CS 188: Artificial Intelligence

Constraint Satisfaction Problems II

Instructor: Anca Dragan

University of California, Berkeley

Today

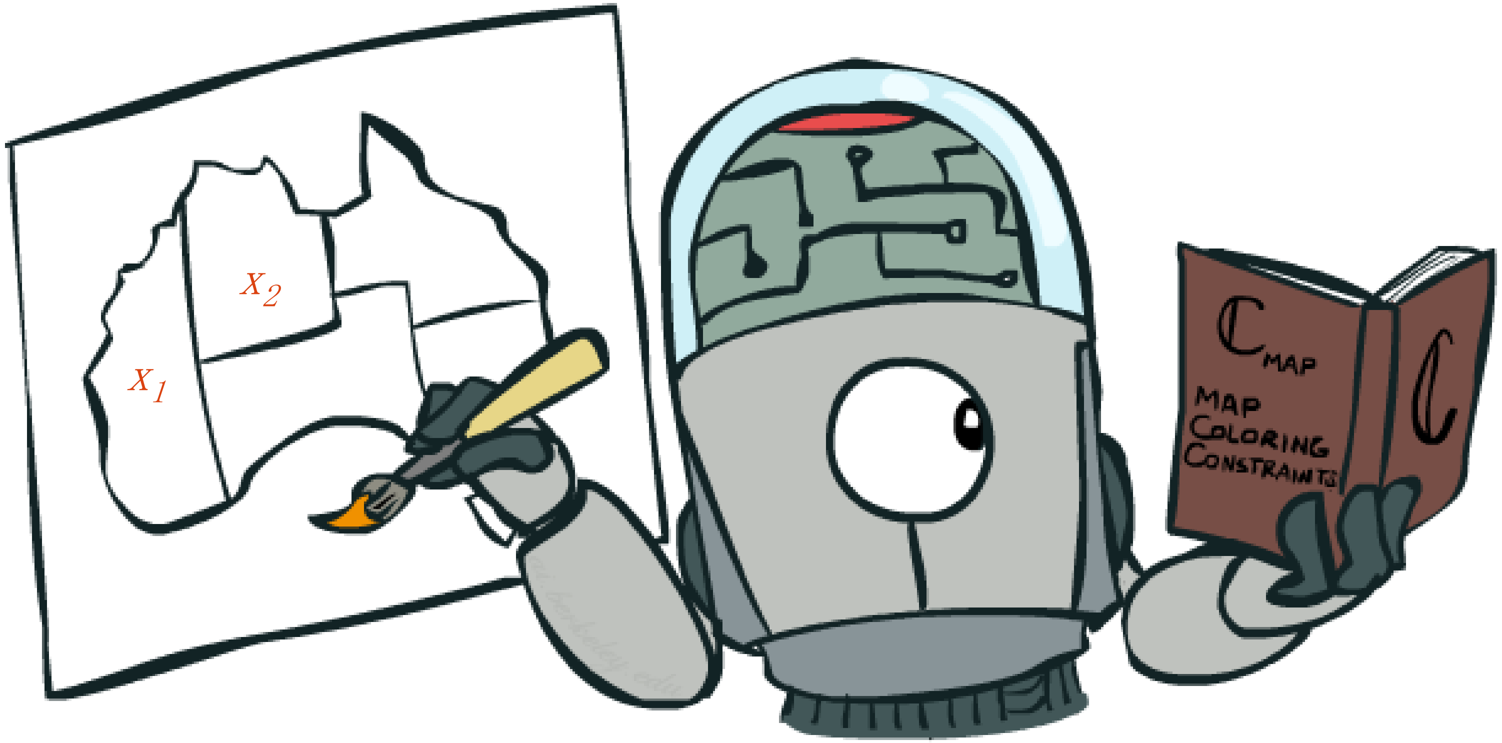
- Efficient Solution of CSPs
- Local Search

Constraint Satisfaction Problems

N variables

domain D

constraints

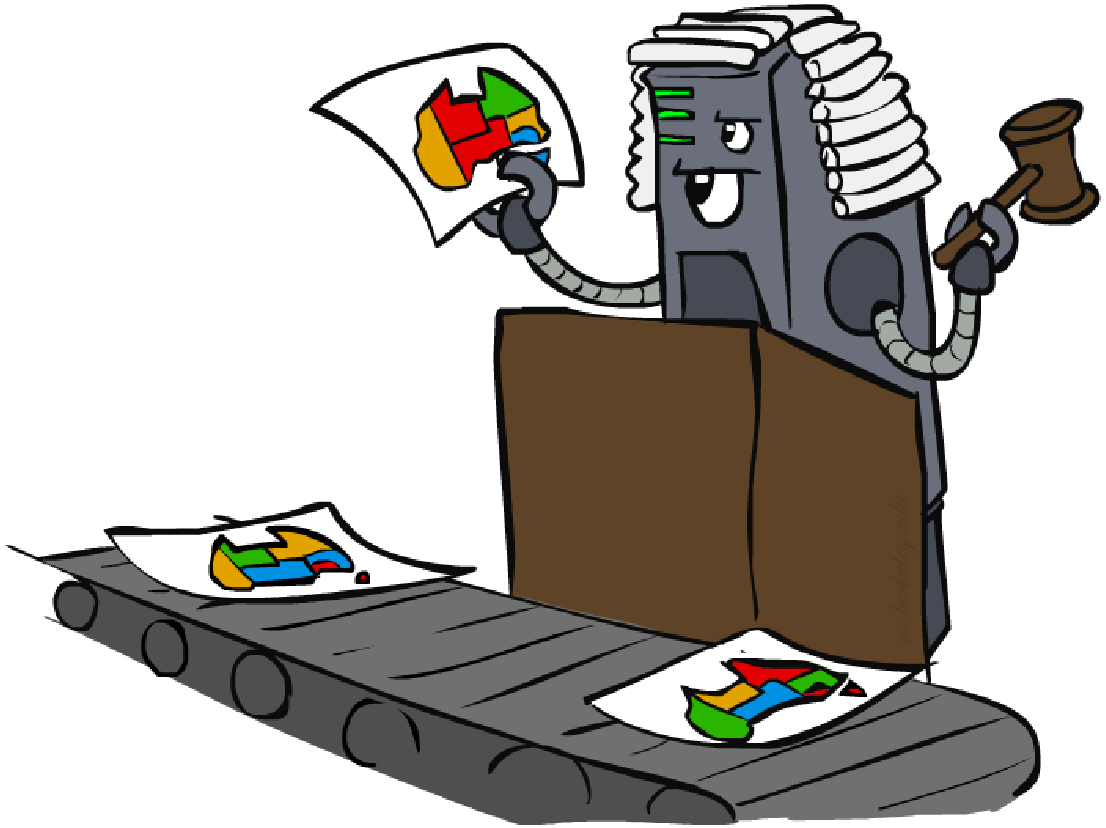


Standard Search Formulation

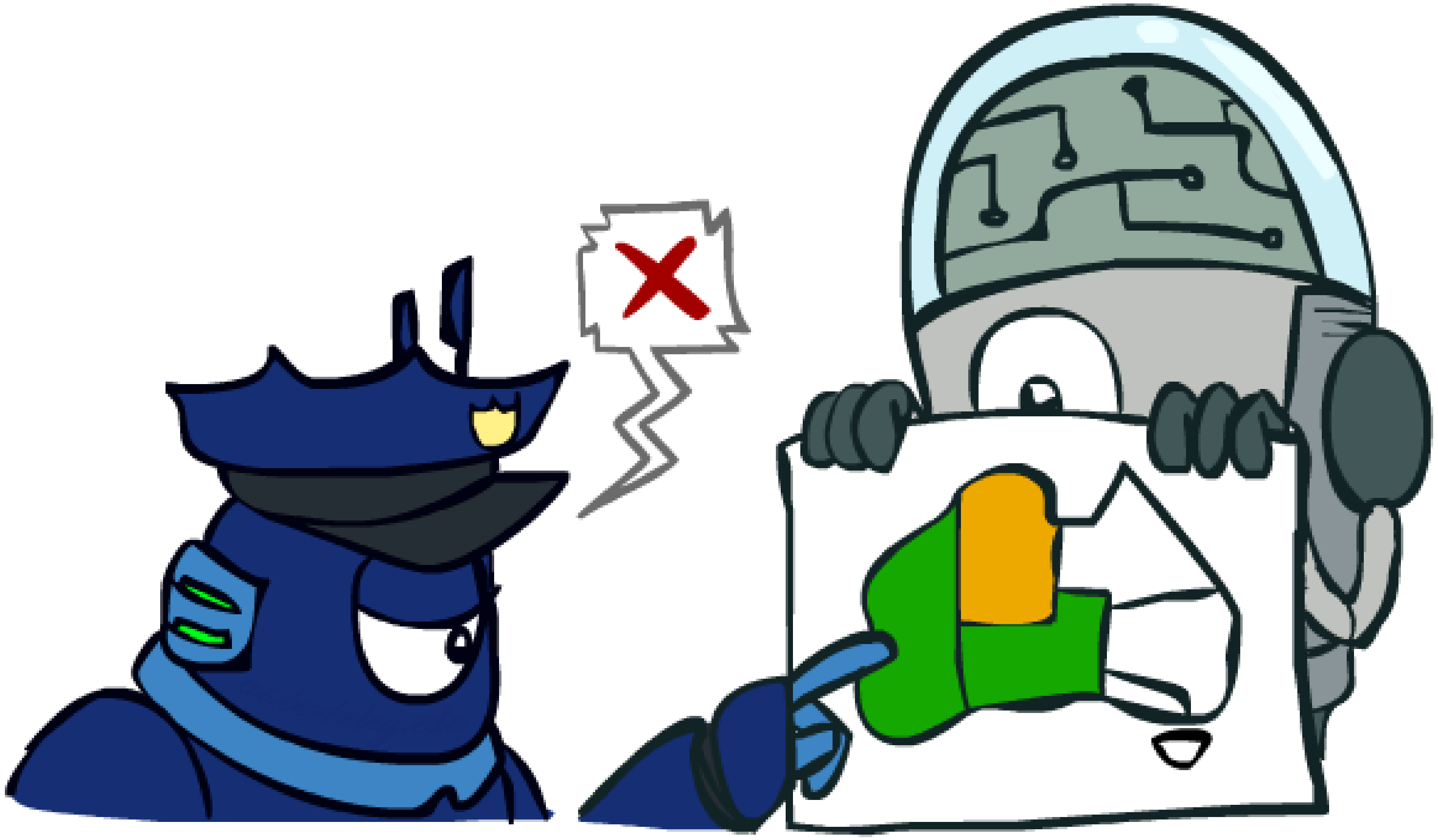
- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We started with the

