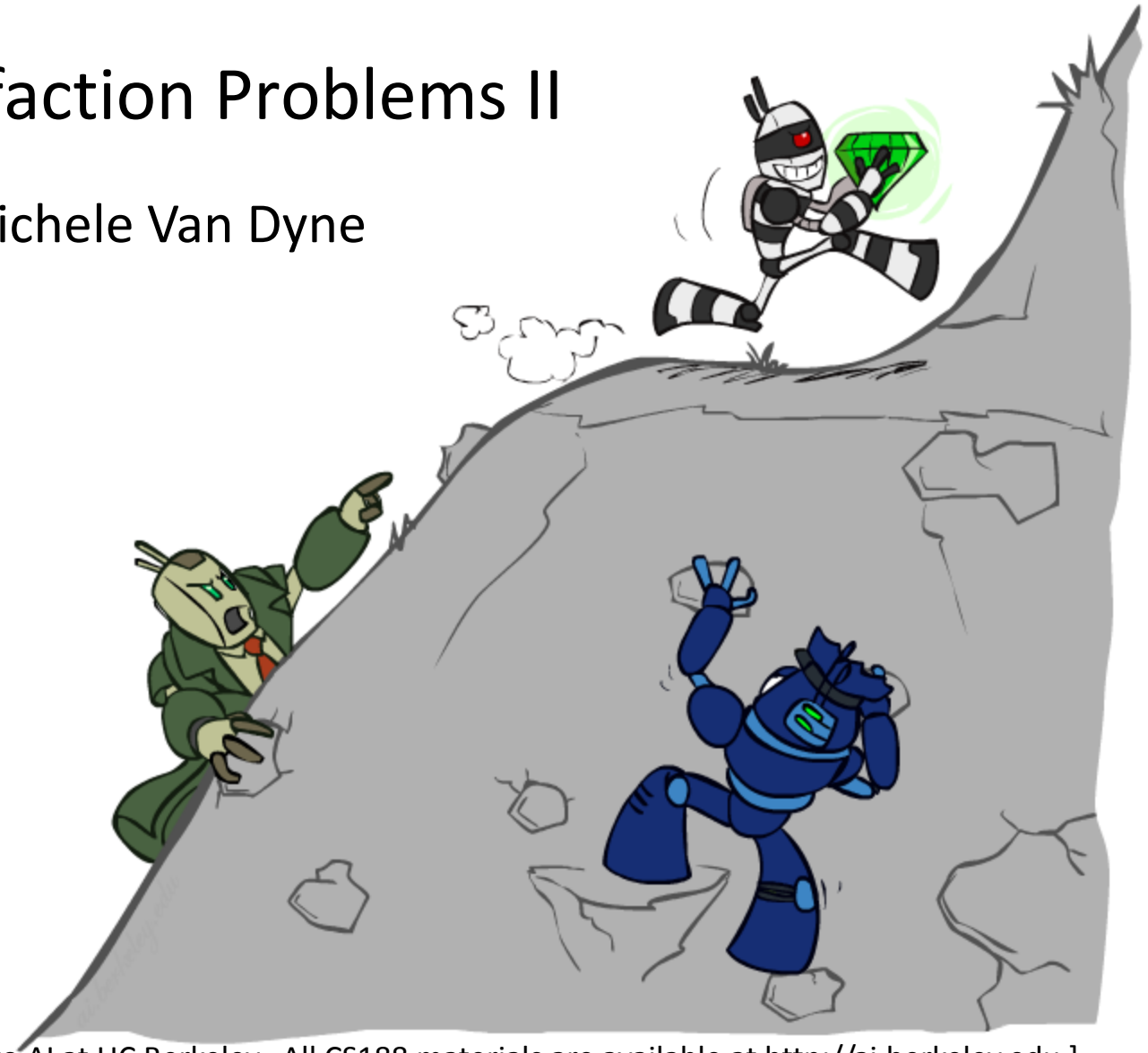# CSCI 446: Artificial Intelligence

Constraint Satisfaction Problems II

Instructor: Michele Van Dyne
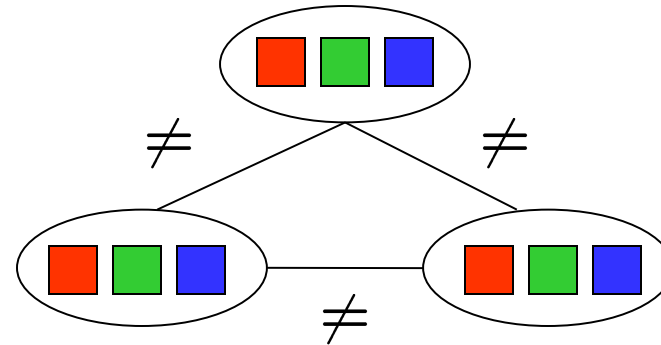
Today

- Efficient Solution of CSPs

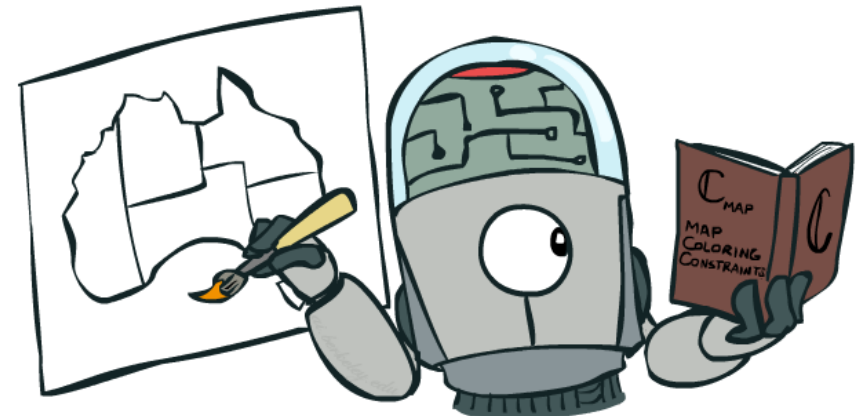- Local Search

Reminder: CSPs

- **CSPs:**
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary

- **Goals:**
  - Here: find any solution
  - Also: find all, find best, etc.

Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```
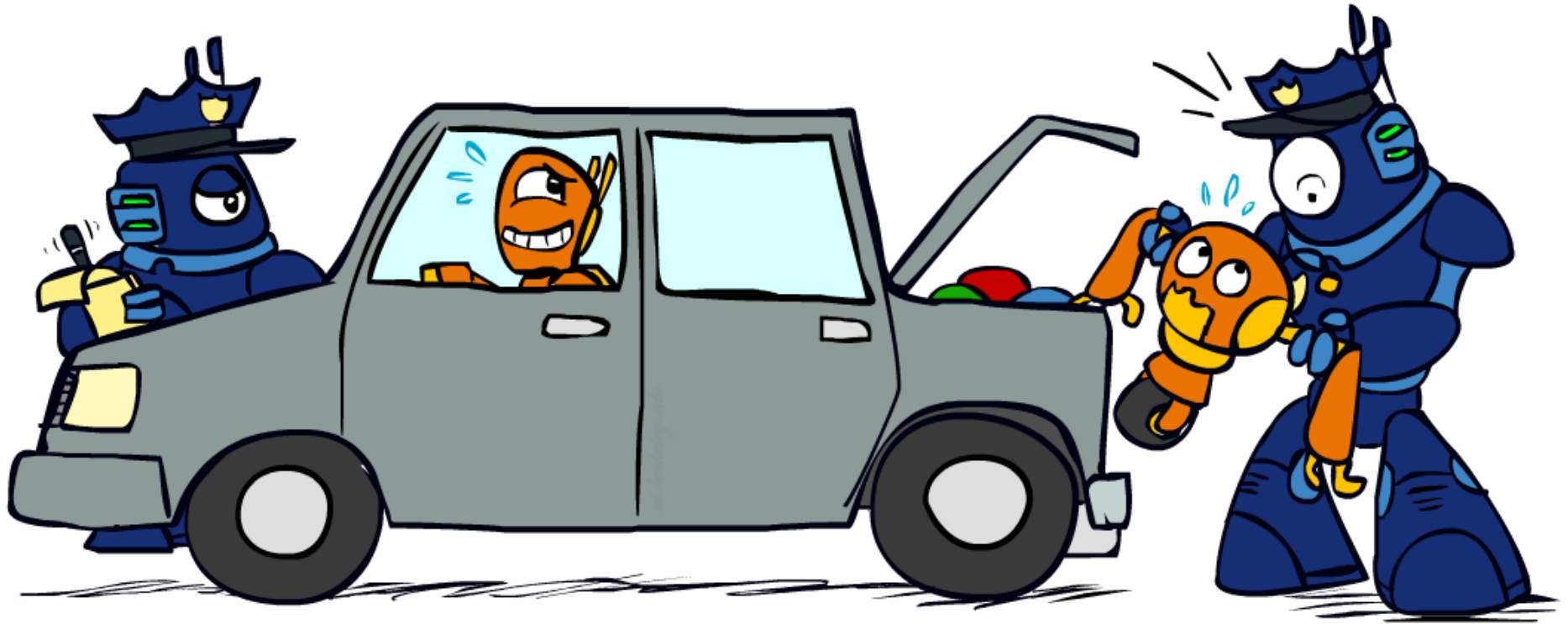
Improving Backtracking

- General-purpose ideas give huge gains in speed
  - … but it's all still NP-hard

