

CS 161 Fundamentals of Artificial Intelligence

Lecture 6

Constraint Satisfaction Problem

Quanquan Gu

Department of Computer Science
UCLA

Jan 26, 2023

Outline

- CSP examples
- Constraint Propagation
- Backtracking search for CSPs
- Problem structure and problem decomposition

Constraint satisfaction problems (CSPs)

A constraint satisfaction problem (CSP) consists of three components: X , D , and C :

- ▶ X : a set of variables $\{X_1, \dots, X_n\}$.
- ▶ D : a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
- ▶ C : a set of constraints. $\langle scope, rel \rangle \in C$: $scope$: tuple of variables; rel : values that those variables can take on

State of CSP: an assignment of values to some or all of the variables

- ▶ **Complete assignment**: every variable is assigned
- ▶ **Partial assignment**: assigns values to only some of the variables
- ▶ **Consistent**: not violate any constraints

Solution: a consistent, complete assignment

Example: Map-Coloring



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

Domains: $D_i = \{red, green, blue\}$

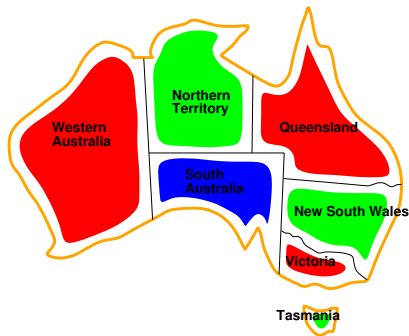
Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in$

$\{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

Example: Map-Coloring contd.



Solutions are assignments satisfying all constraints, e.g.,
 $\{WA = red, NT = green, Q = red, NSW = green$
 $V = red, SA = blue, T = green\}$

Varieties of CSPs

$$5x_1 + 3x_2 \leq 10$$

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- e.g., job scheduling, variables are start/end days for each job
- need a **constraint language**, e.g., $StartJob_1 + 5 \leq StartJob_3$

Continuous variables

- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in poly time by linear programming

$$\begin{array}{l} x_1, x_2 \in \mathbb{R} \\ \min_{x_1, x_2} 0 \\ \alpha \quad (x_1 + 2x_2) \leq 3\alpha \\ \beta \quad (4x_1 + 4x_2) \leq 7\beta \end{array}$$

Varieties of constraints

Unary constraints involve a single variable,

e.g., $SA \neq \text{green}$

Binary constraints involve pairs of variables,

e.g., $SA \neq WA$

Higher-order(Global) constraints involve 3 or more variables,

e.g., cryptarithmic column constraints, Alldif (all of the variables involved in the constraint must have different values)

Preferences (soft constraints), e.g., *red* is better than *green*

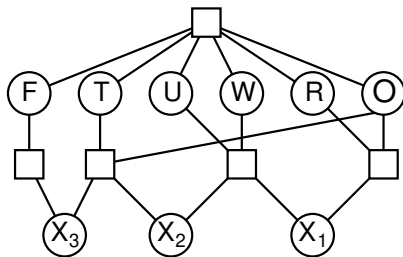
often representable by a cost for each variable assignment

→ constrained optimization problems

Example: Cryptarithmic

factor graph

$$\begin{array}{r} 765 \\ TWO \\ + TWO \\ \hline FOUR \\ 1530 \end{array}$$



Squares represent constraints

Variables: $F T U W R O X_1 X_2 X_3$

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$\text{alldiff}(F, T, U, W, R, O)$$

$$O + O = R + 10 \cdot X_1,$$

$$X_1 + W + W = U + 10 \cdot X_2,$$

$$X_2 + T + T = O + 10 \cdot X_3,$$

$$X_3 = F$$

Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

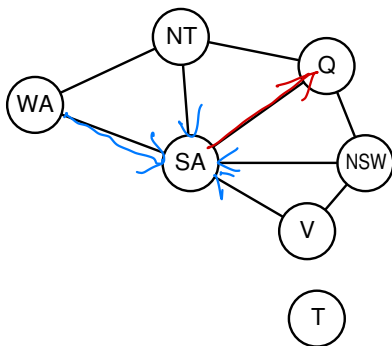
Floorplanning

Notice that many real-world problems involve real-valued variables

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Constraint propagation

$\text{domain}(WA) = \{\text{red}, \text{blue}\}$
 $\text{domain}(SA) = \{\text{blue}\}$
 $WA \rightarrow SA$ is not inconsistent

