

Constraint Satisfaction Problems

Overview

State based model review

State-based models

[Modeling]

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

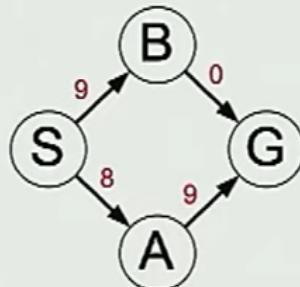
[Inference]

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

[Learning]

Methods	structured Perceptron	Q-learning, TD learning
----------------	-----------------------	-------------------------

State-based models: takeaway 1

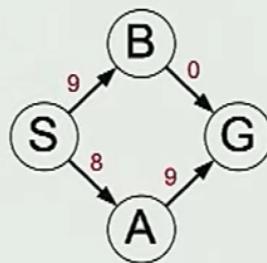


Key idea: specify locally, optimize globally

Modeling: specifies local interactions

Inference: find globally optimal solutions

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)

For the example we had, there are some constraints and you don't have to worry about the order(it doesn't matter)

The Color State Example



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

The order here doesn't matter

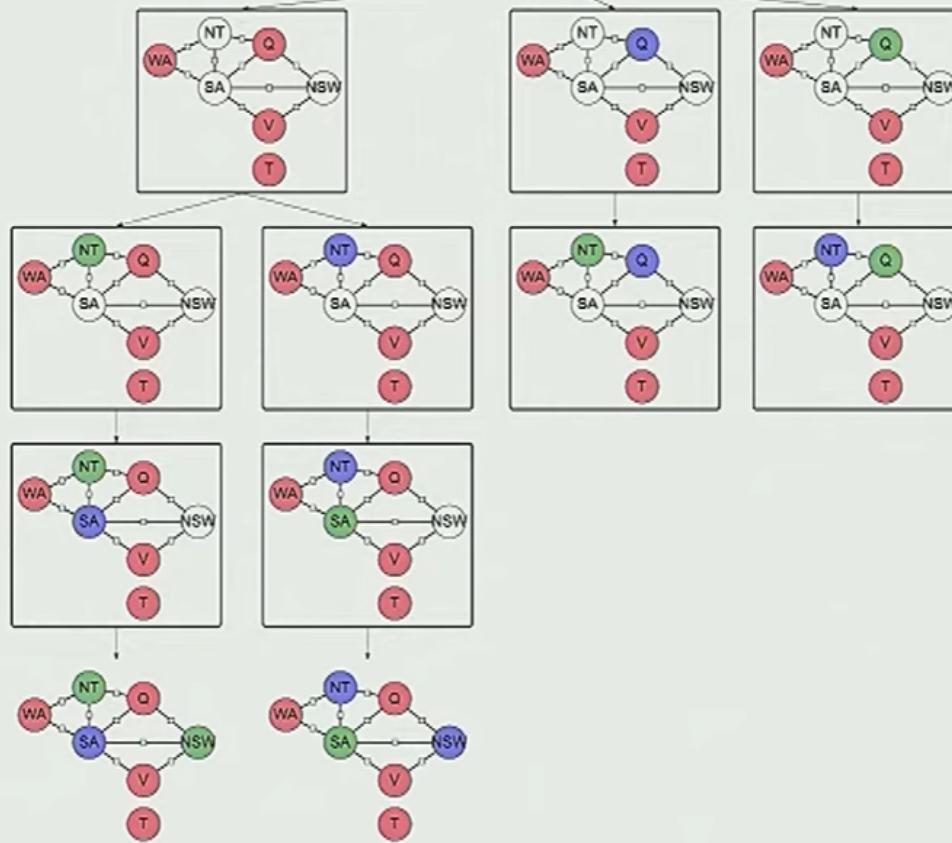
Map coloring



(one possible solution)

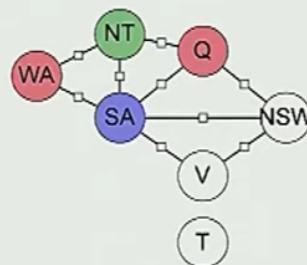
Search

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



We can definitely consider this problem as a search problem, with different states And we give it an initial state Let's say somehow we start with WA, V, T as red, and try to search for solutions for the whole problem

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

So state based model seems too complicated for this, we can try to make things easier

Variable-based models

A new framework...



