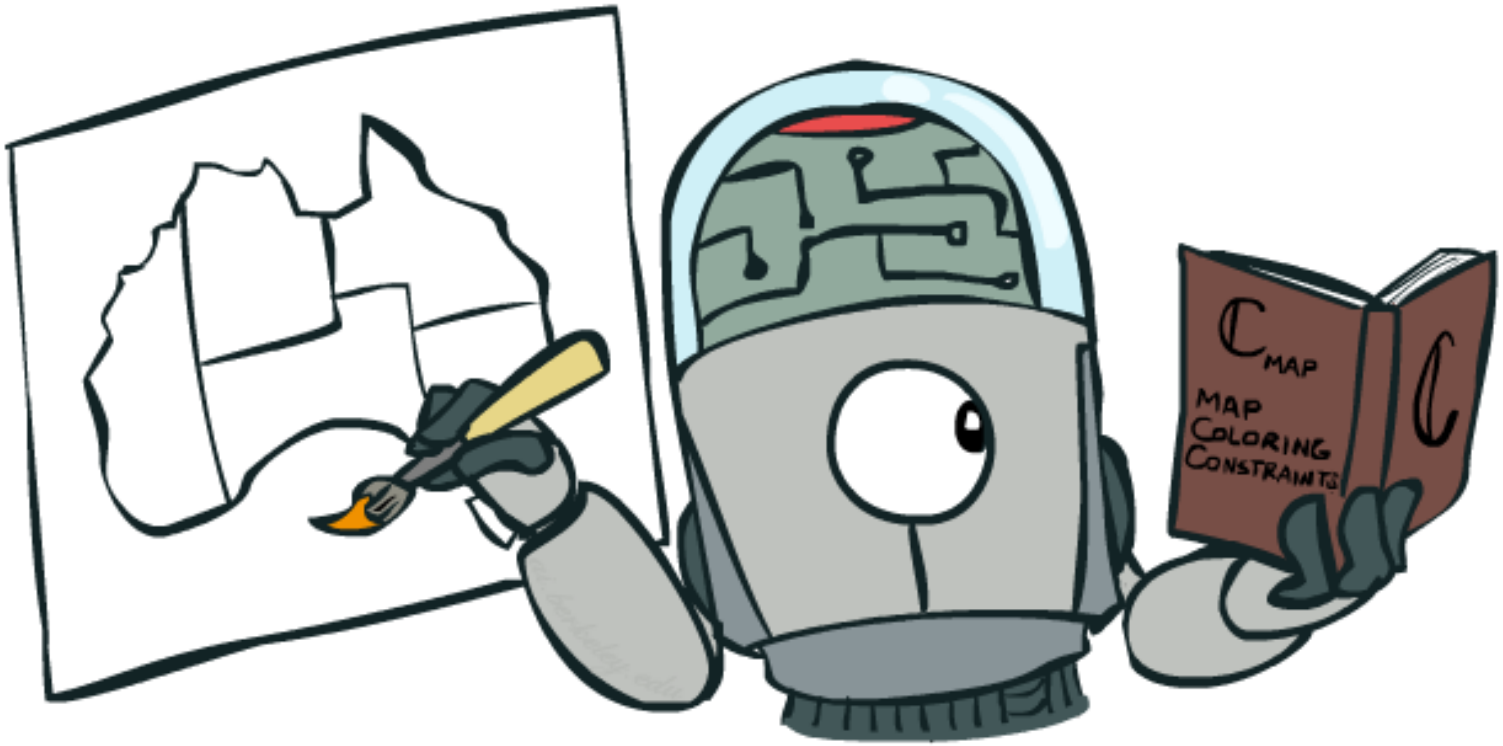
复旦大学大数据学院 魏忠钰
School of Data Science, Fudan University

Constraint Satisfaction Problems

April 18th, 2018

Constraint Satisfaction Problems



Constraint Satisfaction Problems

- Planing: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs (e.g. depths)
  - Heuristics give problem-specific guidance

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - Constraint Satisfaction Problems (CSPs) are specialized for identification problems

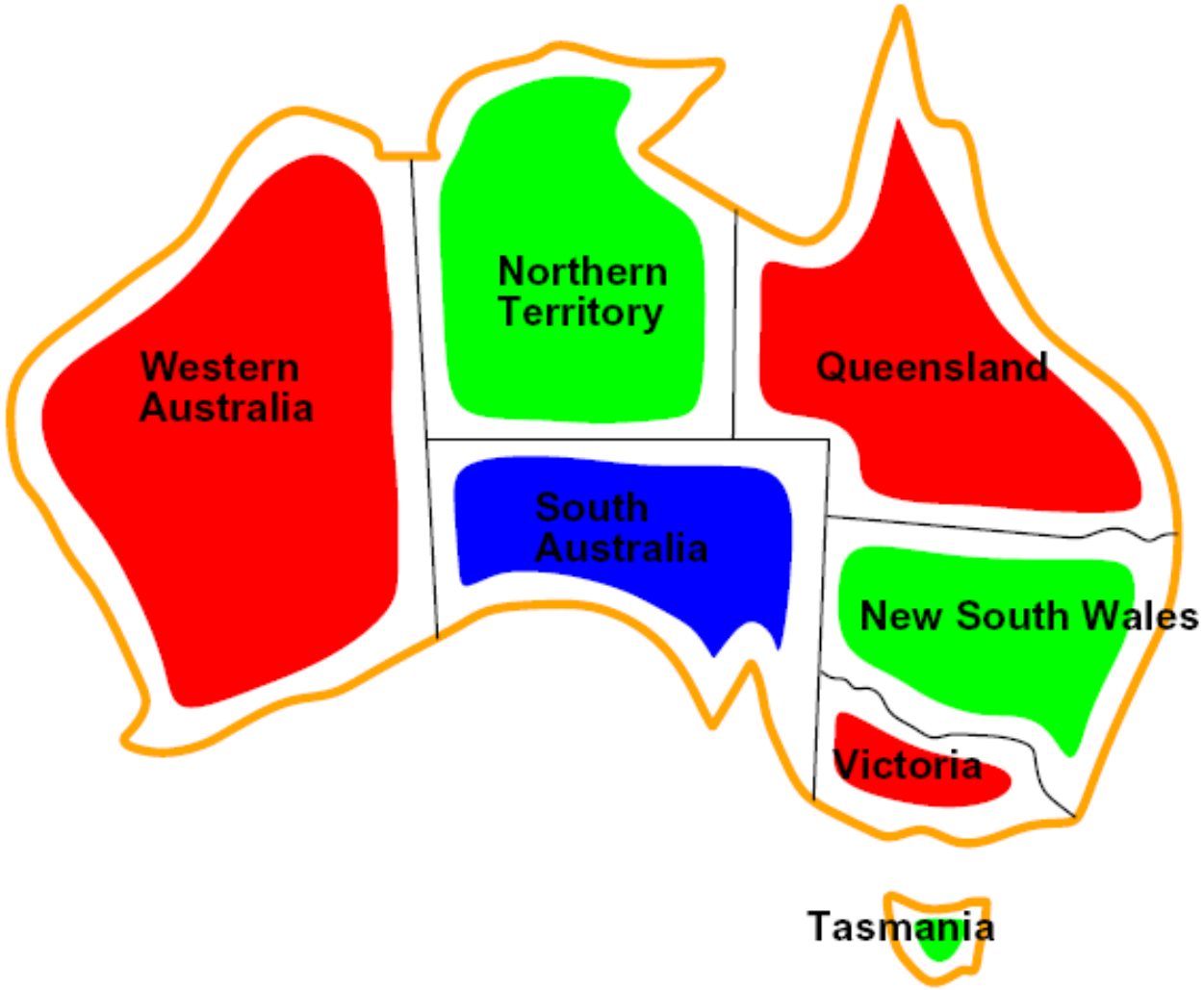
Constraint Satisfaction Problem

- **Standard search problems:**
  - **State is a "black box": arbitrary data structure**
  - **Goal test can be any function over states**
- **Constraint satisfaction problems (CSPs):**
  - **State is represented as a feature vector**
  - **Goal test is a set of constraints to identify the allowable feature vector**

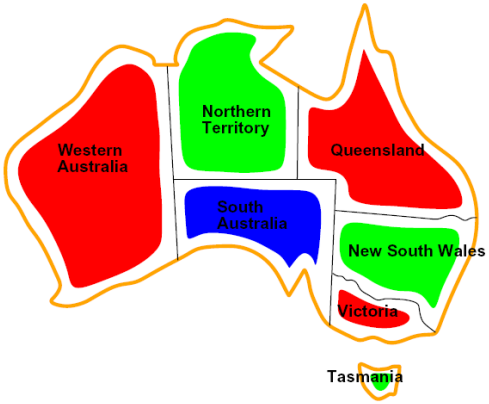
Feature Vector

- We have
  - A set of k variables (or features)
  - Each variable has a domain of different values.
  - A state is specified by an assignment of a value for each variable.
    - height = {short, average, tall},
    - weight = {light, average, heavy}
  - A partial state is specified by an assignment of a value to some of the variables.

CSP Examples



Example: Map Coloring



- Variables: $\mathrm{WA},\ \mathrm{NT},\ \mathrm{Q},\ \mathrm{NSW},\ \mathrm{V},\ \mathrm{SA},\ \mathrm{T}$
- Domains: $D = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$
- Constraints: adjacent regions must have different colors

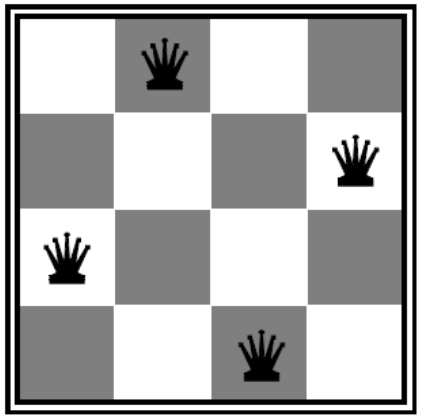
  Implicit: $\mathrm{WA} \neq \mathrm{NT}$

  Explicit: $(\mathrm{WA}, \mathrm{NT}) \in \{(\mathrm{red}, \mathrm{green}), (\mathrm{red}, \mathrm{blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:

  $\{\mathrm{WA}{=}\mathrm{red},\ \mathrm{NT}{=}\mathrm{green},\ \mathrm{Q}{=}\mathrm{red},\ \mathrm{NSW}{=}\mathrm{green},$
  $\mathrm{V}{=}\mathrm{red},\ \mathrm{SA}{=}\mathrm{blue},\ \mathrm{T}{=}\mathrm{green}\}$

Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints

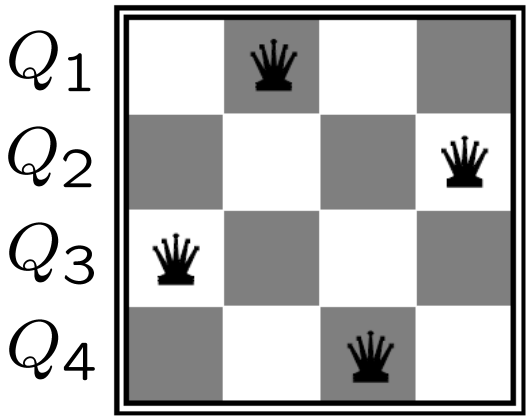


$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

Example: N-Queens

- Formulation 2:
- Variables: Q_k
- Domains: $\{1, 2, 3, \ldots N\}$



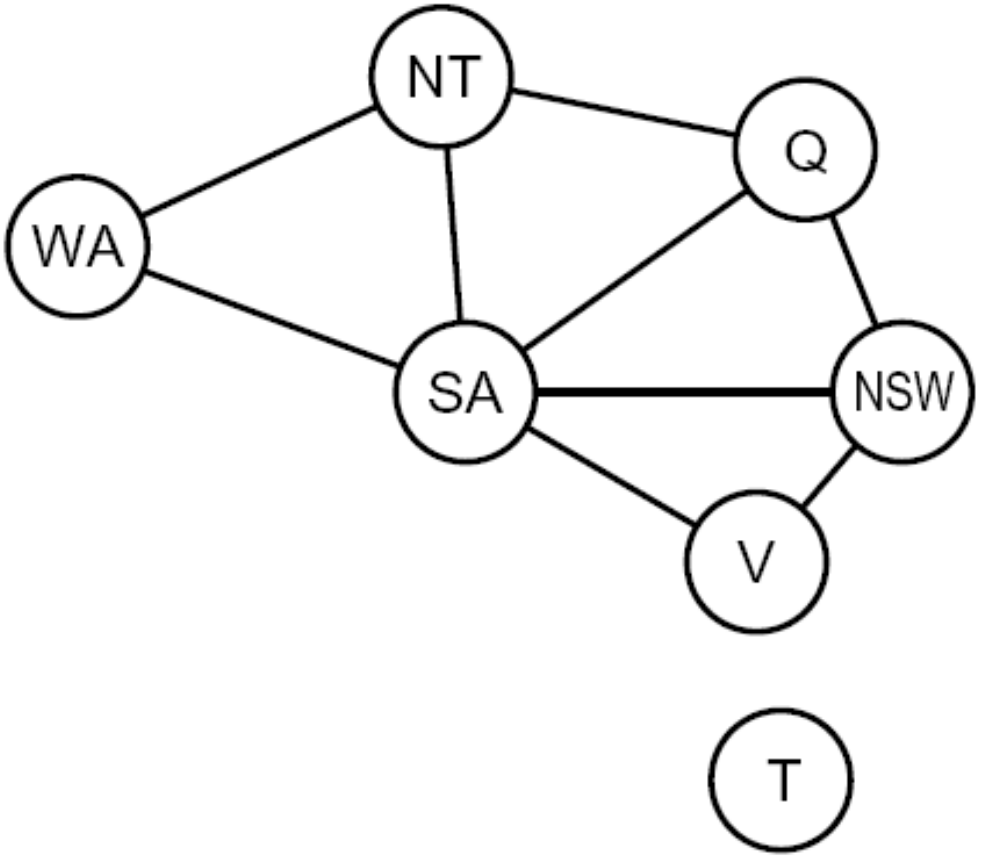
- Constraints:

Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$

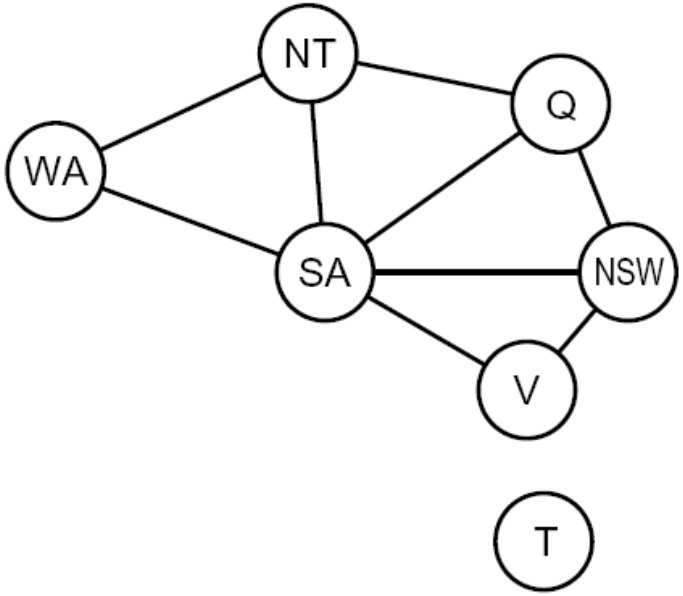
$\ldots$

Constraint Graphs



Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints



Example: Cryptarithmetic

- Variables:

$$F \;\; T \;\; U \;\; W \;\; R \;\; O \;\; X_1 \;\; X_2 \;\; X_3$$

- Domains:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

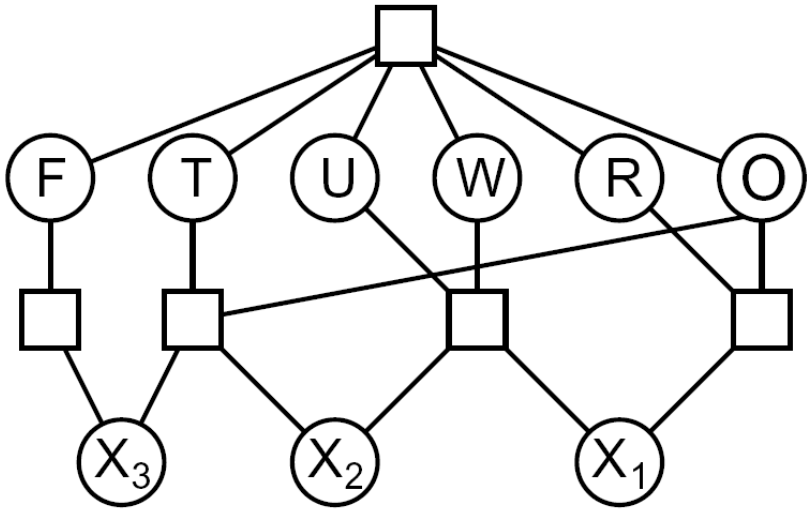
- Constraints:

$$\text{alldiff}(F, T, U, W, R, O)$$

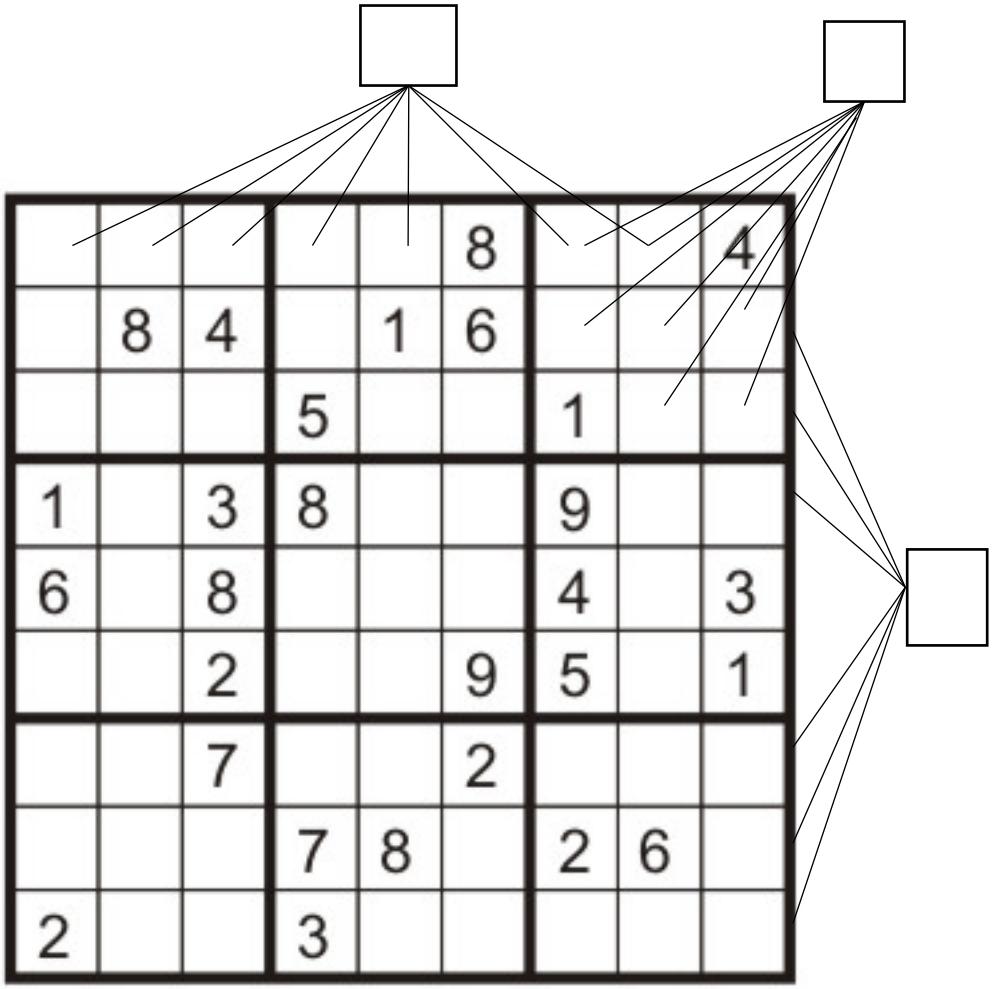
$$O + O = R + 10 \cdot X_1$$

$$\cdots$$

$$\begin{array}{r} \text{T W O} \\ +\ \text{T W O} \\ \hline \text{F O U R} \end{array}$$



Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Transform N-nary CSPs to Binary constraints one

- Show how a single ternary constraint such as A+B = C can be turned into 3 binary constraints by using auxiliary variables. You may assume finite domains. Next show how constraints with more than 3 variables can be treated similarly.
- Step 1: Introduce a new variable Z.
  - Domain is the set of all possible 3-tuples satisfying A+B=C. {<a_1, b_1, c_1>, <a_2, b_2, c_2>, …}
- Step 2: Create new constraints between Z and three original variables respectively (the position in the tuple is linked to an original variable).
  - $< Z, A > \in \{((< a1, b1, c1 >, a1), \dots \}$
  - $< Z, B > \in \{((< a1, b1, c1 >, b1), \dots \}$
  - $< Z, C > \in \{((< a1, b1, c1 >, c1), \dots \}$

Varieties of CSPs and Constraints

Varieties of CSPs

- Discrete Variables
  - Finite domains
    - Size d means $\mathrm{O}(d^n)$ complete assignments
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job

- Continuous variables
  - A + B = C (A, B, C are real numbers)

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable :

    $$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

    $$SA \neq WA$$

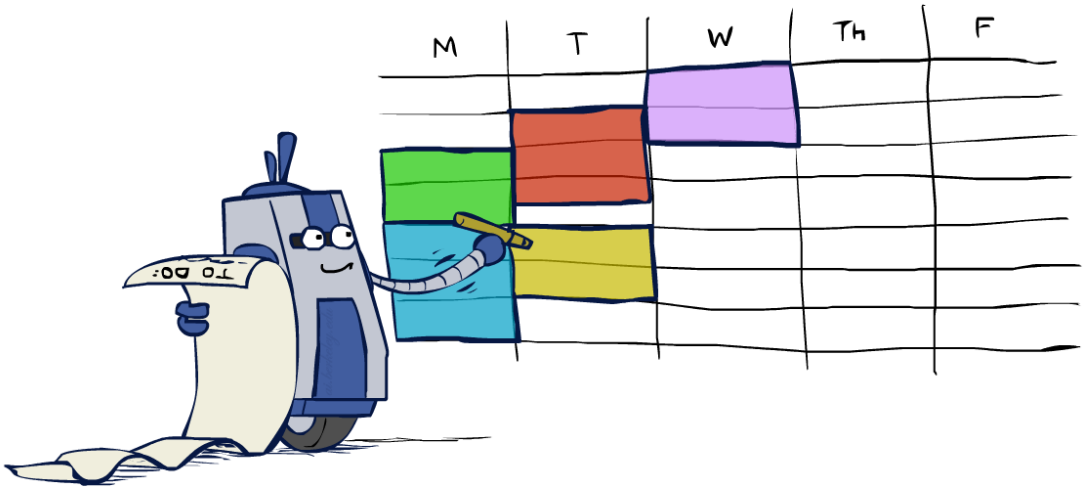
  - Higher-order constraints involve 3 or more variables: e.g., cryptarithmetic column constraints

  - Global Containts, e.g.: AllDiff()

- Preferences (soft constraints):
  - i.e. apple is better than banana
  - Often representable by a cost for each variable assignment

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



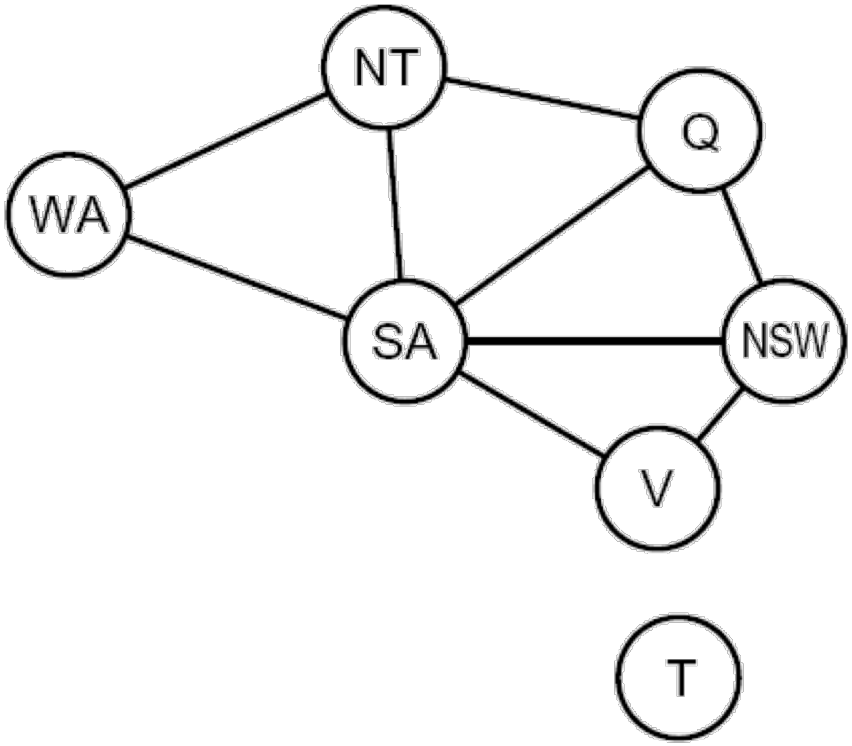
Solving CSPs

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'll start with the straightforward, naïve approach, then improve it

