Introduction to AI

Chapter06： Constraint Satisfaction Problems(CSP)

Pengju Ren
Institute of Artificial Intelligence and Robotics
pengjuren@xjtu.edu.cn

Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Forward checking
- Local search for CSPs

CSP

Standard search problem:

- **state** is a "black box" – any data structure that supports successor function, heuristic function, and goal test

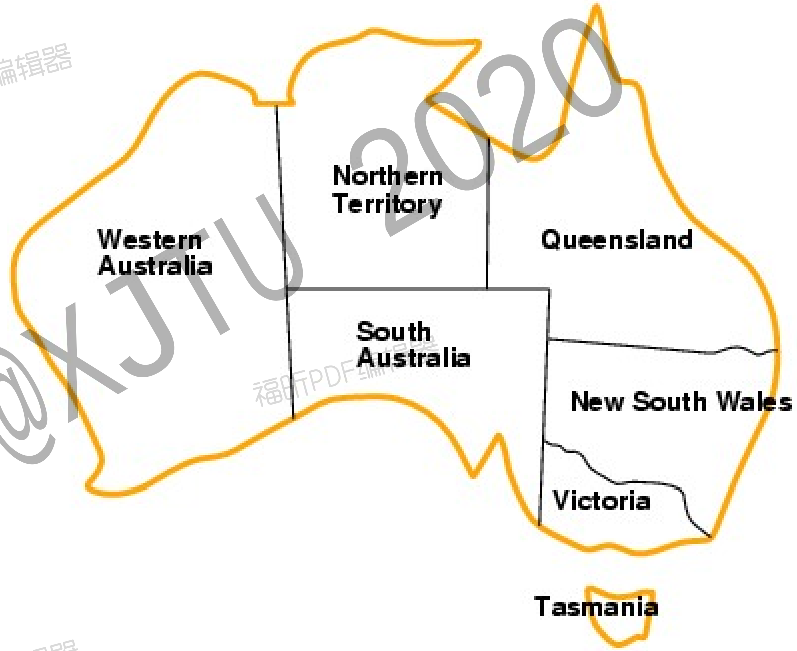
CSP:

- **state** is defined by **variables** X_i with **values** from **domain** D_i
- **goal test** is a set of **constraints** specifying allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

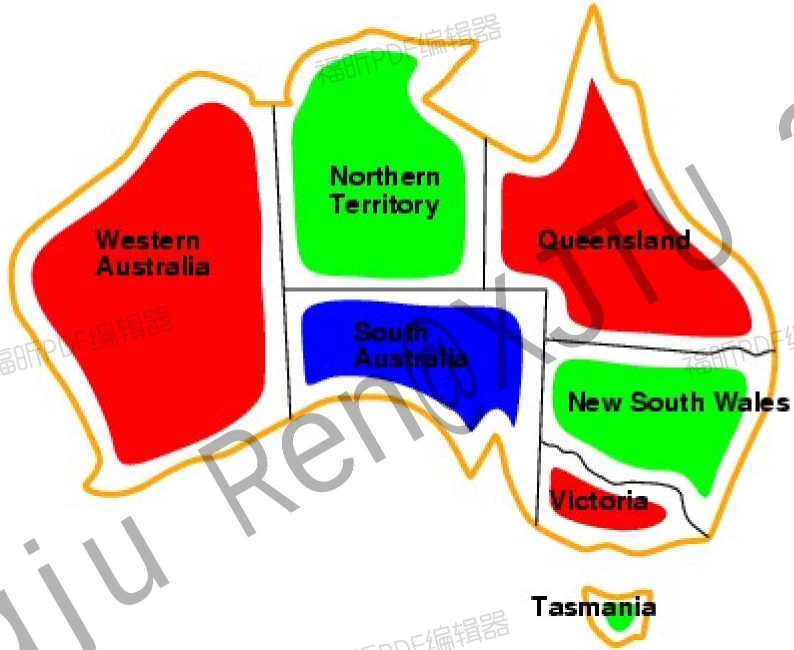
Allows useful **general-purpose** algorithms with more power than standard search algorithms

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 ≠ NT, or (WA,NT) in {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