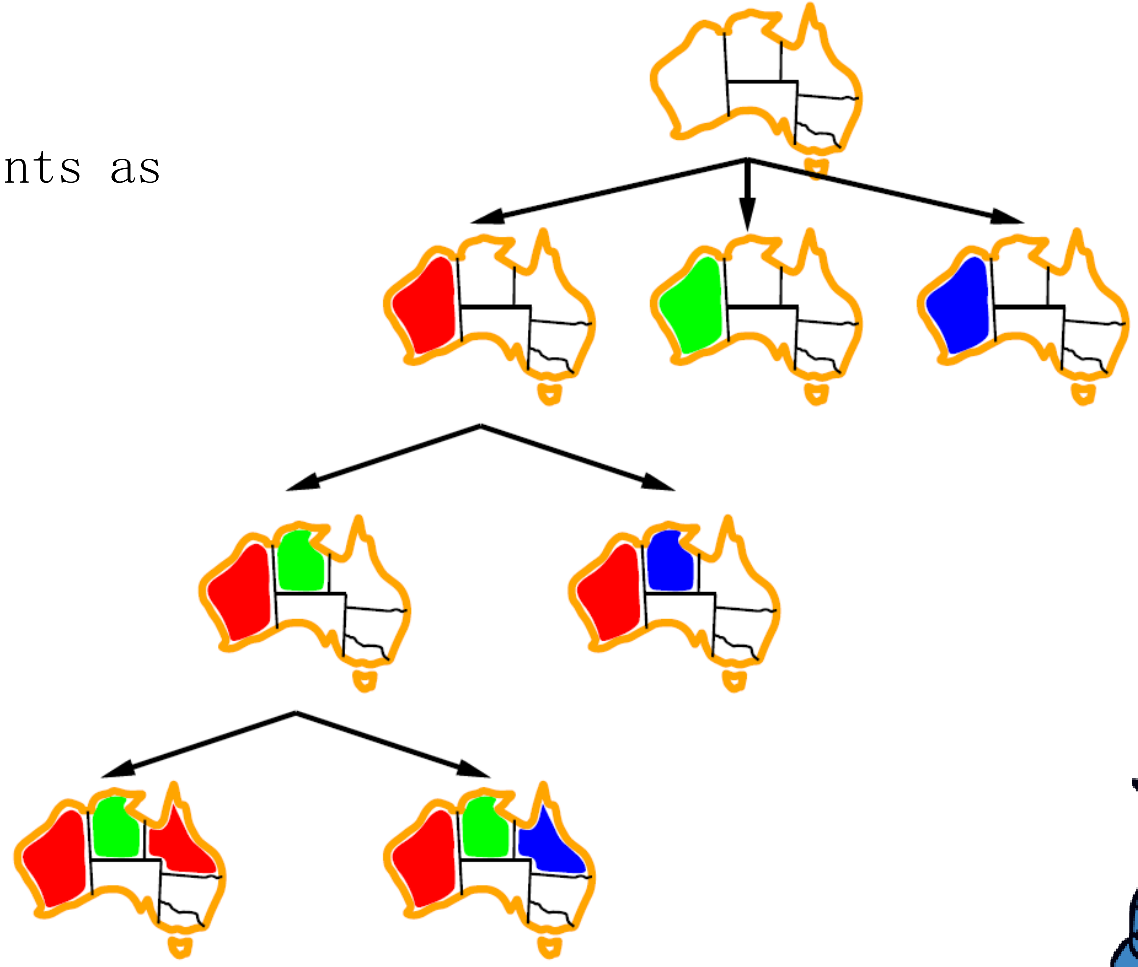


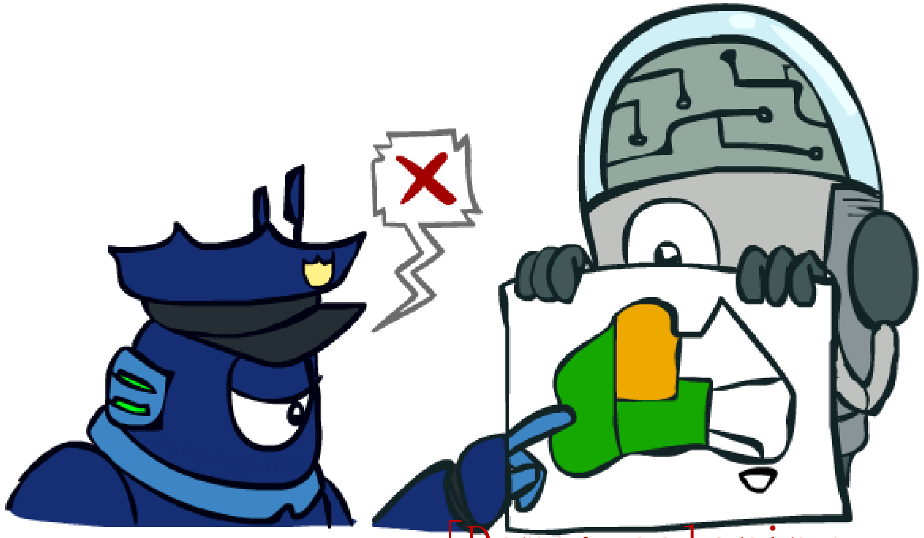
Backtracking Search



Backtracking Search

1. fix ordering
2. check constraints as you go





[Demo: coloring --

Explain it to your neighbor!

Why is it ok to fix the ordering of variables?

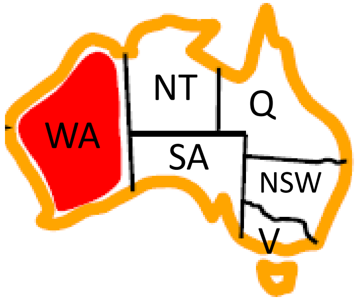
Why is it good to fix the ordering of variables?

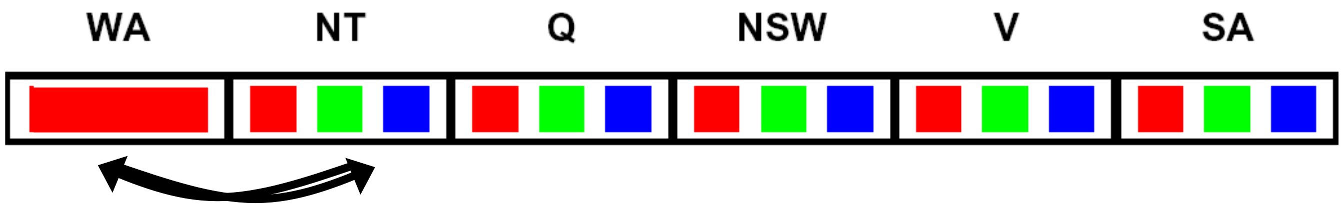
Filtering

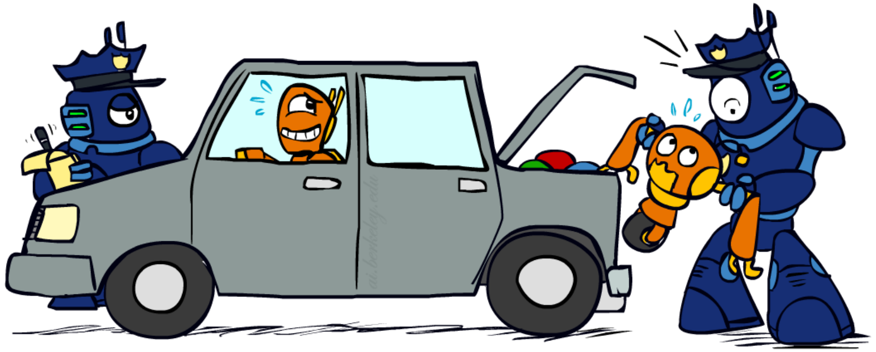
Keep track of domains for unassigned variables and cross off bad options

Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



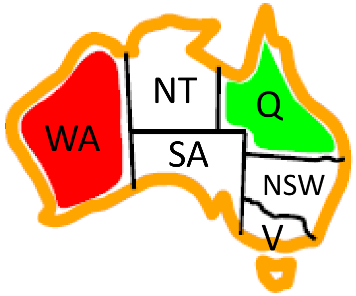


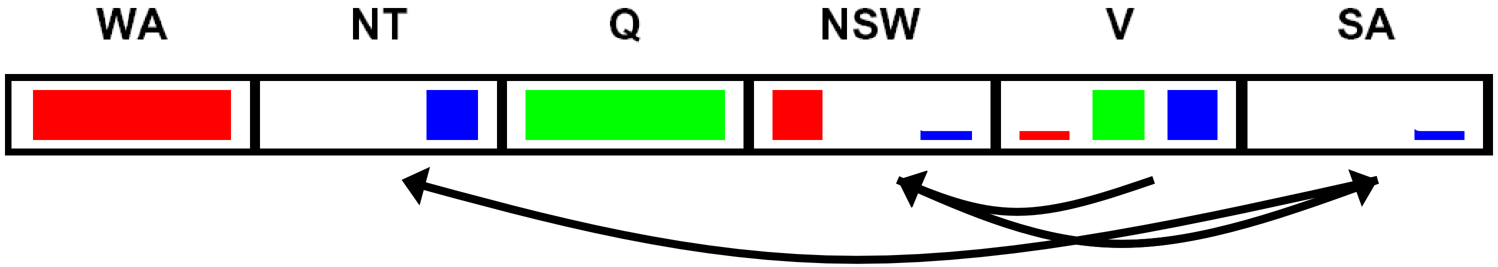


Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:





- Important: 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
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Enforcing Arc Consistency in a CSP

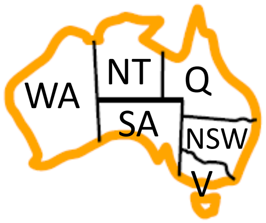
```
function AC-3( csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
    local variables: queue, a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (Xi, Xj) ← REMOVE-FIRST(queue)
        if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue
```

