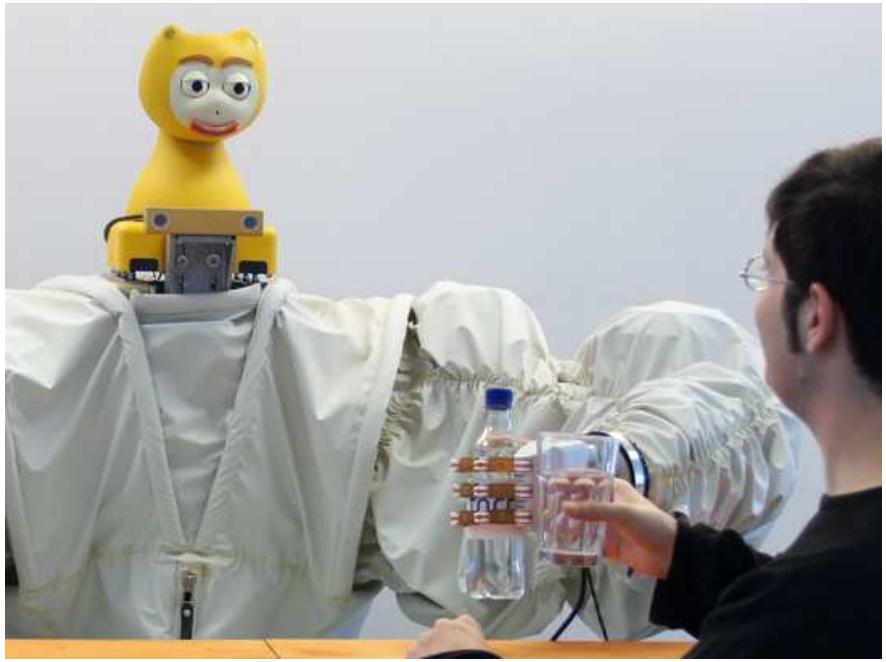
Vorlesung

Grundlagen der Künstlichen Intelligenz

Reinhard Lafrenz / Prof. A. Knoll

Robotics and Embedded Systems
Department of Informatics – I6
Technische Universität München

www6.in.tum.de
lafrenz@in.tum.de
089-289-18136
Room 03.07.055



Wintersemester 2012/13 12.11.2012

Chapter 6 (3rd ed.)

Constraint Satisfaction Problems

Constraint Satisfaction Problems

Standard search problems:

- State is a "`black box"‘, an arbitrary data structure with goal test operators, evaluation schema and successor function

CSP:

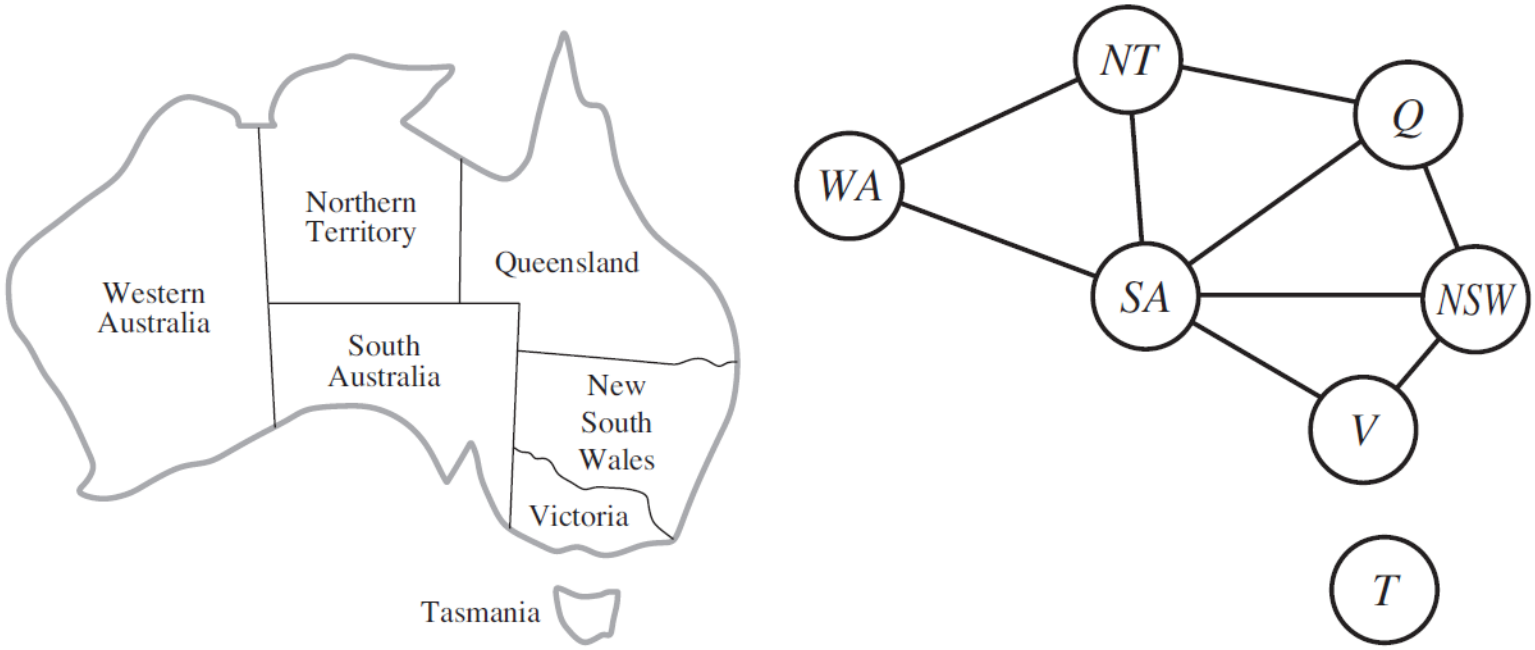
A state is defined by variables X_i with values from a domain D_i

