EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE FAKULTÄT

Artificial Intelligence
Chapter 6: Constraint Satisfaction Problems

Andreas Zell

After the Textbook: Artificial Intelligence,
A Modern Approach
by Stuart Russell and Peter Norvig (3rd Edition)

6 Constraint Satisfaction Problems

- In *search*, each state is a structureless black-box
- Factored representation of a problem:
  - Problem represented as *variables* which take a *value*
  - The legal values for a variable are constrained by a set of *constraints* between variables
  - Problem is solved when a value is assigned to each variable to satisfy all its constraints
  - This is a Constraint Satisfaction Problem (CSP)
- CSP search algorithms use the structure of CSPs for general-purpose search heuristics
  - <u>Idea</u>: identify variable/value combinations that violate constraints.

6.1 Constraint Satisfaction Problems
Formal Definition

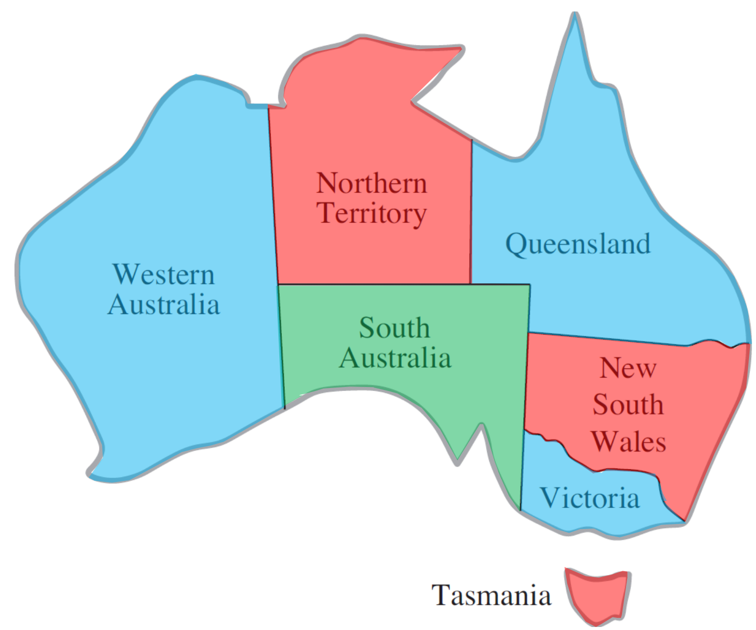
EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Variables **X:** set of n variables: $X = \{X_1, \ldots, X_n\}$
  - D is set of domains; one per variable: $D = \{D_1, \ldots D_n\}$
  - X_i takes values in domain $D_i = \{v_i\}$: i.e. $X_i \in D_i$
  - *Assignment*: set of values for variables $\{X_1 = v_1, X_5 = v_5\}$
  - *Complete Assignment*: assigns values to all variables
  - *Partial Assignment*: assigns values only to a subset
- Constraints **C**: set of m constraints $C = \{C_1, \ldots, C_m\}$ restrict assignments (e.g. $X_1 \neq X_2$ unless $X_1 = X_2 = 0$)
  - *Consistent assignment*: doesn't violate any constraint
  - *Solution*: a consistent complete assignment
- Objective Function (optional): valuation function
  - *Best Solution*: maximizes the objective function

6.1 Constraint Satisfaction Example
Map K-Coloring Problems

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

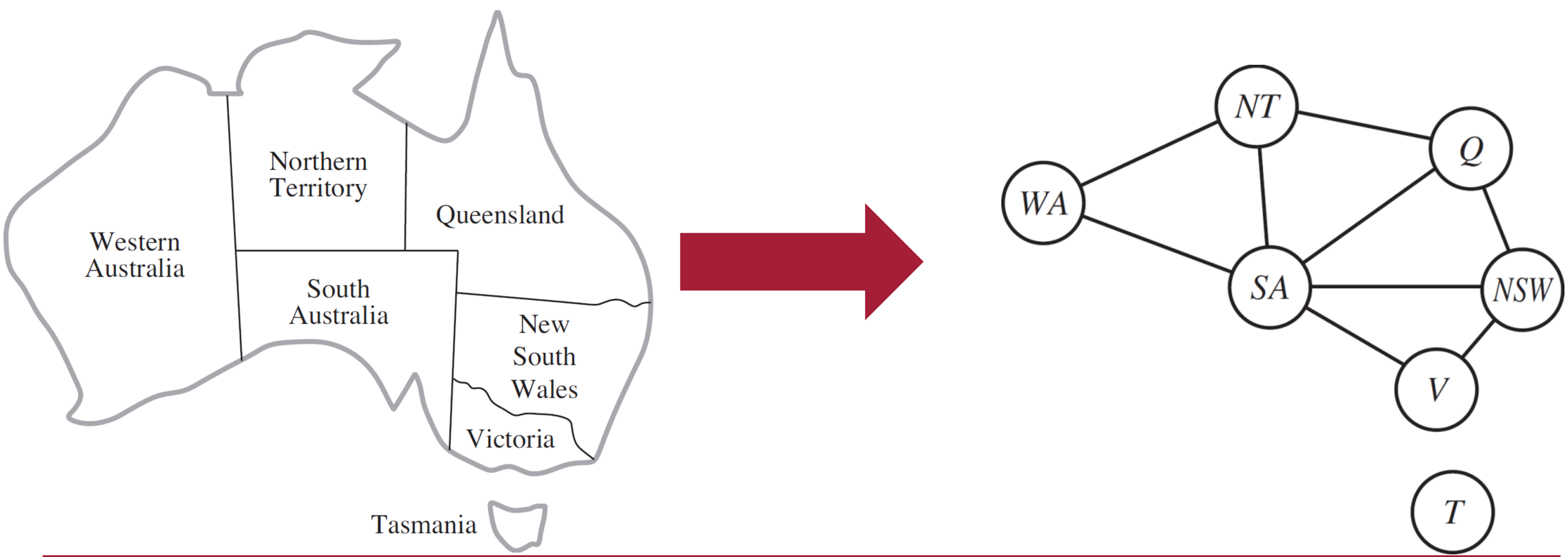
- Map K-coloring problem: every state on a map must be colored with one of K colors
  - Variables: states (values are colors); ex. *WA, NT, SA*
  - Constraints: neighbors cannot be the same color; ex. (*WA≠NT*), (*NT ≠ SA*), (*WA ≠ SA*)



6.1 Constraint Satisfaction Problems
Constraint Graphs

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Constraint Graph**: graph representation of CSP
  - Variables represented by nodes
  - Constraints represented by arcs between nodes
    - Ex. In Australia coloring, arcs mean ‘differ in color’



6.1 Constraint Satisfaction Example Job-shop Scheduling

- Job-shop scheduling: need to choose an order of tasks to accomplish an overall job: e.g. Factory assembly
  - Each task expressed as integer variable for starting time
  - Constraints express duration of task & task ordering
  - Ex. Assembling car: *15* variables for tasks: install *2* axles, affix *4* wheels, tighten wheel nuts (*4*), affix caps (*4*), & inspect assembly
- Duration: we give task T_i a duration d_i
- Precedence: to say T_1 must be before T_2: $T_1+d_1 \leq T_2$
- Disjunctive constraints: allow us to say one task must be done before the other but order doesn't matter
  - Ex. If there is only *1* tool needed for an axle, it doesn't matter which axle is done first, but they can't overlap
  - $(T_1+d_1 \leq T_2)$ or $(T_2+d_2 \leq T_1)$

- Finite Domain: enumerable values (e.g. colors)
  - Constraints can enumerate allowable values
  - If $|D_i|=d$, there are $O(d^n)$ complete assignments
    - n-Queens: each queen can take n positions: $O(n^n)$
  - *Boolean CSP*: values either *true* or *false*
    - Includes some NP-Complete Problems (3-SAT)
- Infinite Domain: countable values (e.g. integers)
  - Constraint language used to express constraints; e.g. $X_1 + 5 \geq 3X_2$
  - Special algorithms can solve linear integer-CSPs
  - No algorithm can solve every nonlinear integer-CSP

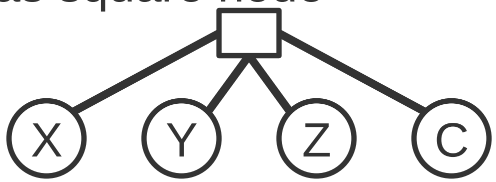
6.1 Variations on the CSP Formalism
Continuously-valued variables

- Infinite Uncountable Domains: (e.g. reals)
  - Occur in many real-world problems; ex. scheduling
- Linear programs: constraints are linear inequalities (or equalities).
  - There are polynomial algorithms for linear programs;
  - Interior point methods are polynomial
  - Simplex methods provide practical fast solutions
- Convex programs: constraints are inequalities on convex functions
  - E.g. quadratic and second-order conic programs
  - Polynomial algorithms again exist