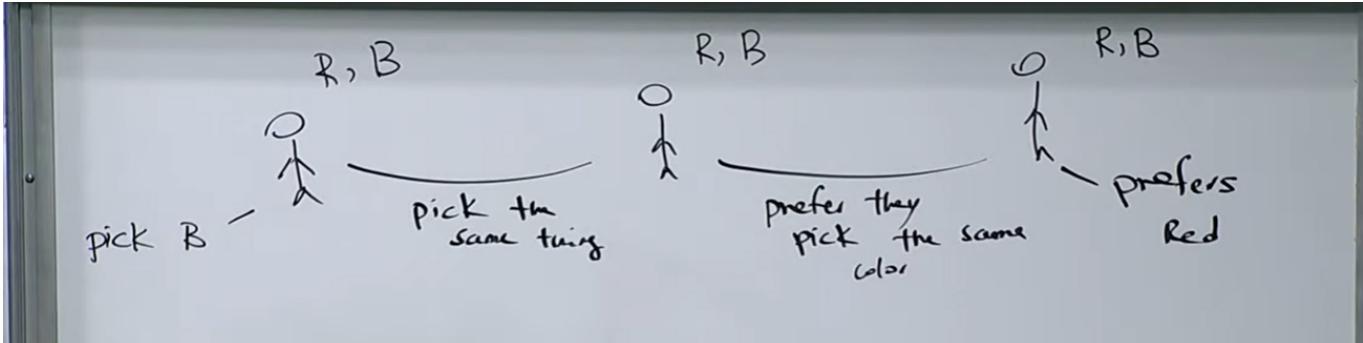
Key idea: variables

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

A higher level of abstraction

Factor graphs

So I have three people, each of them are gonna choose a color either red or blue, and we have a bunch of constraints



Factor graphs are gonna have some number of variables

Factor graph (example)



B or R?

must agree

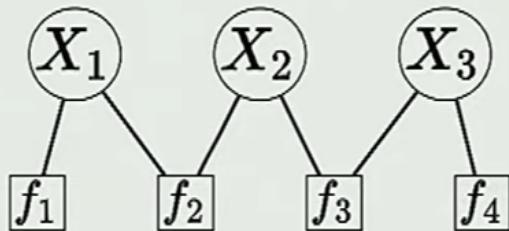


B or R?

tend to agree



B or R?



Factor of X1 is a function of X1, tells me how happy I would be if X1 takes value red or blue

Model the problem

Factor graphs

- Variables: X1, X2, X3
 - For each variable, it's gonna belong to some domain {R, B}
 - Assignment: Assign values to variables
- Factors: reflects the constraints that we need to satisfy
 - Scope of factor: set of variables a factor depend on
 - Arity of a factor: number of variables in the scope

Factor graph (example)

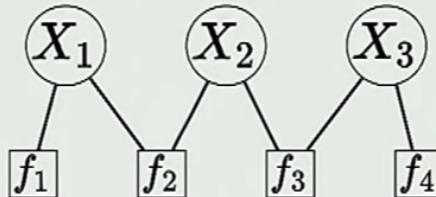
B or R?

must agree

B or R?

tend to agree

B or R?



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

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

[X1 = X2]: An indicator function returns 1 if the statement is true, and 0 otherwise In the factor graph, 0 means I absolutely do not wanna choose this For prefer, you can add some values after the indicator function

Q: does the factor value matter? A: Yes, but in this class we will not focus on this

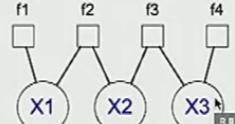
Examples: [vote] [csp] [pair] [chain] [track] [alarm] [med] [dep] [delay] [min] [new]
[\[Background\]](#) [\[Documentation\]](#)

```
// 3 people voting R or B
variable('X1', ['R', 'B'])
variable('X2', ['R', 'B'])
variable('X3', ['R', 'B'])

factor('f1', 'X1', function(x1) {
  return x1 == 'B';
})
factor('f2', 'X1 X2', function(x1, x2) {
  return x1 == x2;
})
factor('f3', 'X2 X3', function(x2, x3) {
  return x2 == x3 ? 3 : 2;
})
factor('f4', 'X3', function(x3) {
  return x3 == 'R' ? 2 : 1;
})

maxVariableElimination()
```

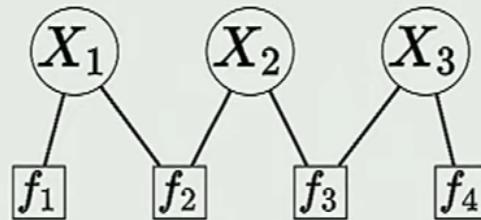
Query: Value of X1,X2,X3 in $\arg \max_x$ Weight(x)
Algorithm: variable elimination (max)



Start of algorithm. Click "Step" to step through it.

