COMS W4701: Artificial Intelligence

Lecture 3: Constraint Satisfaction Problems

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Today

- Constraint satisfaction problems

- Backtracking search
- Ordering heuristics

- Inference and constraint propagation
- Local consistency

States with Structure

- Task environments so far: Fully observable, discrete, deterministic, static

- Transitions and heuristics are problem-specific
- States are *atomic* black boxes
- **Planning** solution is an action **sequence**, or state space **path**

- What if our states have a common **factored** representation?
- We can use *general-purpose* heuristics
- Solution of an **assignment** problem is the **goal** itself, not a path

Constraint Satisfaction Problems

- Special structured search problems with 3 components
  - Variables $X = \{X_1, \dots, X_n\}$
  - Domains $D = \{D_1, \dots, D_n\}$
  - Constraints $C = \{C_1, \dots, C_m\}$

- Goal test: A **complete, consistent** assignment of values to each variable X_i from respective domain D_i s.t. all constraints C are satisfied

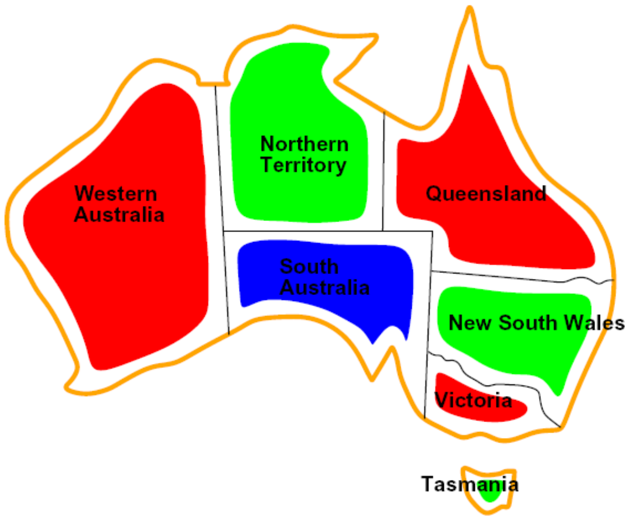
- We may have incomplete or inconsistent assignments along the way

Example: Map Coloring

- Goal: Color a map so that no adjacent territories have the same color

- Variables: $X = \{\text{WA, NT, Q, NSW, V, SA, T}\}$

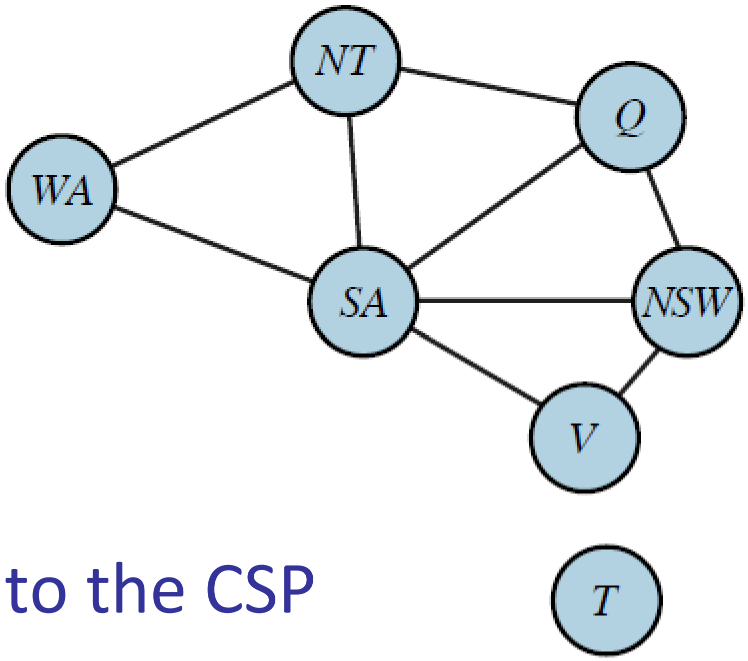
- Domains: $D_i = \{\text{red}, \text{green}, \text{blue}\}$



- Constraints: Implicit vs explicit representation
  - $C = \{\text{WA} \neq \text{NT}, \text{WA} \neq \text{SA}, \text{NT} \neq \text{SA}, \text{NT} \neq \text{Q}, \text{SA} \neq \text{Q}, \text{Q} \neq \text{NSW}, \text{NSW} \neq \text{V}\}$
  - $C = \{(\text{WA}, \text{NT}) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), \ldots\}, \ldots\}$

Constraint Graphs

- Visualization of a CSP in a graph



- Nodes: Variables and domains
- Edges: Presence of binary constraints

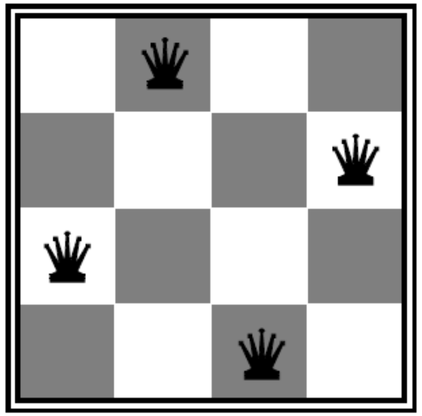
- Certain graph algorithms can help give insight into the CSP
- E.g., a k-connected component indicates a k-nary constraint
- CSPs that are actually tree structures can be solved without backtracking