The goal test is a set of constraints that specify the allowable combinations of values for subsets of variables.

Simple example of a formal representation language.

Allows useful general-purpose algorithms which are more effective than standard search algorithms.

Example: Map-coloring



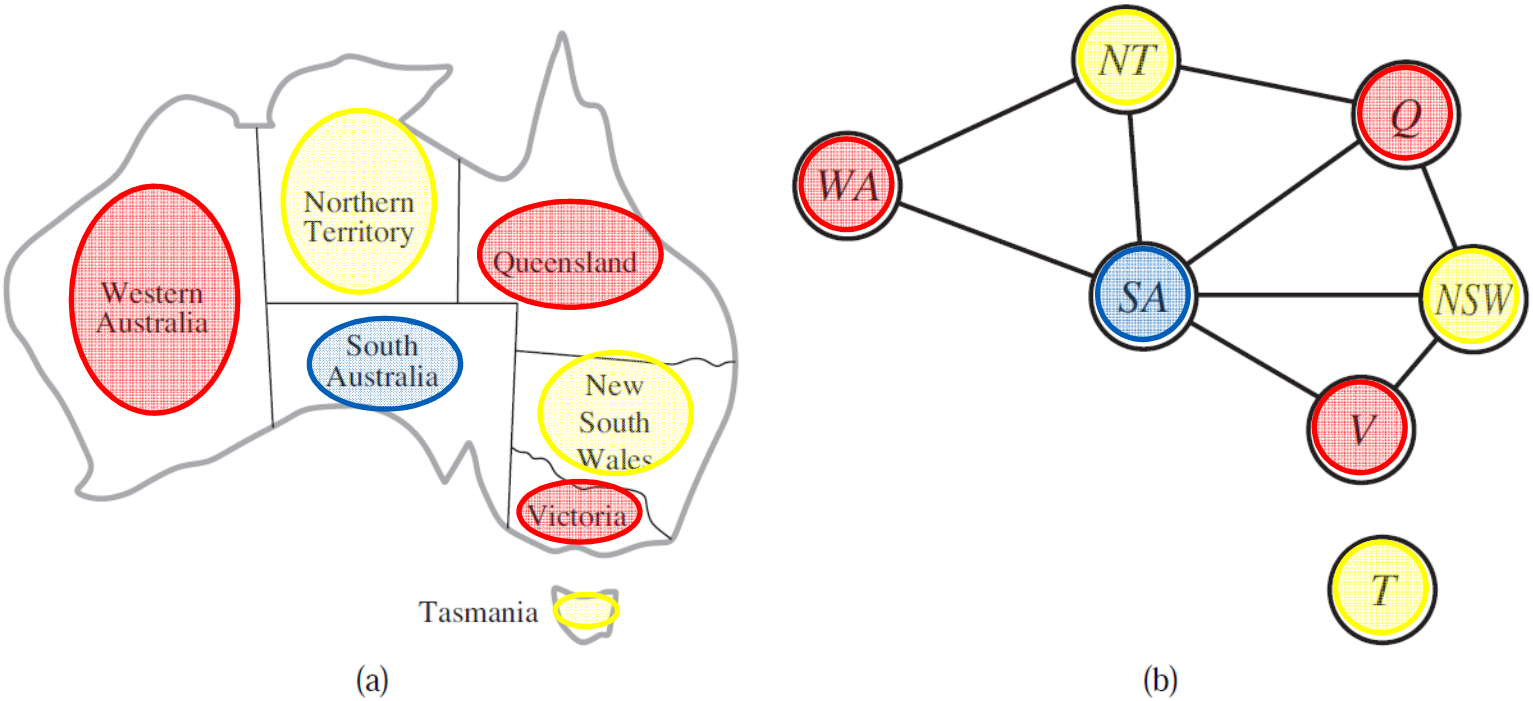
Variables: WA, NT, Q, NSW, V, SA, T

Domains: D_i = {red; yellow; blue}

Constraints: adjacent regions must have different colors

e.g. WA ≠ NT (if the language allows this), or (WA;NT) ∈ {(red; yellow); (red; blue); (yellow; red); (yellow; blue); …}

