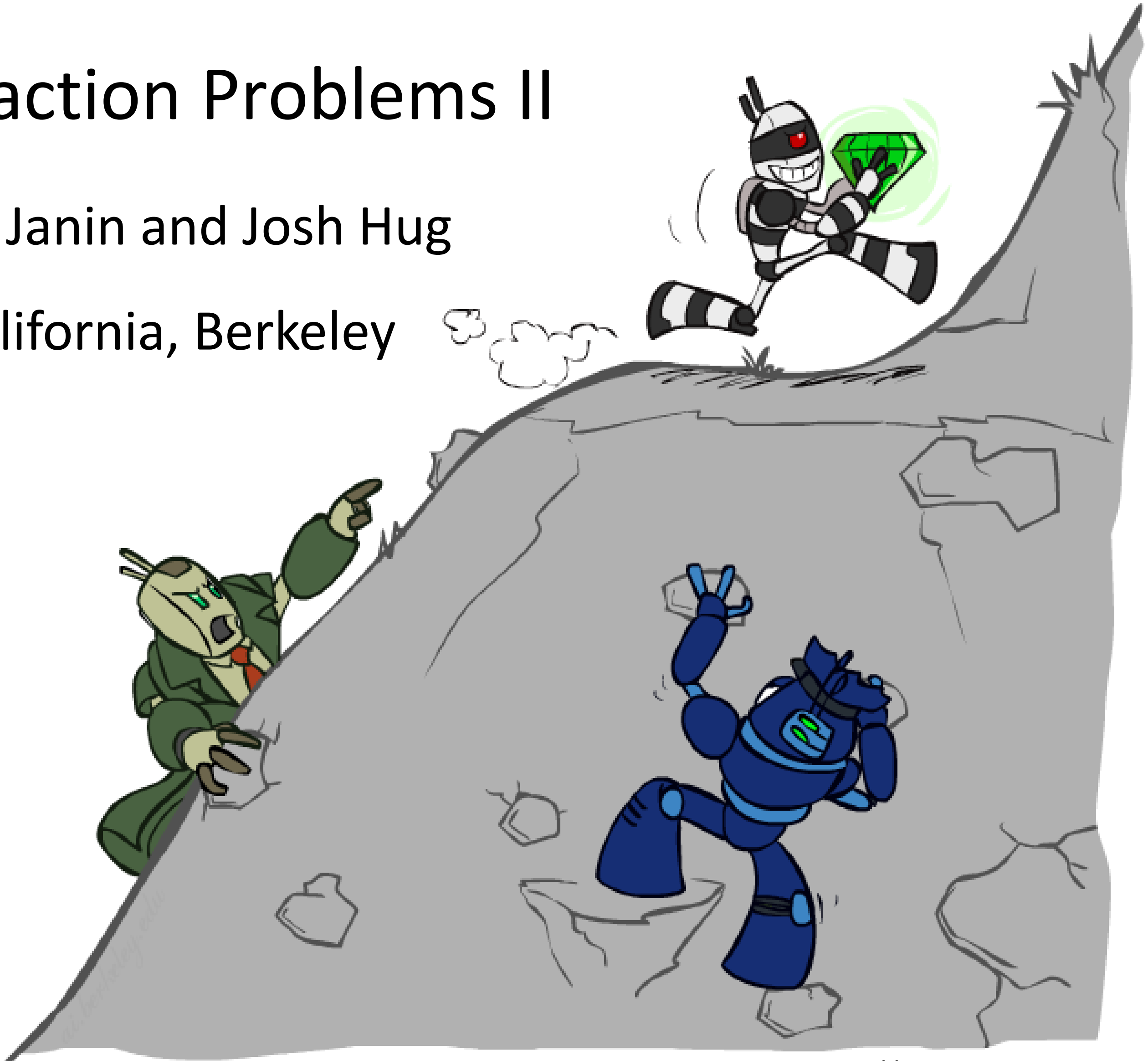
CS 188: Artificial Intelligence

Constraint Satisfaction Problems II

Instructors: Adam Janin and Josh Hug

University of California, Berkeley



[These slides were created by Dan Klein, Pieter Abbeel, and Josh Hug for CS188 Intro to AI at UC Berkeley. All CS188 materials available at http://ai.berkeley.edu.]

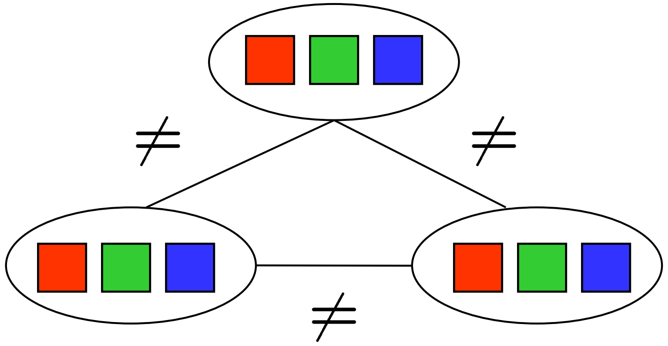
Today

- Efficient Solution of CSPs
  - A few more improvements to CSP-Backtracking.
  - An alternate approach: Iterative Improvement.

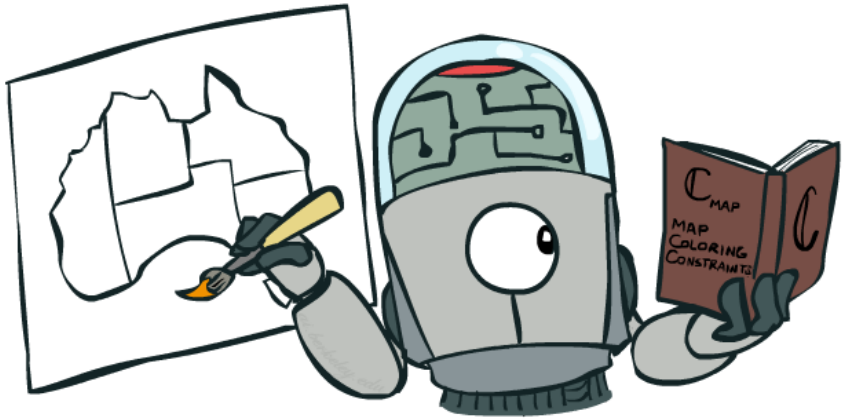
- Local Search

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 CSP-Backtracking

General-purpose ideas give huge gains in speed

✓ Ordering: Can we pick better orderings?

- Yes! Can use most constrained **variable** (MCV) and least constraining **value** (LCV).

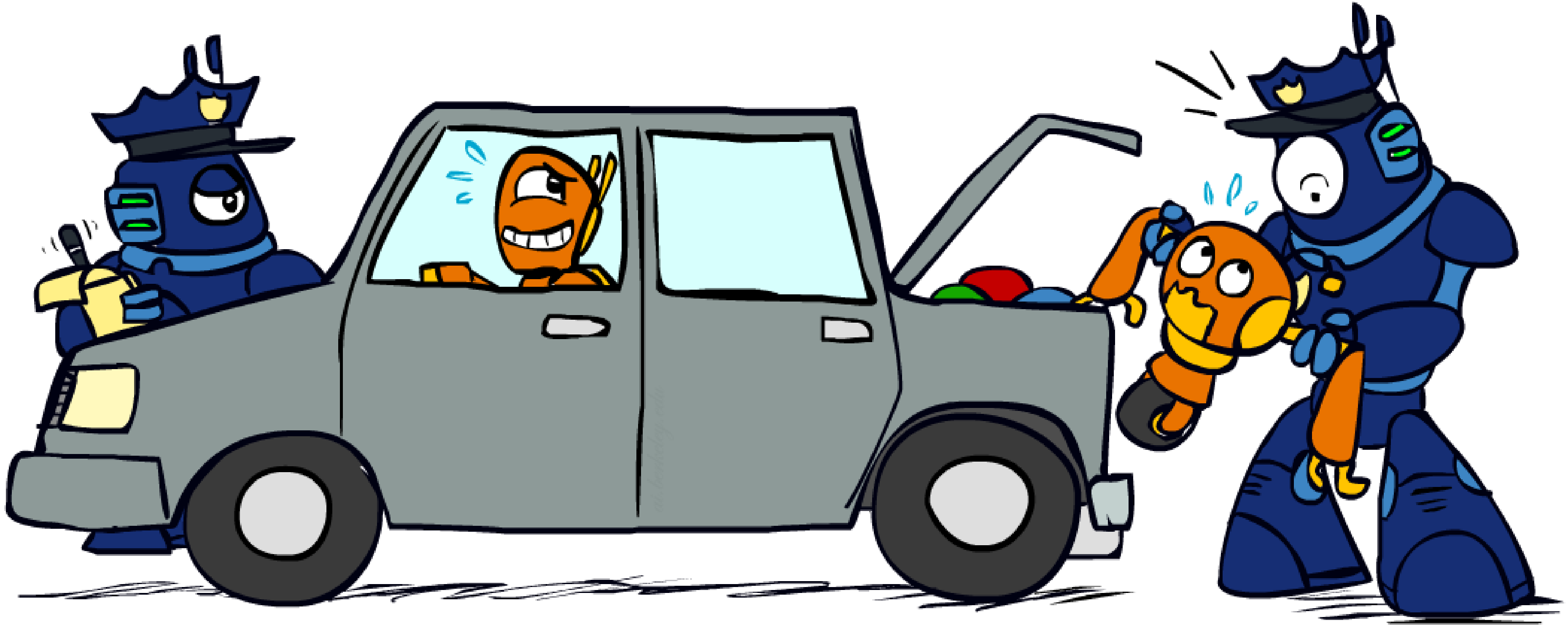
? ▪ Filtering: Can we detect inevitable failure early?

- ✓ Yes! Use forward checking or arc consistency (e.g. AC-3).
  - AC-3 more expensive, but catches problems earlier.
- ? Will discuss new idea of “K-consistency” briefly today.

? ▪ Structure: Can we exploit the problem structure?

Note: Overall problem is still NP-hard.