Types of Constraints

- So far we've seen **binary** constraints relating two variables
- **Unary** constraints involve a single variable, equivalent to *domain reduction*
  - Ex: SA $\neq$ green (implicit), SA $\in$ {green, red} (explicit)
- Higher-order (**global**) constraints relate arbitrary number of variables
  - Ex: *Alldiff* requiring all variables to have different values
- If domains are finite, always possible to rewrite global constraints as binary ones using *auxiliary variables*

Example: n-Queens

- Place n queens on $n \times n$ board s.t. none share a row, column, or diagonal
- Variables: X_{ij}, $1 \le i \le n$, $1 \le j \le n$ (grid spaces)
- Domains: $D_{ij} = \{0,1\}$ (queen or no queen)
- Constraints:



$$\forall i \sum_j X_{ij} = 1 \qquad \forall i,j \sum_k X_{i+k,j+k} \le 1$$

$$\forall j \sum_i X_{ij} = 1 \qquad \forall i,j \sum_k X_{i+k,j-k} \le 1$$

Example: n-Queens

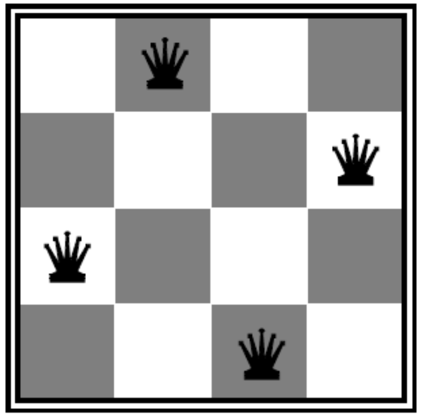
- Alternatively, states can just represent a row of the board (vs grid space)

- Variables: X_i, $1 \leq i \leq n$ (row of the board)

- Domains: $D_i = \{1, \ldots, n\}$ (column containing queen)

- Constraints: $\forall i, j, X_i \neq X_j$

  $\forall i, j, X_j - X_i \neq |j - i|$



Example: Cryptarithmetic

- Variables: $X = \{T, W, O, F, U, R, \dots\}$
- Also carry-overs! C_1, C_2, C_3

- Domains: $D_i = \{0, \dots, 9\}$

- Constraints
  - $Alldiff(X_i)$

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

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

- 7 possible solutions!

$734 + 734 = 1468$
$765 + 765 = 1530$
$836 + 836 = 1672$
$846 + 846 = 1692$
$867 + 867 = 1734$
$928 + 928 = 1856$
$938 + 938 = 1876$

Example: Sudoku

- Variables: One for each open square
- Domains: $\{1, \dots, 9\}$
- Constraints:
  - $Alldiff$ for each column
  - $Alldiff$ for each row
  - $Alldiff$ for each 3×3 square

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8				4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

Solving CSPs with Search

- **States**: Partial assignments (initial state is no assignment)
- **Actions**: Assign value to an unassigned variable from its domain
- **Goal test**: Complete, consistent assignment
- No explicit costs, so maybe something like DFS/BFS?

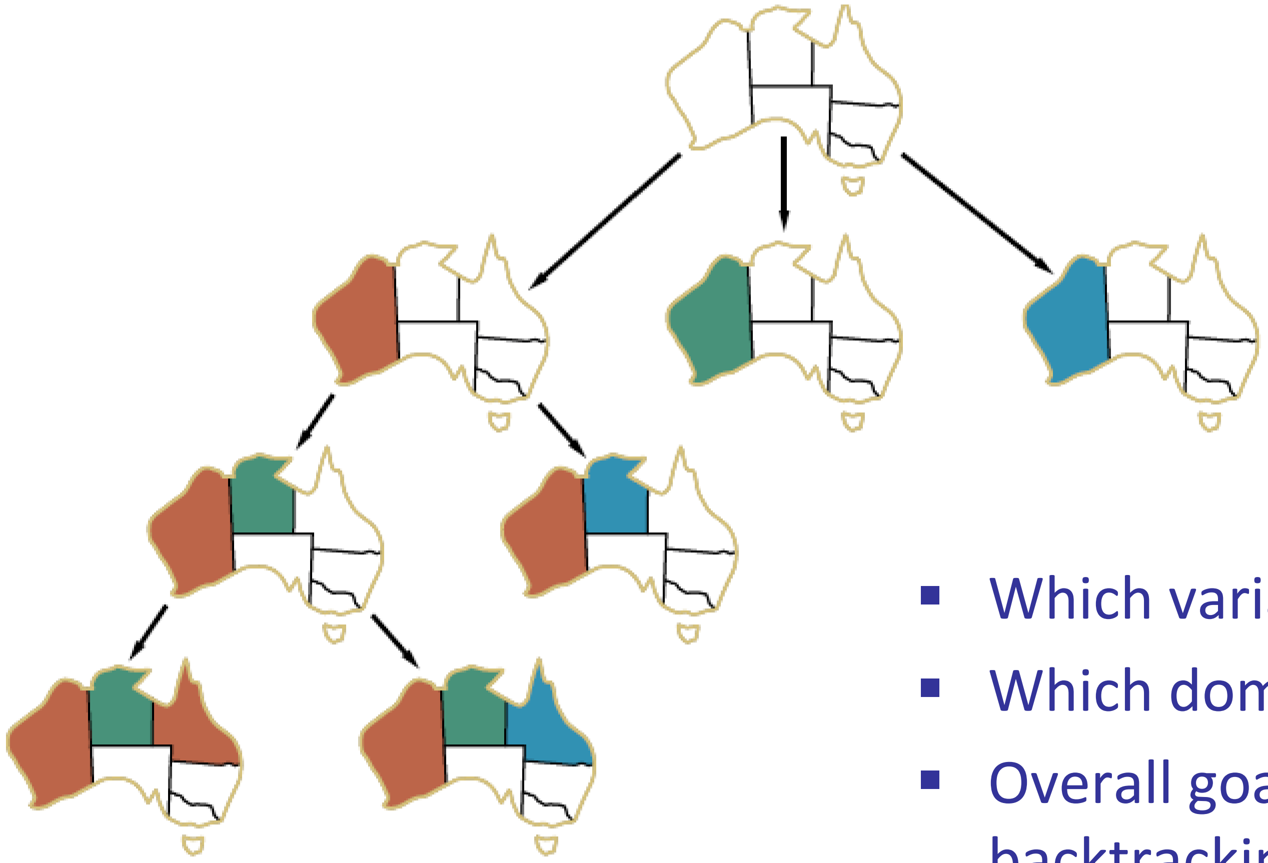
- Problem: Naïve implementation -> lots of repeated states!

