

Constraint Satisfaction Problems



Berlin Chen
Department of Computer Science & Information Engineering
National Taiwan Normal University



References:

1. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Chapter 5
2. S. Russell's teaching materials

Introduction

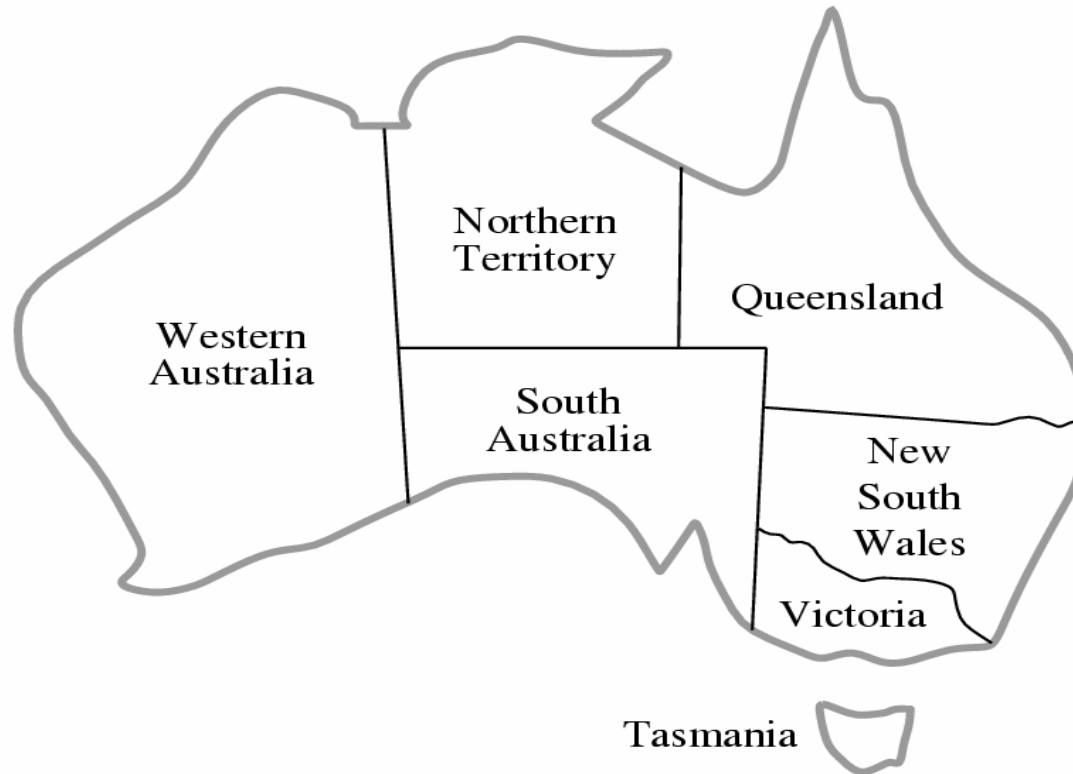
- Standard Search Problems
 - State is a “black box” with no discernible internal structure
 - Accessed by the goal test function, heuristic function, successor function, etc.
- Constraint Satisfaction Problems (CSPs)
 - State and goal test conform to a standard, structured, and very simple representation
 - State is defined by variables X_i with values v_i from domain D_i
 - Goal test is a set of constraints C_1, C_2, \dots, C_m , which specifies allowable combinations of values for subsets of variables
 - Some CSPs require a solution that maximizes an objective function

Derive heuristics
without
domain-specific
knowledge

Introduction (cont.)

- Consistency and completeness of a CSP
 - Consistent (or called *legal*)
 - An assignment that does not violate any constraints
 - Complete
 - Every variable is assigned with a value
 - A “**solution**” to a CSP is a complete assignment satisfying all the constraints

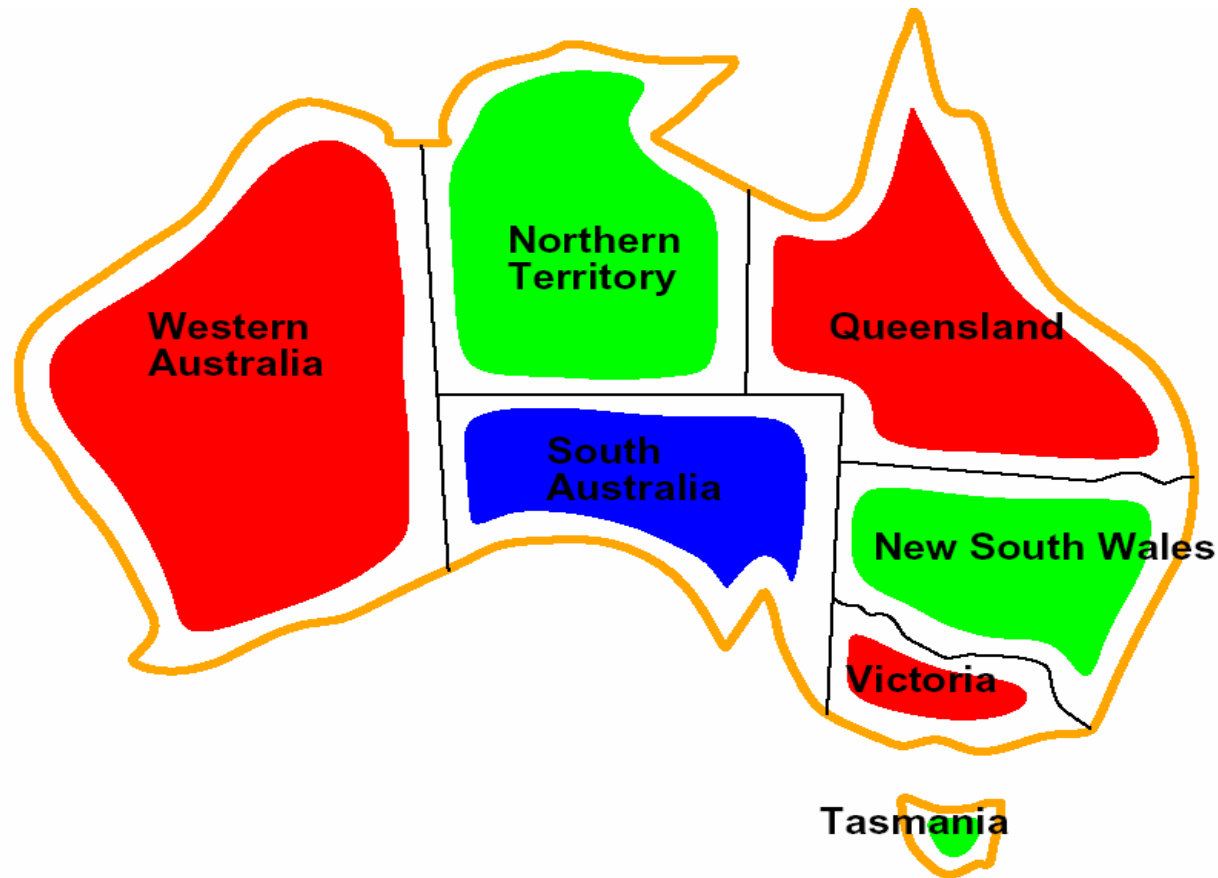
Example: Map-Coloring Problem



- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i = \{red, green, blue\}$
- Constraints: neighboring regions must have different colors