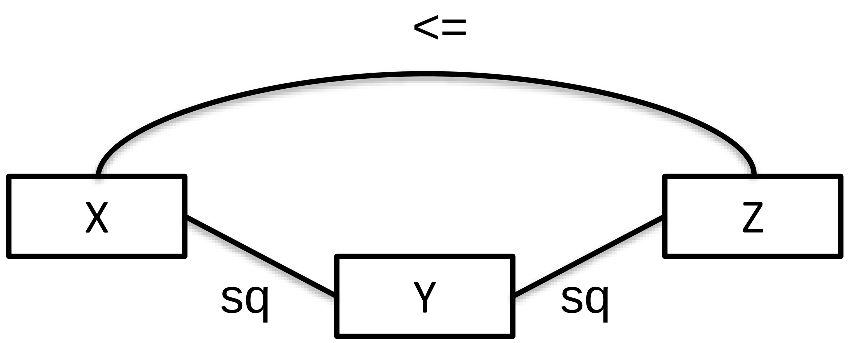
Arc Consistency and Beyond



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

Arc Consistency Practice/Review

- Variables: X, Y, Z
- Domains: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
- Constraints:
  - X is either the square or the square root of Y.
  - Y is either the square or the square root of Z.
  - X is less than or equal to Z.



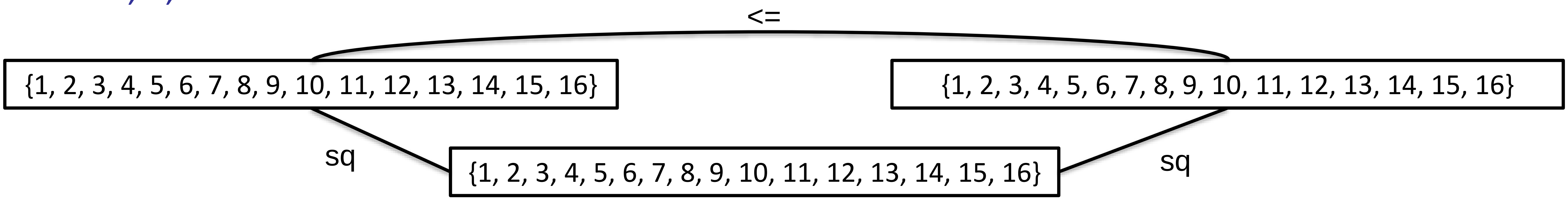
- Some sample solutions:
  - X, Y, Z = `{1, 1, 1}`
  - X, Y, Z = `{2, 4, 16}`
  - X, y, Z = `{9, 3, 9}`

- Problem 1: If we run AC-3 on this constraint graph, what are the possible values for X, Y, Z?
- Problem 2: If we run AC-3, then assign X = 16, then run arc-consistency again, what are the possible values for Y and Z?

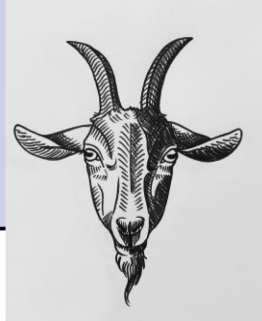
Arc Consistency Practice

- Variables: X, Y, Z
- Domains: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

- Problem 1: If we run AC-3 on this constraint graph, what are the possible values for X, Y, Z?

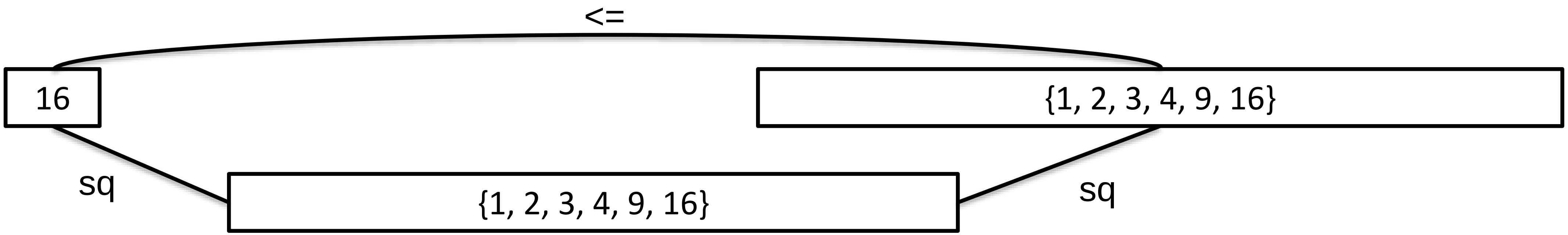


Arc Consistency Practice



- Variables: X, Y, Z
- Domains: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

- Problem 2: If we run AC-3, assign X = 16 and run AC-3 again, what are the possible values for Y and Z?

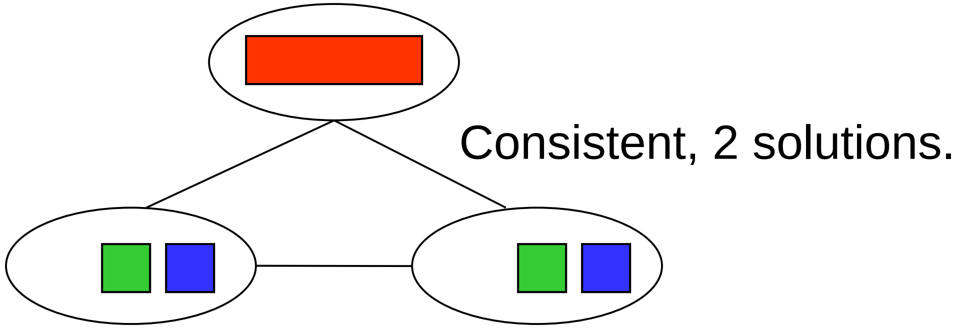


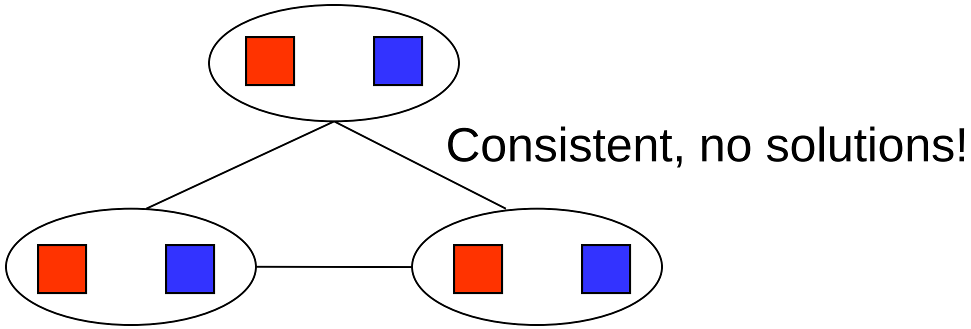
Reminder: If you delete from a node, must re-enqueue all arcs pointing at that node!

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)

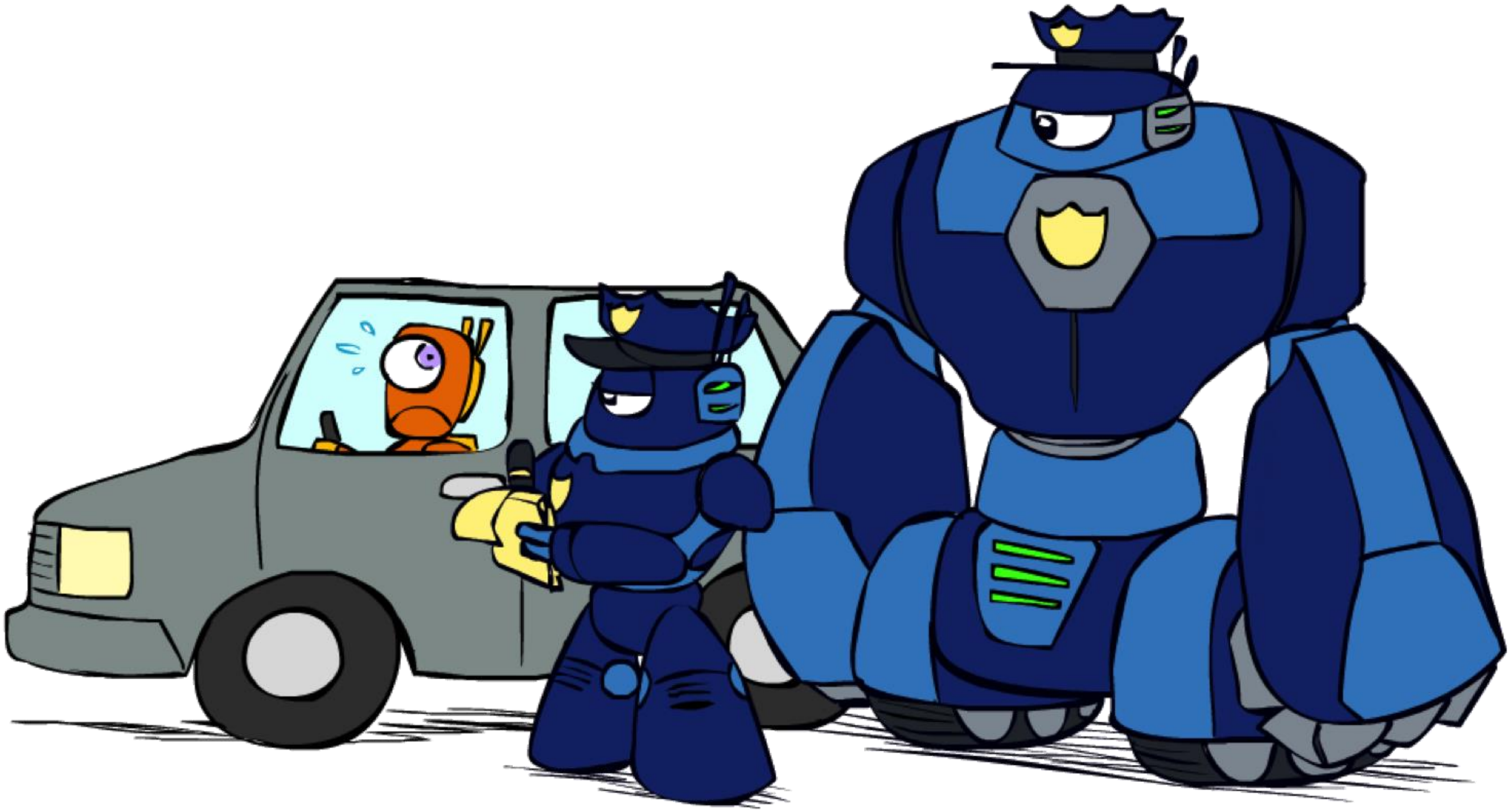
- Arc consistency still runs inside a backtracking search!





