(x_3)$
R	2
B	1

x_1	x_2	x_3	Weight
R	R	R	$0 \cdot 1 \cdot 3 \cdot 2 = 0$
R	R	B	$0 \cdot 1 \cdot 2 \cdot 1 = 0$
R	B	R	$0 \cdot 0 \cdot 2 \cdot 2 = 0$
R	B	B	$0 \cdot 0 \cdot 3 \cdot 1 = 0$
B	R	R	$1 \cdot 0 \cdot 3 \cdot 2 = 0$
B	R	B	$1 \cdot 0 \cdot 2 \cdot 1 = 0$
B	B	R	$1 \cdot 1 \cdot 2 \cdot 2 = 4$
B	B	B	$1 \cdot 1 \cdot 3 \cdot 1 = 3$

[demo]

- A factor graph specifies all the local interactions between variables. We wish to find a global solution. A solution is called an **assignment**, which specifies a value for each variable.
- Each assignment is associated with a weight, which is just the product over each factor evaluated on that assignment. Intuitively, each factor contributes to the weight. Note that any factor has veto power: if it returns zero, then the entire weight is irrecoverably zero.
- In this setting, the maximum weight assignment is $(\mathsf{B}, \mathsf{B}, \mathsf{R})$, which has a weight of 4. You can think of this as the optimal configuration or the most likely outcome.

Assignment weights



Definition: assignment weight

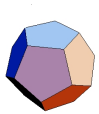
Each **assignment** $x = (x_1, \ldots, x_n)$ has a **weight**:

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

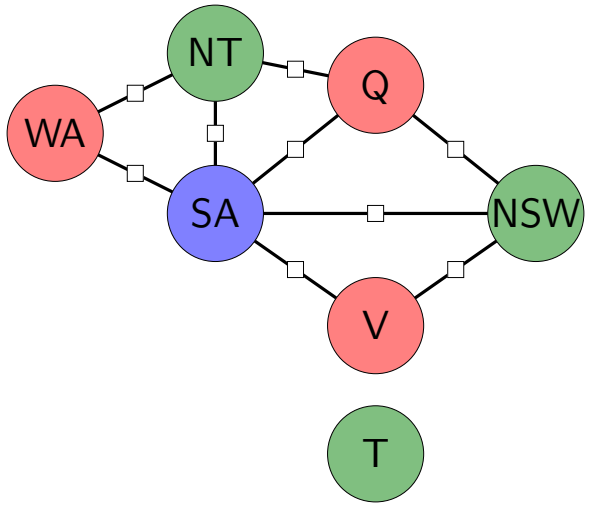
Objective: find the maximum weight assignment

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

- Formally, the **weight** of an assignment x is the product of all the factors applied to that assignment ($\prod_{j=1}^{m} f_j(x)$). Think of all the factors chiming in on their opinion of x. We multiply all these opinions together to get the global opinion.
- Our objective will be to find the **maximum weight assignment**.
- Note: do not confuse the term "weight" in the context of factor graphs with the "weight vector" in machine learning.



Example: map coloring



Assignment:

$x = \{\text{WA} : \text{R}, \text{NT} : \text{G}, \text{SA} : \text{B}, \text{Q} : \text{R}, \text{NSW} : \text{G}, \text{V} : \text{R}, \text{T} : \text{G}\}$

Weight:

$$\text{Weight}(x) = 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

Assignment:

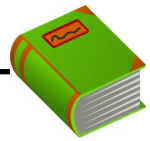
$x' = \{\text{WA} : \text{R}, \text{NT} : \text{R}, \text{SA} : \text{B}, \text{Q} : \text{R}, \text{NSW} : \text{G}, \text{V} : \text{R}, \text{T} : \text{G}\}$

Weight:

$$\text{Weight}(x') = 0 \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0$$

- In the map coloring example, each factor only looks at the variables of two adjacent provinces and checks if the colors are different (returning 1) or the same (returning 0). From a modeling perspective, this allows us to specify local interactions in a modular way. A global notion of consistency is achieved by multiplying together all the factors.
- Again note that the factors are multiplied (not added), which means that any factor has veto power: a single zero causes the entire weight to be zero.

Constraint satisfaction problems



Definition: constraint satisfaction problem (CSP)

A CSP is a factor graph where all factors are **constraints**:

$$f_j(x) \in \{0, 1\} \text{ for all } j = 1, \ldots, m$$

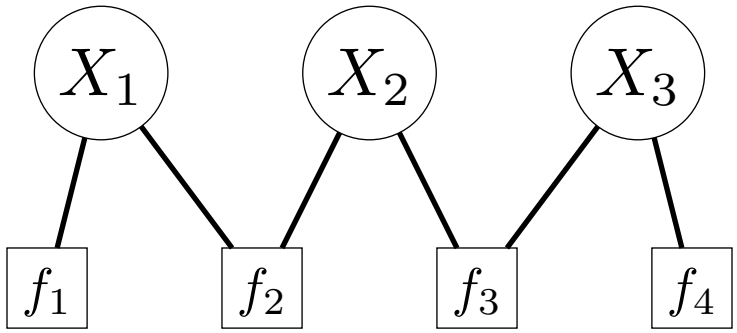
The constraint is satisfied iff $f_j(x) = 1$.