Formal Definition

Factor graph



Definition: factor graph

Variables:

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

Factors:

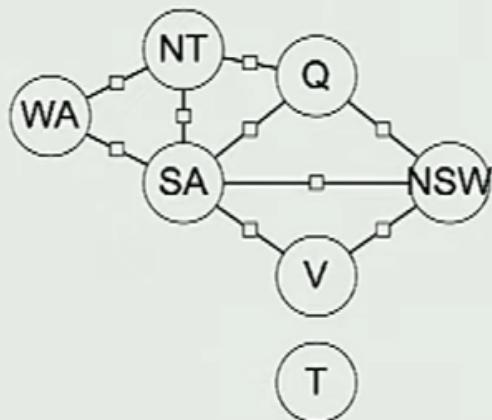
f_1, \dots, f_m , with each $f_j(X) \geq 0$

So more formally, A factor graph is

- A set of variables X_1 to X_n , and each of these variables lies on some domain (in this case R or B)
- And factors from f_1 to f_m , which are functions over X that is going to be ≥ 0 It tells us what are the things that we really want



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:

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

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

...

So the factors here are just telling don't give same color to two neighbours

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

Assignment weights

So factor tells us how happy we are for the single assignment. The weight tells us how good the whole assignment is.

the weight can just be the product of all the factors

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	$1 \cdot 1 \cdot 3 \cdot 1 = 3$

Assignment weights



Definition: assignment weight

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

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

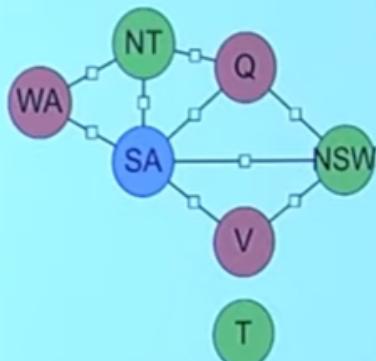
Objective: find the maximum weight assignment

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

Objective: Find the x that maximize the weight



Example: map coloring



Assignment:

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

Weight:

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

Assignment:

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

Weight:

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

19

In this example, the indicator function is just 0 and 1 If we find the solution then the weight is gonna be 1 If not then it's gonna be 0

Constarint satisfaction problems

CSP is just factor graph where all factor function value is either 0 or 1 When we find an assignment where weight = 1, this means we find an assignment which satisfies all the constraints **iff**: If and only if

Constraint satisfaction problems



Definition: constraint satisfaction problem (CSP)

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

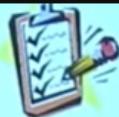
$$f_j(x) \in \{0, 1\} \text{ for all } j = 1, \dots, m$$

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

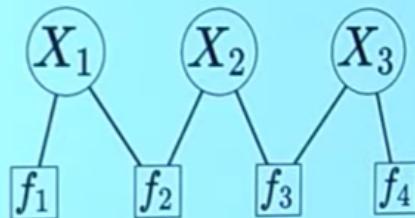


Definition: consistent assignments

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



Summary so far



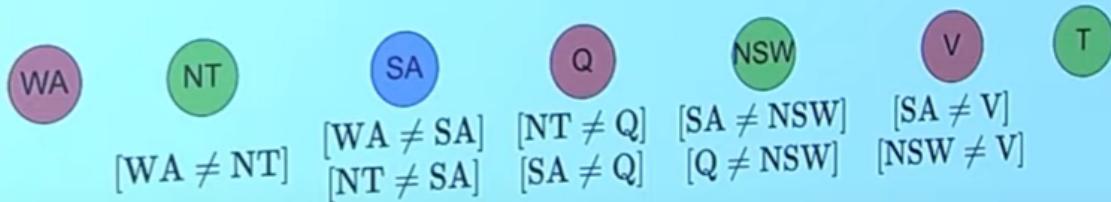
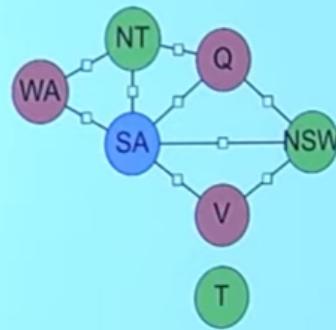
Factor graph (general)
variables
factors
assignment weight

CSP (all or nothing)
variables
constraints
consistent or inconsistent

Dynamic ordering

Let go and talk about how we gonna solve this How to find an assignment to achieve consistency