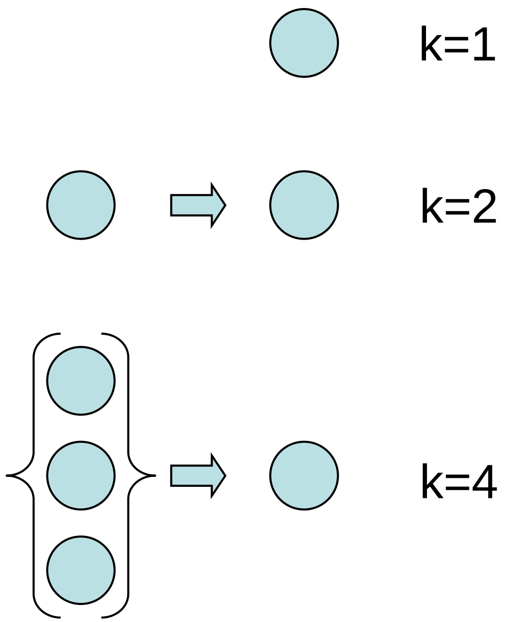
What went wrong here?

K-Consistency

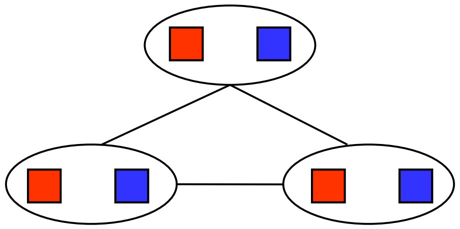


K-Consistency

- 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 k^{th} node



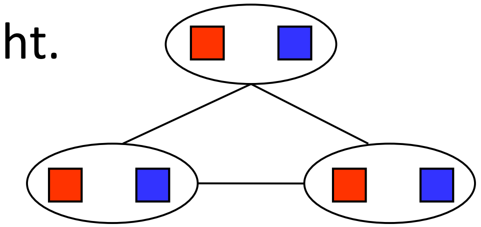
- Higher k more expensive to compute
- (You need to know the k=2 case: **arc consistency**)



This graph is 2-consistent, but not 3-consistent.

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 (a.k.a. 2-consistency) and n-consistency!
  - Example, k = 3 (path consistency) helps avoid situations like that to the right.
  - Won't discuss algorithms for k > 2. If you're curious see this link.



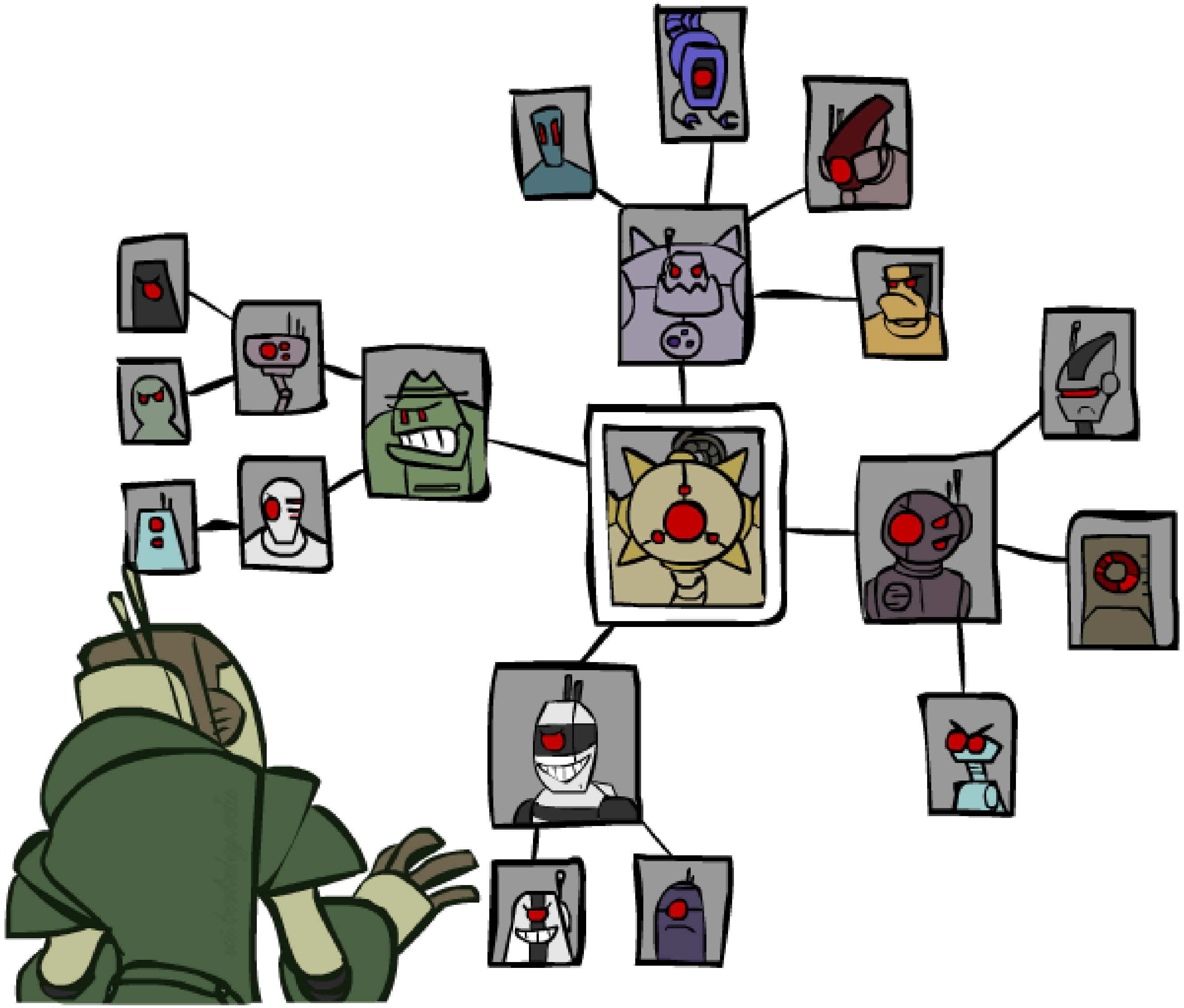
Improving CSP-Backtracking

General-purpose ideas give huge gains in speed

- ✓ Ordering: Can we pick better orderings?
  - Yes! Can use most constrained **variable** (MCV) and least constraining **value** (LCV).
- ✓ Filtering: Can we detect inevitable failure early?
  - Yes! Use forward checking, **arc consistency**, or K-consistency.
  - K-consistency > **arc consistency** > forward checking in both cost and quality of filtering (assuming k > 2).
- ? Structure: Can we exploit the problem structure?

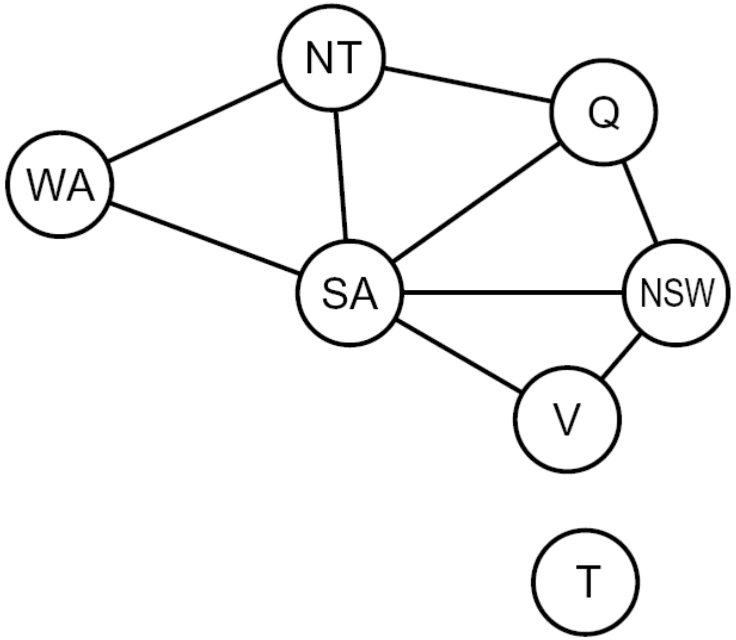
Note: Overall problem is still NP-hard.

Structure

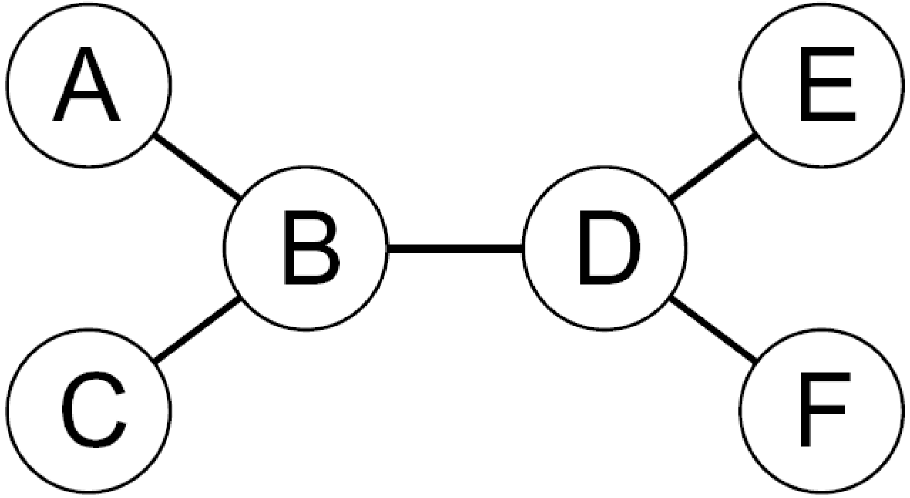


Problem Structure

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



Tree-Structured CSPs

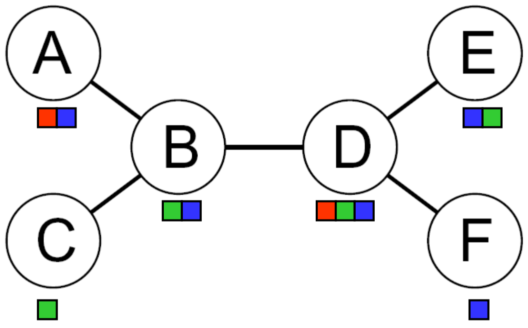


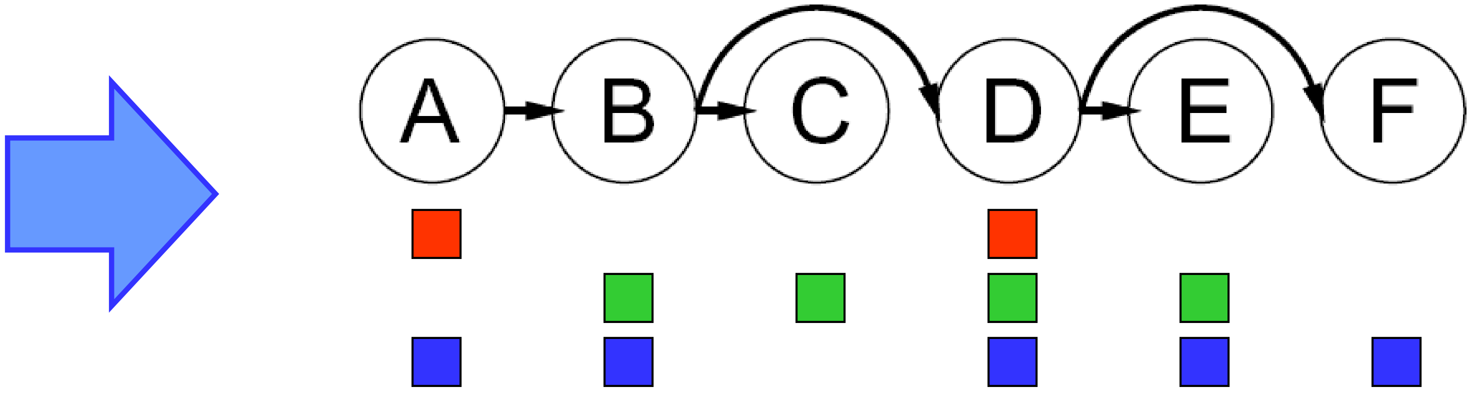
- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\, d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- 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