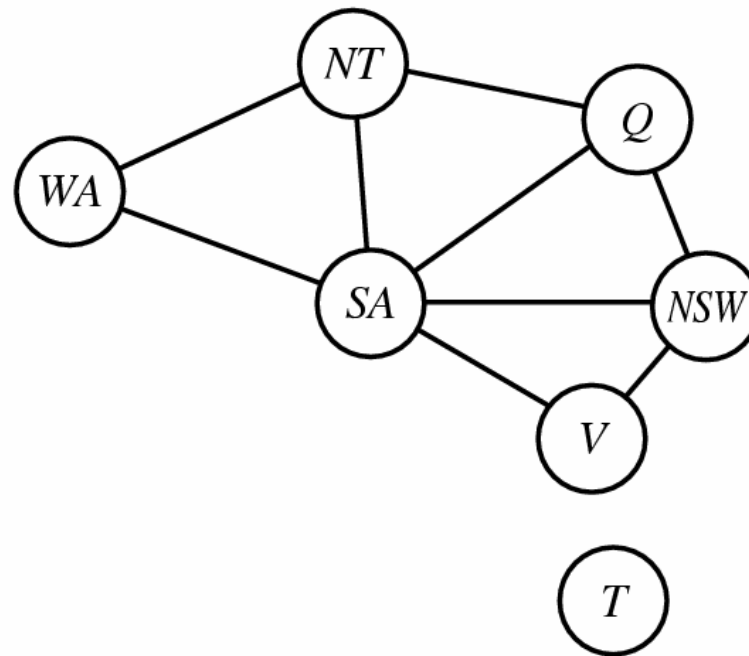
Example: Map-Coloring Problem (cont.)

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



Example: Map-Coloring Problem (cont.)

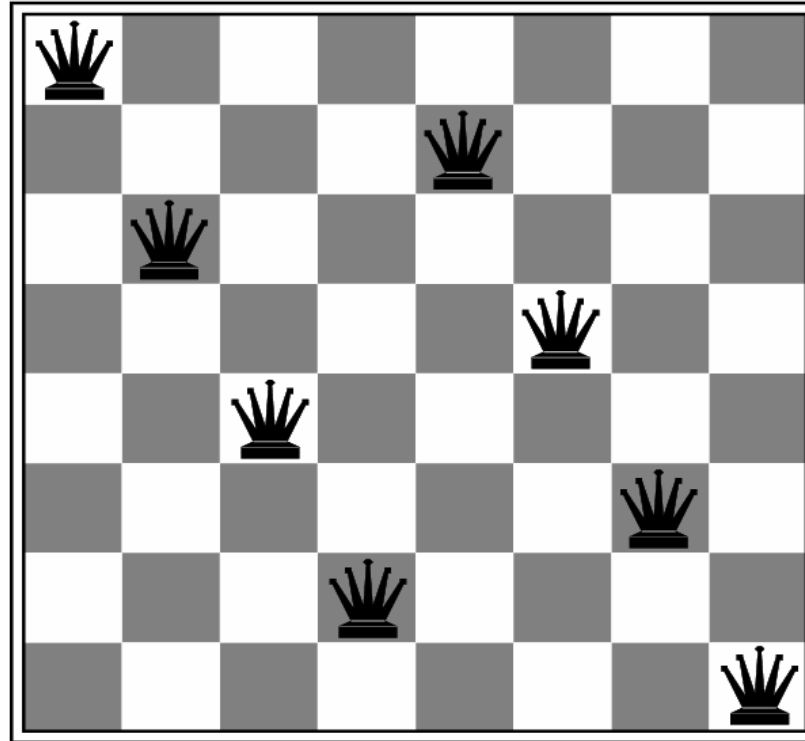
- The CSP can be visualized as a **Constraint Graph**
 - Nodes: correspond to *variables*
 - Arcs: correspond to *constraints*



Constraint Graph

- A visualization of representation of (binary) constraints

Example: 8-Queens Problem



- Variables: Q_1, Q_2, \dots, Q_8
- Domains: $D_i = \{1, 2, \dots, 8\}$
- Constraints: no queens at the same row, column, and diagonal

Benefits of CSPs

- Conform the problem representation to a standard pattern
 - *A set of variables with assigned values*
- Generic heuristics can be developed with no domain-specific expertise
- The structure of constraint graph can be used to simplify the solution process
 - *Exponential reduction*

Formulation

- **Incremental formulation**
 - **Initial state:** empty assignment { }
 - **Successor function:** a value can be assigned to any unassigned variables, provided that no conflict occurs
 - **Goal test:** the assignment is complete
 - **Path cost:** a constant for each step
- **Complete formulation**
 - Every state is a complete assignment that may or may not satisfies the constraints
 - Local search (such as hill-climbing) can be applied

CSPs can be formulated as search problems

Variables and Domains

- Discrete variables

- Finite domains (size d)

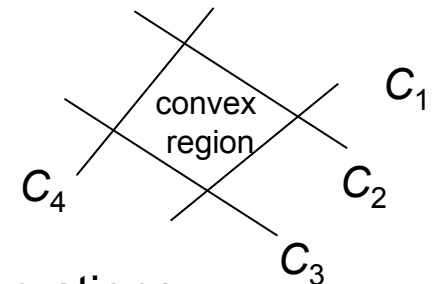
- E.g., color-mapping (d colors), Boolean CSPs (variables are either true or false, $d=2$), etc.
 - Number of complete assignment: $O(d^n)$ (exponential in number of variables)

- Infinite domains (integers, strings, etc.)

- Job scheduling, variables are start and end days for each job
 - A constraint language is needed, e.g., $StartJob_1 + 5 \leq StartJob_3$
 - Linear constraints are solvable, while nonlinear constraints undecidable
 - How to convert to a finite-domain problem ?

- Continuous variables

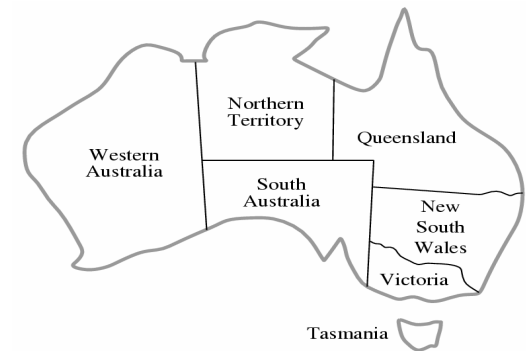
- E.g., start and end times for Hubble Telescope observations
 - Linear constraints are solvable in polynomial time by **linear programming** methods



Constraints

absolute
constraints

- Unary constraints
 - Restrict the value of a single variable
 - E.g., $SA \neq green$
 - Can be simply preprocessed before search
- Binary constraints
 - Restrict the values of a pair of variables
 - E.g., $SA \neq WA$
 - Can be represented as a constraint graph
- High-order constraints
 - Three or more variables are involved when the value-assigning constriction is considered
 - E.g., column constraints in the cryptarithmic problem
- Preference (soft) constraints
 - A cost for each variable assignment
 - E.g., the university timetabling problem
 - Can be viewed as constrained optimization problems

