CS 188: Artificial Intelligence

Constraint Satisfaction Problems II
and Local Search

Instructors: Sergey Levine and Stuart Russell

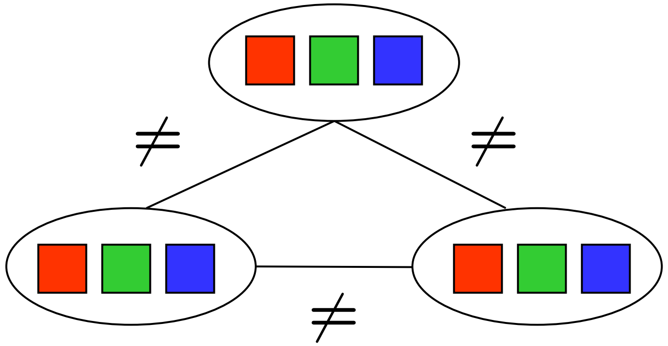
University of California, Berkeley

Today

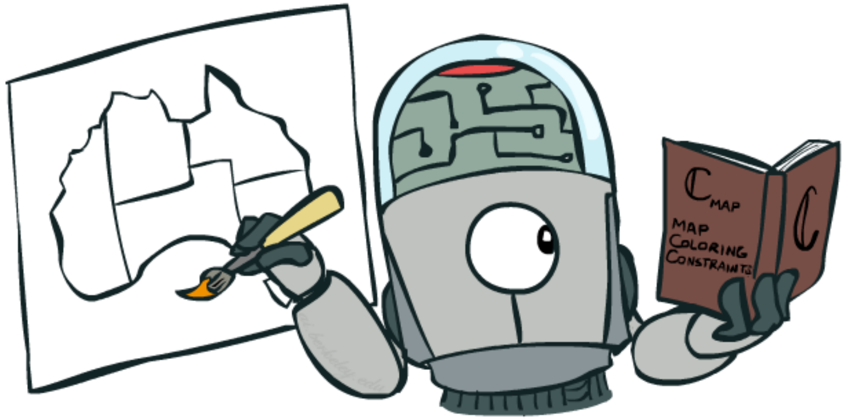
- Structure of CSPs
- Local Search

Reminder: CSPs

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

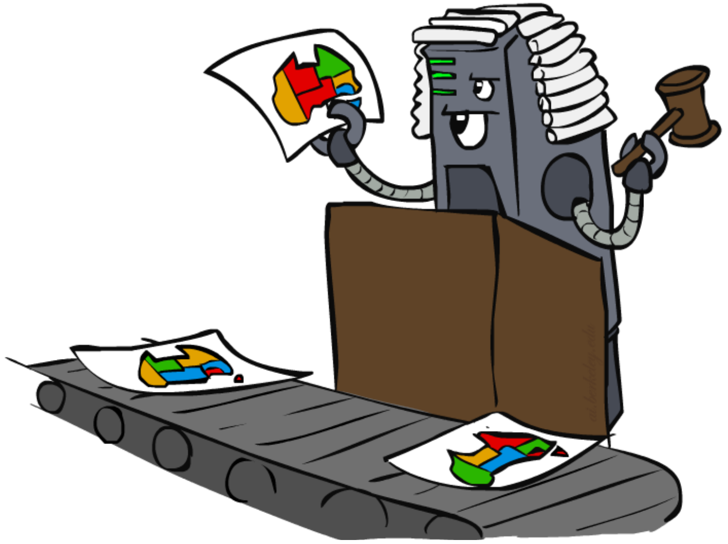


- Goals:
  - Here: find any solution
  - Also: find all, find best, etc.



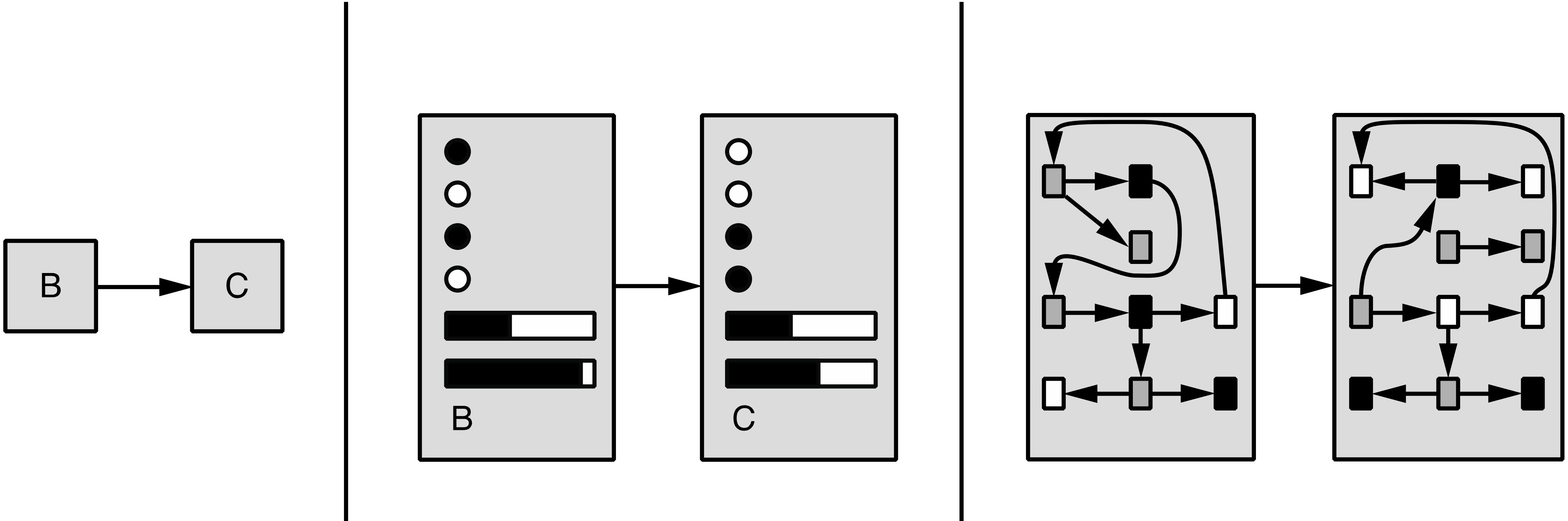
Standard Search Problems

- Standard search problems:
  - State is a ***black box***: arbitrary data structure
  - Goal test is a black box test on states
  - Actions are black box data structures
  - Transition model is a black box function



- Consequences:
  - Have to write new code for every new problem
  - Have to devise heuristics for each new problem
  - Cannot just ***choose actions that achieve the goal***!
- Solution: formal representation for states, actions, goals

Spectrum of representations



(a) Atomic (b) Factored (b) Structured

Search, game-playing

CSPs, planning, propositional logic, Bayes nets, neural nets

First-order logic, databases, probabilistic programs

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 $result \neq failure$ **then return** *result*
            remove $\{var = value\}$ from *assignment*
    return *failure*