Search Methods

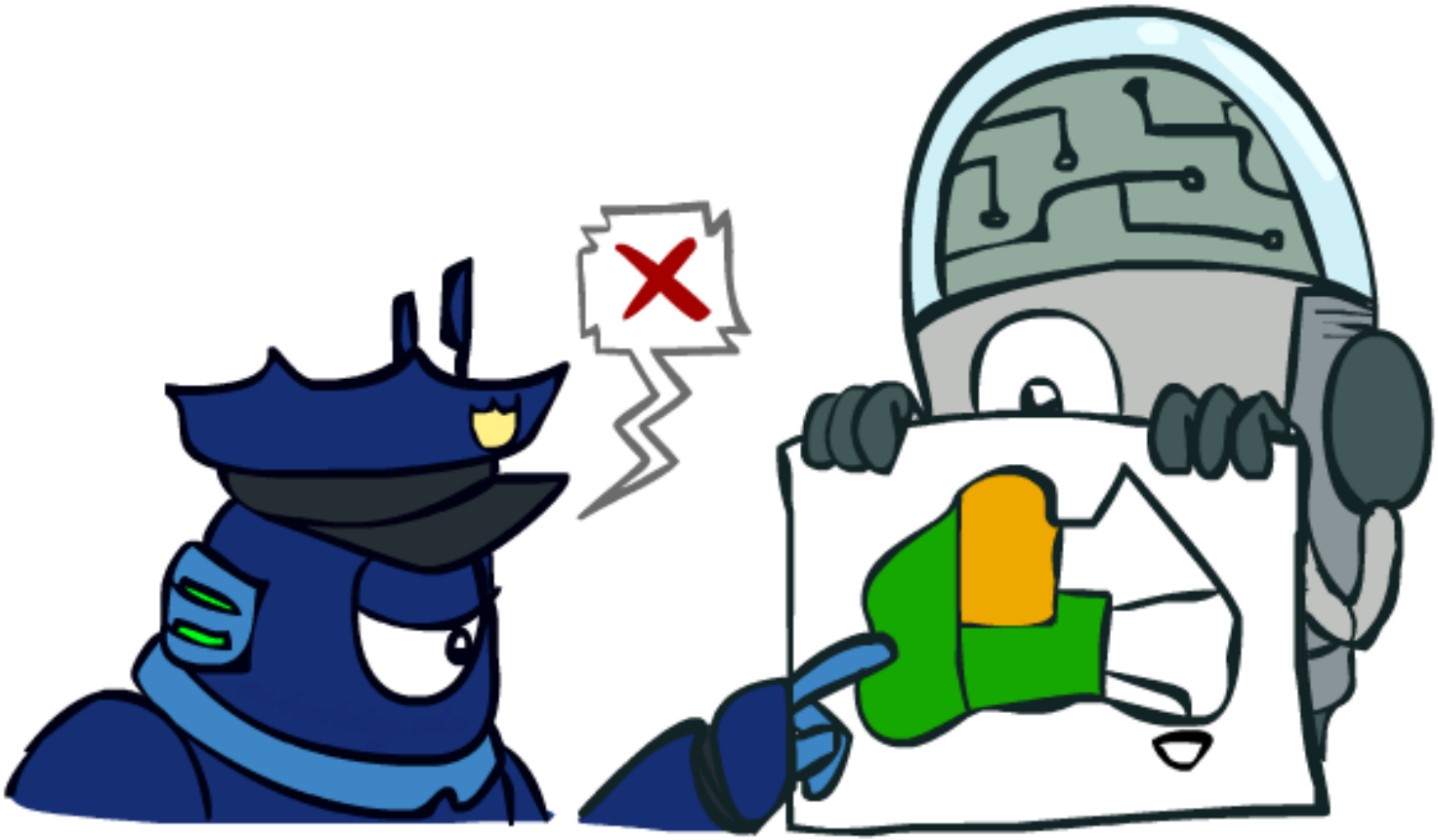
- What would BFS do?



- What would DFS do?

- What problems does naïve search have?
  - Ordering of variables.
  - Choices of values.

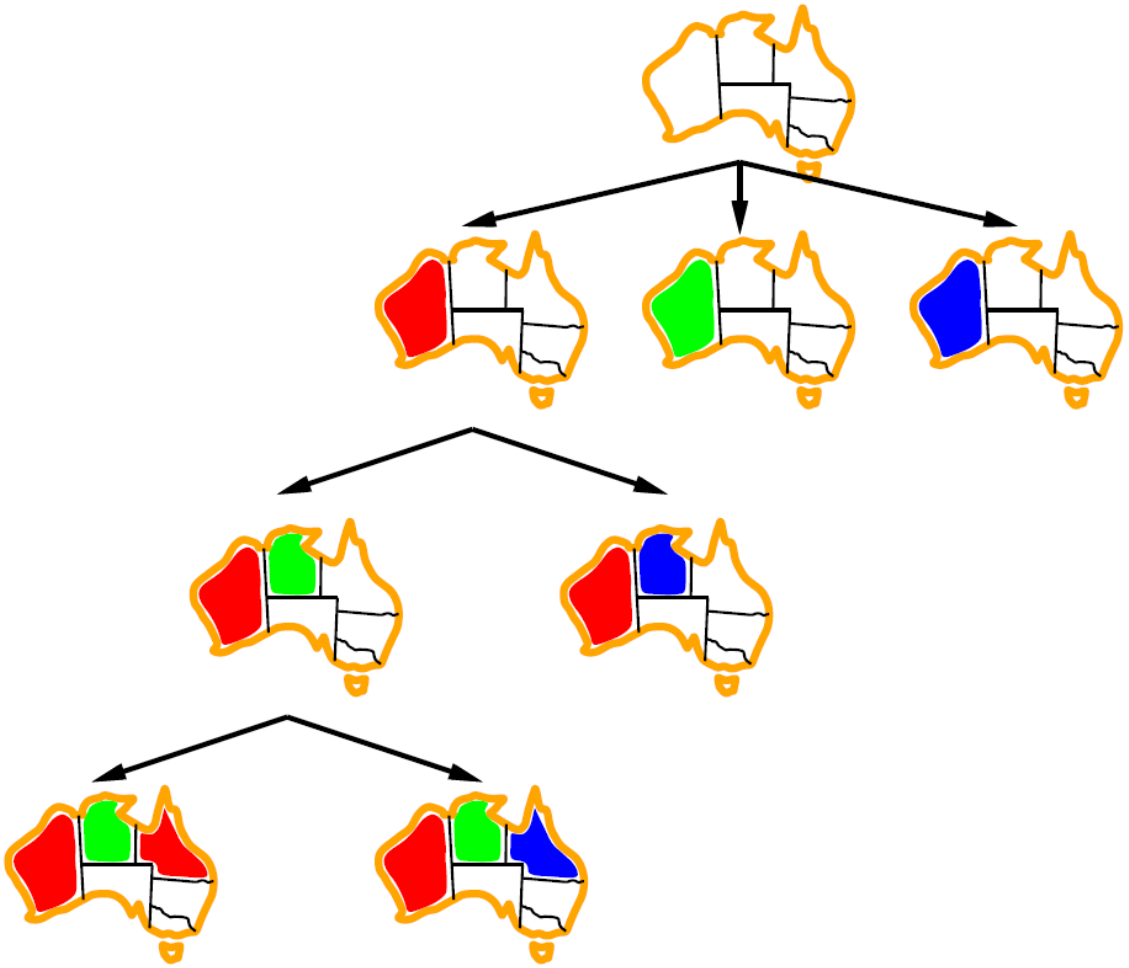
Backtracking Search



Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time in fixed ordering
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - **“Incremental goal test”**
- Depth-first search with these two improvements
  - **is called *backtracking search***

Backtracking Example



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
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

Improving Backtracking

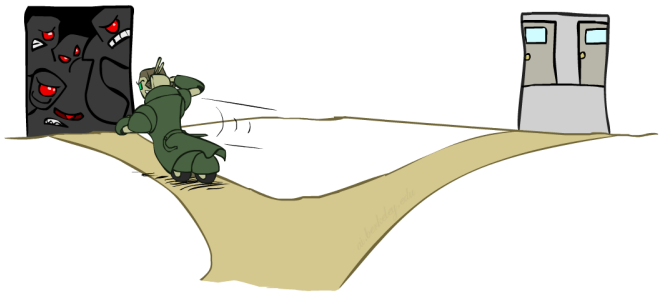
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

Ordering

Ordering: Minimum Remaining Values

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

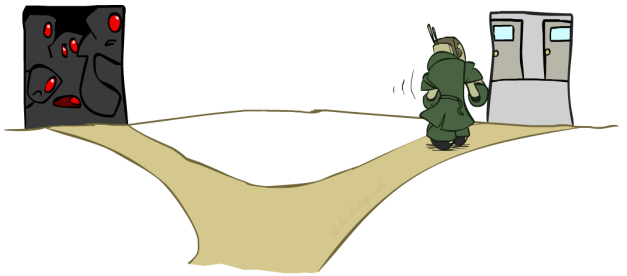
- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



Ordering: Least Constraining Value

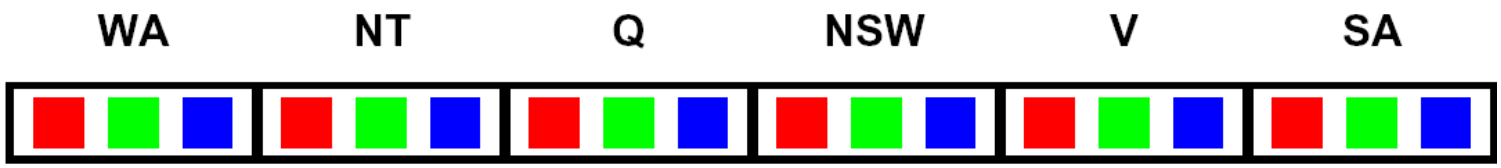
- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

- Why least rather than most?



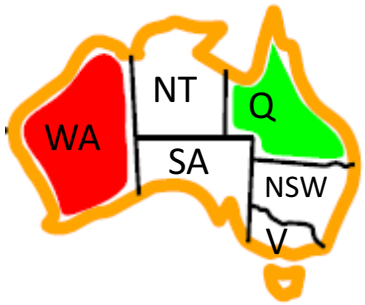
Filtering: Forward Checking

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



Filtering: Constraint Propagation

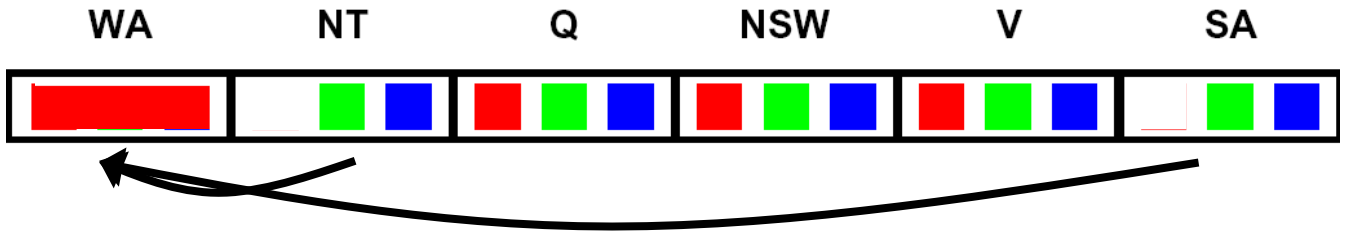
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

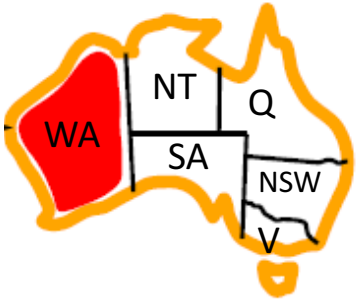


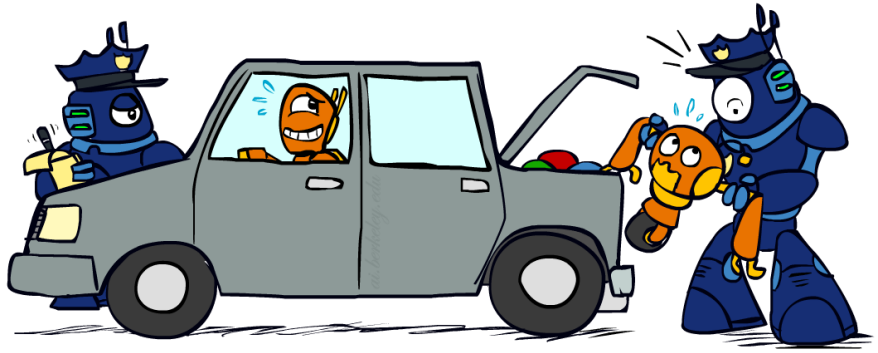
- NT and SA cannot both be blue!
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

- An arc X $\rightarrow$ 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





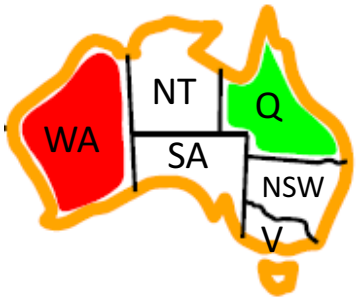


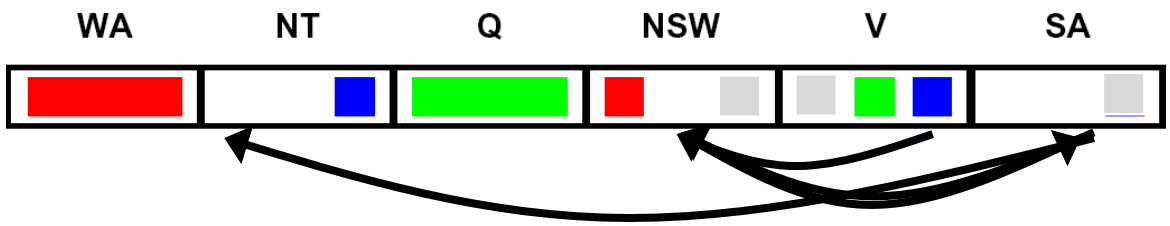
Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

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





Remember: Delete from the tail!