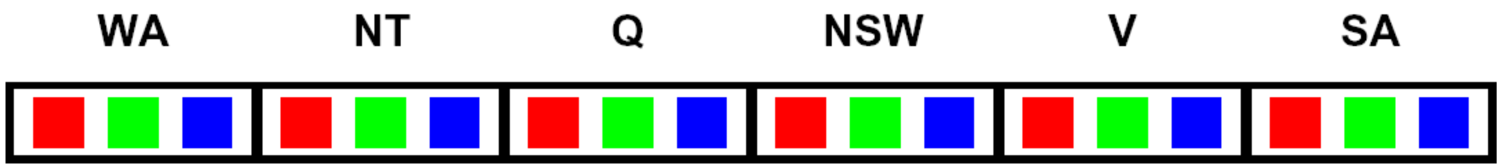
```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[Xi] do
        if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
            then delete x from DOMAIN[Xi];  removed ← true
    return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

Forward Checking – how does it relate?

- Forward checking: Cross off values that violate a constraint when adde to the existing assignment





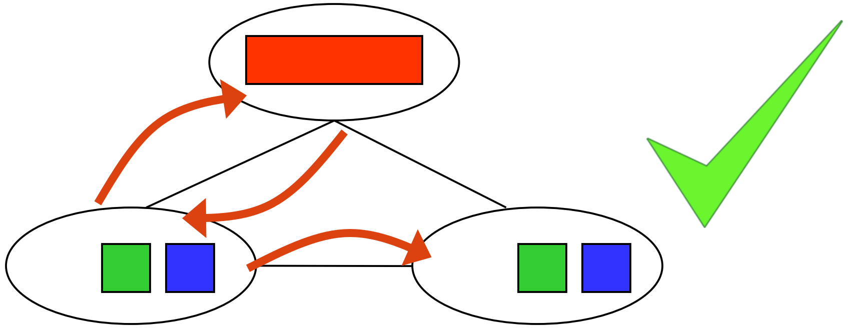
[Demo: coloring -- forward checking]

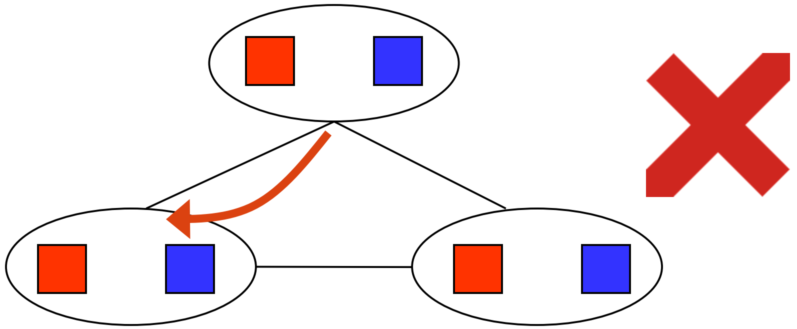
Explain it to your neighbor!

- Forward checking is a special type of enforcing arc consistency, in which we only enforce the arcs pointing into the newly assigned variable.

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

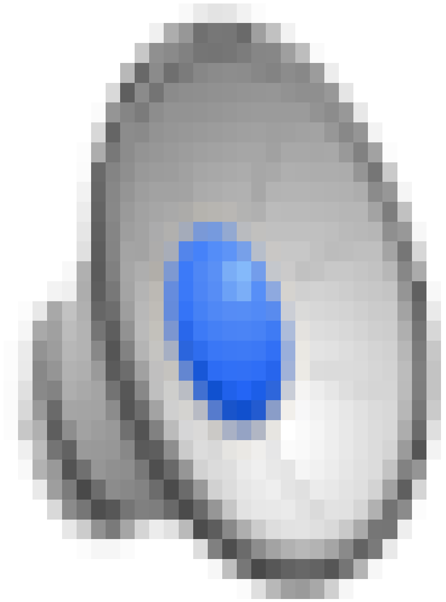




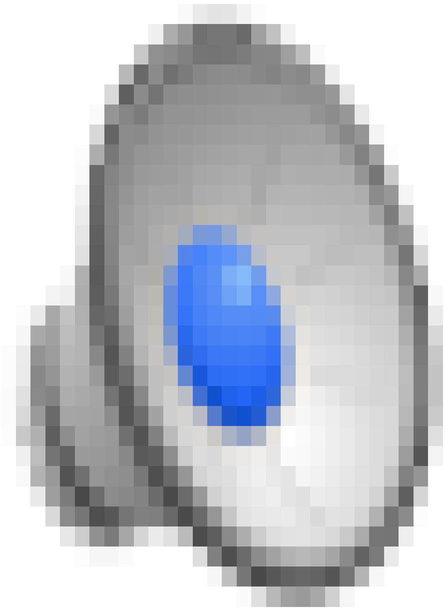
- Arc consistency still runs inside a backtracking search!

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph

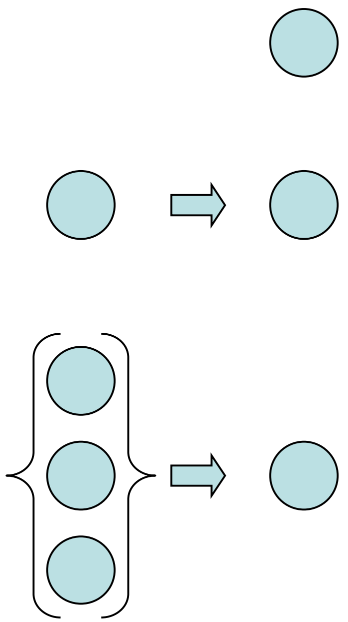


Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph

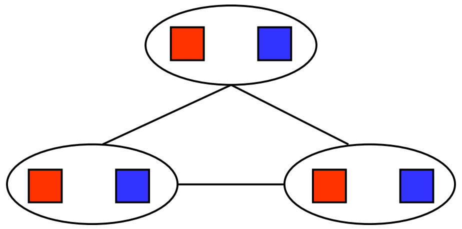


K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node’s domain has a value which meets that node’s unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.



- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)



Strong K-Consistency

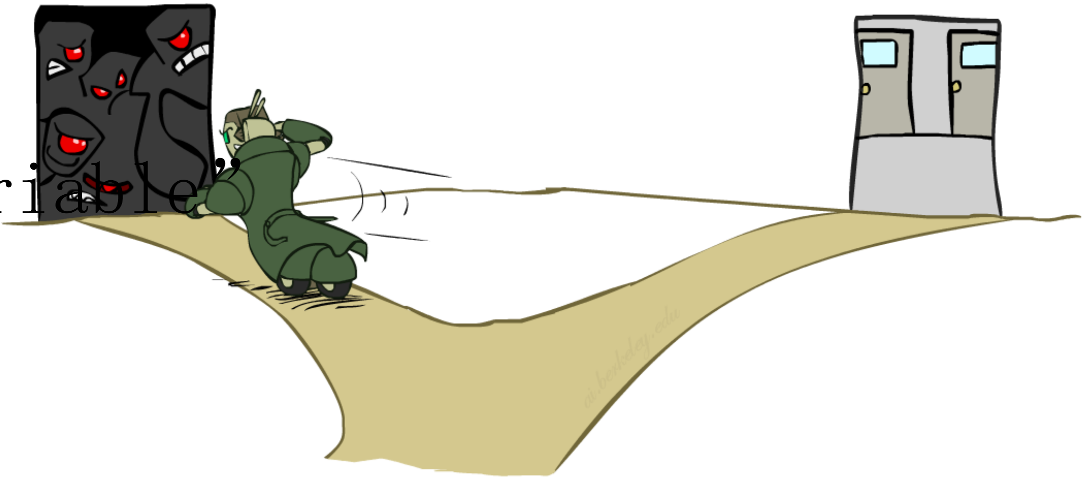
- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

Ordering

Ordering: Minimum Remaining Values

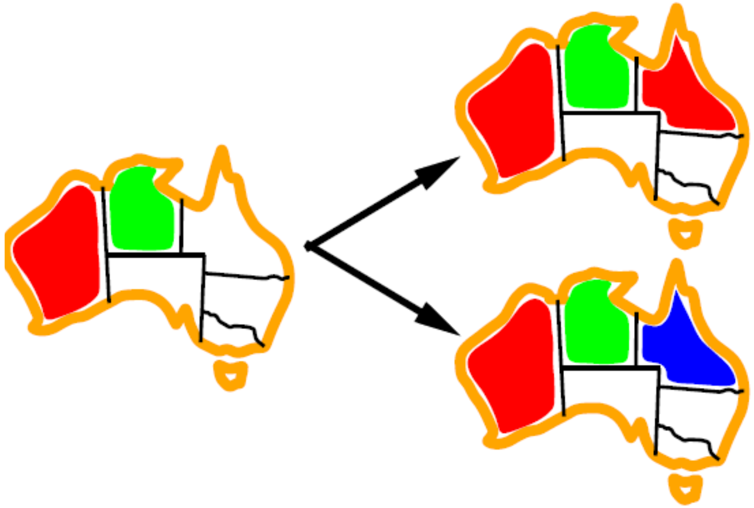
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering

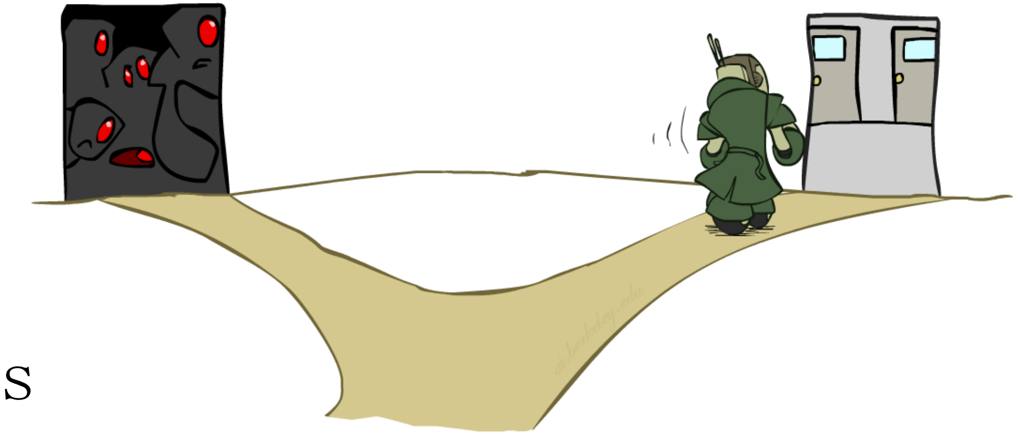


Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?



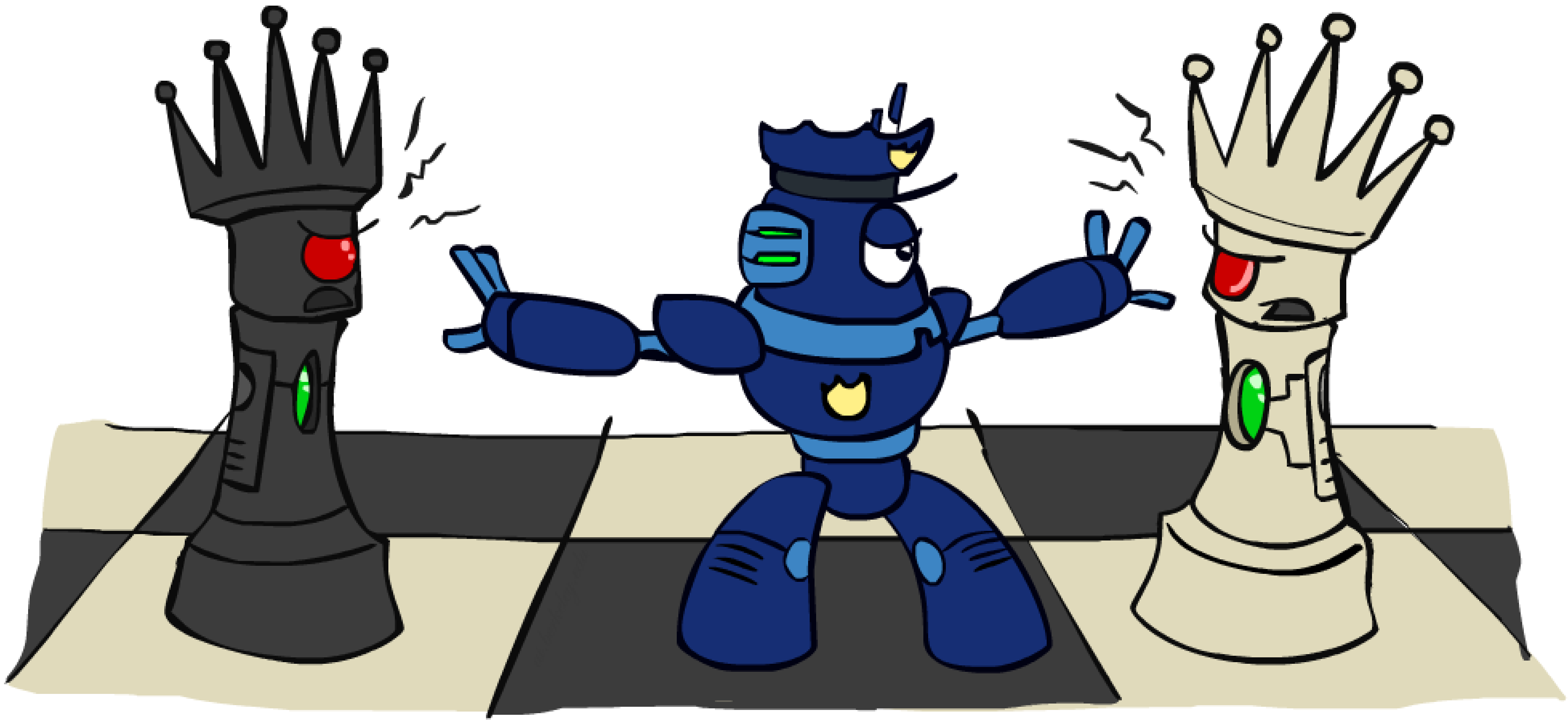
- Combining these ordering ideas makes 1000 queens feasible

Demo: Coloring -- Backtracking + Forward Checking + Ordering

Summary - Pair up!

- Work with your neighbor to write down:
  - How we order variables and why
  - How we order values and why

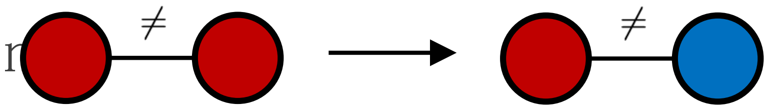
Iterative Improvement



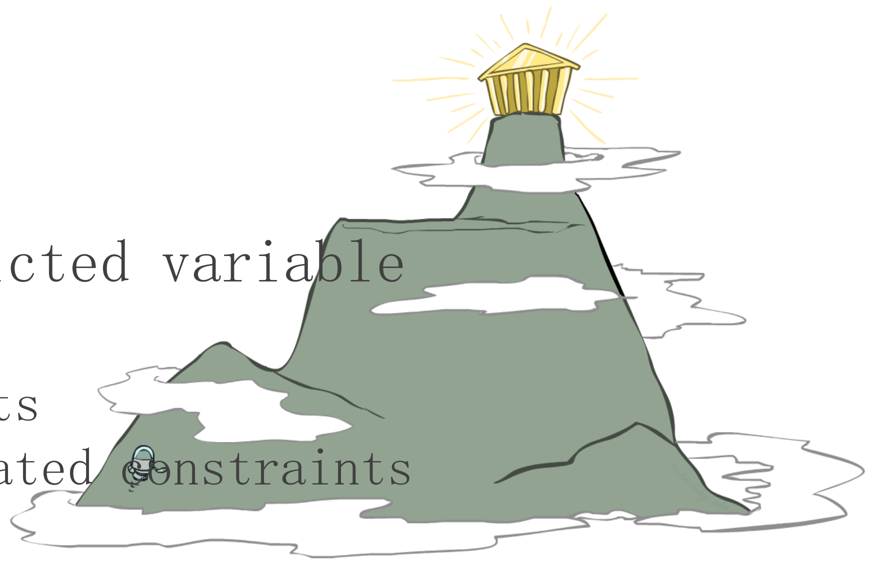
Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned

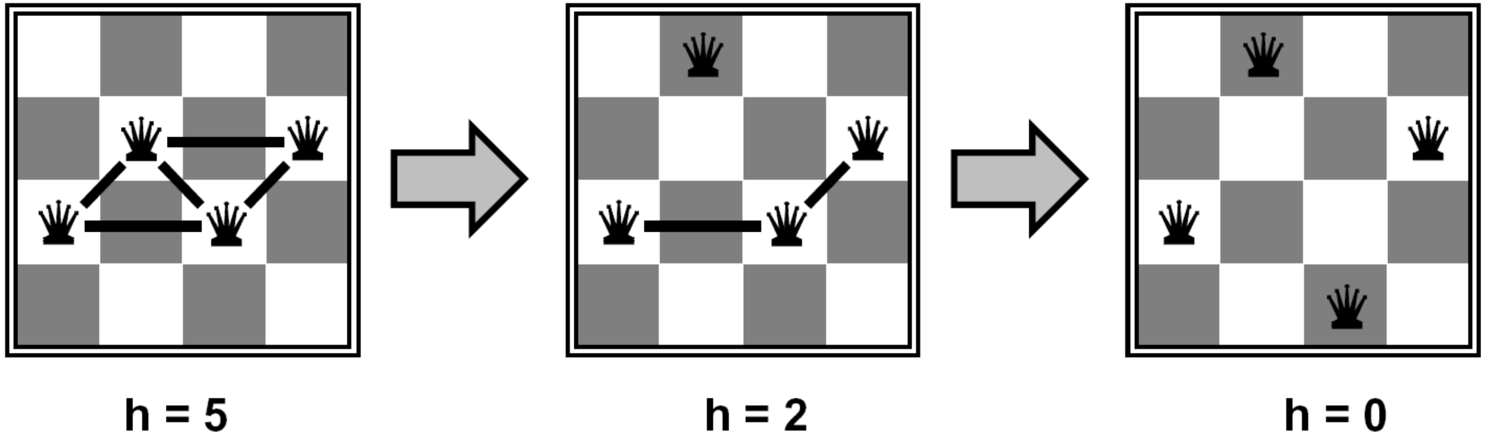
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(x) = total number of violated constraints



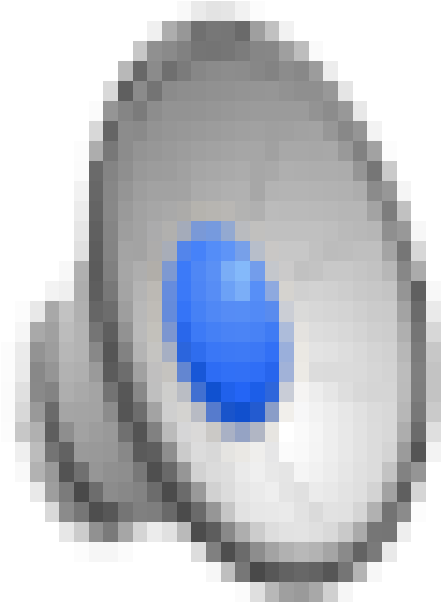
Example: 4-Queens



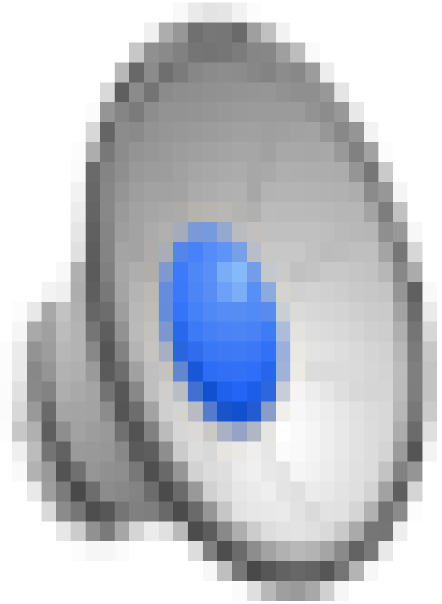
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

Video of Demo Iterative Improvement – n Queens



