

COE 4213564

Introduction to Artificial Intelligence

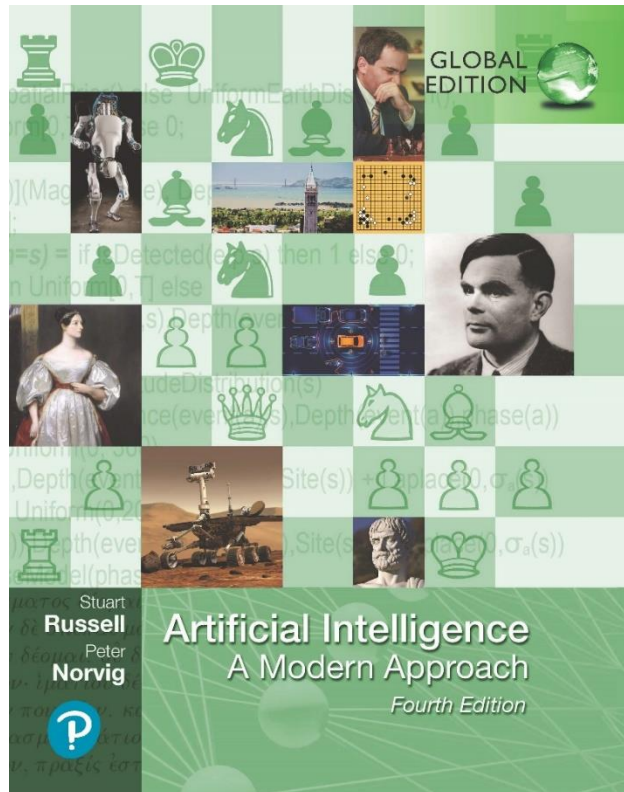
Constraint Satisfaction Problems



Many slides are adapted from CS 188 (<http://ai.berkeley.edu>), CS 322, CIS 521, CS 221, CS182, CS4420.

Artificial Intelligence: A Modern Approach

Fourth Edition, Global Edition

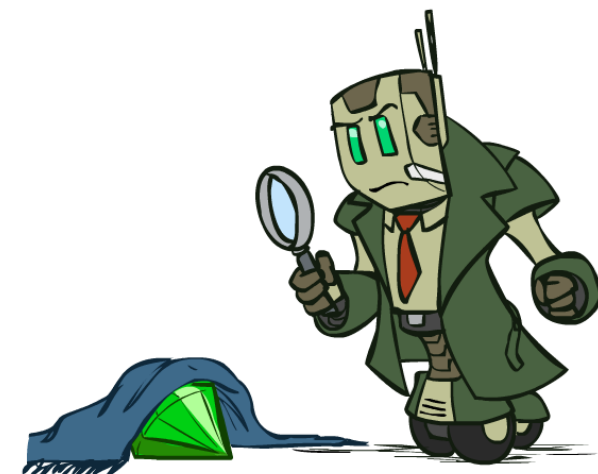
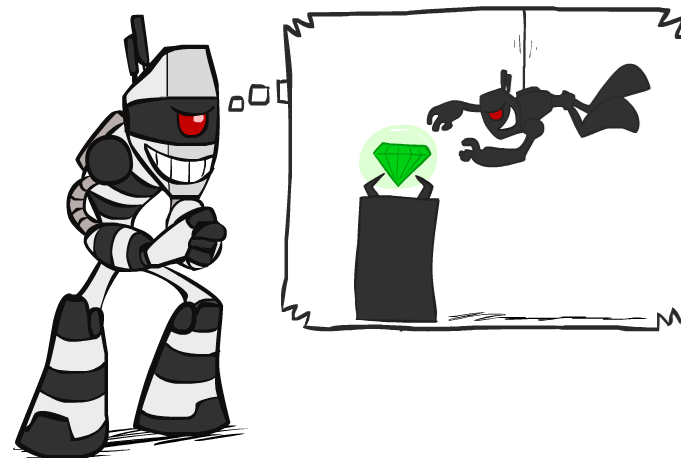


Chapter 5

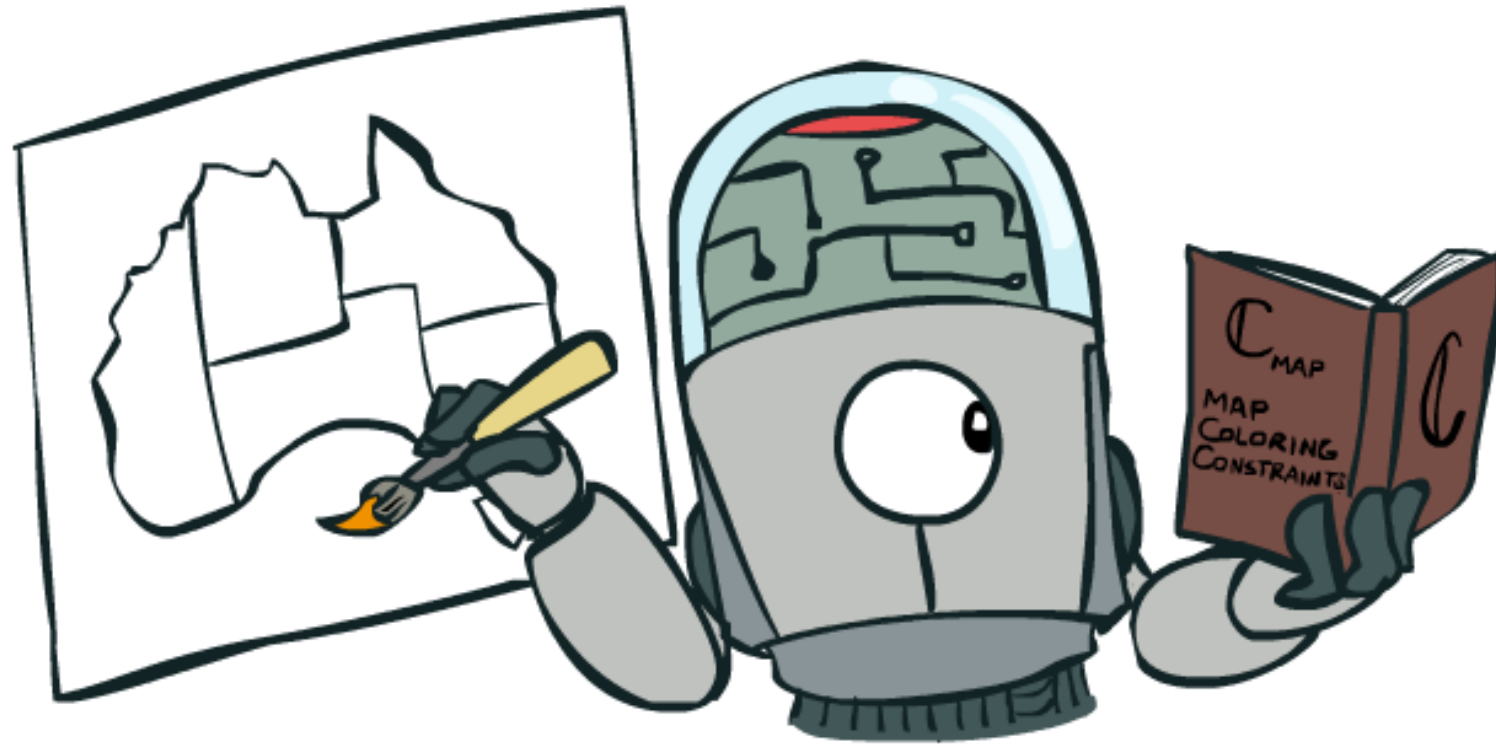
Constraint Satisfaction Problems

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, 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)
 - CSPs are a specialized class of identification problems where we assign values to variables while respecting certain constraints



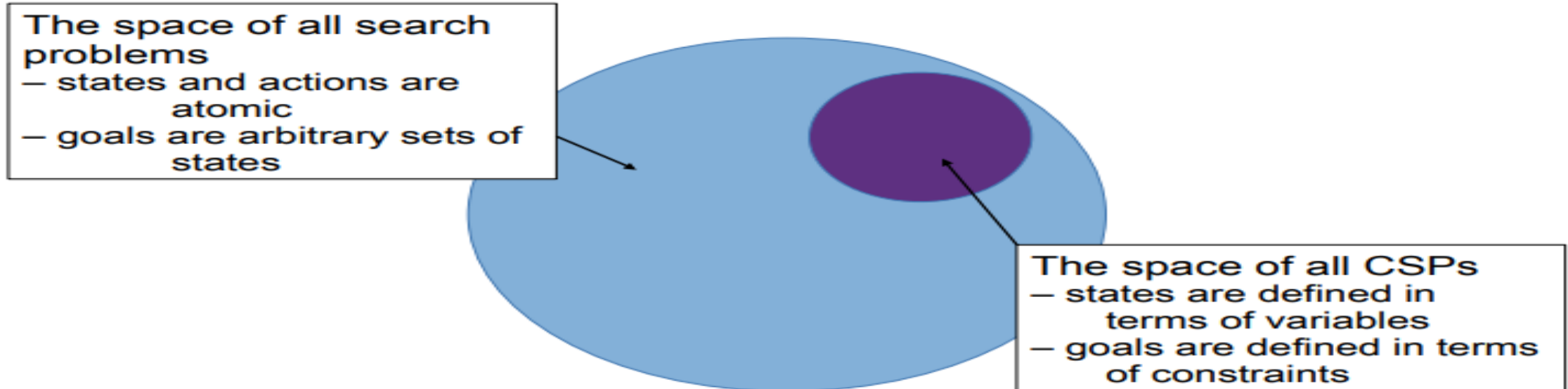
Constraint Satisfaction Problems



Constraint Satisfaction Problems

- Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must **satisfy** a number of constraints or limitations.

$\text{CSPs} \subseteq \text{All search problems}$



A CSP is defined by:

1. a set of variables and their associated domains.
2. a set of constraints that must be satisfied.

Defining Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) consists of three components, X , D , and C :

- X is a set of variables, $\{X_1, \dots, X_n\}$.
- D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable
- C is a set of constraints that specify allowable combination of values

CSPs deal with assignments of values to variables.

- A complete assignment is one in which every variable is assigned a value, and a solution to a CSP is a consistent, complete assignment.
- A partial assignment is one that leaves some variables unassigned.
- Partial solution is a partial assignment that is consistent

Constraint satisfaction problems (CSPs)

Standard search problem:

state is a “black box”—any old data structure that supports goal test, eval, successor

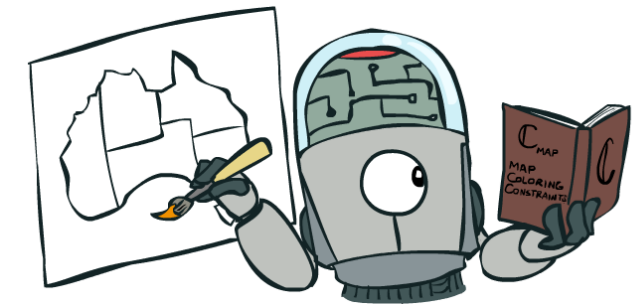
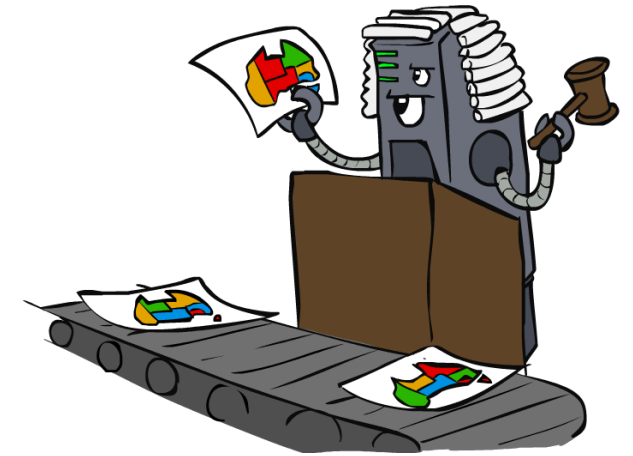
CSP:

state is defined by **variables** X_i with **values** from **domain** D_i

goal test is a set of **constraints** specifying allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power than standard search algorithms



Constraint satisfaction problems (CSPs)

Definition:

A **constraint satisfaction problem (CSP)** consists of:

- a set of **variables** \mathcal{V}
- a **domain** $\text{dom}(V)$ for each variable $V \in \mathcal{V}$
- a set of **constraints** \mathcal{C}

Simple example:

- $\mathcal{V} = \{V_1\}$
 - $\text{dom}(V_1) = \{1,2,3,4\}$
- $\mathcal{C} = \{C_1, C_2\}$
 - $C_1: V_1 \neq 2$
 - $C_2: V_1 > 1$

Another example:

- $\mathcal{V} = \{V_1, V_2\}$
 - $\text{dom}(V_1) = \{1,2,3\}$
 - $\text{dom}(V_2) = \{1,2\}$
- $\mathcal{C} = \{C_1, C_2, C_3\}$
 - $C_1: V_2 \neq 2$
 - $C_2: V_1 + V_2 < 5$
 - $C_3: V_1 > V_2$

Definition:

A **constraint satisfaction problem (CSP)** consists of:

- a set of **variables** \mathcal{V}
- a **domain** $\text{dom}(V)$ for each variable $V \in \mathcal{V}$
- a set of **constraints** \mathcal{C}

Definition:

A **model** of a CSP is an assignment of values to all of its variables that **satisfies** all of its constraints.