6.1 Variations on the CSP Formalism
Degree of constraints

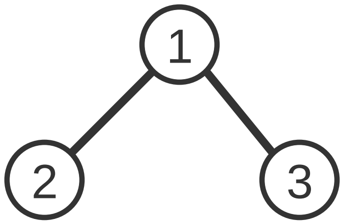
- Unary constraint: limits a single variable (e.g. $X_1 \geq 2$)
- Binary constraint: relationship between pair (e.g. $SA \neq NSW$)
- Higher-order (global) constraints: involve many variables
  - Auxiliary variables can be used to form constraint hypergraph
    - Ordinary variables are circular nodes
    - n-ary constraints represented as square node

X + Y + Z = C



  - Dual Transformation also transforms n-ary constraints to binary
    - Dual graph: each constraint becomes a node and an edge is added between every pair of constraints that share variables

1) X + Y + Z = C
2) X – Z = D
3) Y * C = E

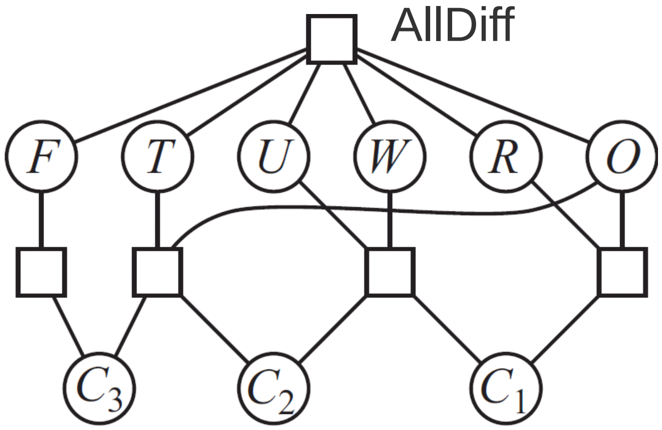


- Cryptarithmetic Puzzle: letters represent distinct digits in arithmetic expression

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

- All letters are different digits: AllDiff(T,W,O,F,U,R)
- Columns (including carries C_i) are arithmetically consistent

$$O \neq R \neq W \neq U \neq T \neq F$$
$$O + O = R + 10 * C_1$$
$$C_1 + W + W = U + 10 * C_2$$
$$C_2 + T + T = O + 10 * C_3$$
$$C_3 = F$$



6.1 Variations on the CSP Formalism
Types of Constraints

- Absolute constraints: constraints that any valid solution must satisfy
  - Eliminate potential variable assignments
- Preference constraints: constraints that one would like to be satisfied if possible
  - Give an implicit ranking over the space of solutions
  - Often, preferences can be encoded as a cost over variable assignments; i.e., assignments that satisfy more preferences have lower costs
  - Such problems are called constraint optimization problems (COP); ex. Linear programming

6.2 Constraint Propagation: Inference in CSPs

- Search Strategy: search over assignment space and use inference to eliminate inconsistencies
- Constraint Propagation: iteratively use constraints to reduce the domains of variables
  - Can be used as pre-processing before search
    - Sometimes this alone can find a solution
  - Can also be used during search (forward checking)
- Key Idea: local consistency
  - Variables & binary constraints form constraint graph
  - Enforce local consistency in graph’s components to remove inconsistent values
    - Ex. Iterate over every *3* node subset

6.2 Constraint Propagation: *K*-Consistency

- A CSP is *k*-consistent if, for any k-1 variables and any consistent assignment to them, a consistent value can be assigned to the k^{th} variable
  - Node consistency (1-consistency): variables are self-consistent
  - Arc consistency (2-consistency): every variable is consistent with its neighbors
  - Path consistency (3-consistency): every pair of neighbors can always be extended to a 3^{rd}

6.2 Constraint Propagation: Node Consistency

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- X_i is node-consistent if all its values satisfy the node's unary constraints.
  - Ex. If people in South Australia (*SA*) dislike *green*, we can eliminate green from *SA*'s domain.
- CSP is node-consistent if all its variables are
- After enforcing node-consistency, all unary constraints can be removed from the CSP
- Further, all n-ary constraints can be converted into binary constraints.
  - Hence, we only consider CSP solvers for binary constraints

6.2 Constraint Propagation: Arc Consistency

- X_i is arc-consistent for X_j if, for all v in D_i, there is a w in D_j such that (v, w) satisfies constraint (X_i, X_j)
  - Example: Take X, Y in {*0,1,2,…,9*} with constraint $X = Y^2$
  - Using arc consistency on X reduces its domain to {*0,1,4,9*}
  - Applying it to Y reduces its values to {*0,1,2,3*}
- CSP is arc-consistent if every variable X_i is arc-consistent for every other variable X_j
- Revise enforces arc-consistency for a variable

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 $revised$

6.2 Constraint Propagation: Enforcing Arc Consistency: AC-3

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- When an inconsistency arc is detected, conflicting values are removed from source variable's domain
  - This may make formerly consistent arcs inconsistent
- AC-3 algorithm: uses a queue to (re-)check arcs
  - If values removed from D_i, all arcs to X_i are re-queued

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*

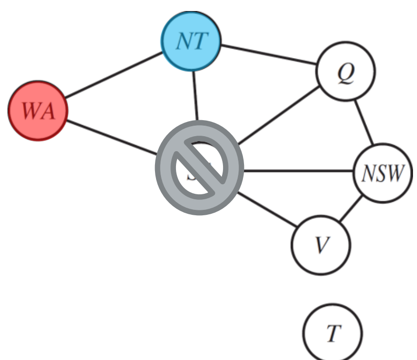
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

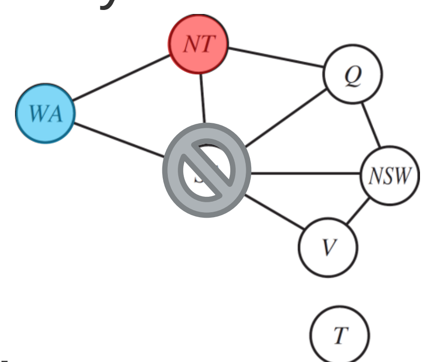
6.2 Constraint Propagation: Generalized Arc Consistency

- Worst-case: n^2 arcs are added at most d times (once per value) and checking arc is $O(d^2)$, so AC-3 is $O(n^2d^3)$
- X_i is generalized arc-consistent for n-ary constraint if, for every value v in D_i, there is an assignment to the remaining variables, which satisfies the constraint
  - Ex. Take X, Y, Z in {*1,2,3,4,5*} with constraint $X < Y < Z$
  - Applying arc-consistency to X makes its domain {*1,2,3*}
  - Applying it to Y makes its domain {*2,3,4*}
- CSPs include NP-complete problems, so Arc-consistency alone cannot find all inconsistencies
  - Ex. In coloring Australia with only 2-colors {*red*, *blue*} every variable is arc-consistent, but at least 3 colors are needed for a consistent assignment

6.2 Constraint Propagation: Path Consistency

- Path consistency uses implicit constraints inferred from triplets of variables
  - $\{X_i, X_j\}$ is *path consistent* for X_k if for every consistent assignment $\{X_i=a, X_j=b\}$, X_k still has a consistent value
  - Example: Australia *2*-color problem: {*red*, *blue*}
  - Enforcing path consistency for any connected triplet (e.g. *WA*, *SA*, & *NT*) will reveal the inconsistency inherent in this problem





- PC-2 algorithm used to achieve path consistency
  - Still cannot solve all CSPs;ex. some maps require 4 colors

6.2 Constraint Propagation: Strong k-Consistency

- A graph is strongly k-consistent if it is k, $(k\ -1)$, …, 2, & 1-consistent
- If a CSP with n variables is strongly n-consistent, it can be solved without backtracking…
  - Assigning k-th variable is always possible because graph is k-consistent for all k ➔ solution achieved in $O(n^2d)$
  - Unfortunately, establishing n-consistency is worst-case exponential in n, both in time & space
- Middle ground? Stronger consistency requires more time but reduces branching factor
  - It is possible but often impractical to calculate smallest k, for which k-consistency ensures no backtracking
  - Practically *2*- or *3*-consistency is used

6.2 Constraint Propagation: Global Constraints

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- AllDiff constraint: variables cannot have same value
  - If there are more variables than values ➔ unsatisfiable
  - If a variable has only 1 value, assign it and remove that value from other variables ➔ quick inconsistency detection
- Resource (*atmost*) constraint: limits the allocation of a resource; e.g. only *10* people for *4* tasks
  - Inconsistency checked by summing minimum of domains
  - Enforced by deleting maximum value of a domain when not consistent with minimum values of other domains.
- Contiguous domains are represented by bounds;
  - CSP is bounds-consistent if lower & upper of X can be satisfied by some value of Y when X & Y are constrained
  - Bounds propagation shrinks domain until consistent

6.2 Constraint Propagation: Example of AC-3 propagation

- <u>Sudoku</u> – logic puzzle in which digits must occur exactly once within each row, column & square

	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		

AllDiff($A1,A2,A3,A4,A5,A6,A7,A8,A9$)
AllDiff($B1,B2,B3,B4,B5,B6,B7,B8,B9$)
…
AllDiff($A1,B1,C1,D1,E1,F1,G1,H1,I1$)
AllDiff($A2,B2,C2,D2,E2,F2,G2,H2,I2$)
…
AllDiff($A1,A2,A3,B1,B2,B3,C1,C2,C3$)
AllDiff($A4,A5,A6,B4,B5,B6,C4,C5,C6$)

To Binary

$A1 \neq A2,\ A1 \neq A3, \ldots A1 \neq A9, A2 \neq A3, \ldots A2 \neq A9, \ldots A8 \neq A9$
$B1 \neq B2,\ B1 \neq B3, \ldots B1 \neq B9, B2 \neq B3, \ldots B2 \neq B9, \ldots B8 \neq B9$

……

6.2 Constraint Propagation: Example of AC-3 propagation

- Examine *E6*:
  - Original domain {*1,2,3,4,5,6,7,8,9*}
  - Row constraints exclude *7,8*
  - Column constraints exclude *5,6,2,9,3*
  - In-box constraints exclude *1*
  - Domain is now {*4*} and so *E6* must be *4*.