Definition: consistent assignments

An assignment x is **consistent** iff $\text{Weight}(x) = 1$ (i.e., **all** constraints are satisfied).

- Constraint satisfaction problems are just a special case of factor graphs where each of the factors returns either 0 or 1. Such a factor is a **constraint**, where 1 means the constraint is satisfied and 0 means that it is not.
- In a CSP, all assignments have either weight 1 or 0. Assignments with weight 1 are called **consistent** (they satisfy all the constraints), and the assignments with weight 0 are called inconsistent. Our goal is to find any consistent assignment (if one exists).

Summary so far



Factor graph (general)	**CSP** (all or nothing)
variables	variables
factors	constraints
assignment weight	consistent or inconsistent

Roadmap

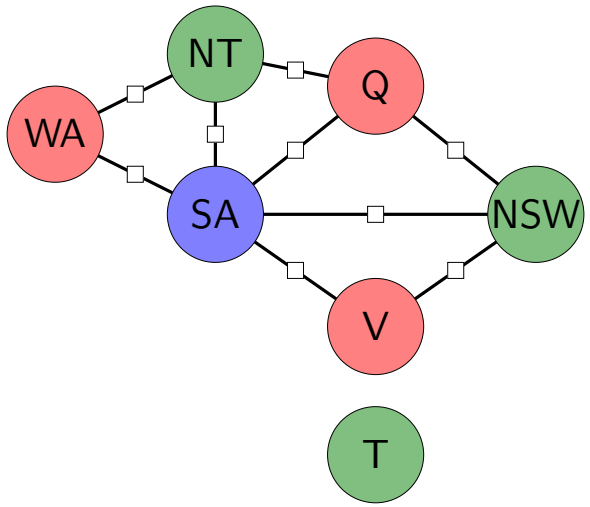
Factor graphs

Dynamic ordering

Arc consistency

Modeling

Extending partial assignments



WA NT SA Q NSW V T

[WA ≠ NT]

[WA ≠ SA] [NT ≠ SA]

[NT ≠ Q] [SA ≠ Q]

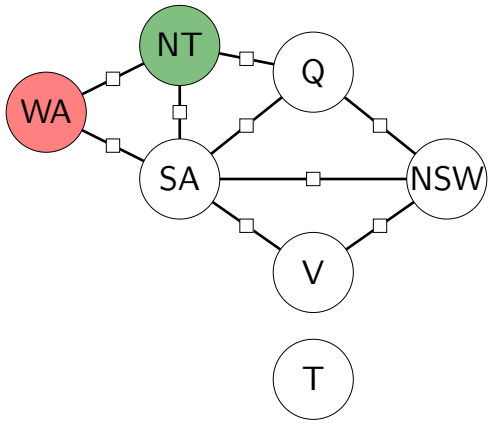
[SA ≠ NSW] [Q ≠ NSW]

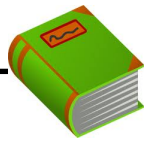
[SA ≠ V] [NSW ≠ V]

- The general idea, as we've already seen in our search-based solution is to work with **partial assignments**. We've defined the weight of a full assignment to be the product of all the factors applied to that assignment.
- We extend this definition to partial assignments: The weight of a partial assignment is defined to be the product of all the factors 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 defined to be the original 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