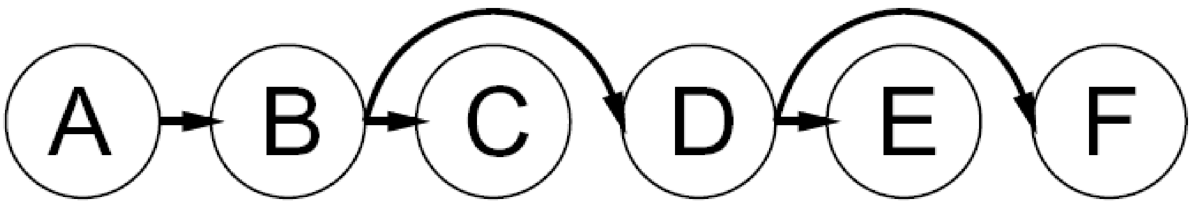


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

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

Tree-Structured CSPs (Tricky Slide. Revisit Later!)

- Claim 1: After backward pass, all parent-to-child arcs are 2-consistent
- Proof: Each X→Y was made 2-consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)

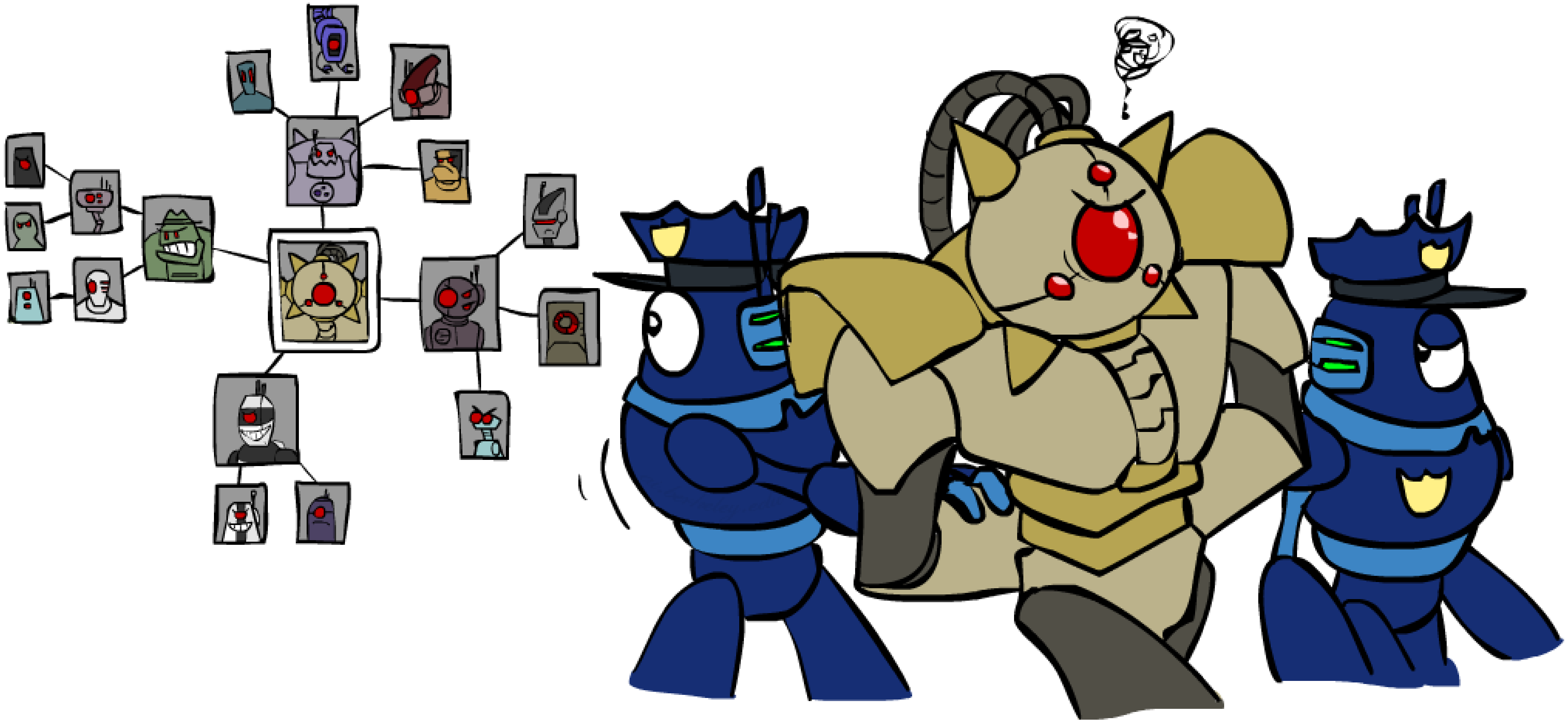


- Claim 2: If parent-to-child arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position, e.g. A->B 2-consistent means **any** choice for A is safe for B.

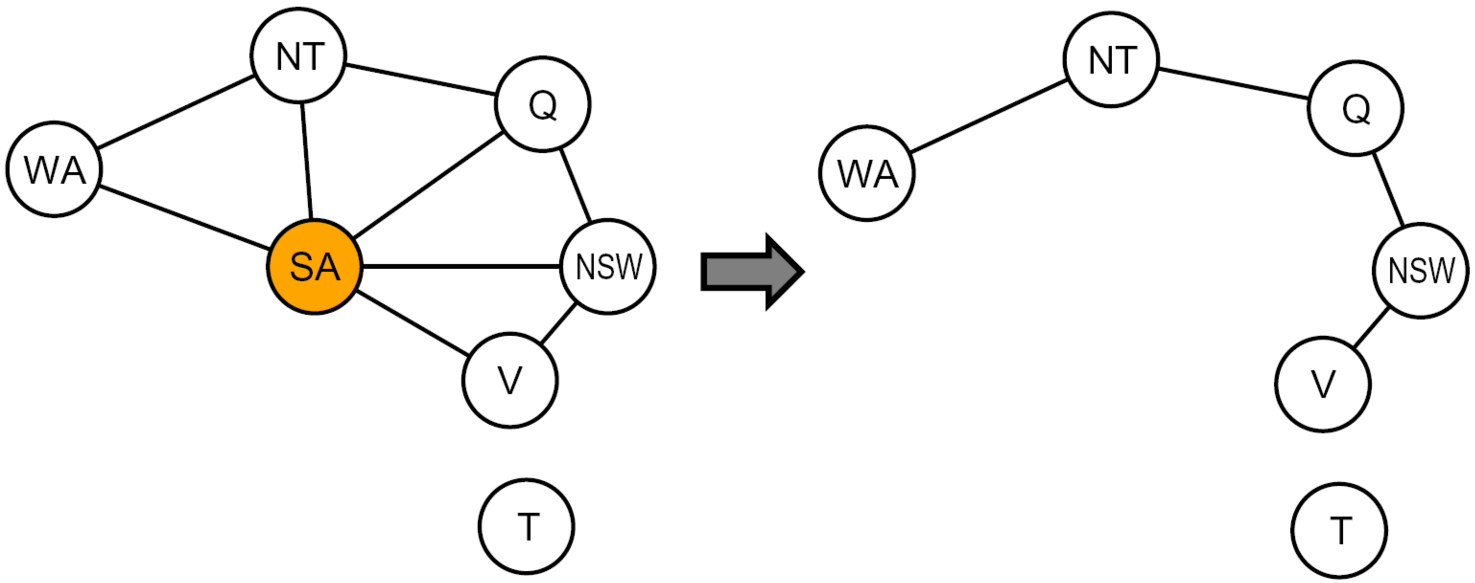
- Why doesn't this algorithm work with cycles in the constraint graph?
  - Arc-consistency doesn't guarantee safety for multiple parents! Induction on position fails.