Video of Demo Iterative Improvement – Coloring

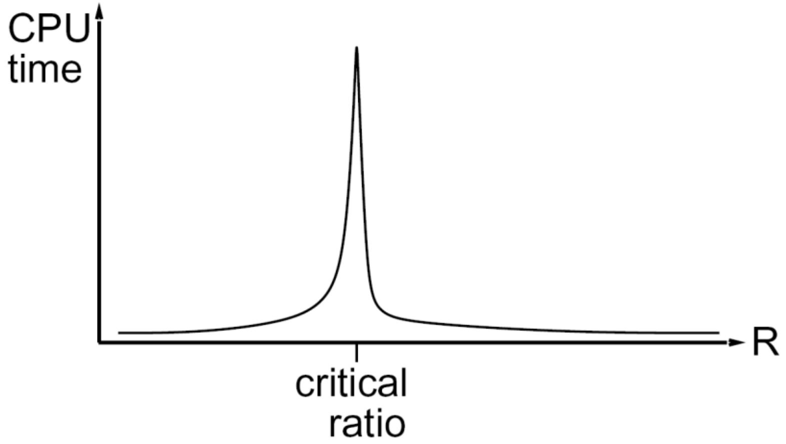


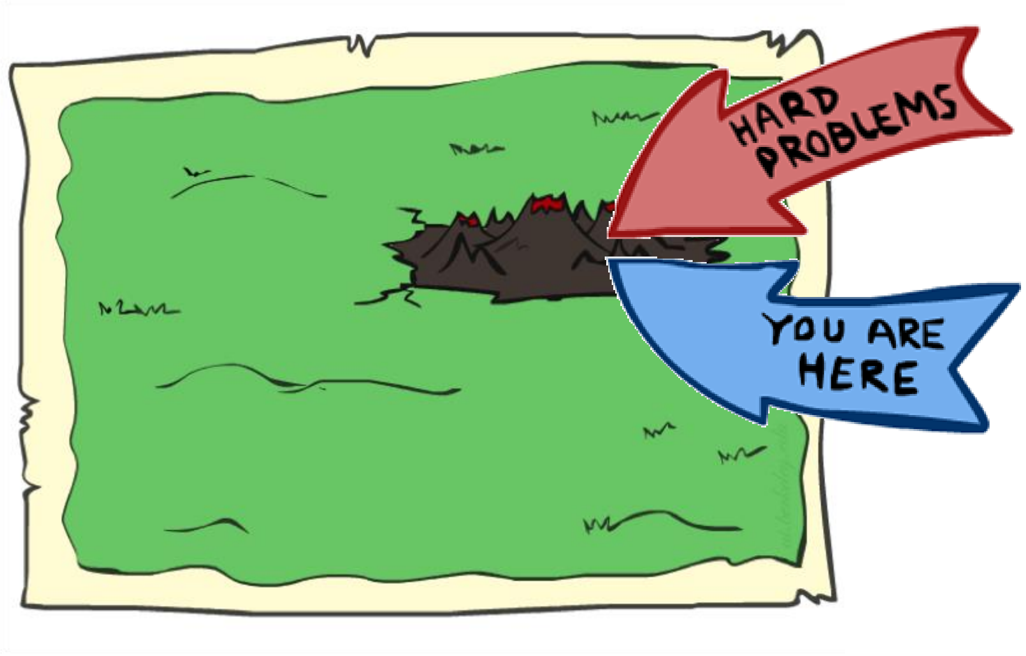
Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!

- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

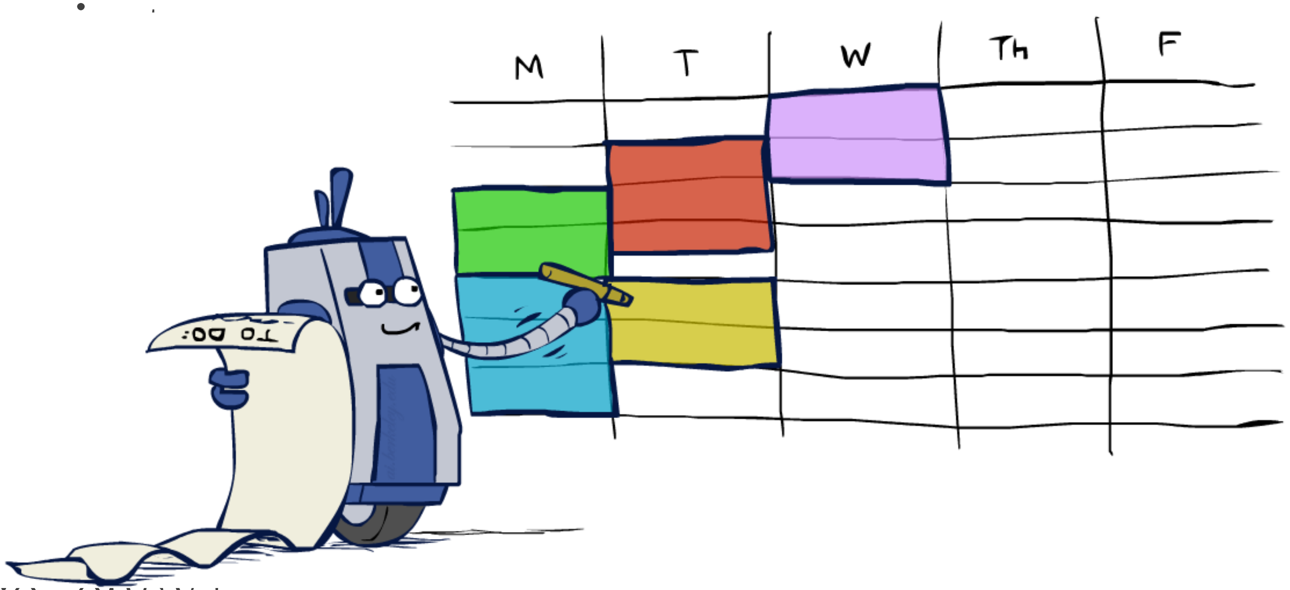
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by const
- Basic solution: backtracking
- Speed-ups:
  - Ordering
  - Filtering
  - Structure – turns out trees are easy!
- Iterative min-conflicts is often effective in practice



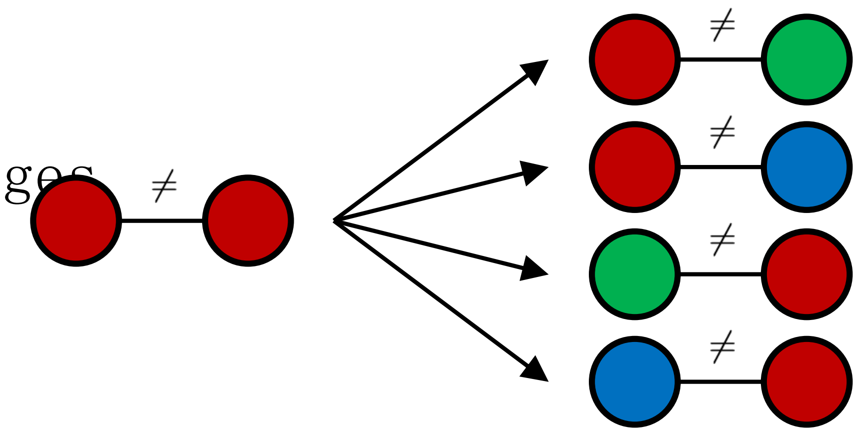
Local Search

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)

- Local search: improve a single option until you can't make it better (no fringe!)

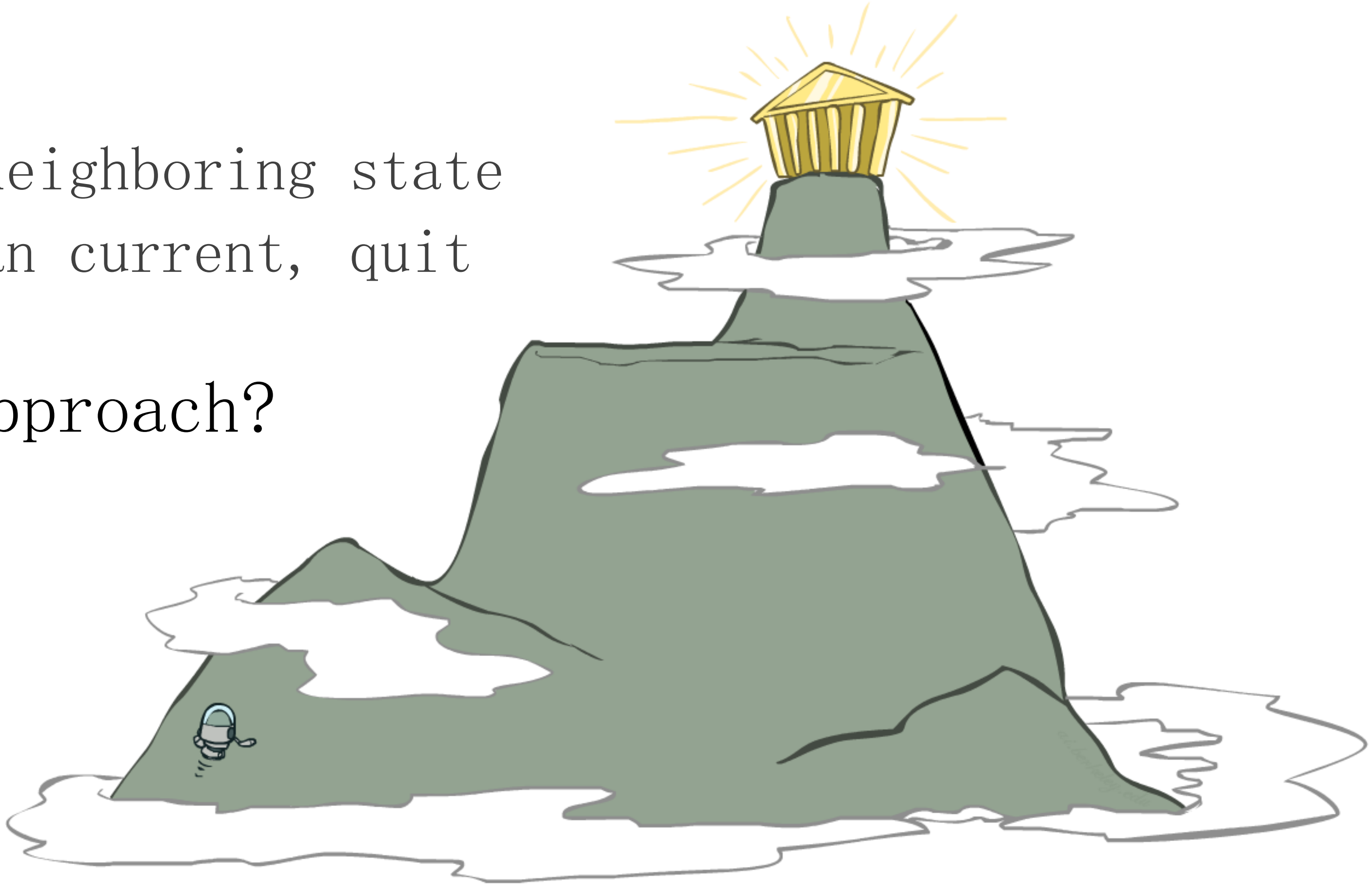
- New successor function: local changes



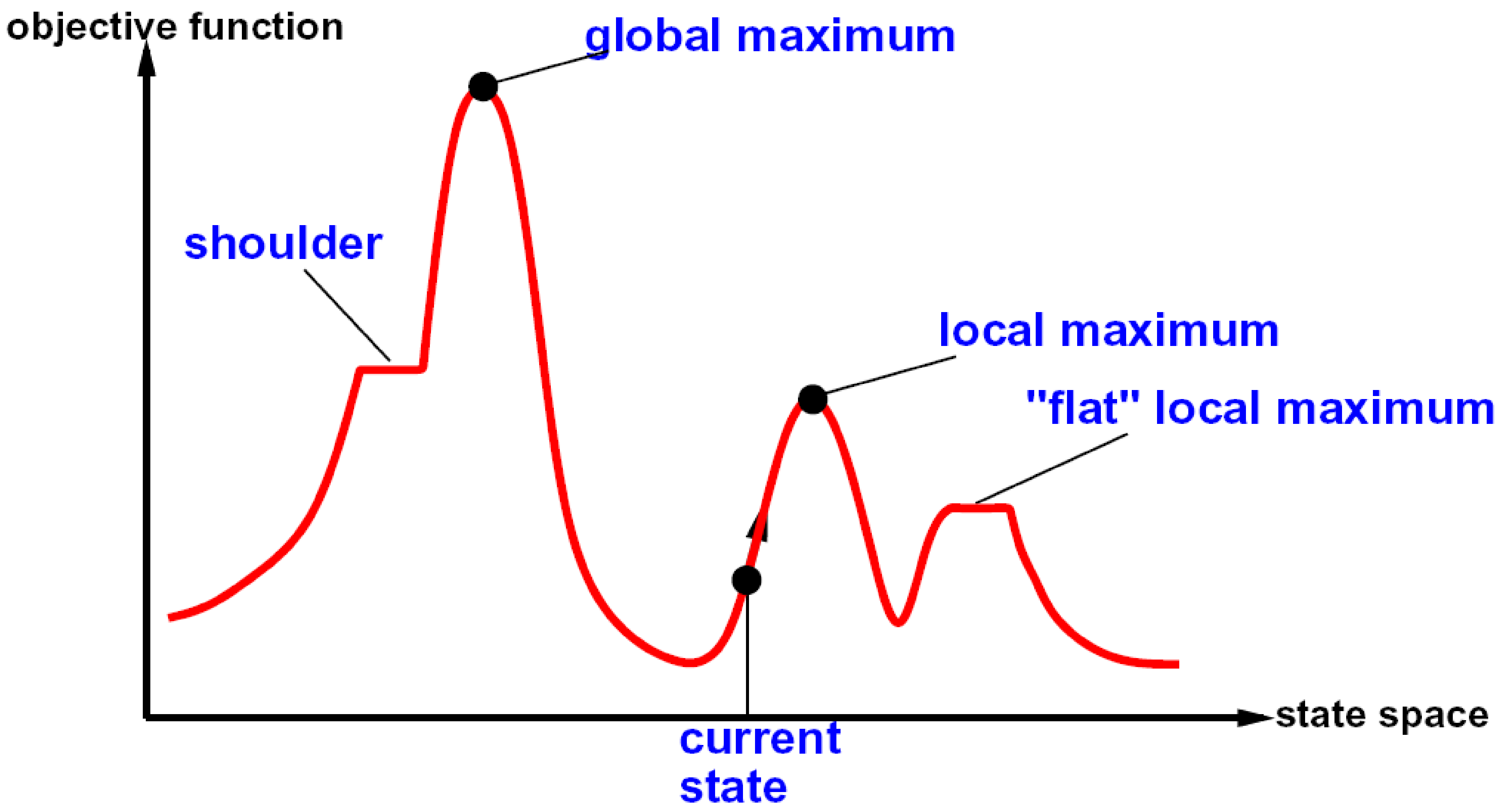
- Generally much faster and more memory efficient (but incomplete and

Hill Climbing

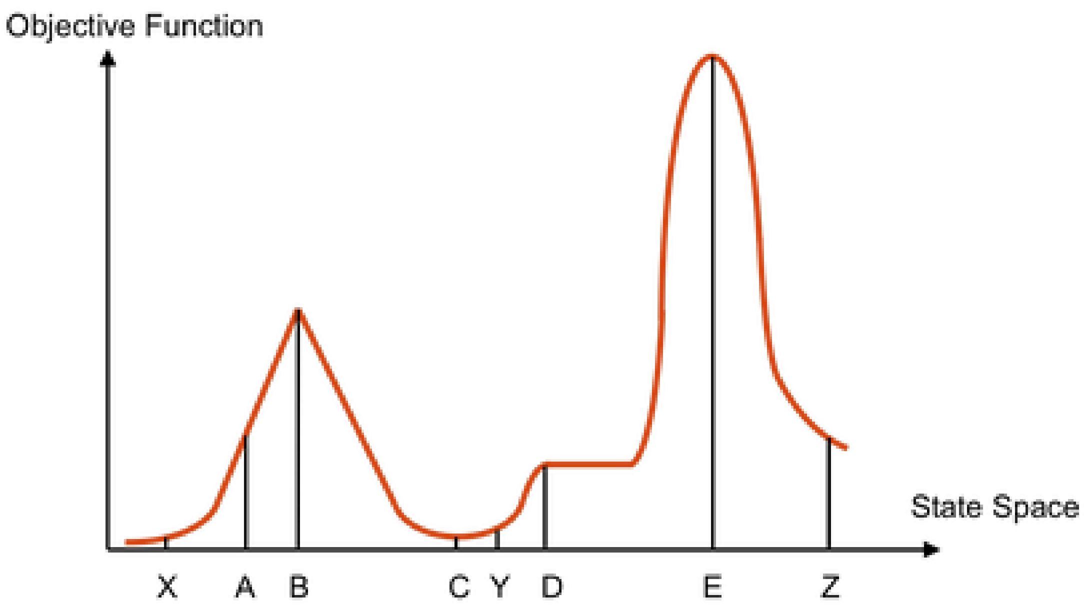
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What’s bad about this approach?
- What’s good about it?



Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

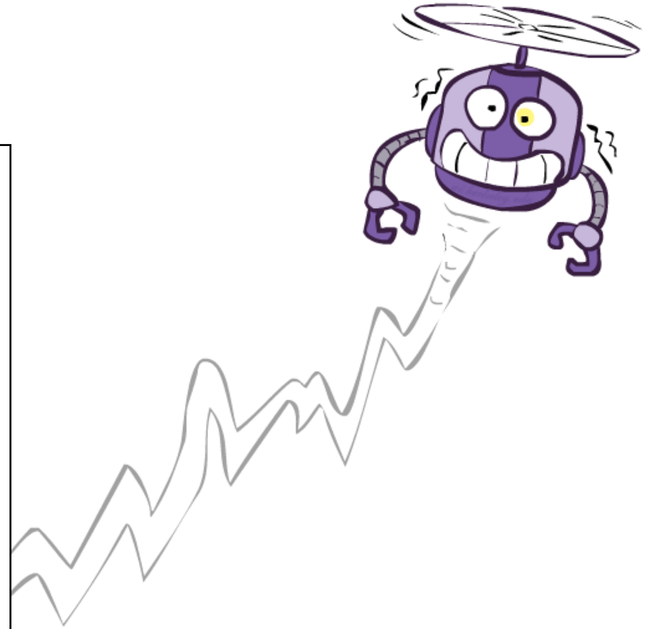
Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(ΔE/T)
```