Improving Backtracking

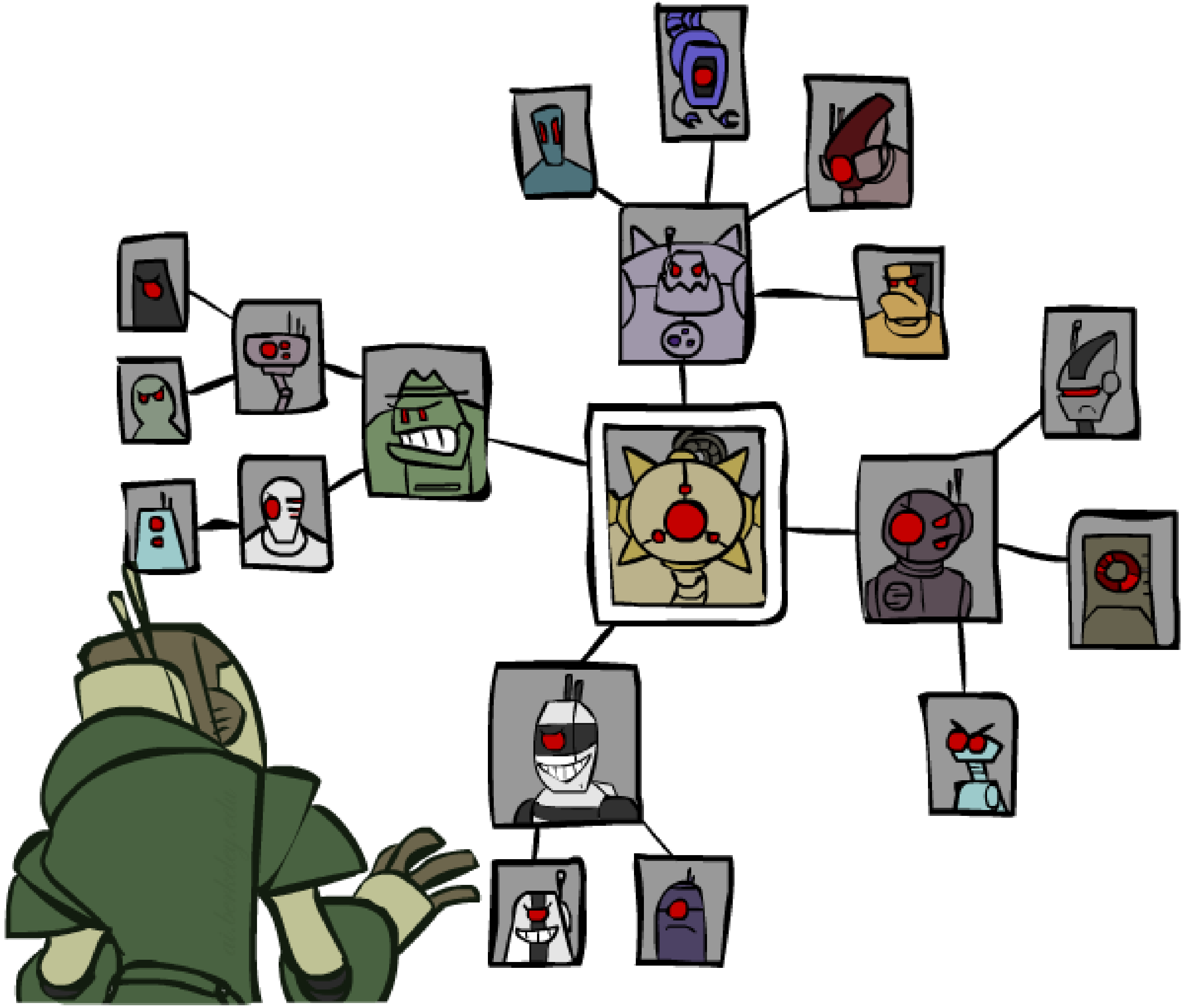
- General-purpose ideas give huge gains in speed
  - … but it's all still NP-hard

- Filtering: Can we detect inevitable failure early?

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

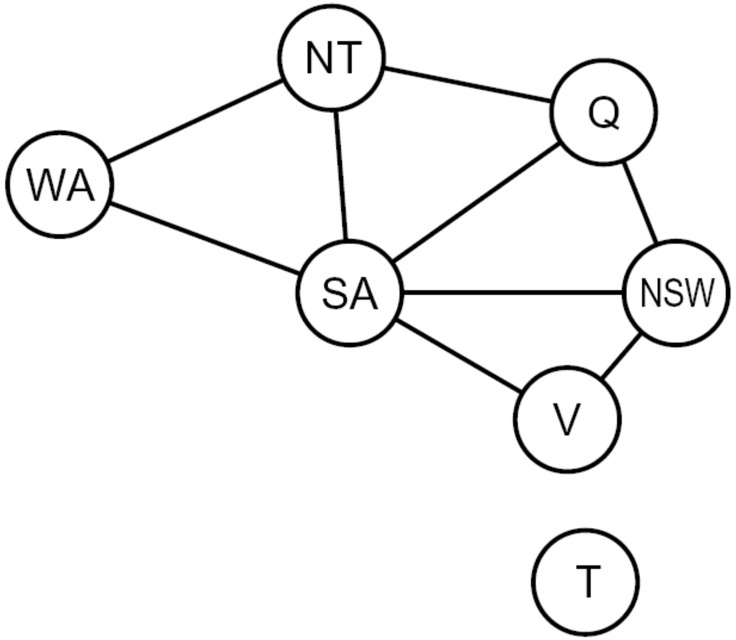
- Structure: Can we exploit the problem structure?

Structure

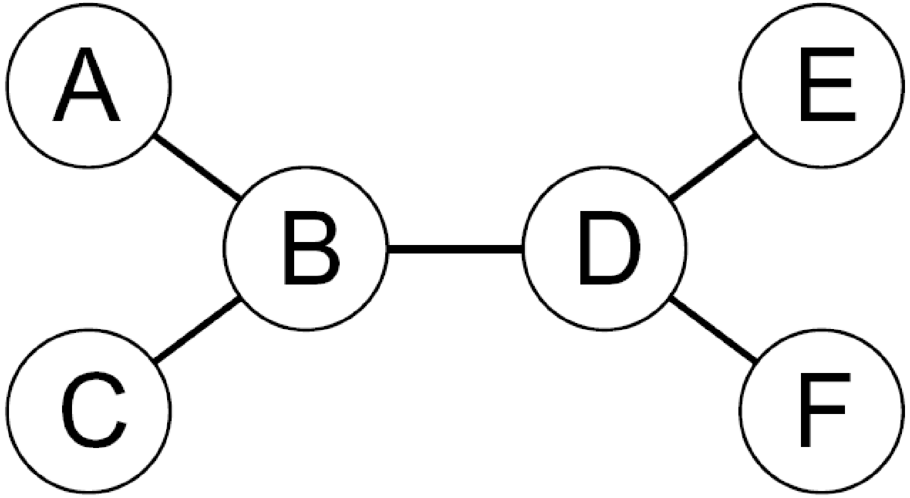


Problem Structure

- 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 each:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20, search 10 million nodes/sec
  - 2^{80} = **4 billion years**
  - $(4)(2^{20})$ = **0.4 seconds**



Tree-Structured CSPs

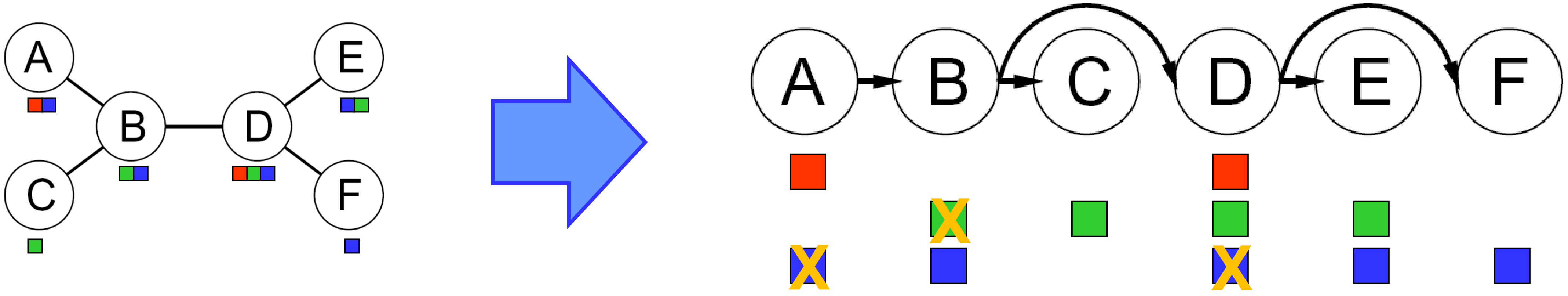


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\, d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning in Bayes nets (later): an example of the relation between structural properties and the complexity of reasoning

Tree-Structured CSPs

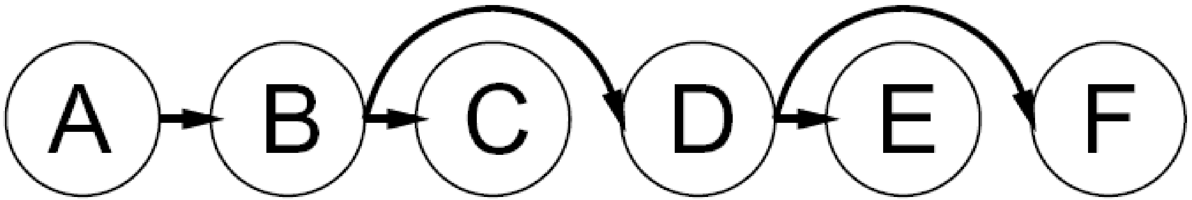
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - 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: $O(n\ d^2)$