- 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

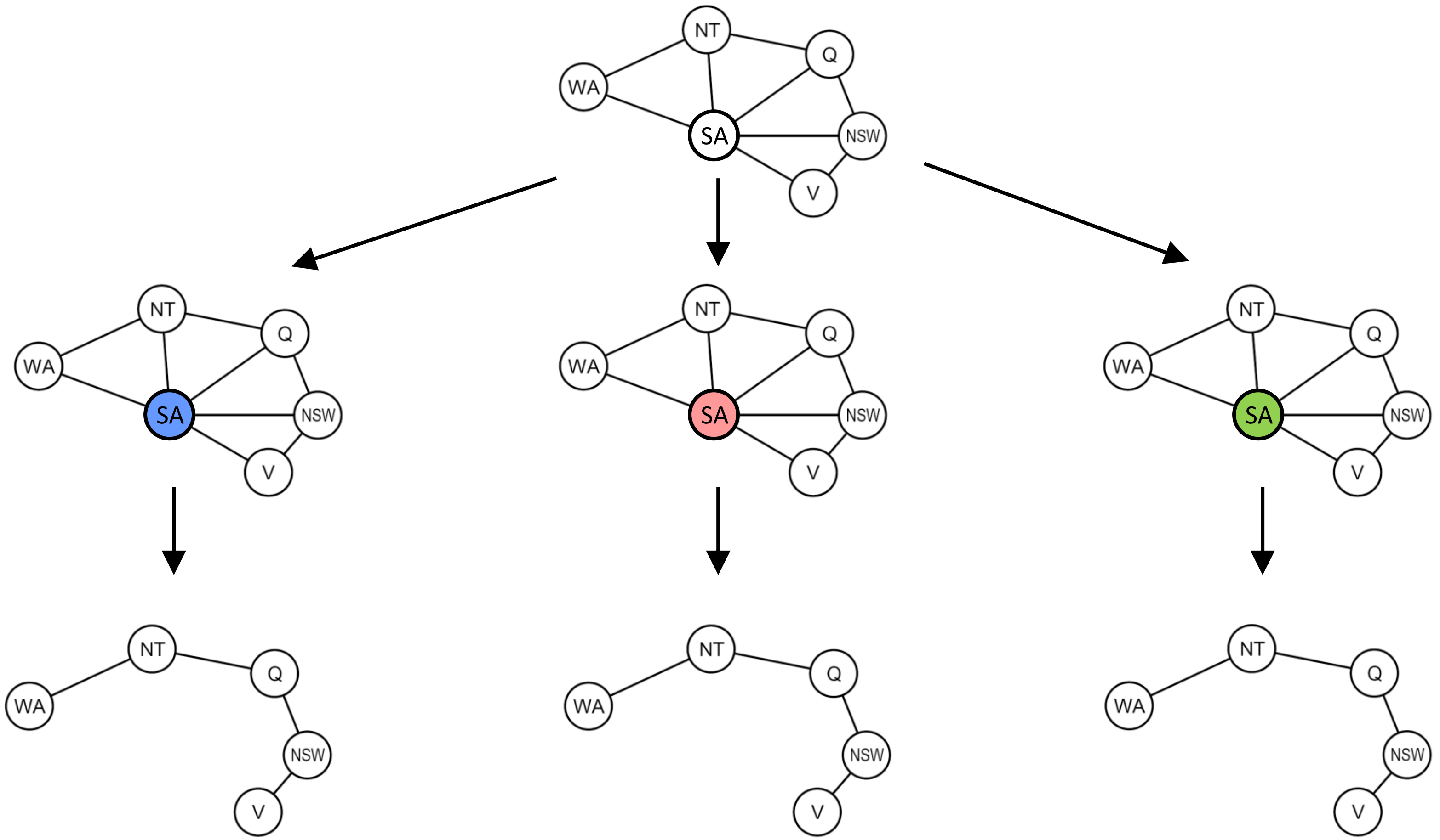
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

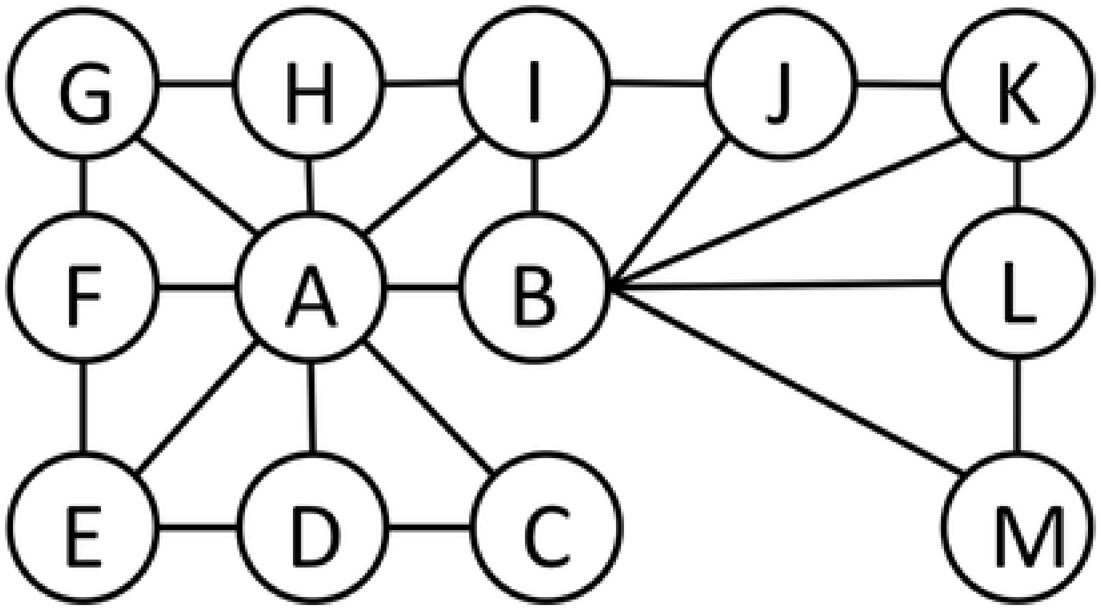
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



Finding the Minimum Cutset

- Find the smallest cutset for the graph below.
  - Set of nodes which, when removed, results in a tree.



Improving CSP-Backtracking

General-purpose ideas give huge gains in speed

✓ Ordering: Can we pick better orderings?

- Yes! Can use most constrained **variable** (MCV) and least constraining **value** (LCV)

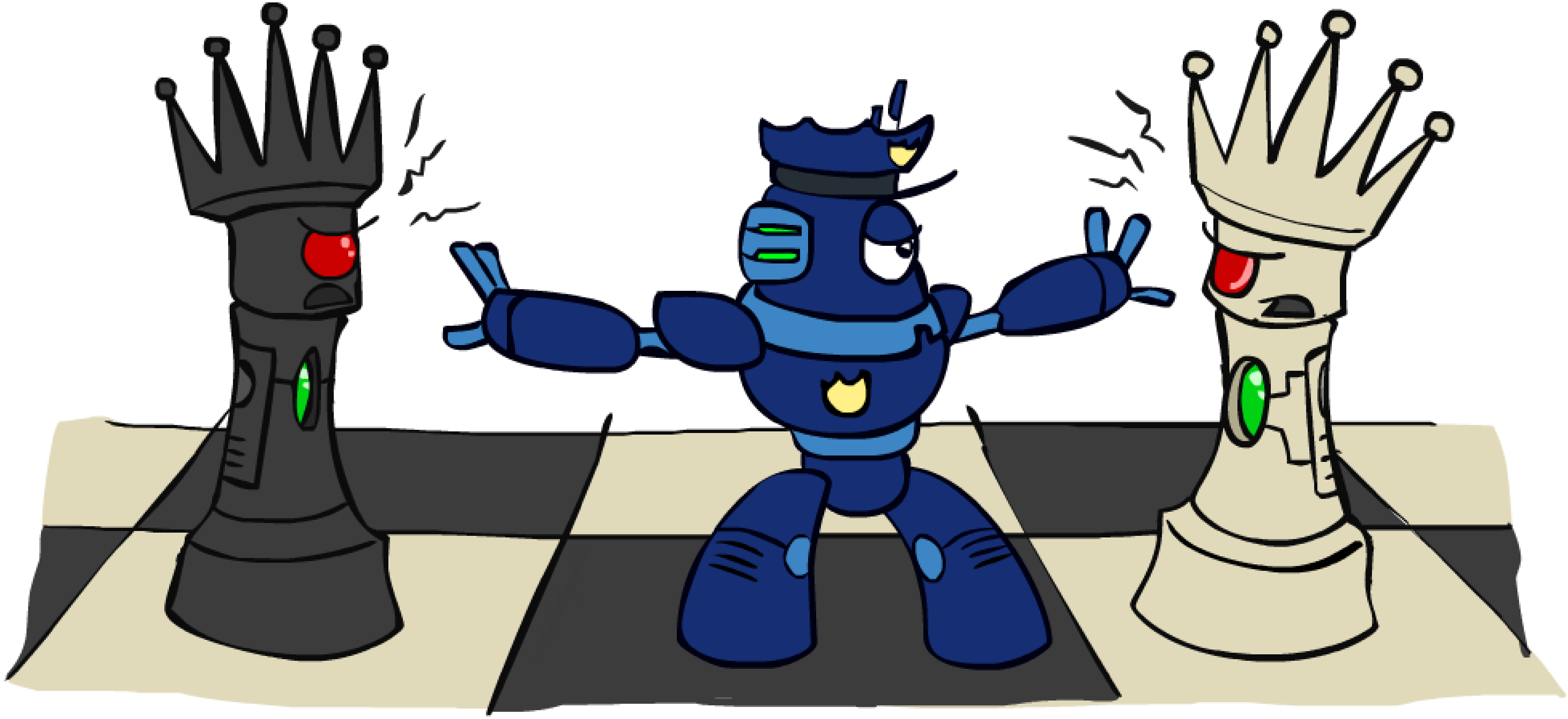
✓ Filtering: Can we detect inevitable failure early?

- Yes! Use forward checking, arc consistency, or K-consistency
- K-consistency > arc consistency > forward checking in both cost and quality of filtering

✓ Structure: Can we exploit the problem structure?

- Tree CSPs: Use tree-algorithm
- Almost tree CSPs: Use cutset conditioning
- Wanna get fancy? Try Tree Decomposition

Alternate Approach: Iterative Improvement

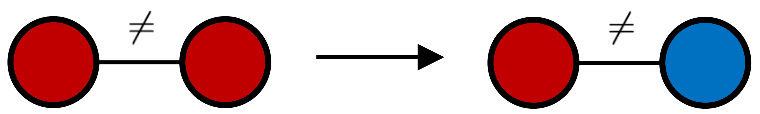


CSP-Backtracking works, but there are other approaches. Iterative Improvement is one such approach.