- 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?

Enforcing Arc Consistency in a CSP

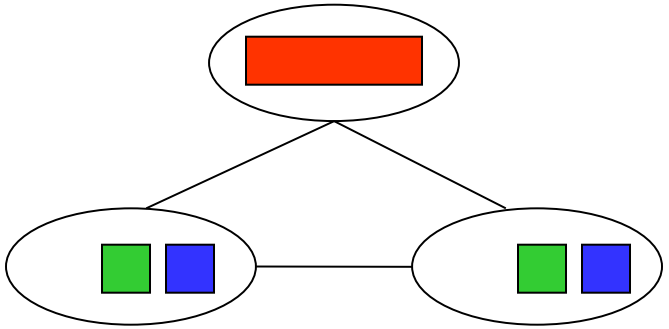
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   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
```

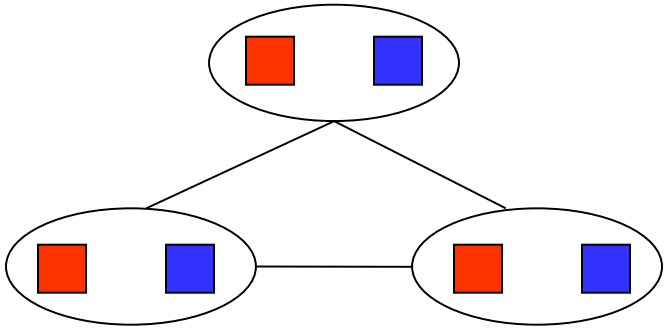
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- How?

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)

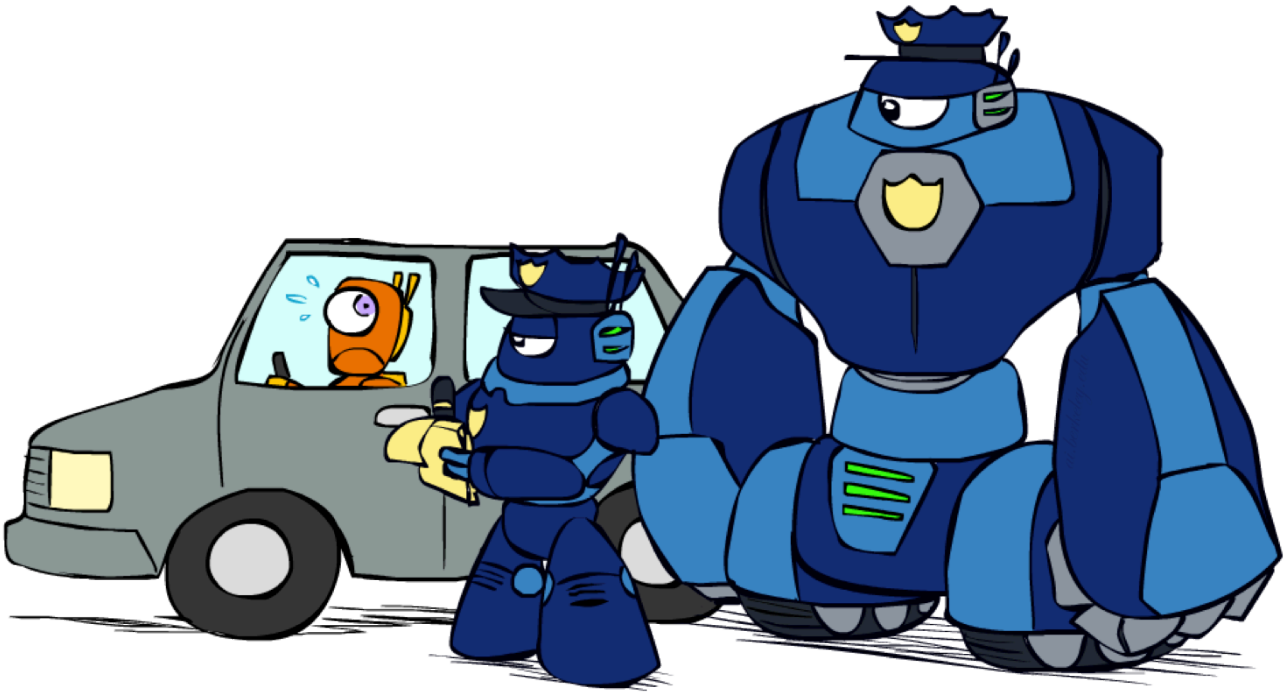
- Arc consistency still runs inside a backtracking search!





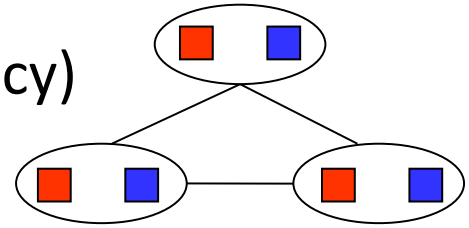
What went wrong here?

K-Consistency



K-Consistency

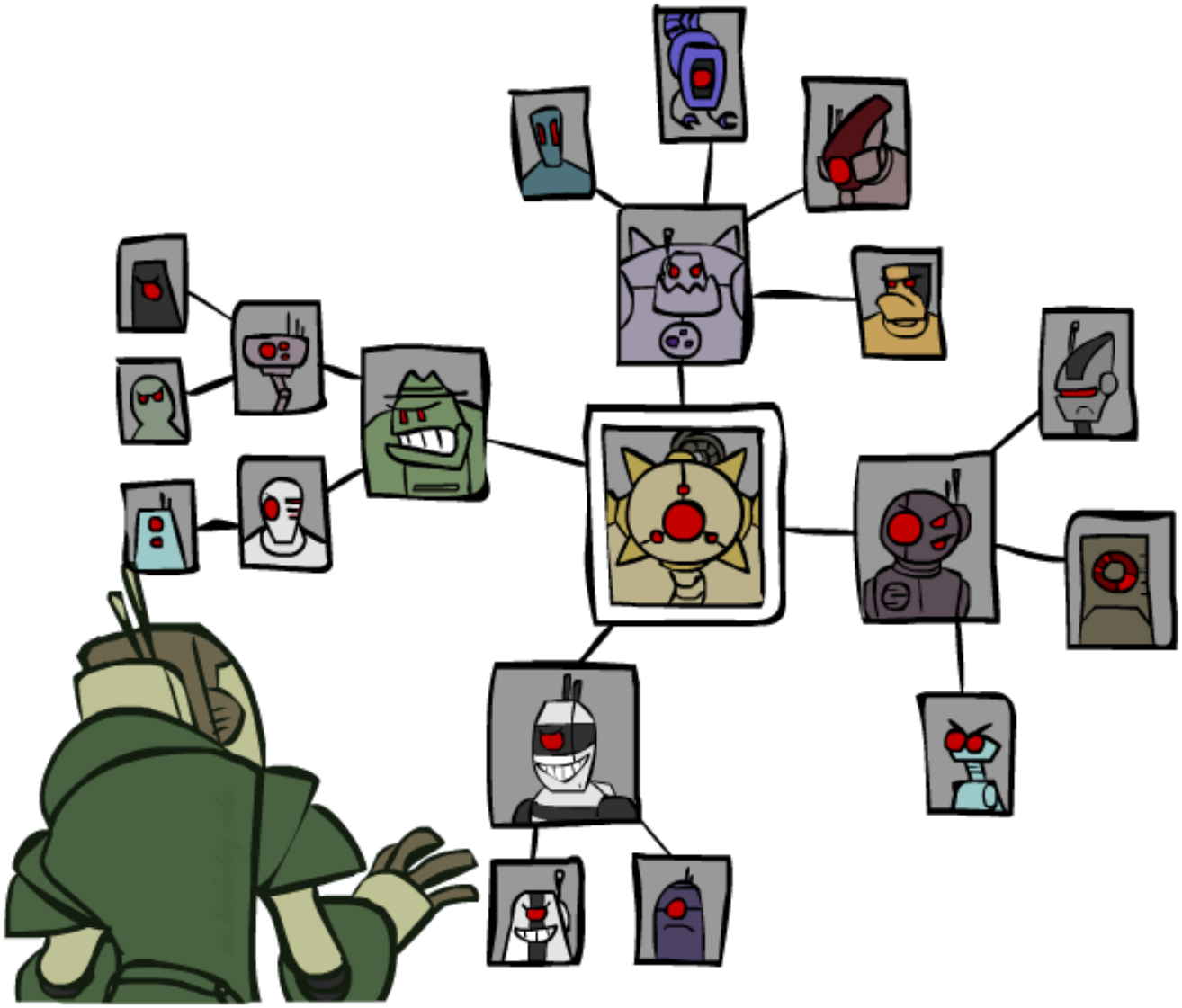
- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each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



Strong K-Consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  1. Choose any assignment to any variable
  2. Choose a new variable
  3. By 2-consistency, there is a choice consistent with the first
  4. Choose a new variable
  5. By 3-consistency, there is a choice consistent with the first 2
  6. …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)
- $O(n^2 d)$ for a solution to Strong K-Consistency problem.

Structure

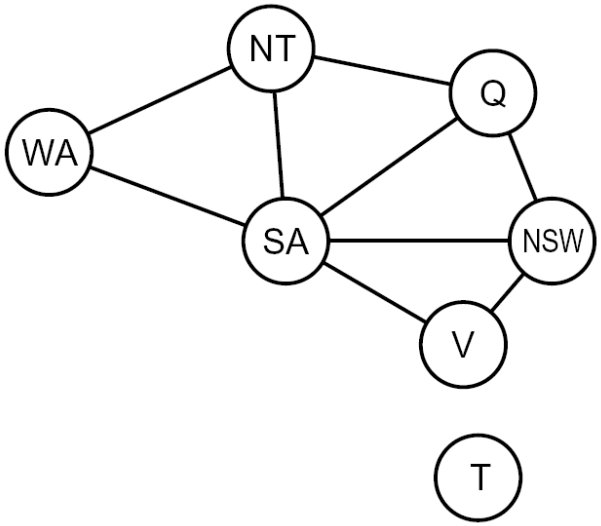


Problem Structure

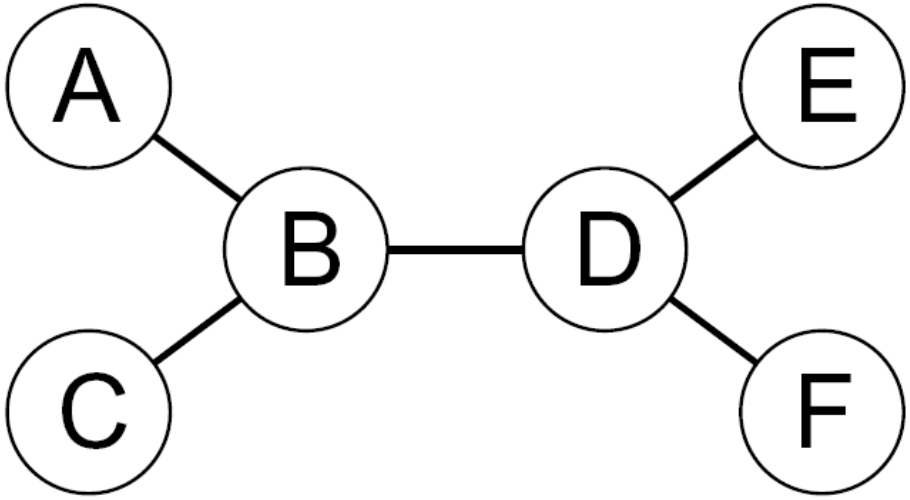
- Extreme case: independent sub-problems
  - Example: Tasmania and mainland do not interact

- Independent sub-problems are identifiable as connected components of constraint graph

- Suppose a graph of n variables can be broken into sub-problems 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