Tree-Structured CSPs

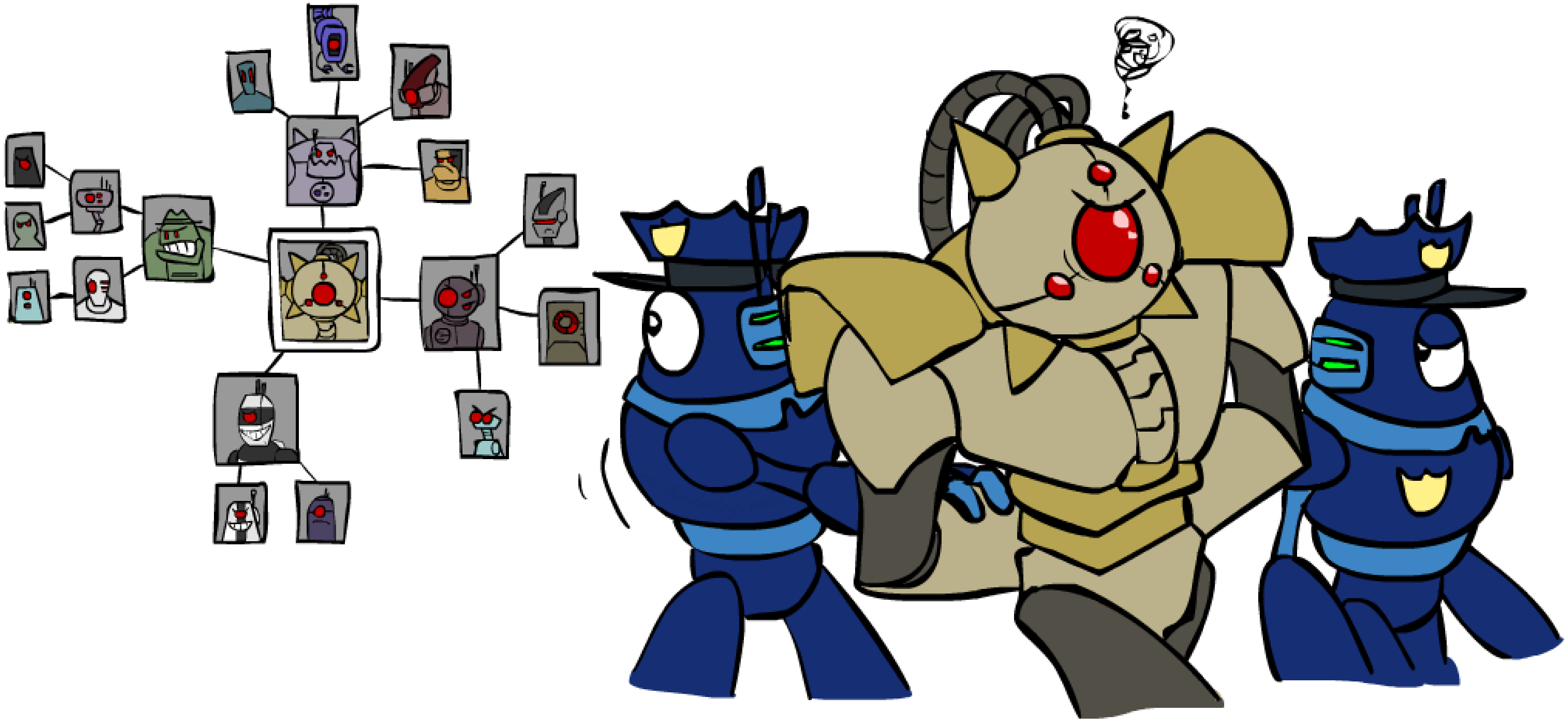
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



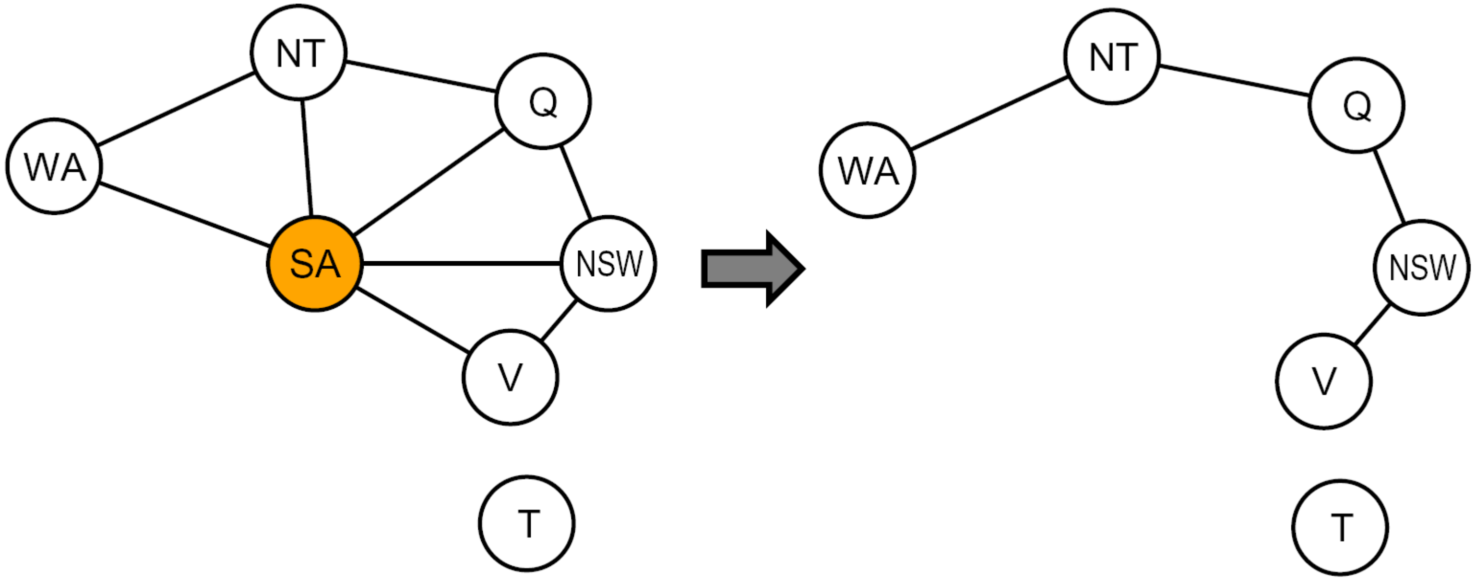
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

- 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 small c
  - E.g., 80 variables, c=10, 4 billion years -> 0.029 seconds

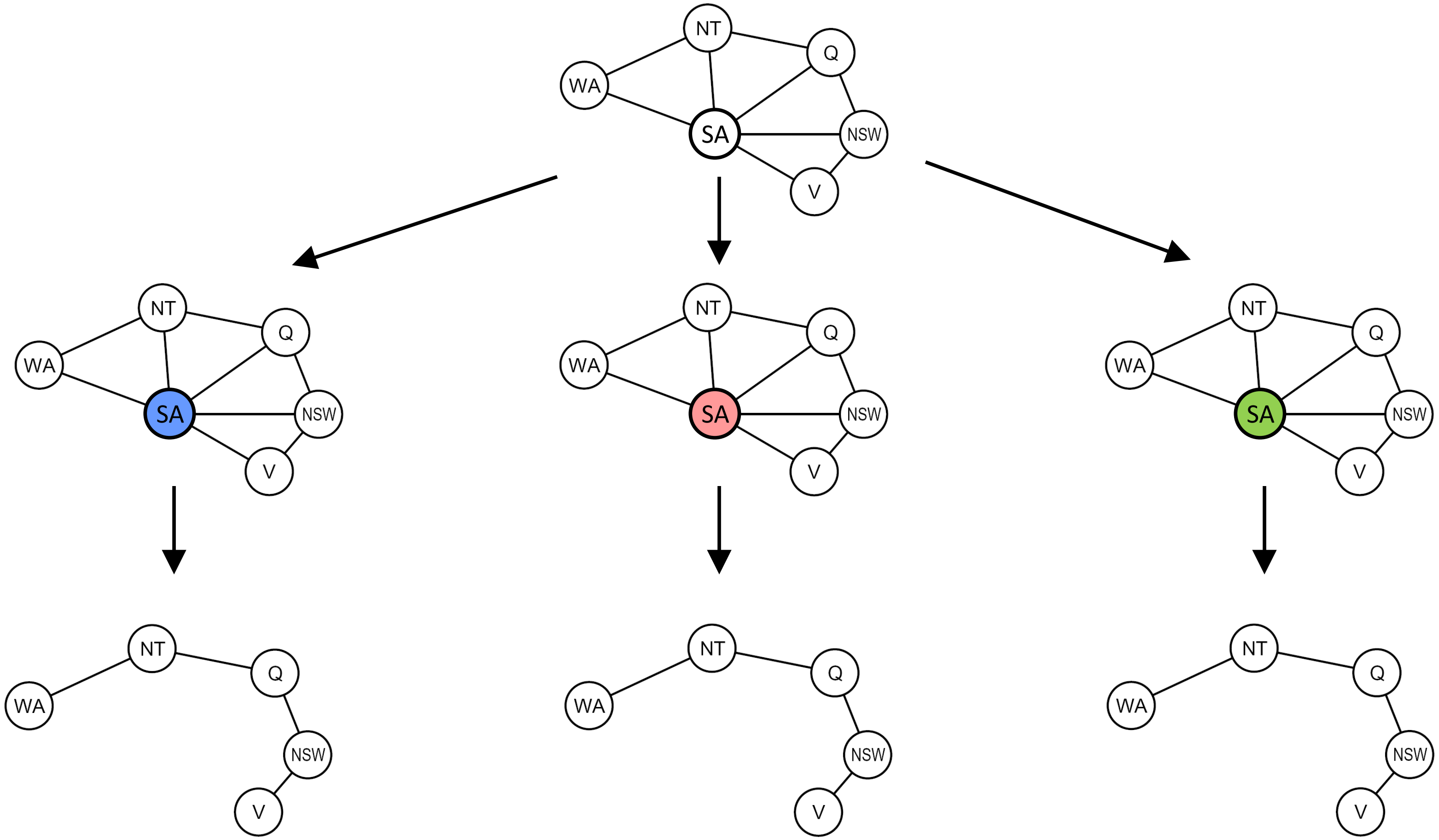
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

