Constraint Satisfaction Problems

Dr. Demetrios Glinos
University of Central Florida

CAP4630 –Artificial Intelligence

Today

- Constraint Satisfaction Problems
- Backtracking Search
- Improving Backtracking
  - Filtering
    - Forward Checking
    - Arc Consistency
    - K-Consistency
  - Ordering
  - Structure
- Iterative Methods

What Problems CSPs Solve

- CSPs are a particular type of search problem
- Identification problem: assignments to variables
  - a state consists of a set of variables
  - variables can take on particular values
  - there are *constraints* on the assignments
  - goal is to find an acceptable assignment of values
- Compare to Planning problem: finding a *path* to a goal state
- Agent perspective:
  - single agent, deterministic actions, complete information, discrete state space

CSP Applications

- A non-exclusive list:
  - Class scheduling: where and when
  - Teacher assignment: who teaches which class
  - Factory scheduling: multiple jobs, multiple stations
  - Transportation scheduling: railroads, shipping, trucking
  - Hardware configuration: including circuit layout
  - Fault diagnosis: for example, for spacecraft systems

CSP Defined

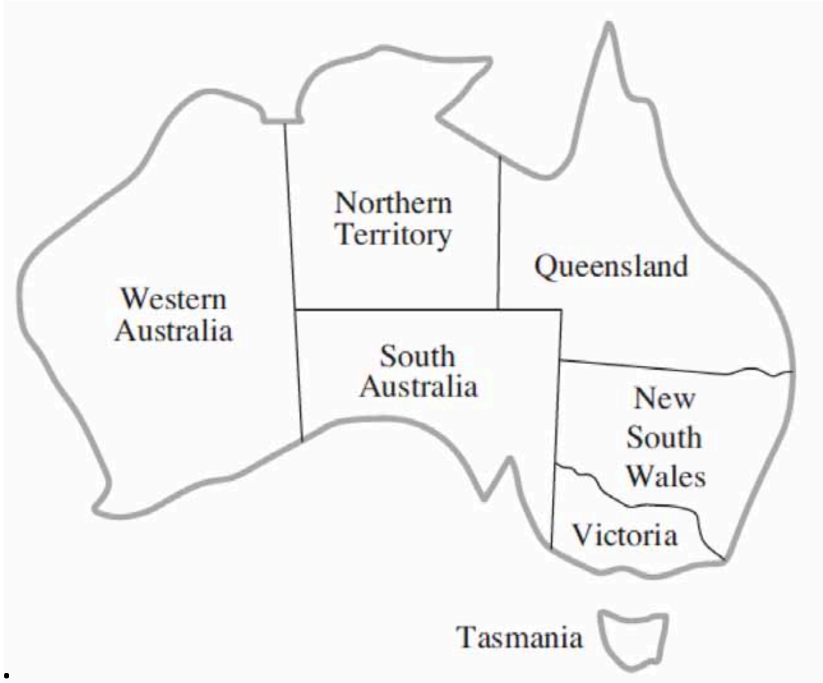
- A Constraint Satisfaction Problem (CSP) is defined by $< X, D, C >$ where
  - X is a set of *variables* $\{ X_1, ..., X_n \}$
  - D is a set of *domains* $\{ D_1, ..., D_n \}$, one for each variable
    - where D_i is a set of allowable *values* $\{ v_1, ..., v_k \}$ for variable X_i
  - C is a set of *contraints* that specify allowable combinations of values
    - where C_i is a tuple $<$ *scope, rel* $>$ and
      - *scope* is some subset of variables
      - *rel* is a relation that defines the contraint

CSP Search Problem

- State space
  - defined by assignment of values to one or more variables
  - includes complete and also partial assignments
  - each must be consistent with constraints
- Successor function
  - we assign values to variables *sequentially*
- Initial state
  - no variables assigned
- Goal test
  - Are all variables assigned?
  - Are all constraints satisfied?

Example: Map Coloring

The problem: Color the map using only 3 colors such that no two adjacent states have same color



X = { WA, NT, Q, NSW, V, SA, T }

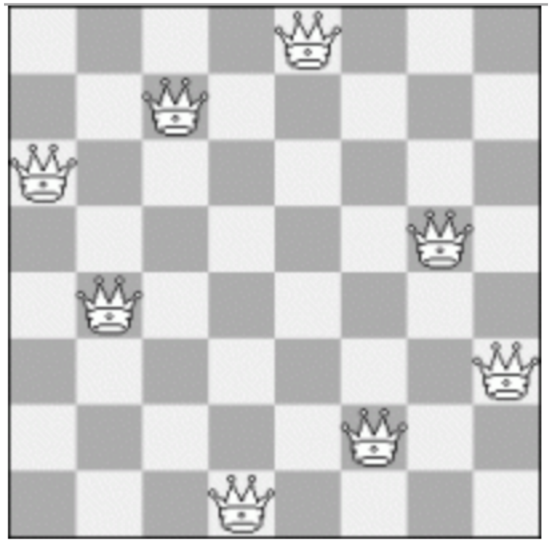
D = { red, green, blue }

We can express constraints 2 ways:

- *Implicitly* using rules: SA≠WA, SA≠NT, WA≠NT, etc.
- *Explicitly* listing allowable combinations:
  e.g., (WA, SA) ∈ { (red, green), (red, blue), ... }

Example: N-Queens

- Problem: Place N queens on a chessboard in such a way that they do not threaten each other
- Variables: Board positions X_{ij}
- Domains: { 0, 1 }
- An assignment: $X_{ij} = 1$ if a queen is on square (i,j)
- Constraints:



No two queens in same row: $\forall i, j, k : (X_{ij}, X_{ik}) \in \{(0,0),(1,0),(0,1)\}$