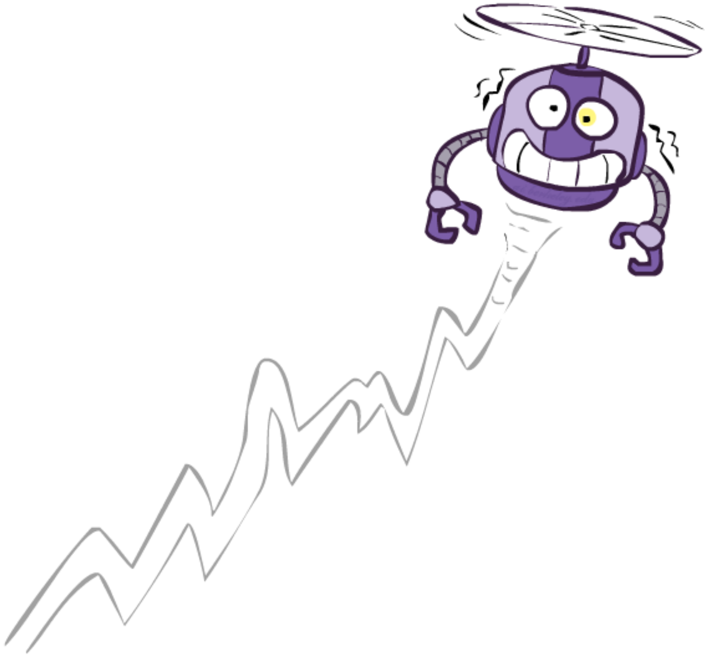


Simulated Annealing

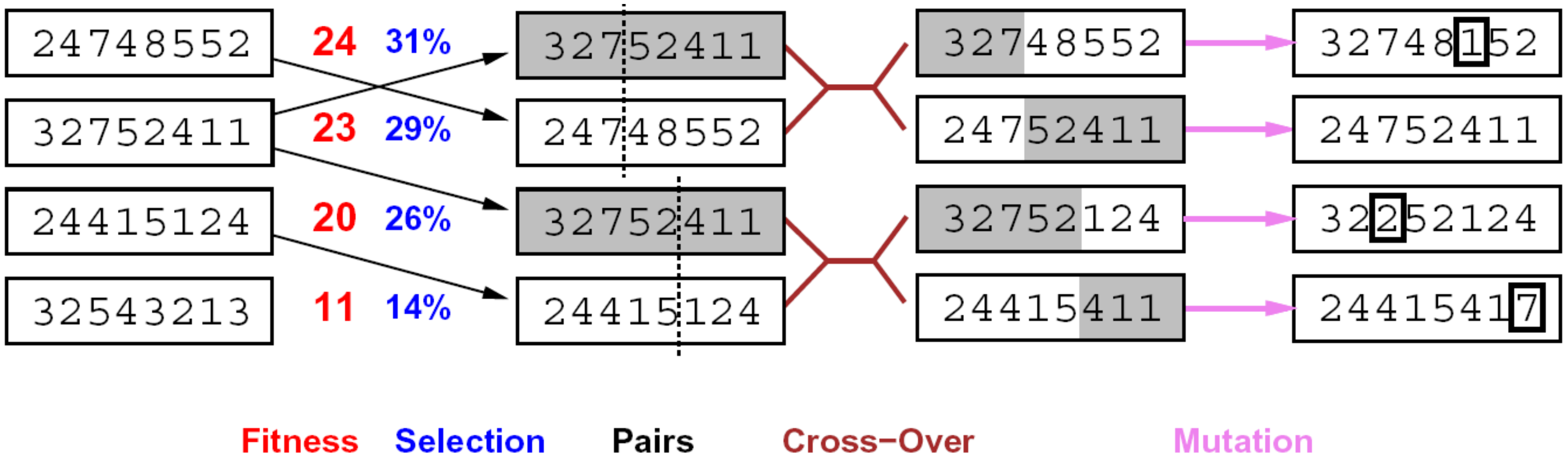
- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!

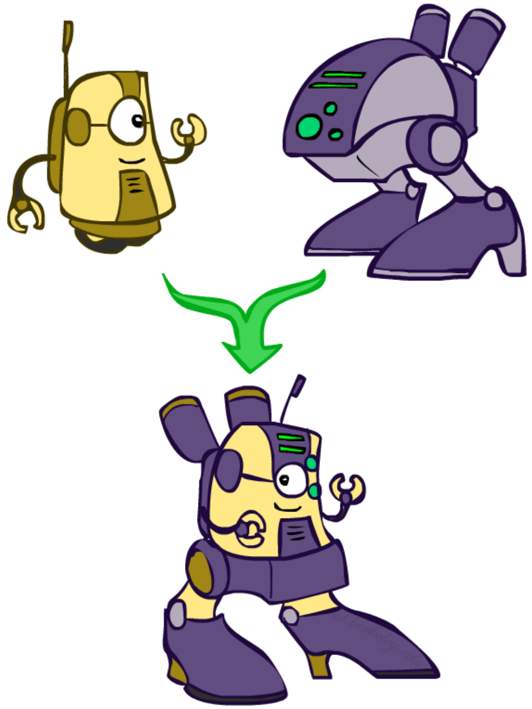
- Is this an interesting guarantee?

- Sounds like magic, but reality is reality
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways



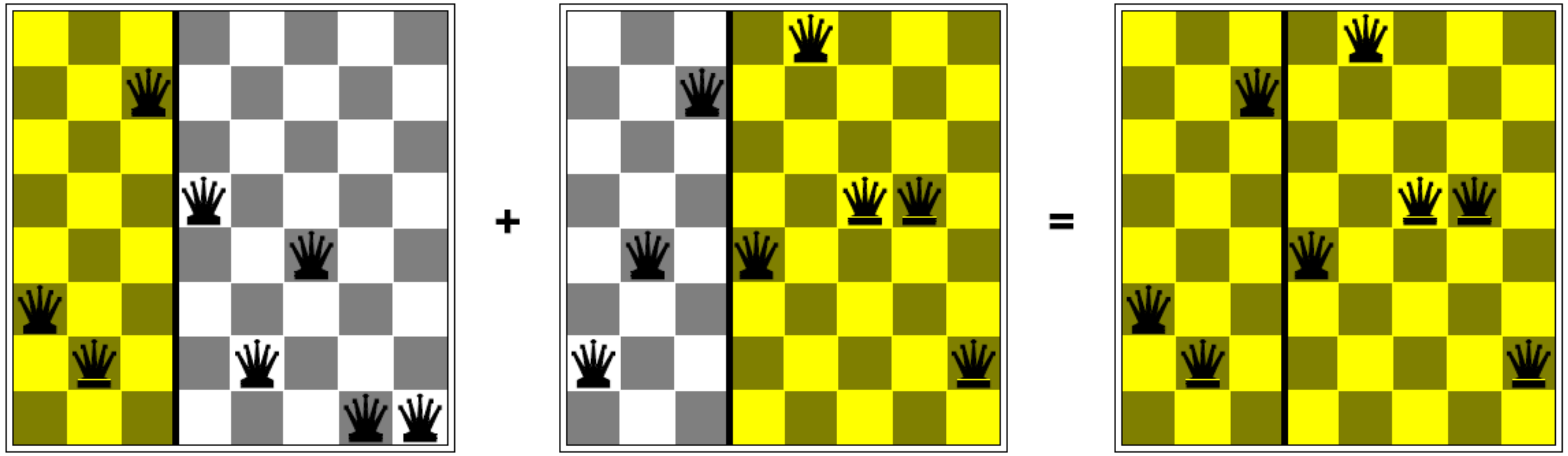
Genetic Algorithms





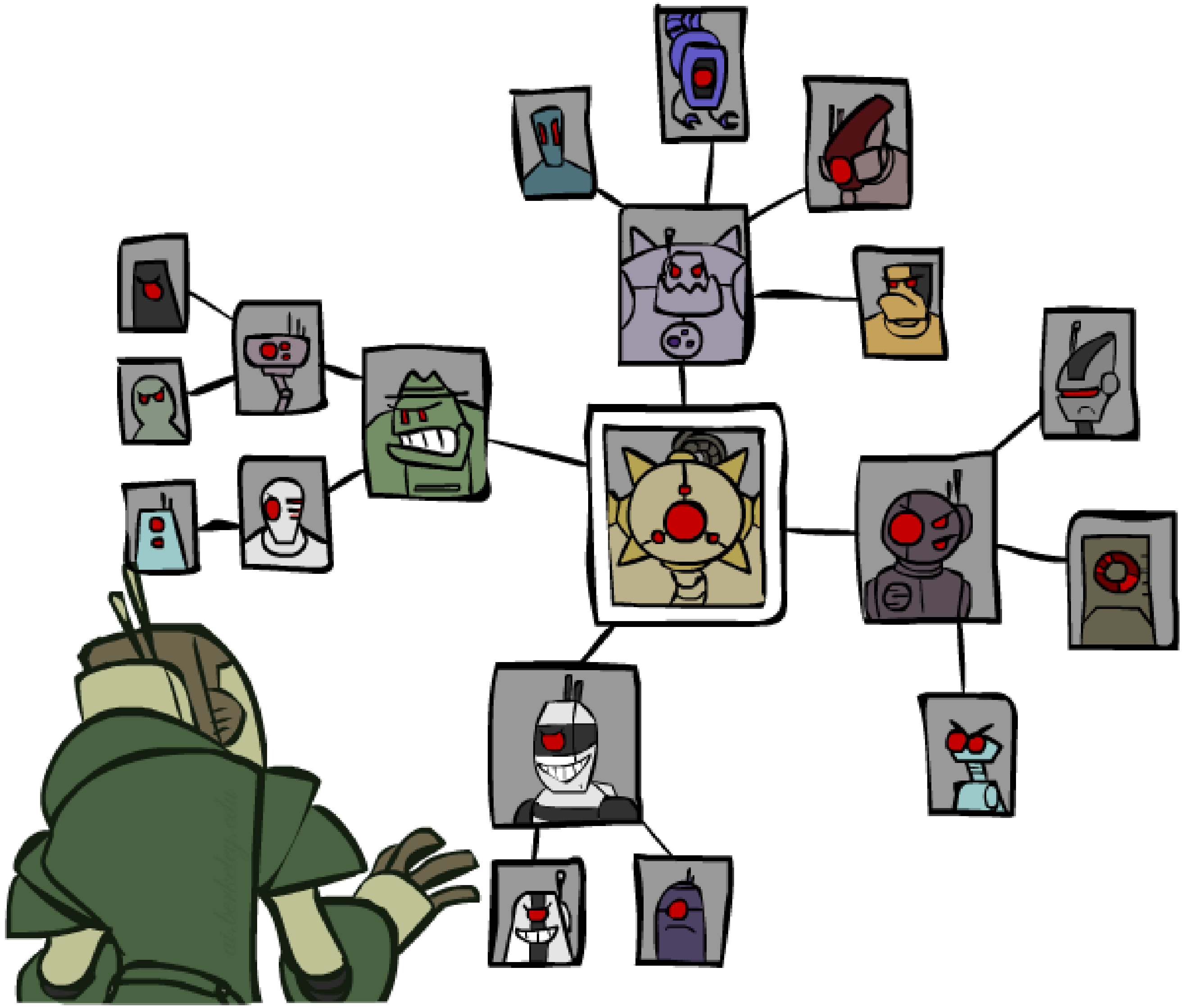
- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Example: N-Queens



- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?

Bonus (time permitting): Structure

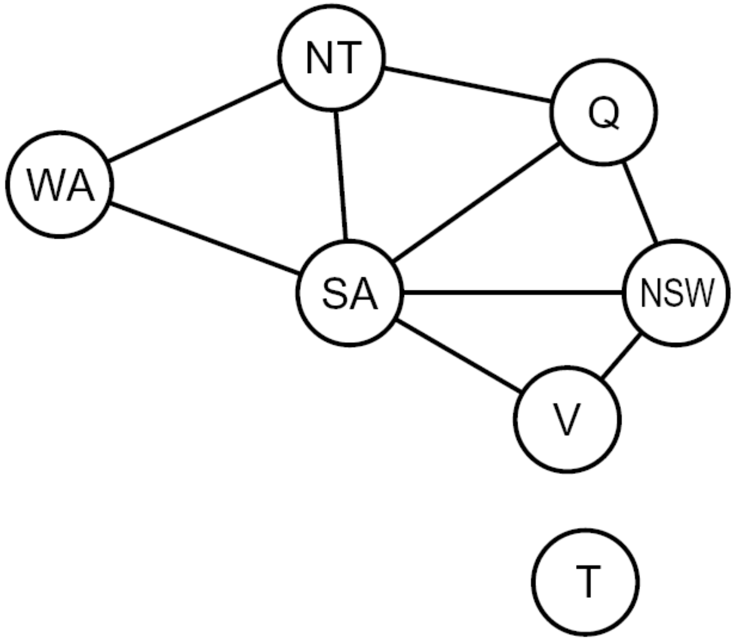


Problem Structure

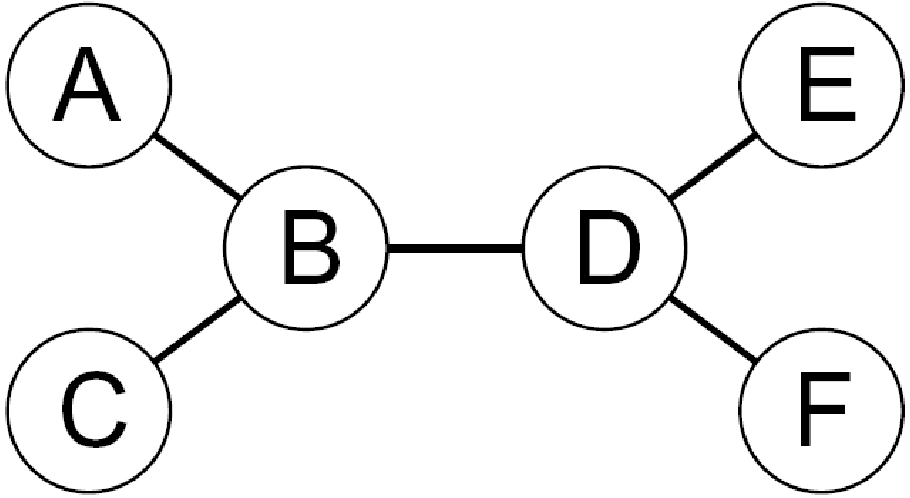
- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact

- Independent subproblems are identifiable as connected components of constraint graph

- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



Tree-Structured CSPs

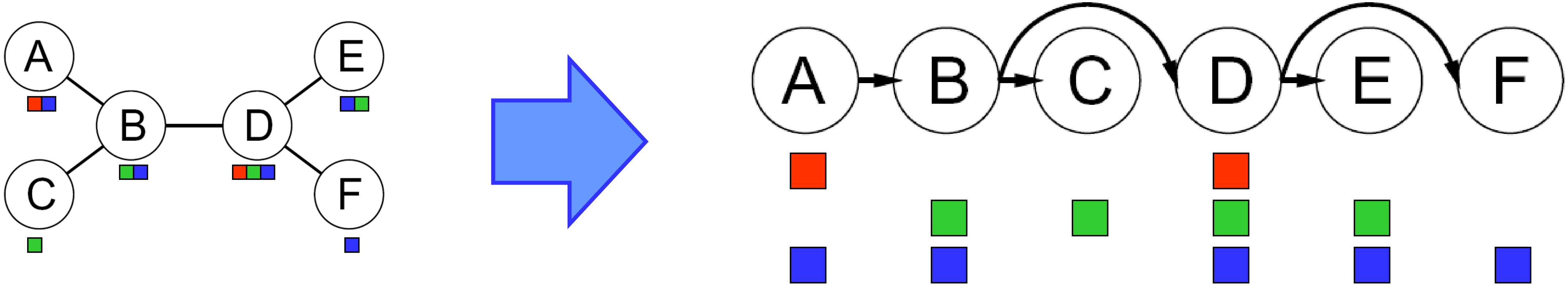


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children

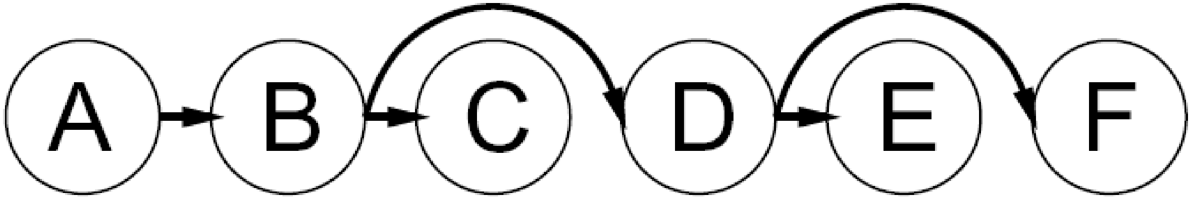


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consisten
    Parent(X_i)

- Runtime: $O(n\ d^2)$ (why?)

Tree-Structured CSPs

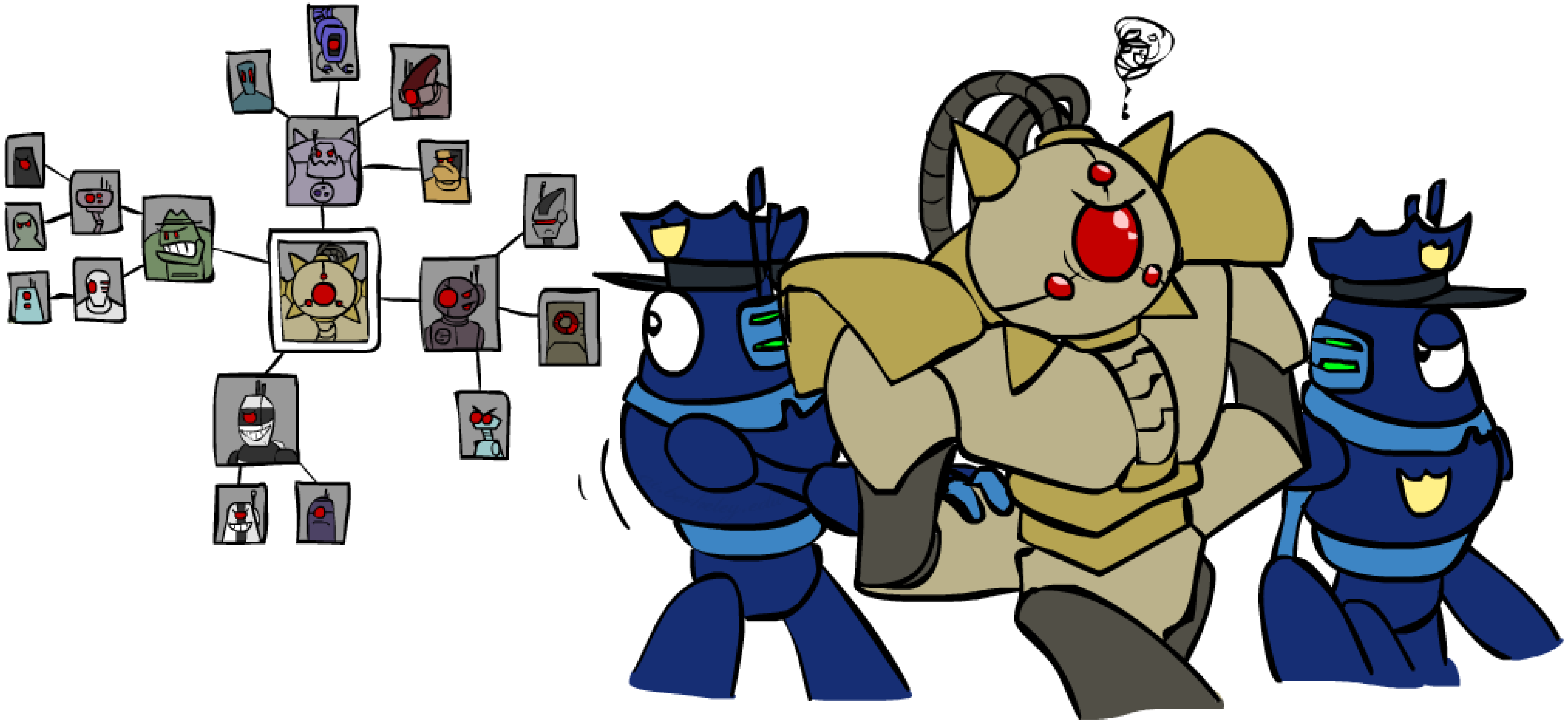
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y’ s domain could not have been reduced thereafter (because Y’ s children were processed before Y)