Example: Map-coloring

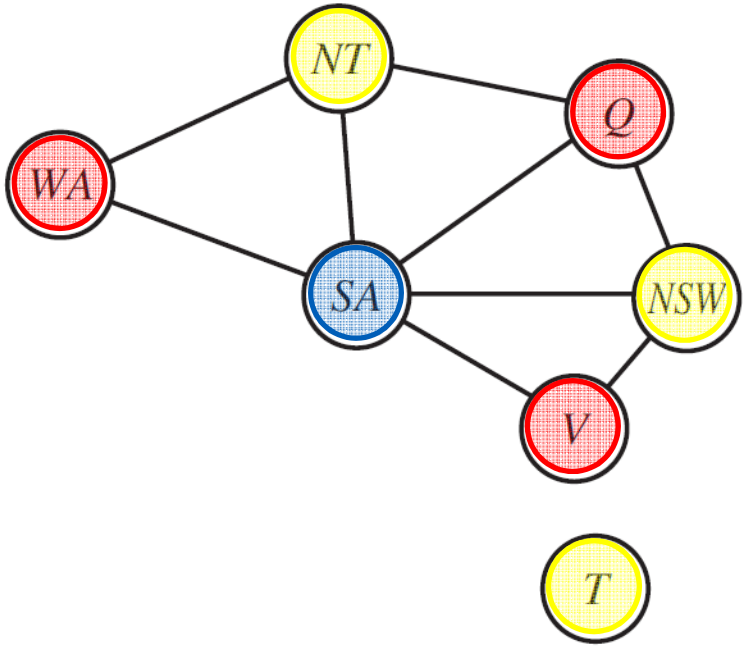


Solutions are assignments satisfying all constraints, e.g.

{WA = red, NT = yellow, Q = red, NSW = yellow,

V = red, SA = blue, T = yellow}

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!

Types of CSPs

Discrete variables

- Finite domains; size d $\Rightarrow$ $O(d^n)$ complete assignments e.g.,
  - Boolean CSPs, incl. Boolean satisfyability (NP-complete)
- infinite domains (integers, strings, …)
  - e.g., job scheduling, variables are start/end days for each job
  - requires a constraint language, e.g. $StartJob_1 + 5 < StartJob_3$
  - linear constraints solvable, nonlinear undecidable

Continuous variables

- e.g. start/end times for Hubble Telescope observations
- linear constraints are solvable in polynomial time by linear programming methods

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

Reducing the Search space with constraints

- In the example: 4 colors possible for each node
- E.g., when assigning SA:=blue, the search space only needs to consider 3 colors:

$$3^5 = 243 \Rightarrow 2^5 = 32$$

Reduction by 87%:

Constraints in real applications

- Assignment problems
  - e.g., who teaches what class or which robot assembles which part
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
  - Different Tasks, each one modelled as variable

Many real-world problems involve real-valued variables

Constraints in real applications

Example: Assembly of a car

- 15 tasks: 2 axes (front, rear), 4 wheels, fix wheel nuts for all wheels, 4 wheel covers, final inspection

- Assign a variable to each task representing the start time
- Order of tasks, given a max. duration d_i for each task i

Constraint: $T_1 + d_1 \leq T_2$

Constraints in real applications

E.g. assembly of an axis take 10 min.

$$Axis_F + 10 \leq Wheel_{RF} \; ; \; Axis_F + 10 \leq Wheel_{LF}$$
$$Axis_R + 10 \leq Wheel_{RR} \; ; \; Axis_R + 10 \leq Wheel_{LR}$$

Assembly of a wheel take 1 min for moving wheel to axis, 2 min to fix the wheel nuts, 1 min to mount the wheel caps

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \; Nuts_{RF} + 2 \leq Cap_{RF}$$

If there is e.g. only one tool to position the 2 axes, these steps need to be sequentialized (disjunctive constraint)

$$(Axis_F + 10 \leq Axis_R) \text{ or } (Axis_R + 10 \leq Axis_F)$$

Solution of CSPs: naive approach

Let's start with the straightforward 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. Every solution appears at depth *n* with *n* variables
   → use depth-first search
2. Path is irrelevant, so can also use complete-state formulation
3. b = (n - ℓ)d at depth ℓ, hence n! · d^n leaves

Solution of CSPs: Backtracking

Depth first search ist complete for CSPs, since maximally *n* operators (assignments of values to variables) are possible.
naive implementation leads to very high branching factor!

Let *Di* be the set of possible values for Vi . The branching factor is then $b = \Sigma_{i=1..n} D_i$.