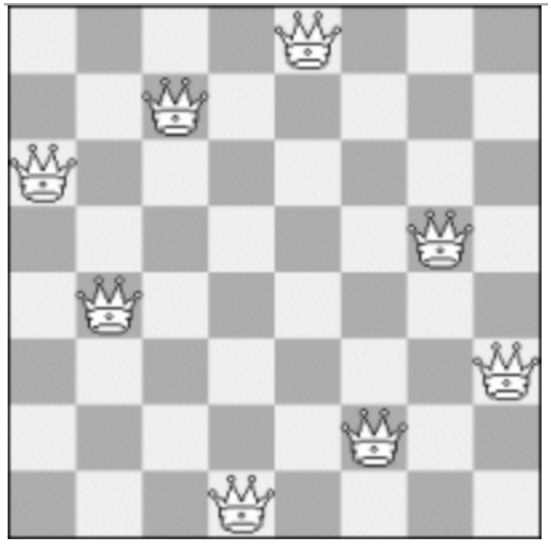
No two queens in same column: $\forall i, j, k : (X_{ij}, X_{kj}) \in \{(0,0),(1,0),(0,1)\}$

No two queens in same diagonal: $\forall i, j, k : (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(1,0),(0,1)\}$

$\forall i, j, k : (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(1,0),(0,1)\}$

N queens: $\sum_{i,j} X_{ij} = N$

Example: N-Queens *(alternate)*



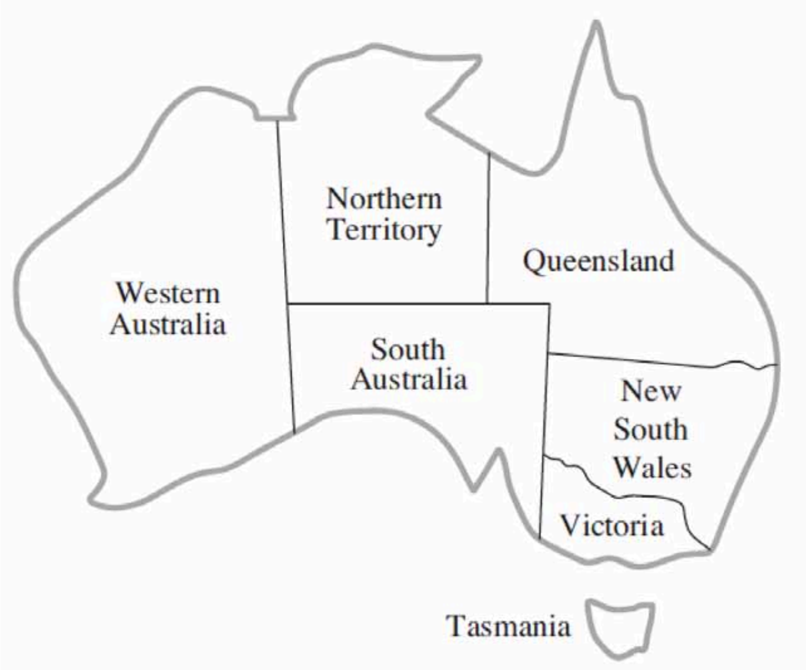
- Problem: Place N queens on a cheesboard in such a way that they do not threaten each other
- Variables: Q_k, i.e., one queen in each ***row***
- Domains: { 1, 2, ..., N }
- An assignment: Q_i = j, if queen i is in column j
- Constraints:

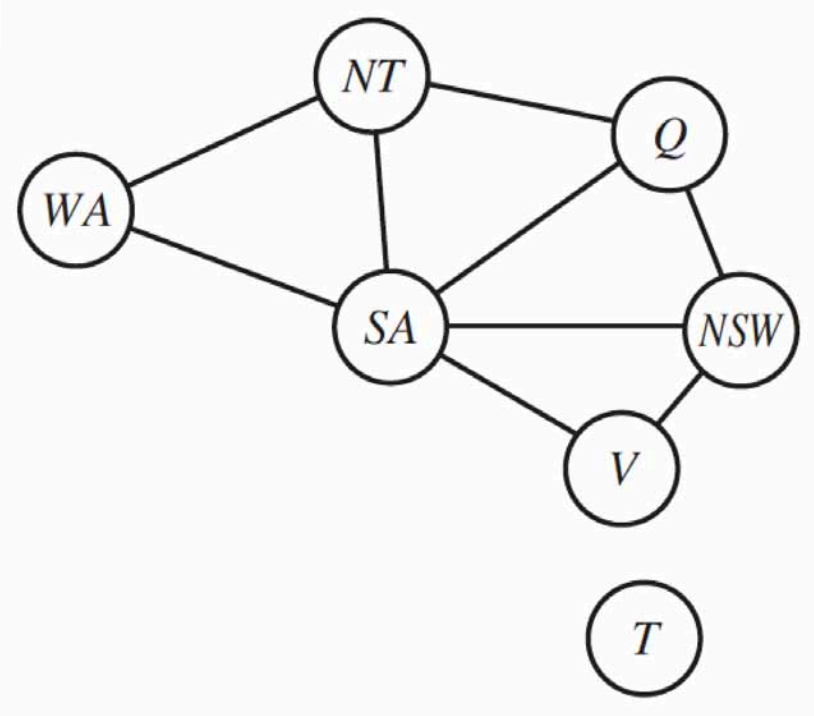
No two queens in same column: $\forall i, j : Q_i \neq Q_j$