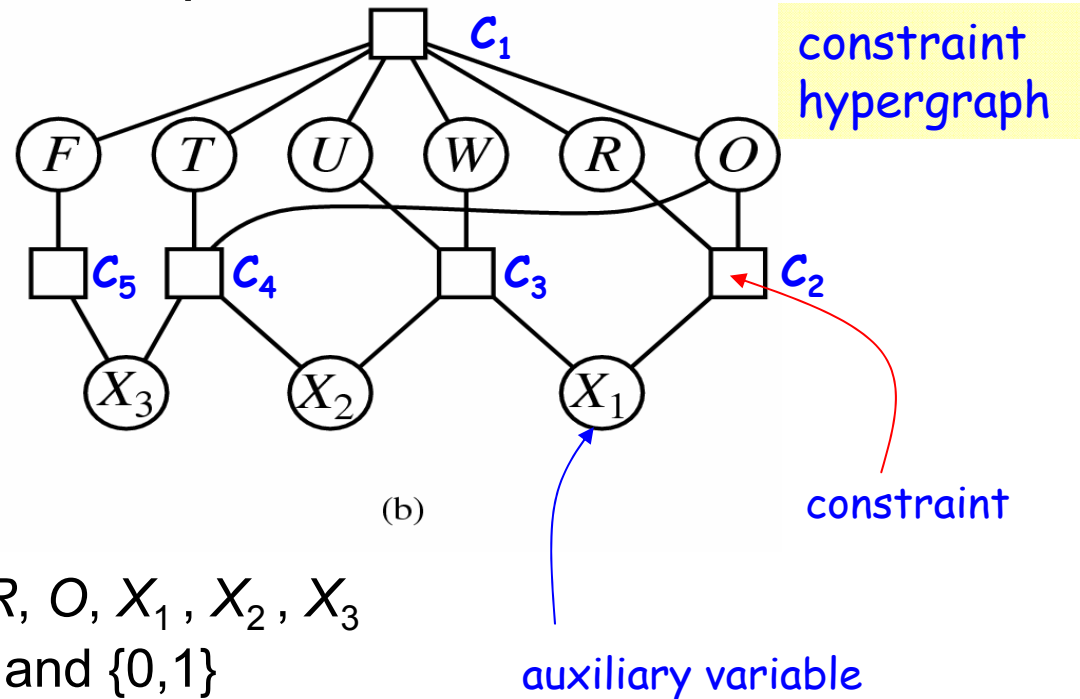


Constraints (cont.)

- Example: the cryptarithmic problem (high-order constraints)

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

(a)

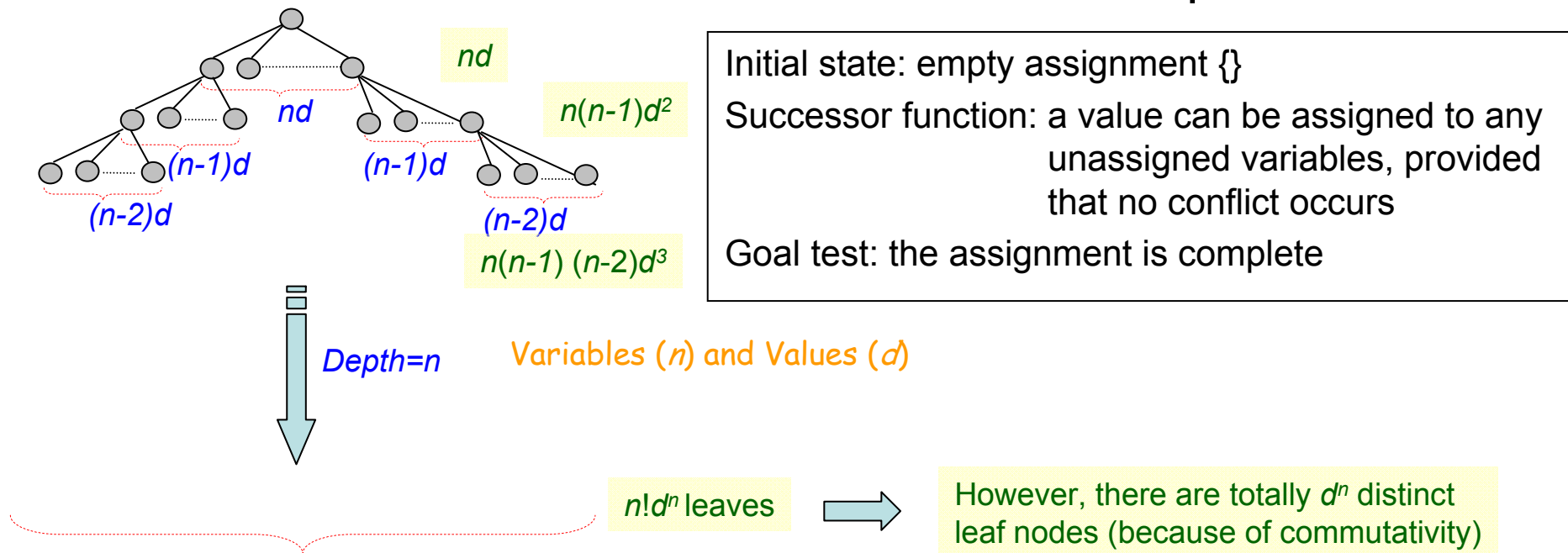


(b)

- Variables: $F, T, U, W, R, O, X_1, X_2, X_3$
- Domains: $\{0, 1, 2, \dots, 9\}$ and $\{0, 1\}$
- Constraints:
 - $Alldiff(F, T, U, W, R, O)$ C_1
 - $O+O=R+10 \cdot X_1$ C_2
 - $X_1+W+W=U+10 \cdot X_2$ C_3
 - $X_2+T+T=O+10 \cdot X_3$ C_4
 - $X_3=F$ C_5

Standard Search Approach

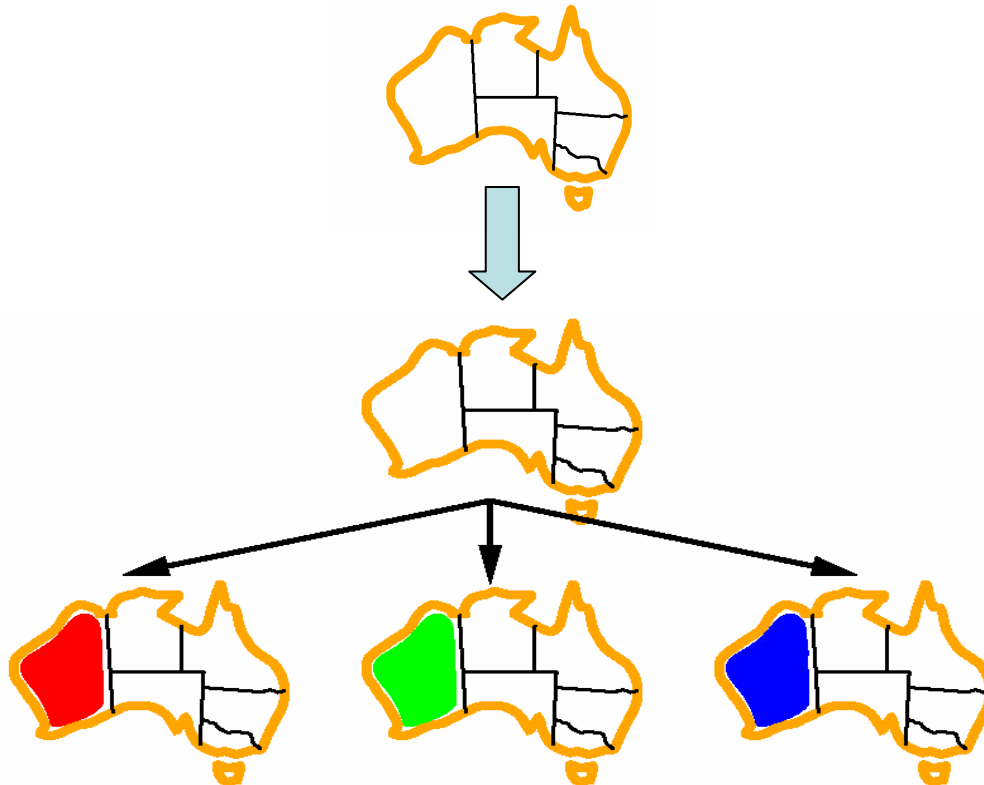
- If **incremental formulation** is used
- Breadth-first search with search tree with depth limit n