Simple example:

- $\mathcal{V} = \{V_1\}$
 - $\text{dom}(V_1) = \{1,2,3,4\}$
- $\mathcal{C} = \{C_1, C_2\}$
 - $C_1: V_1 \neq 2$
 - $C_2: V_1 > 1$

All models for this CSP:

$\{V_1 = 3\}$

$\{V_1 = 4\}$

Possible Worlds

Definition:

A **possible world** of a CSP is an assignment of values to all of its variables.

Definition:

A **model** of a CSP is an assignment of values to all of its variables that **satisfies** all of its constraints.

i.e. *a model is a possible world that satisfies all constraints*

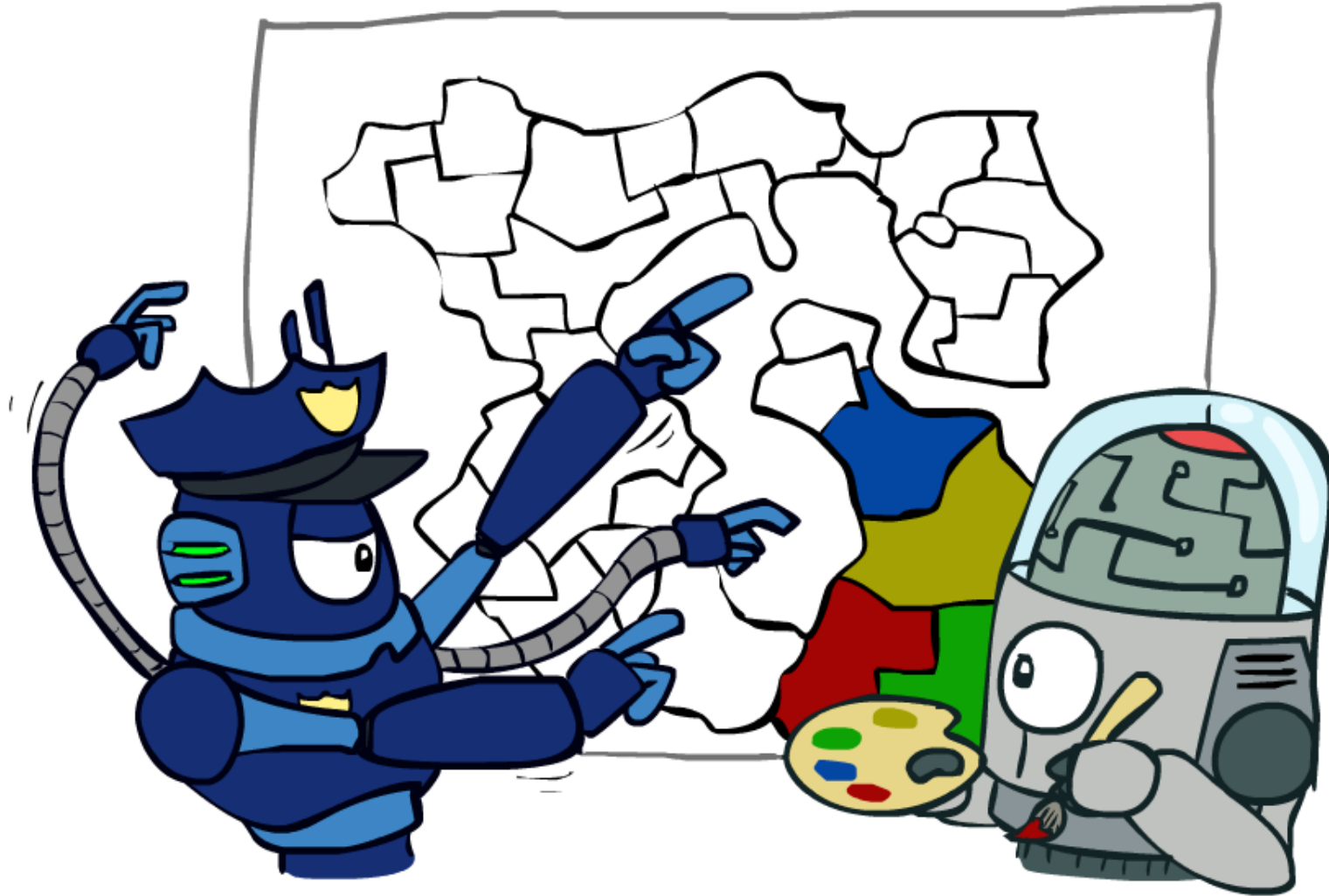
Another example:

- $\mathcal{V} = \{V_1, V_2\}$
 - $\text{dom}(V_1) = \{1, 2, 3\}$
 - $\text{dom}(V_2) = \{1, 2\}$
- $C = \{C_1, C_2, C_3\}$
 - $C_1: V_2 \neq 2$
 - $C_2: V_1 + V_2 < 5$
 - $C_3: V_1 > V_2$

Possible worlds for this CSP:

- $\{V_1=1, V_2=1\}$
- $\{V_1=1, V_2=2\}$
- $\{V_1=2, V_2=1\}$ (a model)
- $\{V_1=2, V_2=2\}$
- $\{V_1=3, V_2=1\}$ (a model)
- $\{V_1=3, V_2=2\}$

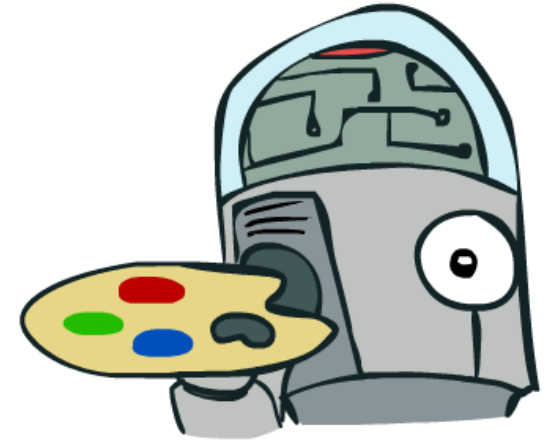
Varieties of CSPs and Constraints



Varieties of CSPs

- Discrete Variables

- Finite domains
 - Size d means $O(d^n)$ complete assignments
 - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
 - E.g., job scheduling, variables are start/end times for each job
 - Linear constraints solvable, nonlinear undecidable



- Continuous variables

- E.g., start/end times for Hubble Telescope observations
- Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)



Varieties of Constraints

- Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq \text{green}$$

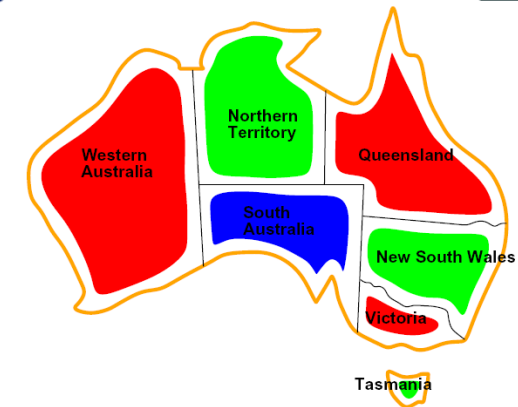
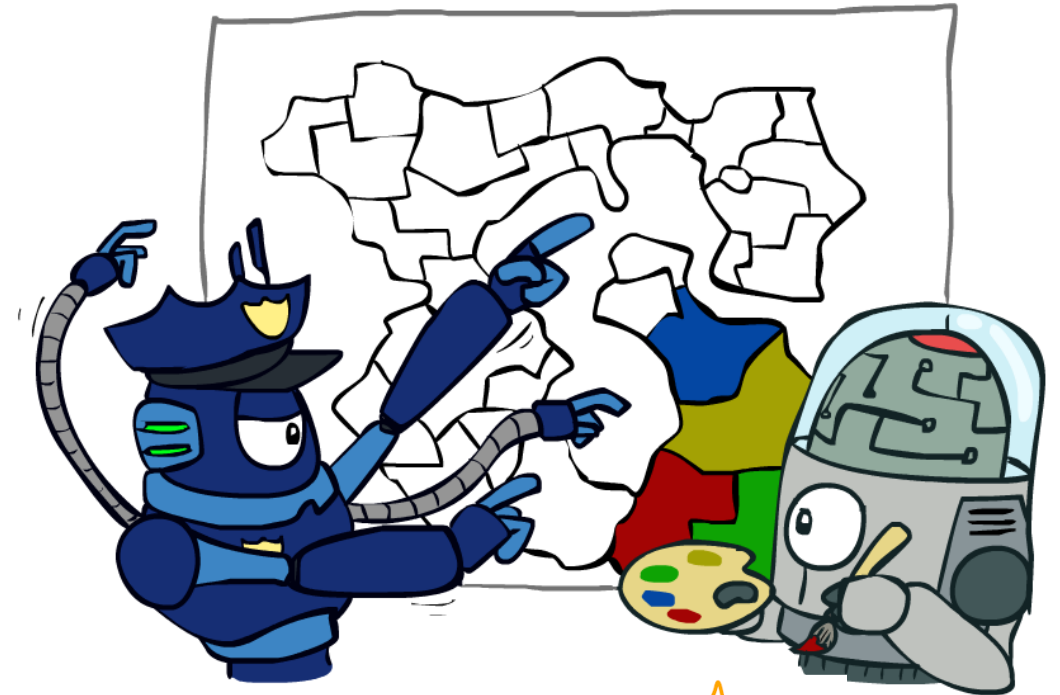
- Binary constraints involve pairs of variables, e.g.:

$$SA \neq WA$$

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

- Preferences (soft constraints):

- E.g., red is better than green
- Often representable by a cost for each variable assignment
- Gives constrained optimization problems
- (We'll ignore these until we get to Bayes' nets)



Constraints

Constraints are **restrictions** on the values that one or more variables can take

- **Unary constraint**: restriction involving a single variable
 - E.g.: $V_2 \neq 2$
- **k-ary constraint**: restriction involving k different variables
 - E.g. binary (k=2): $V_1 + V_2 < 5$
 - E.g. 3-ary: $V_1 + V_2 + V_4 < 5$
 - We will mostly deal with binary constraints
- Constraints can be specified by

1. listing all combinations of valid domain values for the variables participating in the constraint

- E.g. for constraint $V_1 > V_2$ and $\text{dom}(V_1) = \{1,2,3\}$ and $\text{dom}(V_2) = \{1,2\}$:

V_1	V_2
2	1
3	1
3	2

2. giving a **function (predicate)** that returns true if given values for each variable which satisfy the constraint else false: $V_1 > V_2$

Constraints

– Constraints can be specified by

1. listing all combinations of valid domain values for the variables participating in the constraint

- E.g. for constraint $V_1 > V_2$ and $\text{dom}(V_1) = \{1,2,3\}$ and $\text{dom}(V_2) = \{1,2\}$:

V_1	V_2
2	1
3	1
3	2

2. giving a function that returns true when given values for each variable which satisfy the constraint: $V_1 > V_2$

A possible world **satisfies** a set of constraints

– if the values for the variables involved in each constraint are consistent with that constraint

1. They are elements of the list of valid domain values
2. Function returns true for those values

– Examples

- $\{V_1=1, V_2=1\}$ (does not satisfy above constraint)
- $\{V_1=3, V_2=1\}$ (satisfies above constraint)

Scope of a constraint

Definition:

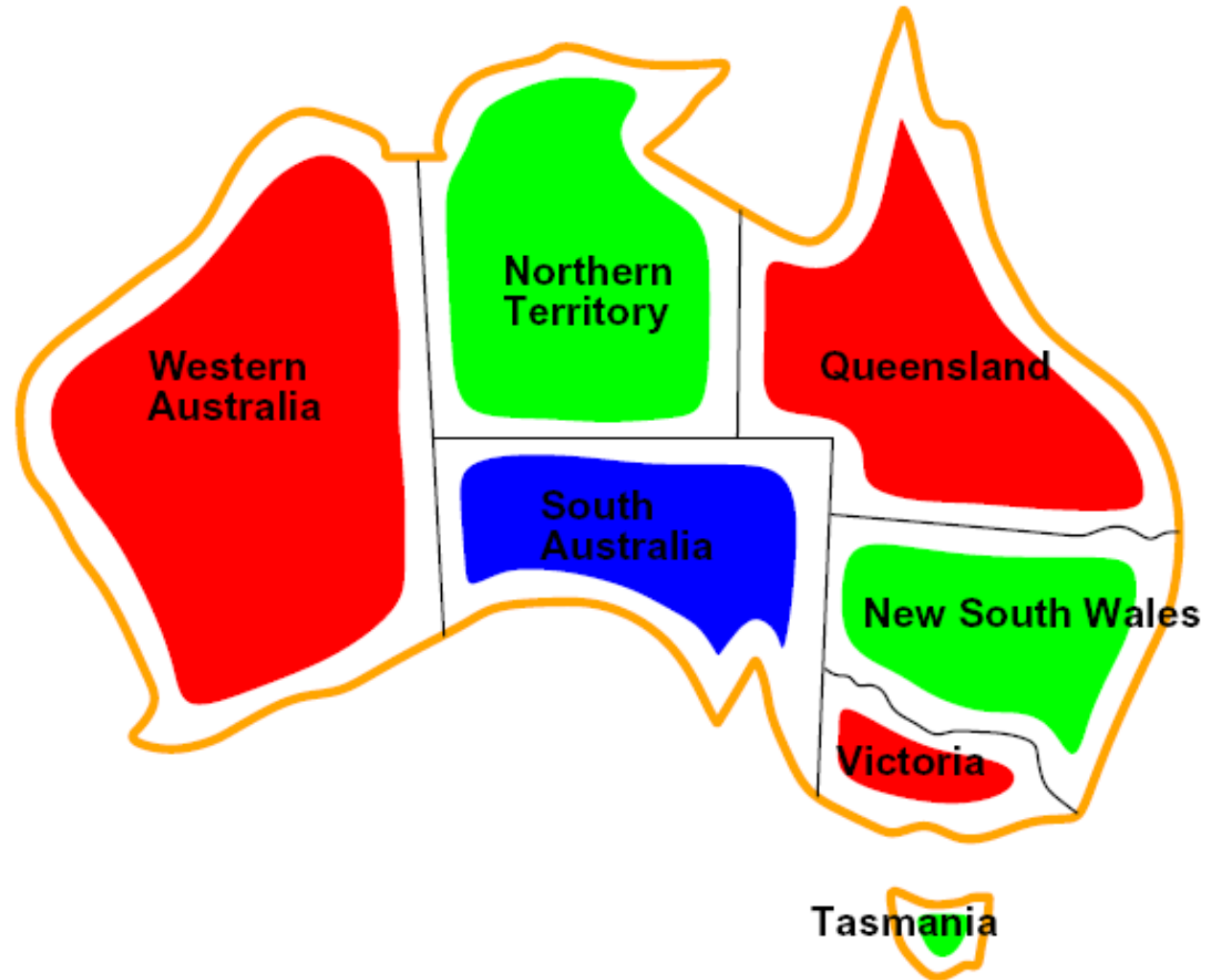
The **scope** of a constraint is the set of variables that are involved in the constraint

- Examples:
 - $V_2 \neq 2$ has scope $\{V_2\}$
 - $V_1 > V_2$ has scope $\{V_1, V_2\}$
 - $V_1 + V_2 + V_4 < 5$ has scope $\{V_1, V_2, V_4\}$
- How many variables are in the scope of a k-ary constraint ?
k variables

Solving Constraint Satisfaction Problems

- Even the simplest problem of determining whether or not a model exists in a general CSP with finite domains is **NP-hard**
 - There is no known algorithm with worst case polynomial runtime.
 - We can't hope to find an algorithm that is polynomial for all CSPs.
- However, we can try to:
 - find efficient (polynomial) **consistency algorithms** that reduce the size of the search space
 - **identify special cases** for which algorithms are efficient
 - work on **approximation algorithms** that can find good solutions quickly, even though they may offer no theoretical guarantees
 - find algorithms that are fast on **typical** (not worst case) cases

CSP Examples



Example: Map Coloring

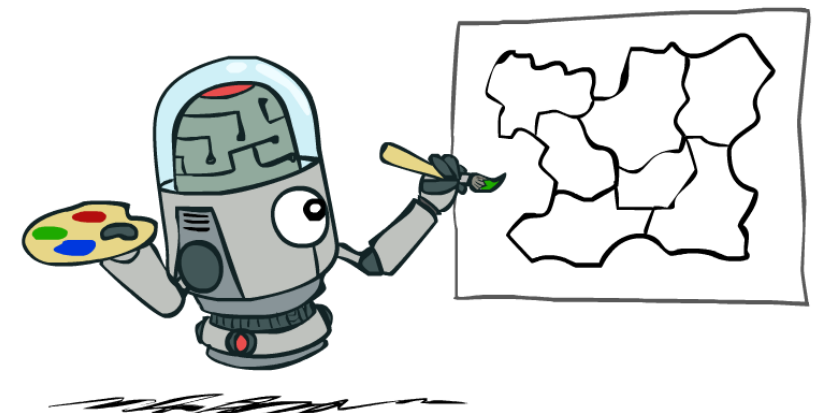
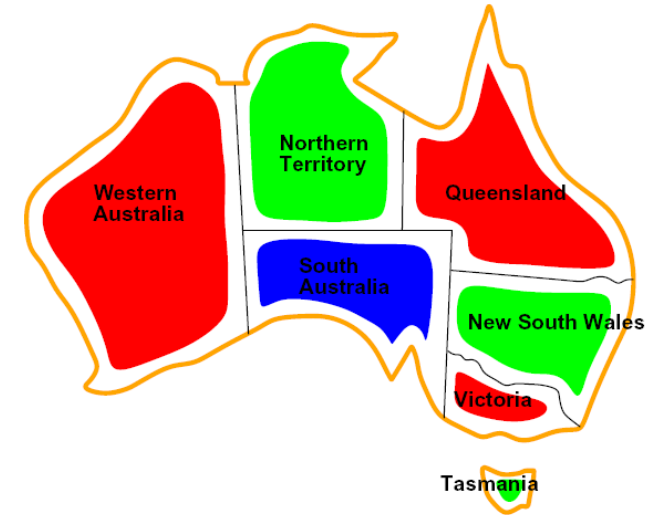
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

Explicit: $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$

- Solutions are assignments satisfying all constraints, e.g.:

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

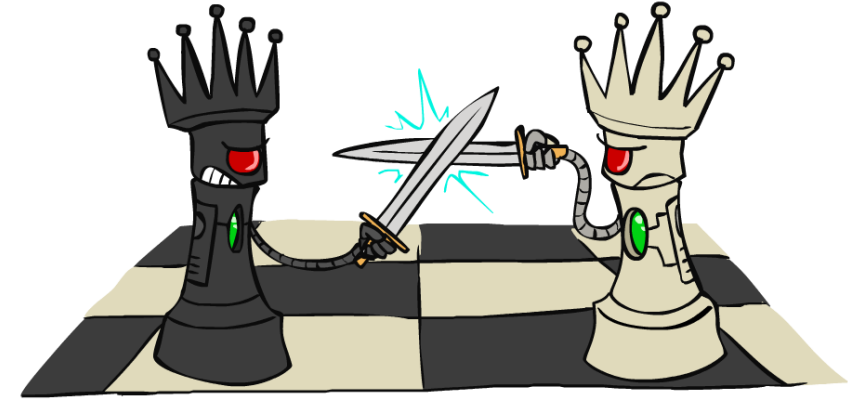
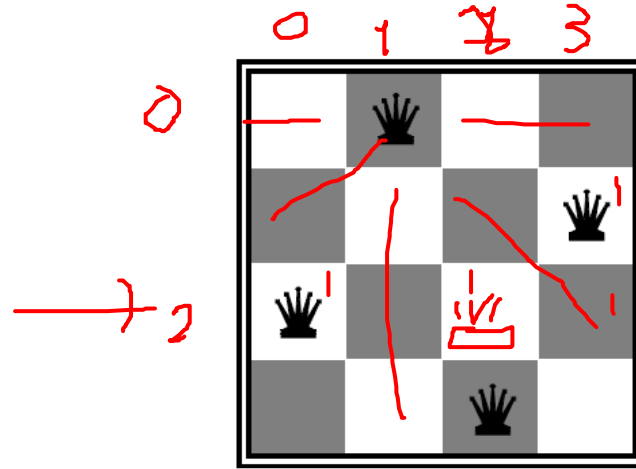


Example: N-Queens

4

■ Formulation 1:

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



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

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

Example: N-Queens

- Formulation 2:

- Variables: Q_k

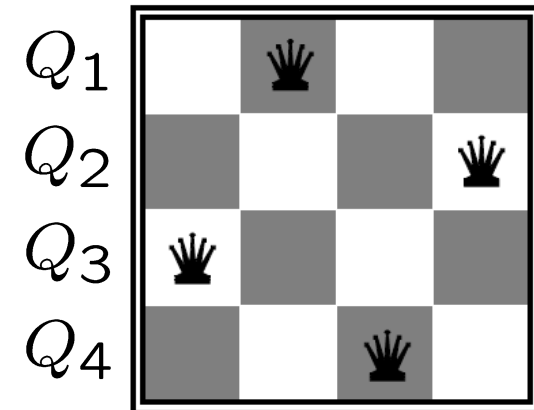
- Domains: $\{1, 2, 3, \dots, N\}$

- Constraints:

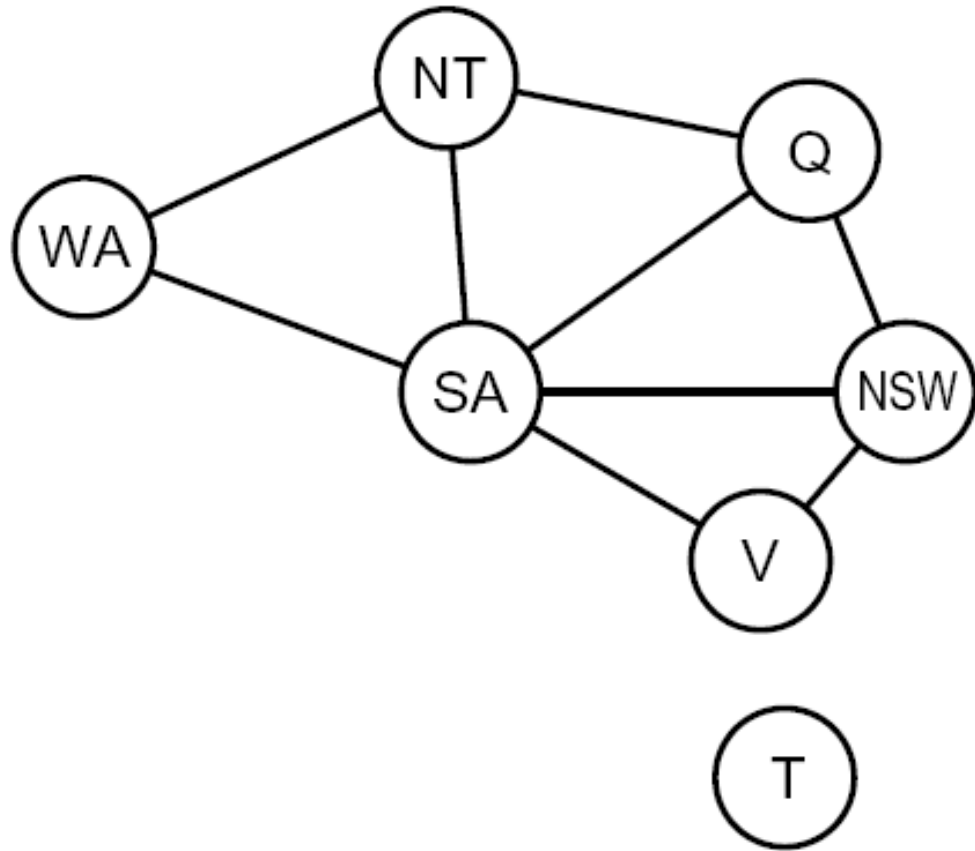
Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

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

...