The order of instantiation of variables is irrelevant for the solution. Therefore, one can select a variable for every expansion step (non-deterministically), i.e. the branching factor is $(\Sigma_{i=1..n} D_i) / n$.
After each operator application, one can test if any constraints are violated. In this case the current node does not need to be expanded further.

depth first search + test = Backtracking

Solution of CSPs: Backtracking

```
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
```


Solution of CSPs: Heuristics for CSPs

Minimum remaining values (MRV) first:

⇒ reduces branching factor!

Most constraining variable first (alternatively):

i.e. the variable with the most constraints on remaining variables.

⇒ reduces future branching factor

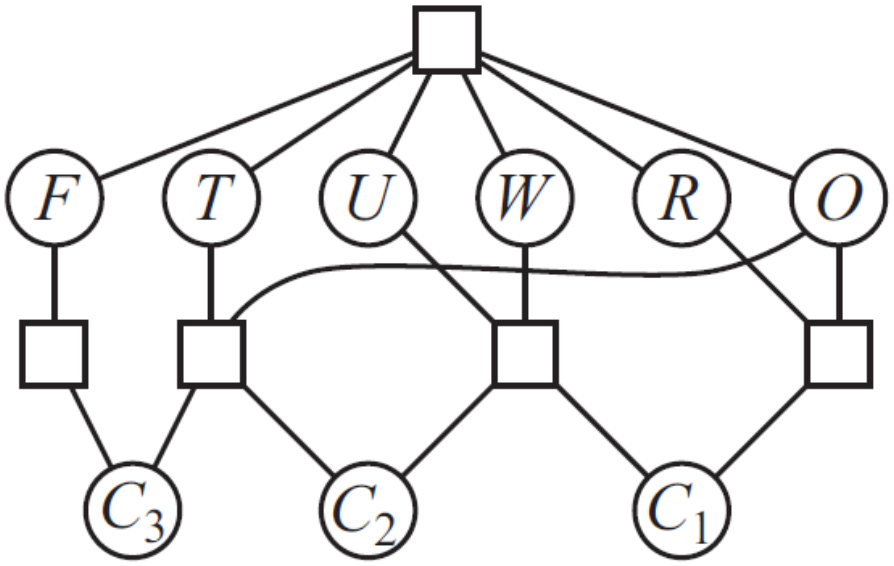
Least constraining value first:

allows more freedom for future decisions

⇒ Now, the 1000-Queens problem is solvable!

Example: Cryptoarithmetic puzzles

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



Variables: F, T, U, W, R, O, C1, C2, C3
Domains: {0; 1; 2; 3; 4; 5; 6; 7; 8; 9}
Constraints: alldiff (F;T;U;W; R;O)
O + O = R + 10 ∗ C1;
C1 + W + W = U + 10 ∗ C2;
C2 + T + T = O + 10 ∗ C3;
C3 = F

Use of auxiliary variables

Thesis: Each constraint of higher order (i.e., with multiple variables) in a finite domain can be transformed into a binary constraint set (i.e., each constraint deals with only 2 variables)

Example for trinary constraint: $A + B = C$

Sketch of proof: Auxiliary variable *AB*: pairs in the form *(A,B)*

Three constraints:

A is the first element of *(A,B)*

B is the second element of *(A,B)*

EVAL(*ab*) = C,

with EVAL(*ab*) =*a*+*b* for each $ab \in AB$

Constraint descriptions with only binary constraints are possible!

Use of auxiliary variables

Thesis: Each constraint of higher order (i.e., with multiple variables) in a finite domain can be transformed into a binary constraint set (i.e., each constraint deals with only 2 variables)

Example for trinary constraint: $A + B = C$

Sketch of proof: Auxiliary variable *AB*: pairs in the form *(A,B)*

Three constraints:

A is the first element of *(A,B)*

B is the second element of *(A,B)*

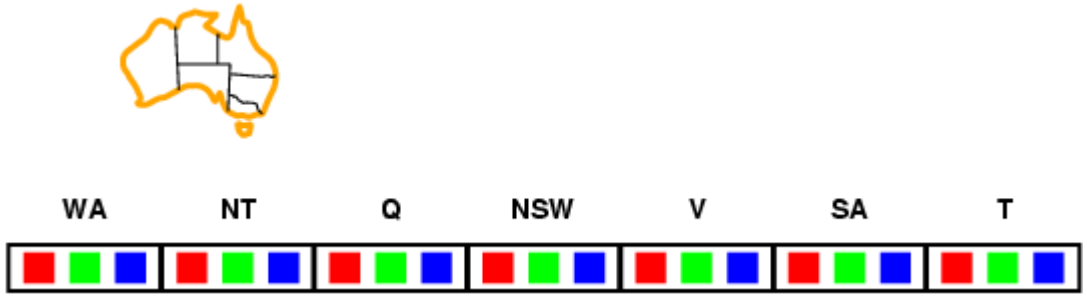
EVAL(*ab*) = C,

with EVAL(*ab*) =*a*+*b* for each $ab \in AB$

Constraint descriptions with only binary constraints are possible!