	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		

6.2 Constraint Propagation: Example of AC-3 propagation

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Examine *I6*:
  - Original domain {*1,2,3,4,5,6,7,8,9*}
  - Row constraints exclude *5,1,3*
  - Column constraints exclude *6,2,4,8,9,3*
  - In-box constraints exclude nothing else
  - Domain is now {*7*} and so *I6* must be *7*.

	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					**4**			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

6.2 Constraint Propagation: Example of AC-3 propagation

- Examine *A6*:
  - Original domain {*1,2,3,4,5,6,7,8,9*}
  - Row constraints exclude *3,2,6*
  - Column constraints exclude *5,4,8,9,7*
  - In-box constraints exclude nothing else
  - Domain is now {*1*} and so *A6* must be *1*.
- AC-3 continues to solve

	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					4			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1	7	3		

6.2 Constraint Propagation: Example of AC-3 propagation

- The puzzle is solved!
- Not all Sudoku puzzles can be solved by arc-consistency alone
- *Naked triples* – find *3* squares within a unit that contain same *3* numbers: {*1,8*}, {*3,8*}, & {*1,3,8*}
  - One of these *3* squares must contain *1, 3 & 8*
  - These are removed from domains of other squares
  - This is enforcing higher-order consistency

	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**

6.3 Search for CSPs
Applying Search to CSPs

- CSP as search (incremental formulation)
  - *Initial State*: empty assignment {}
  - *Successor Function*: assigns to unassigned variables (without creating conflicts)
  - *Goal Test*: is the current assignment complete?
  - *Path Cost*: constant cost (e.g. 1) at each step
- Solutions must be complete → depth n
  - Very amenable to depth-first search
- Solutions are path-independent → complete state formulation is possible
  - Every state is complete, but may not be consistent

6.3 Bactracking Search: Commutativity in CSPs

- Many CSPs require search; e.g. depth-limited s.
- Consider the full search tree used to solve CSPs
  - At first level, n variables can take d different values
  - At second, there are $(n\text{-}1) * d$ possibilities
  - …
  - The total tree has $n! * d^n$ leaves, but there are only d^n possible assignments for the CSP ➔ repeated work
- Generic search ignores important property of CSPs
  - commutativity: order of assignment in CSP is irrelevant
- CSP search algorithms generate successors by only considering a *single* variable at each search node!

6.3 Backtracking Search for CSPs

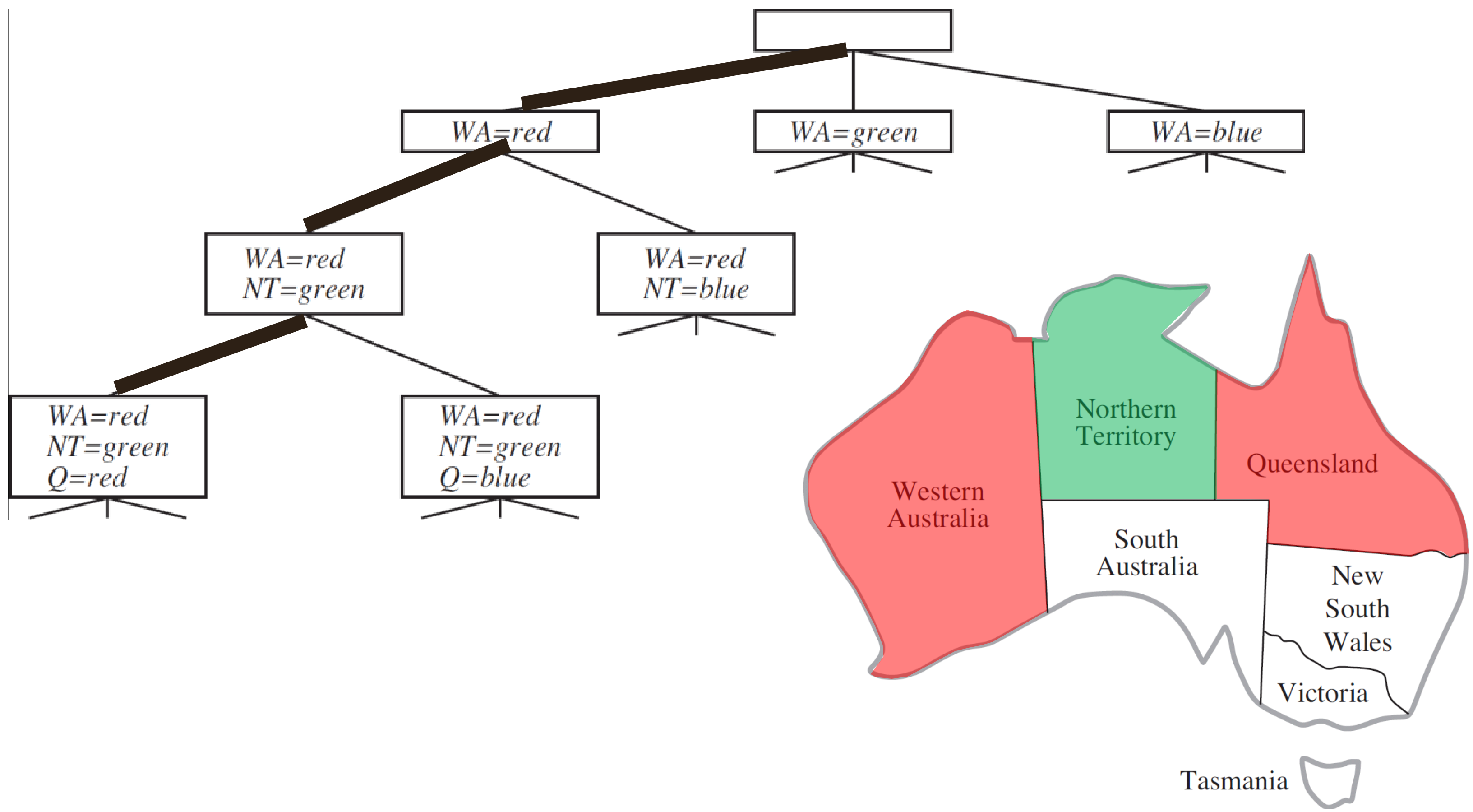
- Backtracking Search – depth-first assignment to one variable at a time, backtracking if no legal values remain.