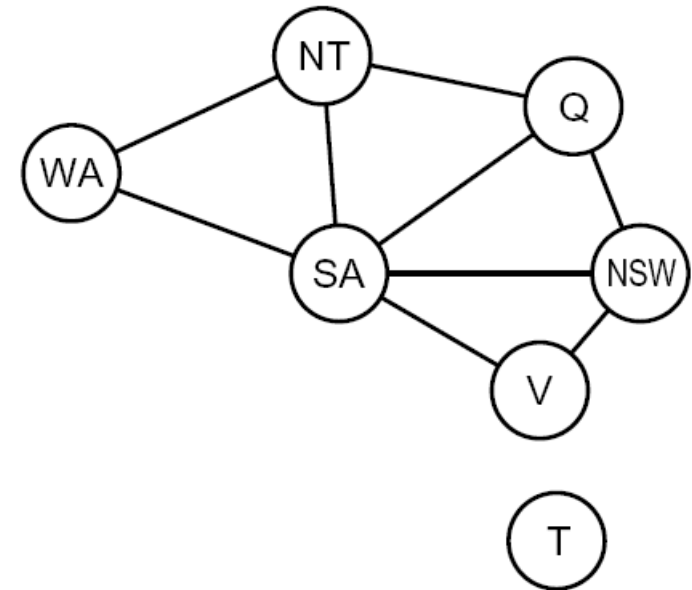


Constraint Graphs



Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



Example: Cryptarithmic

- Variables:

$F T U W R O \underline{X_1} \underline{X_2} \underline{X_3}$

- Domains:

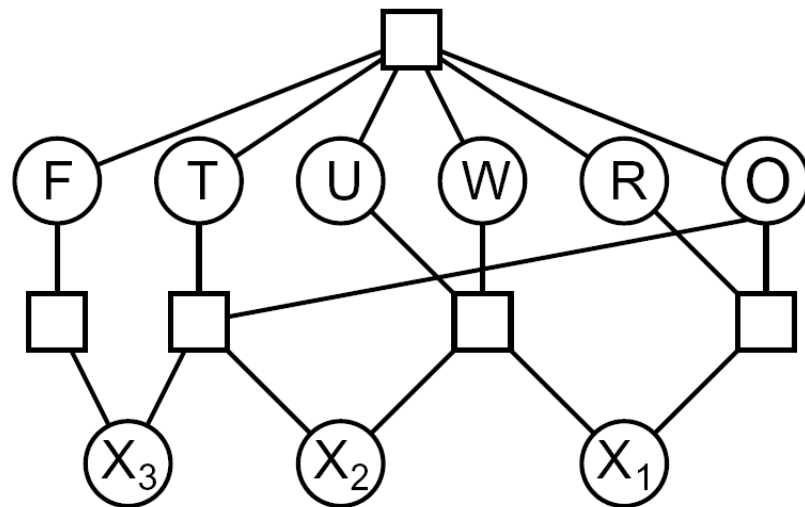
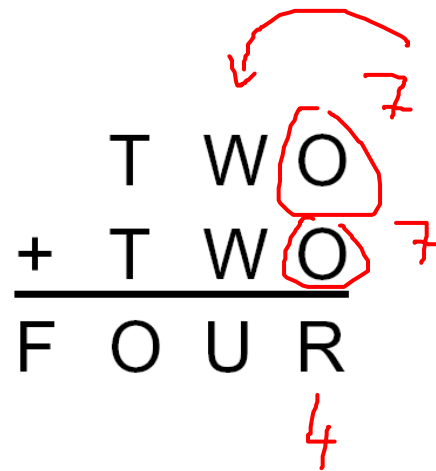
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- Constraints:

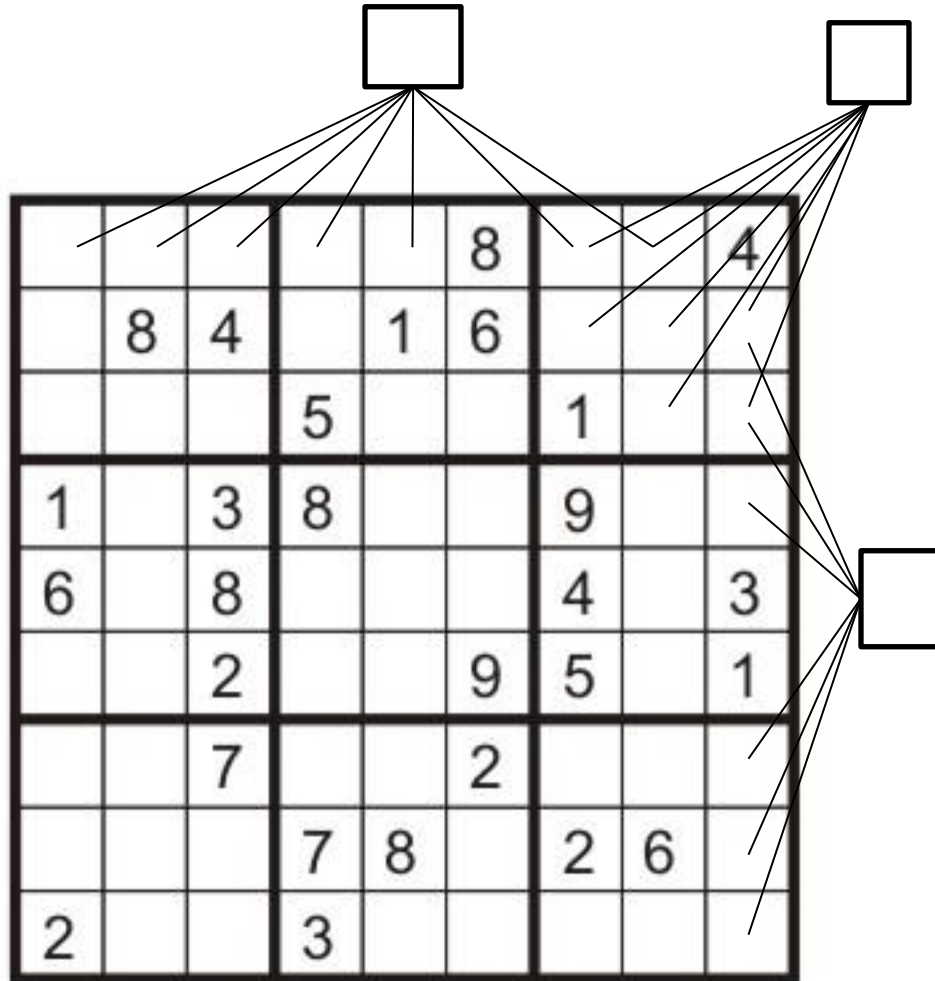
$\text{alldiff}(F, T, U, W, R, O)$

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

\dots



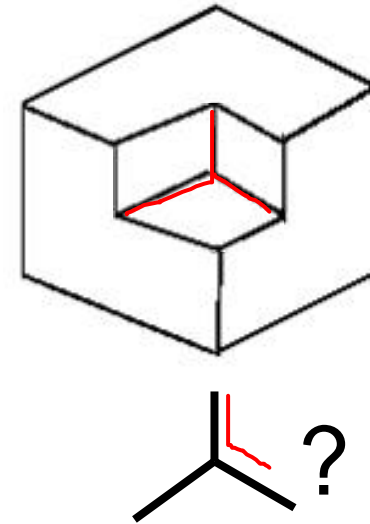
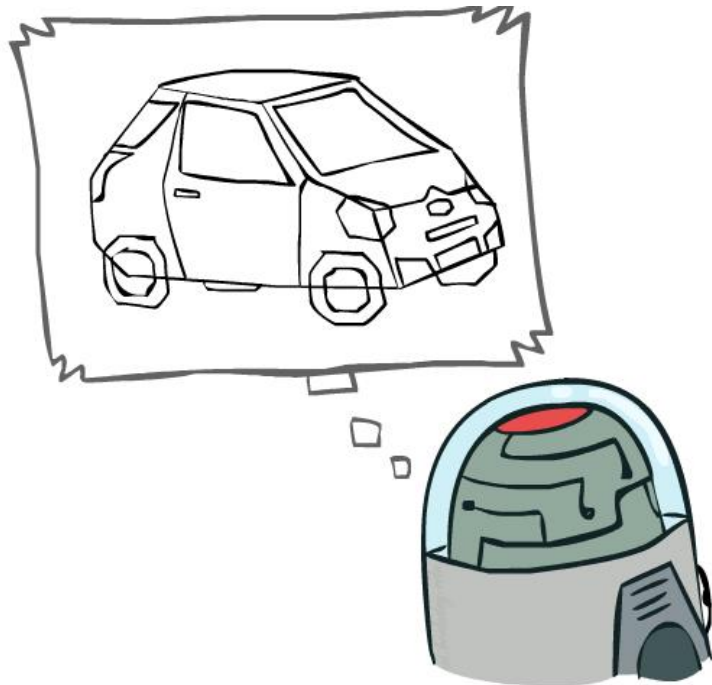
Example: Sudoku



- Variables:
 - Each (open) square
- Domains:
 - $\{1,2,\dots,9\}$
- 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)

Example: The Waltz Algorithm

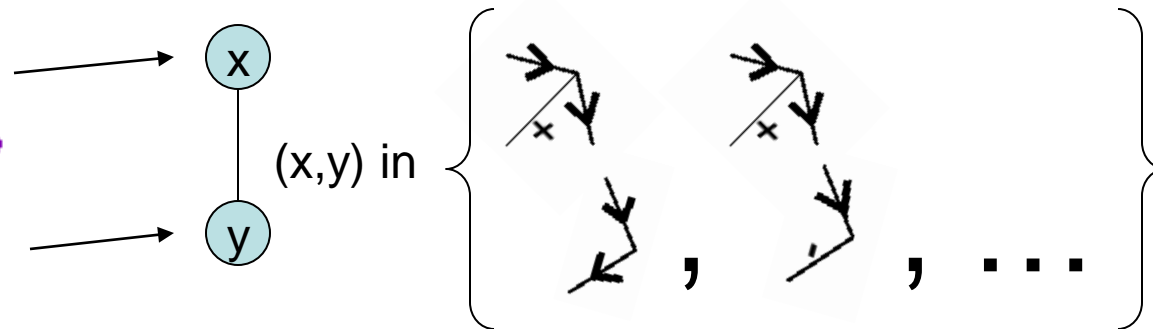
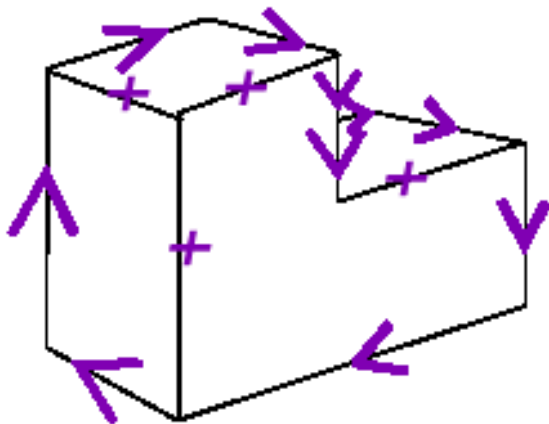
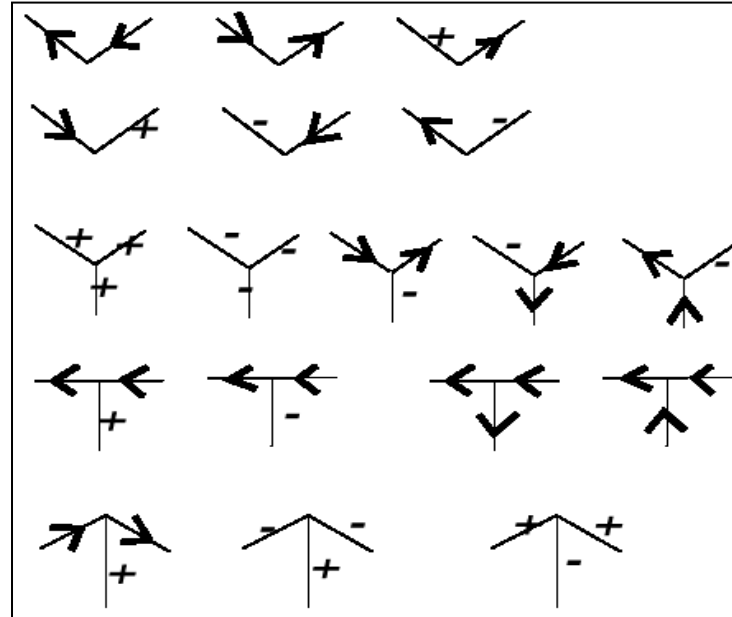
- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP



- Approach:
 - Each intersection is a variable
 - Adjacent intersections impose constraints on each other
 - Solutions are physically realizable 3D interpretations