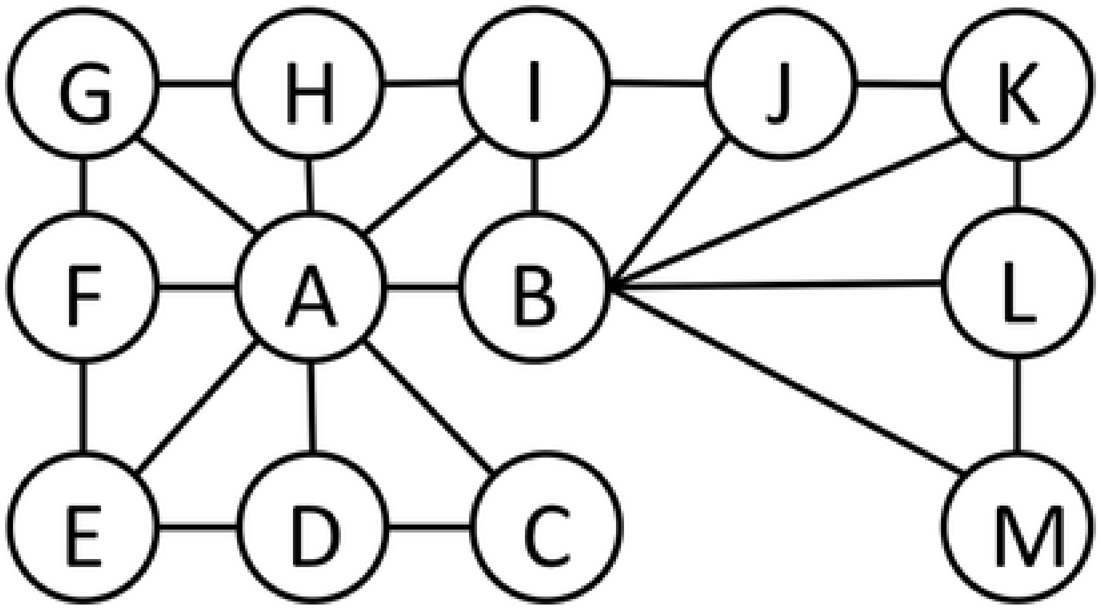
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



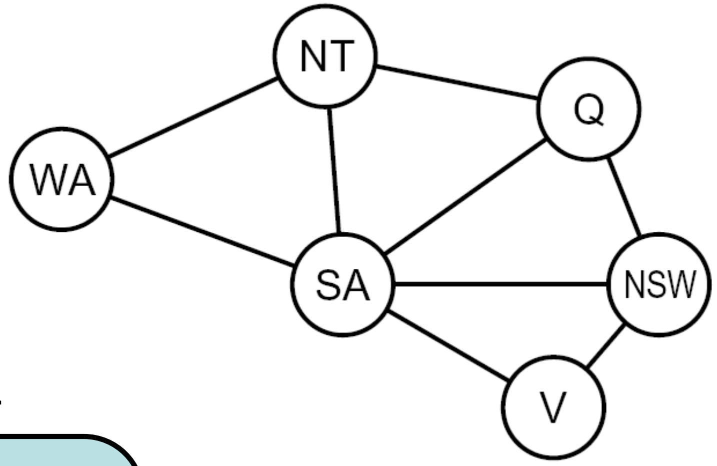
Cutset Quiz

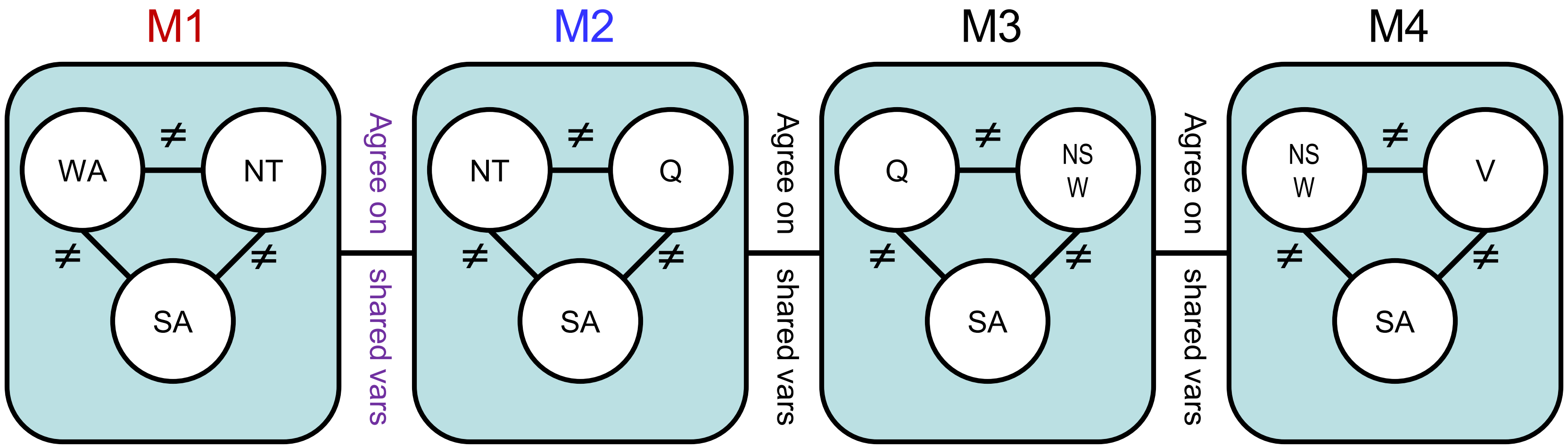
- Find the smallest cutset for the graph below.



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



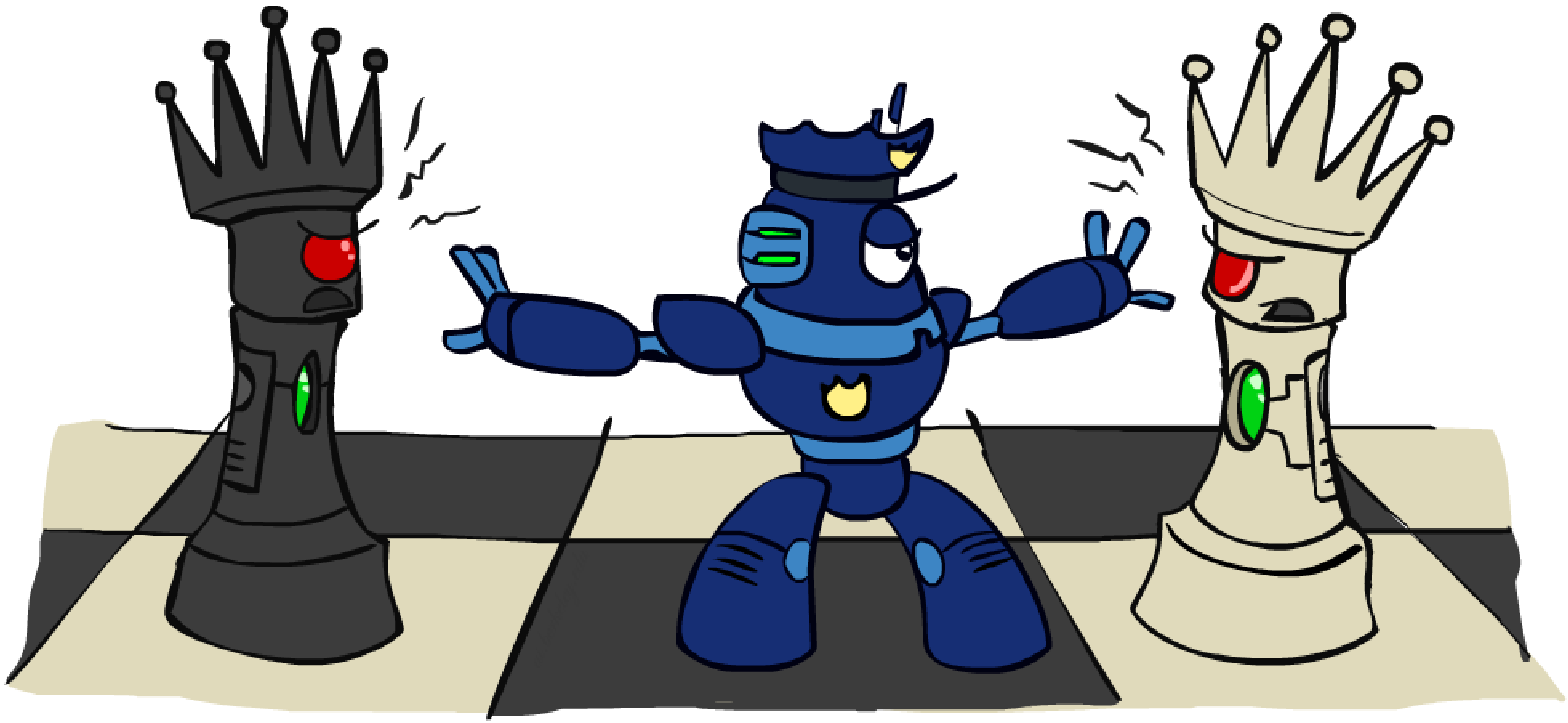


{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
…}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
…}