No two queens in same diagonal: $\forall i, j : \{\neg \exists k \mid (j = i + k) \wedge (Q_j = Q_i + k)\}$

$$\forall i, j : \{\neg \exists k \mid (j = i + k) \wedge (Q_j = Q_i - k)\}$$

Constraint Graphs

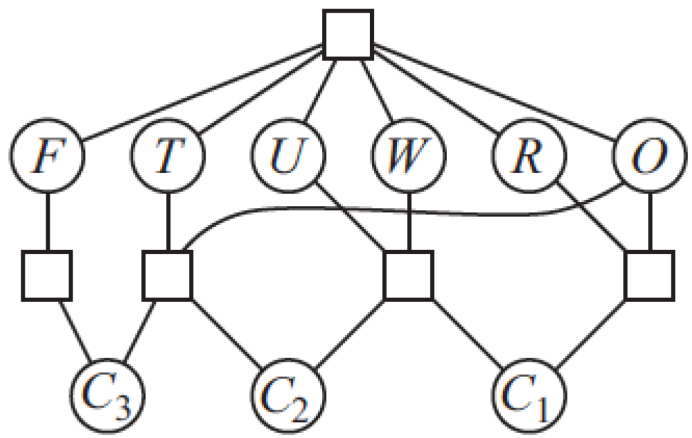




Nodes are the variables
Arcs show the existence of constraints

Example: Cryptarithmetic

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



Alldiff constraint

Column addition constraints

Carry digits

- The problem: assign unique digits to the letters such that the arithmetic relation holds.
- Variables: *{ T, W, O, F, U, R, C_1, C_2, C_3 }*, including carries
- Domains for letters: *{ 0, 1, …, 9 }*, Domains for carries: *{ 0, 1 }*
- Constraints of form: *$O + O = R + 10*C_1$, $C_1 + W + W = U + 10*C_2$, etc.*

  Plus the "Alldiff" constraint (no 2 letters have same digit)

Example: Sudoku

- Problem: Fill in the remaining cells with digits satisfying constraints
- Variables: Each open cell
- Domains: { 1,2, ..., 9 }
- Constraints:
    - 9-way *alldiff* for each row
    - 9-way *alldiff* for each column
    - 9-way *alldiff* for each region

	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		

Example: k-SAT

- The problem: Finding an assignment of truth values to variables that makes a set of disjunctive clauses all true
- Many formulas in propositional logic can be reduced to such form
- Example: $(p \lor q \lor r) \land (-q \lor s \lor t) \land \ldots$ ← *3-SAT example*
- Complexity:
  - for n variables, there are 2^n possible assignments
  - for $k \geq 3$, these problems are NP-complete
  - Practical uses: e.g., fault diagnosis

Types of CSPs

- Discrete Variables
  - Finite domains: $O(d^n)$ complete assignments
    - e.g., Boolean satisfiability
  - Infinite domains (integers, strings)
    - e.g., job scheduling – start/end times for various jobs
    - linear constraints are solvable

- Continuous Variables
  - e.g., scheduling observation start/end times for Hubble telescope
  - linear constraints solvable in polynomial time using Linear Programming methods

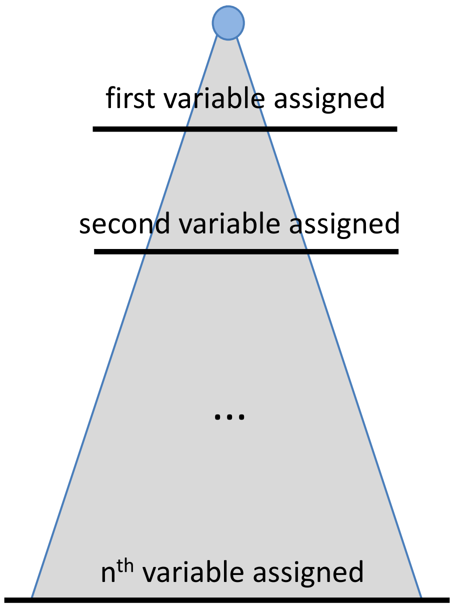
Types of Constraints

- Absolute constraints
  - Unary: e.g., WA ≠ red (usually implemented by just reducing domain)
  - Binary: e.g., WA ≠ NT
  - Higher-order: e.g., cryptarithmetic constraints

- Preference constraints
  - can be violated
  - can be encoded as costs on variable assignments
  - e.g., in map coloring: red is better than blue
  - e.g., in course scheduling program, Prof. A prefers to teach mornings

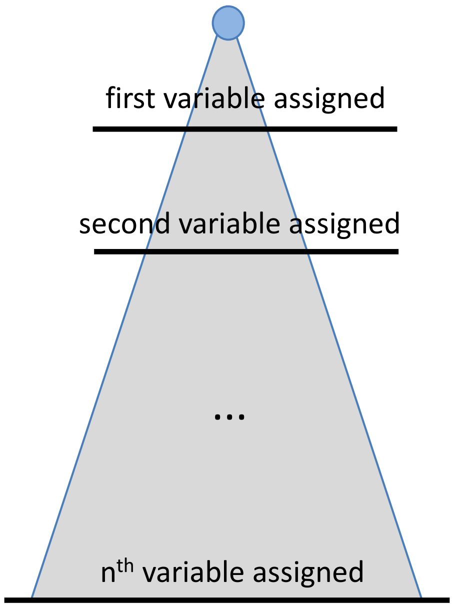
Solving CSPs

- Baseline approach: Standard search
  - Branching factor b = (n)(d) at top level
  - At second level, b = (n-1)(d)
  - For n levels, tree has $(n!)(d^n)$ leaves
  - Yet there are only d^n total complete assignments !!!
- Ignoring these issues for the moment
  - Q: What would BFS do?
  - Q: What would DFS do?
  - Q: Which is preferable for this type of problem?



Solving CSPs

- Baseline approach: Standard search
  - Branching factor b = (n)(d) choices at top level
  - At second level, b = (n-1)(d)
  - For n levels, tree has $(n!)(d^n)$ leaves
  - Yet there are only d^n total complete assignments !!!
- Ignoring these issues for the moment
  - Q: What would BFS do?
  - Q: What would DFS do?
  - Q: Which is preferable for this type of problem?