Example: Map-Coloring

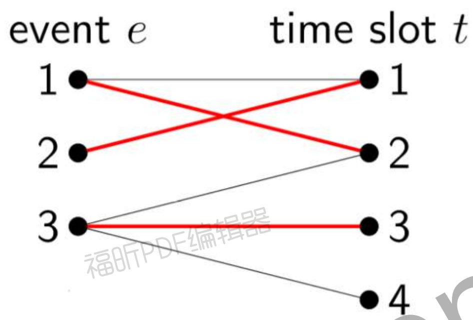


Solutions are complete and consistent assignments,

e.g., WA = red, NT = green, Q = red, NSW = green,

V = red, SA = blue, T = green

Real-world CSPs



- 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

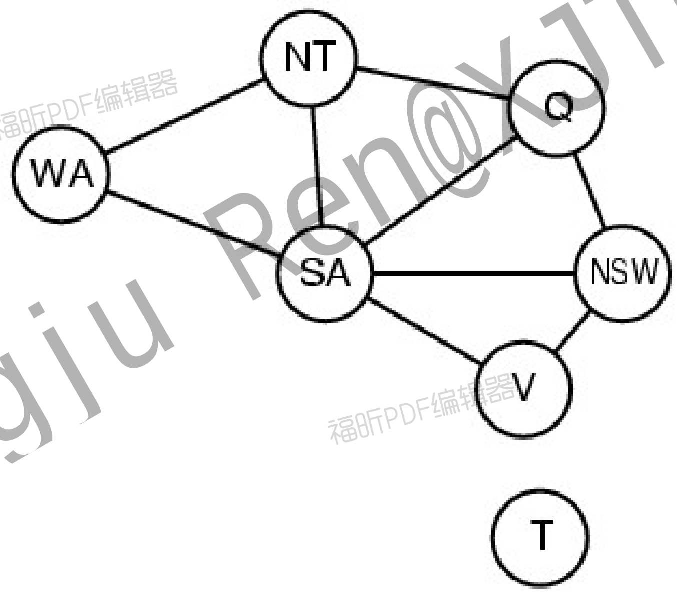
- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

Varieties of CSPs

- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* → $O(d^n)$ 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., $StartJob_1 + 5 \le StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Constraint graph

- **Binary CSP:** each constraint relates two variables
- **Constraint graph:** nodes are variables, arcs are constraints

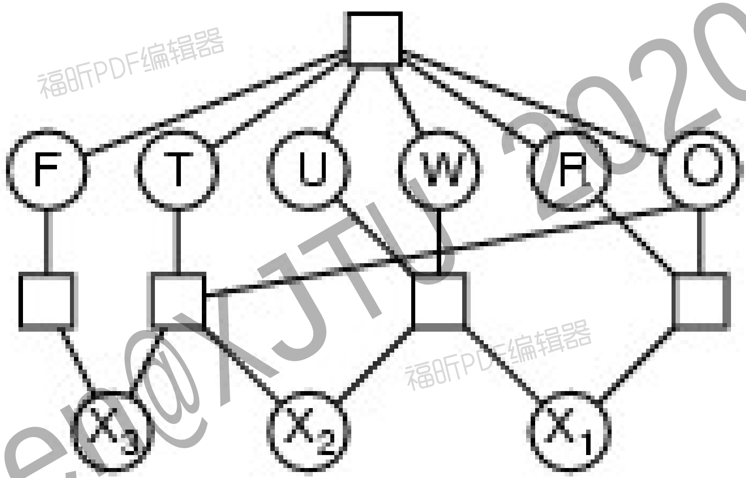


Varieties of constraints

- **Unary constraints involve a single variable,**
  - **e.g., SA ≠ green**
- **Binary constraints involve pairs of variables,**
  - **e.g., SA ≠ WA**
- **Higher-order constraints involve 3 or more variables,**
  - **e.g., cryptarithmetic column constraints**

Example: Cryptarithmetic

```
  T W O
+ T W O
-------
F O U R
```



- **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**: *Alldiff (F,T,U,W,R,O)*
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

Standard search formulation (incremental)

Let's start with the straight forward 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
  → fail if no legal assignments
- 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
     → use depth-first search
  3. Path is irrelevant, so can also use complete-state formulation

Backtracking search

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

[WA = red then NT = green] same as [NT = green then WA = red]

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

  → b = d and there are d^n leaves

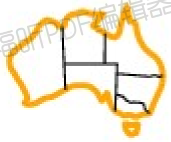
- **Depth-first search** for CSPs with single-variable assignments is called **backtracking search**
- Backtracking search is the **basic uninformed algorithm** for CSPs
- Can solve n-queens for $n \approx 25$