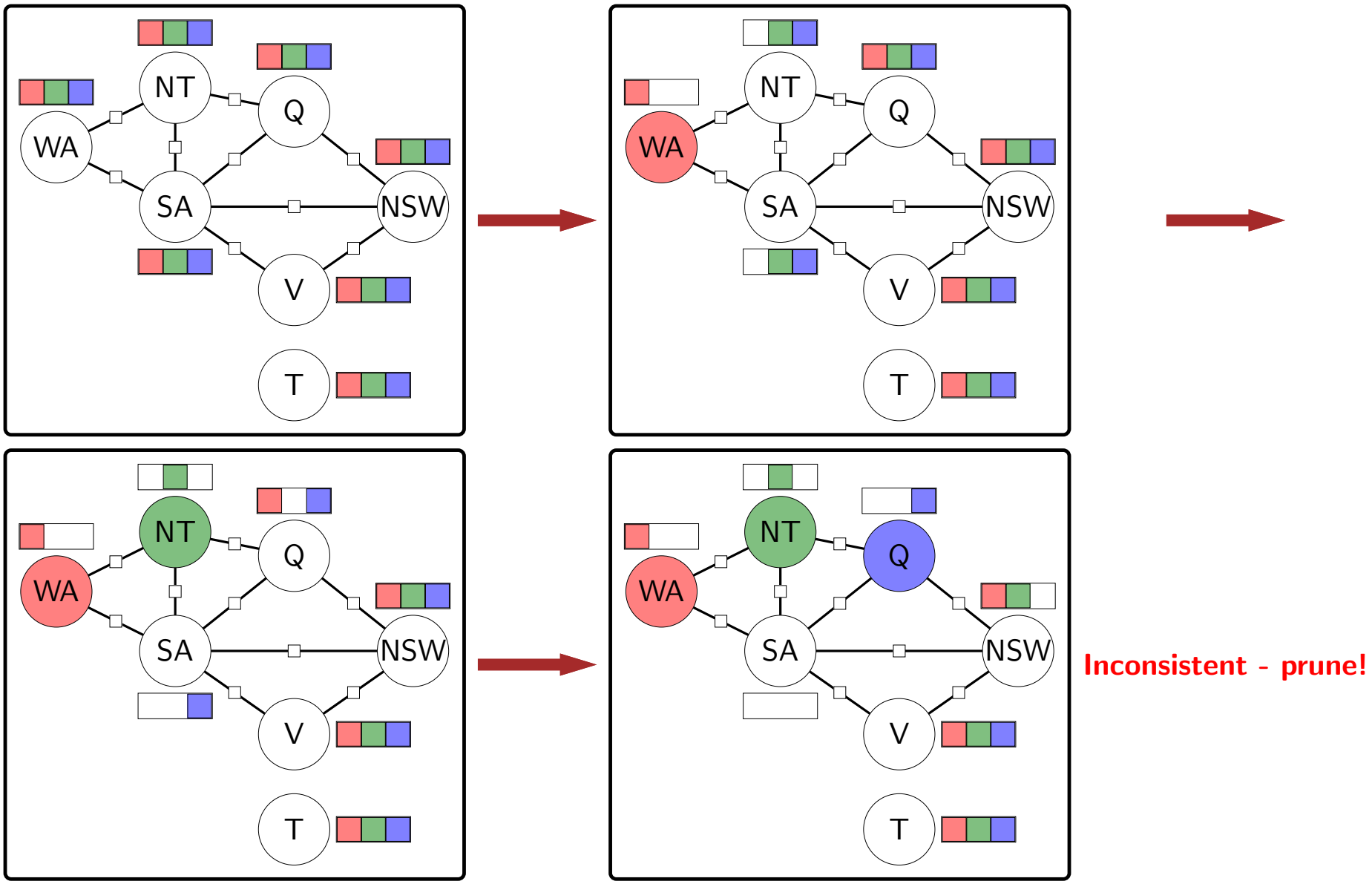
Algorithm: backtracking search

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**
  - 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 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 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 (example)

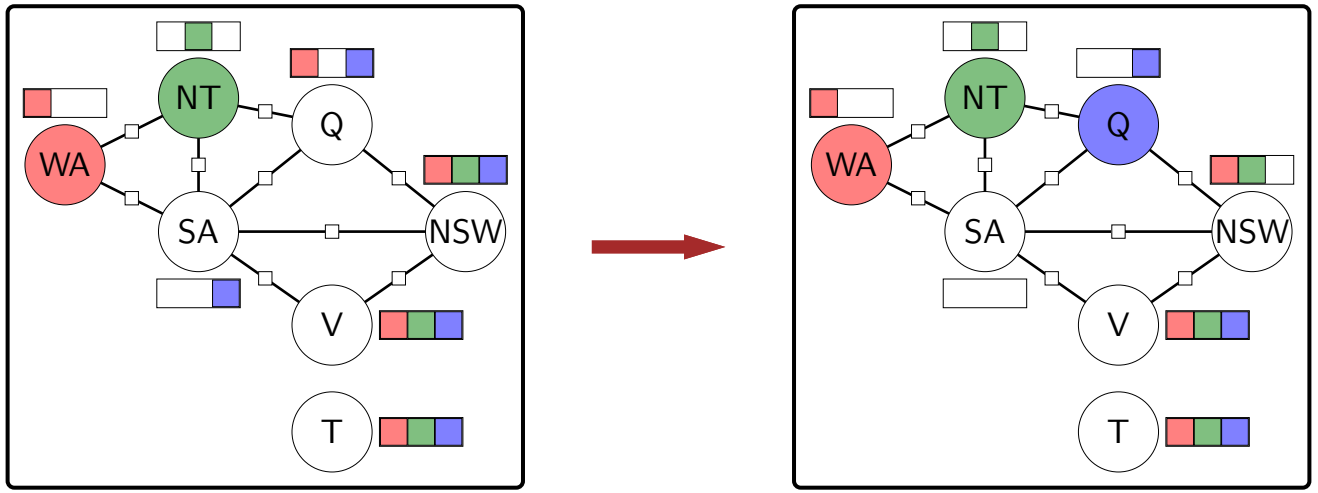


- First, we will look at **forward checking**, which is a way to perform a one-step lookahead. The idea is that as soon as we assign a variable (e.g., WA = R), we can pre-emptively remove inconsistent values from the domains of neighboring variables (i.e., those that share a factor).
- If we keep on doing this and 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.