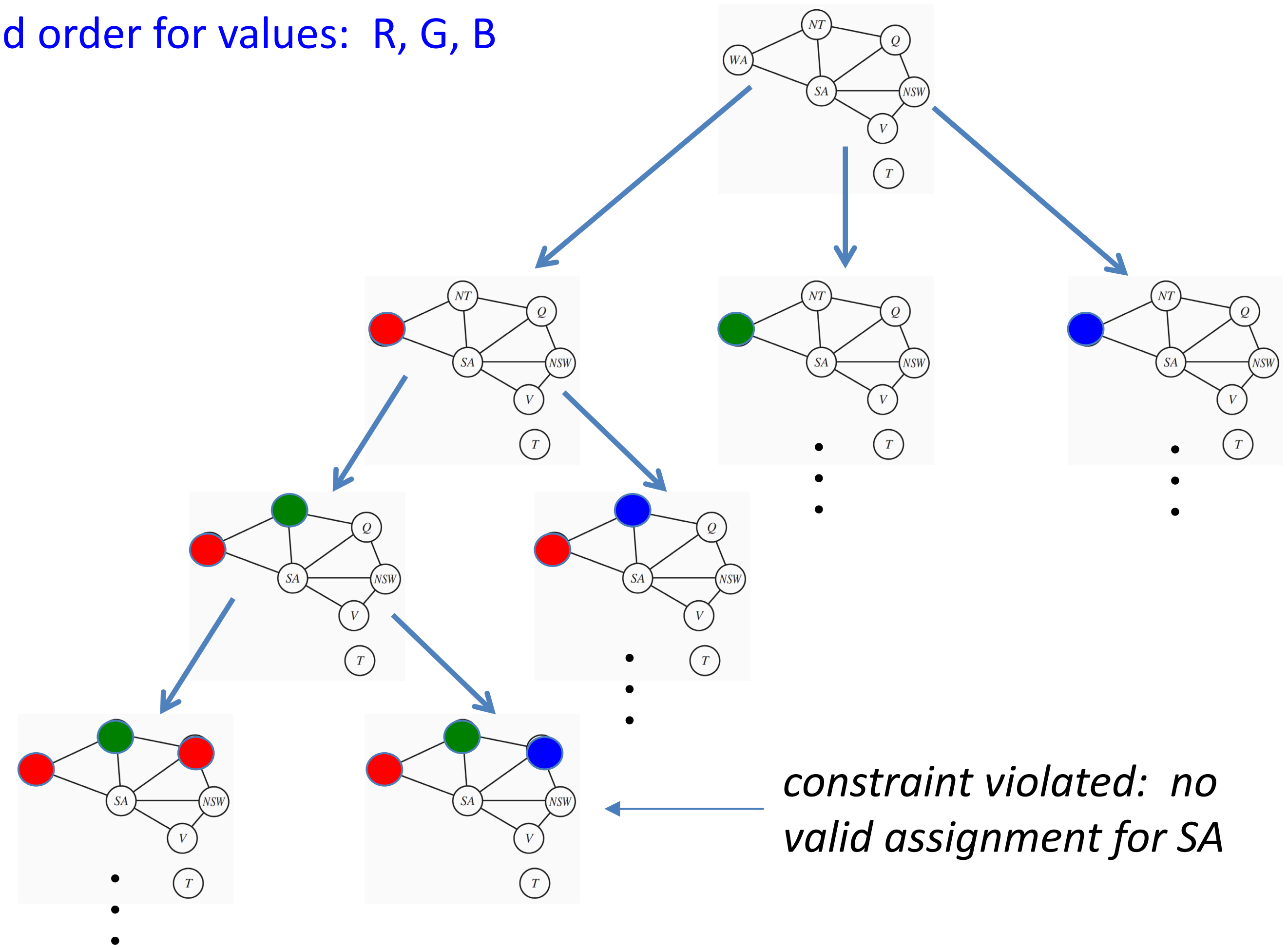
A: All of the goal states are at the bottom of the pyramid, so DFS is preferred

Backtracking Search

- This is the basic ***uninformed*** method for solving CSPs
- Backtracking search is DFS with these 2 improvements:
  - One variable at a time:
    - Use a *fixed ordering*, since assignments are commutative
      - This reduces tree back to O(d^n)
    - The order doesn't matter (yet)
  - Check constraints as you go:
    - Don't conflict with prior assignments
      - There is a computational cost for checking
    - We can think of this as an "incremental" goal test

Backtracking Example

Use fixed order for variables: WA, NT, Q, ...
Use fixed order for values: R, G, B



constraint violated: no valid assignment for SA

Backtracking Search Algorithm

```
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, assignment )
    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, assignment )
            if( inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK( assignment, csp )
                if result ≠ failure then
                    return result
        remove { var = value } from assignment
    return failure
```

Backtracking Search: Optimization Opportunities

```
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, assignment )
    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, assignment )
            if( inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK( assignment, csp )
                if result ≠ failure then
                    return result
        remove { var = value } from assignment
    return failure
```

Backtracking Search: Optimization Opportunities

```
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, assignment )
    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, assignment )
            if( inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK( assignment, csp )
                if result ≠ failure then
                    return result
        remove { var = value } from assignment
    return failure
```