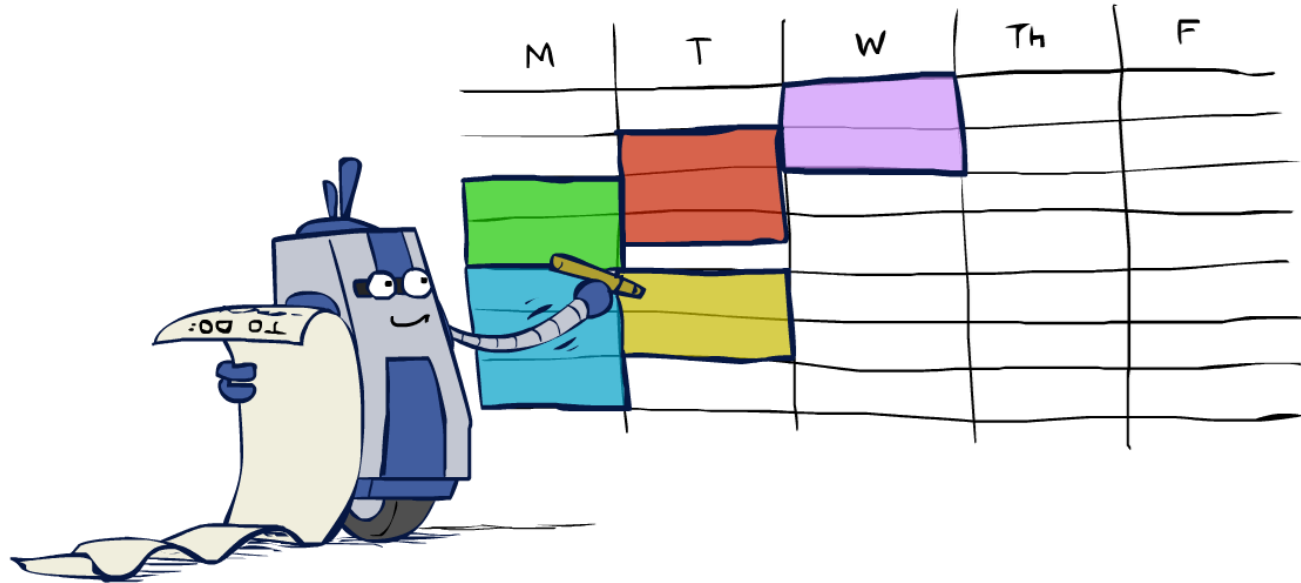
Legal Junctions

- Only certain junctions are physically possible
- How can we formulate a CSP to label an image?
- Variables: vertices
- Domains: junction labels
- Constraints: both ends of a line should have the same label



Real-World CSPs

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



- Many real-world problems involve real-valued variables...

Solving CSPs



Solving Constraint Satisfaction Problems (CSPs)

- A CSP can be solved using generate-and-test paradigm (GT) that systematically generates each possible value assignment and then it tests to see if it satisfies all the constraints.
- A more efficient method uses the backtracking paradigm (BT) that is the most common algorithm for performing systematic search. Backtracking incrementally attempts to extend a partial solution toward a complete solution, by repeatedly choosing a value for another variable.
- Two methods:
 - Generate & Test
 - Graph search with backtracking paradigm (BT)

Generate and Test (GT) Algorithms

- Systematically check all possible worlds
 - Possible worlds: cross product of domains $\text{dom}(V_1) \times \text{dom}(V_2) \times \dots \times \text{dom}(V_n)$
- Generate and Test:
 - **Generate** possible worlds one at a time
 - **Test** constraints for each one.

Example: 3 variables A,B,C

```
For a in dom(A)
  For b in dom(B)
    For c in dom(C)
      if {A=a, B=b, C=c} satisfies all constraints
        return {A=a, B=b, C=c}
fail
```

- If there are k variables, each with domain size d , and there are c constraints, the complexity of Generate & Test is

$O(ckd)$

$O(ck^d)$

$O(cd^k)$

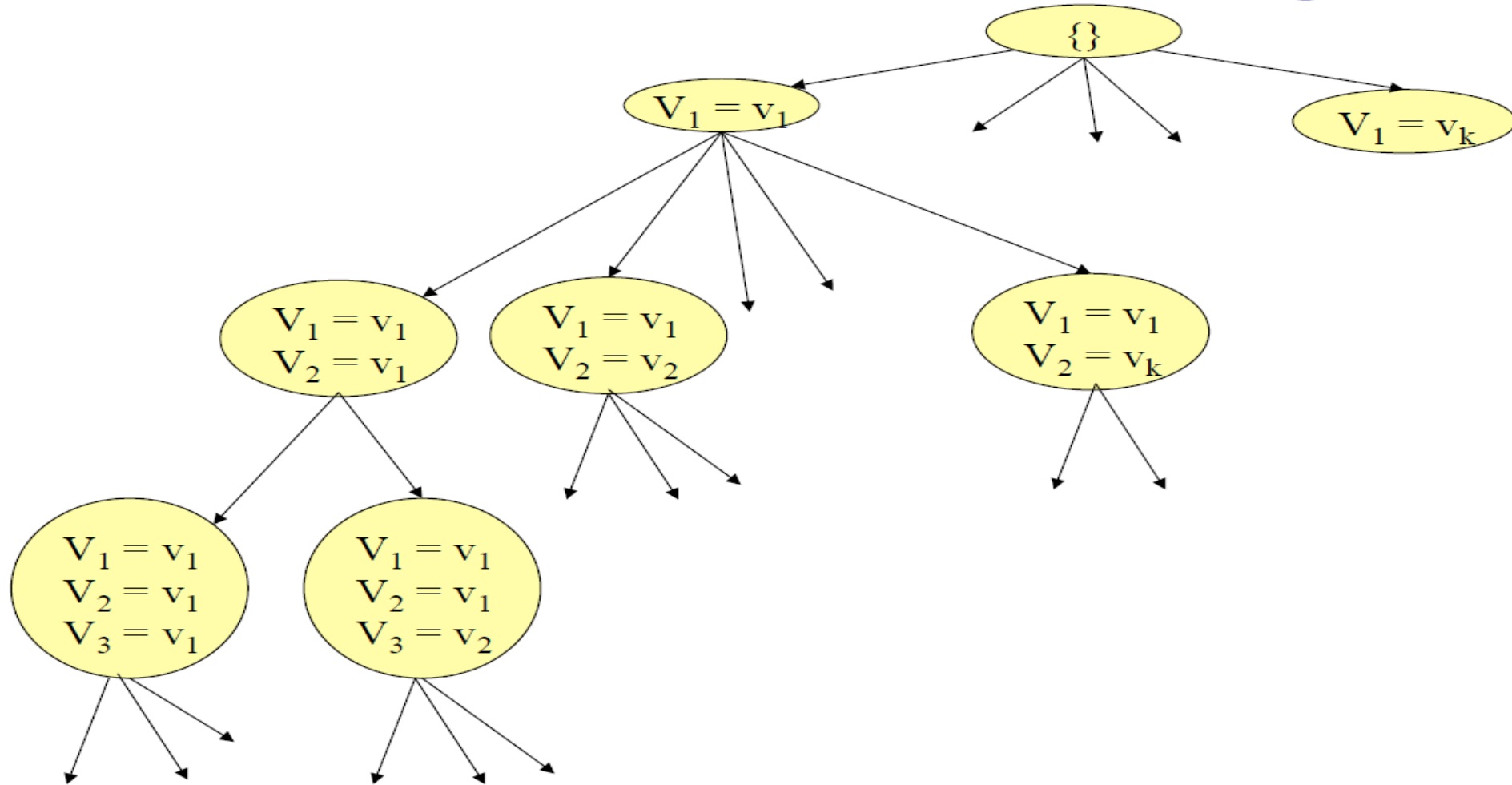
$O(d^{ck})$

- There are d^k possible worlds
- For each one need to check c constraints

CSP as a Search Problem: one formulation

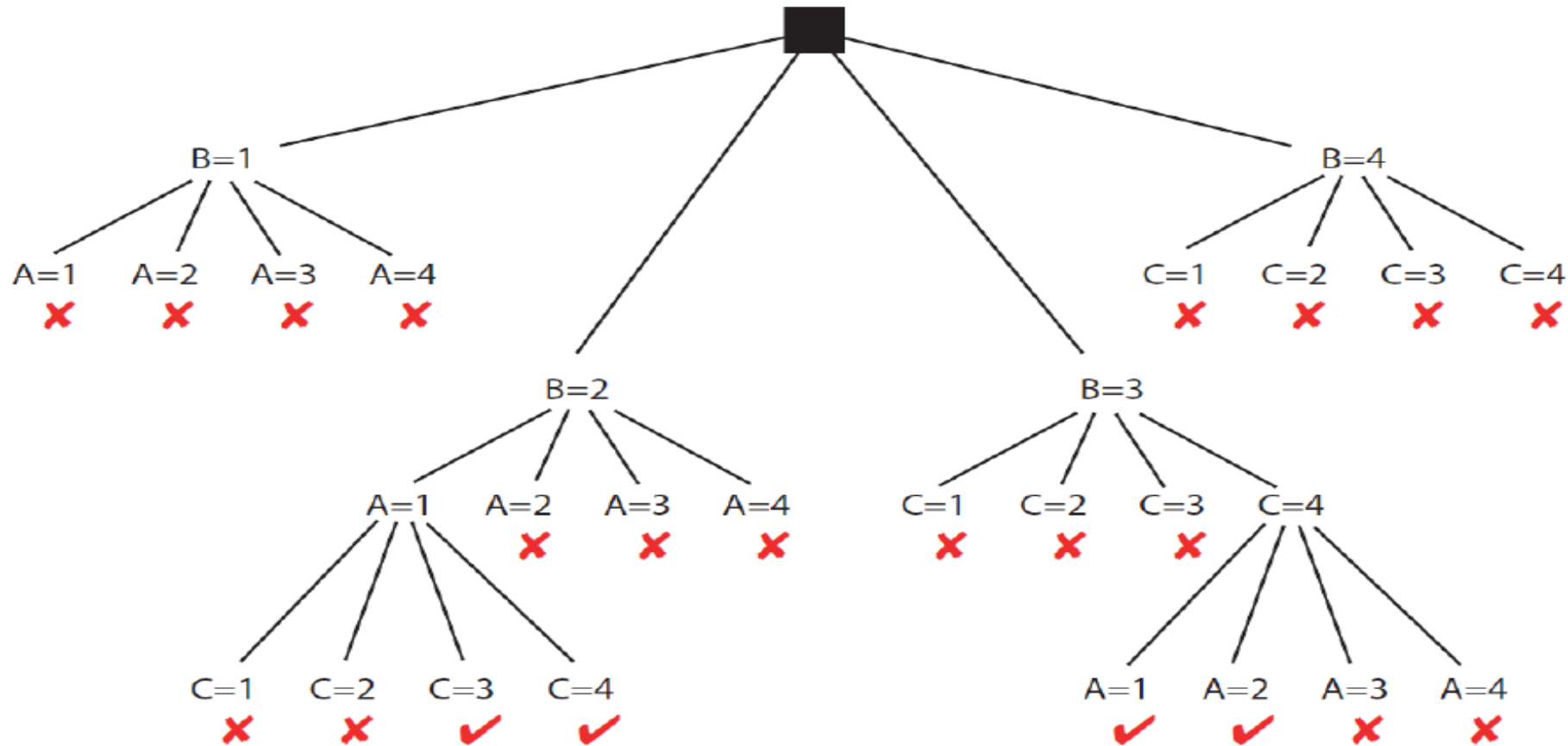
- States: **partial assignment** of values to variables
- Start state: empty assignment
- Successor function: states with the next variable assigned
 - E.g., follow a total order of the variables V_1, \dots, V_n
 - A state assigns values to the first k variables:
 - $\{V_1 = v_1, \dots, V_k = v_k\}$
 - Neighbors of node $\{V_1 = v_1, \dots, V_k = v_k\}$:
nodes $\{V_1 = v_1, \dots, V_k = v_k, V_{k+1} = x\}$ for each $x \in \text{dom}(V_{k+1})$
- Goal state: **complete assignments** of values to variables that **satisfy all constraints**
 - That is, **models**
- Solution: assignment (the path doesn't matter)