Lookahead: forward checking

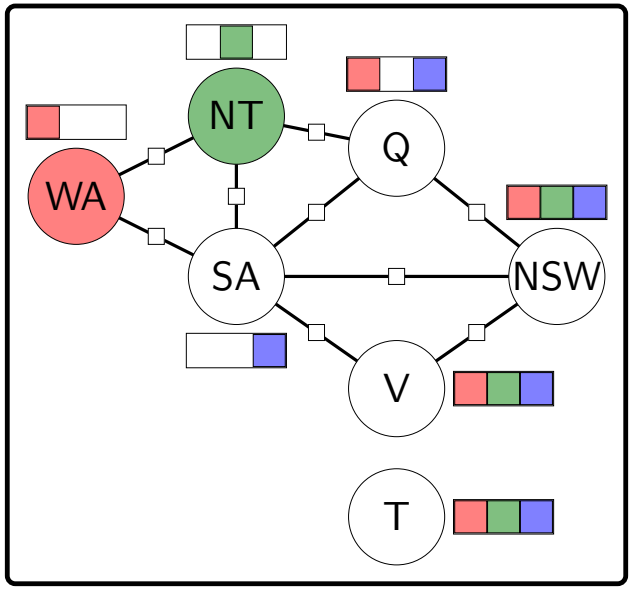
Key idea: forward checking (one-step lookahead)

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



- When unassigning a variable, remember to restore the domains of its neighboring variables!
- The simplest way to implement this is to make a copy of the domains of the variables before performing forward checking. This is foolproof, but can be quite slow.
- A fancier solution is to keep a counter (initialized to be zero) c_{iv} for each variable X_i and value v in its domain. When we remove a value v from the domain of X_i, we increment c_{iv}. An element is deemed to be "removed" when $c_{iv} > 0$. When we want to un-remove a value, we decrement c_{iv}. This way, the remove operation is reversible, which is important since a value might get removed multiple times due to multiple neighboring variables.
- Later, we will look at arc consistency, which will allow us to lookahead even more.

Choosing an unassigned variable



Which variable to assign next?

Key idea: most constrained variable

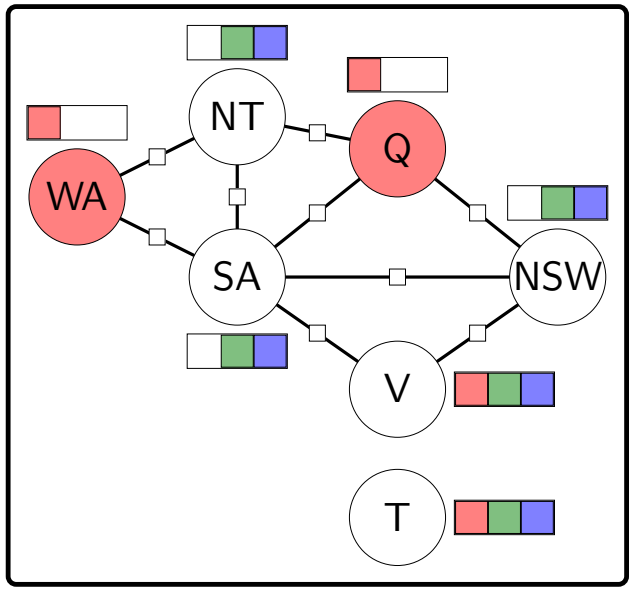
Choose variable that has the fewest consistent values.

This example: SA (has only one value)

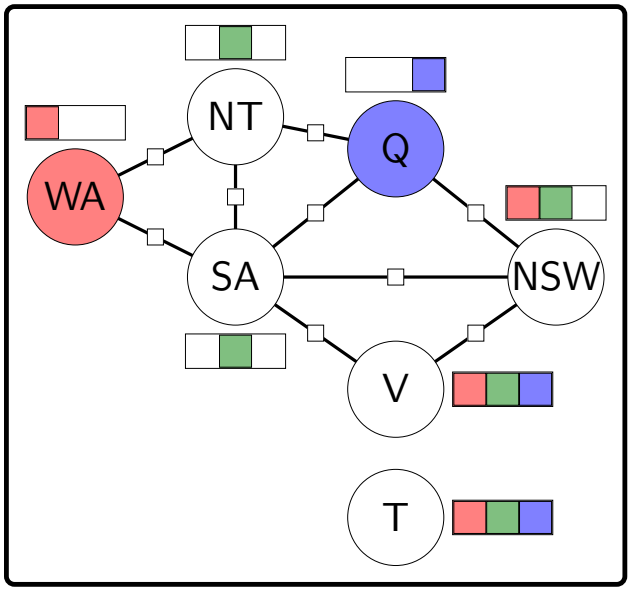
- Now let us look at the problem of choosing an unassigned variable. Intuitively, we want to choose the variable which is most constrained, that is, the variable whose domain has the fewest number of remaining valid values (based on forward checking), because those variables yield smaller branching factors.

Order 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. Here 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
- Choosing value most likely to lead to solution

- The most constrained variable and the least constrained value heuristics might seem conflicting, but there is a good reason for this superficial difference.
- An assignment involves **every** variable whereas for each variable we only need to choose **some** value. Therefore, for variables, we want to try to detect failures early on if possible (because we'll have to confront those variables sooner or later), but for values we want to steer away from possible failures because we might not have to consider those other values.