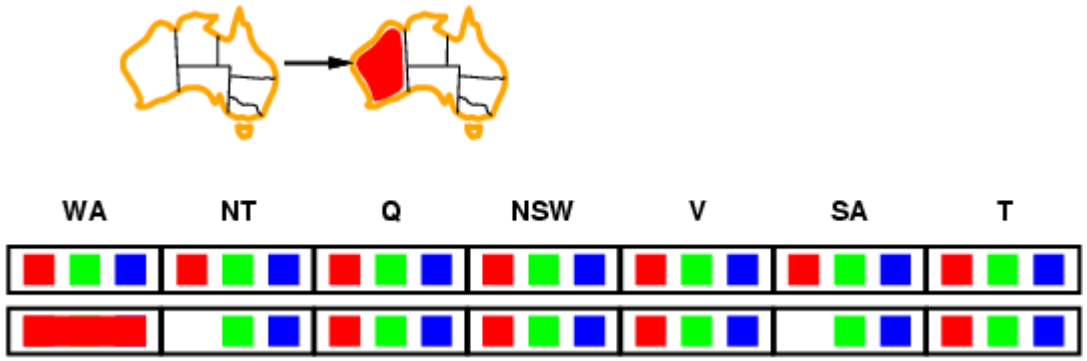
Forward checking

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



Forward checking

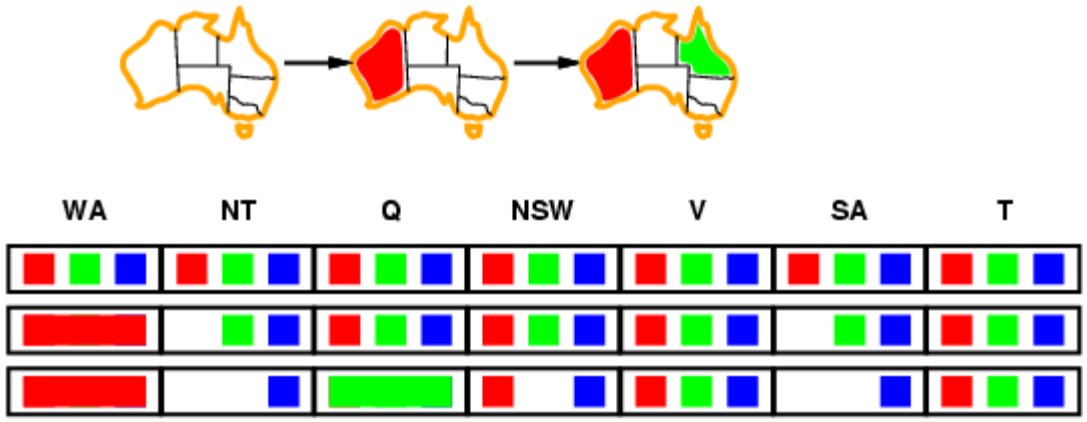
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -



21

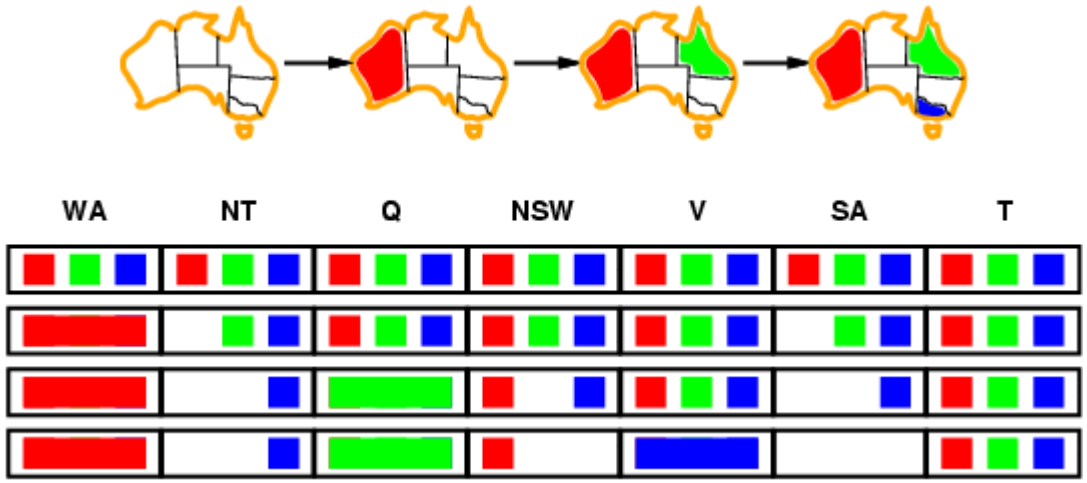
Forward checking

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



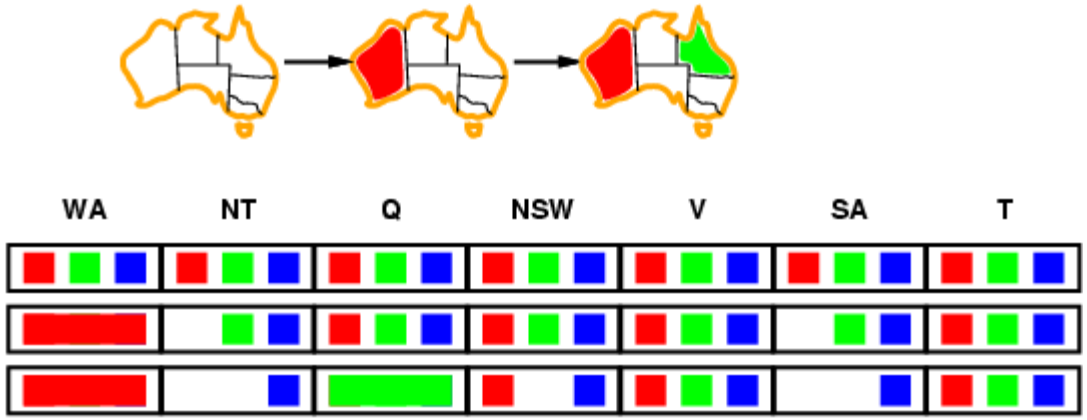
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - 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

Constraint propagation

Main idea: local consistency

- Node consistency: all unitary constraints are satisfied

- Arc consistency between X_i and X_j : For each value in D_i exists a value in D_j which fulfills the binary constraint between X_i and X_j

  Example: Arc consistency for the constraint $Y = X^2$ in the domain {0,1, …, 9}

  This leads to Dom(X) = {0,1,2,3} and Dom(Y) = {0,1,4,9}

  AC-3 algorithm uses arc consistency

AC-3 algorithm

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

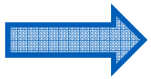
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if RM-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

---

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

Example: 8-queen as CSP

- There are 8 variables $V_1; \ldots ; V_8$, where V_i represents a queen in the i-th column.
- V_i can take values from {1; 2; …; 8}, which represents the row position.
- There are constraints between all pairs of variables which express the rule that no queen can attack any other queen.

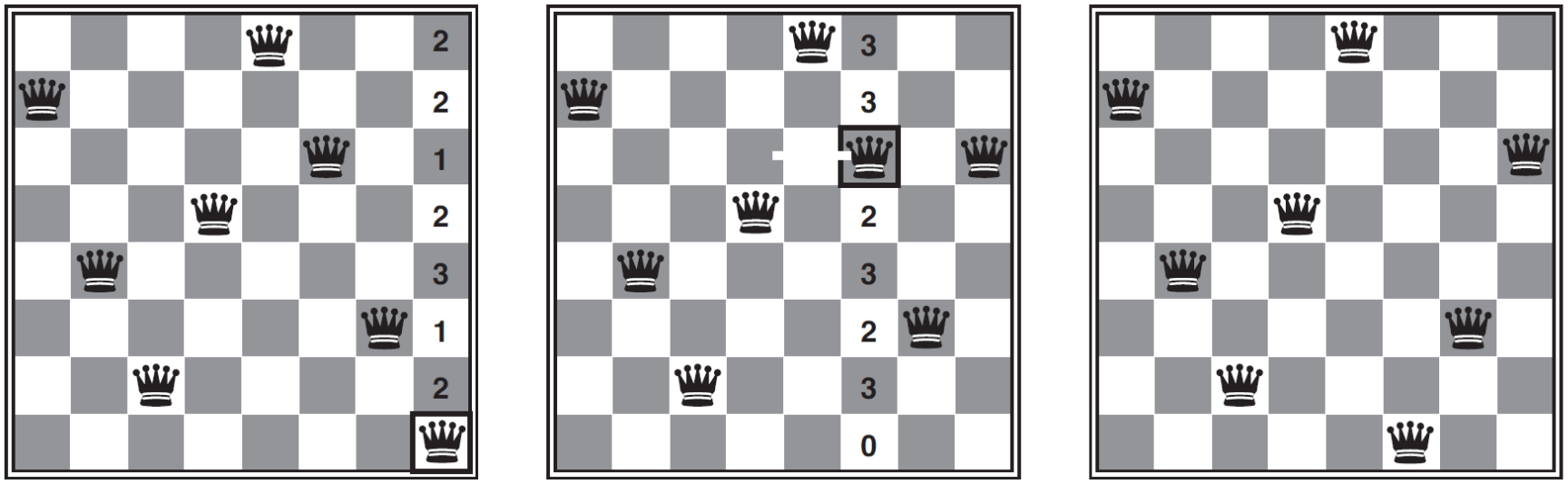


discrete, finite, binary CSP

Example: 8-queen as CSP

Min-conflicts heuristics:

- Choose a conflicting column
- Choose a field with minimal conflicts (attacks), random choice between equal possiblities