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

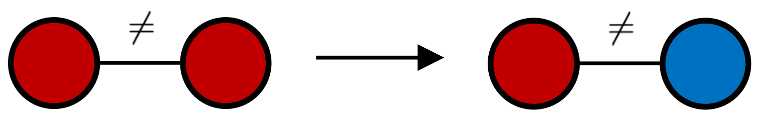
Iterative Improvement



Iterative Algorithms for CSPs

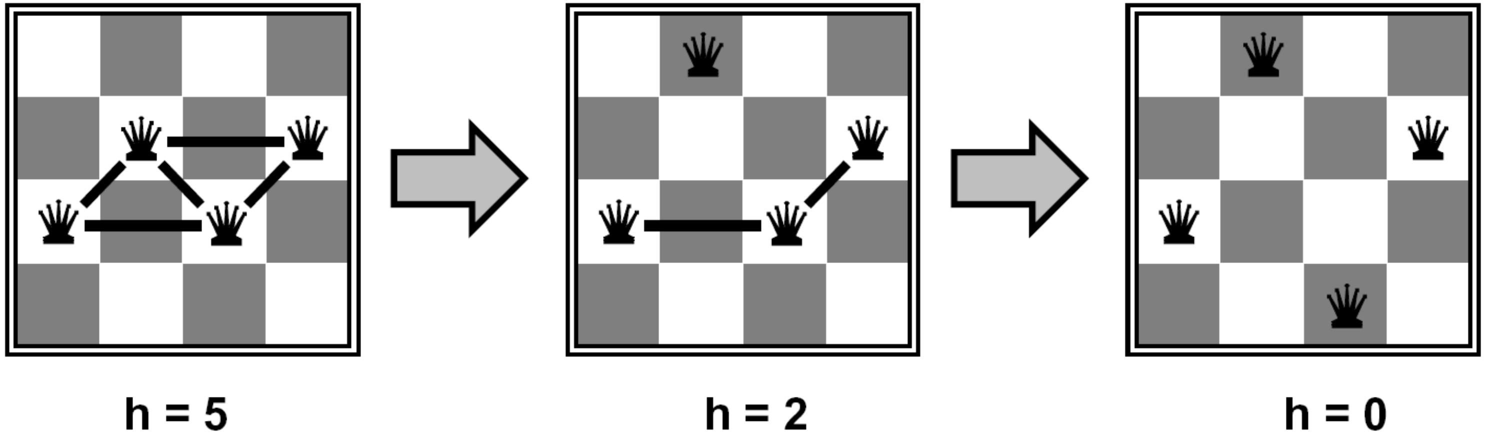
- Local search methods typically work with “complete” states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No tree, no fringe! “New age” algorithm



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest 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

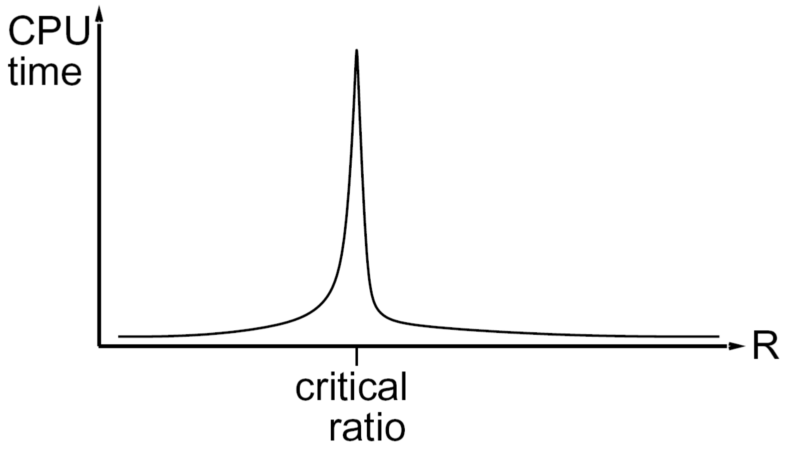
[Demo: coloring – iterative improvement]

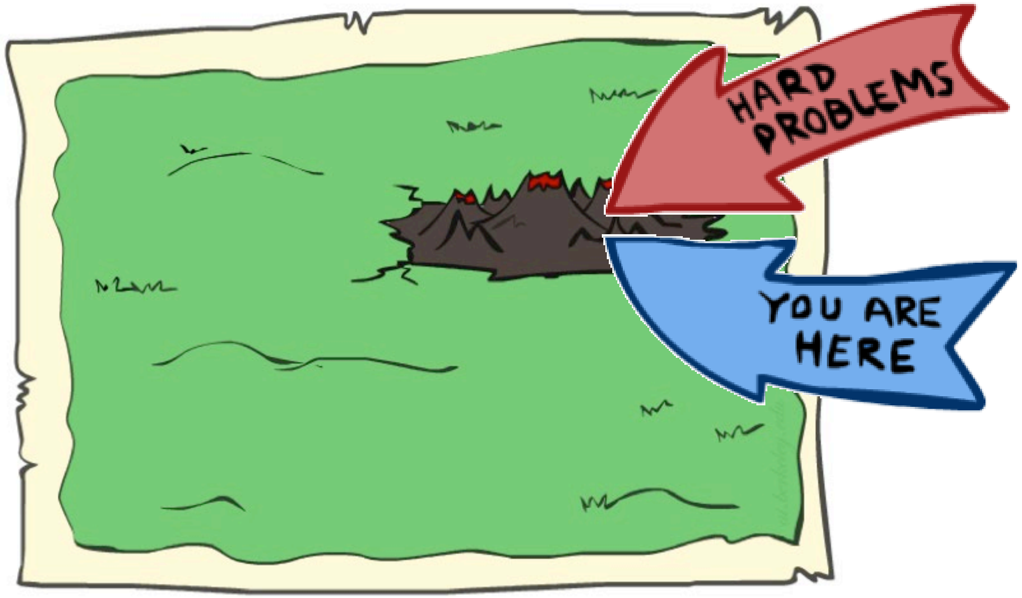
Performance of Min-Conflicts

- 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

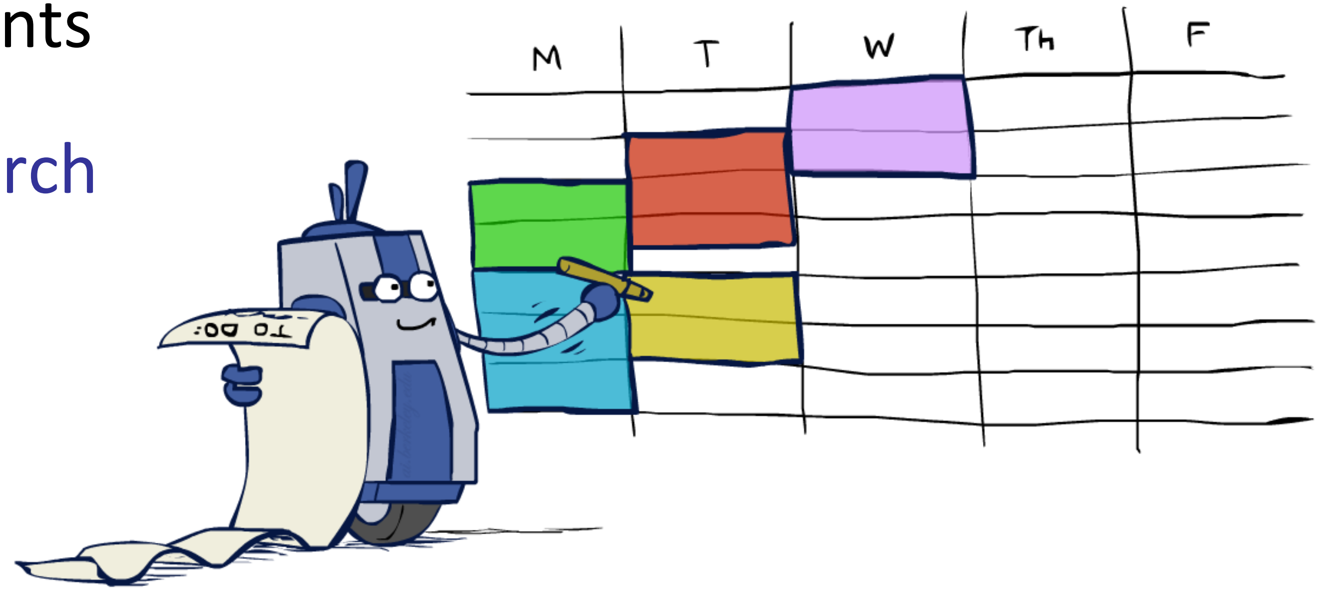
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





Summary: CSPs

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



Break quiz

Given a search problem P expressed in the usual way:

- initial state s_0, states S, actions A, goal test G, transition model $Result(s,a)$

and a time horizon T, construct a CSP C such that C has a solution exactly when P has a solution of length T, and the solution to P can be read off from the solution to C

Hint: You'll need some variables for each time step, including A_t (the action taken at time t). What are the constraints between time steps? Other constraints on particular time steps?