Extending partial assignments

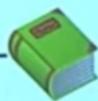
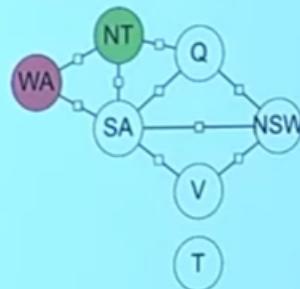


What we might solve this in a human way: We pick some initial value and go with some order, try to find assignments satisfy the constraints

Dependent Factors

Dependent factors

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



Definition: dependent factors

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

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

$D(x, xi)$ x is the assignment xi is the new variable I am picking And it's gonna return a set of factors that depends on x and xi

The Dependent Factor allow us to think about the next thing we should be worried about

Backtracking Search

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:

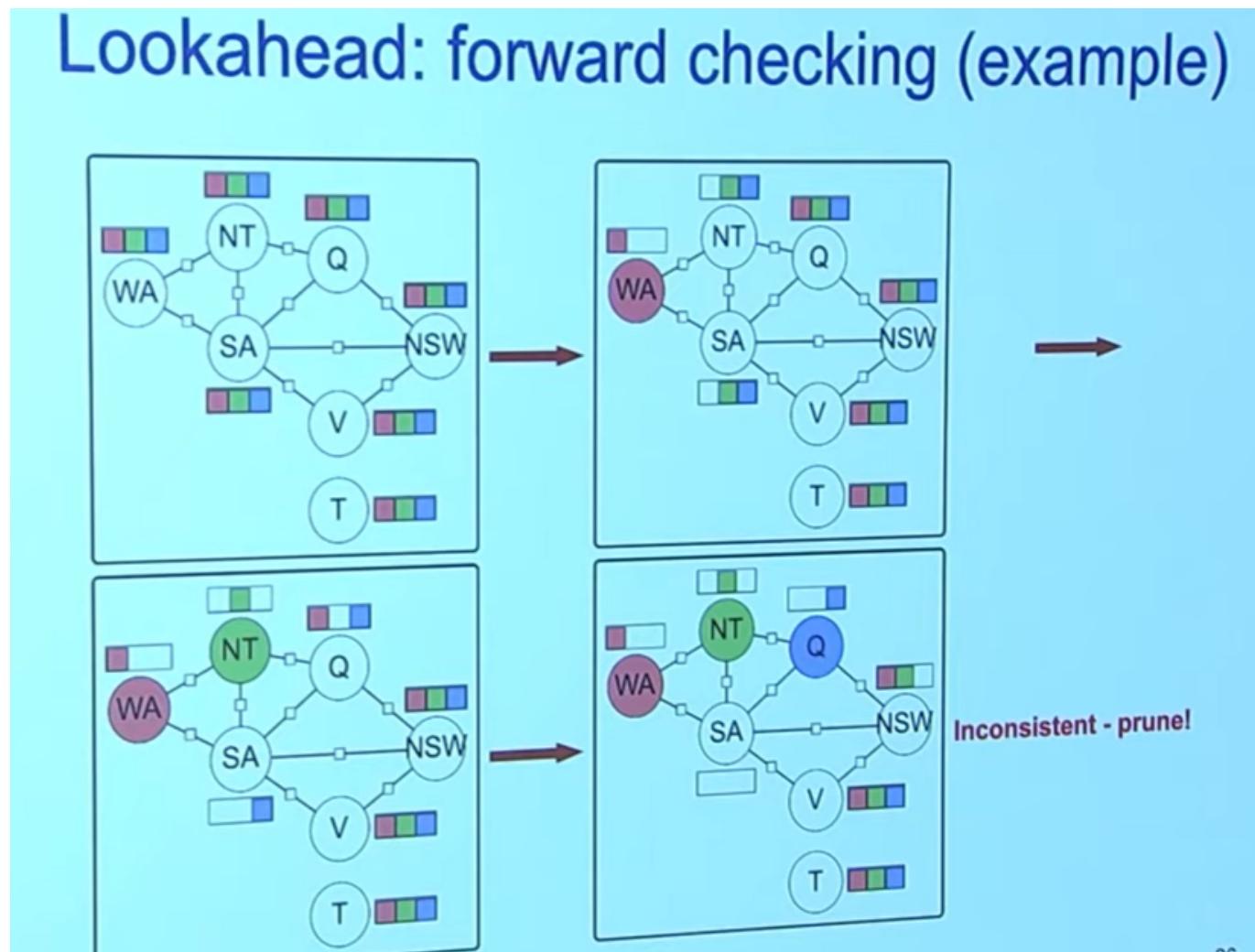
$$\bullet \delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$$