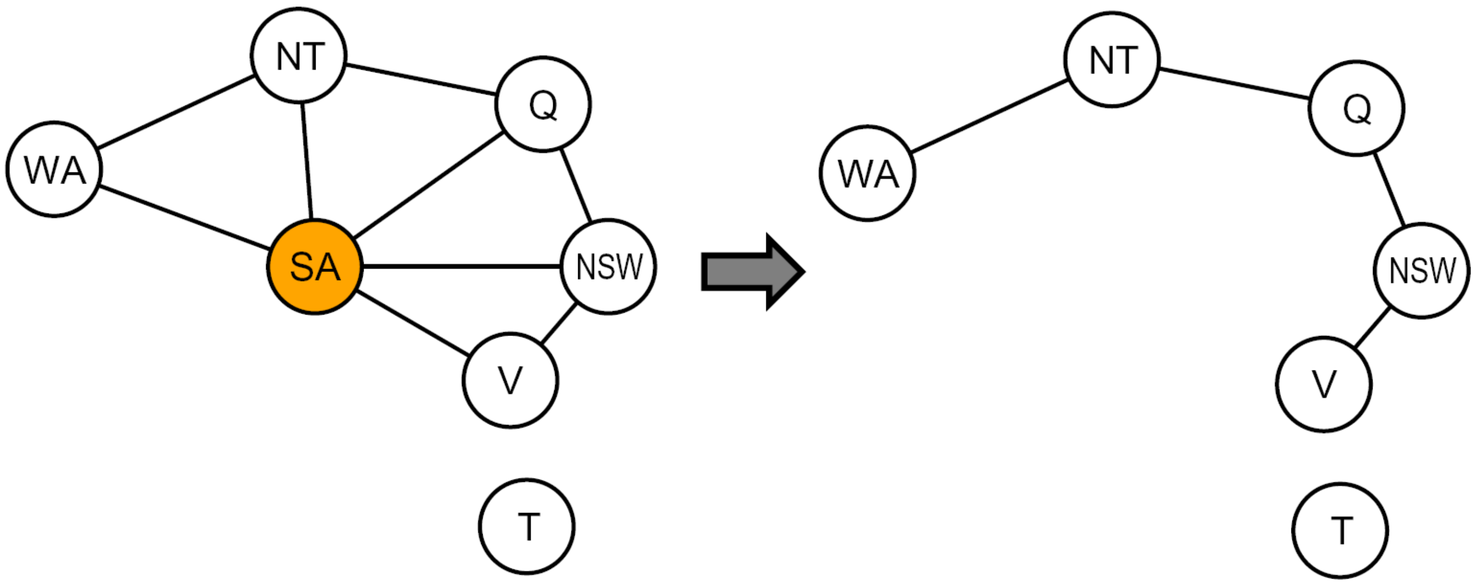
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

- Why doesn’ t this algorithm work with cycles in the constraint graph?

Improving Structure



Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains

- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

- Cutset size c gives runtime O((d^c) $(n-c)$ d^2), very fast for

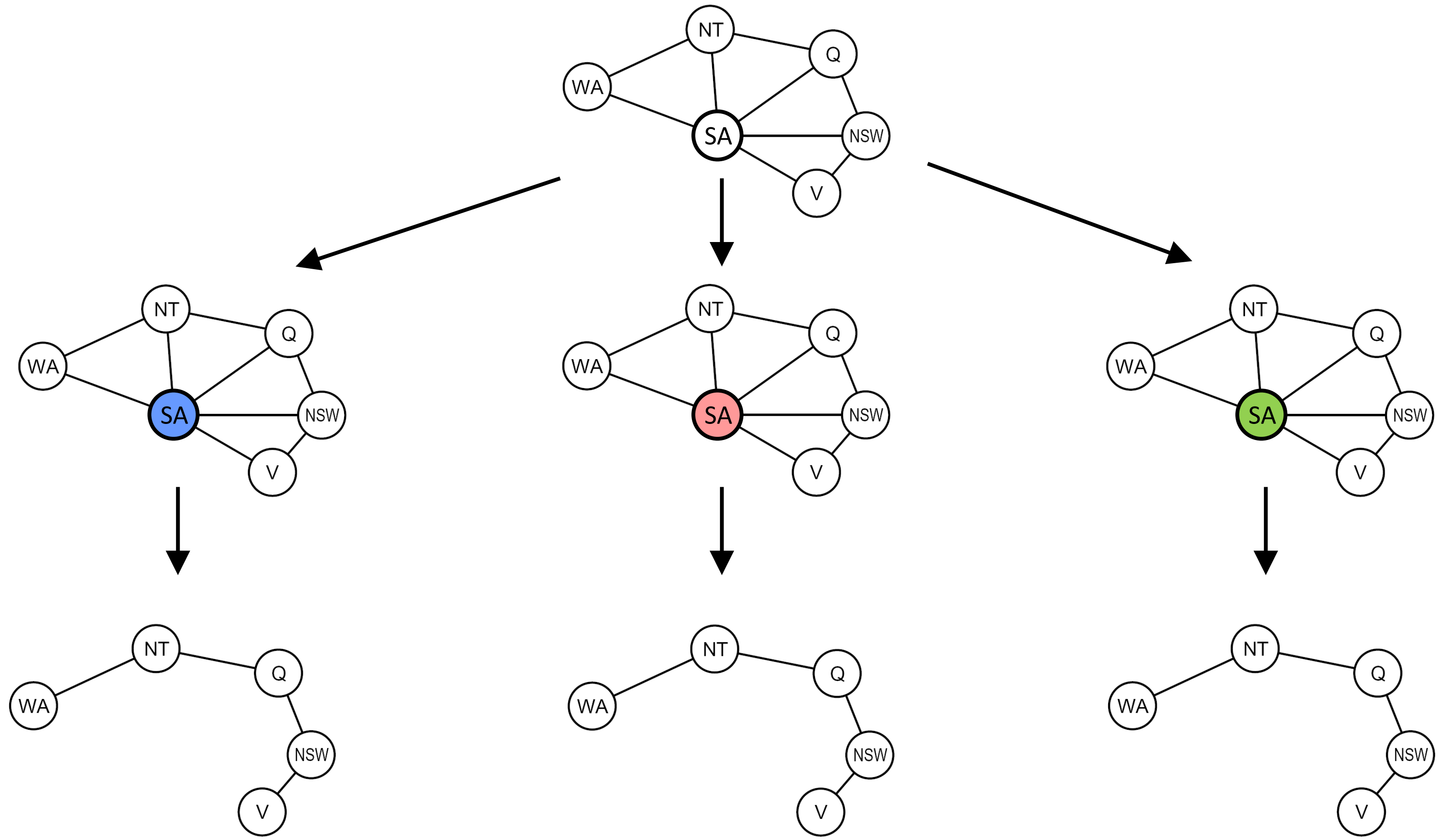
Cutset Conditioning

Choose a cutset

Instantiate the cutset
(all possible ways)

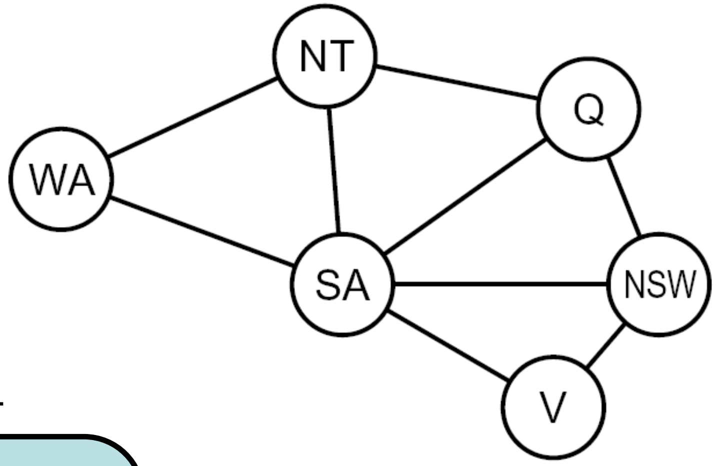
Compute residual CSP
for each assignment

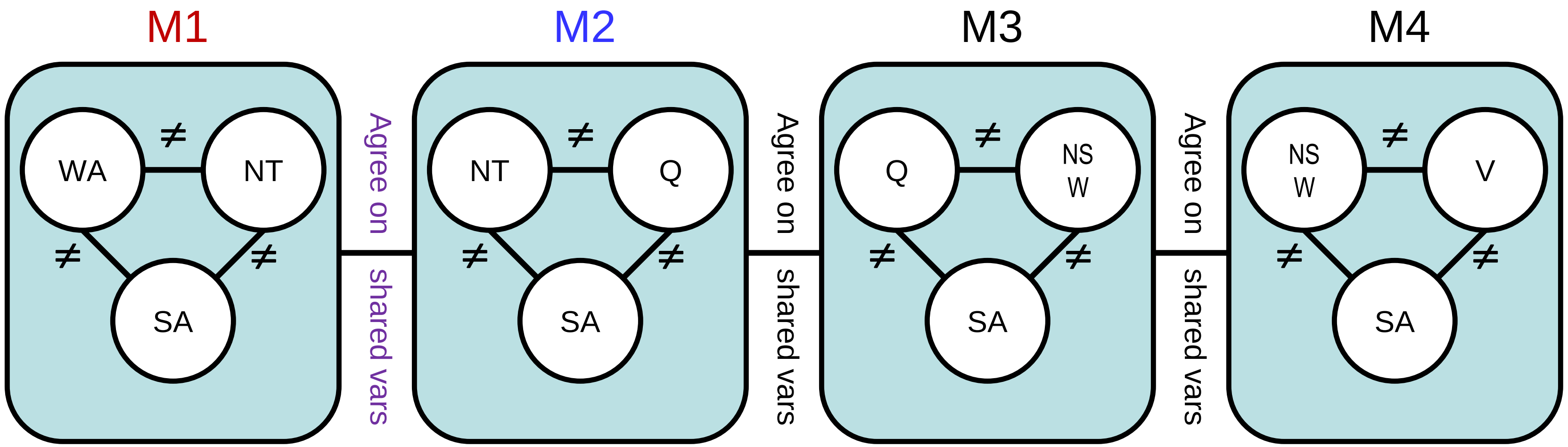
Solve the residual CSPs
(tree structured)



Tree Decomposition*

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions





{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
…}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
…}

Agree: (M1,M2) ∈
{((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), …}

Next Time:

Search when you’ re not the only agent!