Break quiz answer

Variables of the CSP are

- Action variables $A_0, \ldots, A_{T-1}$ each with domain A
- State variables $S_0, \ldots, S_T$, each with domain S

Constraints of the CSP are

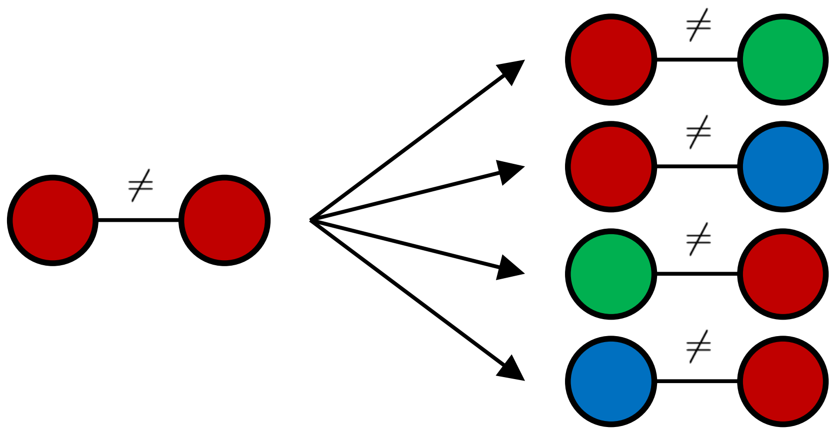
- $S_0 = s_0$
- S_T satisfies goal test G
- For $t = 0, \ldots, T-1$, $S_{t+1} = Result(S_t, A_t)$

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
- New successor function: local changes



- Generally much faster and more memory efficient (but incomplete and suboptimal)
- Pretty much unavoidable when the state is "yourself"

Hill Climbing

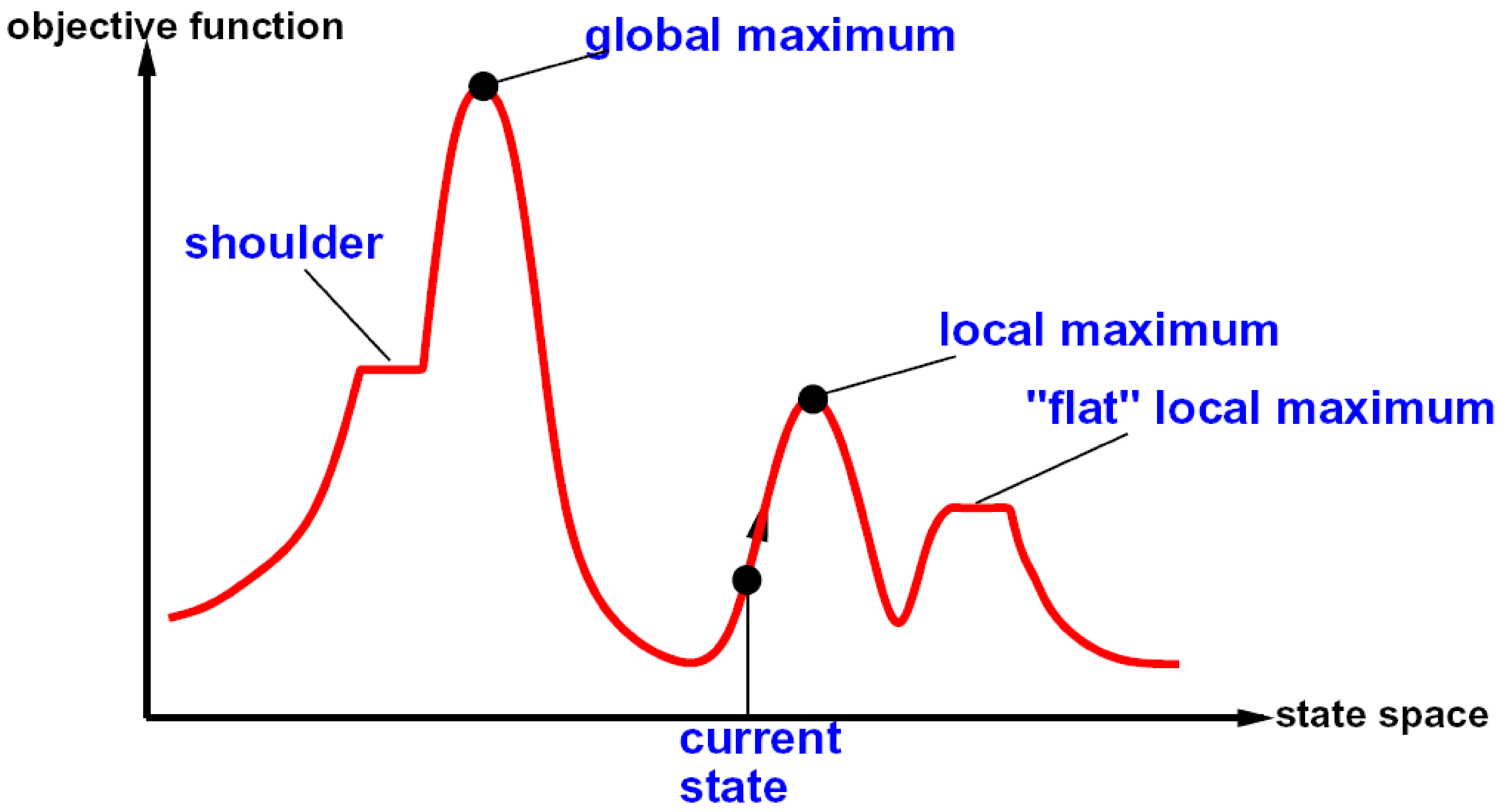
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

Hill Climbing

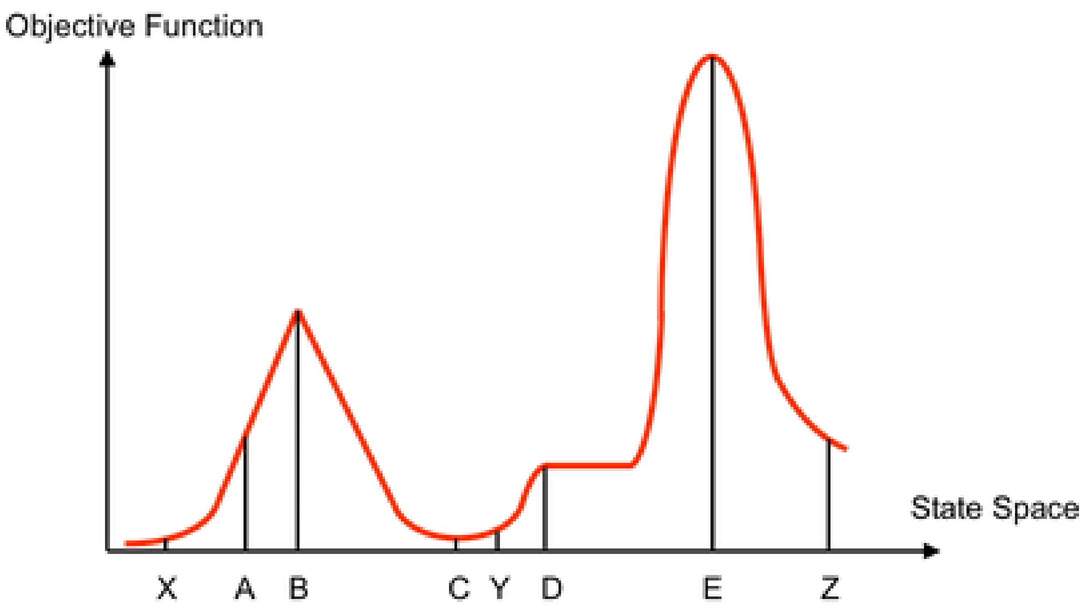
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