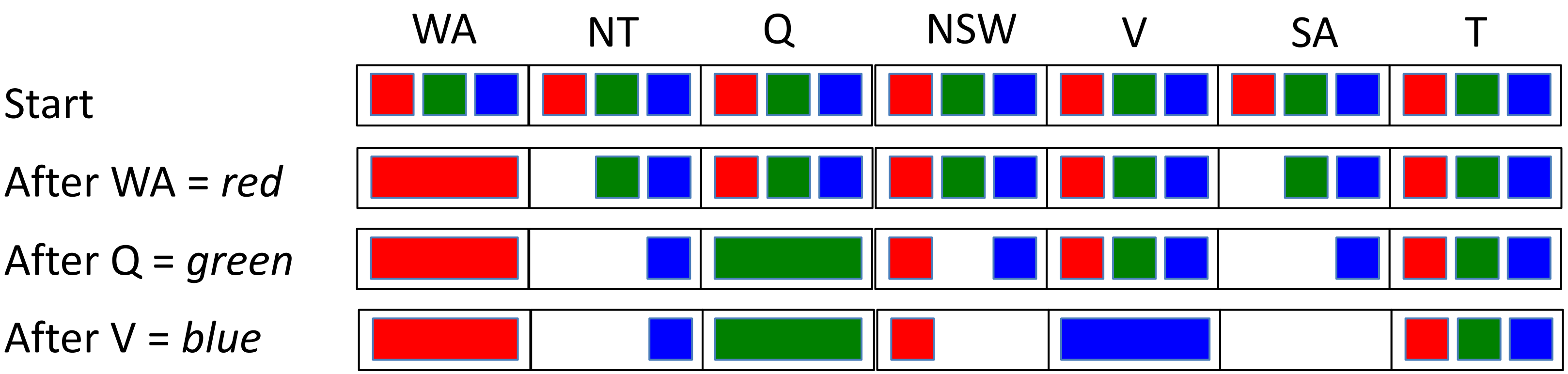
Backtracking Search: Optimization Opportunities

```
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, assignment )
    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, assignment )
            if( inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK( assignment, csp )
                if result ≠ failure then
                    return result
        remove { var = value } from assignment
    return failure
```

Improving Backtracking

- General-purpose (not problem-specific) ideas
  - Can give large increases in performance
  - Filtering:
    - Detecting dead ends early
  - Ordering:
    - Which variables
    - Which values
  - Structure:
    - Simplifying problem

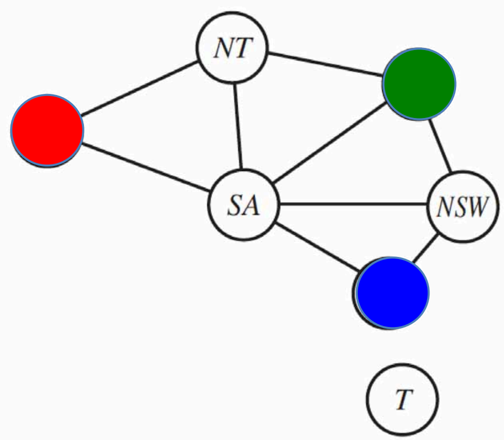
Filtering: Forward Checking



Forward checking uses variable assignments to constrain the domains of unassigned variables by crossing off bad options in unassigned variables

→ If any domain becomes empty, backtrack now

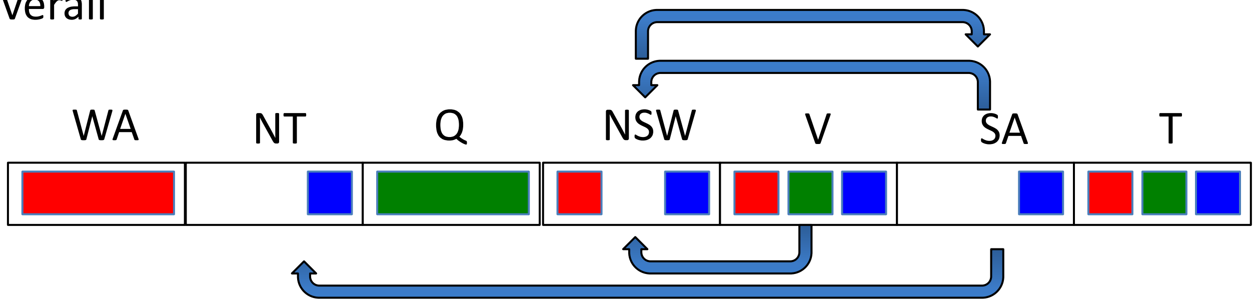
This performs one-step ***constraint propagation***



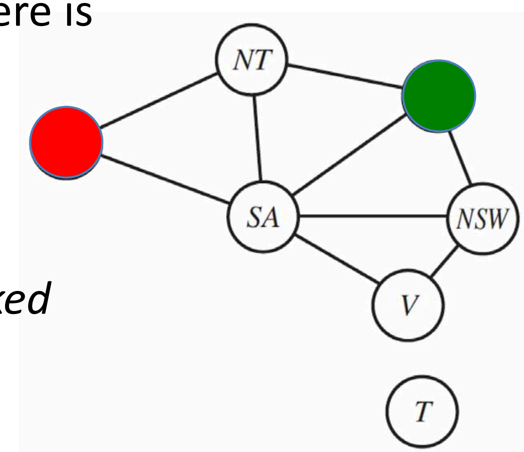
Filtering: Arc Consistency

- Basic idea: Enforcing consistency on each local part of a graph will eliminate inconstencies overall

After Q = Green, (V,NSW) is arc consistent but (SA,NT) is not



- Arc from X_i to X_j, denoted (X_i, X_j), is ***arc-consistent*** if there is at least one possible assignment in X_j for *every* value in domain of X_i
  - if not, then we drop values *from* X_i
- *If X_i loses a value, all neighbors of X_i need to be rechecked*
- Do this for ***all*** arcs before next assignment
- Note: Each binary constraint is ***two*** arcs