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: need to actually prune domains to make heuristics useful!

- Most constrained variable is useful for finding maximum weight assignments in any factor graph as long as there are some factors which are constraints, 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 (CSPs). 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.

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$(x \cup \{X_i : v\}, w\delta, \text{Domains}')$

Roadmap

Factor graphs

Dynamic ordering

Arc consistency

Modeling

Arc consistency

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

Example: numbers

Before enforcing arc consistency on X_i:

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

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

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

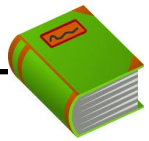
After enforcing arc consistency on X_i:

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

[whiteboard]

- 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



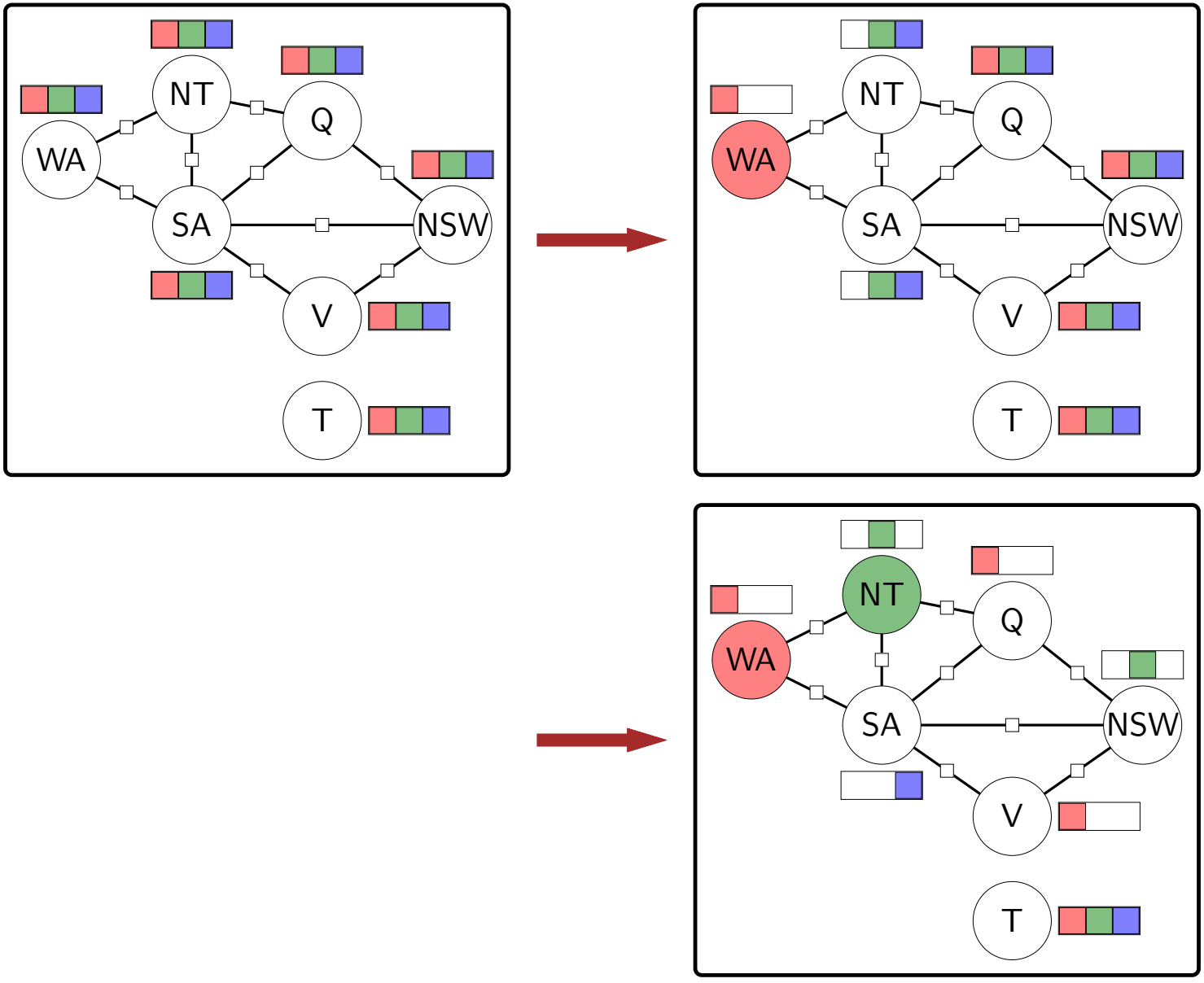
Definition: arc consistency

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.

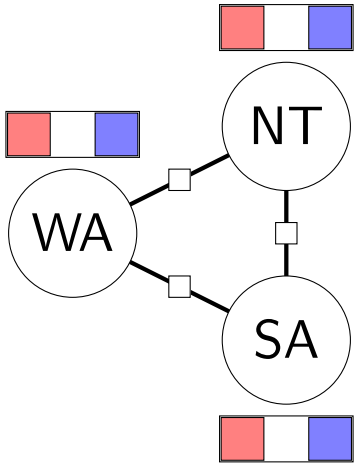
While set is non-empty:

- 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.