CSP as Graph Searching



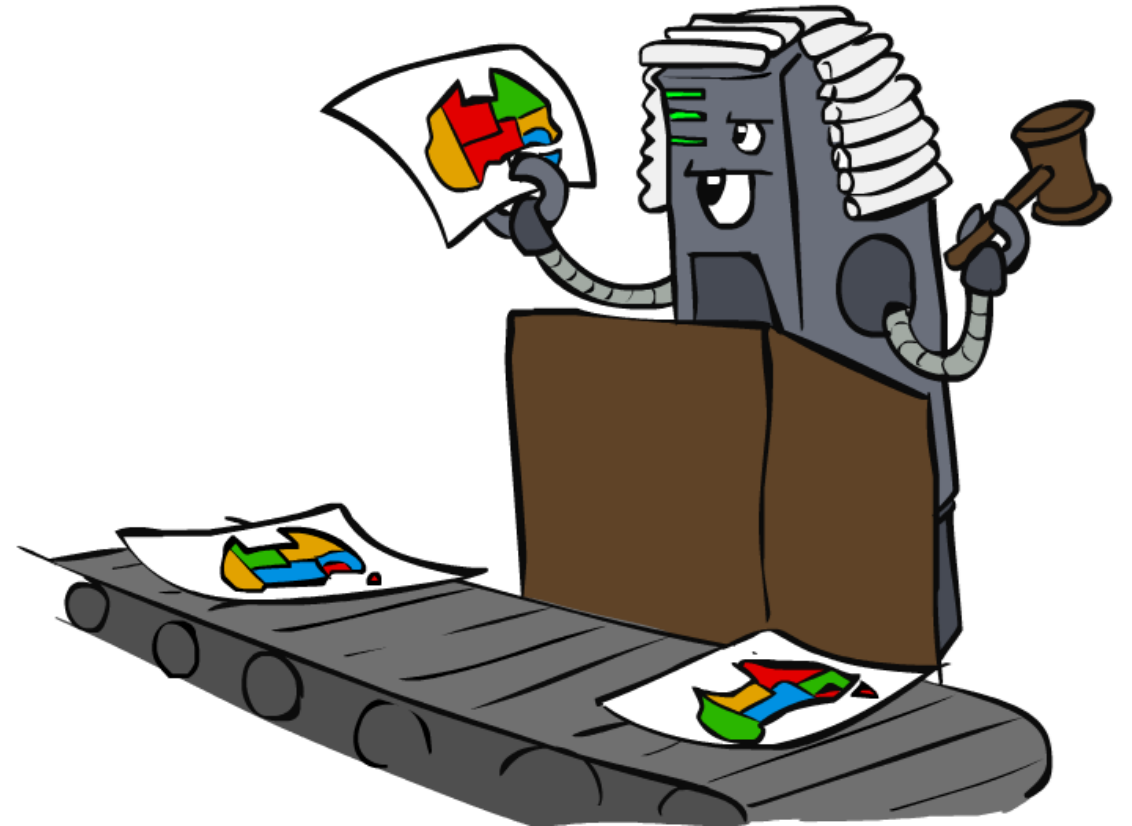
CSP as Graph Searching

- 3 Variables: A,B,C. All with domains = {1,2,3,4}
- Constraints: $A < B$, $B < C$



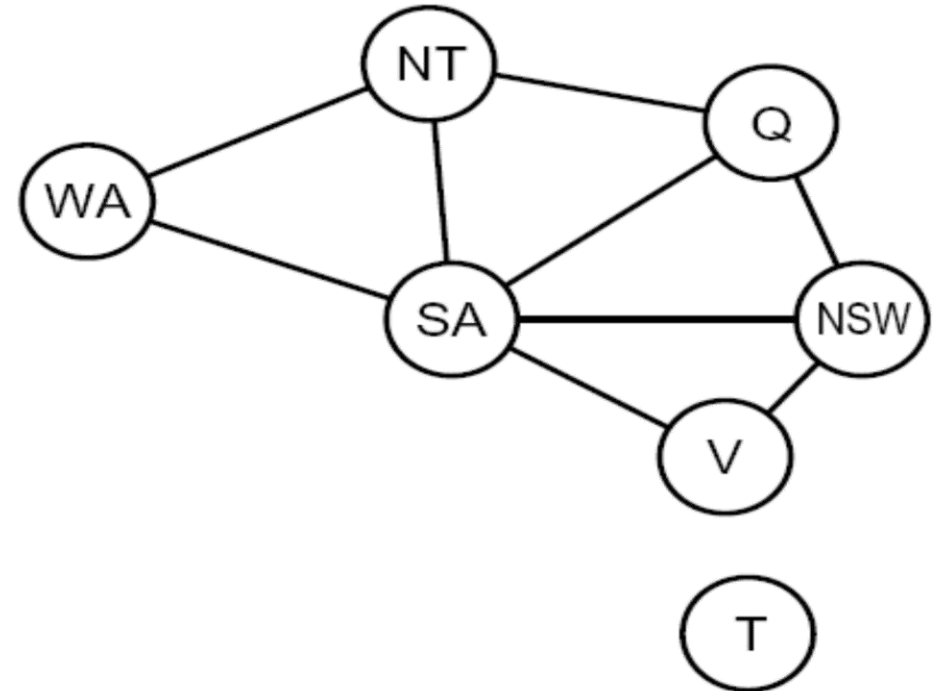
Standard Search Formulation

- 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

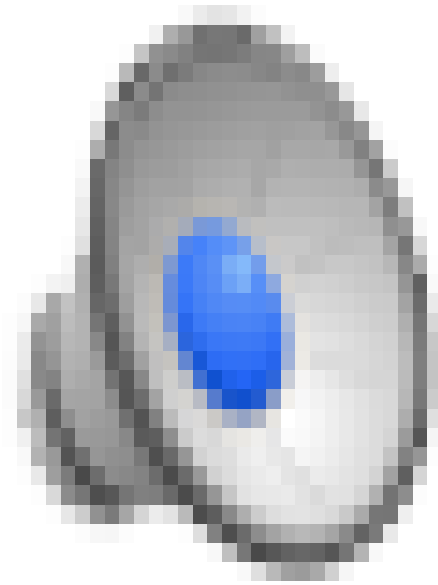
- What would BFS do?
- What would DFS do?



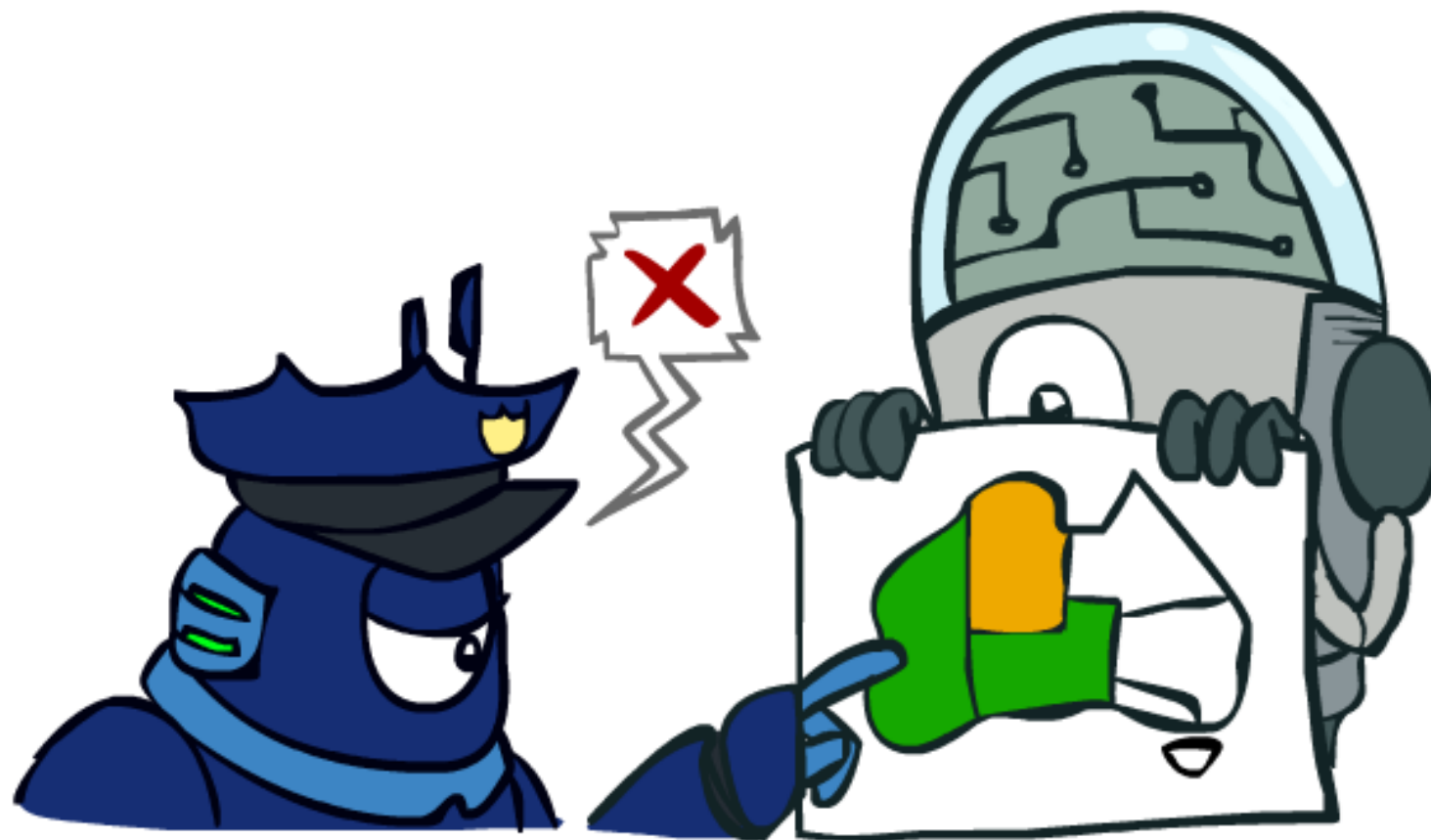
- What problems does naïve search have?
 - For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n .
 - The number of leaves is d^n

Video of Demo Coloring – DFS

https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

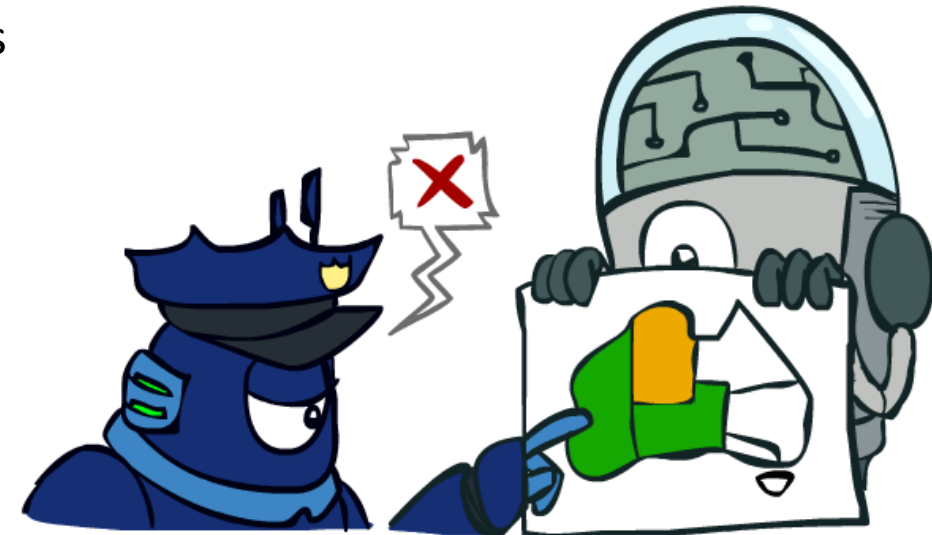


Backtracking Search



Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - I.e. consider only values which do not conflict with 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* (not the best name)
- Can solve n-queens for $n \approx 25$