- Branching factor: Number of unassigned variables × size of domain
  - Branching factor at root: $n \times d$ for n variables with domain size d
  - Branching factor at 2nd level is $(n-1)d$, then $(n-2)d$ at 3rd level, …
  - Total number of leaves / possible goal states: $O(n! \times d^n)$

Backtracking Search

- Idea: CSPs are *commutative*, order of variable assignment doesn't matter
  - (WA = red, NT = green) is the same as (NT = green, WA = red)
- Each search tree tier only needs to correspond to a single variable
  - Branching factor is d at every level, number of leaves is $O(d^n)$
- Running DFS allows us to make one variable assignment at a time and *backtrack* by undoing inconsistent assignments to try alternatives
  - Don't need a frontier—just keep track of what domain values are available

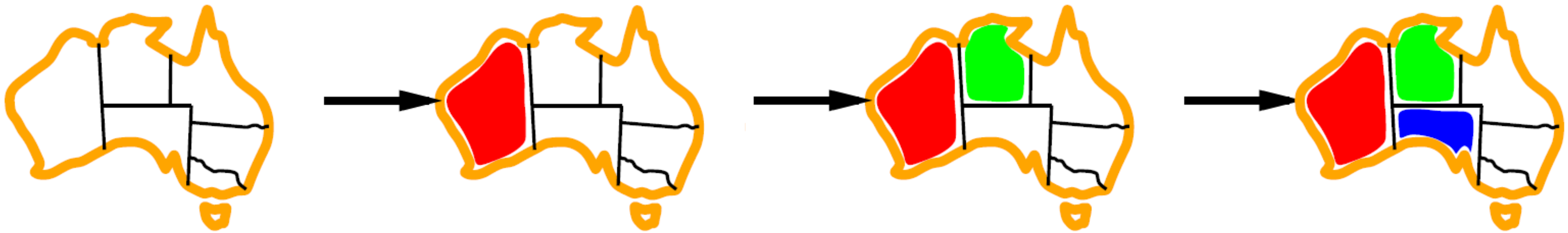
Backtracking Search



- Which variables do we try first?
- Which domain values do we try first?
- Overall goal: Minimize amount of backtracking needed, or wasted effort

Variable Selection

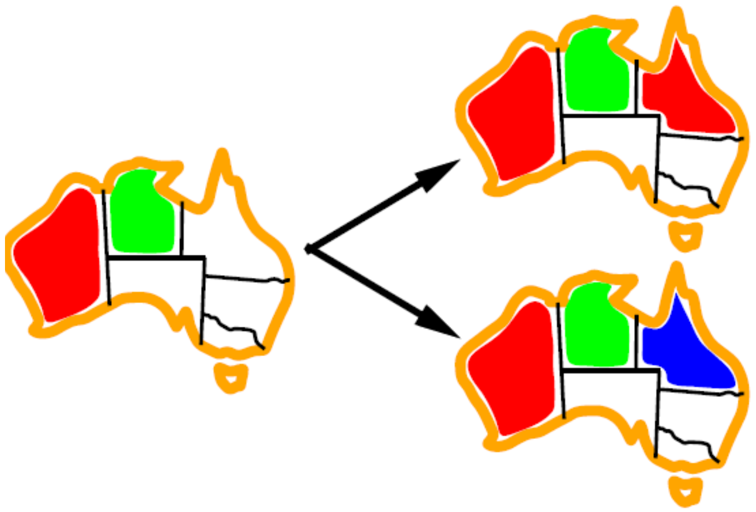
- Heuristic for picking next variable: Minimum remaining values (**MRV**)
  - Also called "most constrained variable" (MCV) or "fail-first"
  - Choose the variable with the fewest legal left values in its domain



- What if we have multiple MRV variables (e.g., initial assignment)?
  - *Degree* heuristic: Pick variable that appears in the most constraints
  - Reduce branching factor on future choices

Domain Value Assignment

- Heuristic for assigning a value: Least constraining value (**LCV**)
  - Choose a value that imposes fewest constraints on future assignments
  - May require some computation



- Why least, not most?
  - We don’t have to use all values, just find an assignment that works
  - Try to look for the most likely ones earlier
  - Keep more options open for other variables