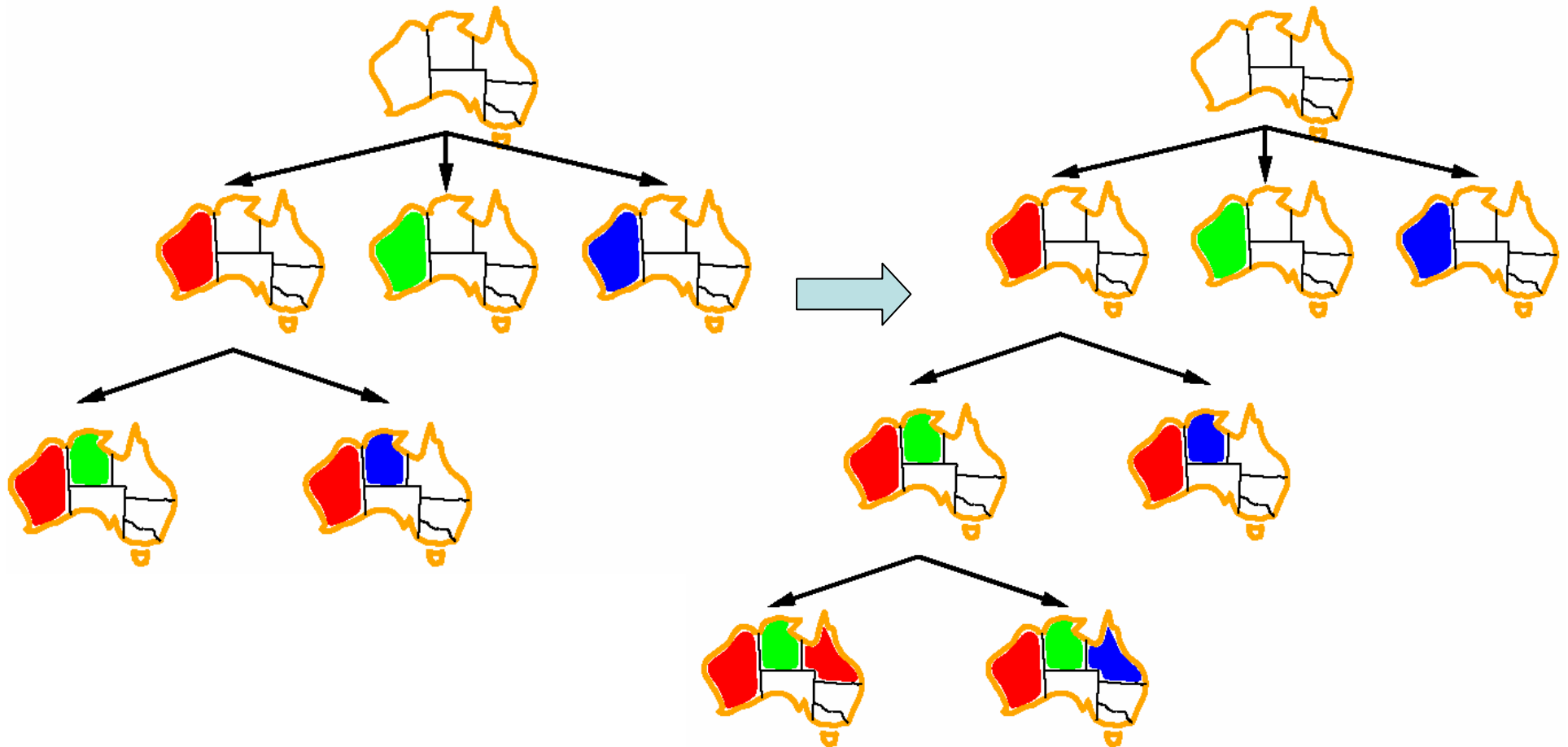
- Every solution appears at depth n with n variable assigned
- DFS (or depth-limited search) also can be applied (smaller space requirement)
- The order of assignment is not important

Backtracking Search

- **DFS** for CSPs (uninformed search)
 - One variable is considered orderly at a time (level) for expansion
 - Backtrack when no legal values left to assign
- The basic uniformed search for CSPs



Backtracking Search (cont.)



Backtracking Search (cont.)

- Algorithm

```
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 ≠ failure then return result
```

```
      remove {var = value} from assignment // if result=failure
```

```
  return failure
```

a list of unassigned variables

decide which variable
to be assigned a value

decide which value
to be used

- When it fails: back up to the preceding variable and try a different value of it
 - Chronological backtracking

Improving Backtracking Efficiency

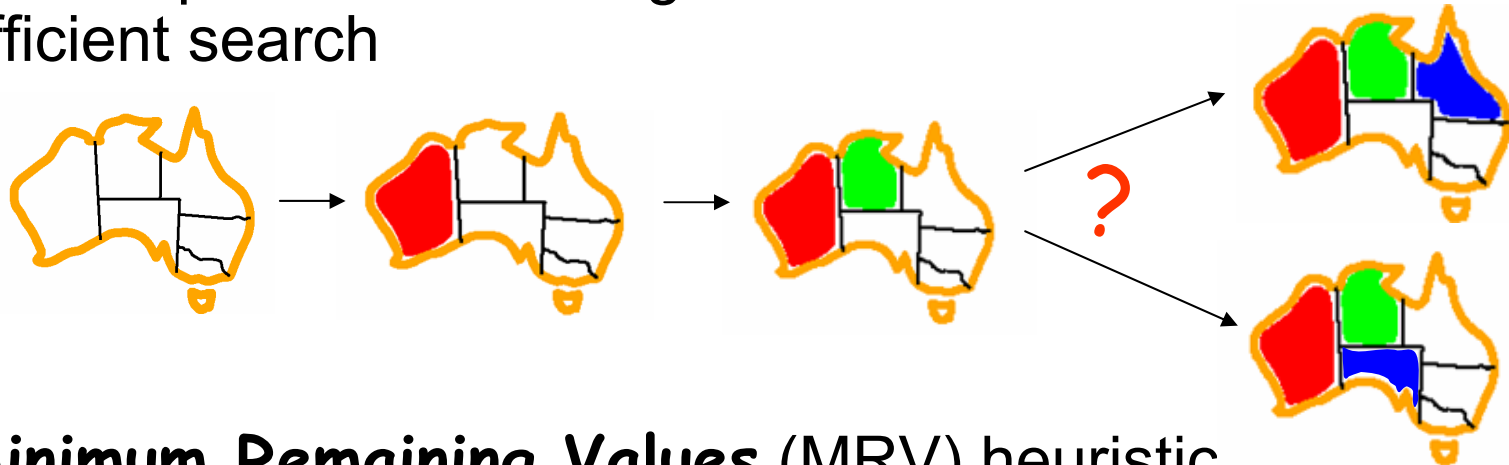
- General-purpose methods help to speedup the search
 - What variable should be considered next?
 - In what order should variable's values be tried?
 - Can we detect the inevitable failure early?
 - Can we take advantage of problem structure?

Improving Backtracking Efficiency (cont.)