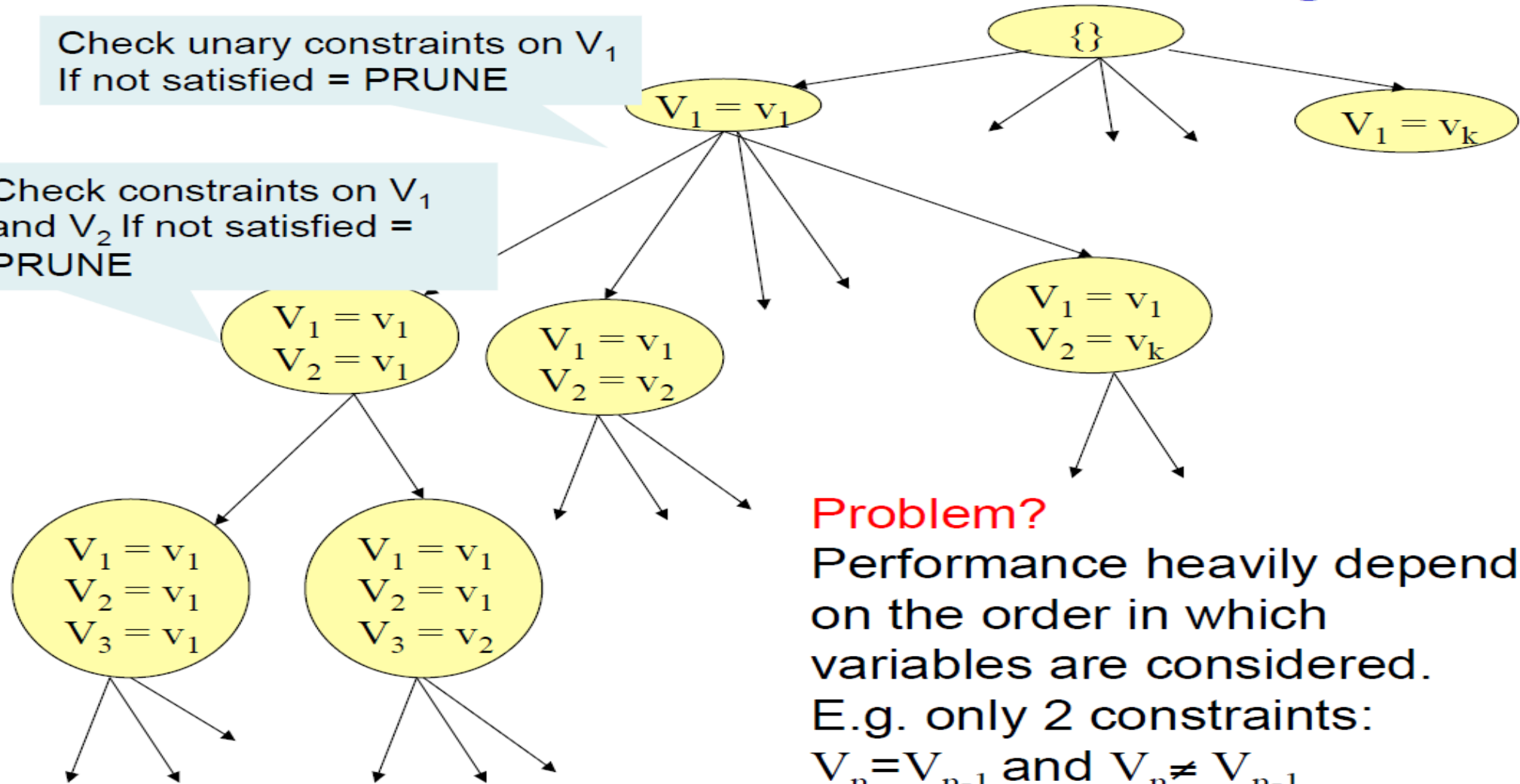
Backtracking Search

- Explore search space via DFS but evaluate each constraint as soon as all its variables are bound.
- Any partial assignment that doesn't satisfy the constraint can be pruned.
- Example:
 - 3 variables A, B, C each with domain {1,2,3,4}
 - {A = 1, B = 1} is inconsistent with constraint $A \neq B$ regardless of the value of the other variables
 - ⇒ Fail! Prune!

CSP as Graph Searching

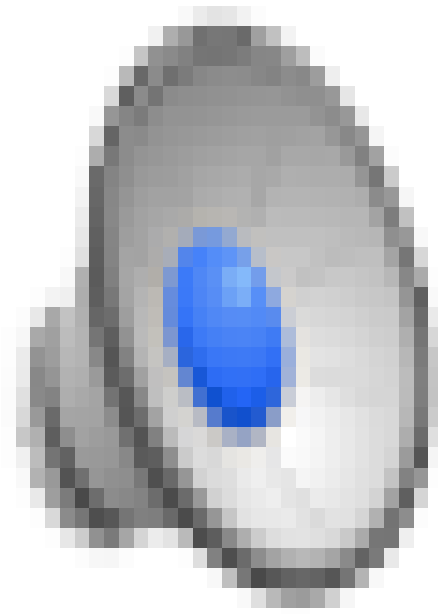
Check unary constraints on V_1
If not satisfied = PRUNE

Check constraints on V_1
and V_2 If not satisfied =
PRUNE

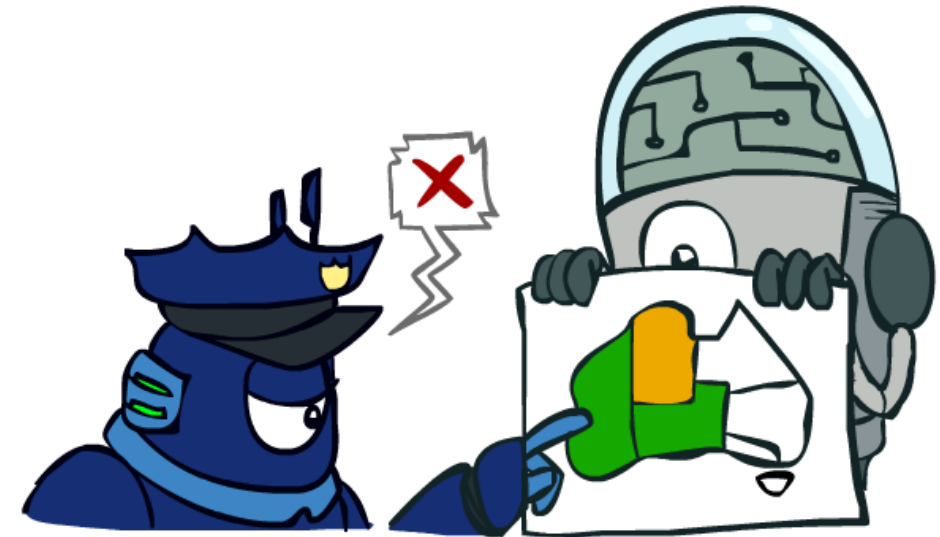
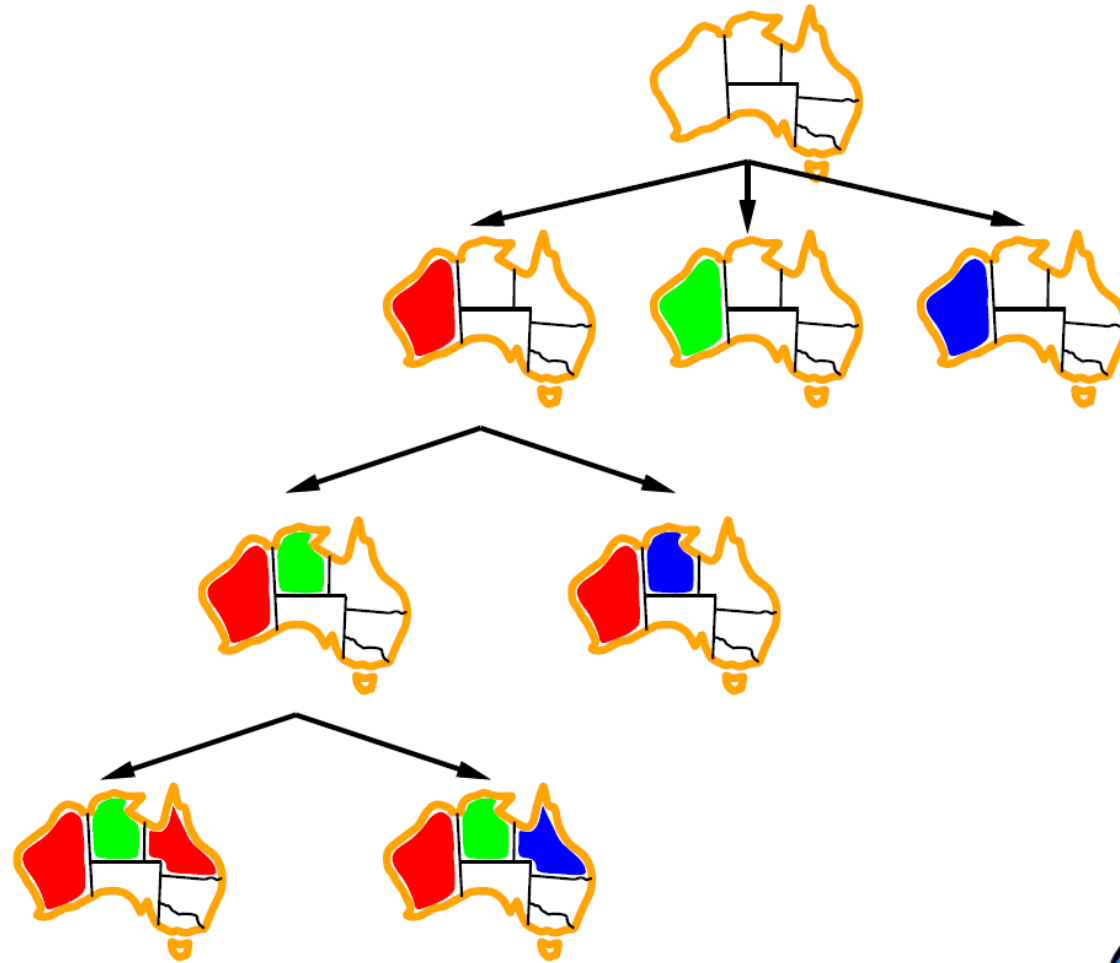


Video of Demo Coloring – Backtracking

https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html



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

https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

[Demo: coloring -- backtracking]

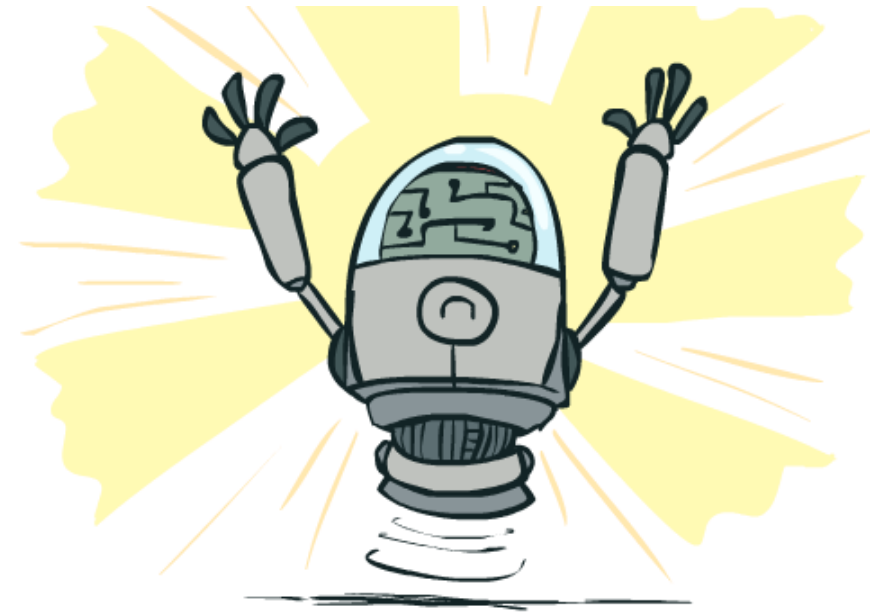
Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure  
  return BACKTRACK(csp, { })  
  