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

function BACKTRACK(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment then
           add {var = value} to assignment
           inferences ← INFERENCE(csp, var, value)
           if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                 return result
       remove {var = value} and inferences from assignment
   return failure
```

6.3 Backtracking Search for CSPs
Example on Australia Coloring

- Backtracking Search – depth-first assignment to one variable at a time, backtracking if no legal values remain.



6.3 Problems for Backtracking Search

1. Which variable should be assigned next?
2. What order should its values be tried?
3. What are the implications of the current variable's assignment to unassigned variables?
4. When a path fails, can the search avoid repeating that failure in subsequent paths?

6.3 Backtracking Search: Variable Ordering

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- What variables should be assigned next?
  - Heuristics needed to select next variable to explore
- Minimum remaining values (MRV) heuristic – select variable with fewest remaining values
  - These variables are most likely to fail quickly & thus prune the search
  - For variables with empty domains, immediately backtrack
- Degree heuristic – select variable in most constraints with other variables
  - Selected variable tends to limit other variables more
  - Often used to break ties for MRV heuristic

6.3 Backtracking Search: Value Ordering

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

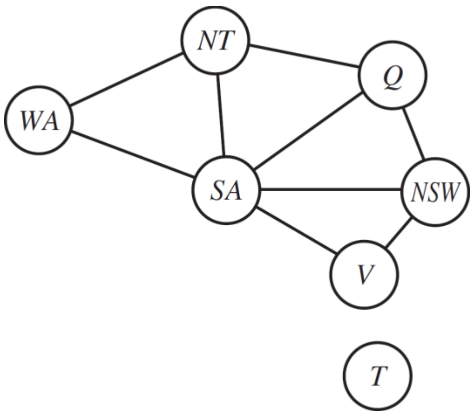
- What order should its values be tried?
  - Once a variable is selected, heuristics needed to select next value to explore
- Least-constraining value heuristic – select value that eliminates the fewest possibilities amongst neighbors in the constraint graph
  - Attempts to maintain flexibility in subsequent variables so that a solution can be found quickly if it exists

6.3 Backtracking Search: Interleaving Search & Inference

- Constraint information can be used to drastically reduce the search space (inference during search)
  - At every variable assignment, we can propagate constraints onto domains of other variables
- Forward Checking: when variable X is assigned, prune domains of unassigned variables
  - If Y is constrained by X, remove any values from Y's domain that are disallowed by X's assignment
  - Pruned domains make MRV (minimum remaining values) heuristic easy to implement and allow for quick detection of inconsistent assignments
  - Only applies arc-consistency (not useful if AC-3 is used)

Search & Inference Example

- Consider forward-checking in Australia map

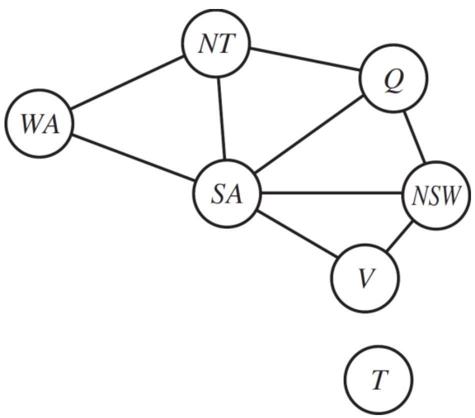


	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	R	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	R	B	G	R B	R G B	B	R G B
After *V=blue*	R	B	G	R	B		R G B

  - In 3rd row, after WA & Q are assigned, NT & SA are reduced to single value