Iterative Algorithms for CSPs

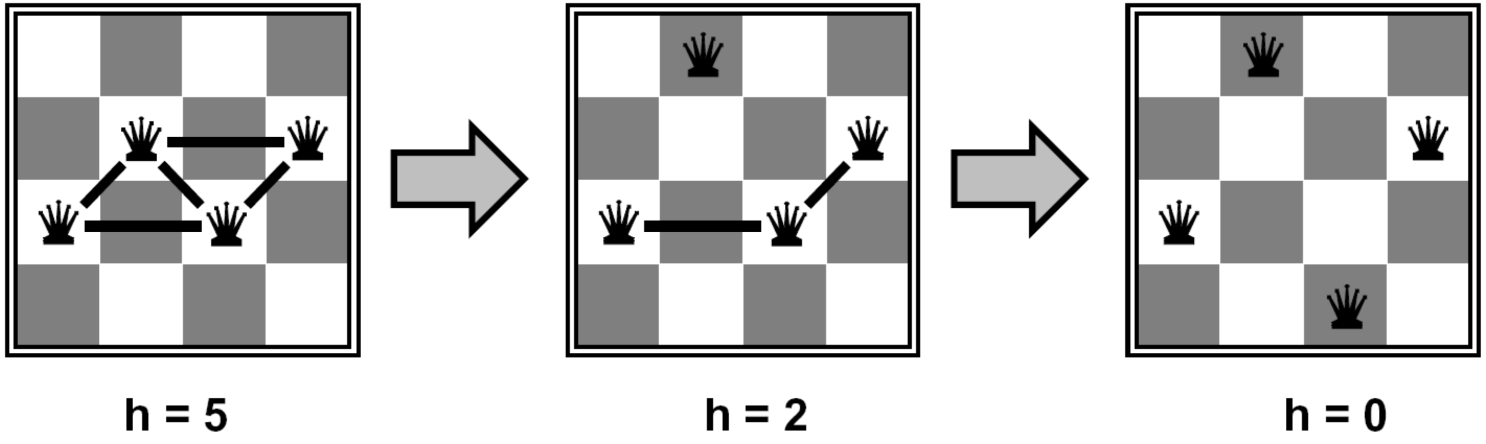
- Local search methods typically work with "complete" states, i.e., all variables assigned

- 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
    - i.e., don't change values if all the other possibilities are actually worse

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

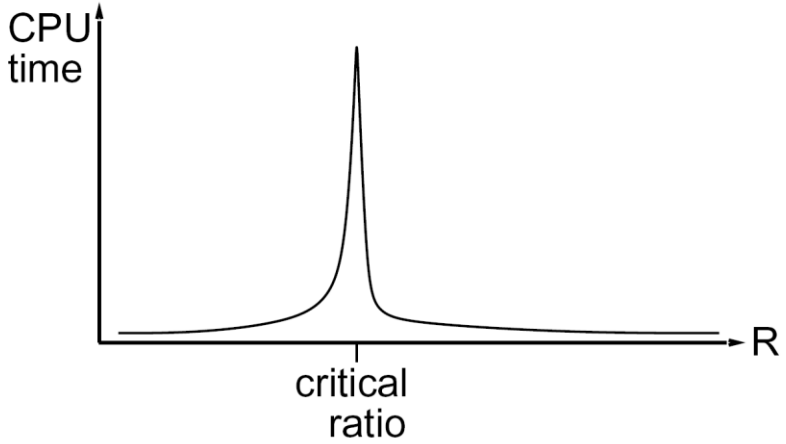
Video of Demo Iterative Improvement – Coloring

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

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



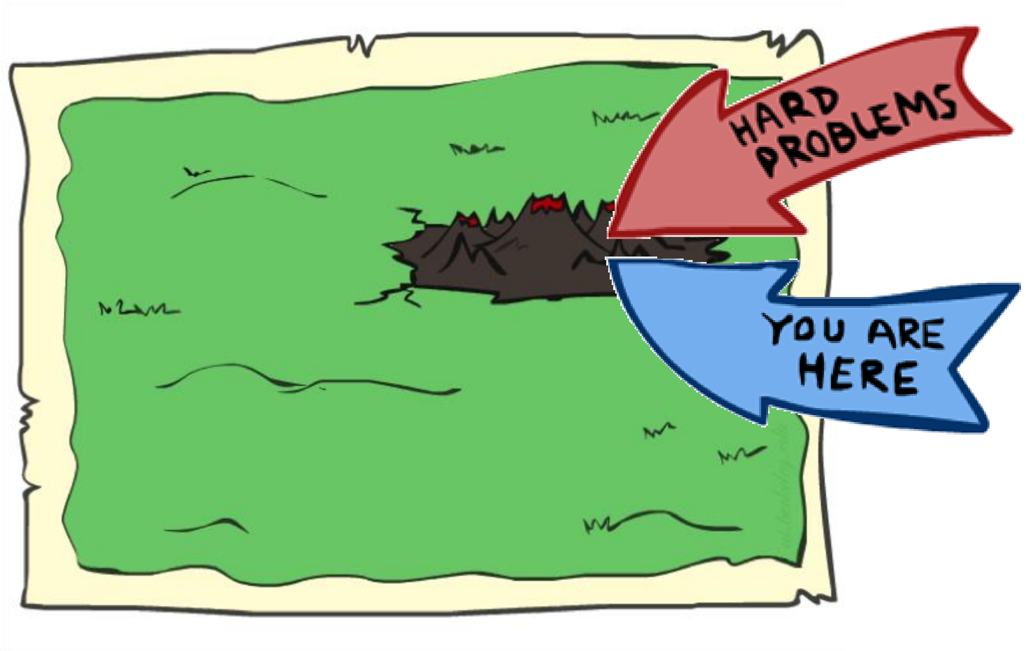
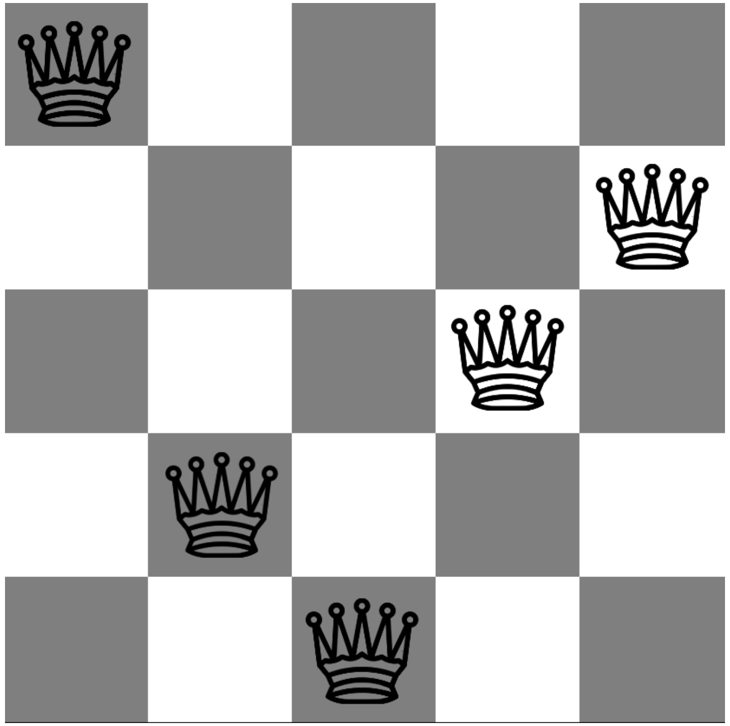


Figure via Stuart Russell

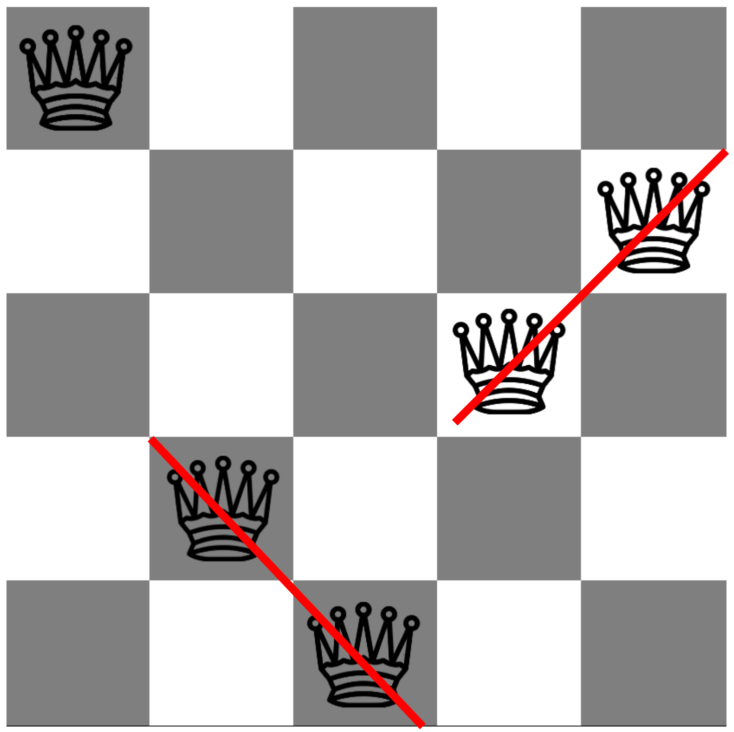
Min-Conflicts and 5-Queens

- For the 5-Queens board shown:
  - How many conflicts are there?
  - What is the best move that Min-Conflicts can make (in terms of # of conflicts)?
    - Recall: Can only move a single queen within a single row.
  - How many conflicts remain after taking the best move?



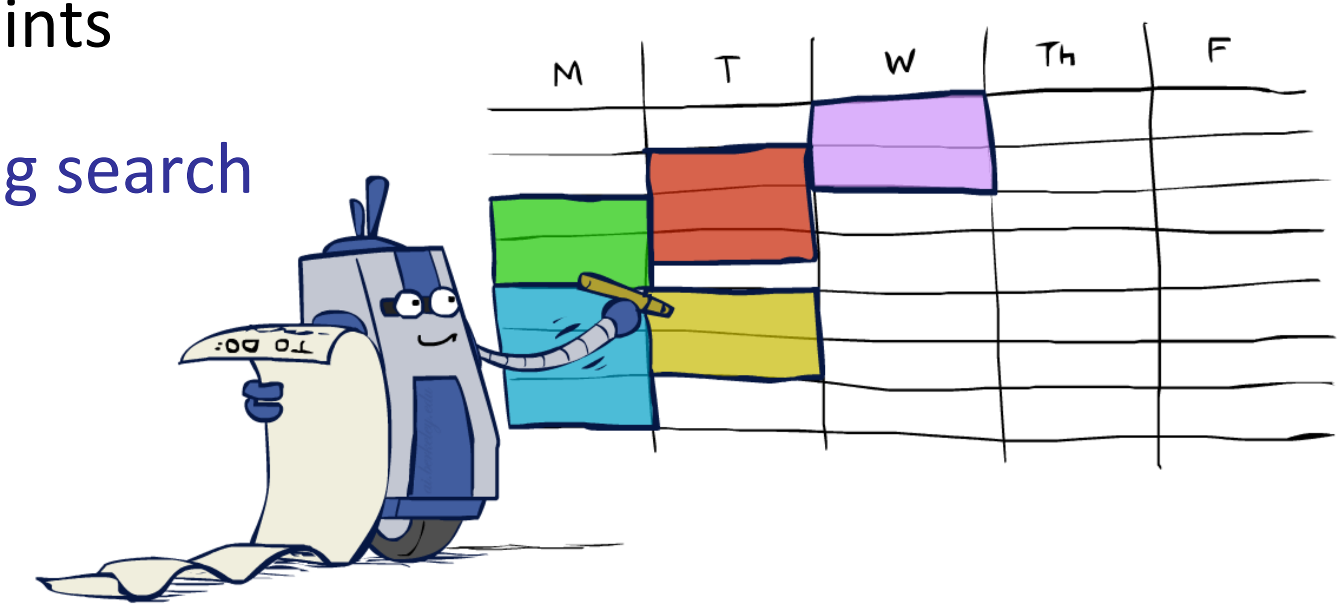
Min-Conflicts, 5-Queens, Sideways Moves

- For the 5-Queens board shown:
  - How many conflicts are there?
    - 2
  - What is the best move that Min-Conflicts can make (in terms of # of conflicts)?
    - There are many possibilities, e.g. moving Queen 2 to Column 2.
  - How many conflicts remain after taking the best move?
    - 2
- A move that does not improve things is known as a "sideways" move.
  - This is in contrast to an "uphill" move.



Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: CSP-backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Alternate solution: Iterative Min-conflicts. Often effective in practice, particularly for huge problems.



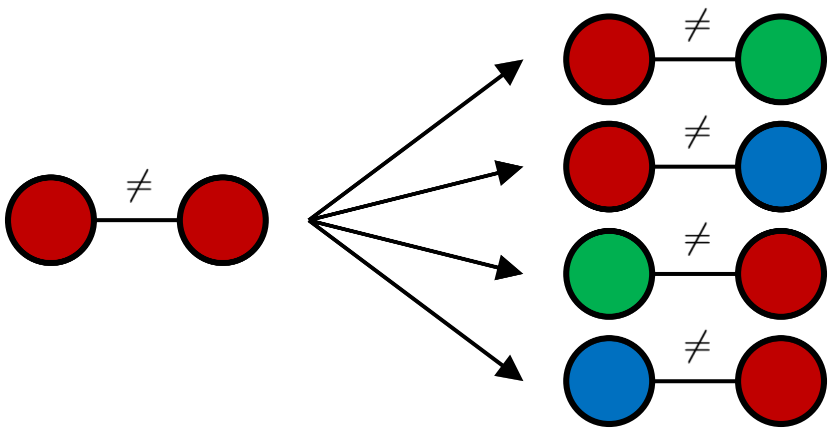
Local Search

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
  - Examples: DFS, BFS, UCS, A*, CSP-Backtracking.

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

- New successor function: local changes

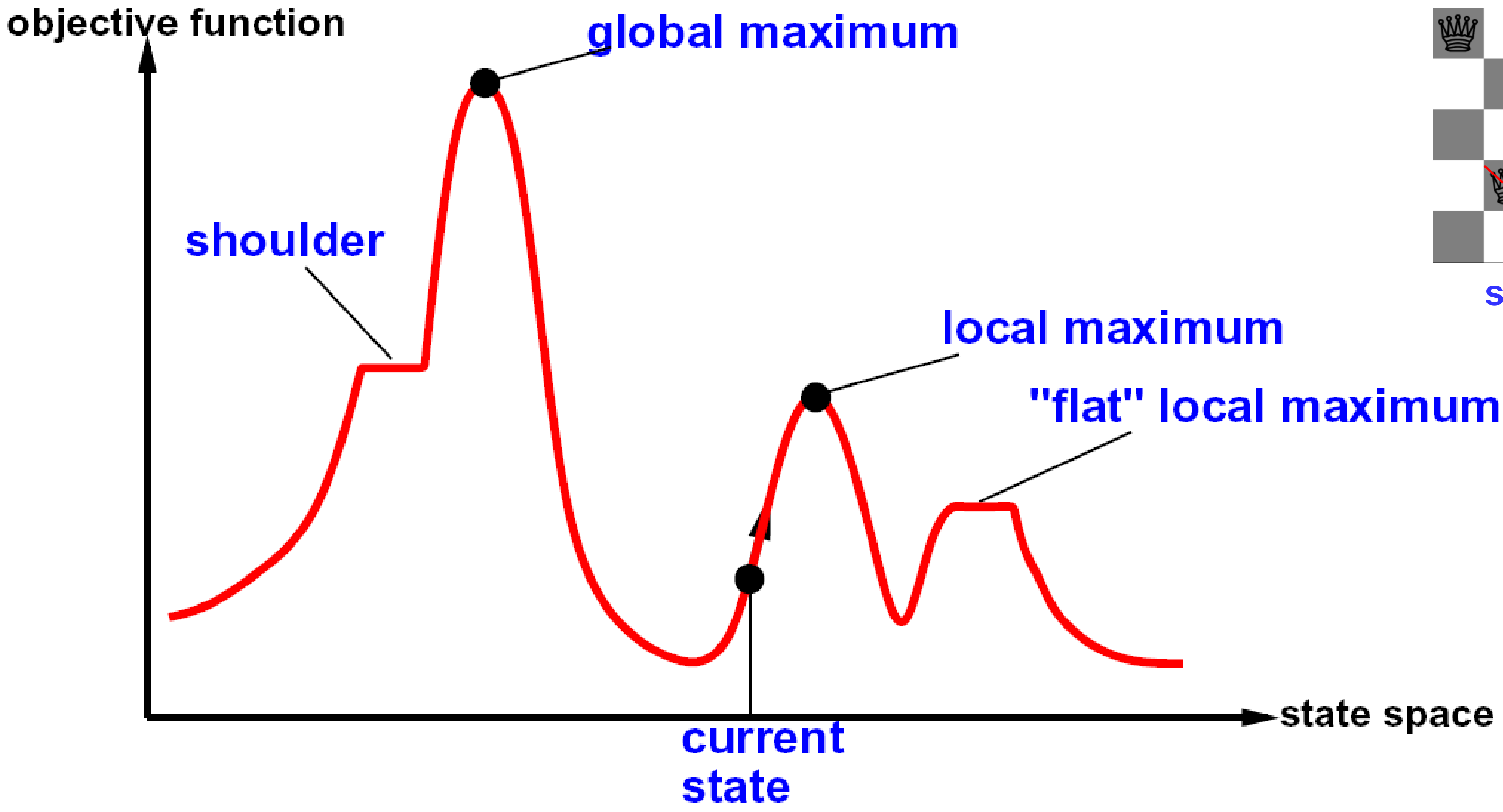


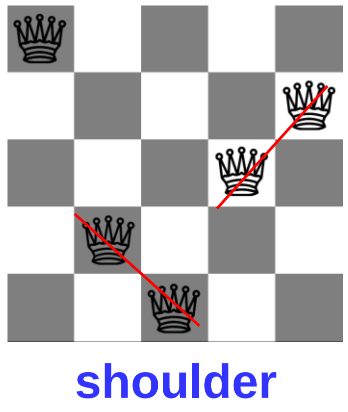
- Generally much faster and more memory efficient (but incomplete and suboptimal)

Hill Climbing

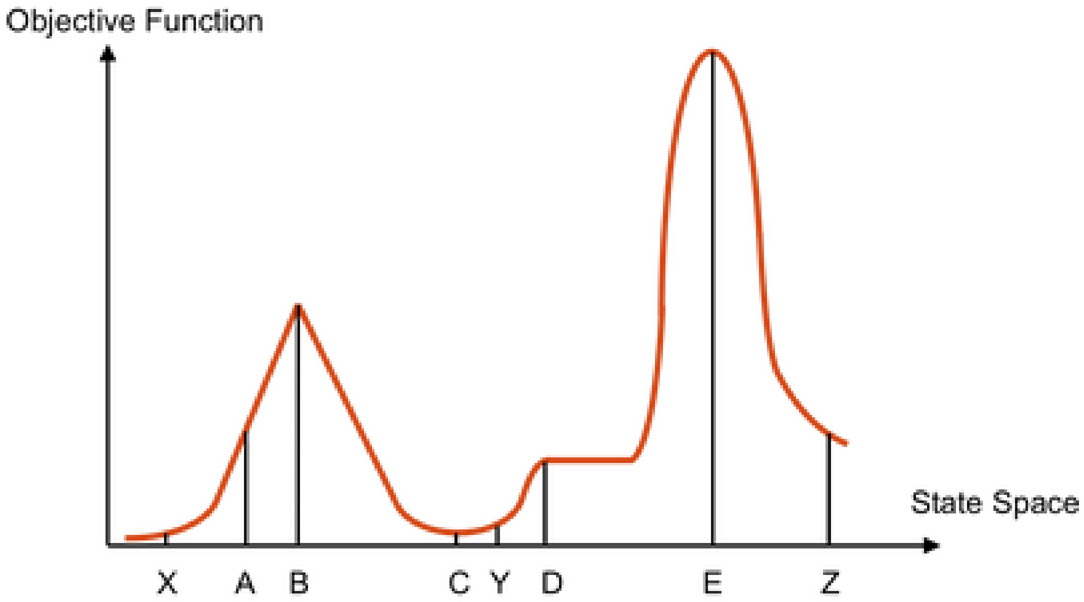
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If **no neighbors better** than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?
- What's good about it?

Hill Climbing Diagram





Hill-Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

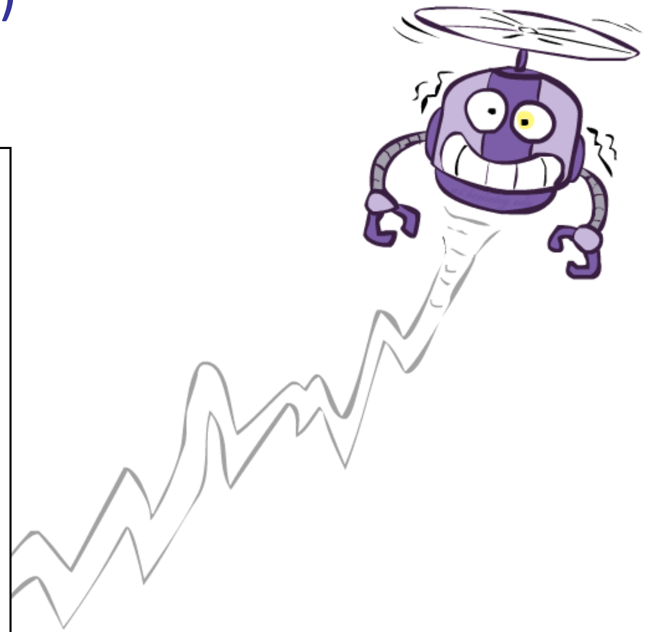
Starting from Z, where do you end up ?

Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves (demo)
  - But make them rarer as time goes on

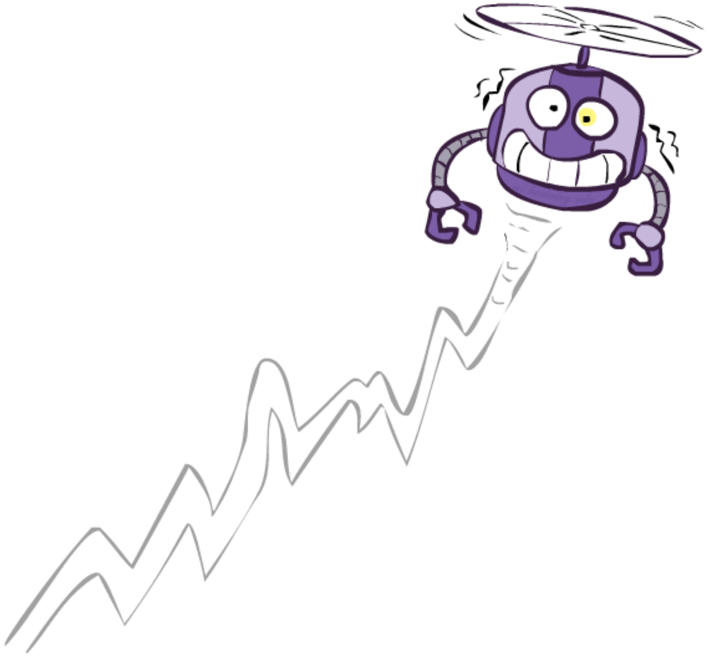
function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

  $current \leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow schedule[t]$
    if $T = 0$ **then return** *current*
    $next \leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** $current \leftarrow next$
    else $current \leftarrow next$ only with probability $e^{\Delta E/T}$



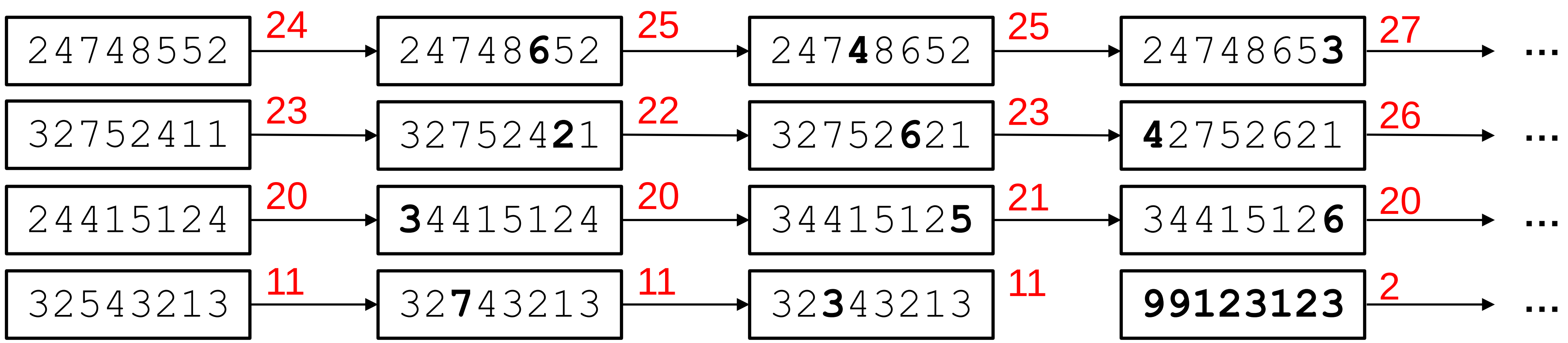
Simulated Annealing

- Theoretical guarantee:
  - If T decreased slowly enough, will converge to optimal state!
- Is this an interesting guarantee?
  - Kinda. Have to run algorithm forever to get guaranteed optimum.
- 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



Random Restarts

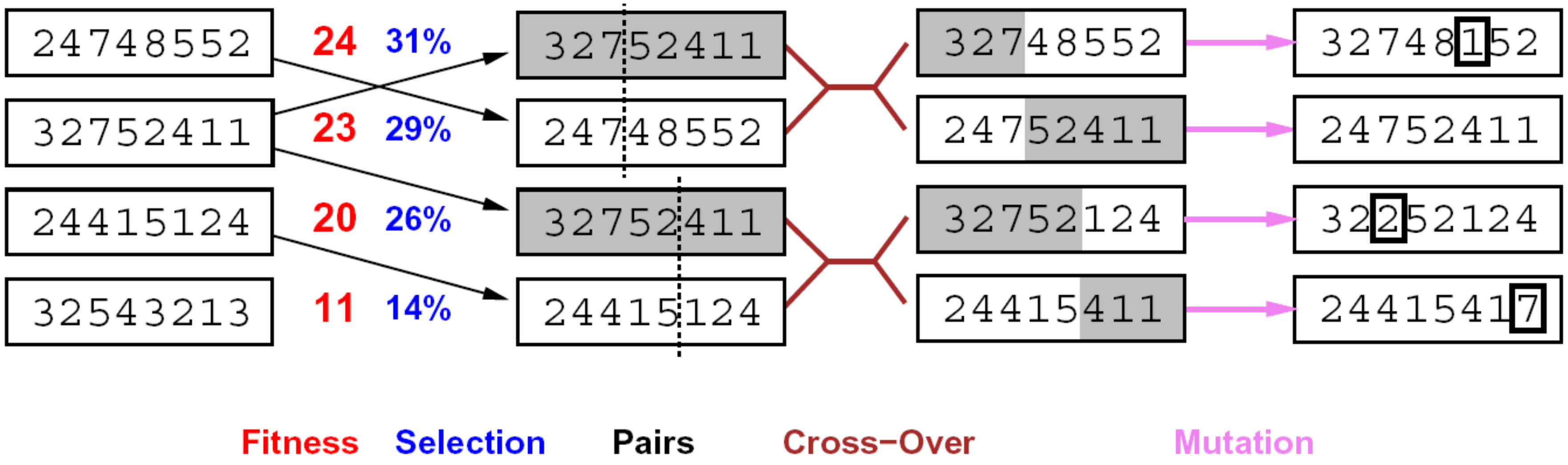
- Vanilla hill climbing and simulated annealing can be parallelized:
  - After running for a while, restart from a new random starting point.
  - Restart conditions include:
    - After running for Z total steps.
    - After spending Z units of time on a shoulder.
    - Anything else you can dream.
  - Can run many random searches in parallel.

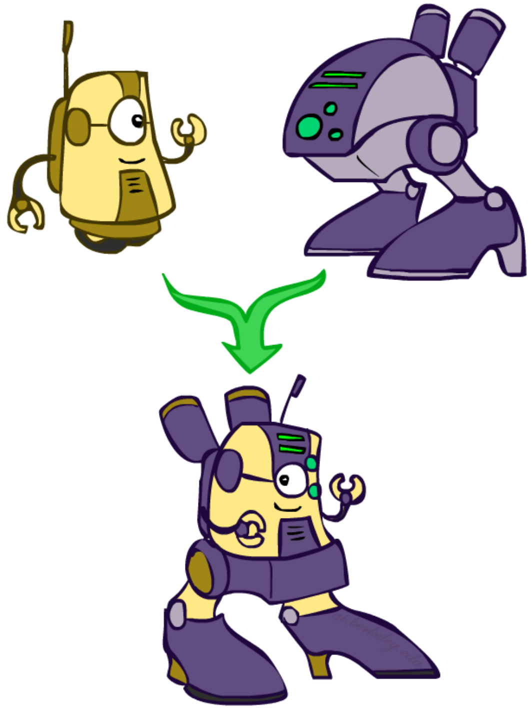


Number of constraints obeyed

4th search performs random restart.

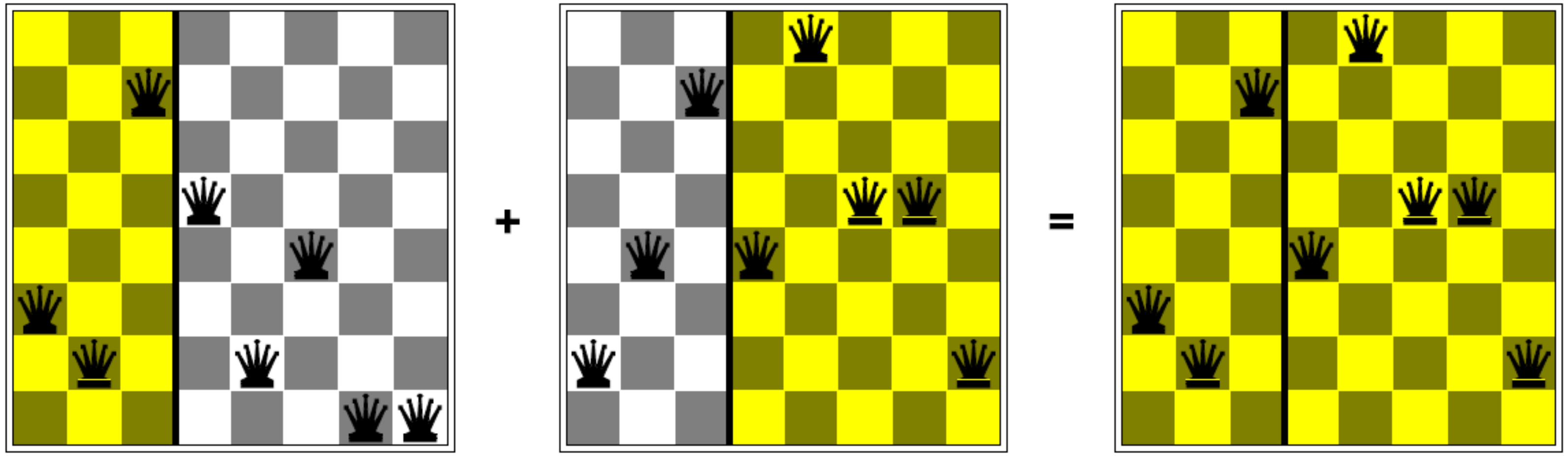
Genetic Algorithms





- 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
    - Instead of taking a step in your landscape, take a huge random leap
- 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?

Next Time: Adversarial Search!

- Example: Picking a move in chess by considering potential opponent's moves.