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

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

Factor graphs

Dynamic ordering

Arc consistency

Modeling

Example: LSAT question

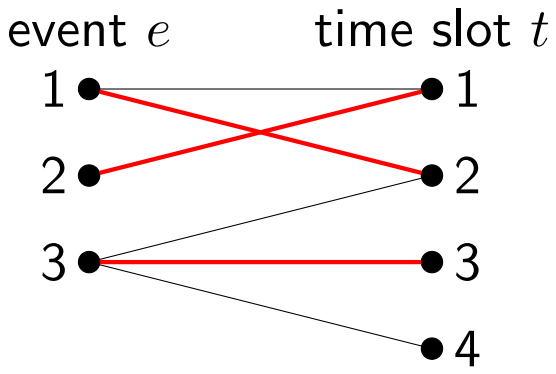
Three sculptures (A, B, C) are to be exhibited in rooms 1, 2 of an art gallery.

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)

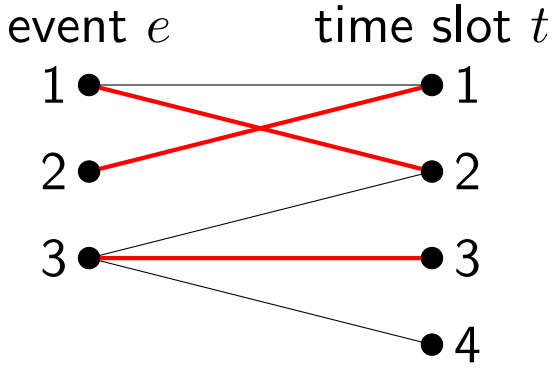


Setup:

- 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, and the purpose of this example is to show two reasonable ways to do it.

Example: event scheduling (section)

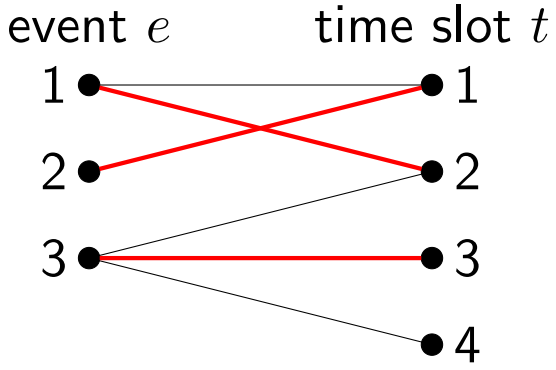


CSP formulation 1:

- Variables: for each event e, $X_e \in \{1, \ldots, T\}$
- Constraints (only one event per time slot): for each pair of events $e \neq e'$, enforce $[X_e \neq X_{e'}]$
- Constraints (only scheduled allowed times): for each event e, enforce $[(e, X_e) \in A]$

- The first formulation is perhaps the more natural one. We make a variable X_e for each event, whose value will be the time slot that the event is scheduled into. Since each variable can only take on one value, we automatically satisfy the requirement that every event must be put in exactly one time slot.
- However, we need to make sure no two events end up in the same time slot. To do this, we can create a binary constraint between every pair of distinct event variables X_e and X'_e that enforces their values to be different ($X_e \neq X_{e'}$).
- Finally, to deal with the requirement that an event is scheduled only in allowed time slots, we just need to add a unary constraint for each variable saying that the time slot X_e that's chosen for that event is allowed.
- Note that we end up with E variables with domain size T, and $O(E^2)$ binary constraints.

Example: event scheduling (section)

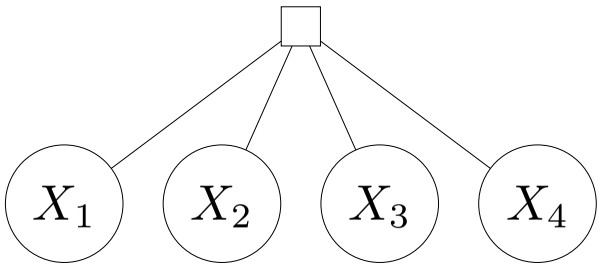


CSP formulation 2:

- Variables: for each time slot t, $Y_t \in \{1, \ldots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event e, enforce $[Y_t = e$ for exactly one $t]$
- Constraints (only schedule allowed times): for each time slot t, enforce $[Y_t = \emptyset$ or $(Y_t, t) \in A]$

- Alternatively, we can take the perspective of the time slots and ask which event was scheduled in each time slot. So we introduce a variable Y_t for each time slot t which takes on a value equal to one of the events or none ($\emptyset$).
- Unlike the first formulation, we don't get for free the requirement that each event is put in exactly one time slot. To add it, we introduce E constraints, one for each event. Each constraint needs to depend on all T variables and check that the number of time slots t which have event e assigned to that slot ($Y_t = e$) is exactly 1.
- On the other hand, the requirement that each time slot has at most one event assigned to it we get for free, since each variable takes on exactly one value.
- Finally, we add T constraints, one for each time slot t enforcing that if there was an event scheduled there ($Y_t \neq \emptyset$), then it better be allowed according to A.
- With this formulation, we have T variables with domain size $E+1$, and E T-ary constraints. We will show shortly that each T-ary constraints can be converted into $O(T)$ binary constraints with $O(T)$ variables. Therefore, the resulting formulation has T variables with domain size $E+1$, $O(ET)$ variables with domain size 2 and $O(ET)$ binary constraints.
- Which one is better? Since $T \gg E$ is required for the existence of a consistent solution, the first formulation is better. If the problem were modified so that not all events had to be scheduled and $T \ll E$, then the second formulation would be better.