- Filtering: Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next? (MRV)
  - In what order should its values be tried? (LCV)

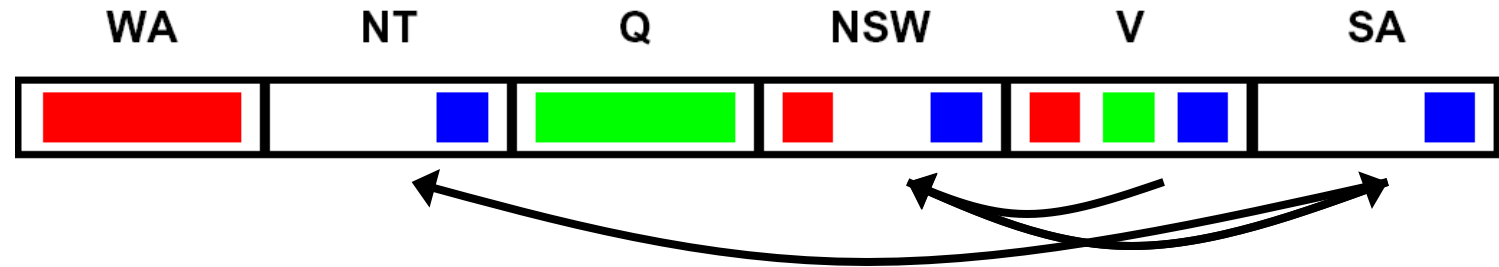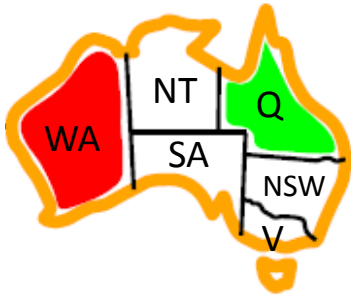- Structure: Can we exploit the problem structure?

Arc Consistency and Beyond

Arc Consistency of an Entire CSP

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


- Arc consistency detects failure earlier than forward checking
- Important: If X loses a value, neighbors of X need to be rechecked!
- Must rerun after each assignment!
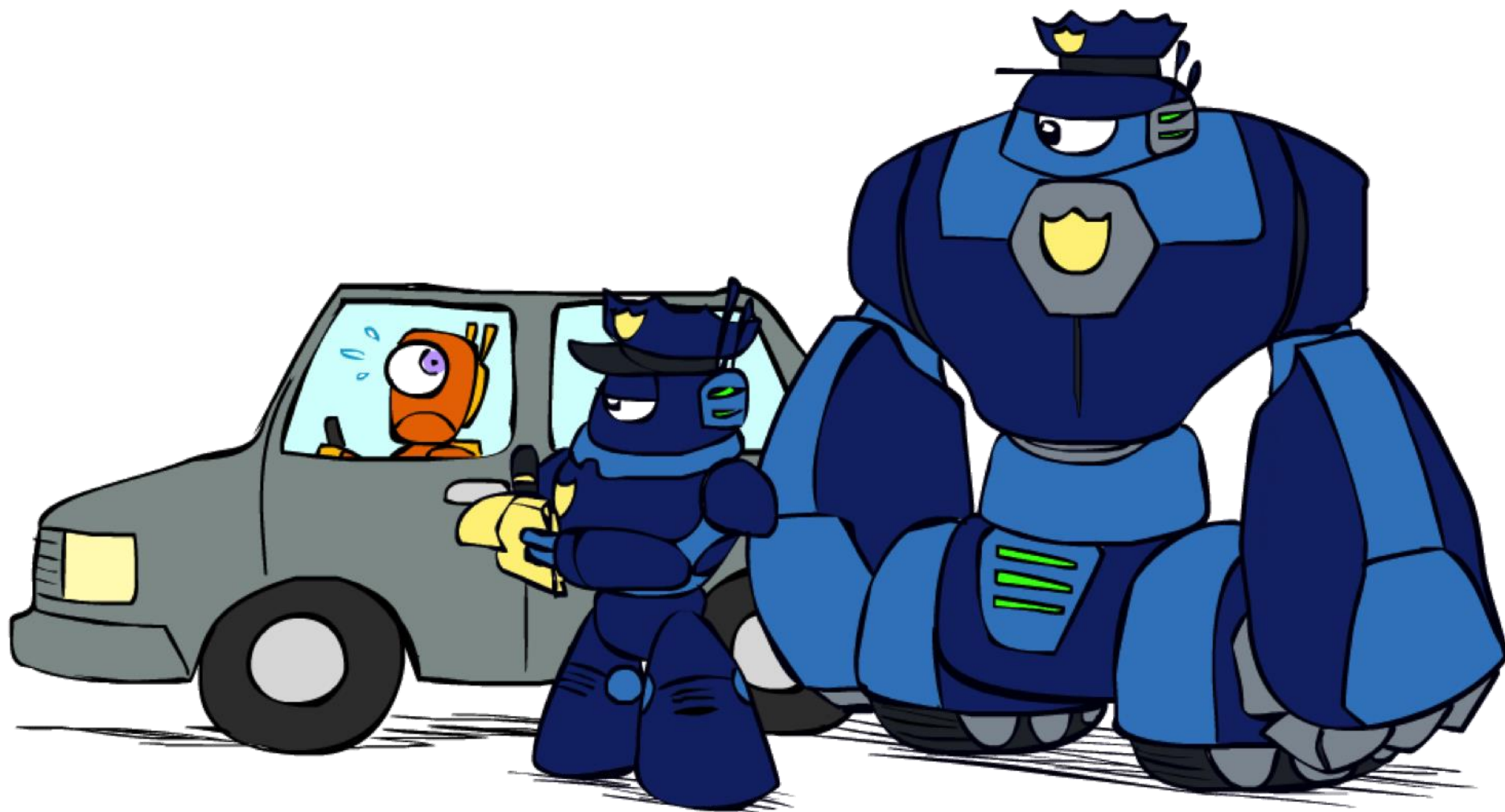
Remember: Delete from the tail!

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!
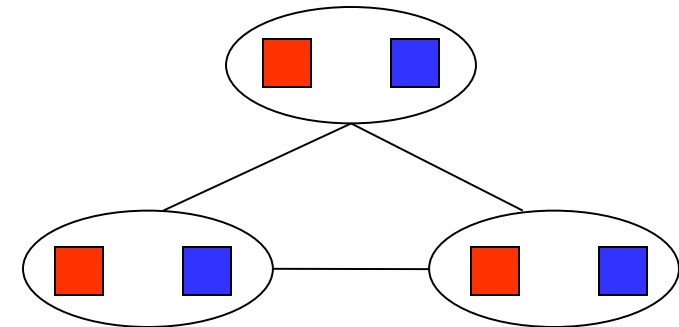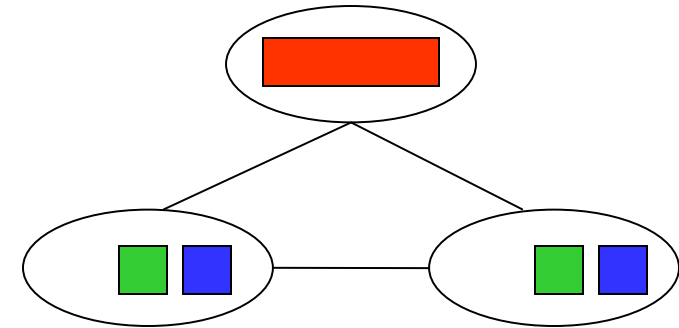
What went wrong here?

K-Consistency

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 kth node.

- **Higher k more expensive to compute**

- **(You need to know the k=2 case: arc consistency)**
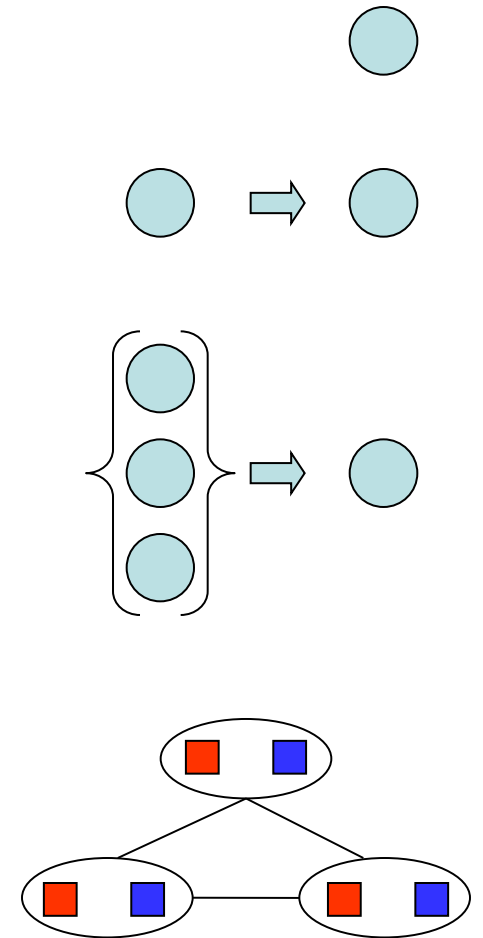
Strong K-Consistency

- 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)