Filtering: Arc Consistency Algorithm

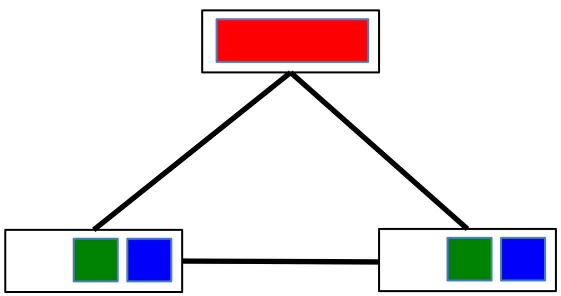
```
function AC-3( csp ) returns false if an inconsistency is found and true otherwise
    inputs: csp, a binary CSP with components ( X, D, C )
    local variables: queue, a queue of arcs, initially all the arcs in csp
    while queue is not empty do
        ( X_i, X_j ) ← REMOVE-FIRST( queue )
        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 ← 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 ← true
    return revised
```

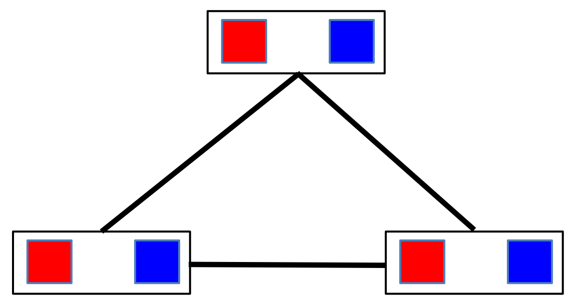
i.e., check all arcs, and if revise any, then recheck all neighbors

i.e., delete a choice if there is no consistent assignment

Filtering: Arc Consistency Limitations

- Arc consistency does not avoid the need for backtracking
- Result after enforcing arc-consistency
  - One solution remains
  - Multiple solutions remain
  - Or no solutions at all



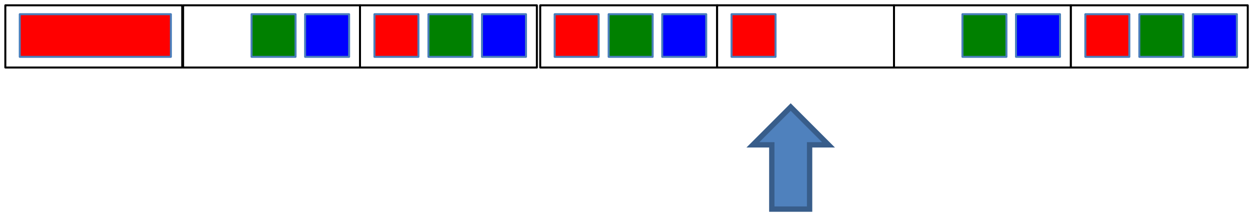


Filtering: Generalized Arc Consistency

- 1-Consistency ("Node Consistency"):
  - Each node's domain contains a value that satisfies the node's unary constraints
- 2-Consistency ("Arc Consistency"):
  - For any pair of nodes, any consistent assignment to one can be extended to the other
- K-Consistency:
  - For every K-subset of nodes, any consistent assignment to K-1 of them can be extended to the k^{th} node
  - Performance penalty can be high
  - In practice, don't generally go higher than 3-consistency ("Path Consistency")
- Strong K-Consistency: K-Consistency + (K-1)-Consistency + (K-2)-Consistency, etc.

Value Ordering: Minimum Remaining Values (MRV)

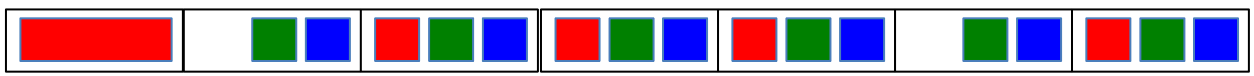
- **Minimum Remaining Values (MRV)**
  - Choose the *variable* with the fewest remaining *values* in its domain



- Basic idea:
  - Better to know sooner rather than later that a path will fail (if it will eventually fail)

Value Ordering: Least Constraining Value (LCV)

- **Least Constraining Value (LCV)**
  - Given a variable, choose the *value* that affects the fewest remaining unassigned variables

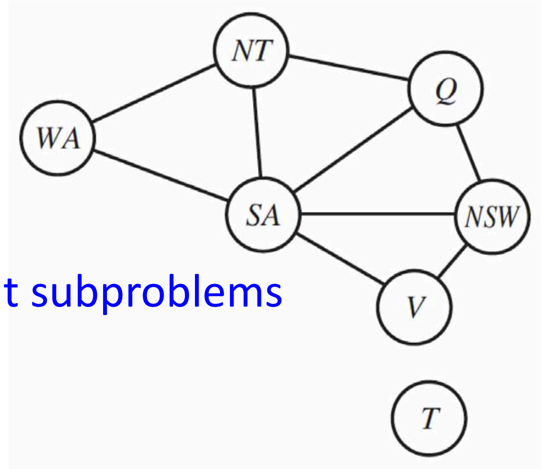


- Basic idea:
  - The more choices, the better chances of success, farther down in the tree

NOTE: Using both ordering ideas, 1000-queens problem becomes feasible !