- Variable Order
 - Minimum remaining value (MRV)
 - Also called "most constrained variable", "fail-first"
 - Degree heuristic
 - Act as a "tie-breaker"
- Value Order
 - Least constraining value
 - If full search, value order does not matter
- Propagate information through constraints
 - Forward checking
 - Constraint propagation

Variable Ordering

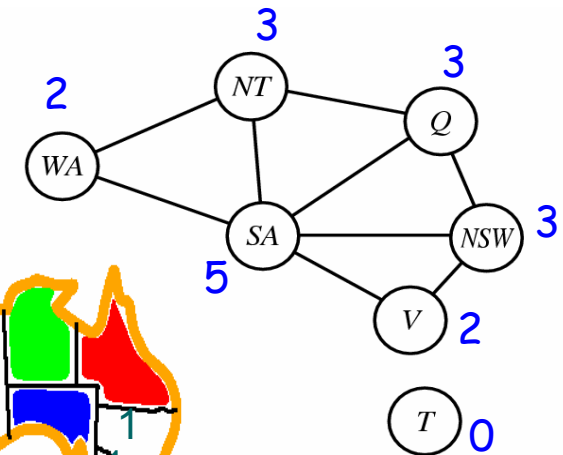
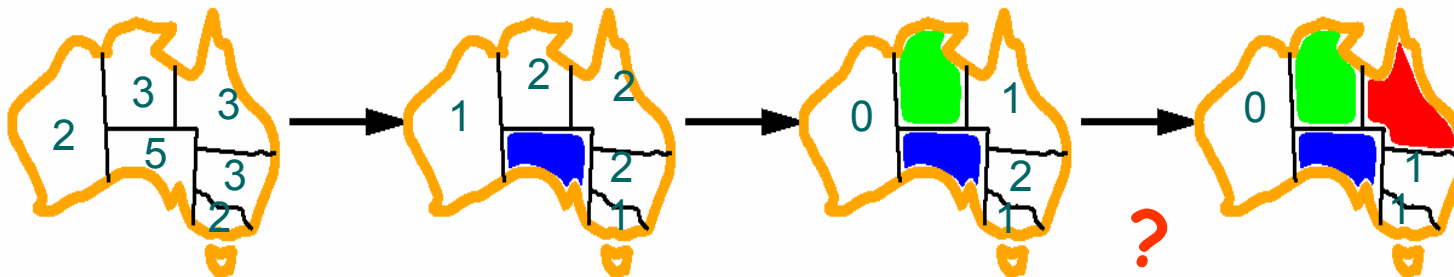
- The simple static ordering seldom results in the most efficient search



- **Minimum Remaining Values (MRV)** heuristic
 - Also called “most constrained variable” or “fail-first” heuristic
 - Choose the **variable** with the most constraints (on values) from the remaining variables
 - If a variable X with zero legal values remained, MRV selects it and causes a failure immediately
 - The search tree can be therefore pruned
 - Reduce the number of branch factor at lower levels ASAP

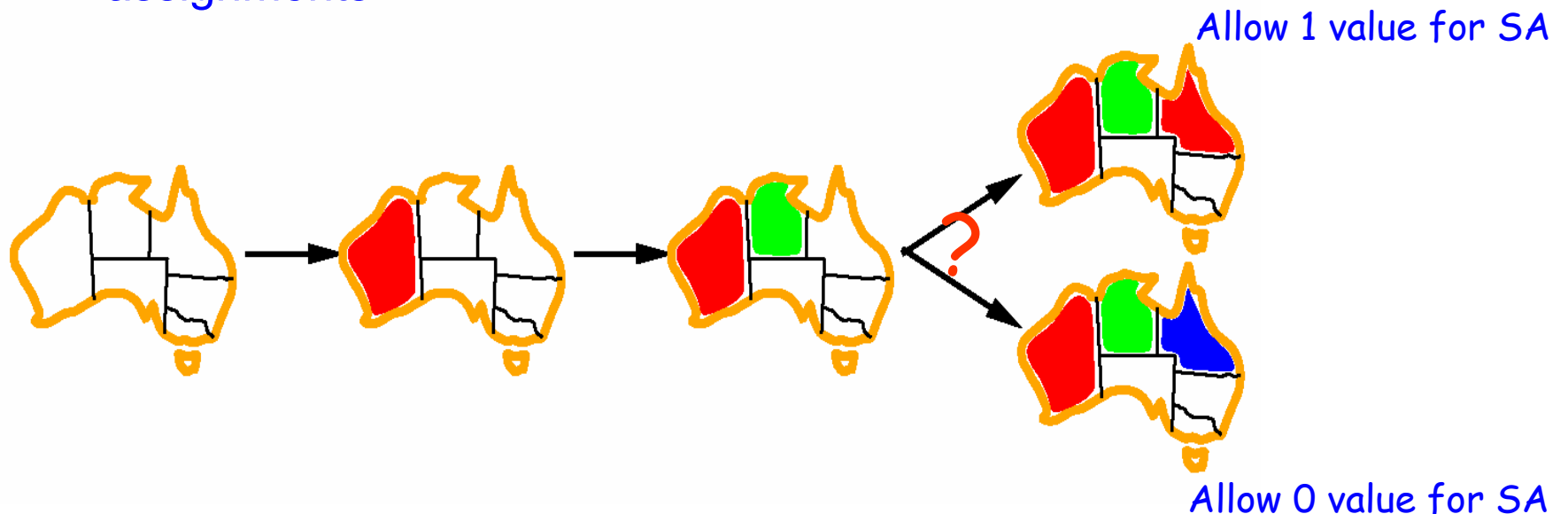
Variable Ordering (cont.)

- MRV doesn't help at all in choosing the first region to color in Australia
 - All regions have three legal colors
- So, the **degree heuristic** can be further applied
 - Select the **variable** that is involved in the largest number of constraints on other unassigned variables
 - A useful tie-breaker!
 - Reduce the branch factor on future choices



Value Ordering

- **Least-Constraining-Value** heuristic
 - Given a variable, choose the value that rules out the fewest choices of values for the remaining (neighboring) variables
 - I.e., **leave the maximum flexibility for subsequent variable assignments**



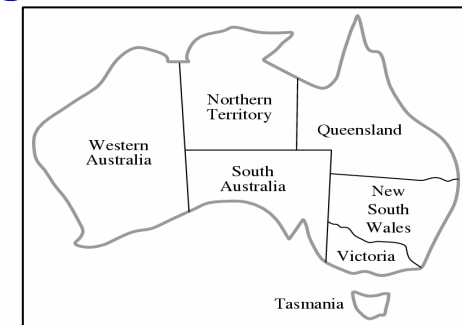
- If all the solutions (not just the first one) are needed, the value ordering doesn't matter

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
<i>n</i> -Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

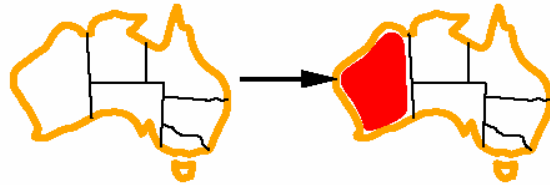
Figure 5.5 Comparison of various CSP algorithms on various problems. The algorithms from left to right, are simple backtracking, backtracking with the MRV heuristic, forward checking, forward checking with MRV, and minimum conflicts local search. Listed in each cell is the median number of consistency checks (over five runs) required to solve the problem; note that all entries except the two in the upper right are in thousands (K). Numbers in parentheses mean that no answer was found in the allotted number of checks. The first problem is finding a 4-coloring for the 50 states of the United States of America. The remaining problems are taken from Bacchus and van Run (1995), Table 1. The second problem counts the total number of checks required to solve all *n*-Queens problems for *n* from 2 to 50. The third is the “Zebra Puzzle,” as described in Exercise 5.13. The last two are artificial random problems. (Min-conflicts was not run on these.) The results suggest that forward checking with the MRV heuristic is better on all these problems than the other backtracking algorithms, but not always better than min-conflicts local search.