Structure
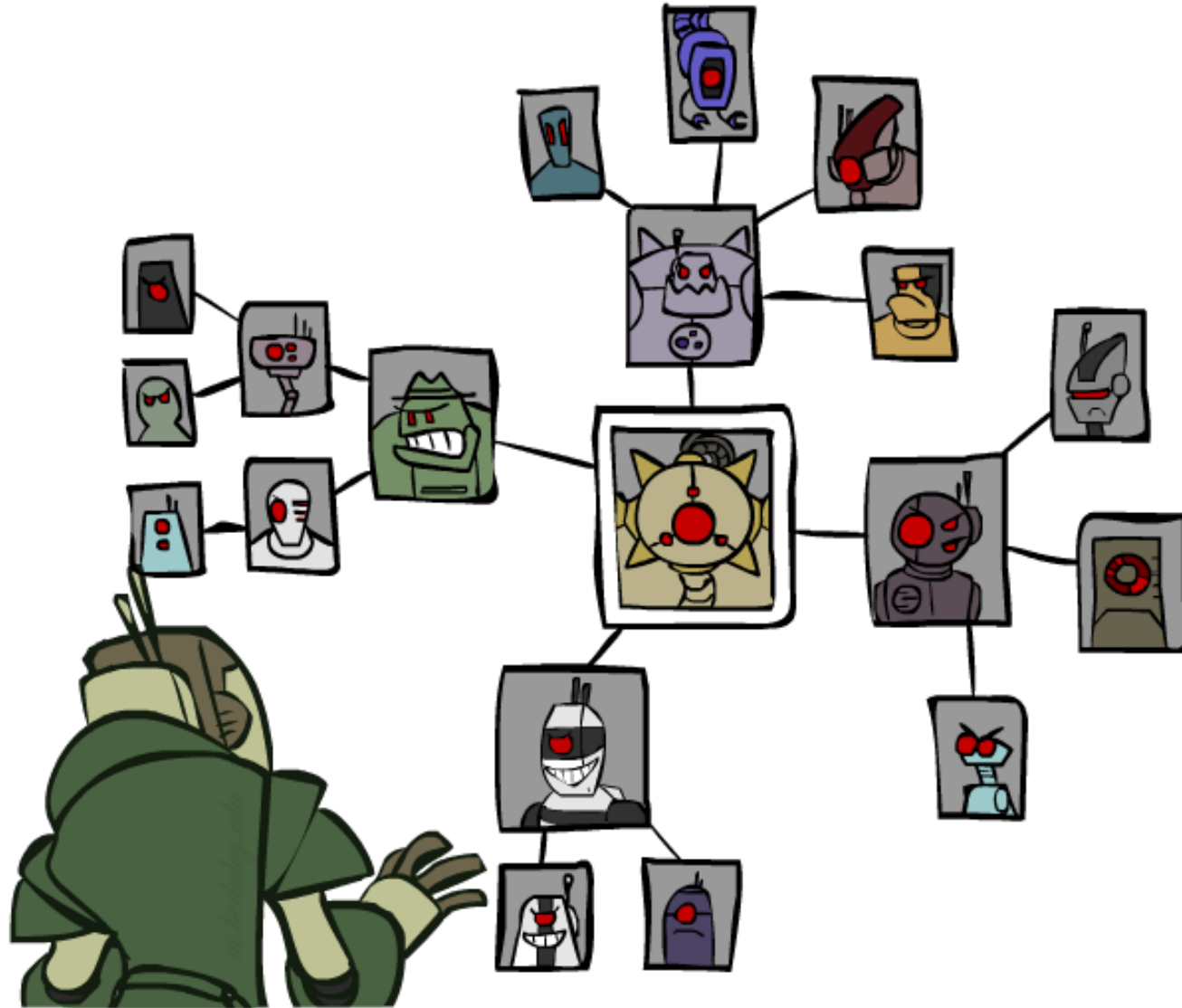
Problem Structure

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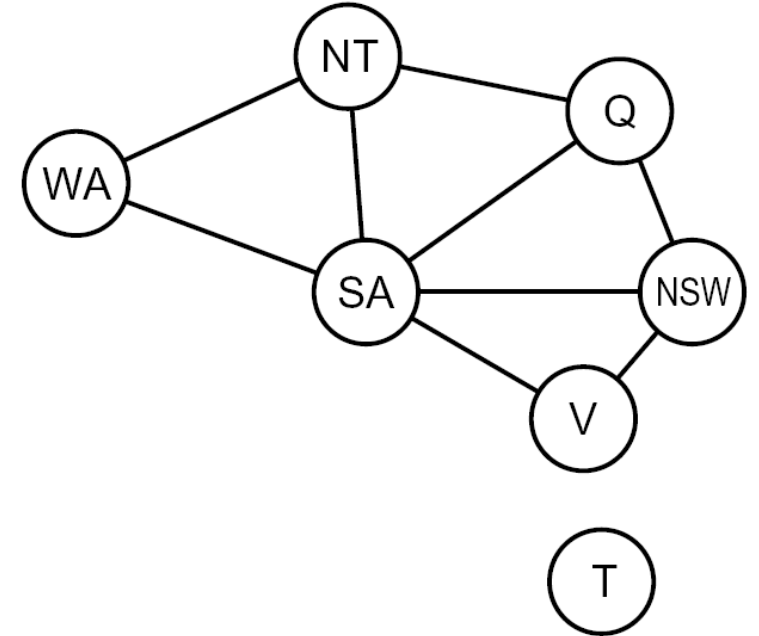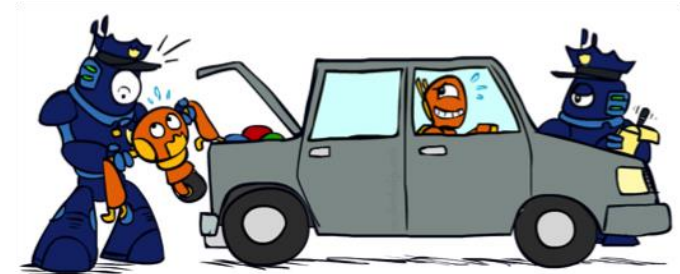
Tree-Structured CSPs
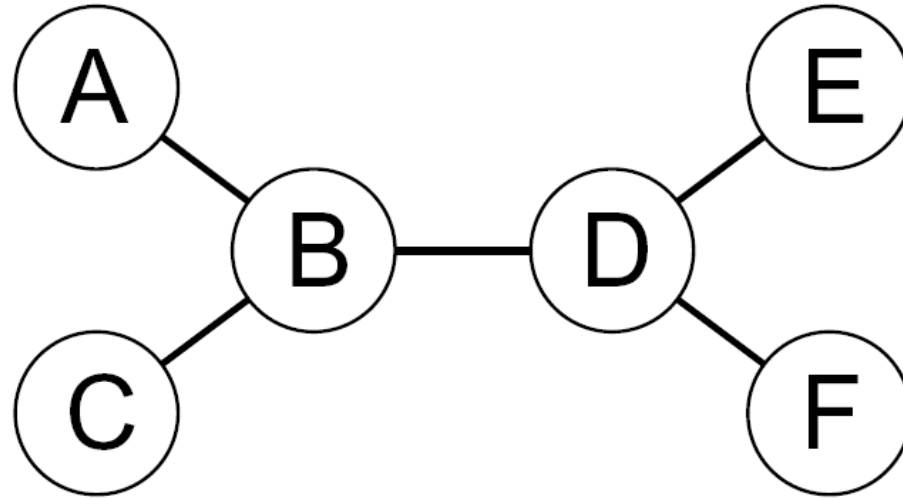
- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\, d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

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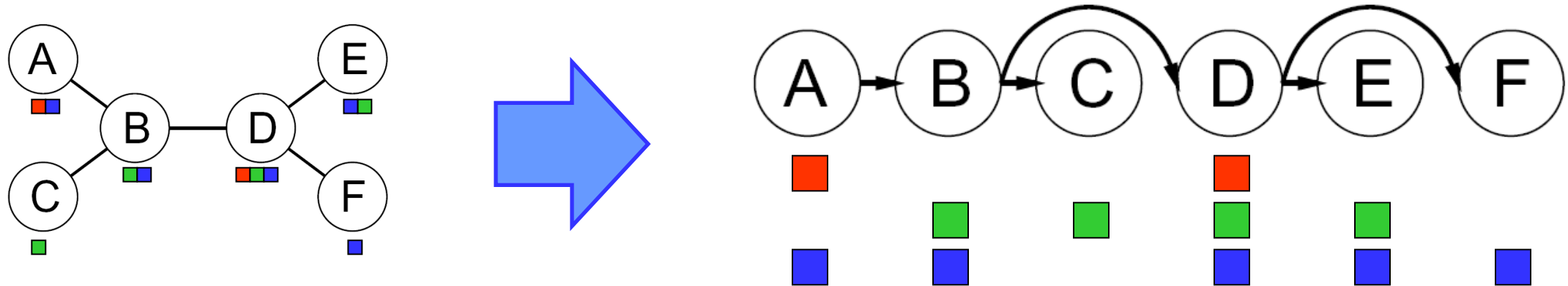
  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