Improving Backtracking: Inference

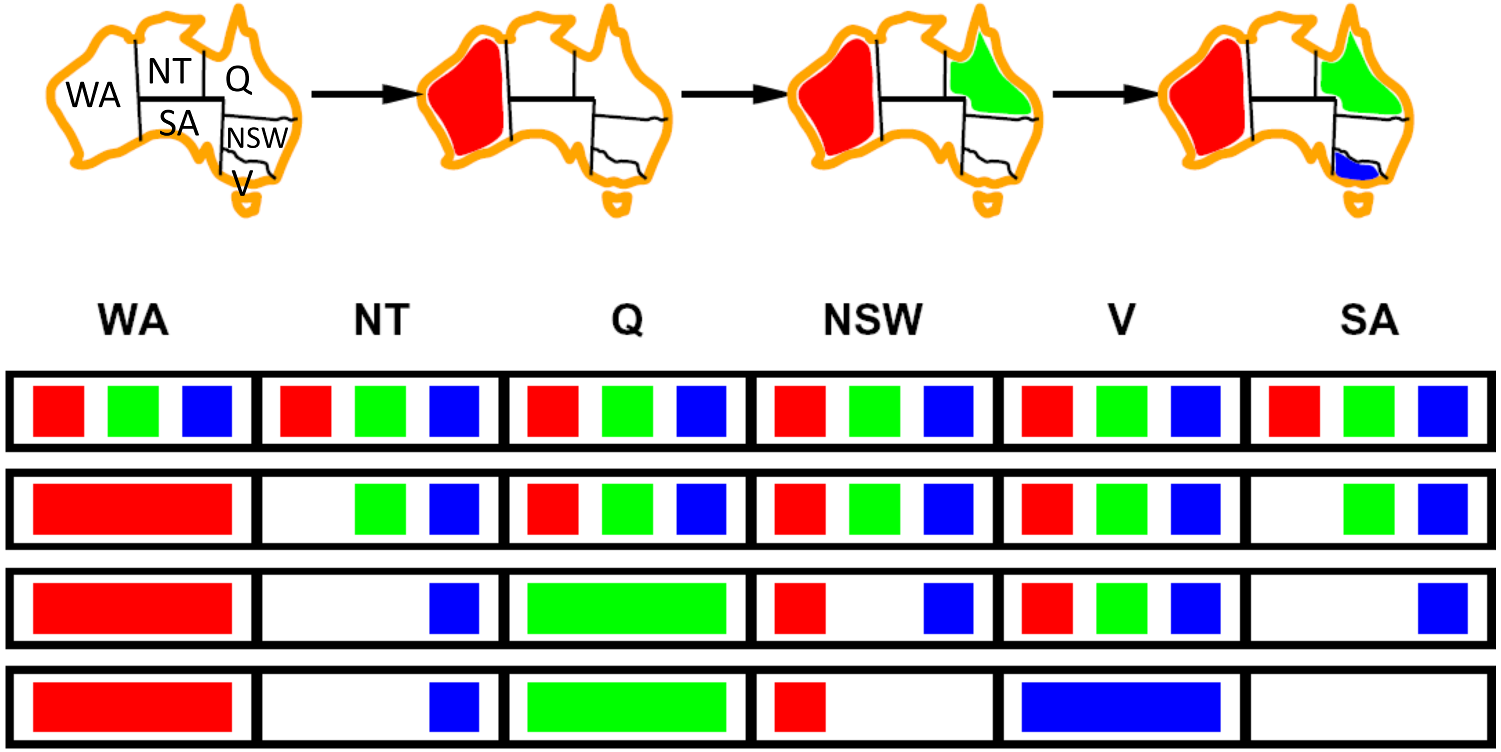
- Both MRV and LCV heuristics assume some sort of “forward checking”
- We can detect inevitable failure earlier by checking constraints as we go

- Idea: Delete inconsistent domain values as we make assignments
- If any domain is left empty, we should not proceed

- Domain reduction also better informs MRV on selecting variables
- Inference can also inform how to use the LCV heuristic

Forward Checking

- **Forward checking**: After assigning a variable, check other variables related to it by a constraint and eliminate inconsistent domain values
- If any domain becomes empty, current assignment is denoted a failure



Local Consistency

- Forward checking is an example of **constraint propagation**
- Can be interleaved with search, or done as a preprocessing step prior to search
- Result is **local consistency**: remaining domain values do not violate any constraints

- **Node consistency**: All unary constraints are satisfied
  - Can always be done prior to starting any search process

- **Arc consistency**: No binary constraint is violated by any variable's domain values
  - X_i is arc-consistent with respect to X_j if every value in D_i can be paired with a value in D_j such that no constraint between X_i and X_j is violated.

Arc Consistency

- X_i being arc consistent with X_j does not imply that X_j is arc consistent with X_i!

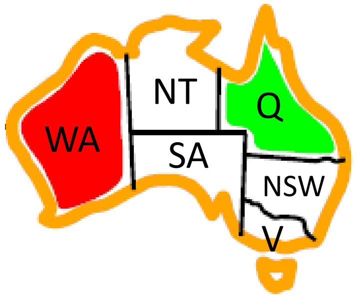
- Example: $Y = X^2$, both with domains $\{0,1,\ldots,9\}$
  - To make X arc consistent with Y: Reduce domain of X to $\{0,1,2,3\}$
  - To make Y arc consistent with X: Reduce domain of Y to $\{0,1,4,9\}$
- Entire CSP is arc consistent iff both domains are reduced