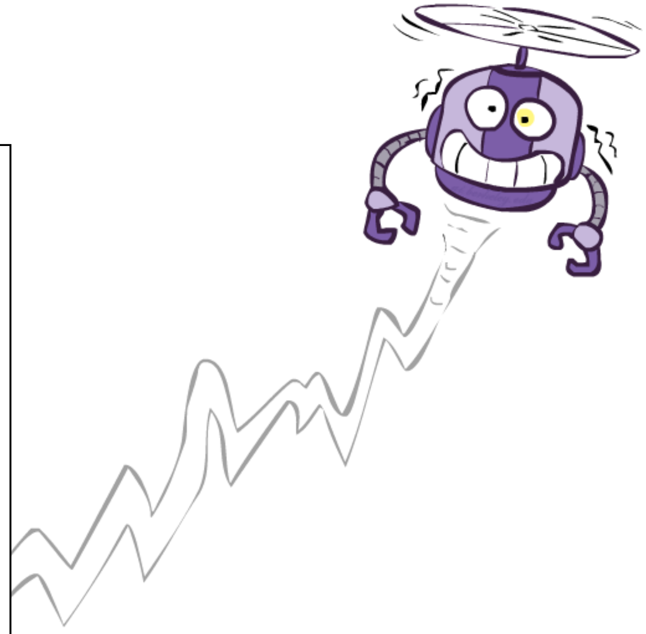
Starting from Z, where do you end up ?

Simulated Annealing

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

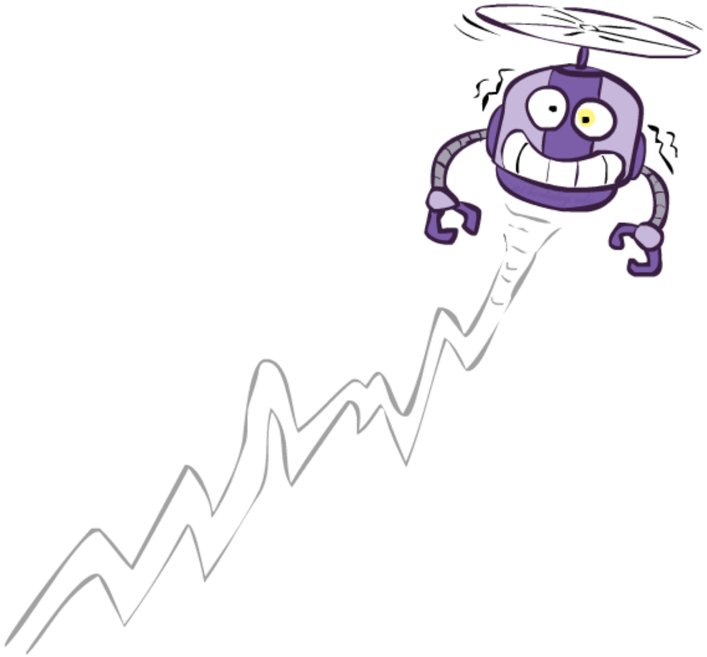
function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

  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

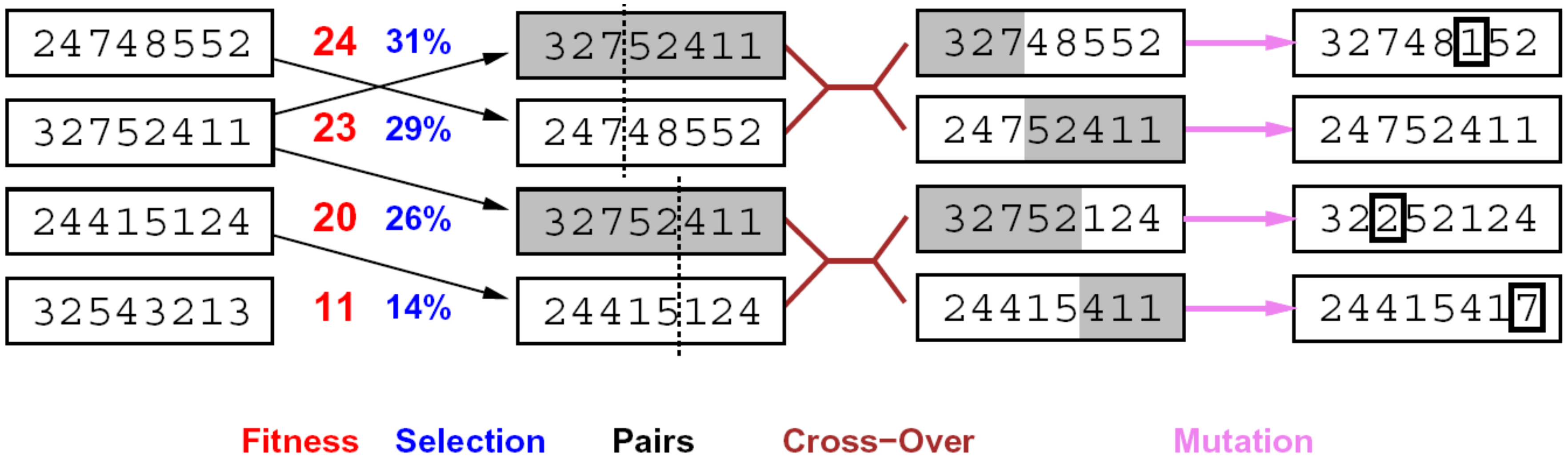
- Theoretical guarantee:
  - Stationary distribution (Boltzmann): $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough,
    will converge to optimal state!

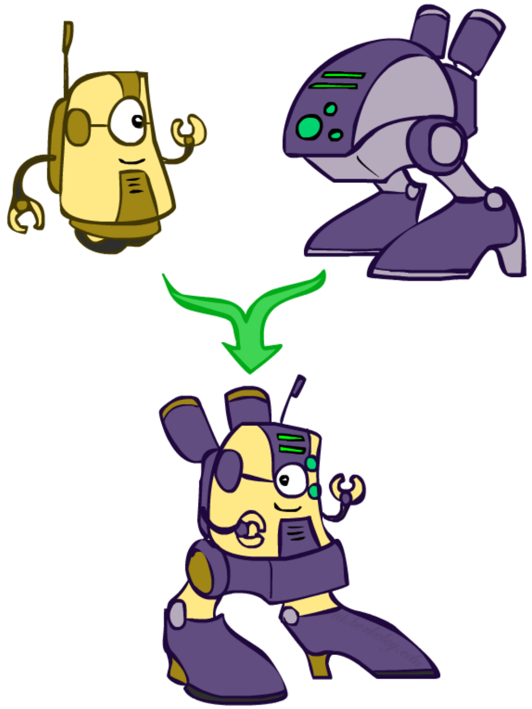


- Is this an interesting guarantee?

- 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
  - “Slowly enough” may mean exponentially slowly
  - Random restart hillclimbing also converges to optimal state…