- 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)
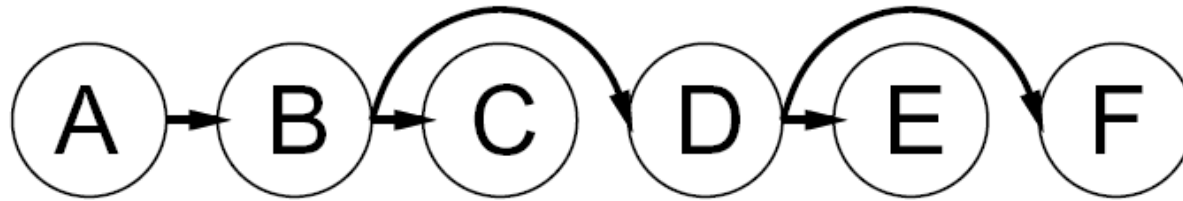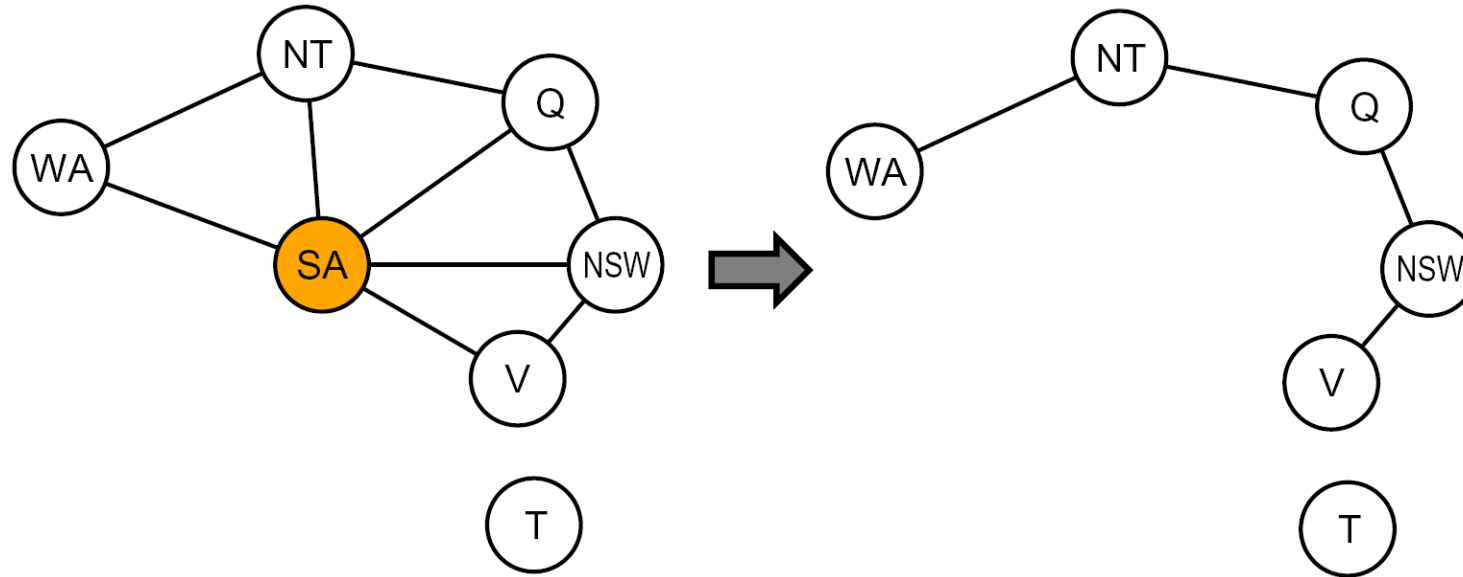

- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

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

- Note: we'll see this basic idea again with Bayes' nets

Improving Structure
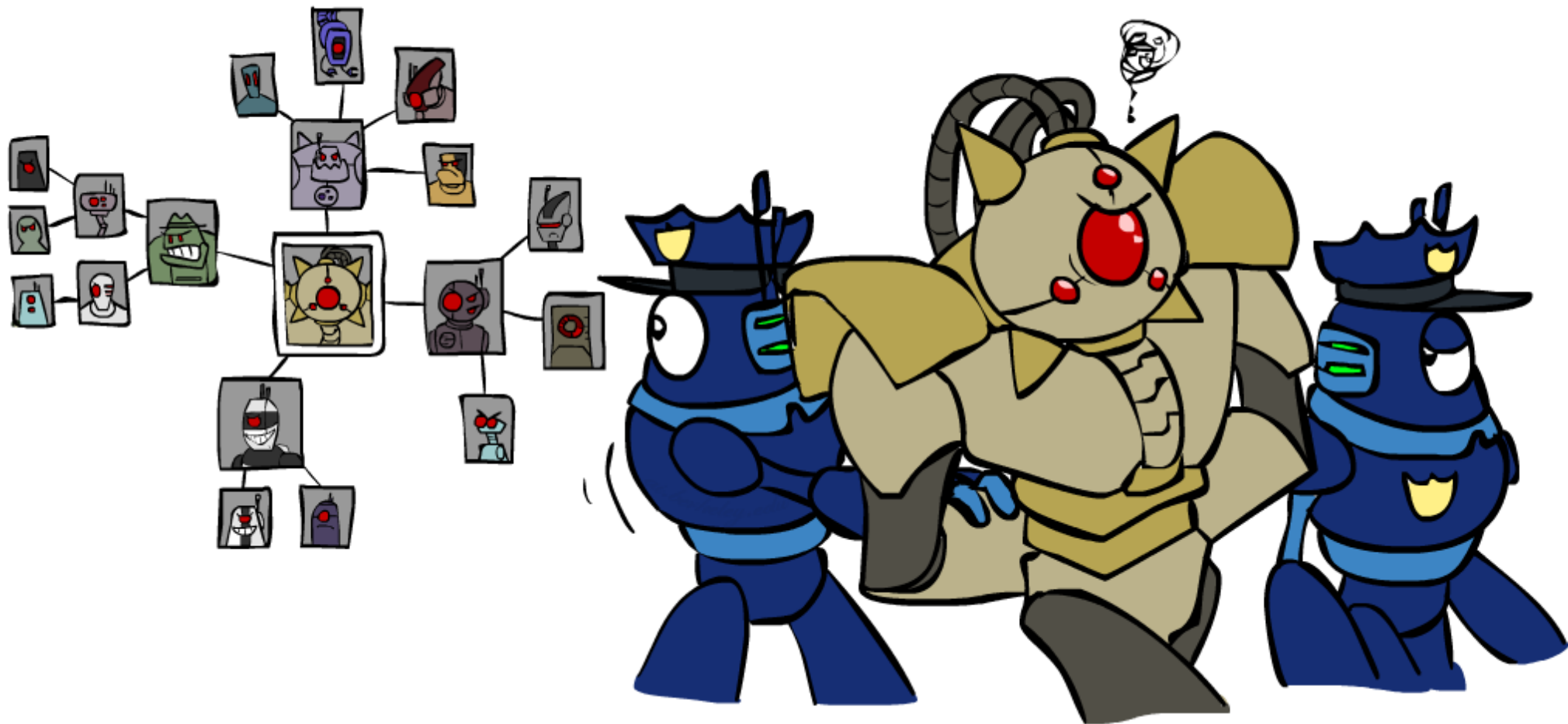
Nearly Tree-Structured CSPs

- 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

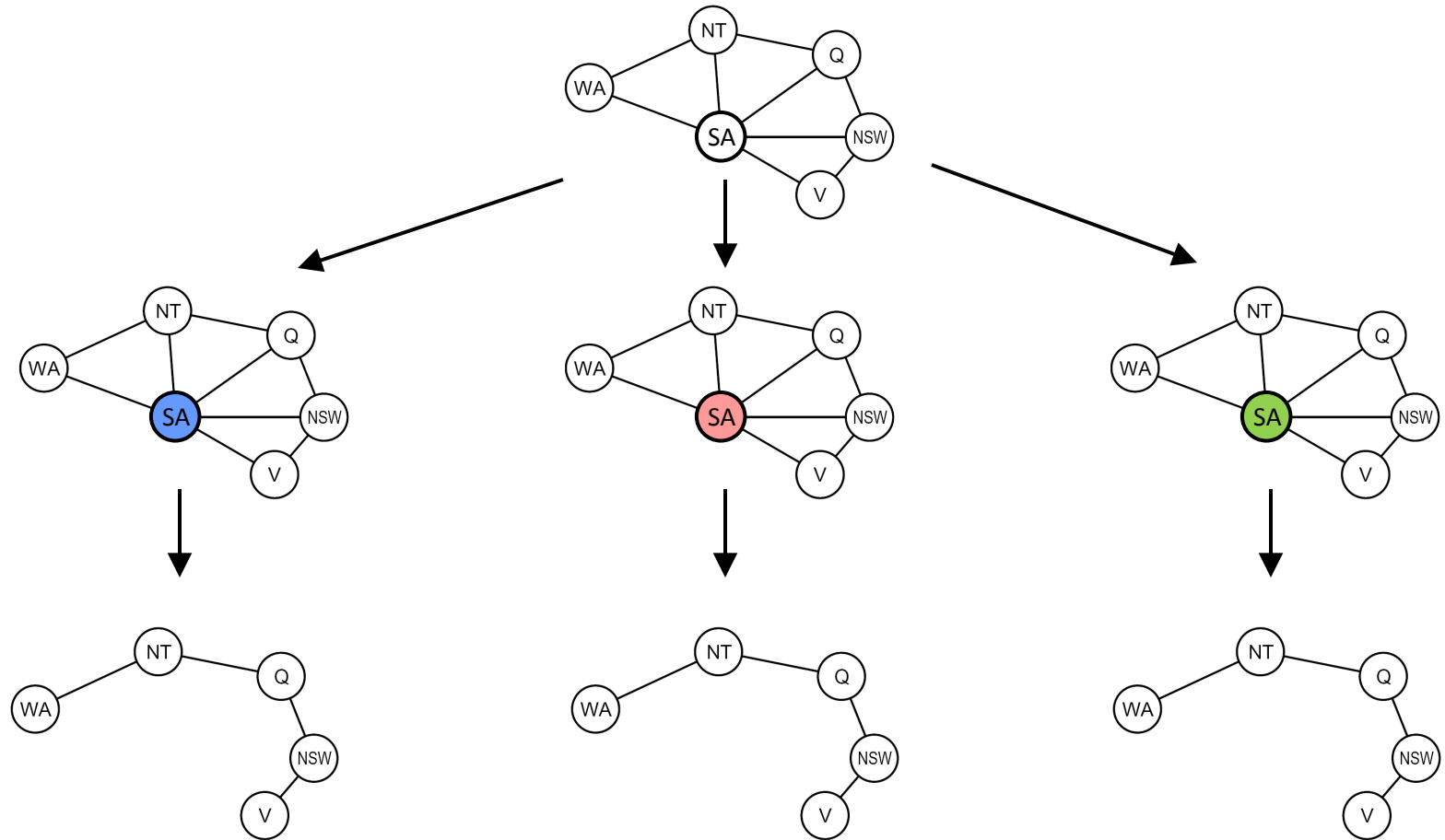- Cutset size c gives runtime $O(\ (d^c)\ (n-c)\ d^2\ )$, very fast for small c