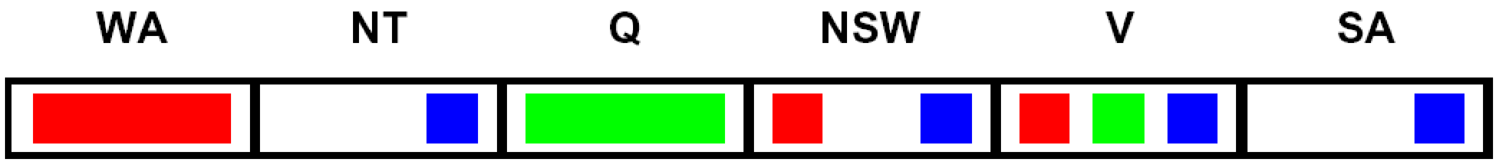
- Another issue: Arc consistency may not be preserved if a domain is reduced
- If X_i is arc consistent with X_j and X_j is changed (e.g. due to forward checking), X_i *may no longer be arc consistent with* X_j

AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints* whose variables' domains have changed until all are arc consistent

- More general than forward checking
- Must maintain a dynamic queue of constraints to check

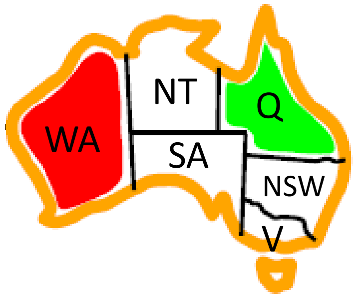


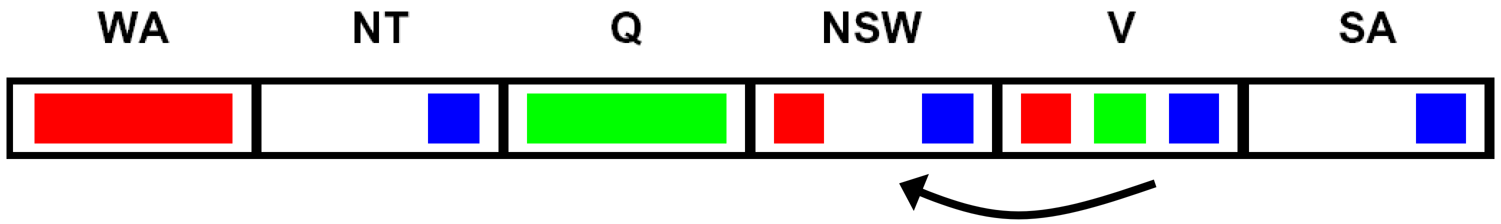


AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints whose variables' domains have changed until all are arc consistent*

- More general than forward checking
- Must maintain a dynamic queue of constraints to check

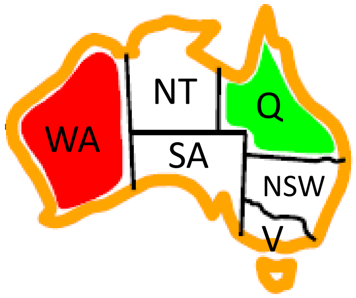


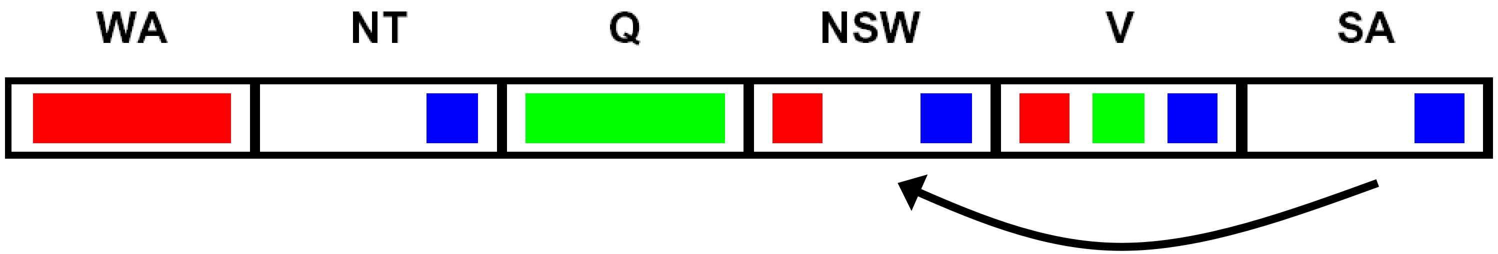


AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints* whose variables' domains have changed until all are arc consistent

- More general than forward checking
- Must maintain a dynamic queue of constraints to check

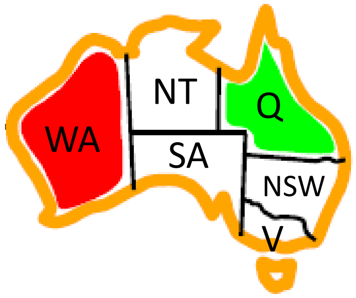


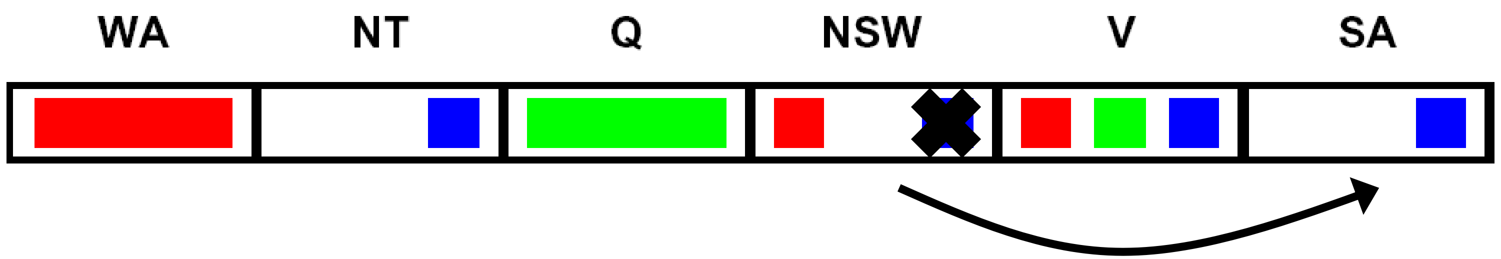


AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints* whose variables' domains have changed until all are arc consistent