Forward Checking

- Not only consider constraints on a variable!
- But propagate constraint information from assigned variables to connected unassigned variables
- Keep track of remaining legal values for unsigned variables, and terminate the search when any variable has no legal values
 - Remove the inconsistent value of the unassigned variable
 - Before searching is performed on the unsigned variables



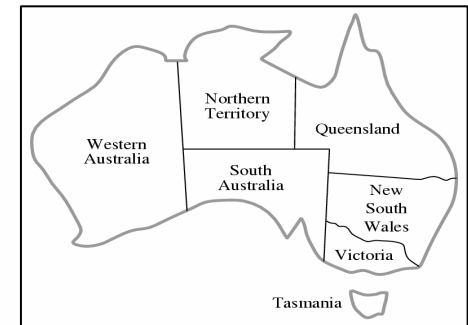
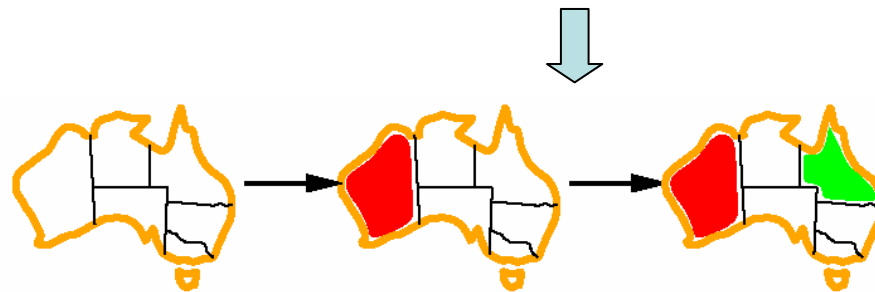
Forward Checking (cont.)



Note: MRV, degree heuristic etc., were not used here



after
WA=red



after
Q=green

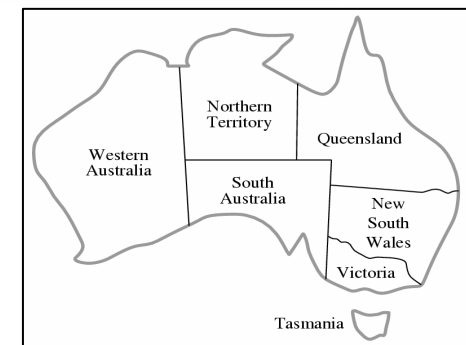
Forward Checking (cont.)



after
V=blue

Forward checking doesn't provide early detection for all inconsistency

- *NT* and *SA* can't both be blue



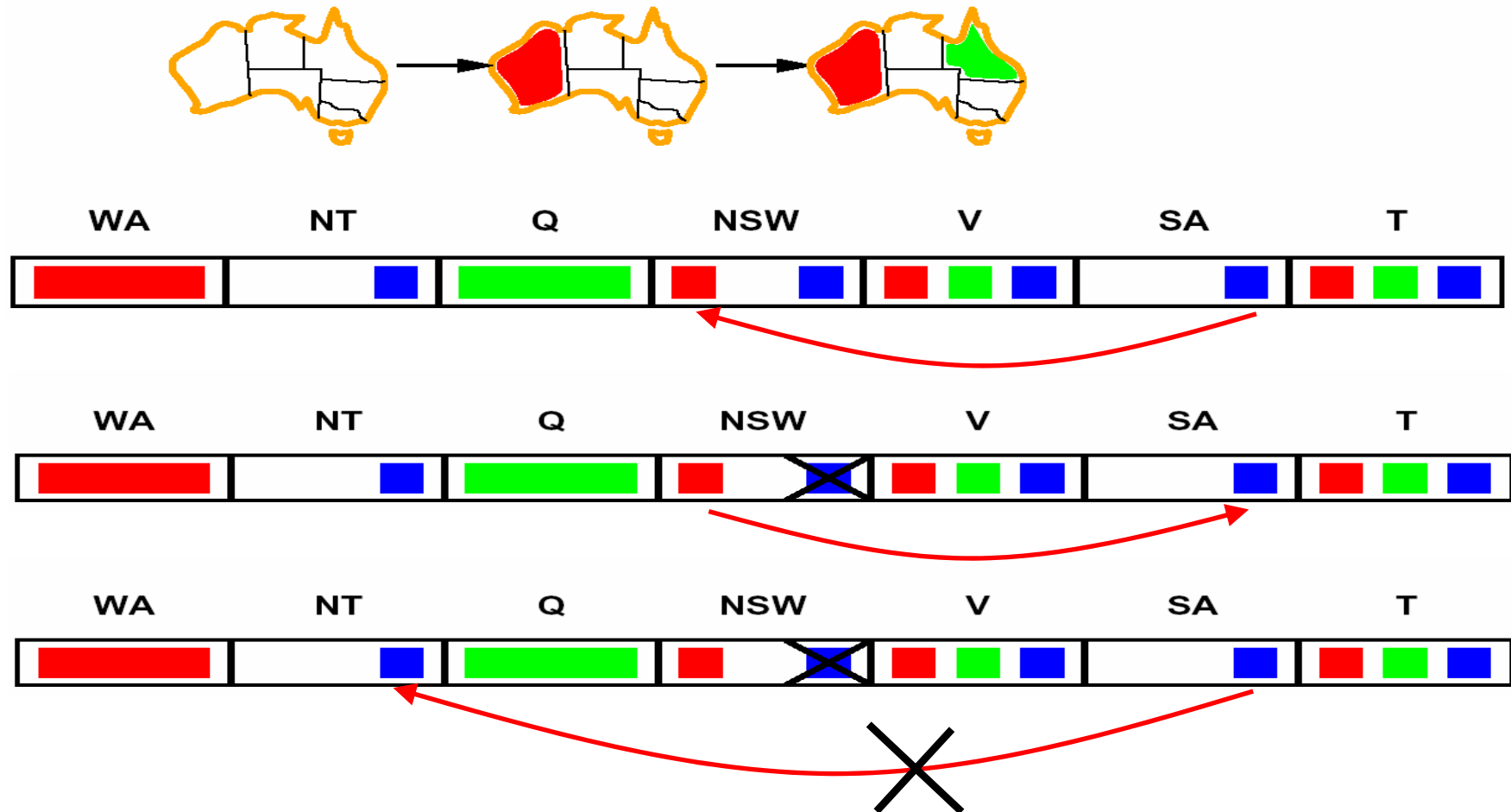
Constraint Propagation

- Repeated enforce constraints locally
- Propagate the implications of a constraint on one variable onto other variables
- Method
 - Arc consistency

Arc Consistency

- $X \rightarrow Y$ is consistent iff
 - for every value x of X there is some value y of Y that is consistent (allowable)
 - A method for implementing constraint propagation exists
 - Substantially stronger than forward checking

Arc Consistency (cont.)



- 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 (cont.)