Cutset Conditioning

Choose a cutset

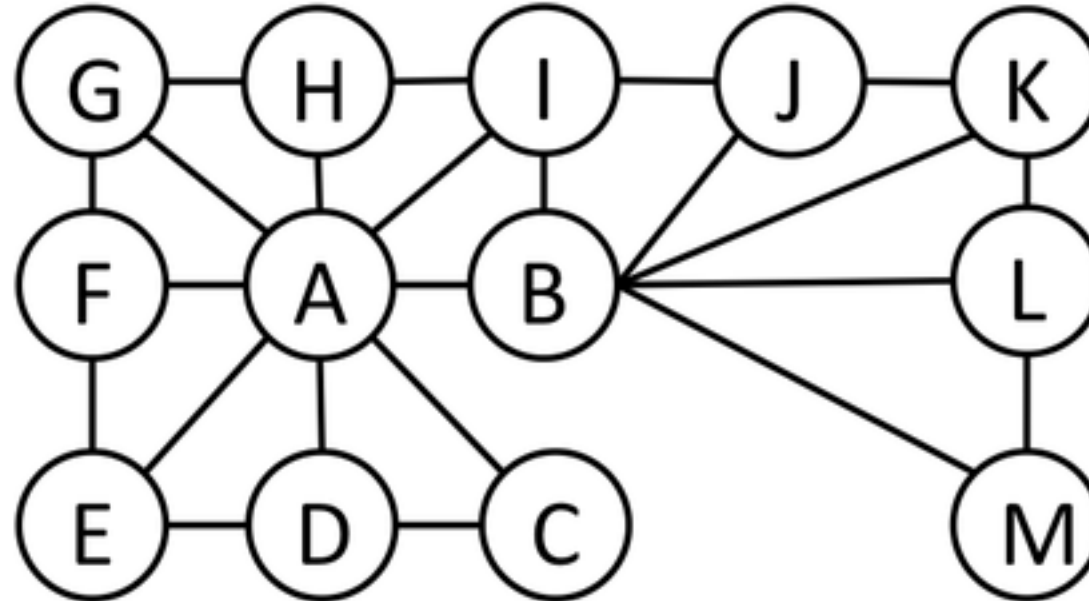Instantiate the cutset
(all possible ways)

Compute residual CSP
for each assignment

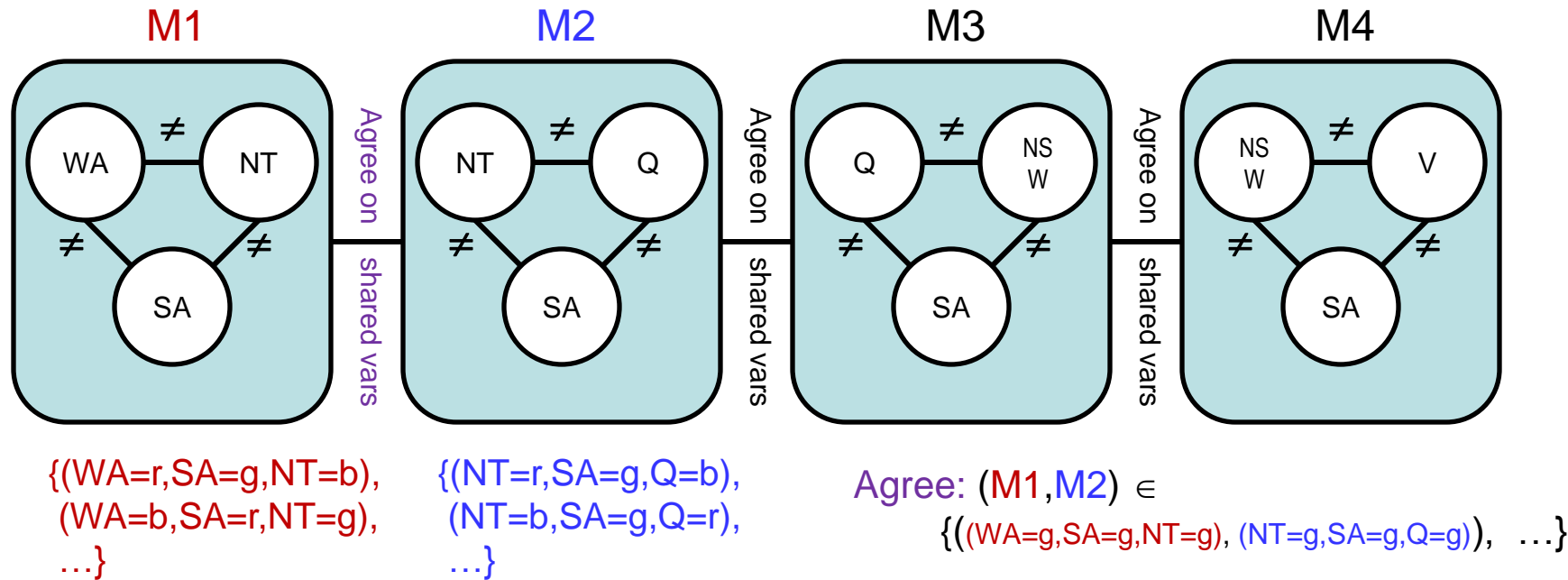Solve the residual CSPs
(tree structured)

Cutset Quiz

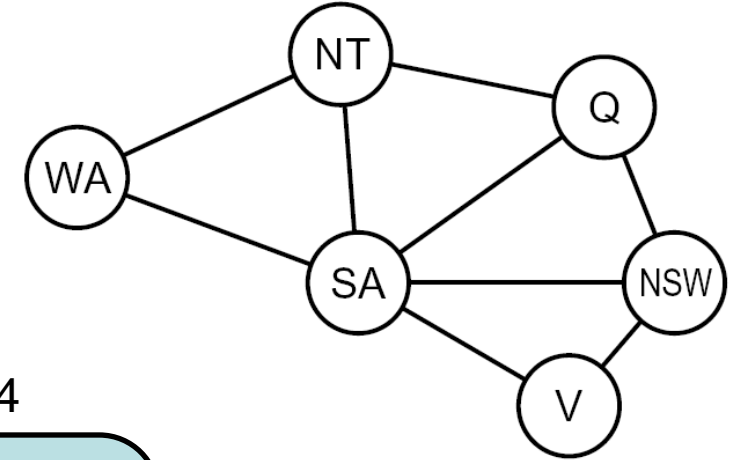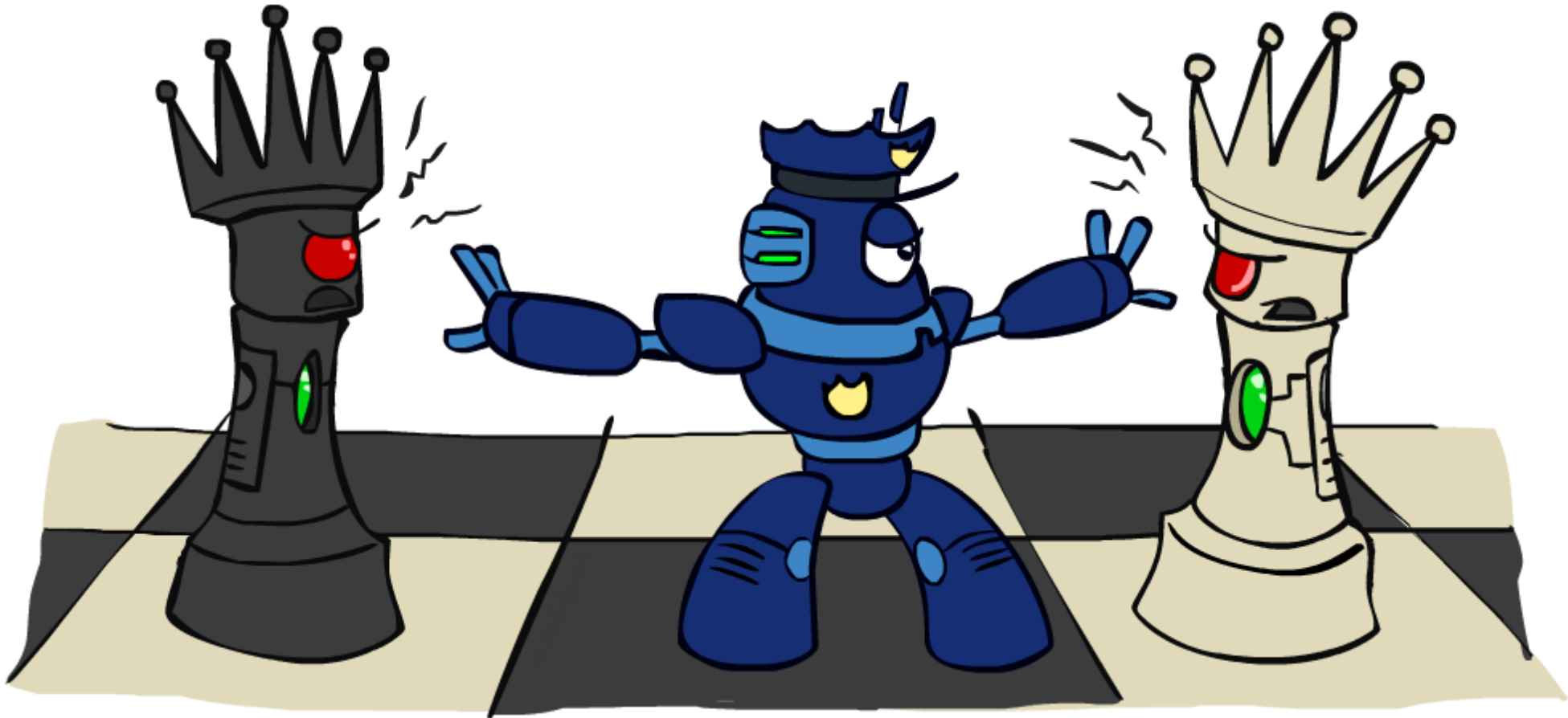- Find the smallest cutset for the graph below.