  - In 4th row, after V is assigned, the domain of SA is empty ➔ inconsistency of assignment is detected

6.3 Backtracking Search: Interleaving Search & Inference

- Forward Checking does not detect inevitable inconsistencies; e.g., after 2nd color assignment
  - It makes current variable arc-consistent, but not others



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	R	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	R	B	G	R B	R G B	B	R G B
After *V=blue*	R	B	G	R	B		R G B

There is no consistent assignment now!

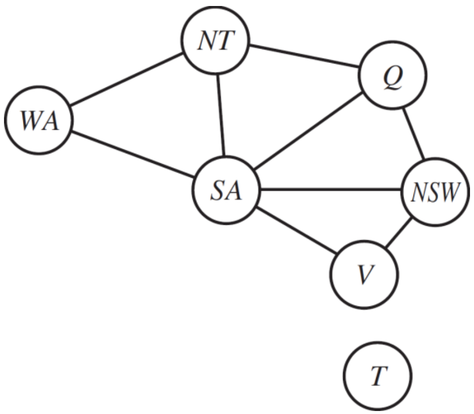
- Constraint Propagation: the process of using the implications of a constraint to restrict the search
  - We need to use fast constraint propagation during search

6.3 Backtracking Search: Applying Arc Consistency

- Recall: Arc consistency is constraint propagation technique for directed arcs between variables
  - Arc from X to Y is **consistent** if, for *every value* in X, there is a corresponding consistent value for Y
- Maintaining Arc Consistency (MAC) – uses AC-3 when X_i is assigned for every unassigned X_j with constraint (X_j, X_i)
  - AC-3 then propagates constraints in usual way
  - AC-3 will fail if an inconsistency is detected ➔ backtrack
- MAC is strictly more powerful than forward checking
  - Propagates constraints beyond current variable
  - More computation, but usually worth it

6.3 Backtracking Search: MAC Example

- Consider MAC in our Australia map example



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	R	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	R	B	G	R B	R G B	B	R G B
After *V=blue*	R	B	G	R	B		R G B

- *blue* deleted from NSW to make arc NSW➔SA consistent

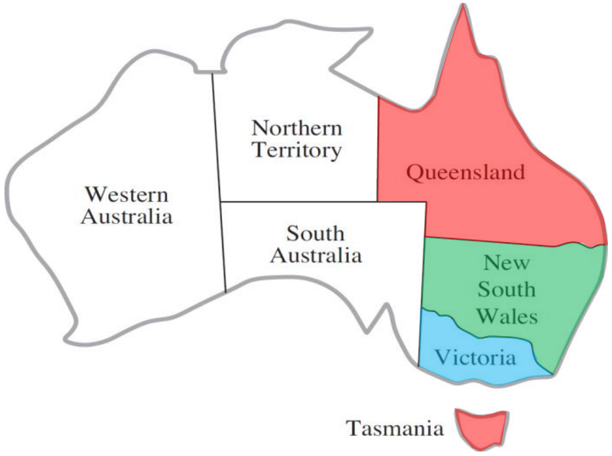
	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
After *Q=green*	R	B	G	R B	R G B	B	R G B

- Deleting *blue* from SA lets us detect inconsistency earlier

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
After *Q=green*	R	B	G	R B	R G B	B	R G B

6.3 Backtracking Search: Conflict Resolution Strategies

- Traditional backtracking: when branch fails, back up to last variable & try different value
  - Chronological Backjumping: on failure, reconsider last choice
- Strategy inefficient in many situations
  - eg. Consider variable ordering Q, NSW, V, T, SA and the current assigment {Q=red,NSW=green,V=blue,T=red}.



  - Assignment to SA not possible, but backtracking reassigns T, which will not resolve the conflict with SA

6.3 Backtracking Search: Intelligent Backtracking

- Idea: backtrack to variable that caused failure
  - Conflict set of X: assigned variables constrained with X
  - Backjumping: backtrack to first variable within conflict set
  - Modify backtracking-search to accumulate conflict set
  - Forward checking already implicitly computes conflict set:
    - when assigning X causes a value to be deleted from Y, add X to Y's conflict set
    - When last value deleted from Y, add Y's conflict set to X's
- Backjumping occurs if every value conflicts current assignment, but forward checking will detect this