- Algorithm $O(n^2 d^3)$

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* $O(n^2)$ $C_2^n = n \times (n-1)$

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*

If some values of a nodes X_i is removed, arcs pointing to it (X_k, X_i) must be reinserted on the queue for checking again

$O(d)$ each variable at most has d values to be removed

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff we 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 the constraint between X_i and X_j

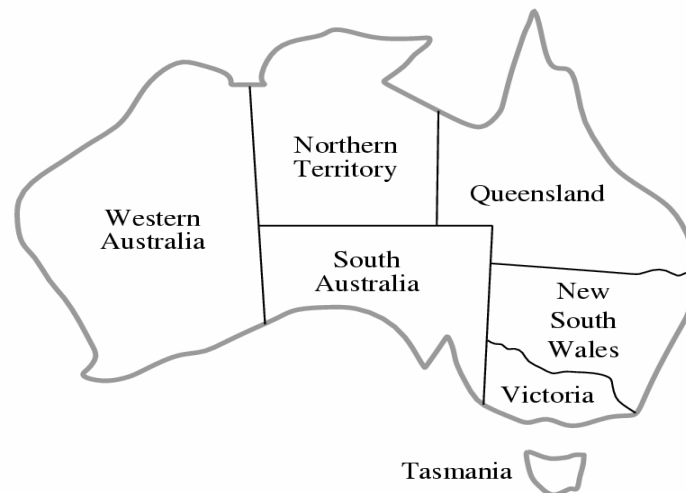
then delete x from DOMAIN[X_i]; *removed* \leftarrow true $O(d)$

return *removed*

d : number of values in the domain of each variable

Arc Consistency (cont.)

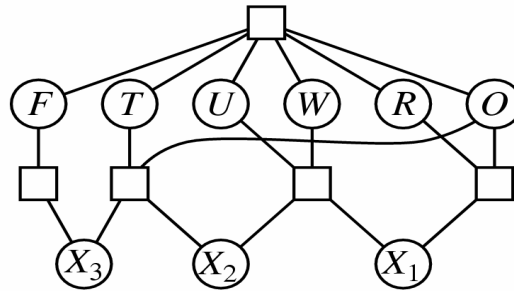
- Arc consistency doesn't reveal every possible inconsistency !
 - E.g. a particular assignment $\{WA=red, NSW=red\}$ which is inconsistent but can't be found by arc consistency algorithm
 - *NT, SA, Q* have two colors left for assignments
 - Arc consistency is just 2-consistency
 - 1-consistency, 2-consistency, ..., k-consistency, etc.



Handling Special Constraints

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

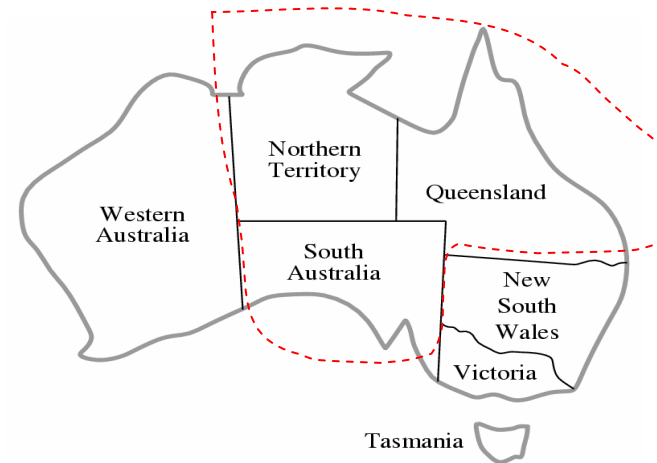
(a)



(b)

Alldiff (F, T, U, W, R, O)
 # variables m
 # value n
 $m > n$?

Alldiff (NT, SA, Q)
 # variables $m=3$
 # value $n=2$
 $m > n$?

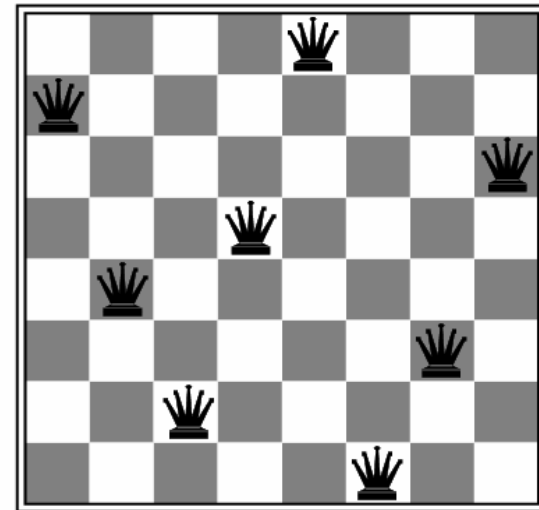
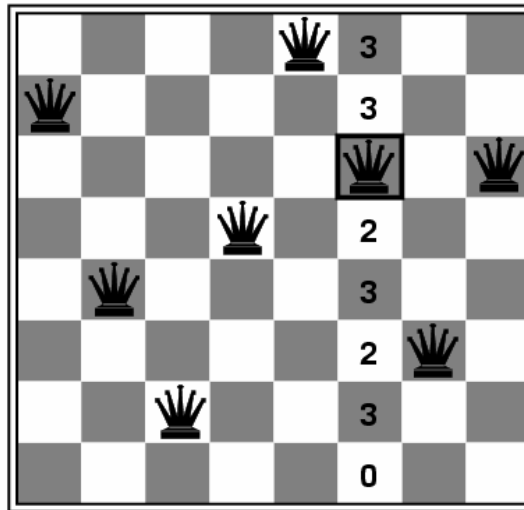
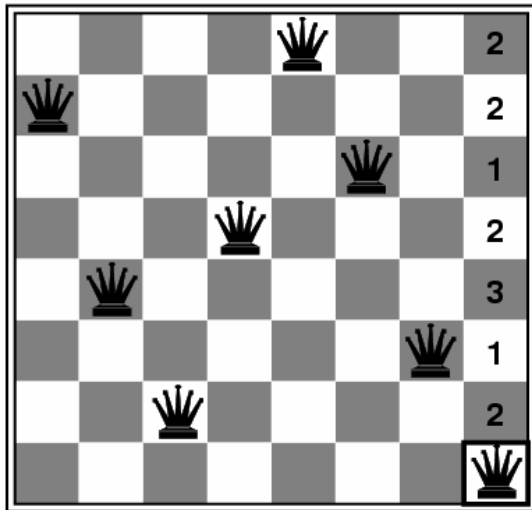


Local Search for CSPs

- If **complete formulation** is used
- Local search can easily be extended to CSPs with objection functions
 - Hill-climbing, simulated annealing etc. can be applied
- Method
 - Allow states with unsatisfied constraints
 - **Operators**
 - **reassign variable values**
 - **Variable selection**
 - **Randomly select any conflict variable**
 - E.g. the “min-conflicts” heuristic
 - For a given variable, selecting the value that results the minimum number of conflicts with other variables
 - E.g., hill-climbing with $h(n)$ =total number of violated constraints

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♠	13	16	13	16
♠	14	17	15	♠	14	16	16
17	♠	16	18	15	♠	15	♠
18	14	♠	15	15	14	♠	16
14	14	13	17	12	14	12	18

Local Search for CSPs (cont.)



- Especially suitable for problems for on-line settings

Local Search for CSPs (cont.)