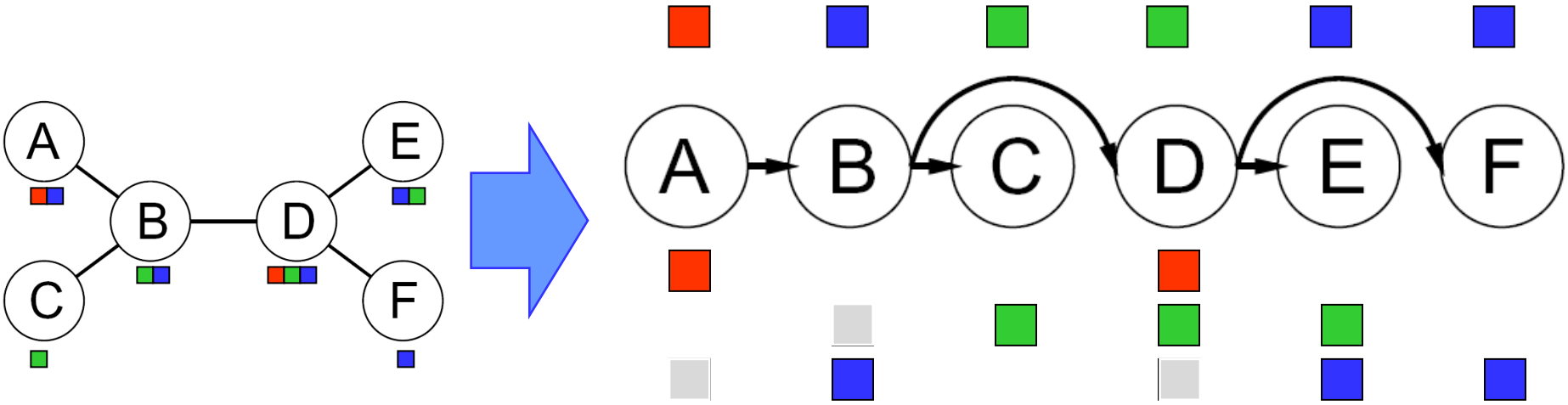
Tree-Structured CSPs



- 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 $O(d^n)$

Tree-Structured CSPs

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



  - Check the arc consistency from child to parent
  - Assign value from root to leaves

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
  inputs: *csp*, a CSP with components X, D, C

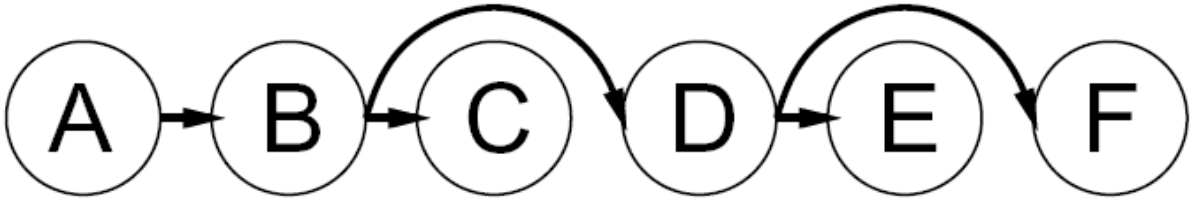
  $n \leftarrow$ number of variables in X
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, *root*)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    assignment$[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

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

Tree-Structured CSPs

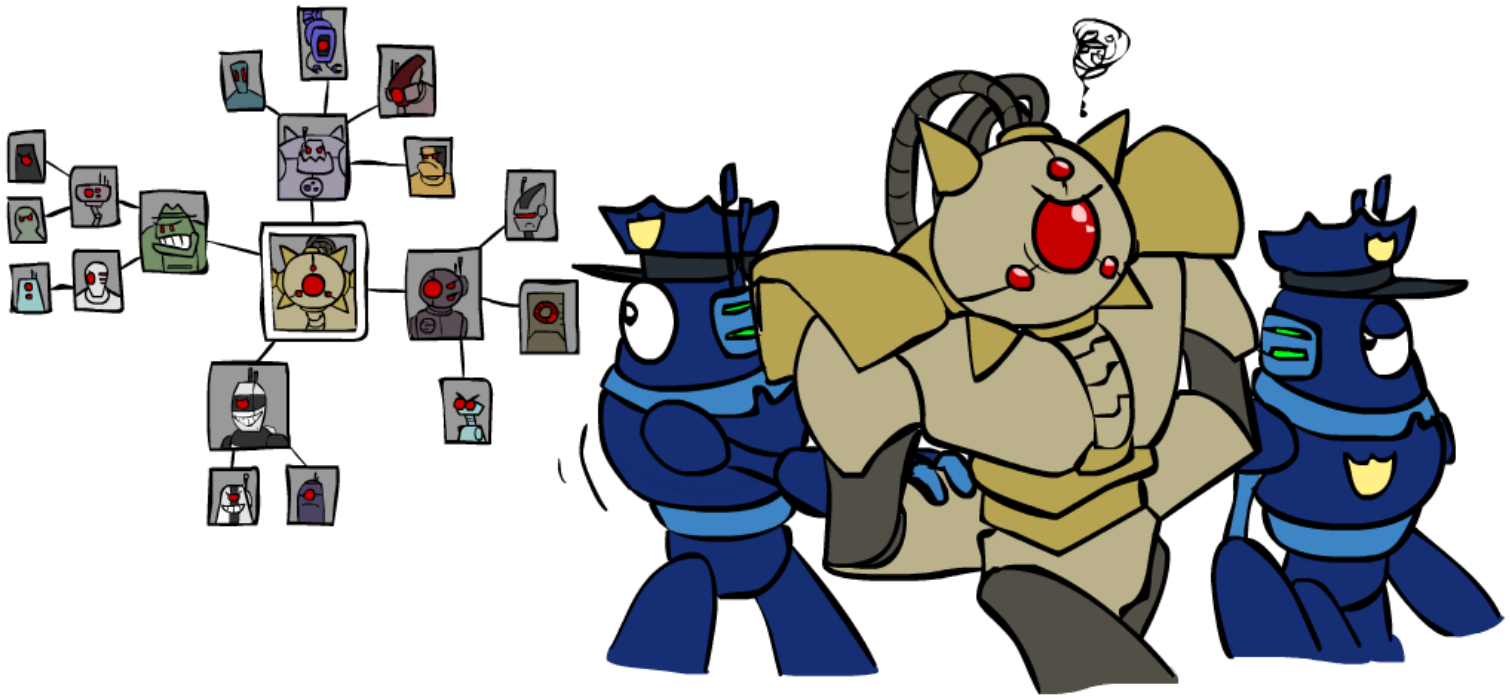
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
  - Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



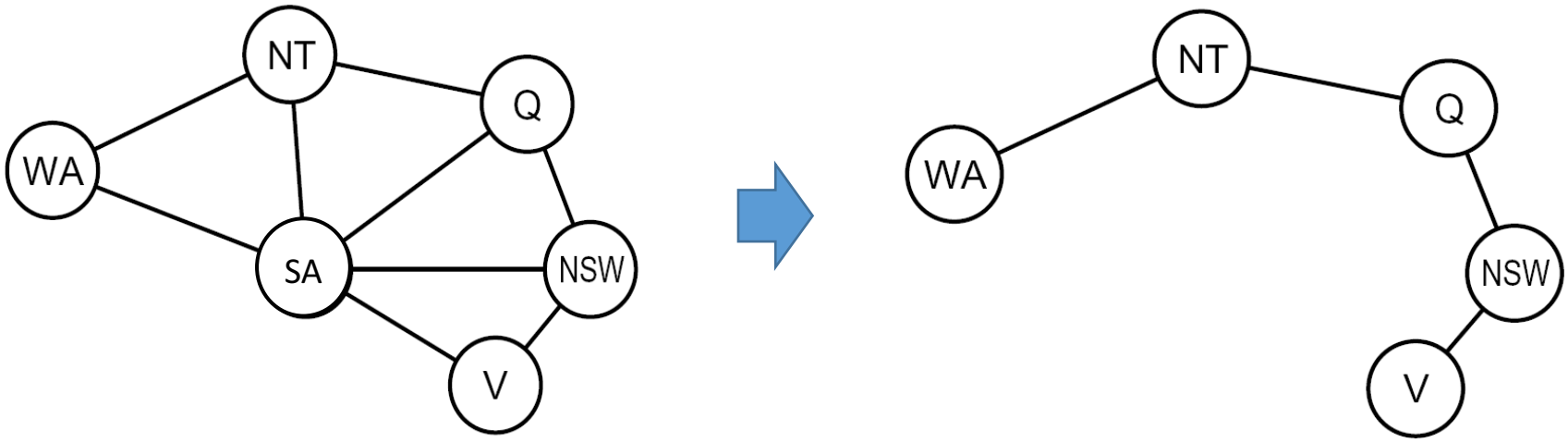
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
  - Given one value of a parent node, there is always a valid value in the child node to make the assignment consistent.

- 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
- Cycle Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cycle Cutset size c gives runtime O((d^c) (n-c) d^2)

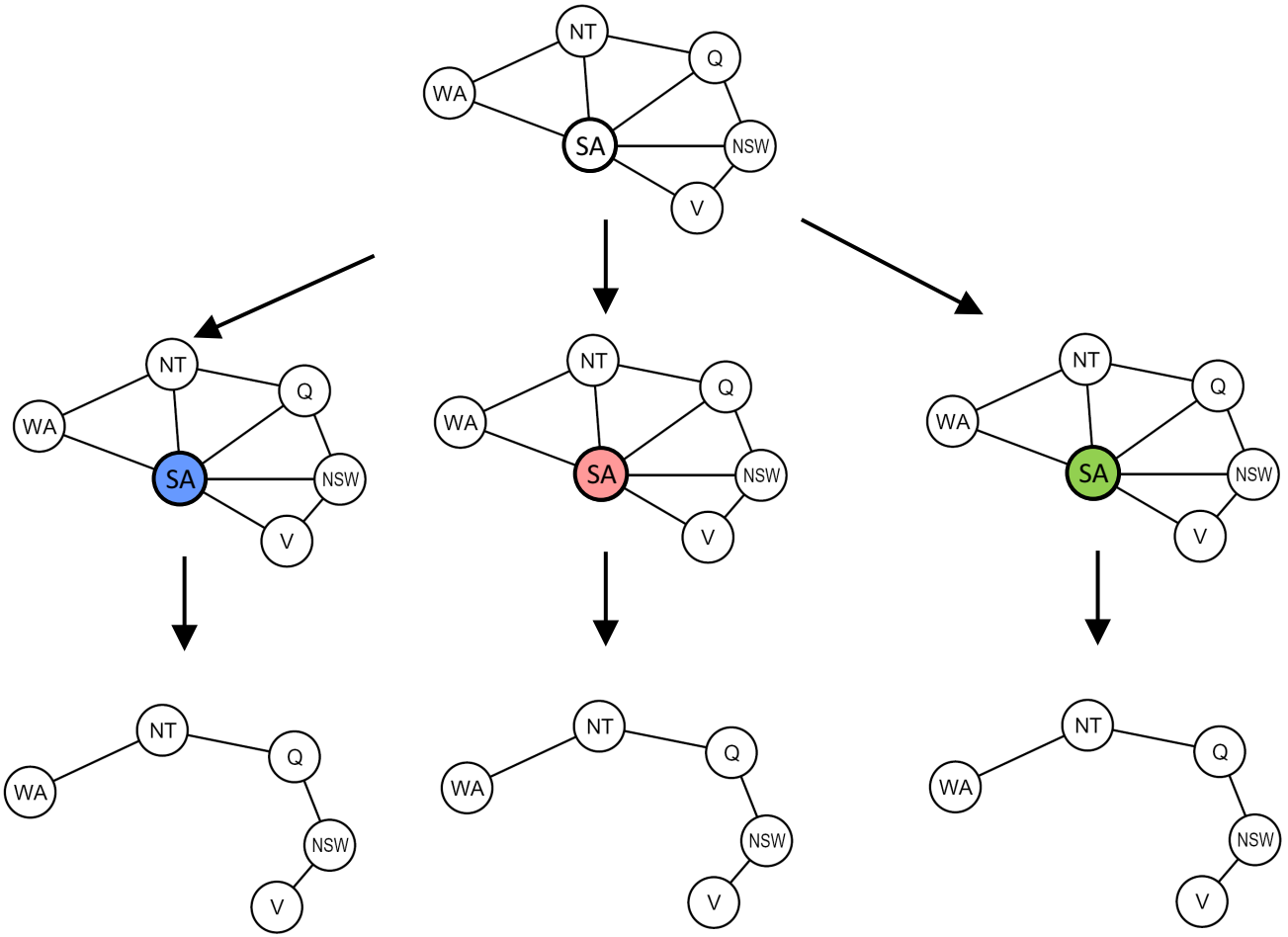
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

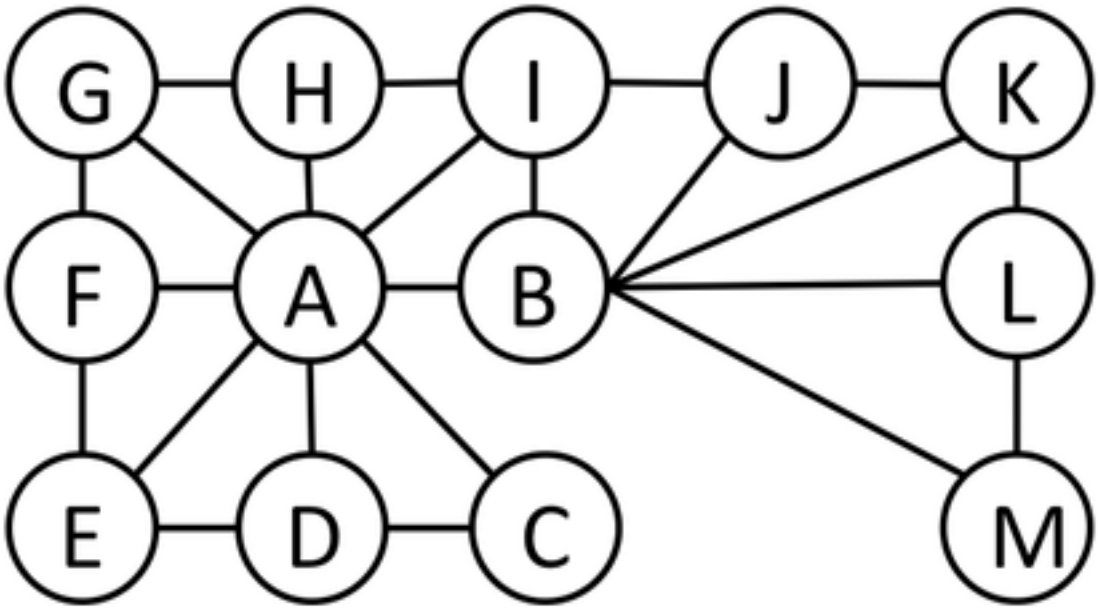
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