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)),   …}

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(n) = total number of violated constraints

Example: 4-Queens

h = 5   h = 2   h = 0
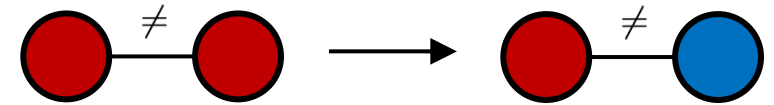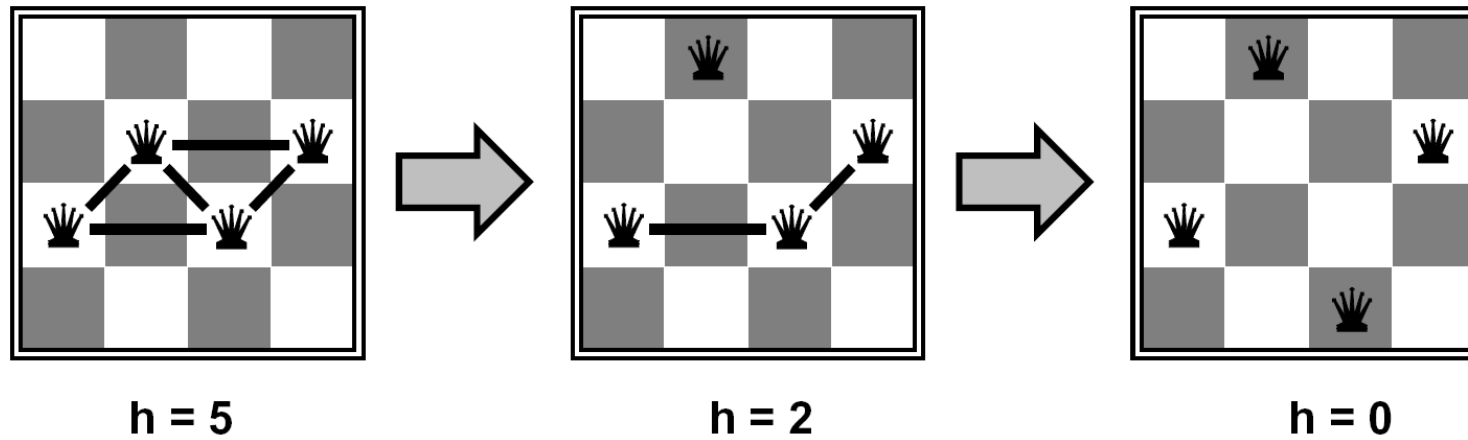
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

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

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

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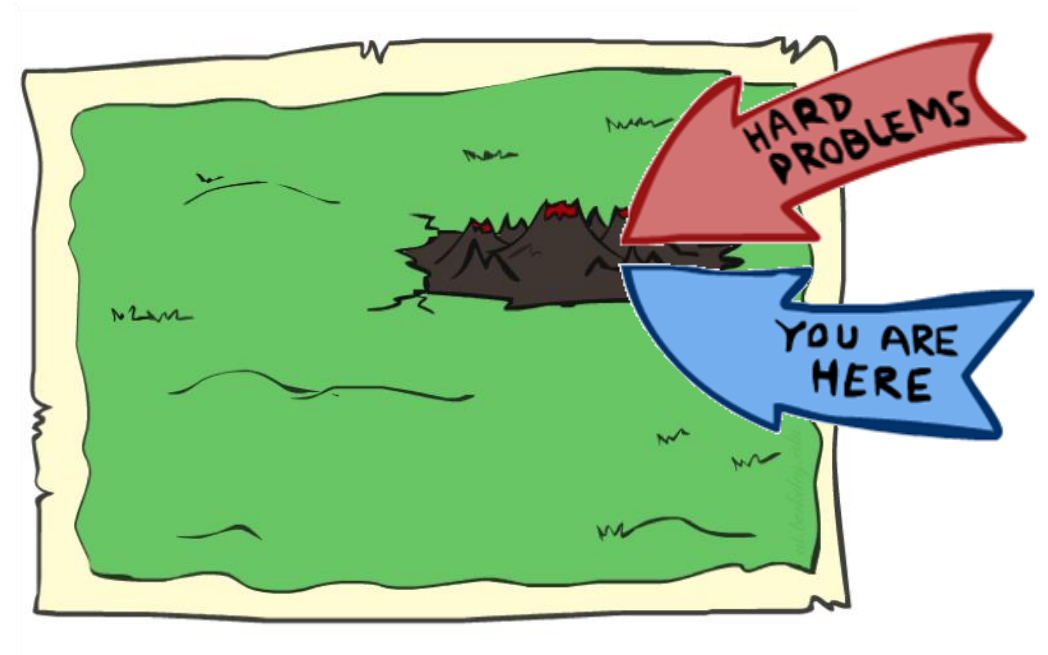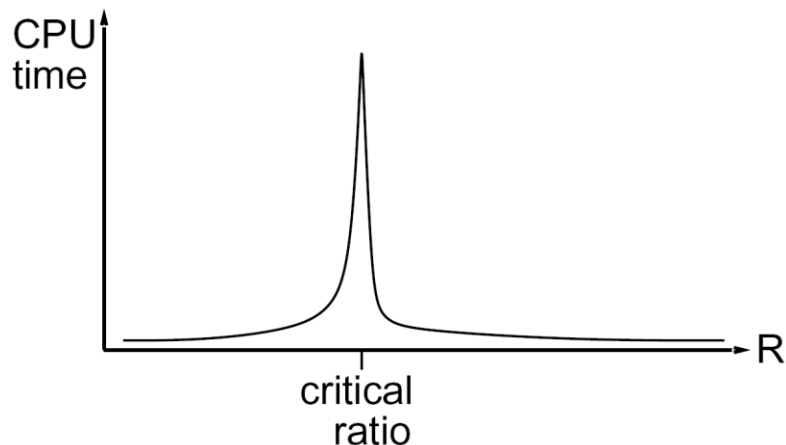
Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constrai