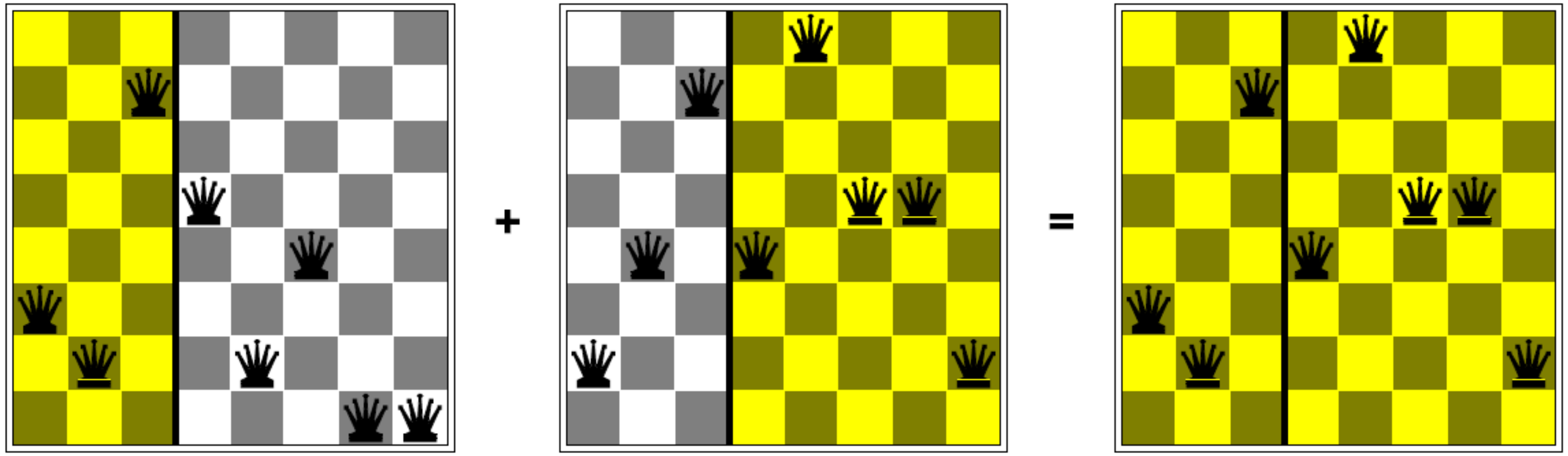
Genetic Algorithms





- 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

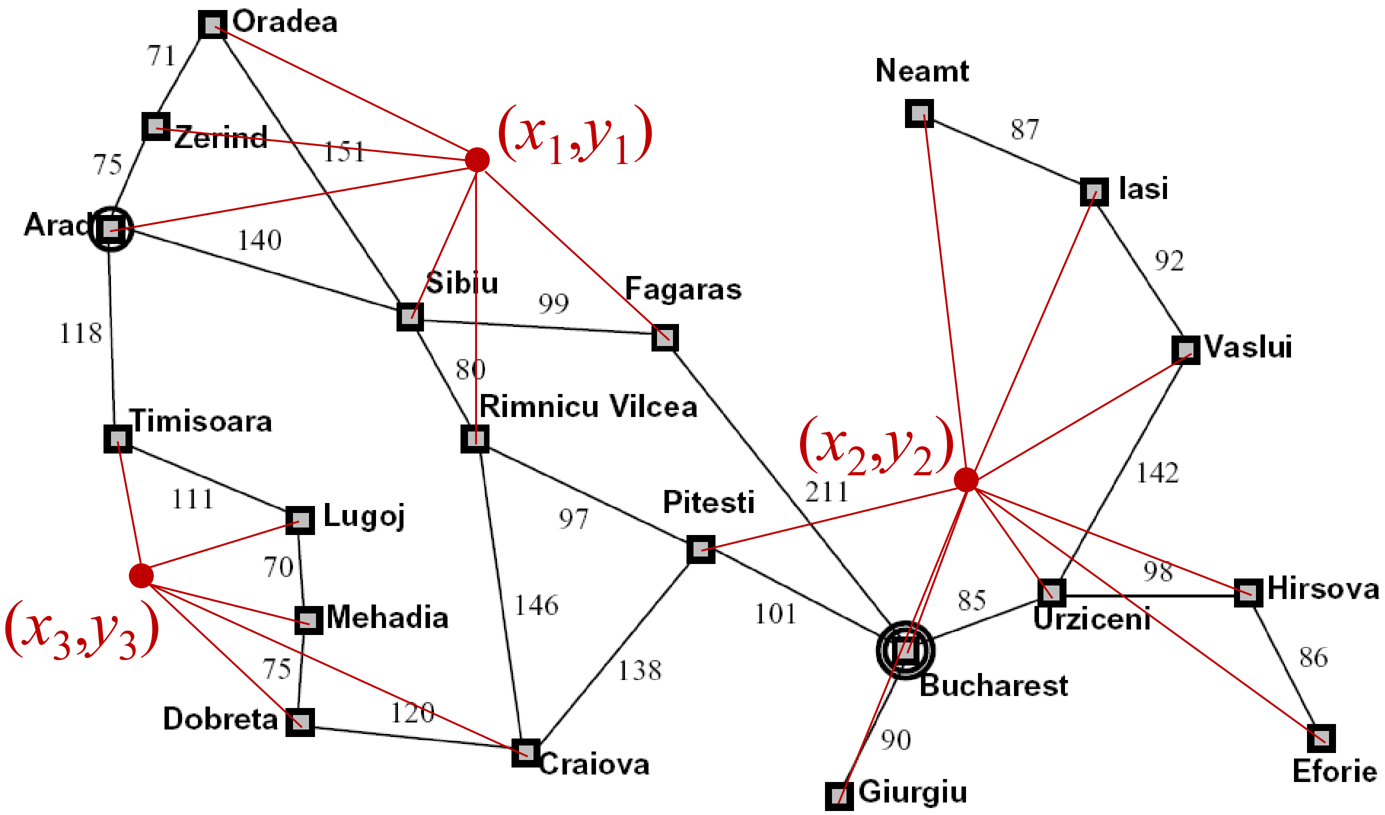


- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?

Local Search in Continuous Spaces

- Put 3 airports in Romania to minimize the sum of squared distance of each city to its nearest airport
- Variables: x_1,y_1,x_2,y_2,x_3,y_3
- C_i = set of cities nearest to i
- Cost $f(x_1,y_1,x_2,y_2,x_3,y_3)$ =

$$\sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

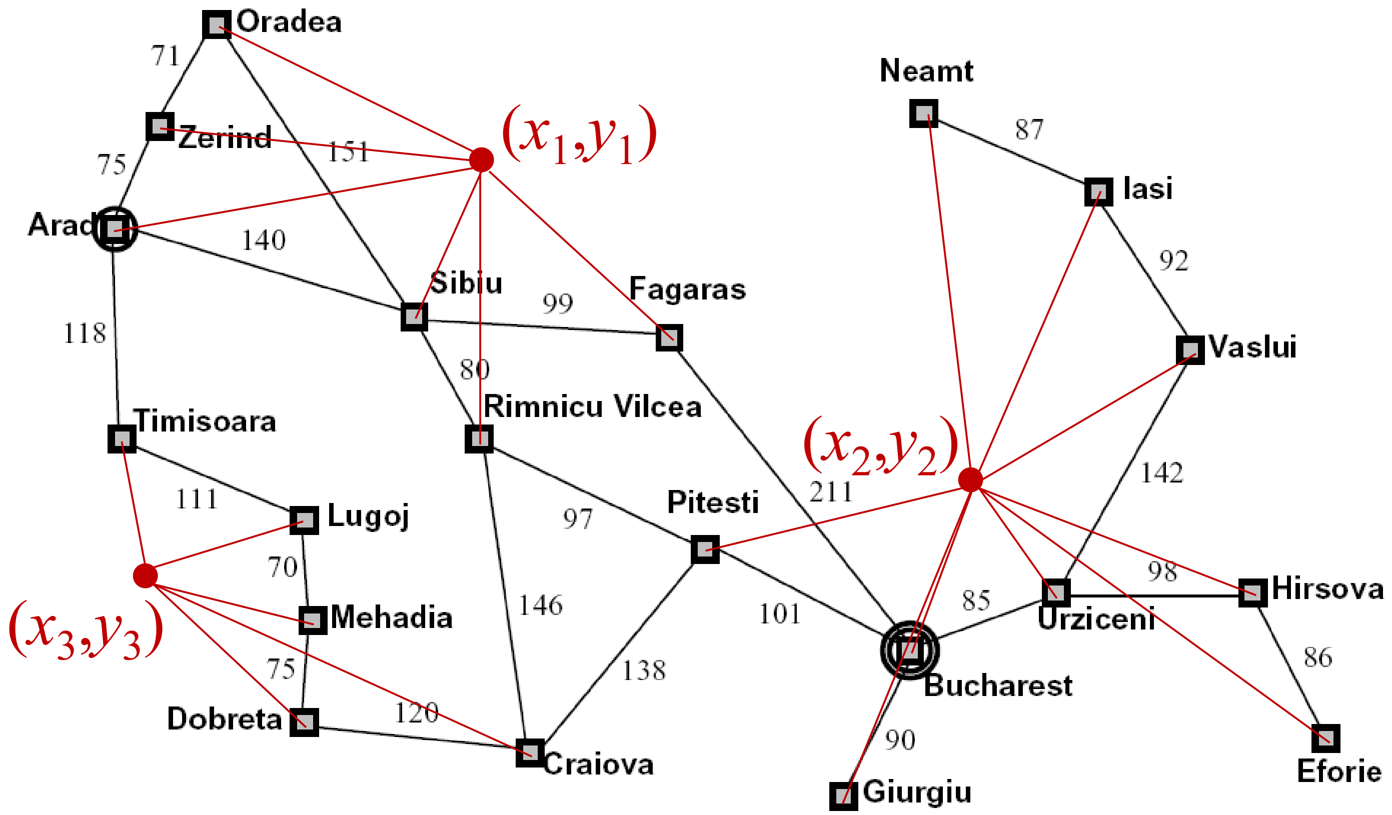


Local Search in Continuous Spaces

- Cost $f(x_1,y_1,x_2,y_2,x_3,y_3) =$

$$\sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

- Method 1: discretize, compute *empirical gradient* $f(x_1+dx,y_1,x_2,y_2,x_3,y_3)$ etc.
- Method 2: stochastic descent: generate small random vector d**x** and accept if f(**x**+d**x**) < f(**x**)



Local Search in Continuous Spaces

- Cost $f(x_1,y_1,x_2,y_2,x_3,y_3) =$

$$\sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

- Method 3: take small step along gradient vector

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c)$$