Using Problem Structure

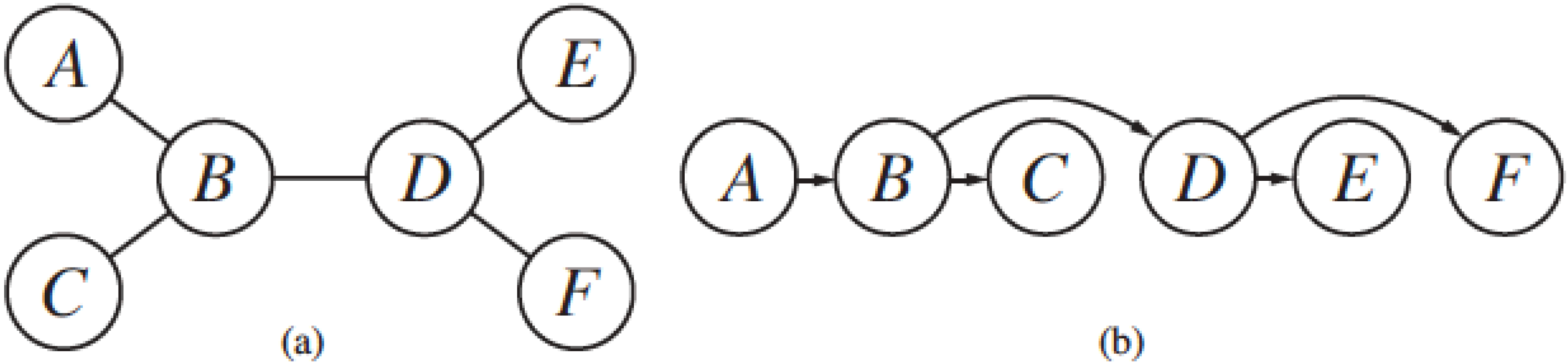
- Independent subproblems
  - e.g., Tasmania can be any color, always



- Constraint graph can be composed of independent subproblems

- Savings can be huge
  - Given: graph with n variables can be decomposed into problems of size c
  - Worst-case solution cost: $O\ ((n/c)(d^c))$, which is linear in n
    - Example: n = 80, d = 2, c = 20
    - $2^{80} \cong$ 4 billion years at 10 million nodes/sec
    - compare to $(4)(2^{20}) \cong 0.4$ seconds at 10 million nodes/sec

Tree-Structured CSPs

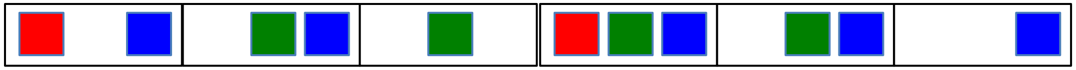


- **Tree-structured CSP algorithm**
  - Graph must be a tree (no cycles)
  - Select a start node and do a topological sort to convert (a) to (b)
  - ***Remove backward***, starting from right end, removing inconsistent
  - ***Assign forward***, starting from first node

  - Forward pass will not need to backtrack
  - Runtime is O(n * d^2) ← *d^2 is due to backward pass*