- More general than forward checking
- Must maintain a dynamic queue of constraints to check

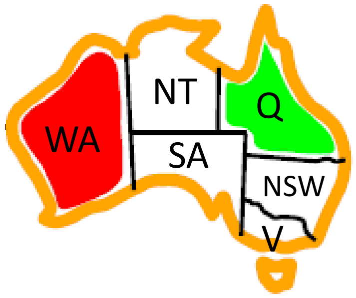


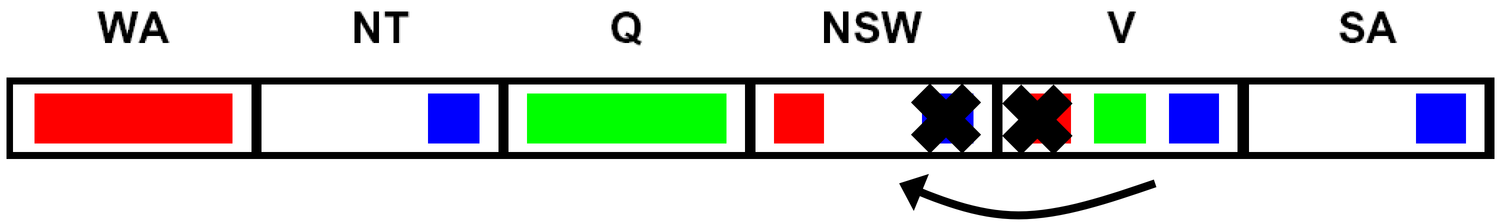


AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints whose variables' domains have changed until all are arc consistent*

- More general than forward checking
- Must maintain a dynamic queue of constraints to check

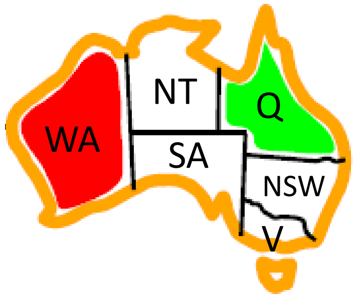


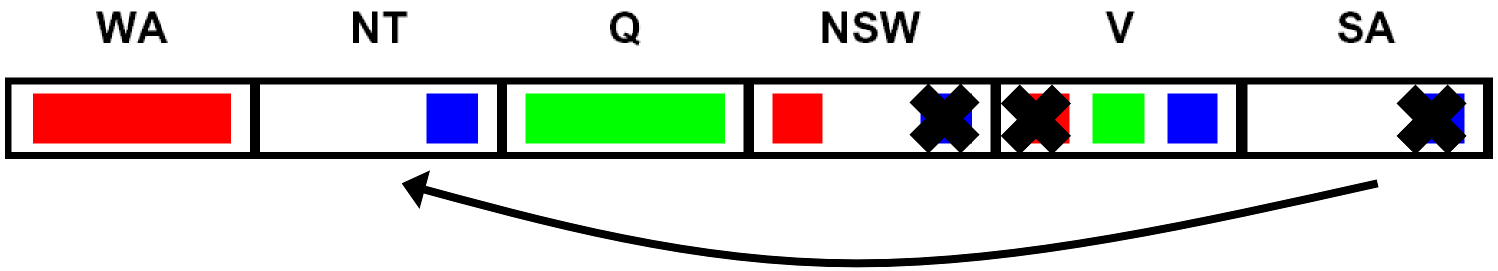


AC-3 Algorithm

- To enforce arc consistency, e.g. during backtracking search, *loop over all constraints whose variables' domains have changed until all are arc consistent*

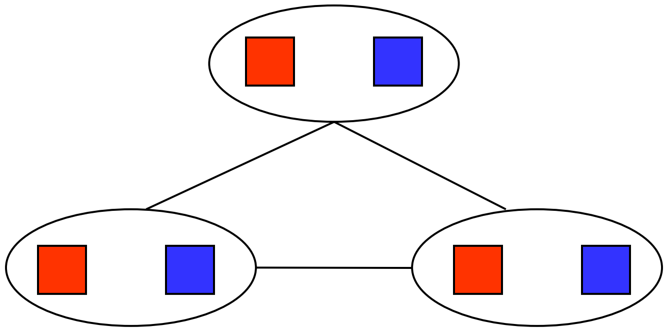
- More general than forward checking
- Must maintain a dynamic queue of constraints to check





Path and k-Consistency

- At least one arc inconsistent $\Rightarrow$ no solution
- BUT arc consistency not sufficient to guarantee a solution!
- Idea: Look at more than two variables at a time



- $\{X_i, X_j\}$ is **path consistent** with X_m if for every arc consistent assignment to $\{X_i, X_j\}$, there is an assignment to X_m consistent with X_i and X_j

- ***k*-consistency**: For every consistent assignment to a set of $k-1$ variables, a consistent value can be assigned to the kth variable
- A **strongly *n*-consistent** CSP ($n, n-1, \ldots, 1$ consistent) is guaranteed a solution, but determining n-consistency is exponentially hard (NP-complete)!