Backtracking search

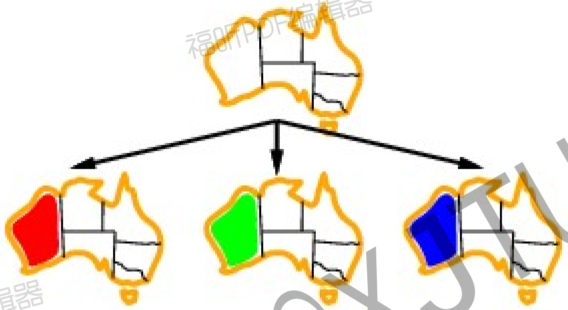
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
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 according to Constraints[csp] then
         add { var = value } to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failue then return result
         remove { var = value } from assignment
   return failure
```

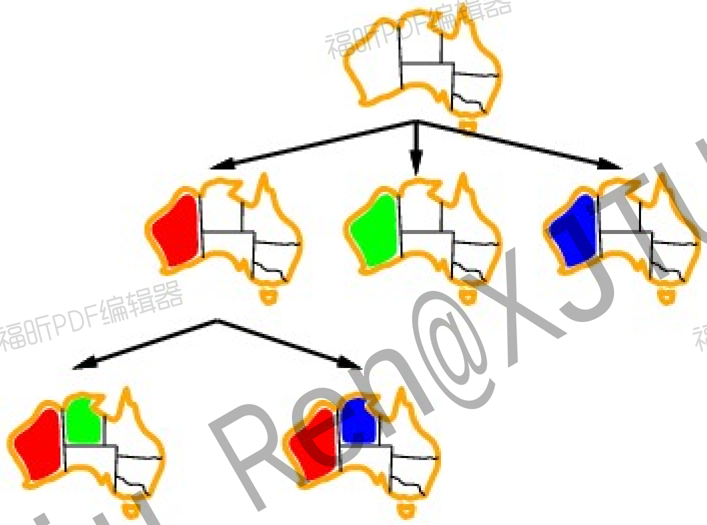
Backtracking example



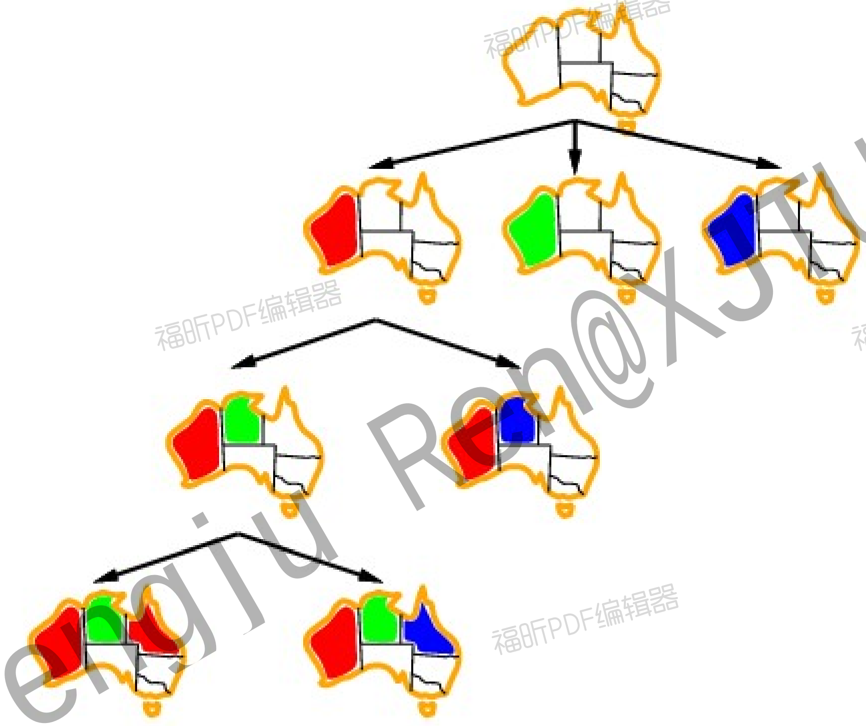
Backtracking example

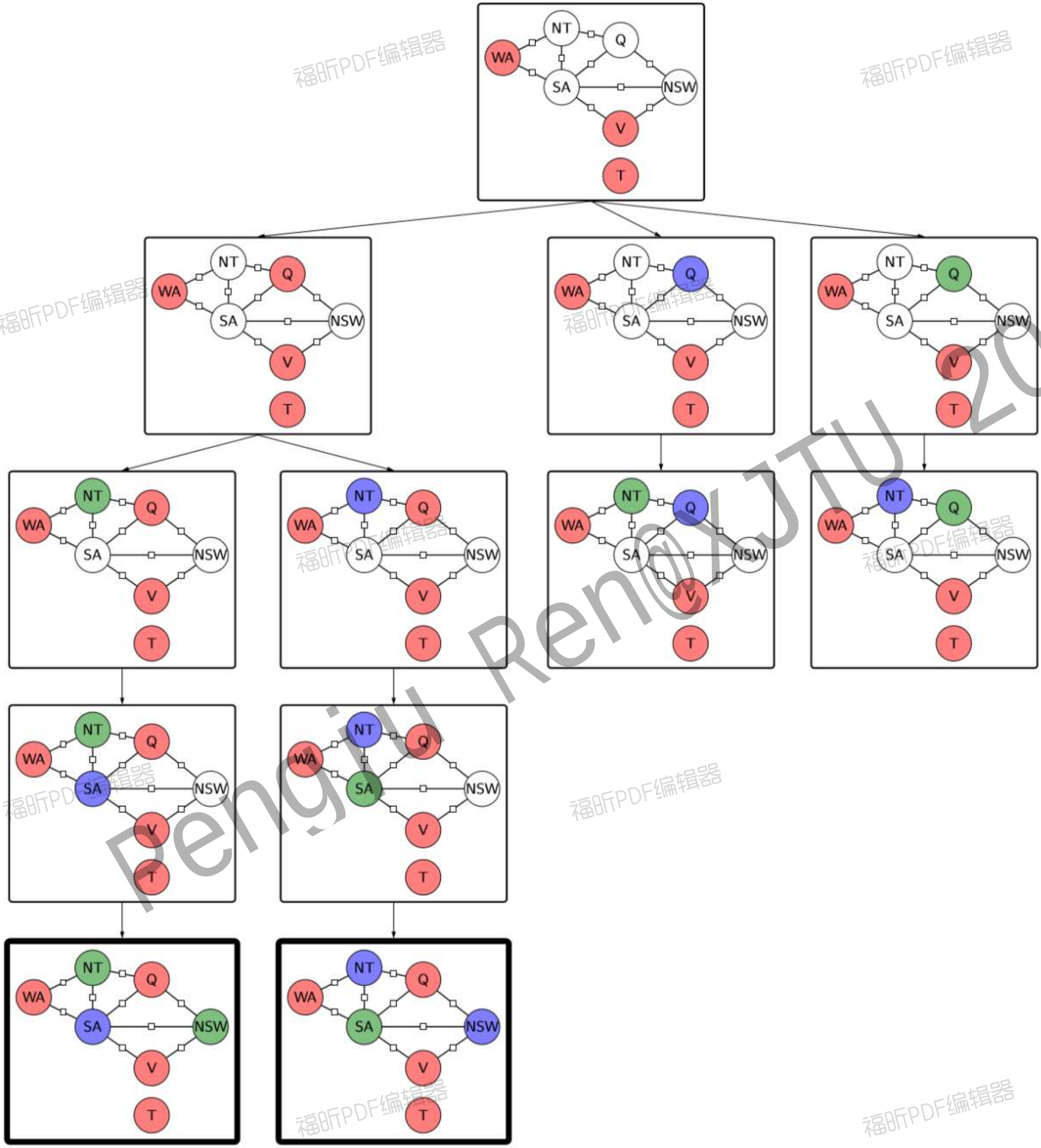


Backtracking example



Backtracking example



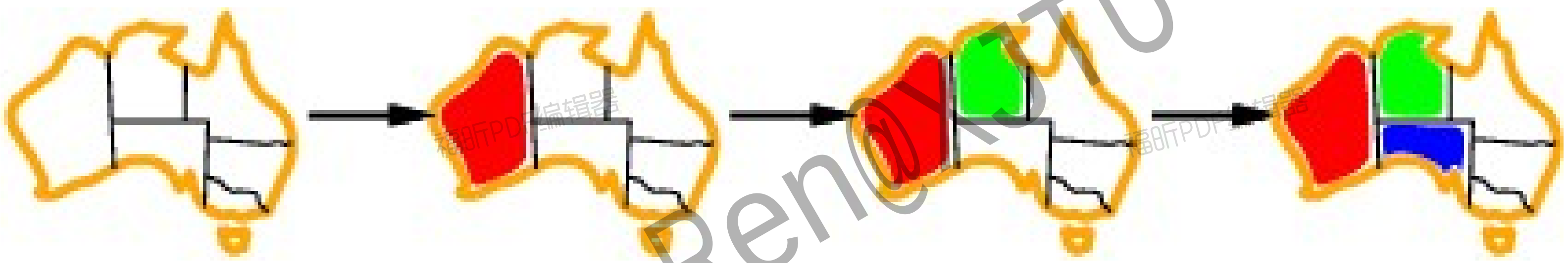

WA
NT
Q
SA
NSW
V
T

Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Most constrained variable

- **Most constrained variable:**