- **Constraint propagation:** using the constraints to reduce the number of legal values for a variable. $SA-WA$ is consistent
- May be done as a preprocessing step, before search starts!
- **Node consistency:** all the values in the variable's domain satisfy the variable's unary constraints
- **Arc consistency:** every value in its domain satisfies the variable's binary constraints

For any variables X_i, X_j ,

- ▶ X_i is arc-consistent with respect to X_j : for **every** value in D_i , there is some value in D_j that satisfies the binary constraint on the arc (X_i, X_j)

$\text{domain}(WA) = \{\text{red}, \text{green}\}$
 $\text{domain}(SA) = \{\text{blue}, \text{red}\}$

Arc consistency algorithm

- Each binary constraint becomes two arcs, one in each direction.
- AC-3: remove values from the domains of variables until no more arcs are in the queue

```
function AC-3(csp) returns the CSP, possibly with reduced domains  
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$   
local variables: queue, a queue of arcs, initially all the arcs in csp
```

```
while queue is not empty do  
   $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$   
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then  
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do  
      add  $(X_k, X_i)$  to queue  
  return true
```

```
function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds  
removed  $\leftarrow$  false  
for each  $x$  in DOMAIN[ $X_i$ ] do  
  if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$   
  then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true  
  return removed
```

AC-3

- if AC-3 revises D_i , then we add (X_k, X_i) , where X_k is a neighbor of X_i , since the change in D_i might enable further reductions in the domains of D_k
- Time complexity: $O(n^2 d^3)$
 - ▶ Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete
 - ▶ At most n^2 arcs
 - ▶ Checking consistency of an arc can be done in d^2 time

$$O(n^2 \cdot d^2 \cdot d)$$

$\downarrow \quad \quad \downarrow \quad \quad \downarrow$
arcs #IV # added back

Arc consistency example: Map-Coloring



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

Domains: $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in$

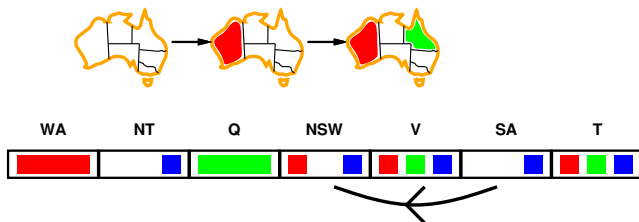
$\{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y

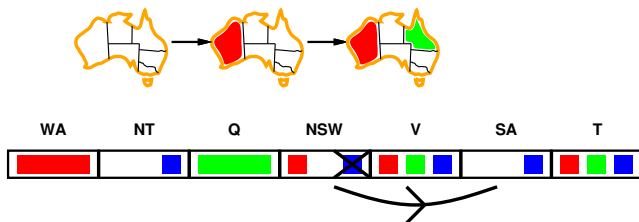


Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y

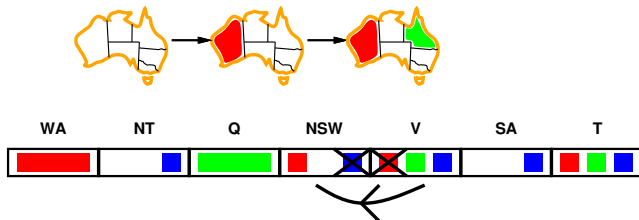


Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y



If X loses a value, neighbors of X need to be rechecked

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it
States are defined by the values assigned so far

- **Initial state**: the empty assignment, $\{\}$
- **Successor function**: assign a value to an unassigned variable that does not conflict with current assignment.
 \Rightarrow fail if no legal assignments (not fixable!)
- **Goal test**: the current assignment is complete