Special Constraints

- Consistency for certain constraint types can be checked quickly

- *Alldiff* with m variables and n unique values: immediately declare no solution if $m > n$

- Suppose we have domains with lower and upper bounds
- **Bounds propagation**: If constraints are equality or inequality (e.g., **resource constraints**), we can use them to tighten the bounds and make them consistent

- Ex: X_1, X_2, X_3, all with domains $[1,5]$
  - Constraint $\sum X_i = 13$: Reduce domains to $[3,5]$
  - Constraint $\sum X_i \leq 5$: Reduce domains to $[1,3]$

Introducing New Constraints

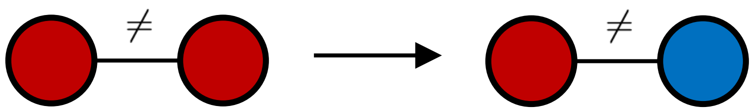
- We can also improve CSP solvers by *introducing* new constraints

- Recall: We backtrack whenever we see an inconsistent solution, either in the present (leaf of the search tree) or in the future (via constraint propagation)
- **Constraint learning**: Record the current assignment as a constraint

- A CSP may have multiple solutions due to *value symmetry*
- Ex: In map coloring, there are $d!$ solutions by permutation
- Idea: Introduce **symmetry-breaking constraints** (e.g., $NT < SA < WA$) to reduce the number of solutions and shrink the search tree

Local Search

- Methods so far: Build up solution incrementally, check constraints
- **Local search**: Start with an arbitrary complete assignment, *modify* it until consistent
- No frontier to maintain, no backtracking!