  choose the variable with the fewest legal values

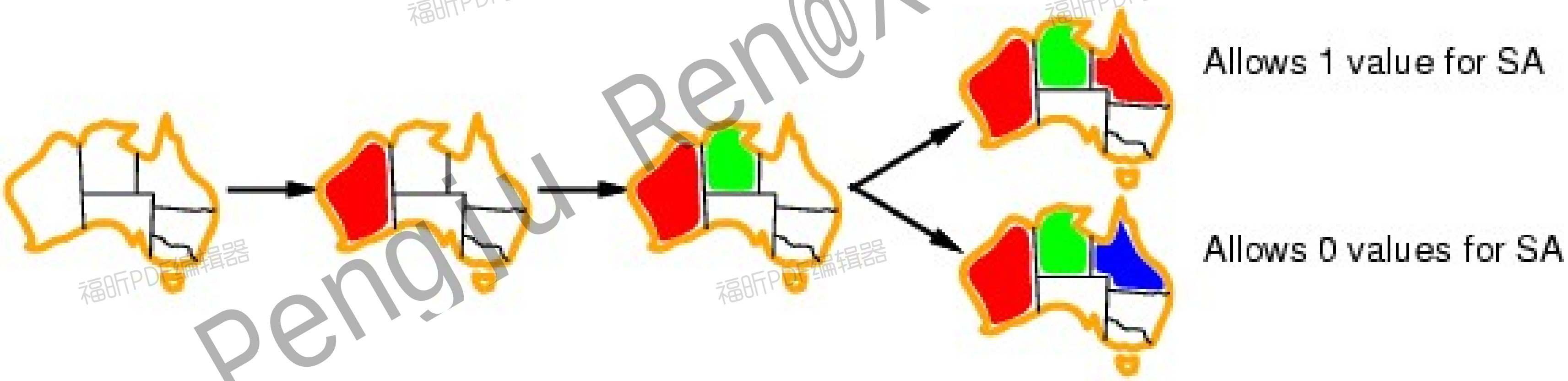


- **a.k.a. minimum remaining values (MRV) heuristic**

Which variable should be assigned next?

Least constraining value

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

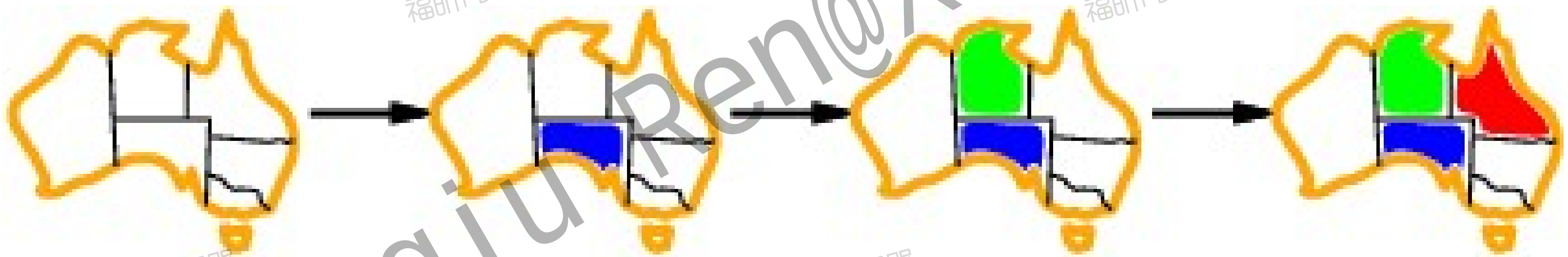


- Combining these heuristics makes 1000 queens feasible

Which variable should be assigned next?

Most constraining variable

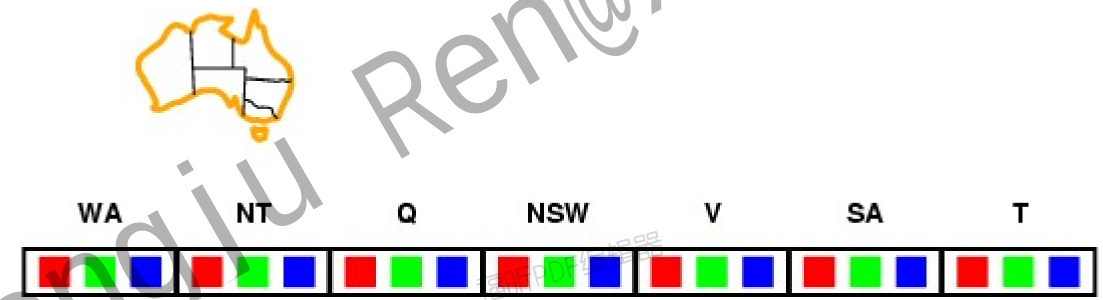
- Tie-breaker among most constrained variables
- Most constraining variable (Degree heuristic):
  - choose the variable with the most constraints on remaining variables



In what order should its values be tried?

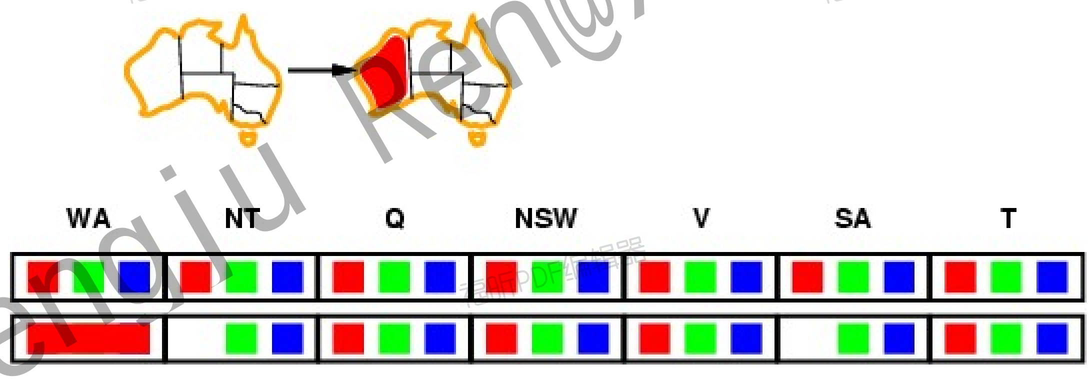
Forward checking

- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values



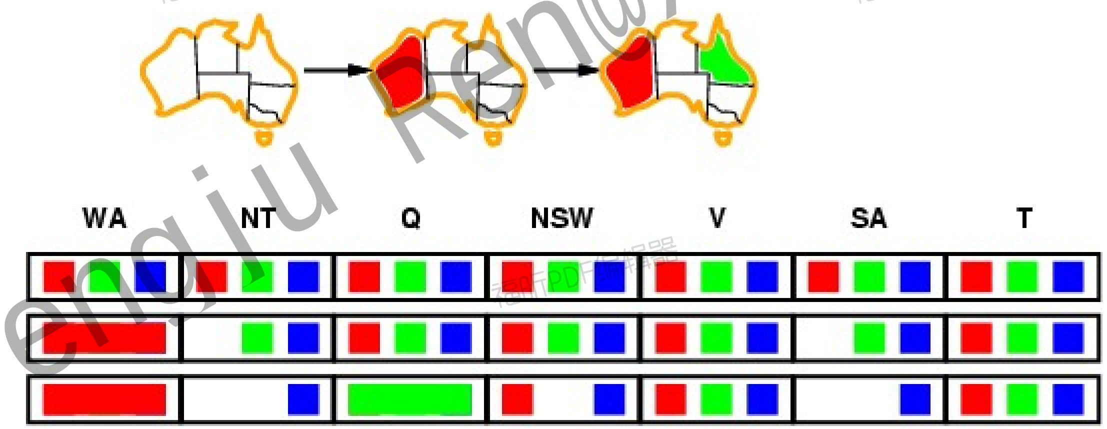
Forward checking

- **Idea**:
  - **Keep track of remaining legal values for unassigned variables**
  - **Filtering：cross off bad options (violate constrains)**
  - **Terminate search when any variable has no legal values**



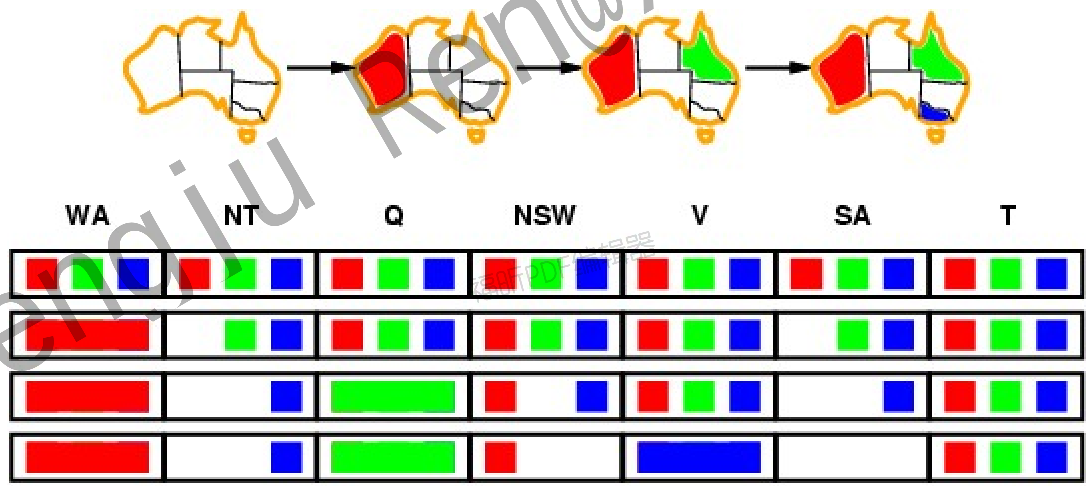
Forward checking

- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values



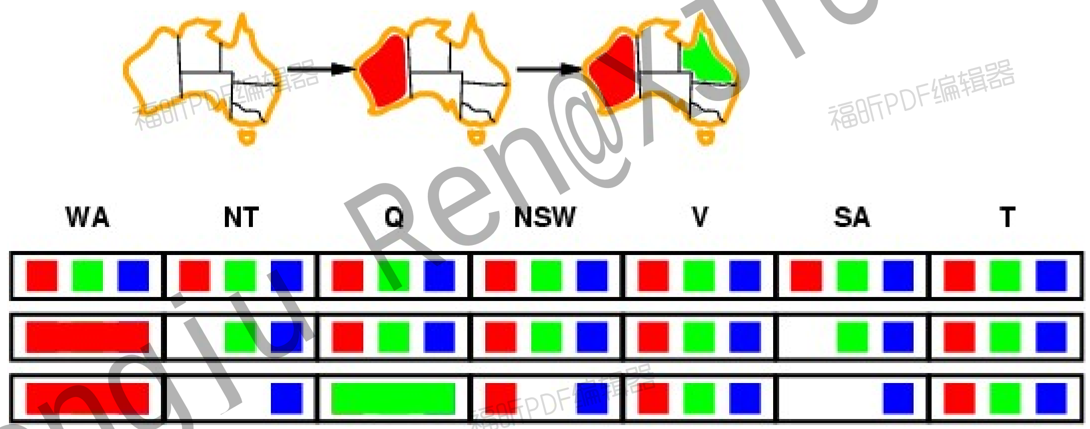
Forward checking

- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Filtering：cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values



Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

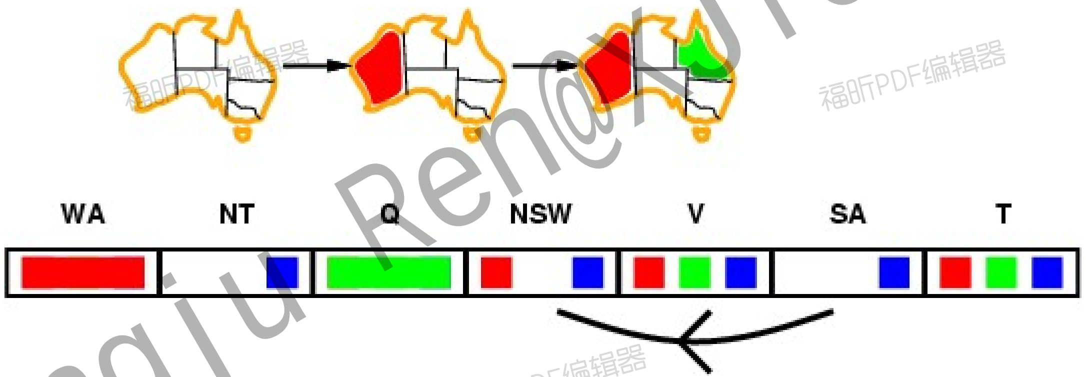


- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

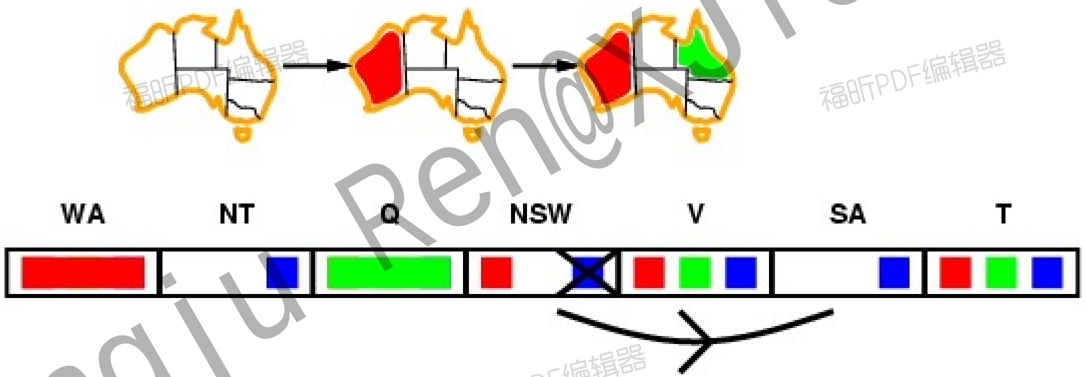
  for every value x of X there is some allowed y



Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

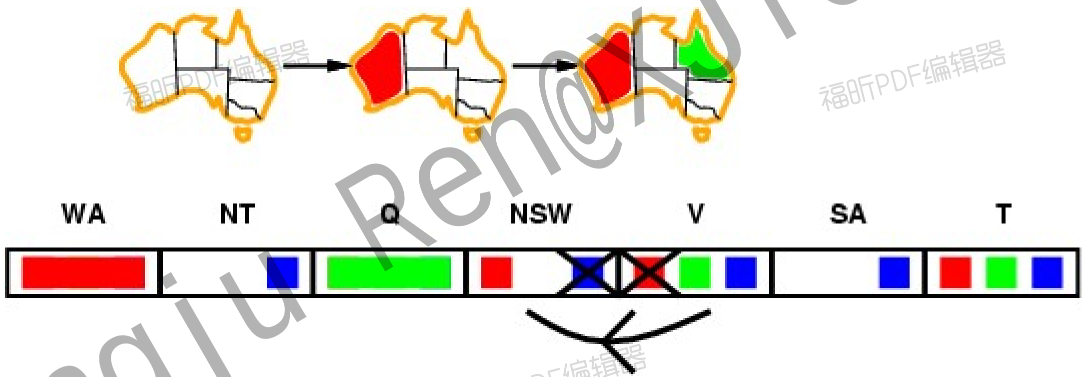
  for every value *x* of *X* there is some allowed *y*



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

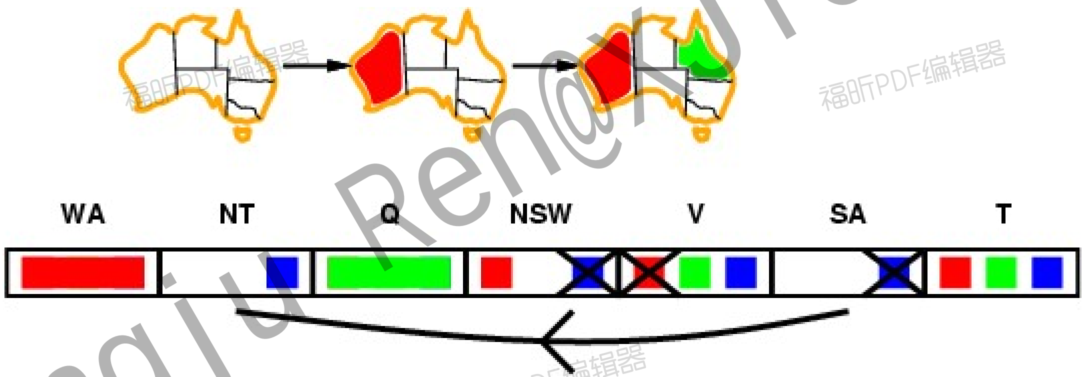


- If X loses a value, neighbors of X need to be rechecked

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
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency algorithm AC-3

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

- **Time complexity: O(n^2d^3)，n is number of variables;**
  d is number of domains;

Example