  - Every branch pruned by backjumping is also pruned by forward checking (which can be used to make conflict set)
  - Backjumping useful, but we need deeper notion of conflict

6.3 Backtracking Search: Conflict-Directed Backjumping

- New conflict set - the set of preceding variables that caused assignment to X & all *subsequent* variables to fail
  - For variable with empty domain, use traditional conflict set
  - Otherwise, when variable X_j fails, backjump to most recent variable X_i in $conf(X_j)$ and make

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

- Using this notion leads to conflict-directed backjumping

6.3 Backtracking Search: Conflict-Directed Backjumping

- Example: Australia coloring
  - Consider sequence *WA=red*, *NSW=red* & *T=red* followed by *NT=blue*, *Q=green*, *V=blue*, & then *SA*

  - There is clearly no consistent assignment
    - Conflict set for *SA* = {*WA*, *NT*, *Q*}
    - Backjump to *Q*, which fails, *conf*(*Q*) = {*NT*, *NSW*}+{*WA*}
    - Backjump to *NT*, which fails, *conf*(*NT*)={*NSW*,*WA*}
    - Thus backjumping will skip over *T* & go to problem

6.3 Backtracking Search: Constraint Learning

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- We would also like to avoid re-making mistakes
  - Upon reaching a contradiction, some subset of conflict set is inherently responsible
- Conflict Learning – finding minimum set of assignments that caused the contradiction
  - Responsible variables & values are called no-good
  - No-goods avoided by adding constraints to CSPs or caching them
- Conflict learning can be used by forward-checking or backjumping & is one of most important techniques used by modern CSP solvers

6.4 Local Search for CSPs

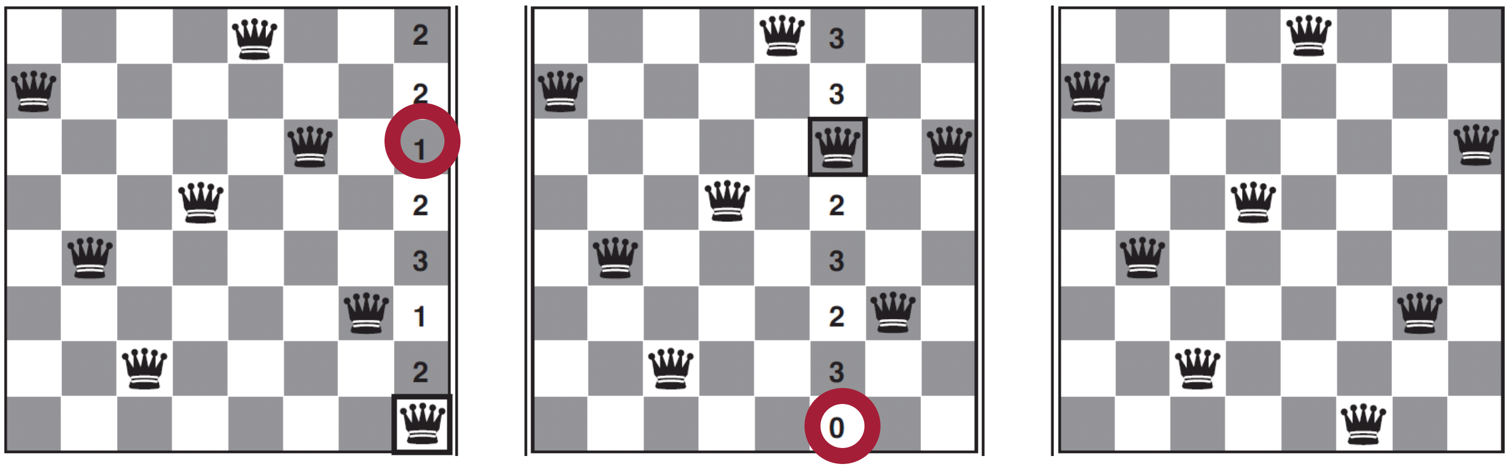
- Local search procedures also used for CSPs
  - Uses complete state formulation giving an initial value to every variable (usually not consistent)
  - In choosing new value, min-conflicts heuristic selects value with fewest conflicts with other variables

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*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)
    set *var* = *value* in *current*
  return *failure*

6.4 Local Search for CSPs
Example N-Queens

- Local search with min-conflicts heuristic can be used for N-Queens

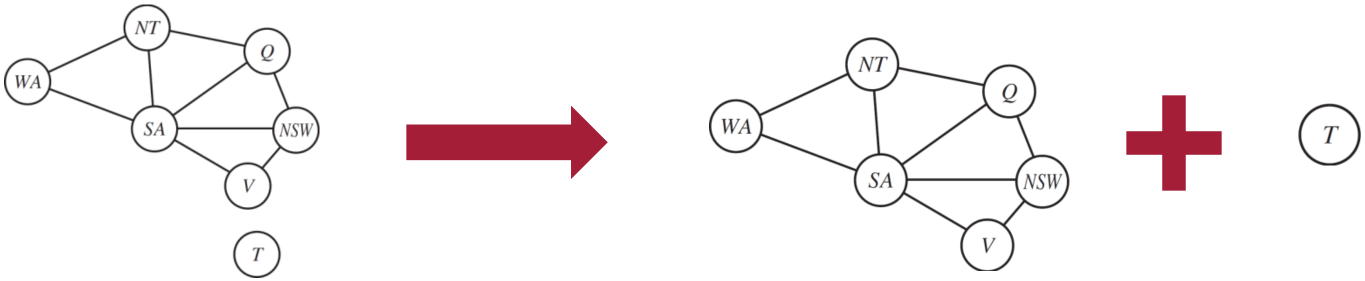


- Local Search for N-Queens is almost independent of size
  - Solves even 1 million queens in average of 50 moves
  - Solutions of N-Queens are densely distributed

- Plateau search allows moves to states of same score as current state
- Tabu search forbids revisiting recently-visited states by keeping small queue
- Simulated annealing also used
- Constraint weighting – concentrates search on difficult constraints by iteratively weighting them
  - All constraints given an initial weight $W_i = 1$
  - Variable/value selected to minimize total weight after change
  - Weights of remaining constraints are increased by *1*

6.5 The Structure of CSPs

- How can we use constraint structure?
  - Basic idea: decompose problem into subproblems