- 1) This is the same for all CSPs! 😊
- 2) Every solution appears at depth n with n variables
 \Rightarrow use depth-first search
- 3) Path is irrelevant, so can also use complete-state formulation
- 4) $b = (n - \ell)d$ at depth ℓ , hence $n!d^n$ leaves!!!! 😞

$$\begin{aligned} & \underbrace{(n-0)}_{n!} d \underbrace{(n-1)}_{d^n} d \cdots \underbrace{(n-\ell)}_{d^n} d \cdots \underbrace{1}_{d^n} d \\ &= n! d^n \end{aligned}$$

Backtracking search

Variable assignments are **commutative**, i.e.,

$[WA = \text{red} \text{ then } NT = \text{green}]$ same as $[NT = \text{green} \text{ then } WA = \text{red}]$

Only need to consider assignments to a single variable at each node

$\Rightarrow b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called **backtracking** search

Backtracking search is the basic uninformed algorithm for CSPs

$$= \cancel{(n-0)} \cdot d \quad \cancel{(n-1)} \cdot d \quad \dots \quad \cancel{(n-l)} \cdot d \quad \dots \quad 1 \cdot d$$

d^n

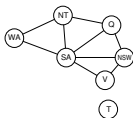
Backtracking search

- **Backtracking search:** depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign

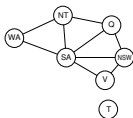
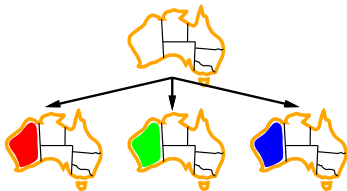
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
if assignment is complete then return assignment
var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
if value is consistent with assignment given CONSTRAINTS[csp] then
add {var = value} to assignment
result ← RECURSIVE-BACKTRACKING(assignment, csp)
if result ≠ failure then return result
remove {var = value} from assignment
return failure
```

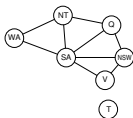
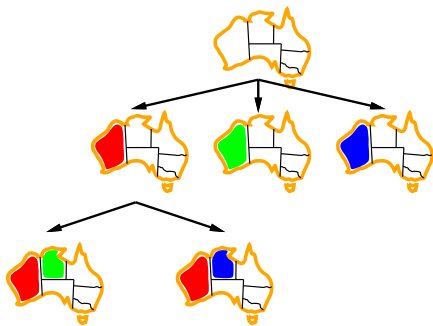
Backtracking example



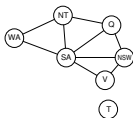
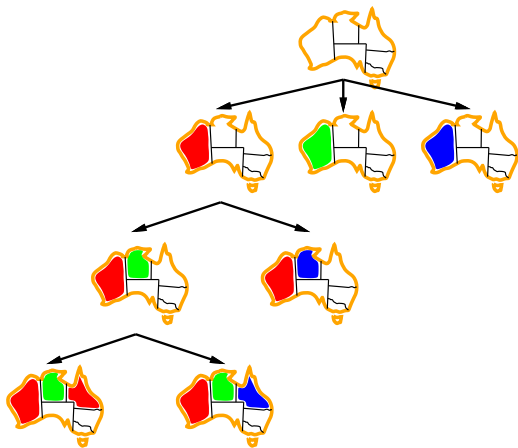
Backtracking example



Backtracking example



Backtracking example



Backtracking example: 4-Queens as a CSP

Assume one queen in each column. Which row does each one go in?

Variables Q_1, Q_2, Q_3, Q_4

Domains $D_i = \{1, 2, 3, 4\}$

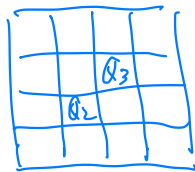
Constraints

$Q_i \neq Q_j$ (cannot be in same row)

$|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

Translate each constraint into set of allowable values for its variables

E.g., values for (Q_1, Q_2) are $(1, 3) (1, 4) (2, 4) (3, 1) (4, 1) (4, 2)$

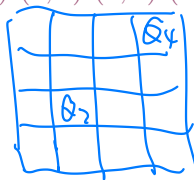


$$Q_2 = 2$$

$$Q_3 = 3$$

$$|Q_2 - Q_3| = 1$$

$$|2 - 3| = 1$$



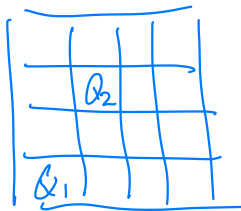
$$Q_2 = 2$$

$$Q_4 = 4$$

$$|Q_2 - Q_4| = 2$$

$$|2 - 4| = 2$$

Backtracking example: 4-Queens as a CSP



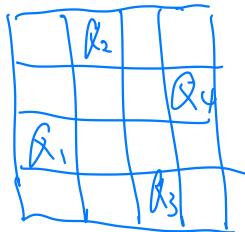
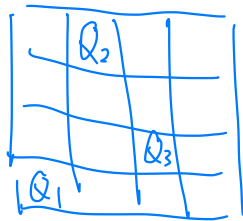
Backtracking search:

(Q_1, Q_2, Q_3, Q_4) :

$(1, X, X, X) \rightarrow (1, 3, X, X) \rightarrow$ No legal assign for Q_3 ,
backtracking

$(1, 4, X, X) \rightarrow (1, 4, 2, X) \rightarrow$ No legal assign for Q_4 , backtracking

$(2, X, X, X) \rightarrow (2, 4, X, X) \rightarrow (2, 4, 1, X) \rightarrow (2, 4, 1, 3)$, Bingo!



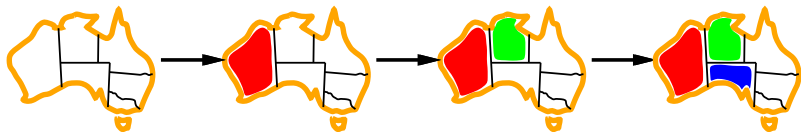
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

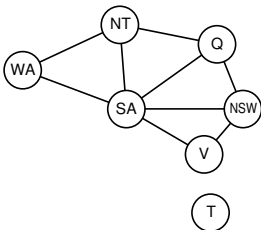
1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

- Minimum remaining values (MRV): choose the variable with the fewest legal values



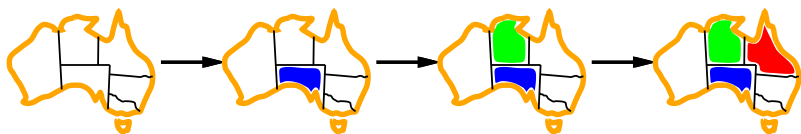
- WA:3 \rightarrow NT: 2 \rightarrow SA:1



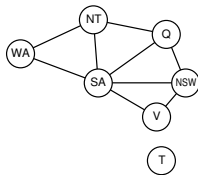
Degree heuristic

Tie-breaker among MRV variables

Degree heuristic: choose the variable with the most constraints on remaining variables

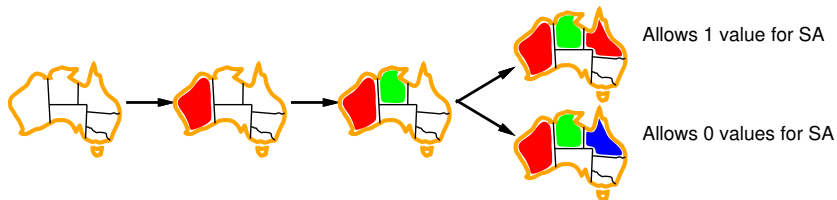


• SA:5 \rightarrow NT: 3 \rightarrow Q:3



Least constraining value

Given a variable, choose the **least constraining value**: the one that rules out the fewest values in the remaining variables



- Leave the maximum flexibility for subsequent variable assignments

Forward checking

- How do we detect failure early (if there is no solution?)