- If $\delta = 0$: continue
- Domains' \leftarrow Domains via LOOKAHEAD
- Backtrack($x \cup \{X_i : v\}, w\delta, \text{Domains}'$) Stanford

Backtrack($x, w, \text{domains}$)
 x: Partial assignment we have
 w: Weight we have so far
 domains: domain of the variables that I have so far

- If X is complete then we are happy
 - If X is already a complete assignment, then we'll update the best thing we have and return (base case)
- Choose unassigned variable X_i
 - Figure put what is the next variable I am gonna pick
- Order values in Domain i of chosen X_i
 - We gonna pick some value for the variable, from the domain
 - Maybe the only two color available for variable X_i is R and B, and we gonna order R and B with some heuristic
- And for each value in this order we decided, we gonna update the weight w
 - $\text{delta} - \text{add on weight}$ is gonna be the product of all factors of
 - partial assignments whatever you have decided so far (X)
 - Union whatever value ur looking at for this new variable X_i to pick (color for example)
 - F_j are the dependent factors of (partial assignment, new variable), meaning we are lookint current partial assignment and x_i
 - If $\text{delta} == 0$, continue, meaning this assignment does not work(it made everything 0), and we should try other things
 - If $\text{delta} != 0$, meaning his assignment works and we can update our domain, that's the thing that can save us time
- Backtrack()
 - X: we extended it by value v
 - w: whatever weight we started * delta
 - domain: updated domain

Update domain - forward checking approach



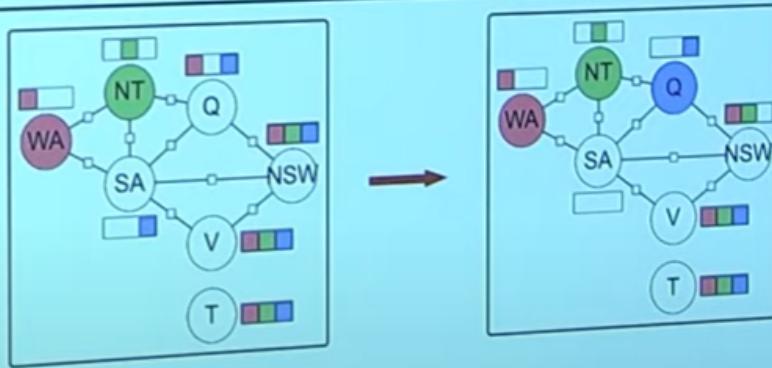
One example is: let's say we decided WA to be red, we can just look at its neighbours and see if we can update domain for them (and we found they cannot use red)

Lookahead: forward checking



Key idea: forward checking (one-step lookahead)

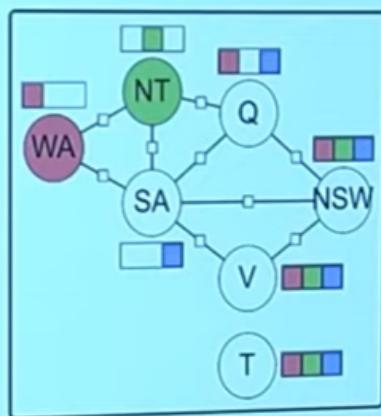
- After assigning a variable X_i , eliminate inconsistent values from the domains of X_i 's neighbors.
- If any domain becomes empty, don't recurse.
- When unassign X_i , restore neighbors' domains.



Heuristic

Choose which unassigned variable X_i

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)

28

Pick the **most constraint variable** Why?

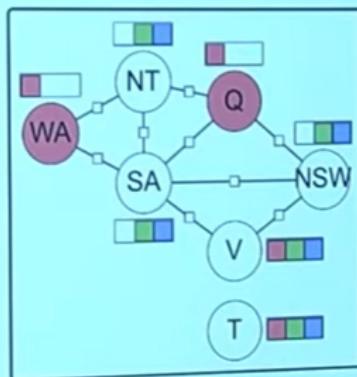
- Less options
- If it is gonna fail, let fail early

What Value to choose

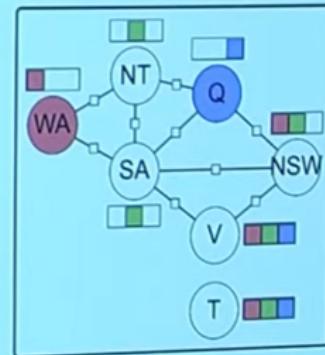
Pick the least constraint value to leave options for other variables around you

Order values of a selected variable

What values to try for Q?