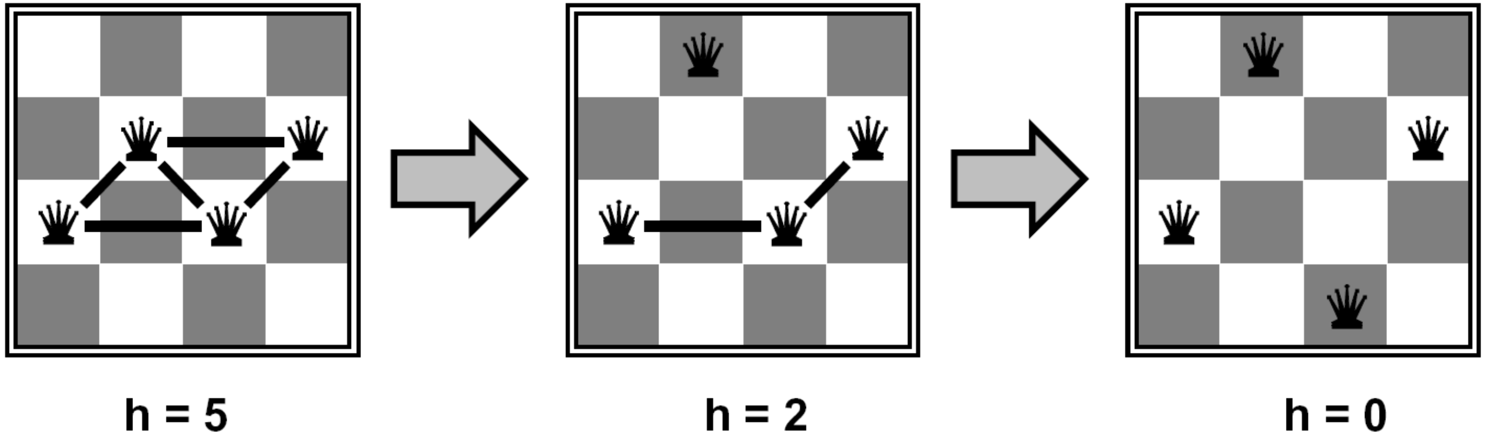


- **Min-conflicts**: Reassign a variable to a value that *minimizes conflicts*

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

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

Example: 4-Queens

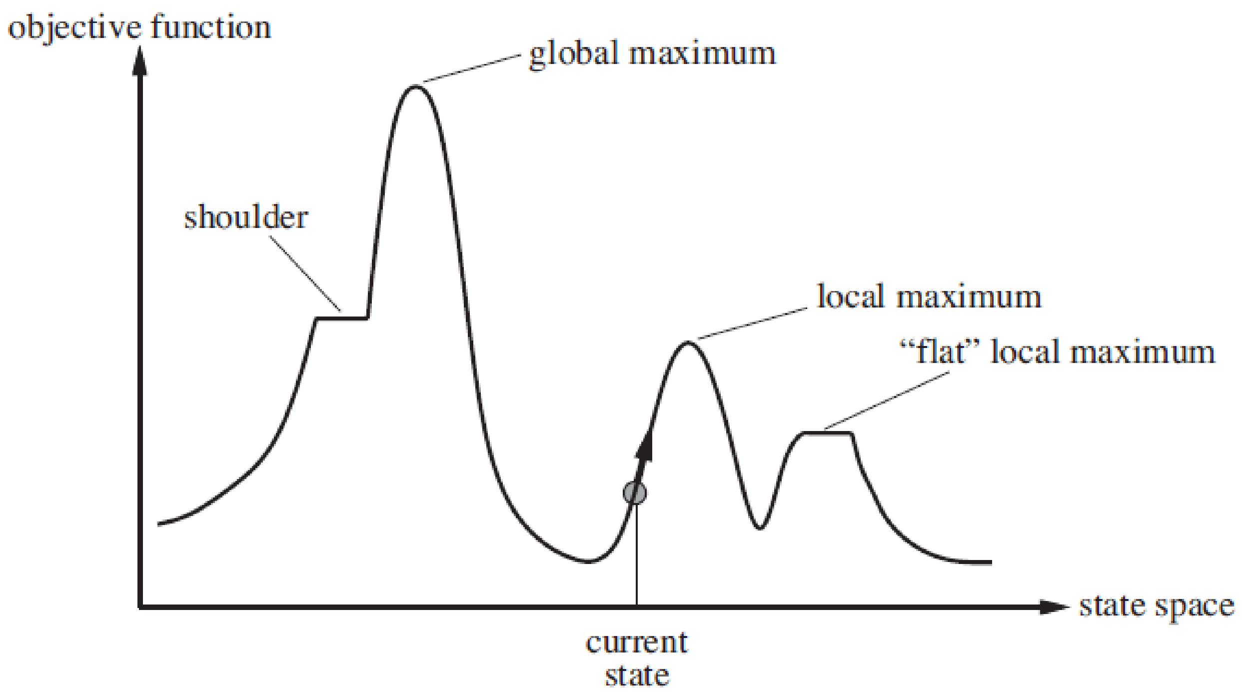


- Can be very effective in practice, roughly independent of problem size
- E.g., can solve the 1M-queens problem in an average of 50 iterations

- Also useful for solving problems online as they change
- For example, repair existing solutions as new constraints come and go

Local Search and Optimization*

- Local search generally useful when we care about goal more than path
- E.g., optimization problems with **objective function**, no specific goal test
- Pros: Very little memory, can work well in large or infinite state spaces



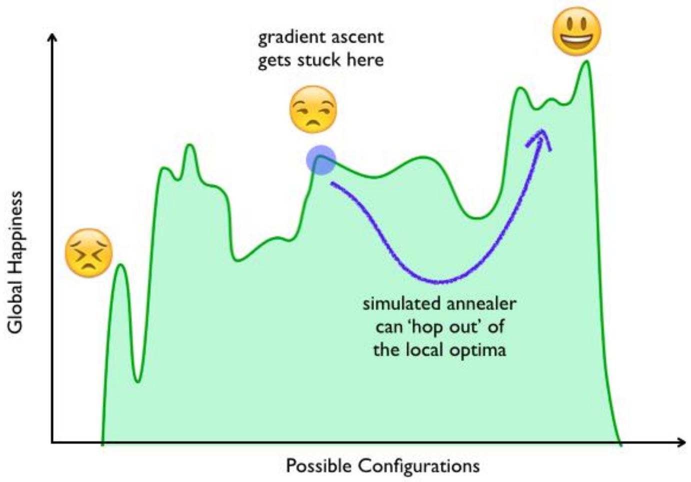
Hill-Climbing Methods*

- Many local search algorithms fall into category of *hill-climbing*
- No search tree, no knowledge beyond immediate neighbors
- Greedy search method, can get stuck at local maxima if unlucky

- Variations of simply ascending the hill:
- **Stochastic**: Pick random moves from time to time
- **First-choice**: Pick a random move that leads to a better successor
- **Random-restart**: Try different initial states if stuck

Other Methods*

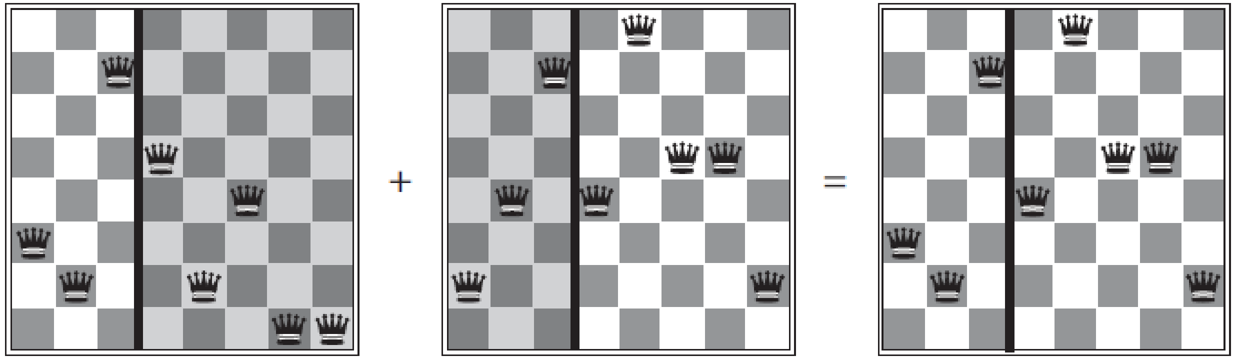
- **Simulated annealing**: Combine hill climbing with random walks to get unstuck from local maxima



https://eat.zesty.com/post/how-simulated-annealing-can-improve-your-lunch

- **Local beam search**: Maintain parallel searches but share information among threads

- **Genetic algorithms**: Generate successor states by combining two parent states



Summary

- CSPs are assignment search problems with very specific structure
- Unlike general DFS, backtracking search can be very effective
- General-purpose heuristics: MRV, LCV

- Inference can further improve search performance
- Maintain local consistency through constraint propagation, take advantage of or add new common constraints

- Local search: Alternative search method with few memory requirements