- If we do AC-3 before the search, already know if there is failure!
- What if we do not do AC-3 before?

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Forward checking

- How do we detect failure early (if there is no solution?)
- If we do AC-3 before the search, already know if there is failure!
- What if we do not do AC-3 before?

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward checking

- How do we detect failure early (if there is no solution?)
- If we do AC-3 before the search, already know if there is failure!
- What if we do not do AC-3 before?

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>

Forward checking

- How do we detect failure early (if there is no solution?)
- If we do AC-3 before the search, already know if there is failure!
- What if we do not do AC-3 before?

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

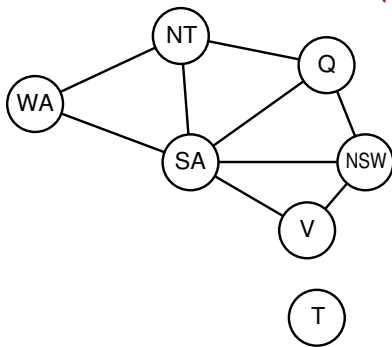


WA	NT	Q	NSW	V	SA	T
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>

Problem structure

backtracking search: $O(3^7)$

$\longrightarrow O(3^6)$
reduce
the time complexity



$C = 6$

Tasmania and mainland are **independent subproblems**

Identifiable as **connected components** of constraint graph

Problem structure contd.

Suppose each subproblem has c variables out of n total

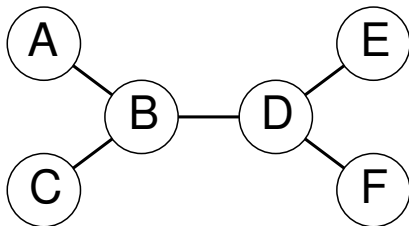
Worst-case solution cost is $n/c \cdot d^c$, **linear** in n

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

Tree-structured CSPs



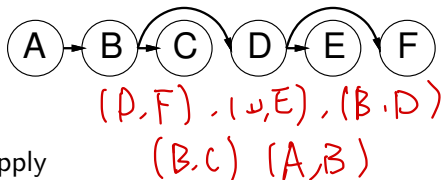
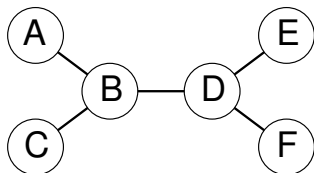
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

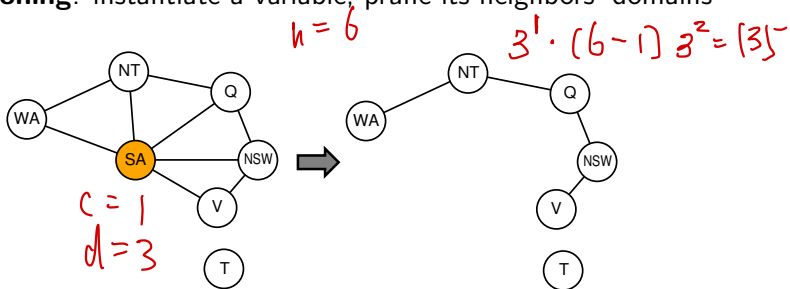


2. For j from n down to 2, apply $\text{REMOVEINCONSISTENT}(Parent(X_j), X_j)$
3. For j from 1 to n , assign X_j consistently with $Parent(X_j)$

$$\left. \begin{array}{l} \text{step 2} \\ \text{step 3} \end{array} \right\} \begin{array}{l} O(nd^2) \\ O(n) \end{array} \Rightarrow O(nd^2 + n) = O(nd^2)$$

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \Rightarrow$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small c
tree-structured CSP

Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:

- allow states with unsatisfied constraints

- operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by **min-conflicts** heuristic:

- choose value that violates the fewest constraints

- i.e., hillclimb with $h(n)$ = total number of violated constraints

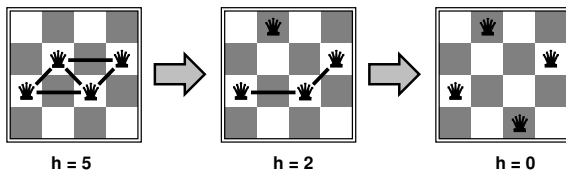
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n) = \text{number of attacks}$



Summary

CSPs are a special kind of problem:

- states defined by values of a fixed set of variables

- goal test defined by **constraints** on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice

Acknowledgment

The slides are adapted from Stuart Russell, Guy Van den Broeck et al.