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for *csp*

initialization (randomly generated or ...)

for $i = 1$ to *max_steps* **do**

if *current* is a solution for *csp* **then return** *current*

randomly select a variable

var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]

value ← the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)

set *var* = *value* in *current*

select the value of the variable
with minimum conflicts

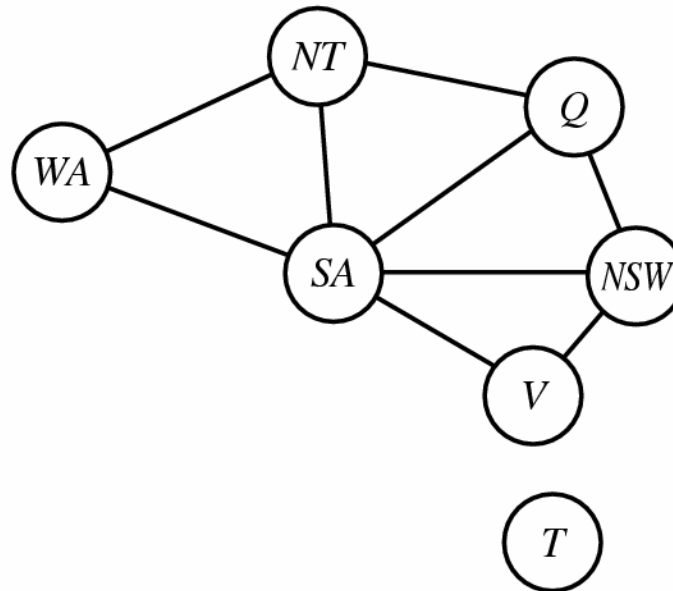
return *failure*

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
<i>n</i> -Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Figure 5.5 Comparison of various CSP algorithms on various problems. The algorithms from left to right, are simple backtracking, backtracking with the MRV heuristic, forward checking, forward checking with MRV, and minimum conflicts local search. Listed in each cell is the median number of consistency checks (over five runs) required to solve the problem; note that all entries except the two in the upper right are in thousands (K). Numbers in parentheses mean that no answer was found in the allotted number of checks. The first problem is finding a 4-coloring for the 50 states of the United States of America. The remaining problems are taken from Bacchus and van Run (1995), Table 1. The second problem counts the total number of checks required to solve all *n*-Queens problems for *n* from 2 to 50. The third is the “Zebra Puzzle,” as described in Exercise 5.13. The last two are artificial random problems. (Min-conflicts was not run on these.) The results suggest that forward checking with the MRV heuristic is better on all these problems than the other backtracking algorithms, but not always better than min-conflicts local search.

Problem Structure

- The structure of the problem represented by the constraint graph can be used to find solutions quickly



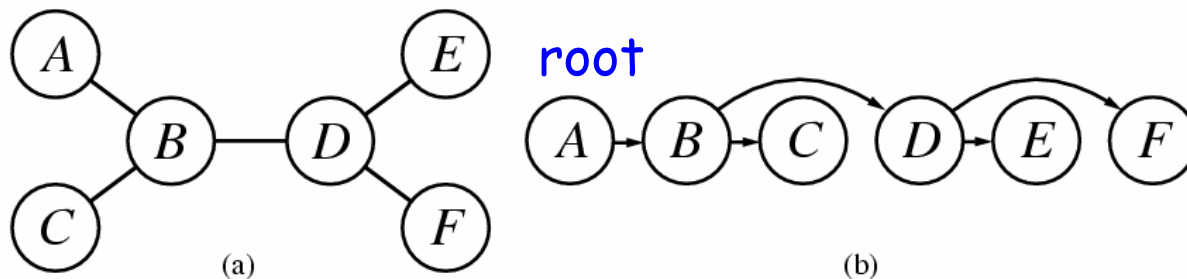
- E.g., Tasmania and the mainland are independent sub-problems
- Identify the connected components (as sub-problems) of constraint graph to reduce the solution time

Problem Structure (cont.)

- Suppose that each sub-problem has c variables out of n total
 - With decomposition
 - Worse-case solution cost: $n/c \cdot d^c$ linear in n
 - Without decomposition
 - Worse-case solution cost: d^n exponential in n
 - E.g., $n=80, d=2, c=20$
 - $2^{80}=40$ billion year at 10 million nodes/sec
 - $4 \times 2^{20}=0.4$ seconds at 10 million nodes/sec
- Completely independent sub-problems are rare
 - Sub-problems of a CSP are often connected

Tree-Structured CSPs

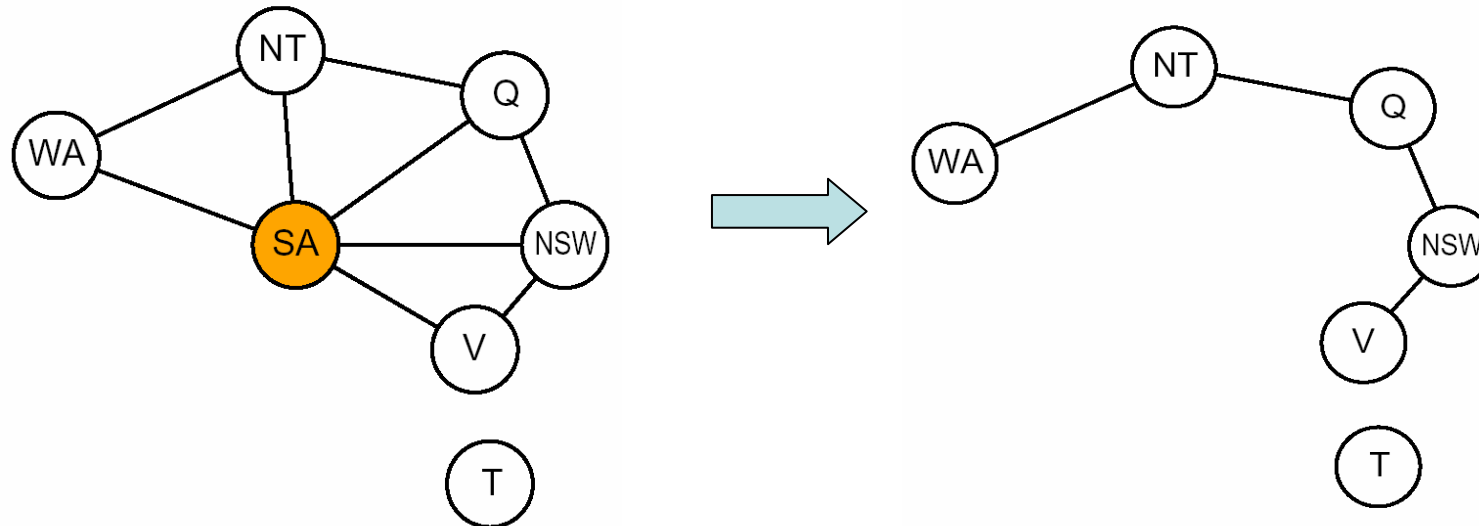
- Tree-Structured CSPs are the simplest ones
 - Can be solved in time linear in the number of variables



- Algorithm for tree-structured CSPs
 1. Choose a variable as the 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 arc consistency to the arc (X_i, X_j) , where X_i is the parent of X_j , remove the values from $\text{Domain}[X_j]$ as necessary $O(nd^2)$, if no loops Why in reverse order ?
 3. For j from 1 to n , assign X_j consistently with parent X_i

Reducing Constraint Graphs to Trees

- Method 1 $O(d^{n-c}d^2)$
 - Initiate a subset of variables S (cycle cutset, with size c) such that the remaining constraint graph is a tree
 - Prune the domains of the remaining variables that are inconsistent with S $O((n-c)d^2) \leftarrow$ Tree-structure CSP
 - If the remaining CSP has a solution, return it together with the assignment for S $O(d^c)$



- E.g., if the value assigned to SA is a wrong one, do again !

Reducing Constraint Graphs to Trees (cont.)

- Method 2
 - Construct a tree decomposition of the constraint graph into a set of connected subproblems
 - Each subproblem is solved independently and the resulting solutions are then combined
 - Properties
 - Every variable must appear in at least one subproblem
 - Two variables connected by a constraint must appear together
 - A variable connecting some subproblems must appear in all of them