$$2 + 2 + 2 = 6 \text{ consistent values}$$



$$1 + 1 + 2 = 4 \text{ consistent values}$$



Key idea: least constrained value

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

29

We are gonna pick the one that leaves the most options possible So we gonna order the value of selected variable X_i by decreasing number of consistent values of neighboring variables

This will only work if we are doing forward checking

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

Arc Consistency

For forward checking, we are just checking neighbor node and update domain For Arc Consistency, it goes through the whole CSP and tries to update the furthur nodes ahead of us The main idea is to reduce the branching factors

Arc consistency

Idea: eliminate values from domains \Rightarrow reduce branching



Example: numbers

Before enforcing arc consistency on X_i :

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

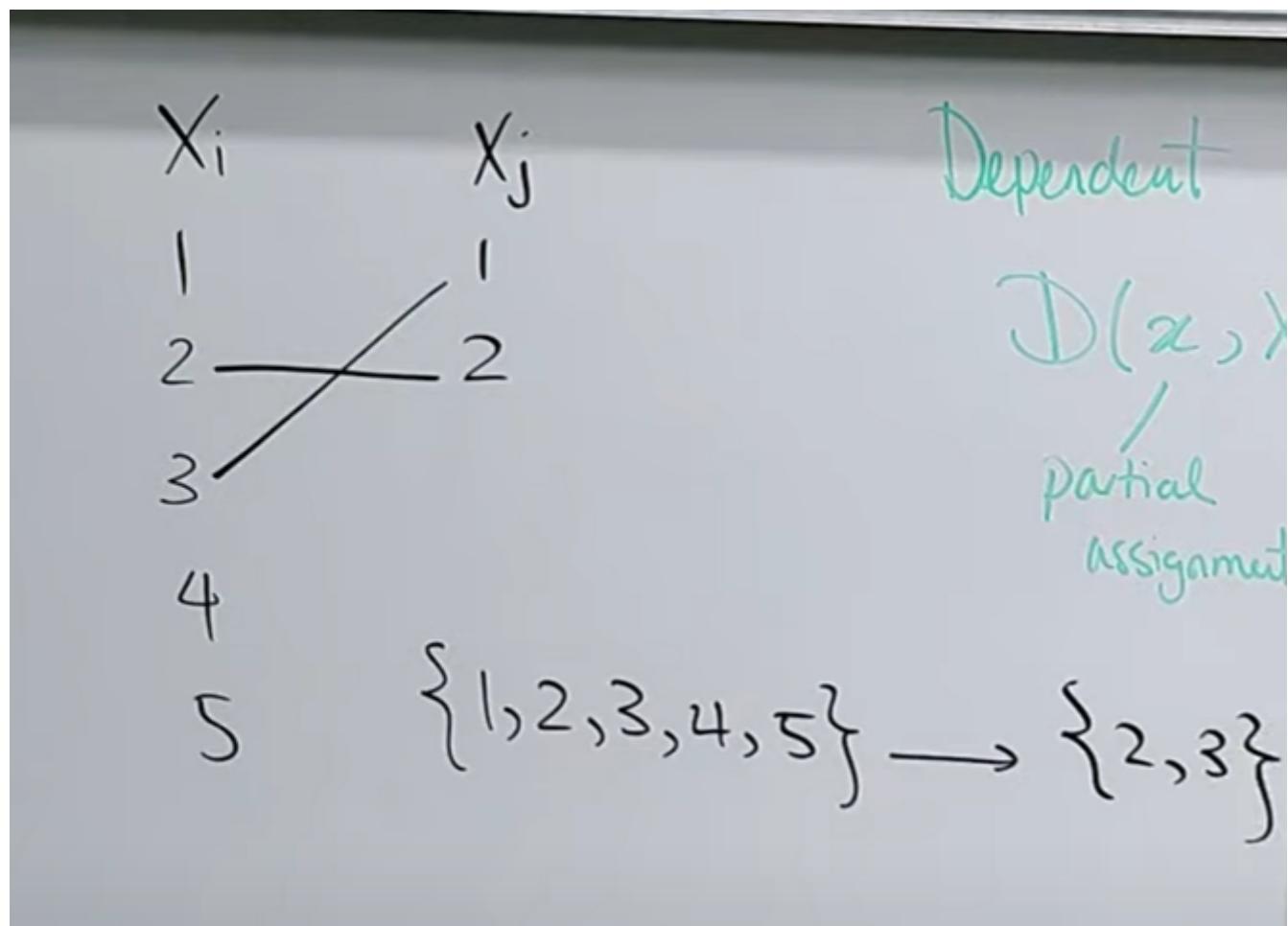
$$X_j \in \text{Domain}_j = \{1, 2\}$$

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

After enforcing arc consistency on X_i :

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

[whiteboard]



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 .