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Most constrained variable: choose the variable with the fewest legal values

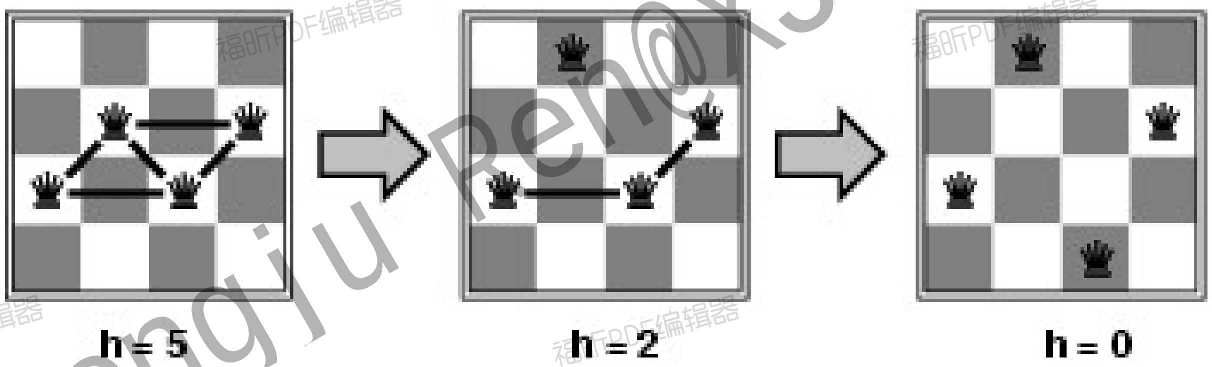
A4={1, 2, 3, 4, 5, 6, 7, 8, 9}
A6={1, 2, 3, 4, 5, 6, 7, 8, 9}
E3={1, 2, 3, 4, 5, 6, 7, 8, 9}
E6={1, 2, 3, 4, 5, 6, 7, 8, 9}
I6 ={1, 2, 3, 4, 5, 6, 7, 8, 9}

Local search 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., hill-climb with *h(n)* = total number of violated constraints

Example: 4-Queens

- **States**: 4 queens in 4 columns (4^4 = 256 states)
- **Actions**: move queen in column
- **Goal test:** no attacks
- **Evaluation:** *h(n)* = number of attacks



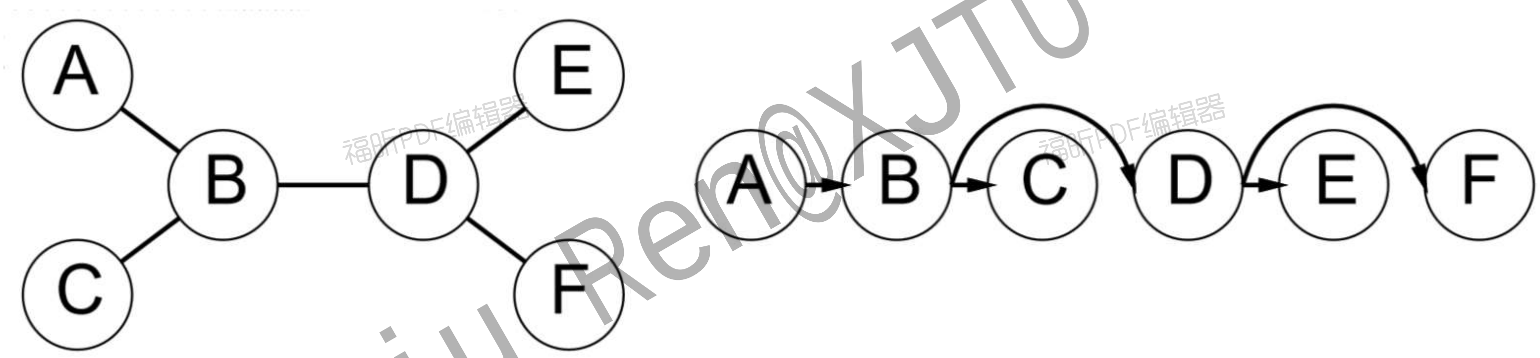
- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)

Structure and decomposition