Tree-Structured CSP Example

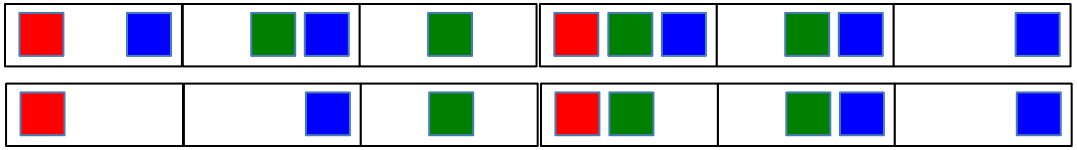
Start



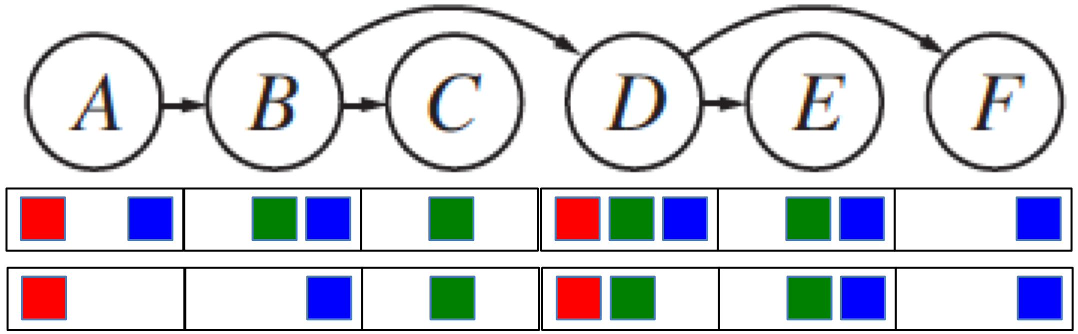
Tree-Structured CSP Example

Start

After removes



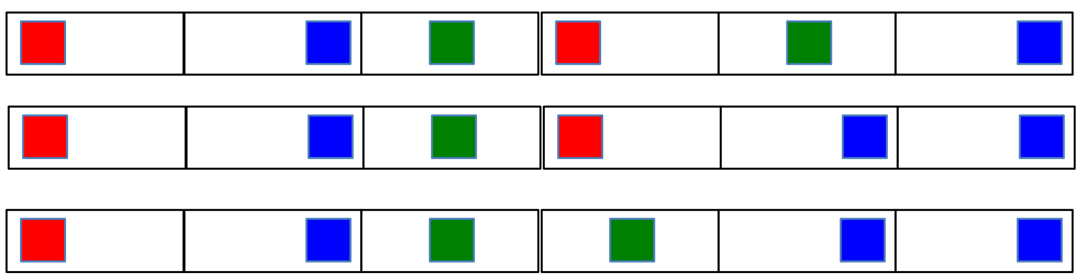
Tree-Structured CSP Example



Start

After removes

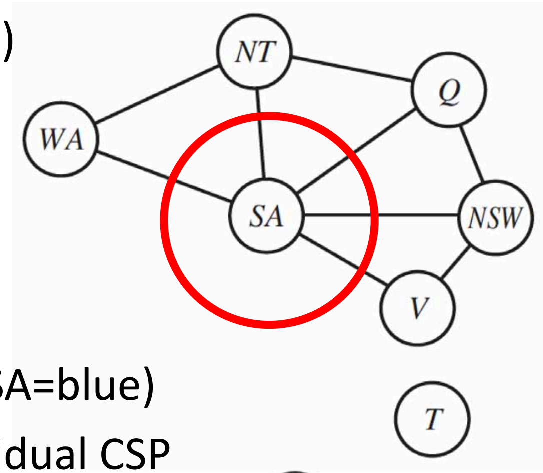
Valid assignments

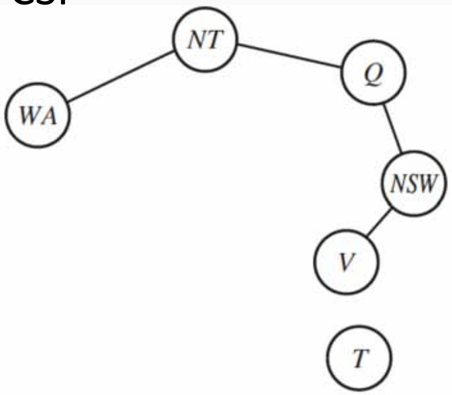


Using Structure: Cutsets

- Basic idea:
  - Find a subset of nodes whose removal will result in a tree structure
  - Instantiate the cutset (all possible assignments)
  - Cut out the cutset (compute residual CSPs)
  - Solve the (tree structured residual CSPs)

  - Here, our cutset is { SA }
  - Try all assignments for SA (SA=red, SA=green, SA=blue)
  - For each assignment to SA, try to solve the residual CSP



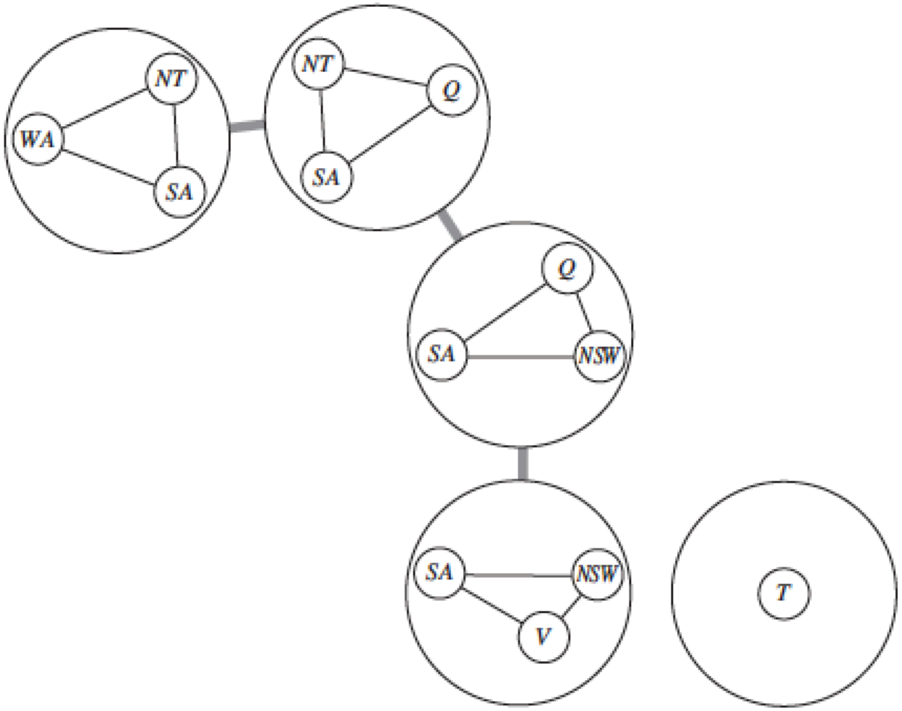


  - Runtime is O((dc)*(n-c)*d^2), where c = cutset size

Tree Decomposition

- Basic idea:
  - Create a tree-structured graph of “mega-variables”
  - Overlap ensures consistency

- Solution procedure
  - Solve each subproblem separately
  - Solve the constraints connecting the subproblems using our tree-structured CSP algorithm



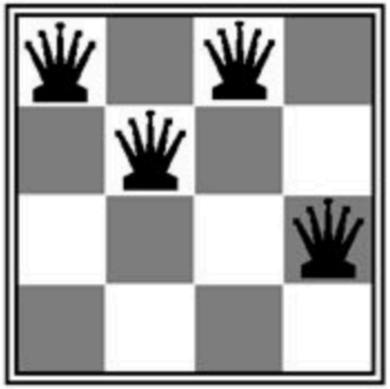
Iterative Methods for CSPs

- **Iterative methods** use local search methods that work with complete assignments and iterate to satisfy the constraints
- Basic idea
  - Start with a complete assignment with unsatisfied constraints
  - Use operators to reassign variable values
  - Iterate until a solution found or exhaust all possibilities
  - Note: No fringe – work with just one assignment !
- **Iterative Min Conflicts algorithm**
  - Variable selection: random choice from among conflicting variables
  - "Min conflicts" value selection: choose value that results in fewest constraint violations

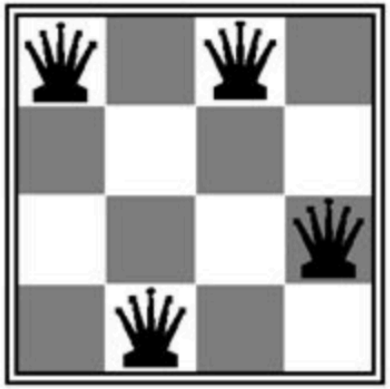
  → This is hill climbing with heuristic h(n) = total number of constraints violated

Example: 4-Queens (reprise)

h = 5



h = 3



h = 1

- States:
  - 4 queens, 1 in each column (4^4 = 256 total states)
- Operator:
  - Move a queen vertically in its column
- Goal test:
  - No queen threatens another
- Heuristic:
  - h(n) = number of binary attacks

Q: What's the next move?