  - Ex. In Australia, Tasmania can be colored separately



- Independent subproblems – parts of the CSP that can be solved separately
  - Find connected components of constraint graph to break it into subgraphs, CSP_i
  - If assignment S_i solves CSP_i then $\bigcup_i S_i$ solves CSP
  - If each CSP_i has c variables of total n, each can be solved in $O(d^c)$ ➔ total time is $O(d^c\, n/c) \ll O(d^n)$

6.5 The Structure of CSPs
Tree-structured CSPs

- Constraint graph is a tree if every pair of nodes is connected by at most one path.
  - CSP is directed arc consistent (DAC) for ordering X_1, X_2, … , X_n if & only if X_i is arc-consistent with X_j for all $j > i$
- For a tree-structured CSP, solution in linear time
  - Choose any node as root & construct a topological sort (every child in tree appears after its parent)

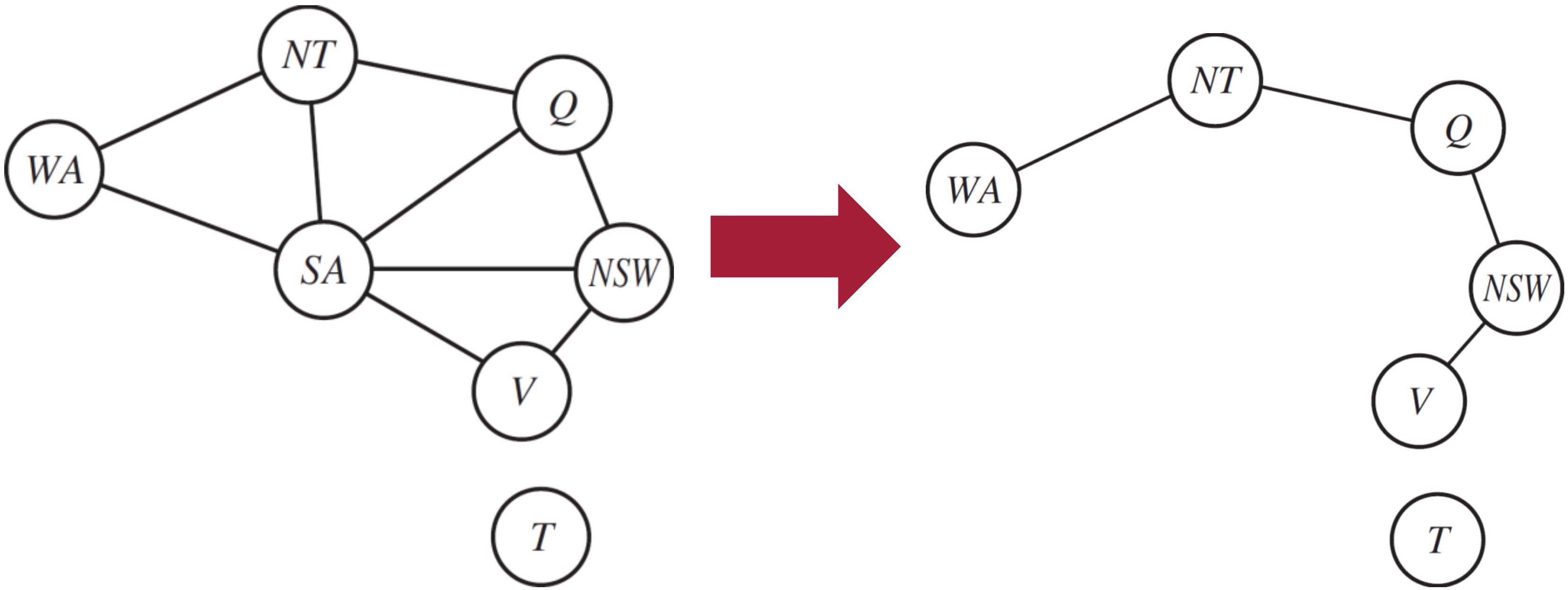
  - Every tree of size n has n-1 arcs ➔ DAC in $O(n)$ steps
  - Each step searches over 2 domains of size d ➔ $O(nd^2)$
  - Then assign from parents to children without backtracking

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  $assignment \leftarrow$ an empty assignment
  $root \leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, $root$)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    $assignment[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

6.5 The Structure of CSPs
General CSPs with tree algo

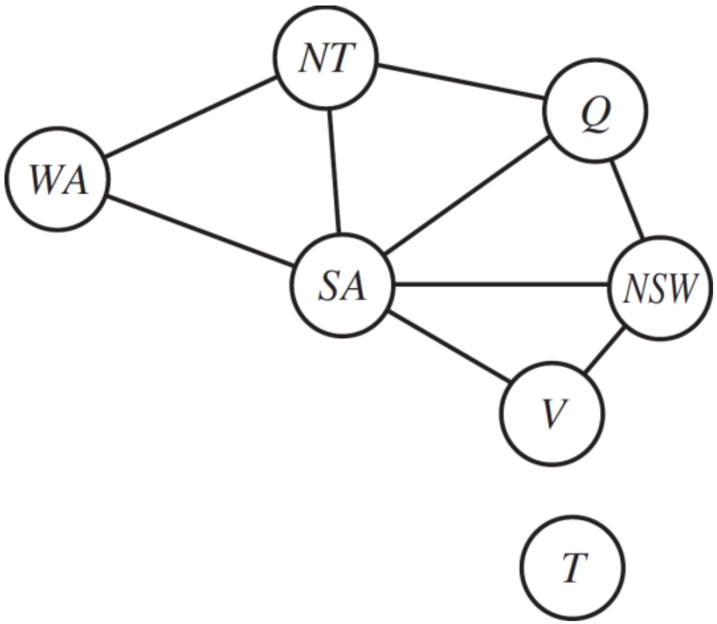
- Approaches to solving non-Tree CSPs:
  - Remove variables to make CSP become a tree
  - Join variables into combined nodes with tree structure
- Ex. Australia becomes tree once SA assigned

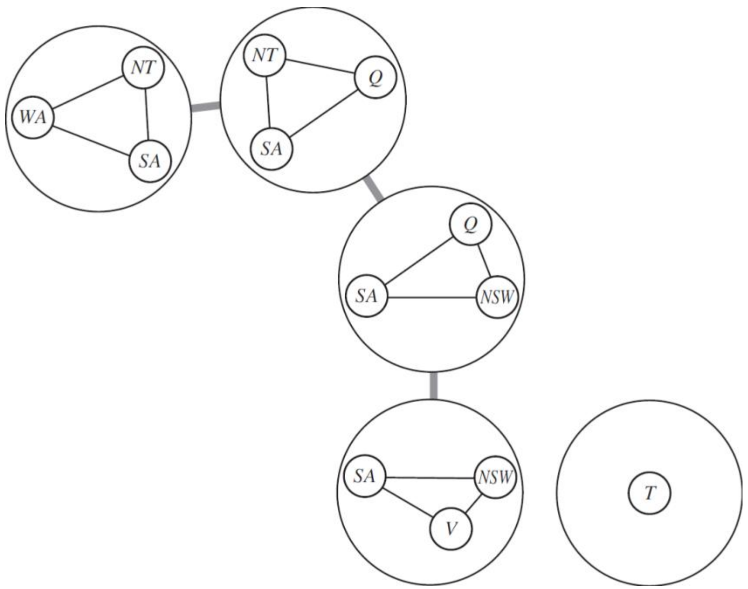


- Removing variables (Cutset Conditioning):
  - Choose subset of variables S, whose removal makes CSP into a tree – S is a cycle cutset; T is remainder
  - For every consistent assignment s to S
    - Remove all values from domains of variables in T that are inconsistent with s
    - Use tree-solver to solve T
    - If there is an assignment to T, it is a solution to the CSP
- If cutset is size c, algorithm in $O(d^c * (n-c)d^2)$
  - If CSP is tree-like, c will be small
  - In worst-case, though, c can be $(n-2)$
  - Finding smallest cycle cutset is NP-hard; but efficient approximations are possible

Tree Decomposition

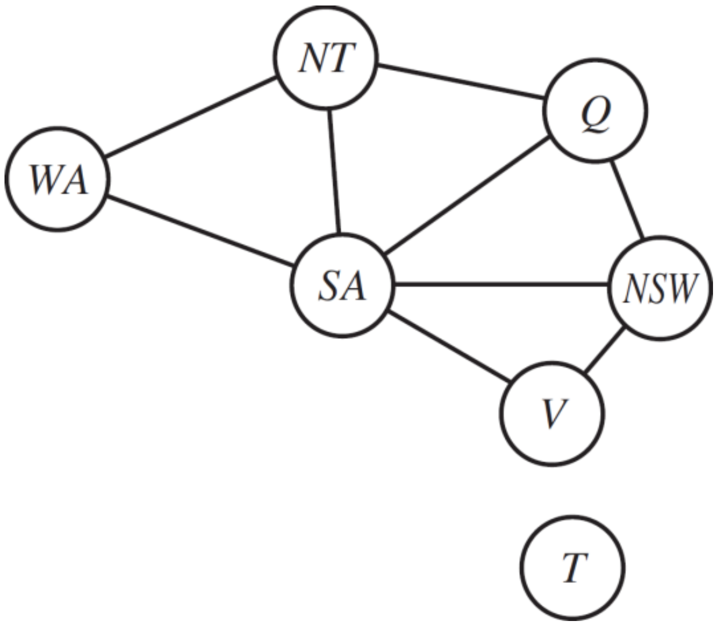
- Constraint graph can also be decomposed into tree where nodes are subproblems
  - Each subproblem is connected component in graph
  - If each subproblem is small, it works well

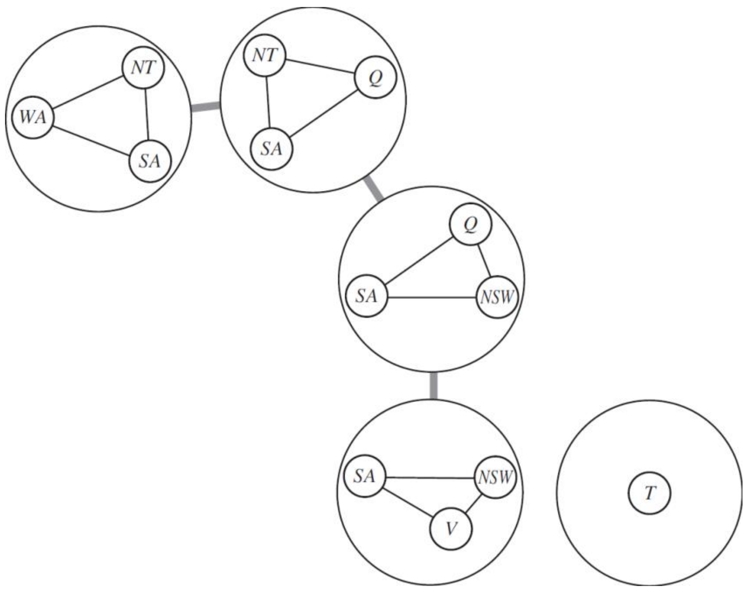




6.5 The Structure of CSPs
Tree Decomposition

- Tree decomposition must satisfy:
  1. Every variable appears in at least one subproblem
  2. If two variables are involved in a constraint, they must appear together in at least one subproblem
  3. If variable appears in 2 subproblems, it must appear in every subproblem on the path between them





6.5 The Structure of CSPs
Tree Decomposition

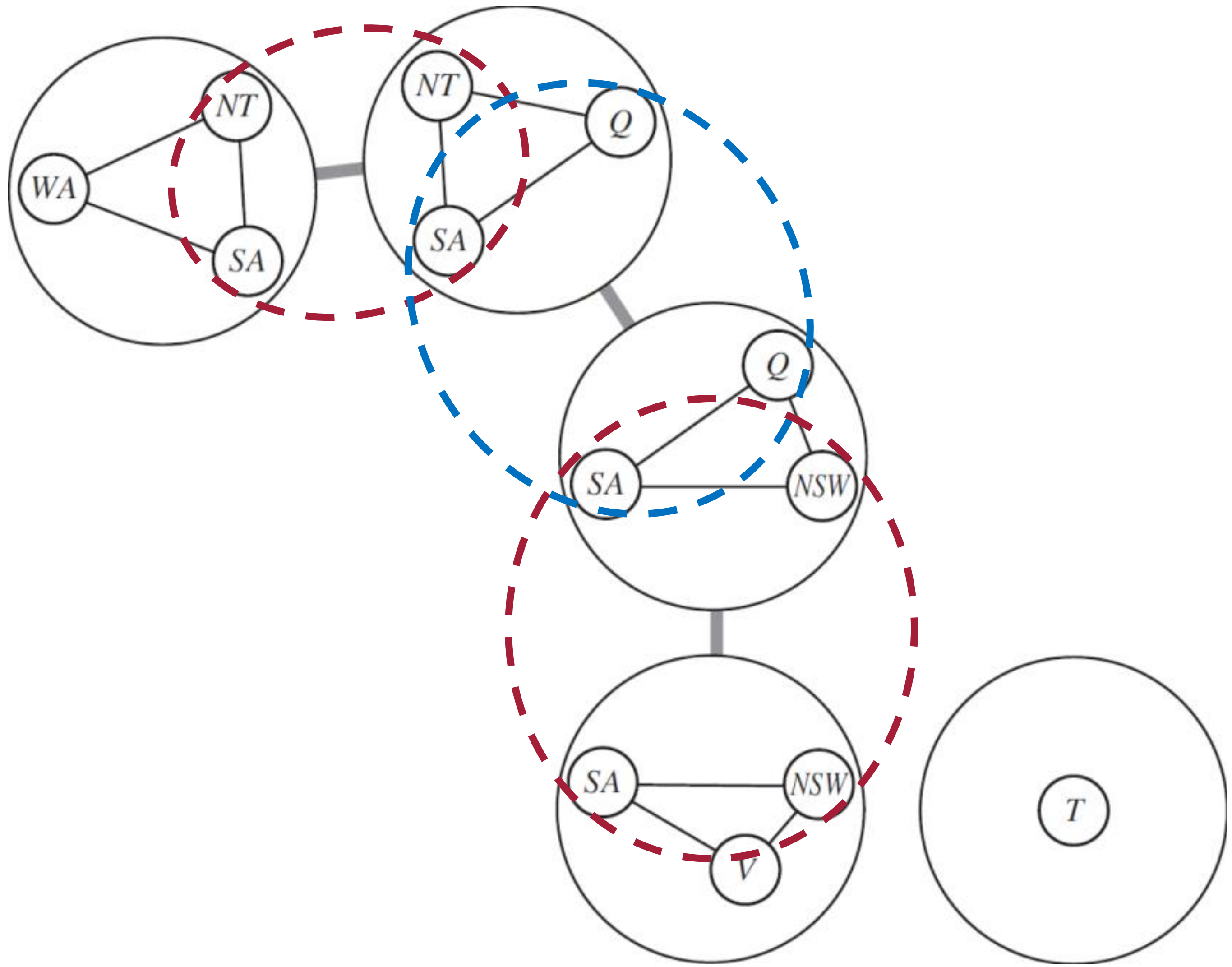
EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Tree decomposition properties:
  1. Every variable appears in at least one subproblem
     - Ensures that every variable is represented
  2. If two variables are involved in a constraint, they must appear together in at least one subproblem
     - Ensures that every constraint is represented
  3. If variable appears in 2 subproblems, it must appear in every subproblem on the path between them
     - Ensures that duplicated variables have same value
     - This is enforced by the links between subproblems

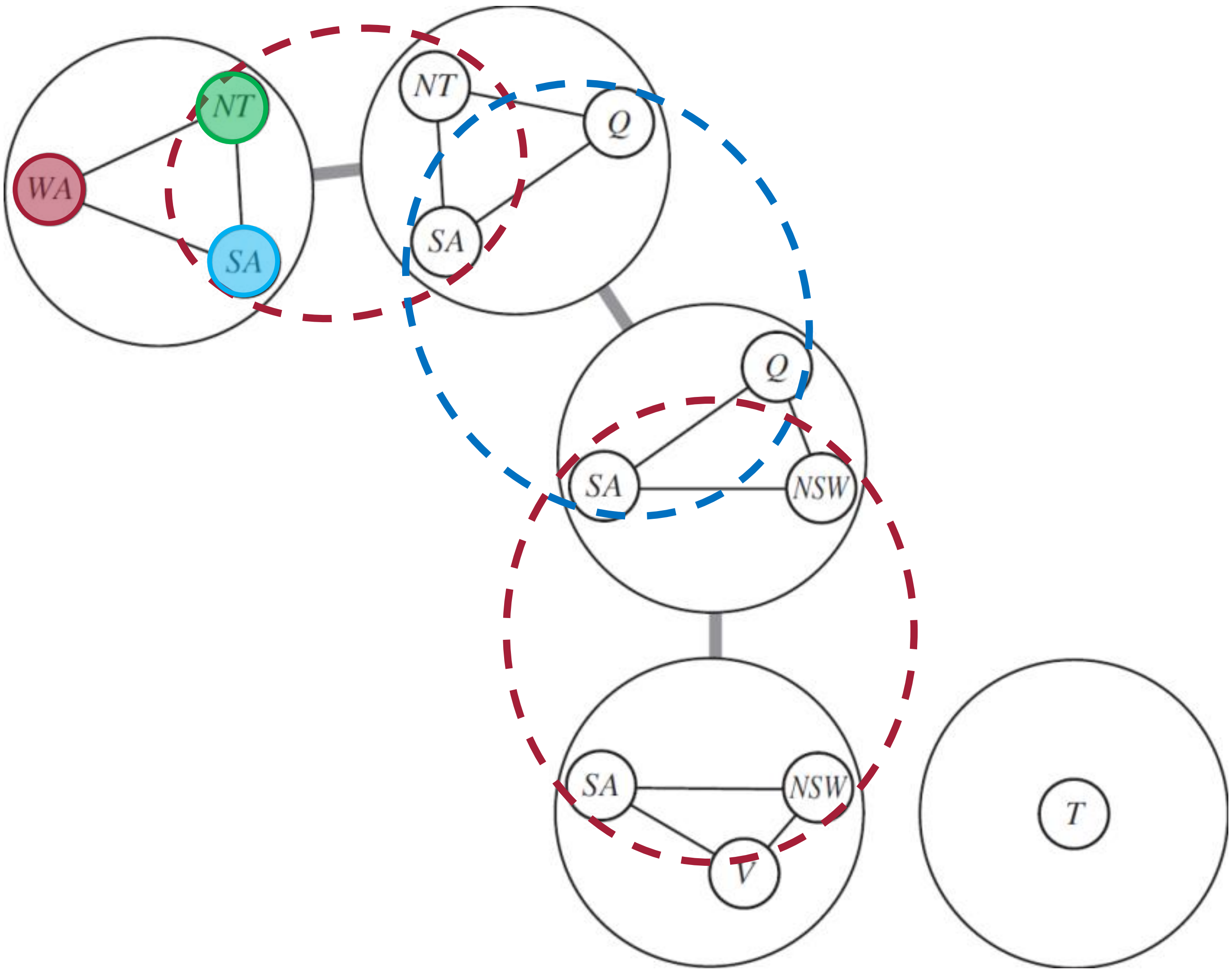
6.5 The Structure of CSPs
Solving with Tree Decomposition

- Algorithm for solving a tree-decomposed CSP
  1. Solve each subproblem separately to define its domain
     - Ex. The *WA,SA,NT* subproblem has 6 solutions: (R,G,B), (R,B,G), (B,R,G), (B,G,R), (G,R,B), & (G,B,R)
  2. If any subproblem has no solution, there is no solution
  3. Otherwise, treat each subproblem as a ‘mega-variable’ whose values are the solutions to the subproblem
  4. The constraint between a pair of subproblems is that their shared variables *must agree*
     - Ex. The *WA,SA,NT* subproblem and its neighbor *NT,SA,Q* overlap with *NT* & *SA* ➔ if (*WA=red,SA=blue,NT=green*) the only legal assignment to *NT,SA,Q* makes *Q=red*
  5. Solve the tree by using the efficient tree-solver algorithm to find consistent assignments to subproblems

Example: Tree Decomposition

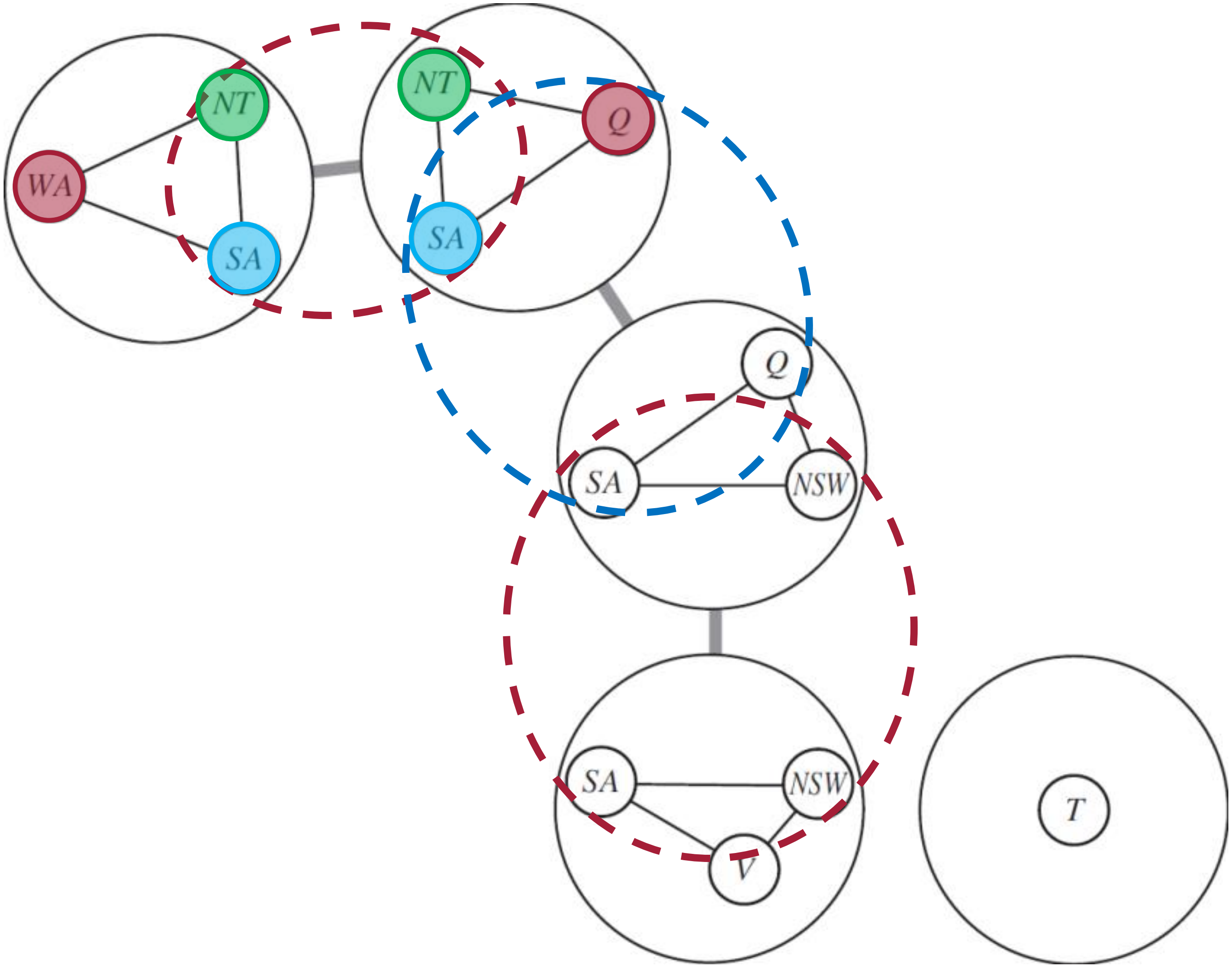


6.5 The Structure of CSPs
Example: Tree Decomposition

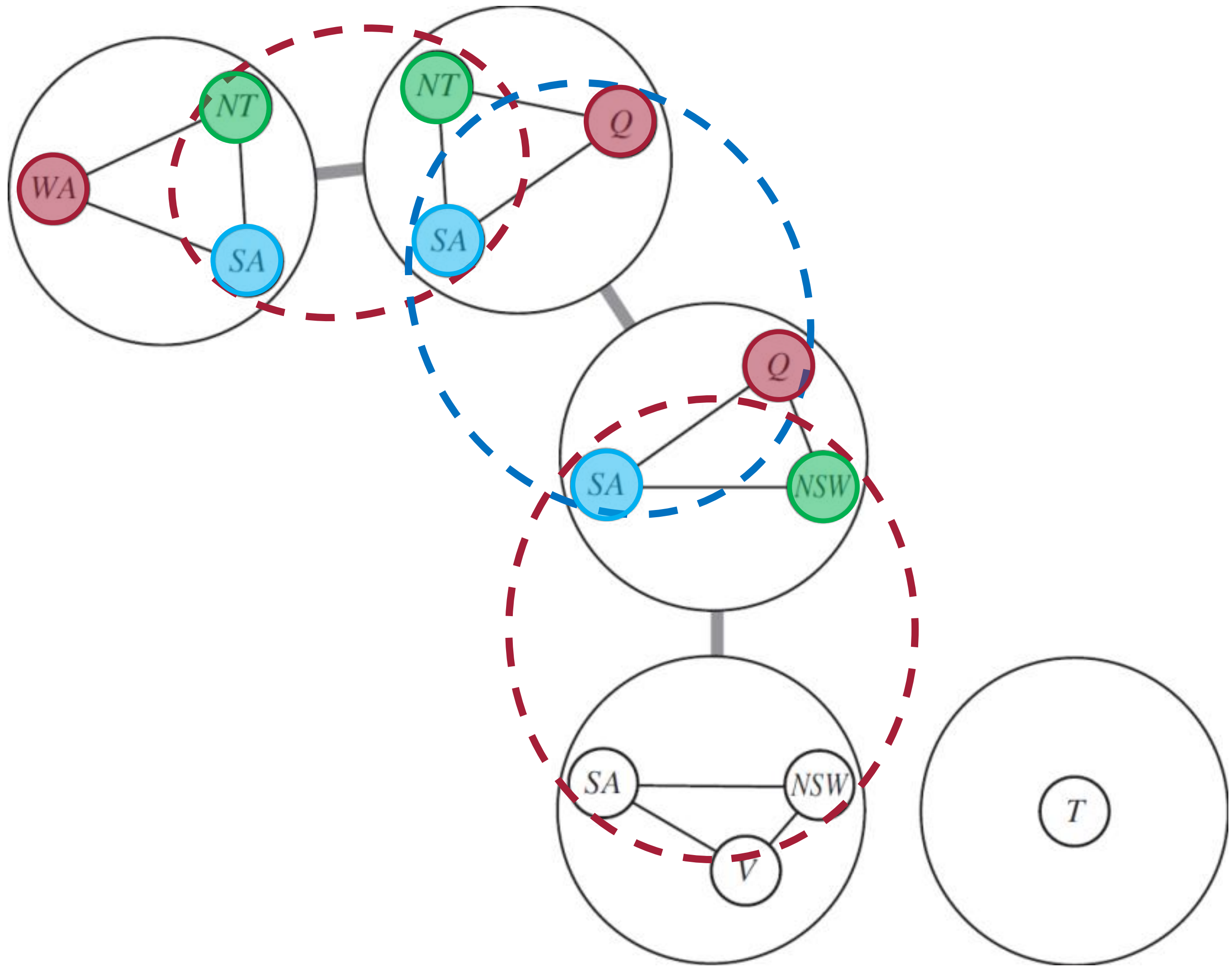


6.5 The Structure of CSPs
Example: Tree Decomposition

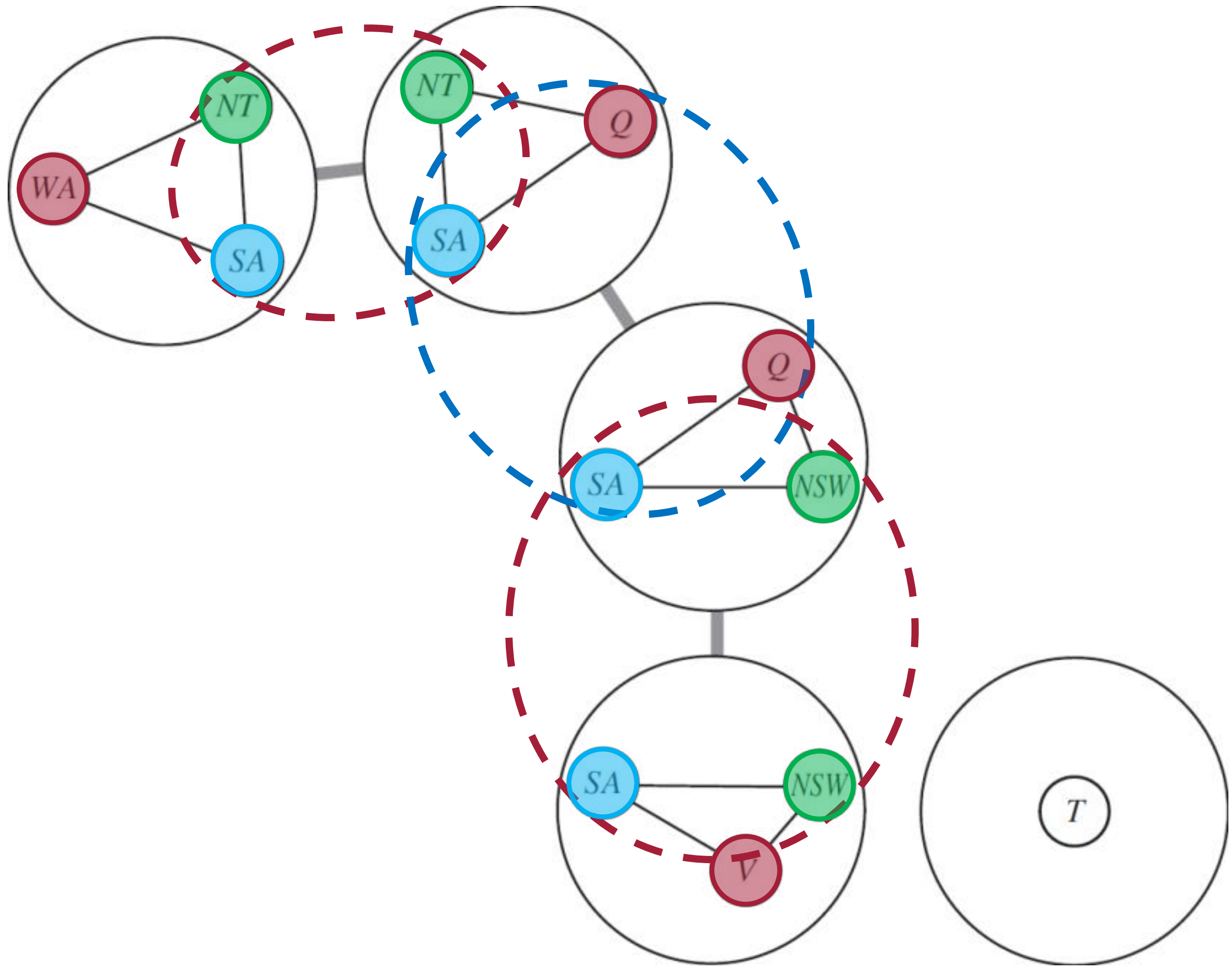
EBERHARD KARLS UNIVERSITÄT TÜBINGEN



6.5 The Structure of CSPs
Example: Tree Decomposition



6.5 The Structure of CSPs
Example: Tree Decomposition



6.5 The Structure of CSPs
Decomposing a CSP into a Tree

- There are many tree decompositions of a constraint graph; we want one with the smallest subproblems
  - Tree width of *tree* is size of its largest subproblem minus 1
  - Tree width of a *graph* is the minimum tree width of all decompositions
  - A tree with tree width w can be solved in $O(nd^{w+1})$
  - Hence, CSPs with bounded tree width are polynomial
- Finding a decomposition with minimal tree width is NP-hard; again, there are efficient heuristic methods

6.5 The Structure of CSPs
Structure in Values

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Value Symmetry – assignments that are structurally equivalent; ex. Colors can be permuted
  - For map coloring (k colors), there are $k!$ permutations
  - We would like to reduce search space so only non-equivalent solutions exist
- Symmetry-breaking constraints – constraints added to the CSP to prevent equivalent solutions
  - Ex. (Australia) constraint $NT < SA < WA$ (alphabetic) ensures only one of the $k!$ solutions can be found
  - Polynomial algorithms to eliminate all but one symmetric solution, but NP-hard to eliminate all symmetry within intermediate sets of values during search