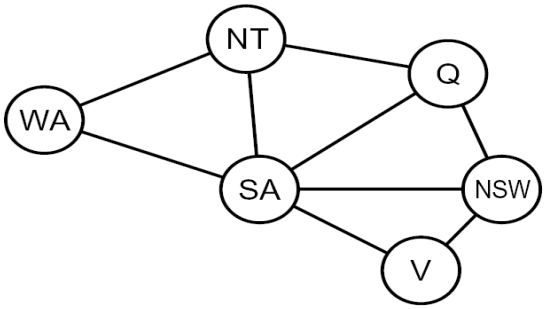
Cutset Quiz

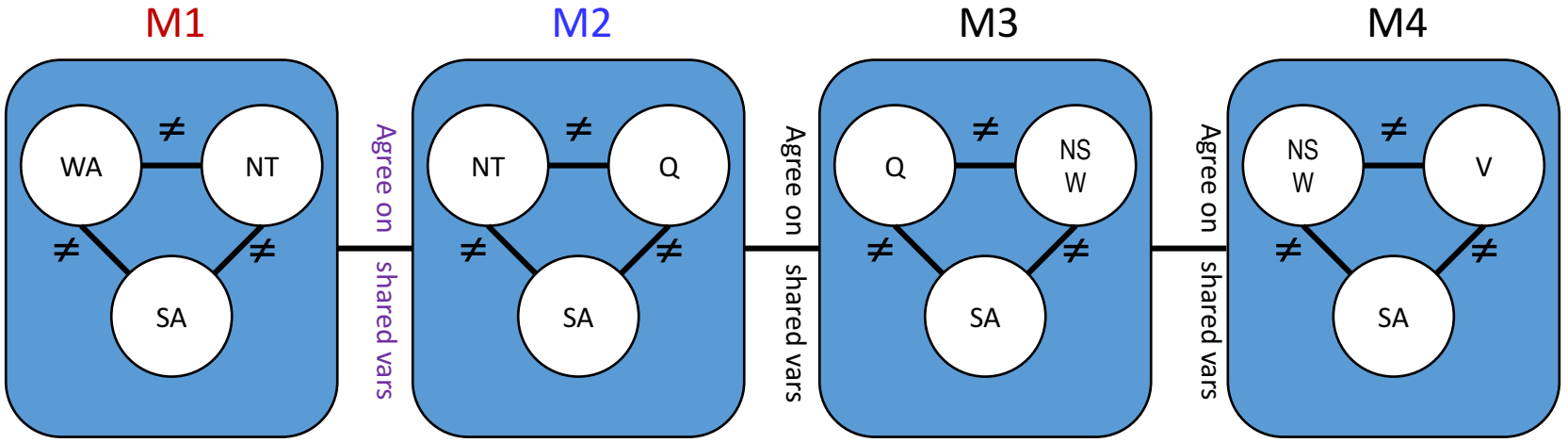
- Find the smallest cycle cutset for the graph below.



Tree Decomposition

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



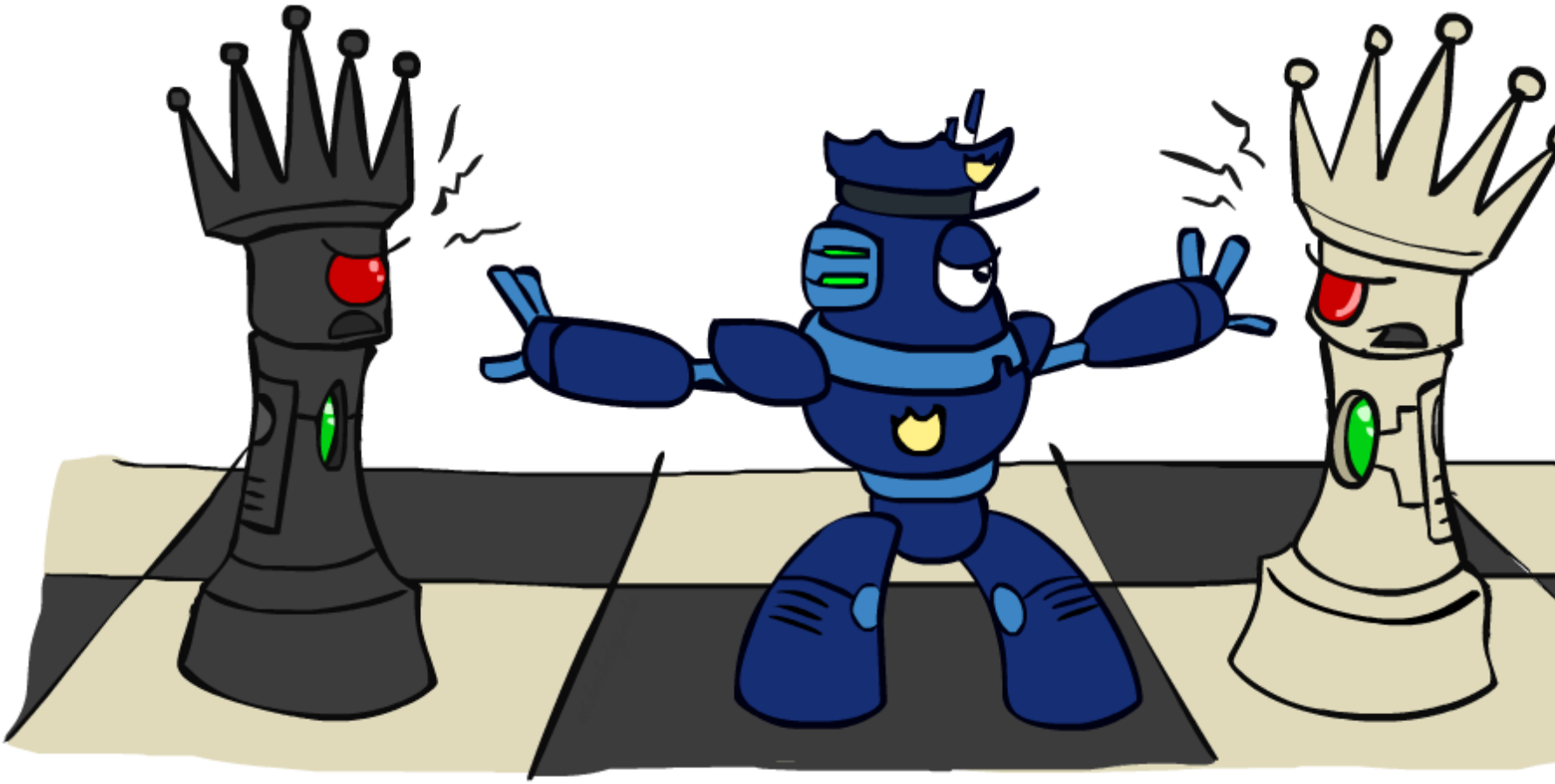


{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
…}

{(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



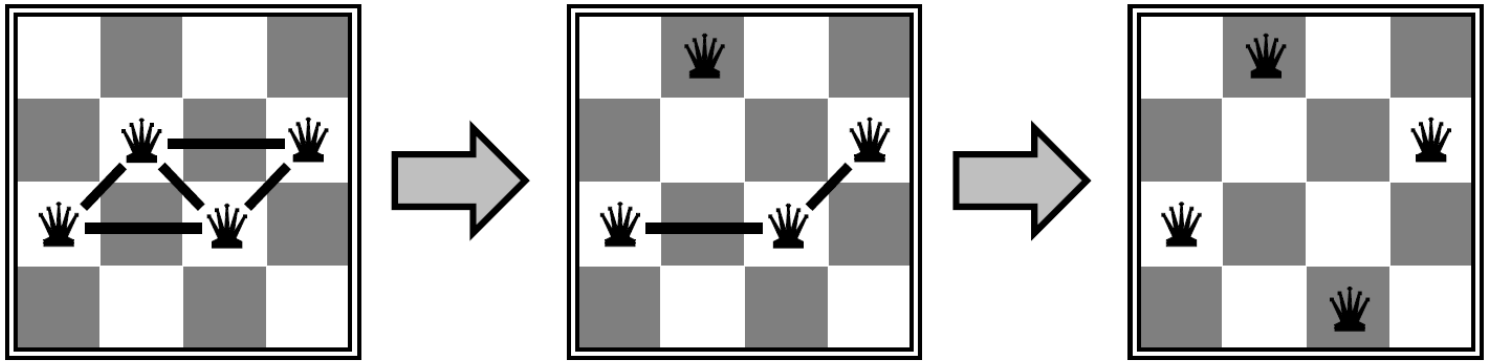
Iterative Algorithms for CSPs

- 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

- 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



- 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

Min-Conflicts Algorithm

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($var, v, current, csp$)
    set $var = value$ in $current$
  return *failure*

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

- Just like a hill climbing approach

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
  - ~50 for most of the N-queen problems

Local Search

- Systematic searches: keep paths in memory, and remember alternatives so search can backtrack. Solution is a path to a goal.
- Path may be irrelevant, if only the final configuration is needed (8-queens, IC design, network optimization, …)

Local Search versus Systematic Search

- Systematic Search
  - BFS, DFS, IDS, Best-First, A*
  - Keeps some history of visited nodes
  - Always complete for finite search spaces, some versions complete for infinite spaces
  - Good for building up solutions incrementally
    - State = partial solution