Basically it ensures everything should be consistent between X_i and X_j . If it is inconsistent, remove it, and tries to make all factors are not equal to 0



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 .

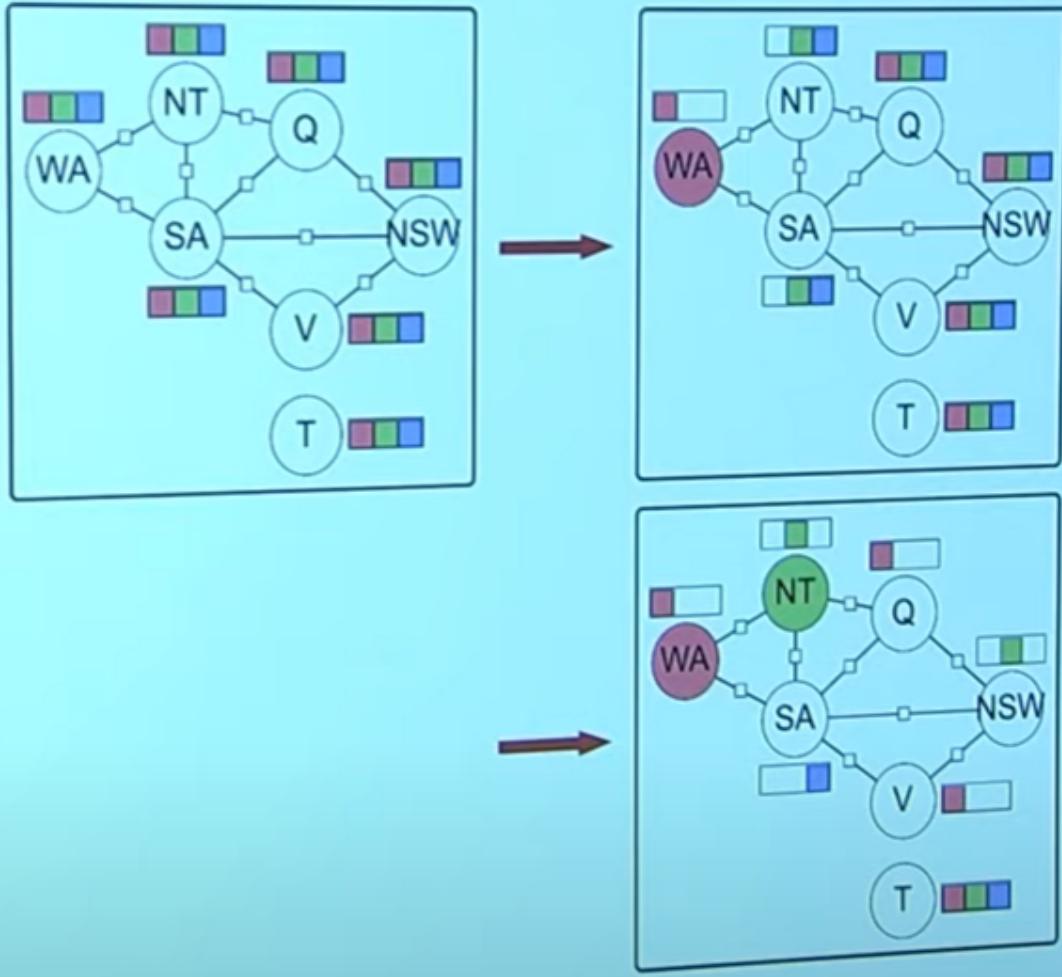


Algorithm: enforce arc consistency

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

The only thing we are touching is X_i , to make it consistent with some other 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.

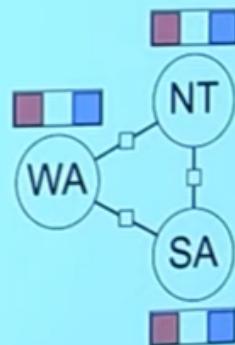
While set is non-empty:

- 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.

If the domain is changed, add the node back in Time complexity worst case scenario $O(ed^3)$

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...



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.)