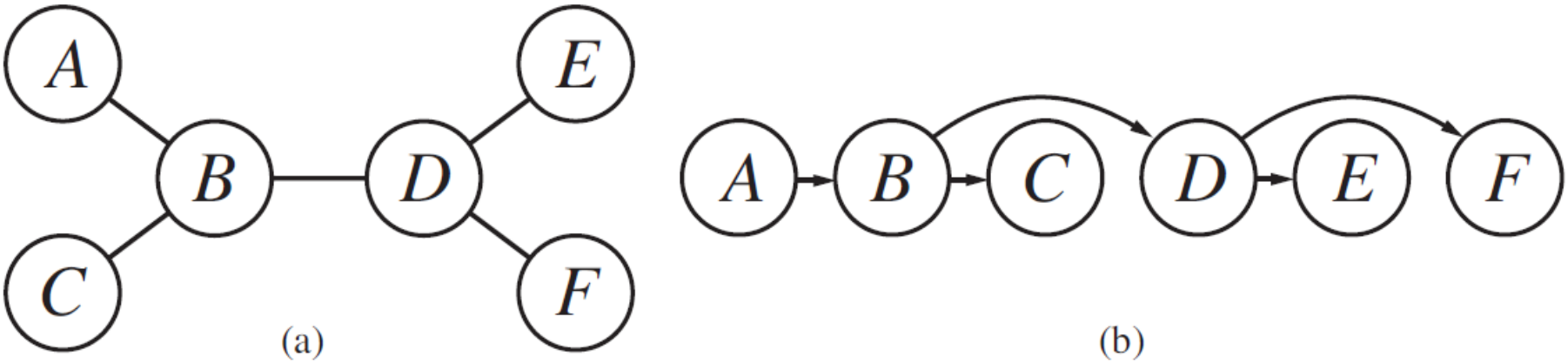


Min-conflicts was used for scheduling of the Hubble-Telescope. Reduction of computation time from 3 weeks to 10 minutes!

Using the structure of problems

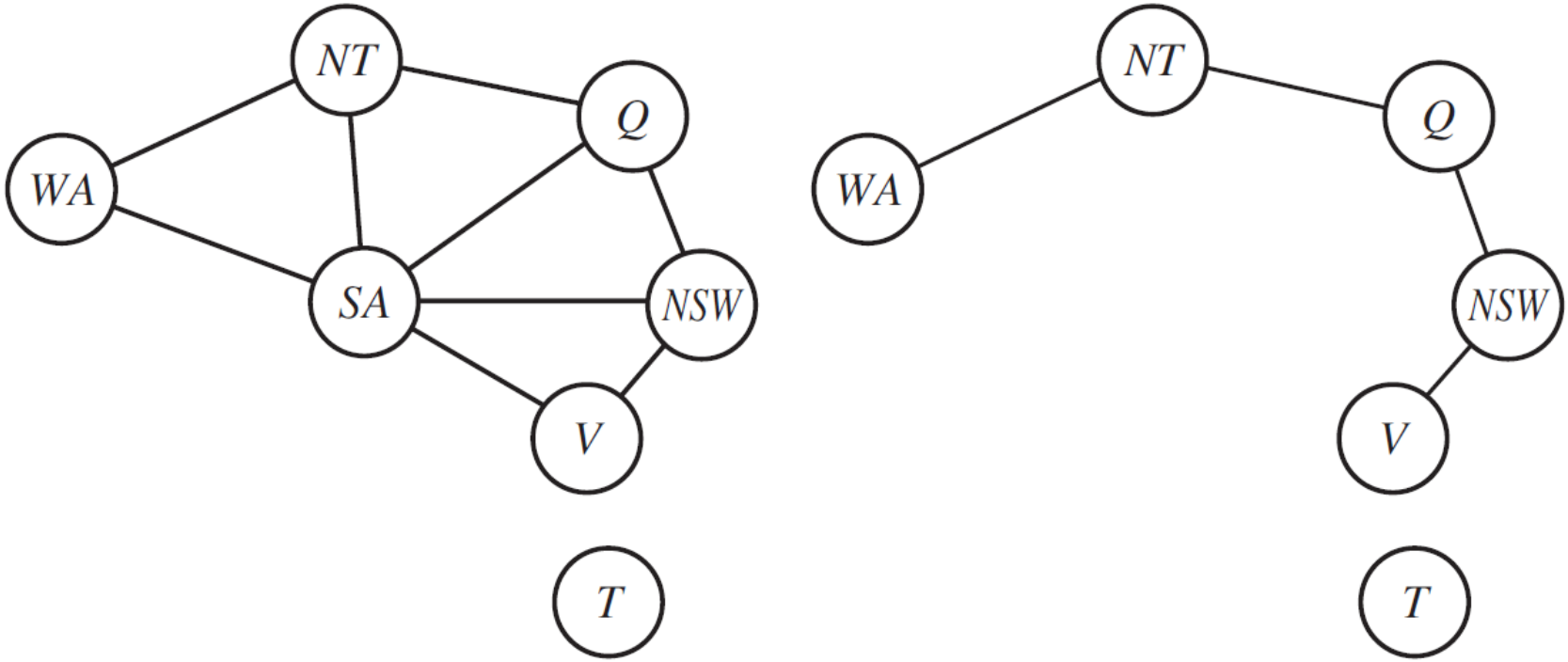
Topological order of the nodes in case of tree-structured CSPs:

- Tree structure leads to linear time



- Now try to reduce problems to trees

Using the structure of problems



Reduce graphs to trees

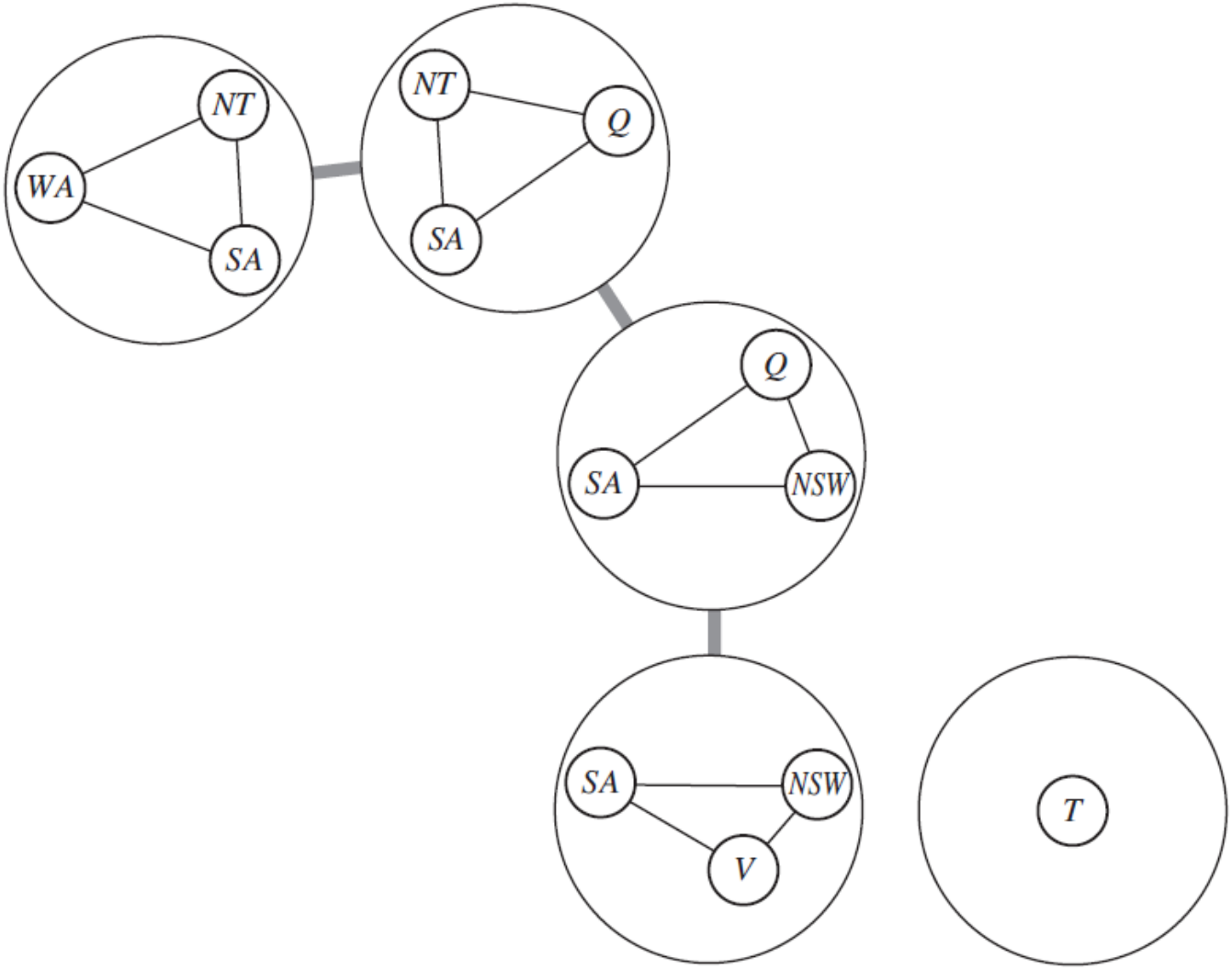
- Delete nodes: Assign values to some variables, so that remaining variables form a tree

Using the structure of problems

Tree decomposition: Devide and conquer

- Each variable in the original problem appears in one of the sub-problems
- 2 constrained variables have to appear (with the constraint) in at least one of the sub-problems
- If a variable appears in two sub-problems, it needs to be present in each sub-problem along the connecting path

Using the structure of problems



Summary

- CSPs: state represented by variable-value pairs
- Set of constraints on variables (unary, binary, and higher-order)

- Backtracking = depth first search + test

- Min-conflicts heuristics are very successful and easy

- Reduction of complexity by reduction to trees instead of graphs