    - Action = extend partial solution

Local Search versus Systematic Search

- Local Search
  - Gradient descent, Greedy local search, Simulated Annealing, Genetic Algorithms
  - Does not keep history of visited nodes
  - Not complete. May be able to argue will terminate with “high probability”
  - Good for “fixing up” candidate solutions
    - State = complete candidate solution that may not satisfy all constraints
    - Action = make a small change in the candidate solution

Local search

- **Hill Climbing**
- Simulated annealing
- Local beam search
- Genetic Algorithm
- Gradient decent for continues function

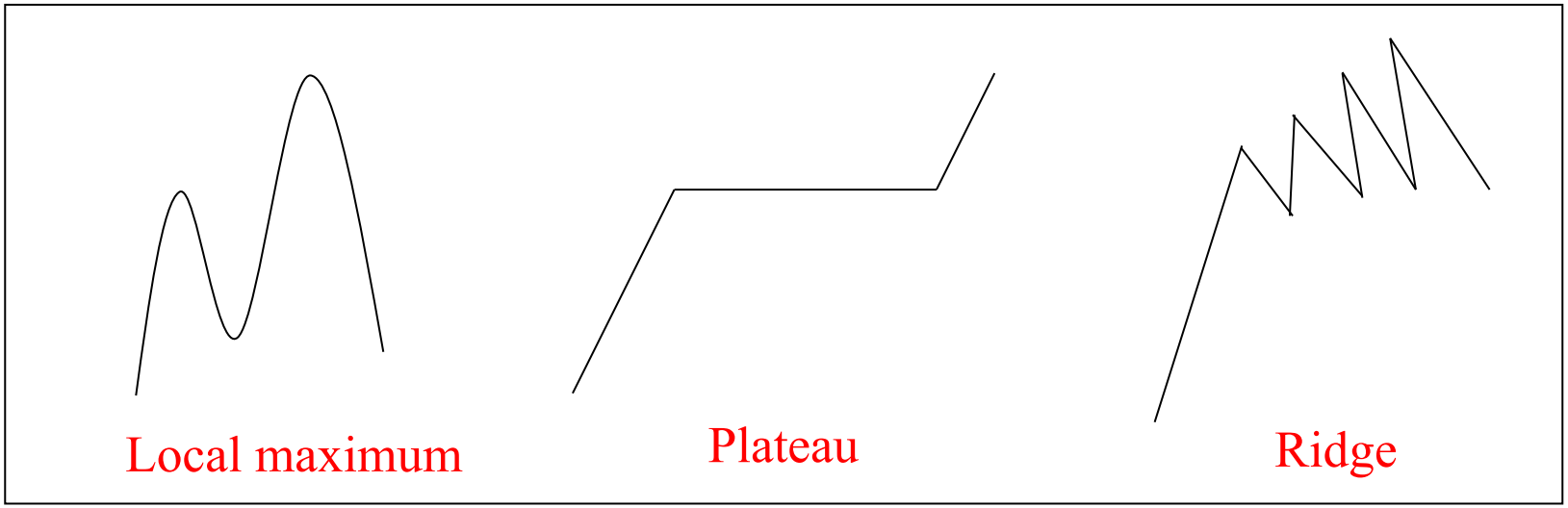
Hill-climbing search

- Generate nearby successor states to the current state
- Pick the best and replace the current state with that one.

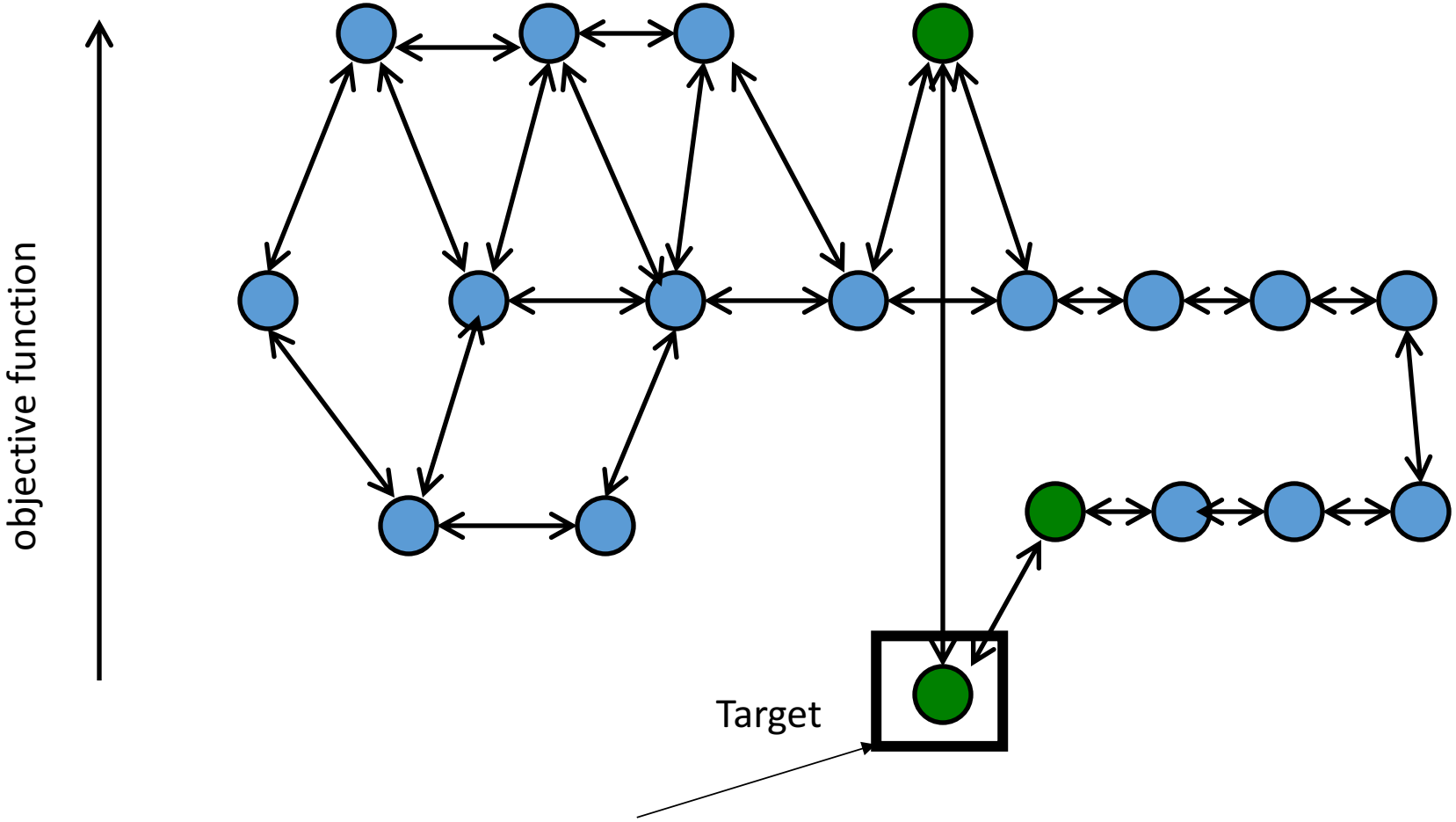
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

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

Hill-climbing search

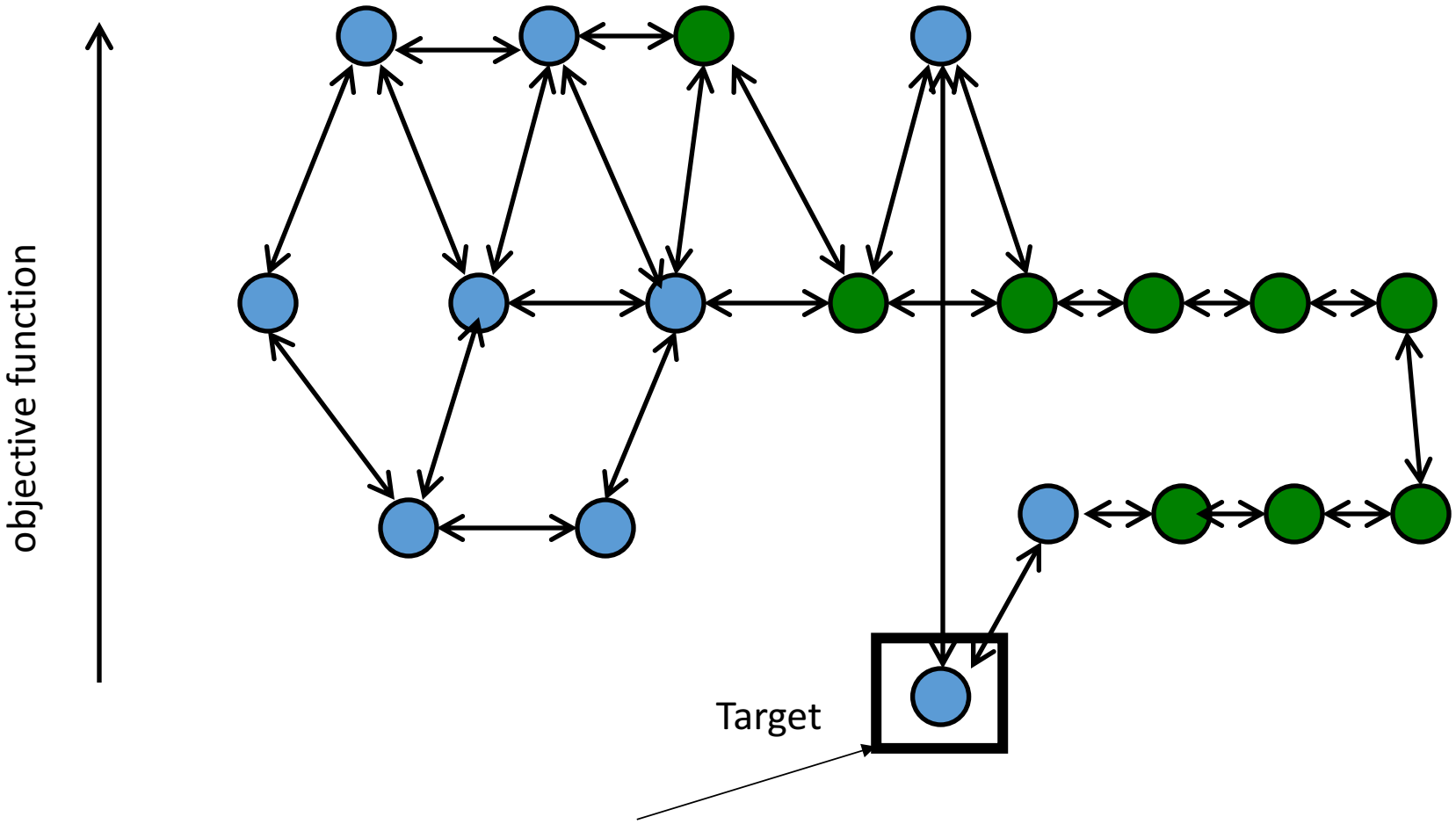
- Local maximum: a peak that is lower than the highest peak, so a suboptimal solution is returned
- Plateau: the evaluation function is flat, resulting in a random walk
- Ridge: result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.



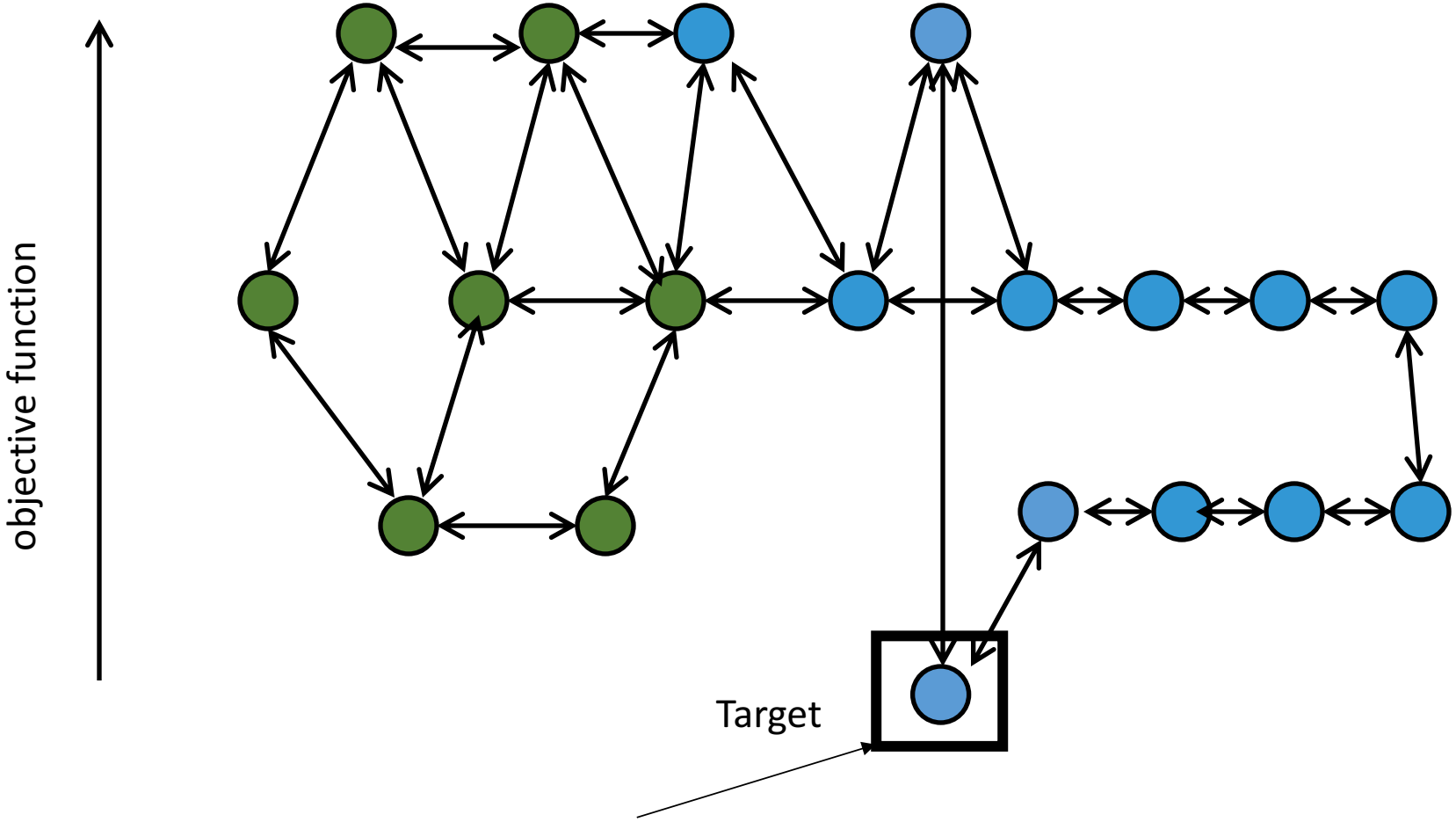
States Where Greedy Search Must Succeed



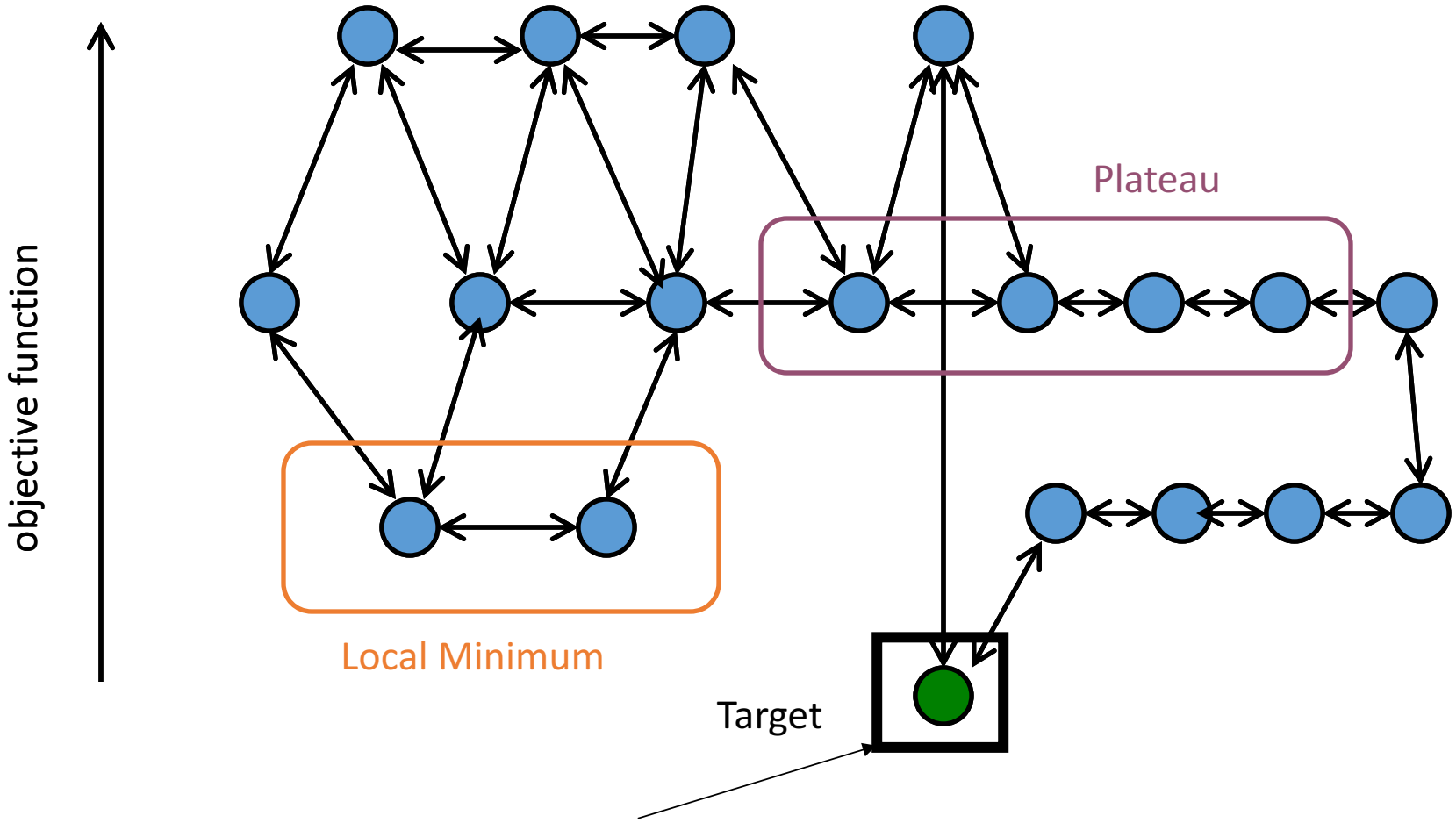
States Where Greedy Search Might Succeed



States Where Greedy Search Will NOT Succeed



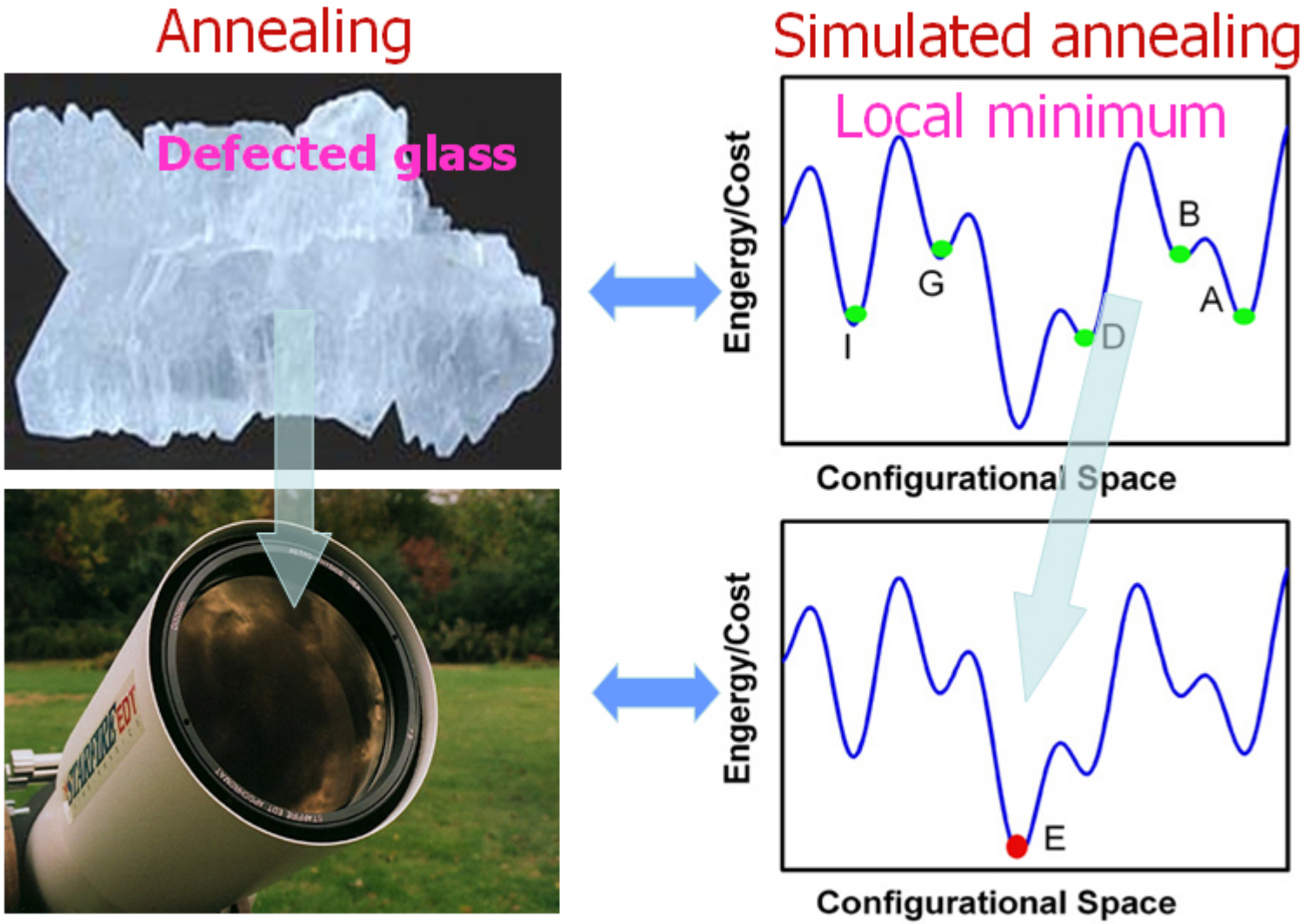
Greedy Search Landscape



Local search