- Basic solution: backtracking sea

- Speed-ups:
  - Ordering
  - Filtering
  - Structure

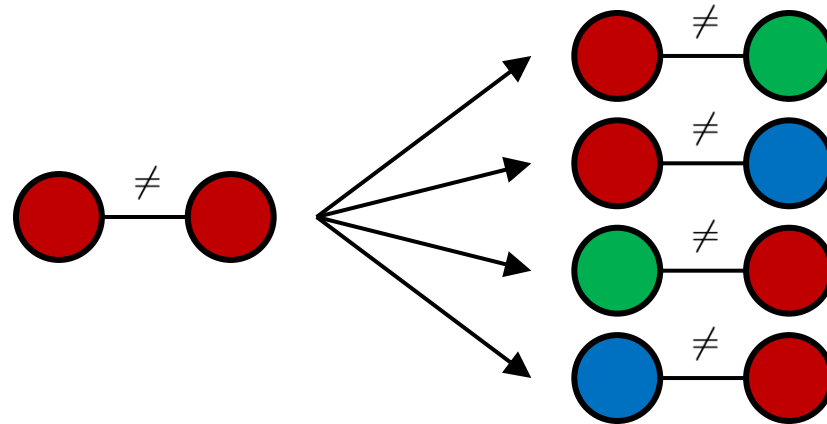- Iterative min-conflicts is often effective in practice
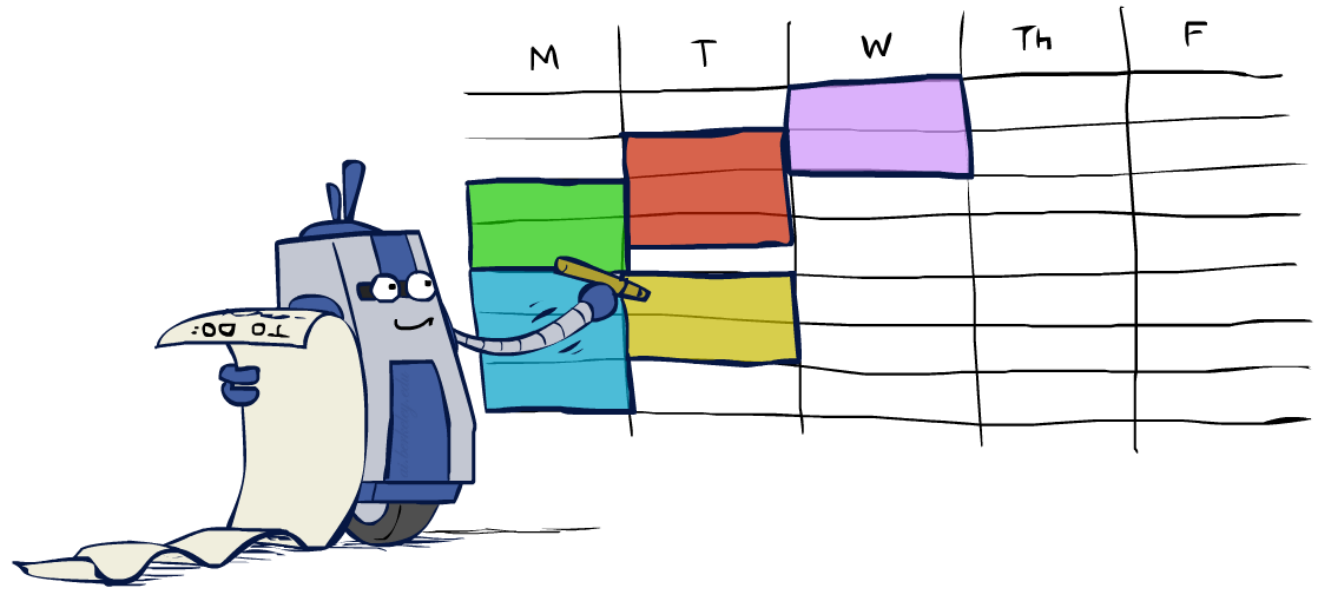
Local Search

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!)

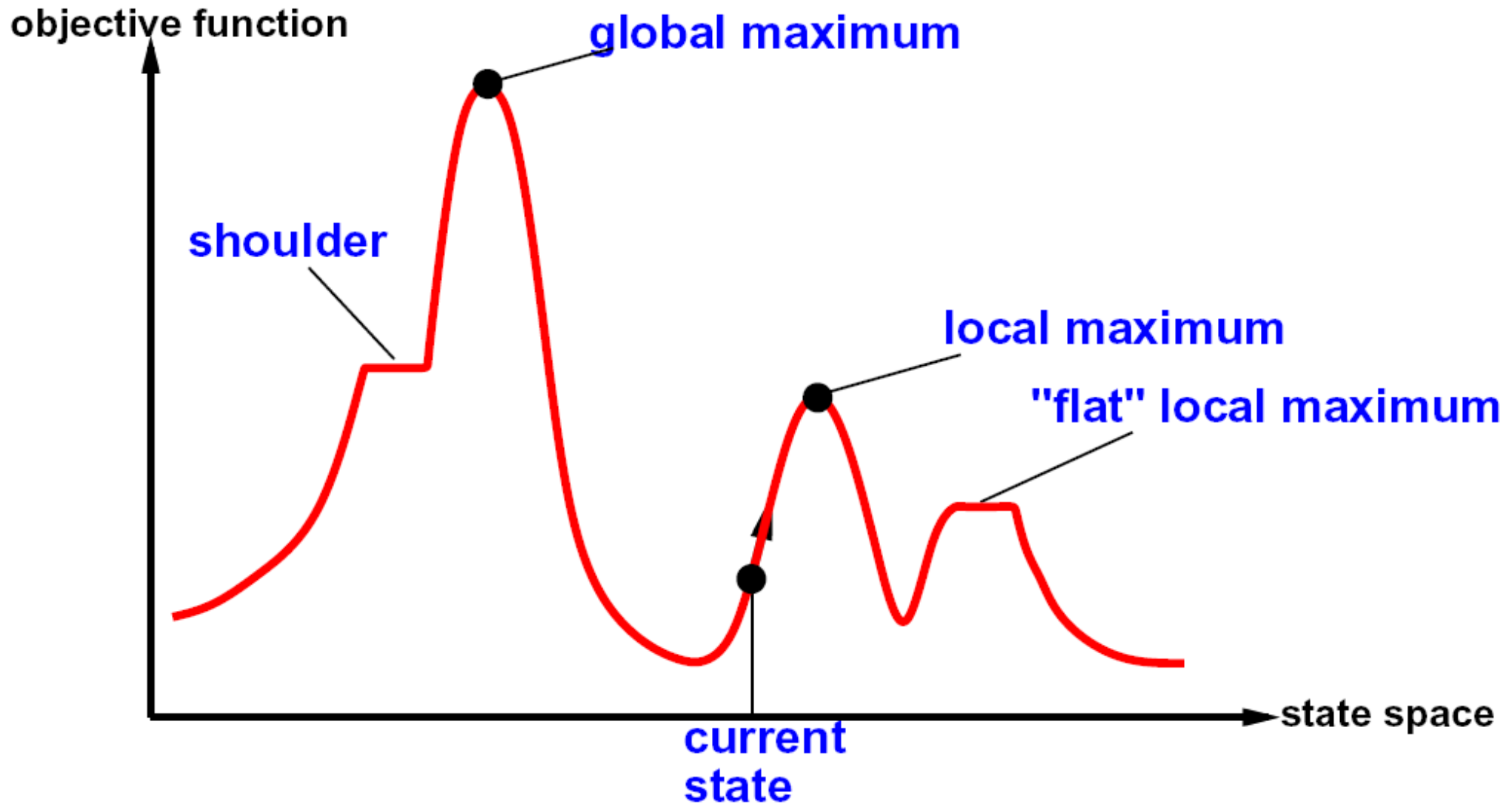- New successor function: local changes

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

Hill Climbing

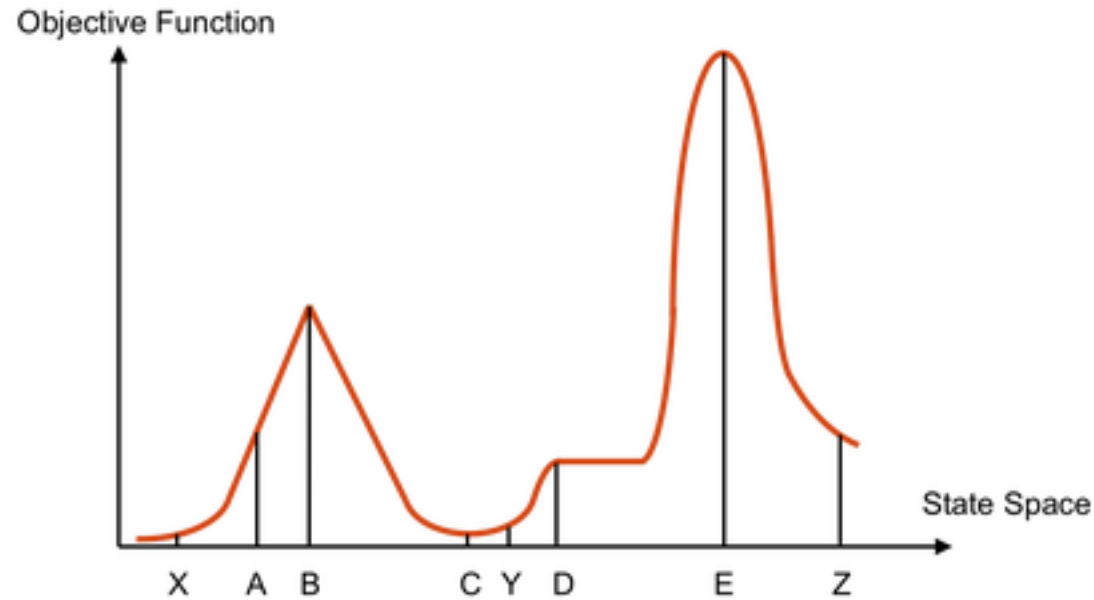- **Simple, general idea:**
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- **What's bad about this approach?**
  - Complete?
  - Optimal?