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:

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

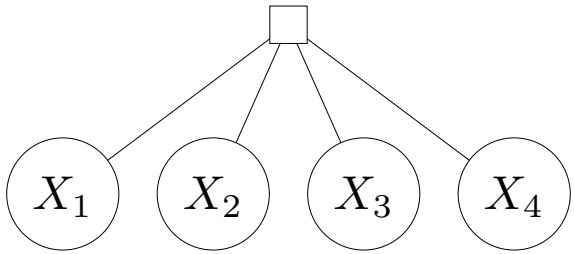
$$\text{Weight}(\{X_1 : 0, X_2 : 1, X_3 : 0, X_4 : 0\}) = 1$$

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

What if inference only take unary/binary factors?

- 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.

N-ary constraints: attempt 1 (section)



Key idea: auxiliary variable

Auxiliary variables hold intermediate computation.

Factors:

Initialization: $[A_0 = 0]$

Processing: $[A_i = A_{i-1} \vee X_i]$

Final output: $[A_4 = 1]$

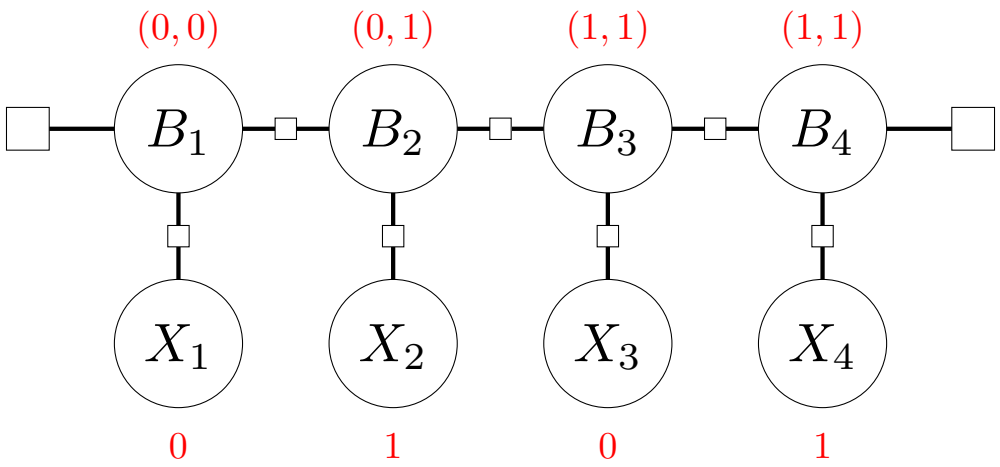
i	0	1	2	3	4
X_i:		0	1	0	1
A_i:	0	0	1	1	1

Still have factors involving 3 variables...

- The key idea is to break down the computation of the n-ary constraint into n simple steps. As a first attempt, let's introduce an auxiliary variable A_i for $i = 1, \ldots, n$ which represents the OR of variables $X_1, \ldots, X_i$. Then we can write a simple recurrence that updates A_i with A_{i-1}. The constraint $[A_n = 1]$ enforces that the OR of all the variables is 1.
- It is important to note that while our intuitions are based on procedurally computing A_i's, one after the other, these computations are actually represented declaratively as constraints in the CSP.
- We have eliminated the massive n-ary constraint with ternary constraints (depending on A_i, A_{i-1}, X_i). Can we replace the ternary constraint with unary and binary constraints?

N-ary constraints: attempt 2 (section)

Key idea: pack A_{i-1} and A_i into one variable B_i



Variables: B_i is (pre, post) pair from processing X_i

Factors:

Initialization: $[B_1[1] = 0]$

Processing: $[B_i[2] = B_i[1] \vee X_i]$

Final output: $[B_4[2] = 1]$

Consistency: $[B_{i-1}[2] = B_i[1]]$

- The key idea to turn the ternary constraint $[A_i = A_{i-1} \vee X_i]$ into a binary constraint is to merge A_{i-1} and A_i into one variable, represented as one variable B_i. The variable B_i will represent a pair of booleans, where $B_i[1]$ represents A_{i-1} and $B_i[2]$ represents A_i.
- Now, the ternary constraint is just a binary constraint: $[B_i[2] = B_i[1] \vee X_i]$!
- However, note that A_{i-1} is represented twice, both in B_i and B_{i-1}. So we need to add another binary constraint to enforce that the two are equal: $[B_{i-1}[2] = B_i[1]]$.
- The initialization and final output factors are the same as before.