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]

Examples: [vote] [csp] [pair] [chain] [track] [alarm] [med] [dep] [delay] [min] [new]
[\[Background\]](#) [\[Documentation\]](#)

```
// Create your own factor graph!
// Call variable(), factor(), query() followed by an
// inference algorithm.
```

```
variable('A', [1,2])
variable('B', [1,2])
variable('C', [1,2])

factor('f1', 'A B', function(a,b){
return a!=b;
})

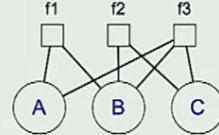
factor('f2', 'B C', function(b,c){
return b==c;
})

factor('f3', 'A B C', function(a,b,c){
return (a==2) + (b==2) + (c==2) <= 1;
})

sumVariableElimination()
```

Query: $\mathbb{P}(A, B, C)$

Algorithm: **variable elimination (sum)**



Algorithm done.

Final factor: final

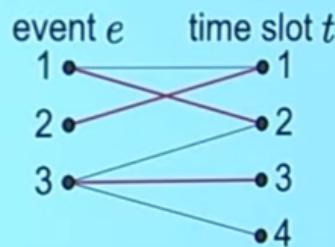
A	B	C	final(A,B,C)	$\mathbb{P}(A,B,C)$
1	2	1	1	1
2	1	1	1	2

A	B	f1(A,B)	E
1	2	1	1
2	1	1	2

- . E events , T time slots
- . each e in exactly one t.
- . each time slot t can have at most one event .
- . $(e,t) \in A$

Event and Timeslot should both belong some given set A

Example: event scheduling (section)

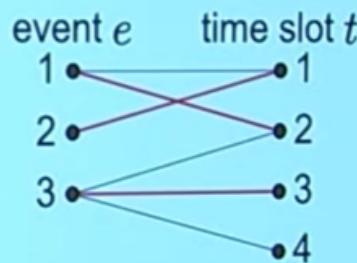


CSP formulation 1:

- Variables: for each event e , $X_e \in \{1, \dots, T\}$
- 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 $[(e, X_e) \in A]$

E as variable (we get constraint 1 for free) E number of variables of domain of size T E^2 binary factors (constraint 2) + E number of unary factors (constraint 3)

Example: event scheduling (section)



CSP formulation 2:

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

In this formulation: constraint 1: Each timeslot have an event or nothing constraint 2: each event have one unique $t \Rightarrow$ all t have no duplicated e (think about sculpture example). This gonna be a t -ary factor

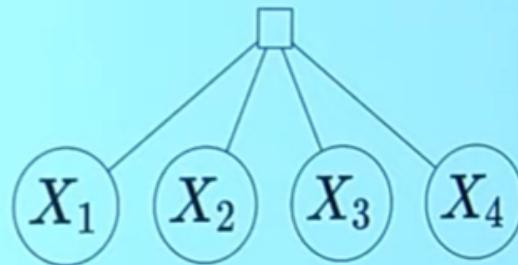
- So I have less factors but it is t -ary

Which one to choose? It really depends on the number of t or e , which one is greater.

If we have less event but more timeslots, formulation 1 is better (e^2 binary constraints). If we have less timeslot but more events, then formulation 2 is better ()

If we have n -ary constraint, we can transform it into binary constraint that are equivalent. The reason is usually our algorithm requires to have binary or unary constraints

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:

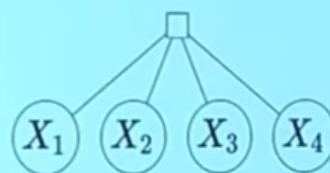
$$\text{Weight}(\{X_1 : 0, X_2 : 0, X_3 : 0, X_4 : 0\}) = 0$$

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

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

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

N-ary constraints: attempt 1 (section)



Key idea: auxiliary variable

Auxiliary variables hold intermediate computation.

Factors:

Initialization: $[A_0 = 0]$

$$i \quad 0 \quad 1 \quad 2 \quad 3 \quad 4$$

$$X_i: \quad 0 \quad 1 \quad 0 \quad 1$$

$$A_i: \quad 0 \quad 0 \quad 1 \quad 1 \quad 1$$

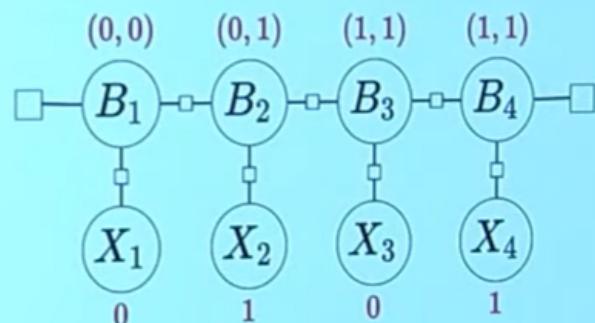
Processing: $[A_i = A_{i-1} \vee X_i]$

Final output: $[A_4 = 1]$

The idea is to define a new variable

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