- Hill Climbing
- **Simulated annealing**
- Local beam search
- Genetic Algorithm
- Gradient decent for continues function

Simulated Annealing



Simulated Annealing

- Combination of random walk and hill climbing.
- Generate a random new neighbor from current state.
- If it's better take it.
- If it's worse then take it with some probability proportional to the temperature and the delta between the new and old states.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

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

Simulated Annealing

- The algorithm wanders around during the early parts of the search, hopefully toward a good general region of the state space
- Toward the end, the algorithm does a more focused search, making few bad moves

Local search

- Hill Climbing
- Simulated annealing
- **Local beam search**
- Genetic Algorithm
- Gradient decent for continues function

Local Beam Search

- Keep track of k states rather than just one, as in hill climbing

- Begins with k randomly generated states
- At each step, all successors of all k states are generated
- If any one is a goal, Done!
- Otherwise, selects best k successors from the complete list, and repeats

Local Search in Continuous Spaces

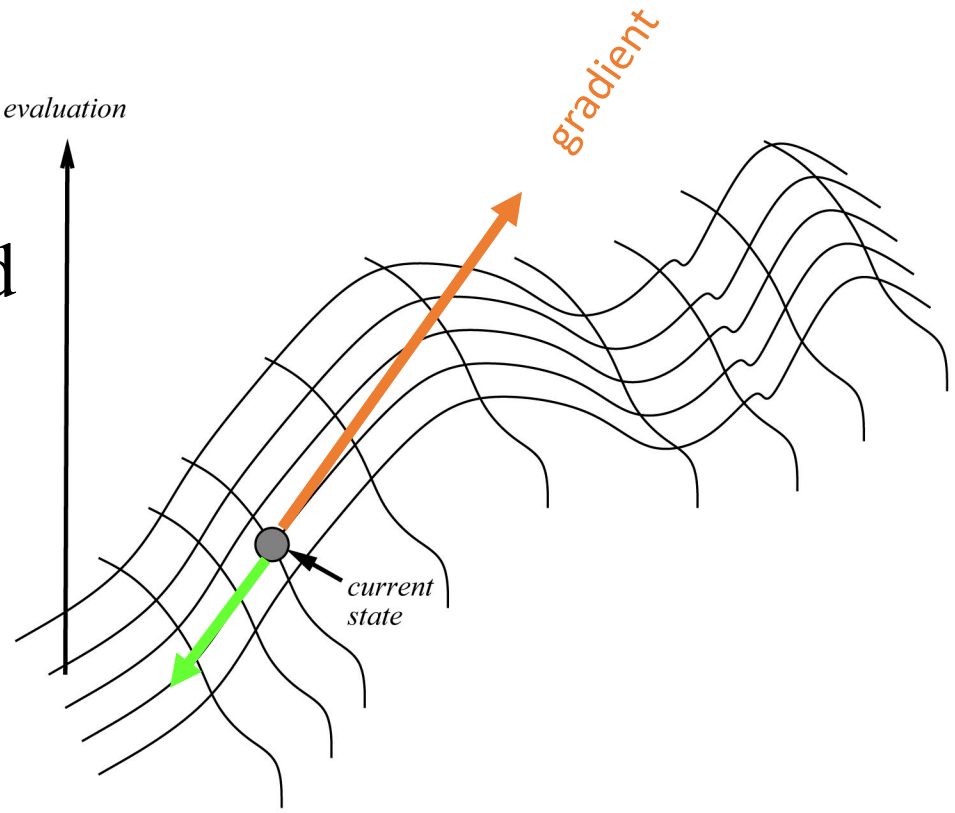
S = initial state vector

$f(S)$ = quantity to optimized

α = step size

until Goal_Test(S) do

$$S = S - \alpha \frac{\partial f}{\partial S}(S)$$



negative step to minimize *f*
positive step to maximize *f*

Wrap-up

- If you are interested, refer to chapter 4.1 – 4.2