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)]

- Now 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

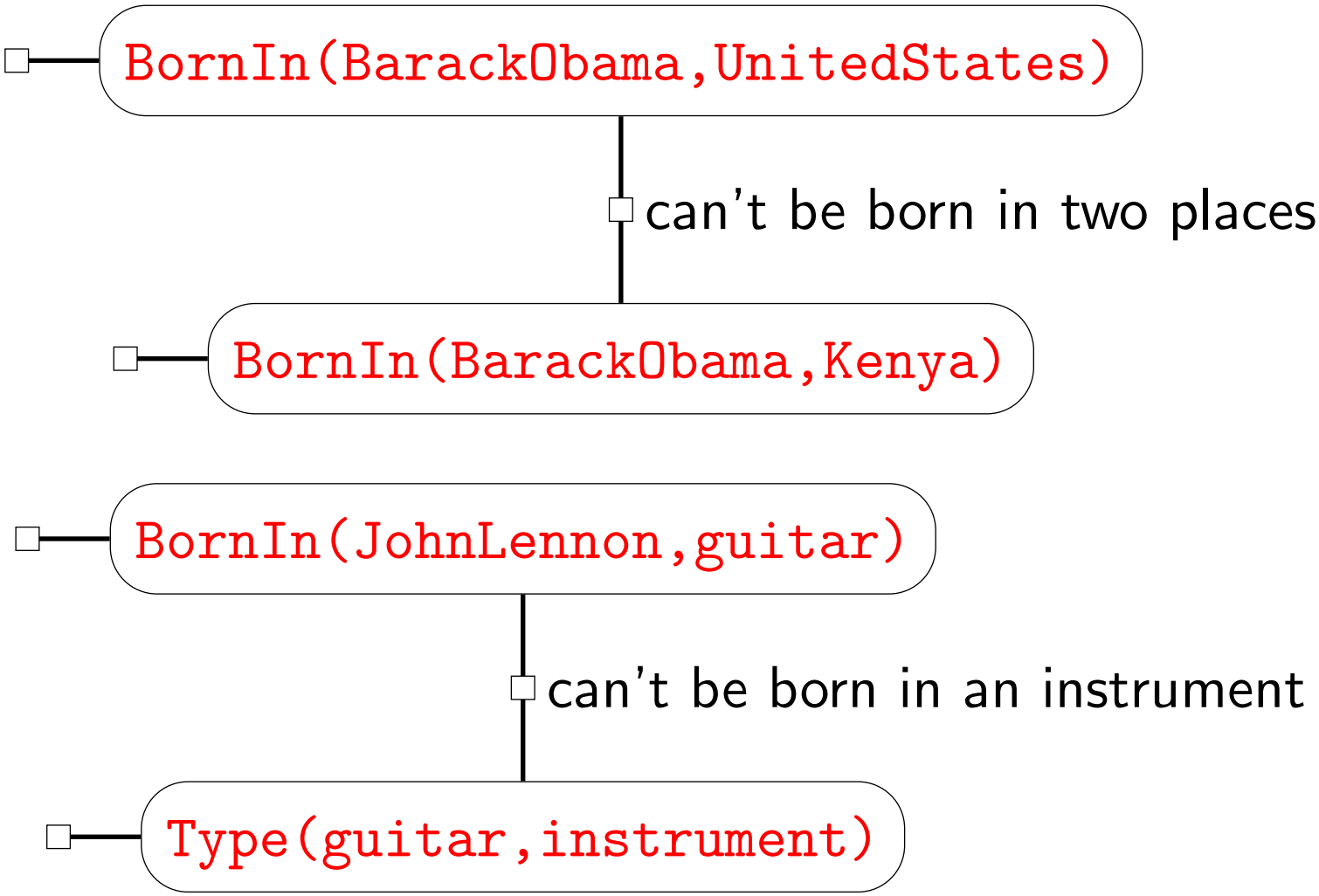
Typical predictions of classifiers:

`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?

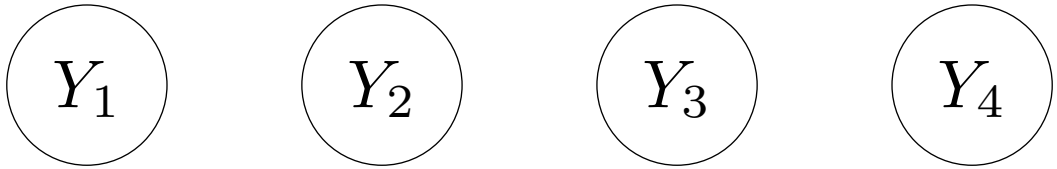
- State-of-the-art methods typically use machine learning, casting it as a classification problem. However, relation extraction is a very difficult problem, and even the best systems today often fail, producing nonsensical facts.
- A key observation is that these classification decisions are not independent, 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

$$[Y_i \text{ consistent with } Y_{i'}]$$

- 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.

Summary

- Factor graphs: modeling framework (variables, factors)
- Key property: ordering decisions pushed to algorithms
- Algorithms: backtracking search + dynamic ordering + lookahead
- Modeling: lots of possibilities!