- **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
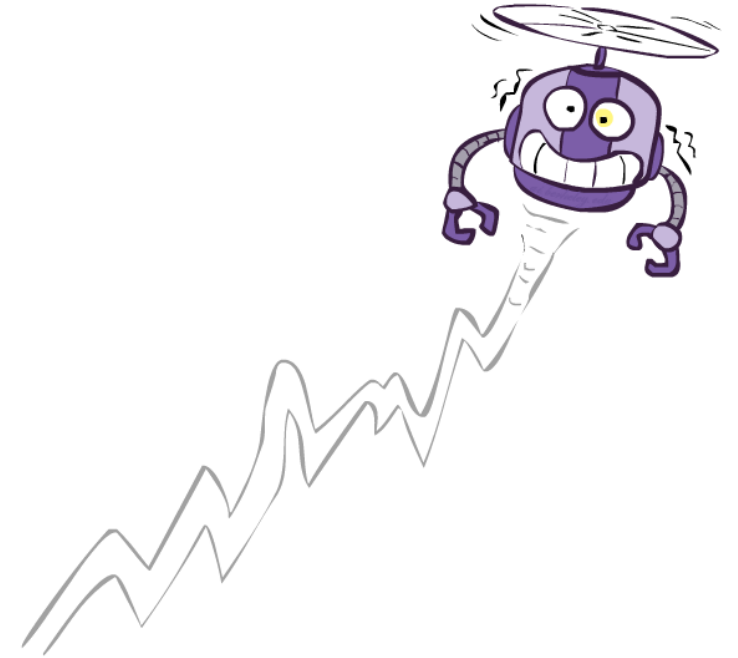 - But make them rarer as time goes on

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 $t \leftarrow 1$ **to** ∞ **do**
        $T \leftarrow schedule[t]$
        if $T = 0$ **then return** *current*
        next ← a randomly selected successor of *current*
        $\Delta E \leftarrow$ VALUE[*next*] − VALUE[*current*]
        if $\Delta E > 0$ **then** *current* ← *next*
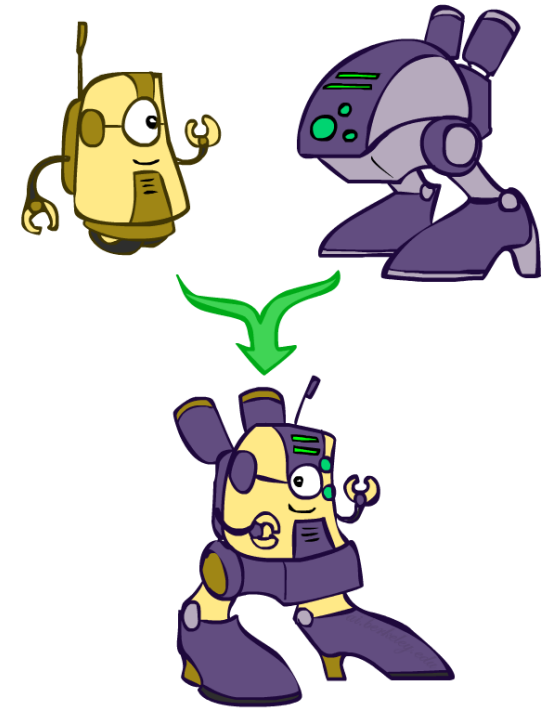   **else** *current* ← *next* only with probability $e^{\Delta E/T}$
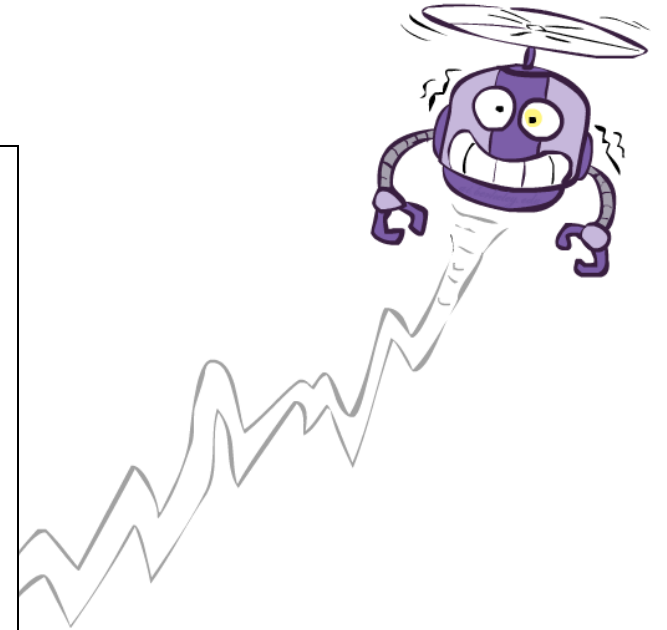
Simulated Annealing

- **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

	24	31%							

24748552   **24** 31%   32752411   32748552 → 32748**1**52

32752411   **23** 29%   24748552   24752411 → 24752411

24415124   **20** 26%   32752411   32752124 → 32**2**52124

32543213   **11** 14%   24415124   24415411 → 2441541**7**

Fitness   Selection   Pairs   Cross–Over   Mutation
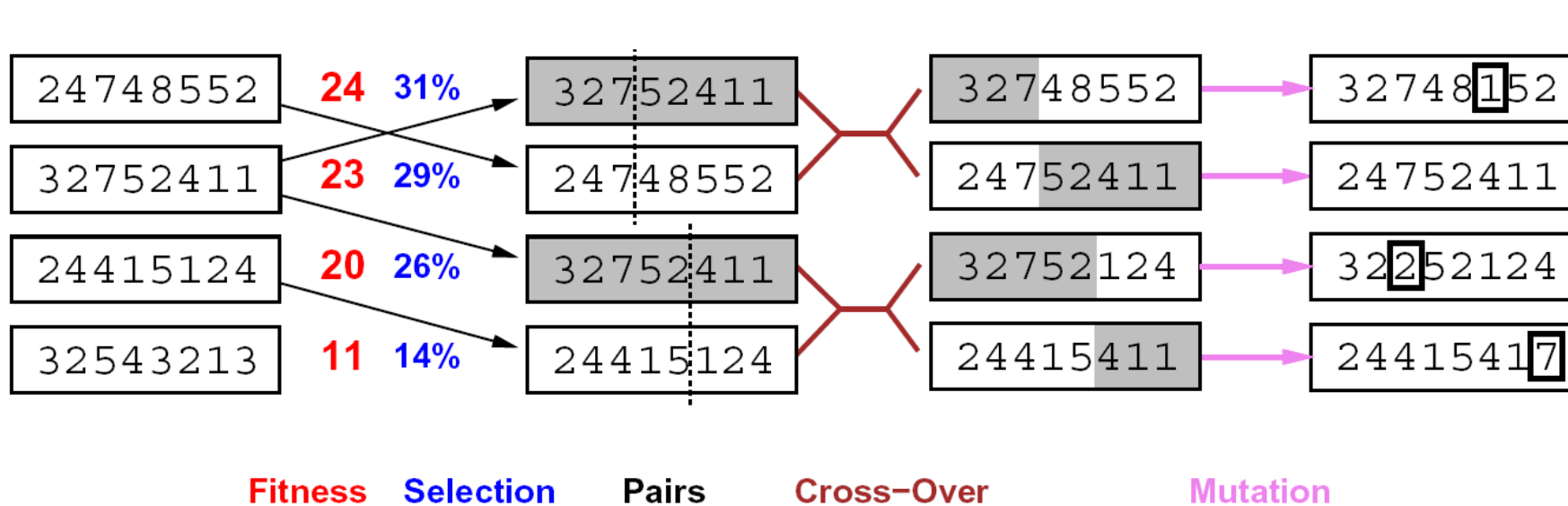
- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety

- Possibly the most misunderstood, misapplied (and even maligned) technique around

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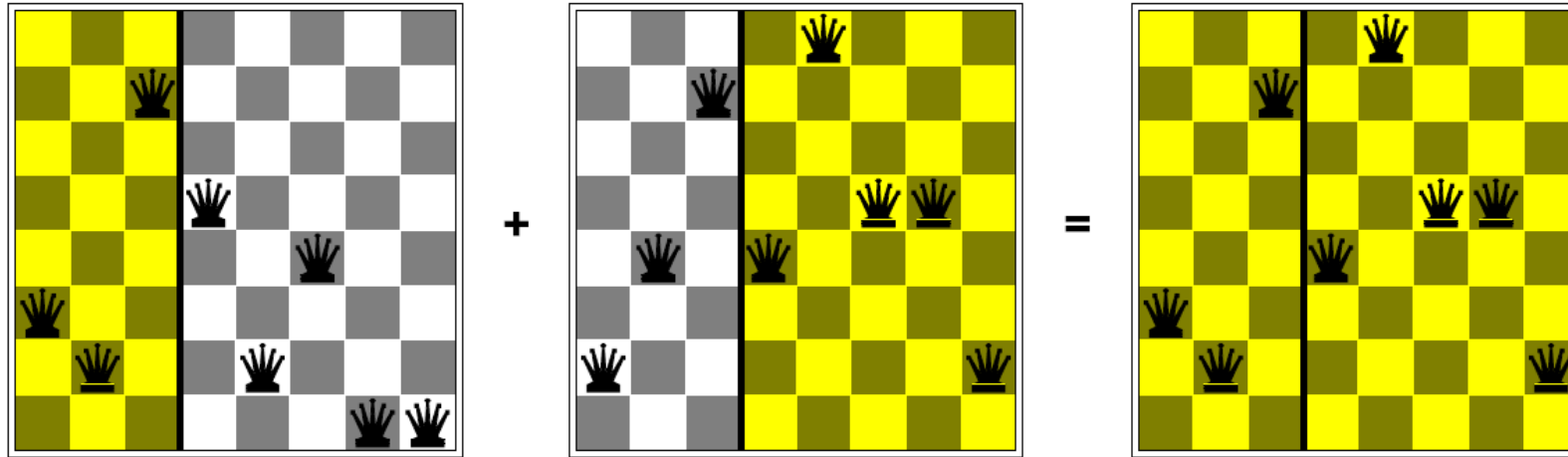
- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?

Today

- Efficient Solution of CSPs

- Local Search