The structure of problem, represented as constraint graph, can be used to find solution quickly, and the only way to deal with real world problem is to *decompose it in to many subproblems*.

Tree-structured CSPs

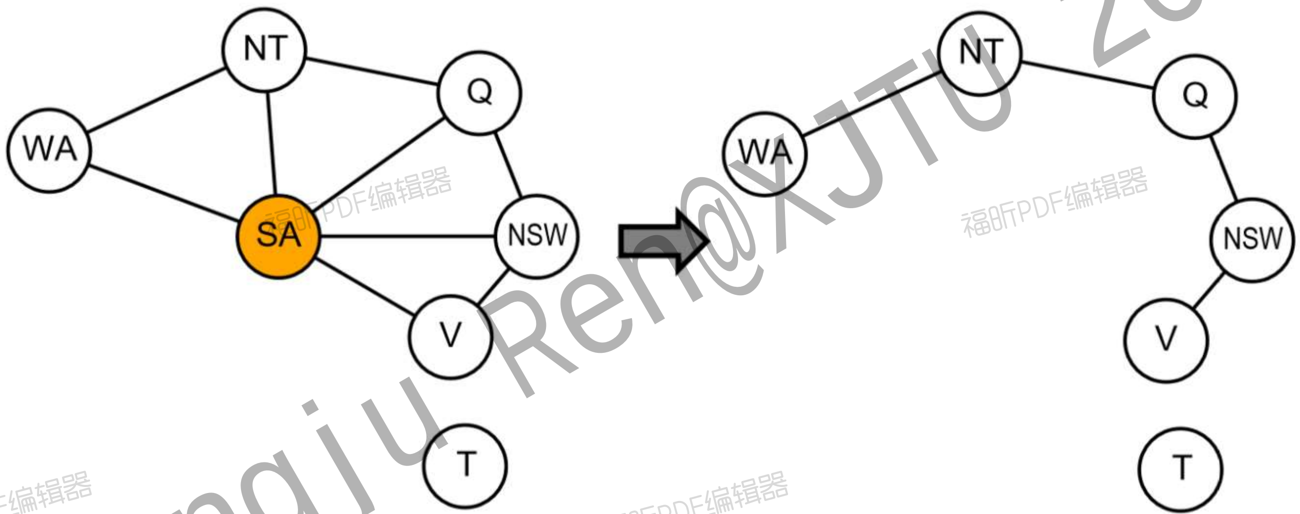
Theorem: if the constraint graph has no loops, the CSP can be solved in time $O(nd^2)$



- Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- For j from n down to 2, apply *ARC-CONSISTENT(Parent(Xj),Xj)*
- For i from 1 to n, assign *Xj* consistently with *Parent(Xi)*

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 => runtime $O(d^c \cdot (n-c)d^2)$ **very fast for small c**

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 (AC-3)
- Iterative min-conflicts is usually effective in practice
- Tree-structured CSPs can be solved in linear time