function BACKTRACK(csp, assignment) returns a solution or failure  
  if assignment is complete then return assignment  
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)  
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do  
    if value is consistent with assignment then  
      add {var = value} to assignment  
      inferences ← INFERENCE(csp, var, assignment)  
      if inferences ≠ failure then  
        add inferences to csp  
        result ← BACKTRACK(csp, assignment)  
        if result ≠ failure then return result  
        remove inferences from csp  
      remove {var = value} from assignment  
  return failure
```

Figure 5.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES implement the general-purpose heuristics discussed in Section 5.3.1. The INFERENCE function can optionally impose arc-, path-, or k -consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are retracted and a new value is tried.

Improving Backtracking

- General-purpose ideas give huge gains in speed
- 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?



Filtering



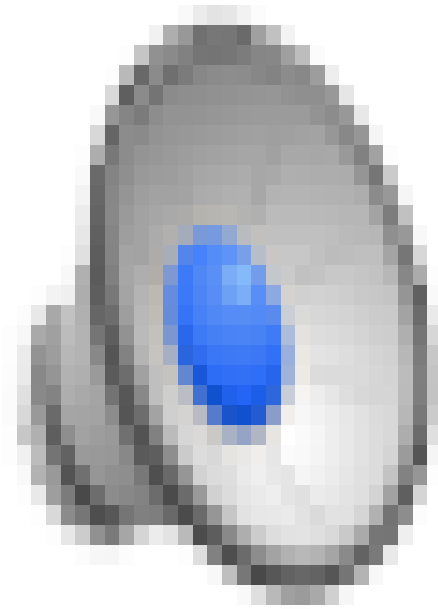
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



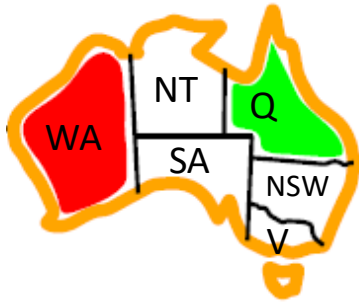
Video of Demo Coloring – Backtracking with Forward Checking

https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html



Filtering: Constraint Propagation

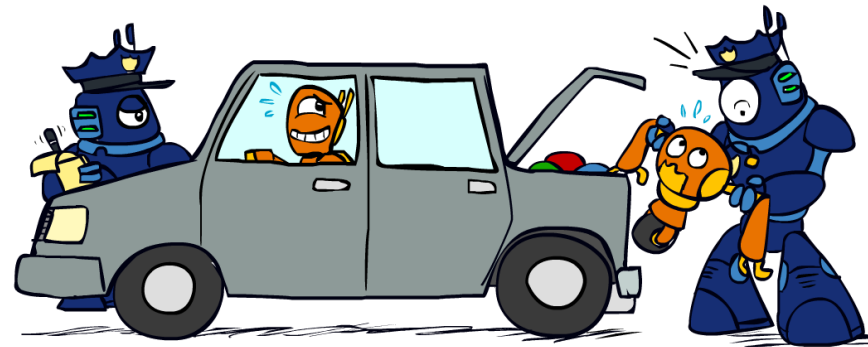
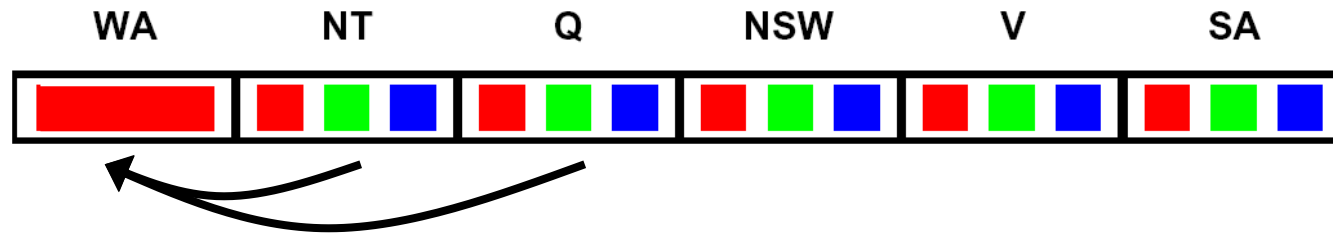
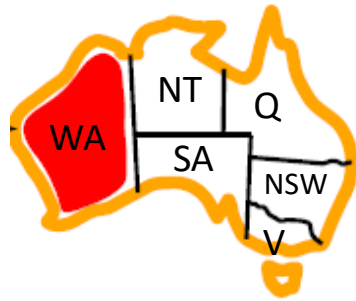
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- 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!

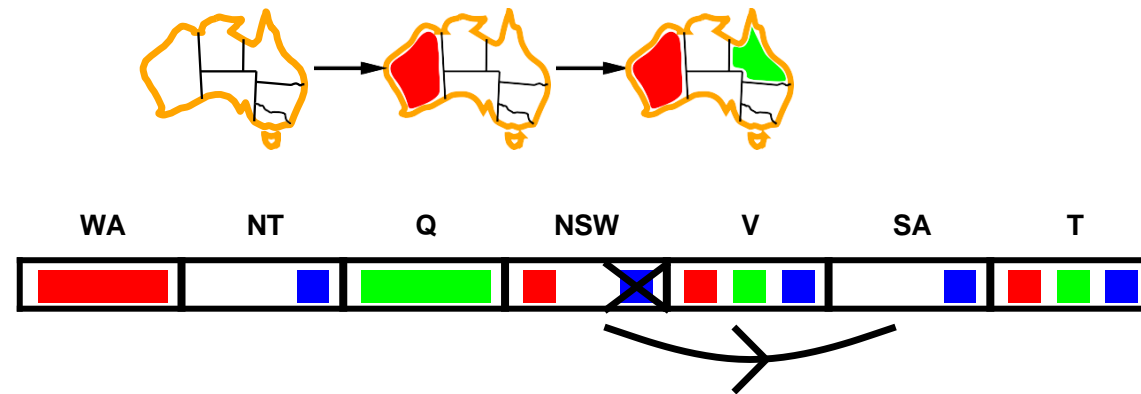
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y

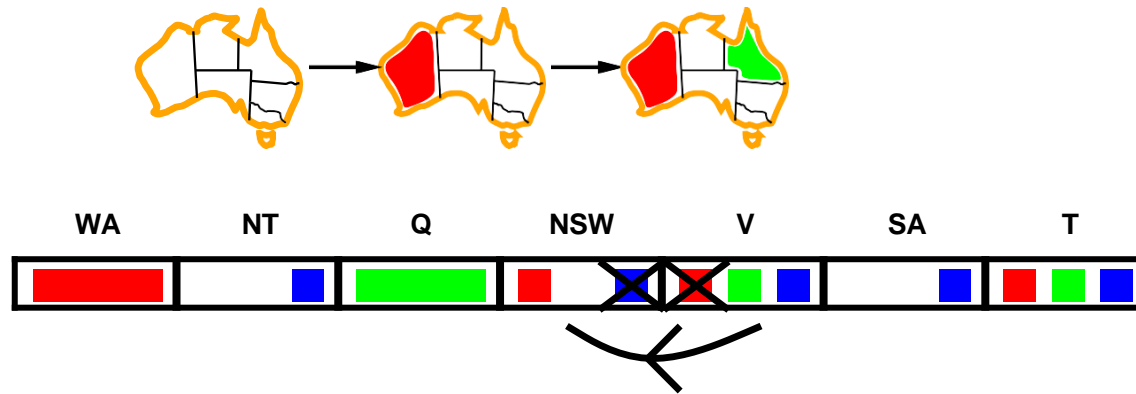


Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y



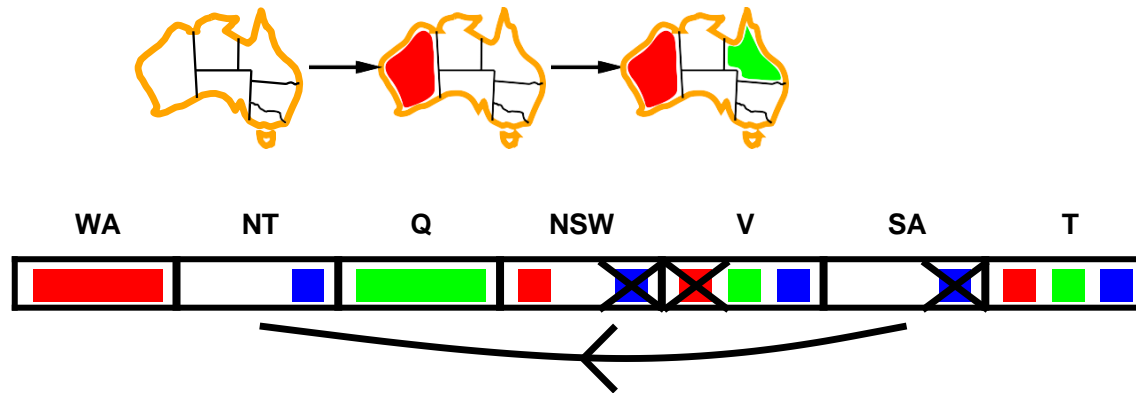
If X loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y



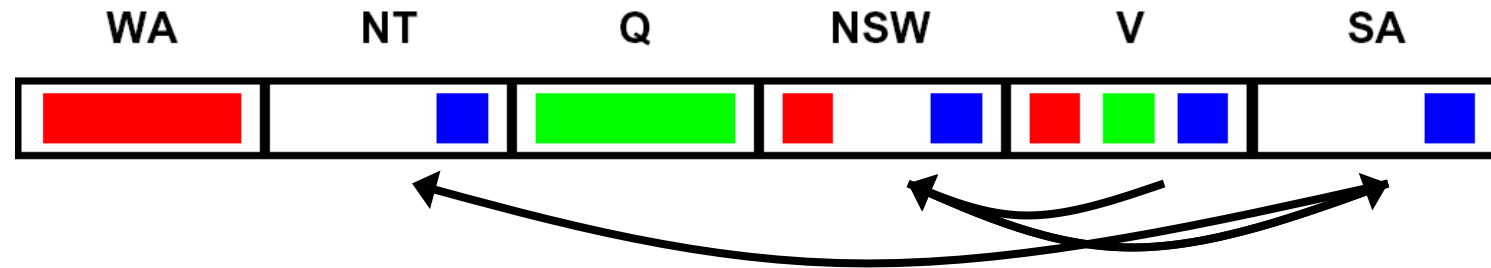
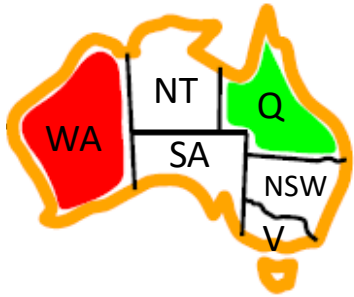
If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc Consistency of an Entire CSP

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



- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

*Remember:
Delete from
the tail!*

Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
   $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add  $(X_k, X_i)$  to queue

```

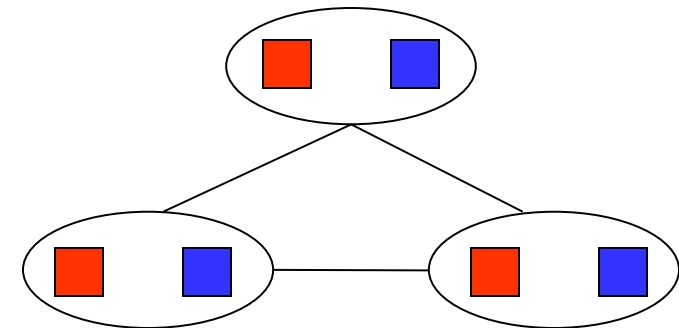
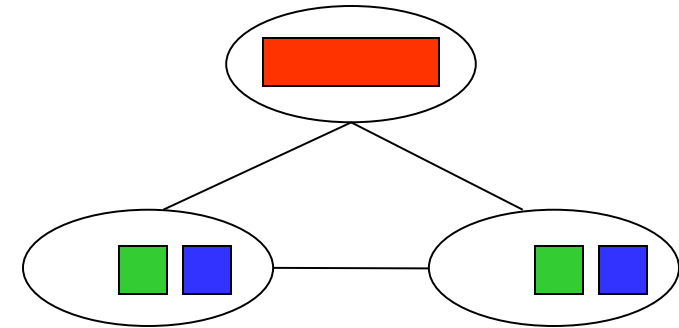
```
function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed

```

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

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?

[Demo: coloring -- forward checking]

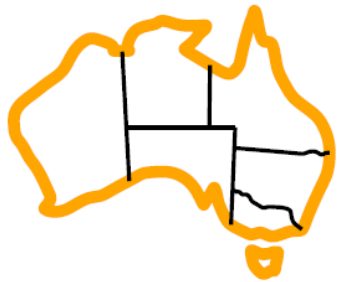
[Demo: coloring -- arc consistency]

Ordering

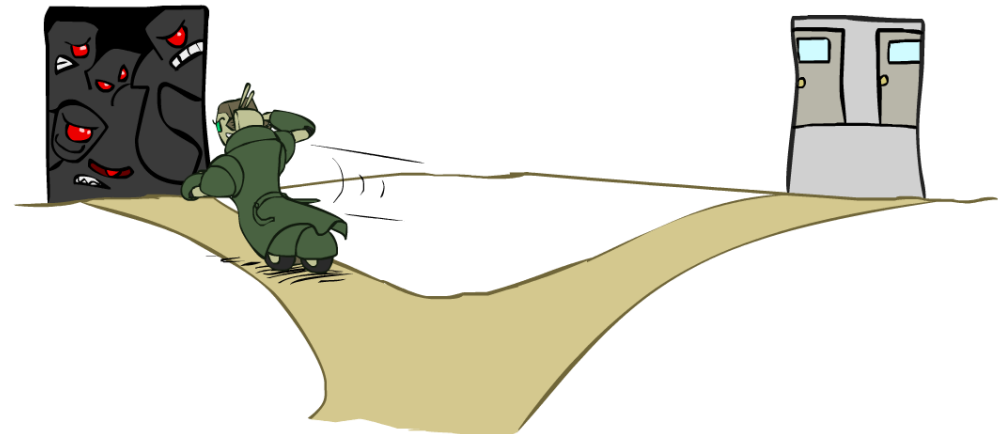


Ordering: Minimum Remaining Values

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

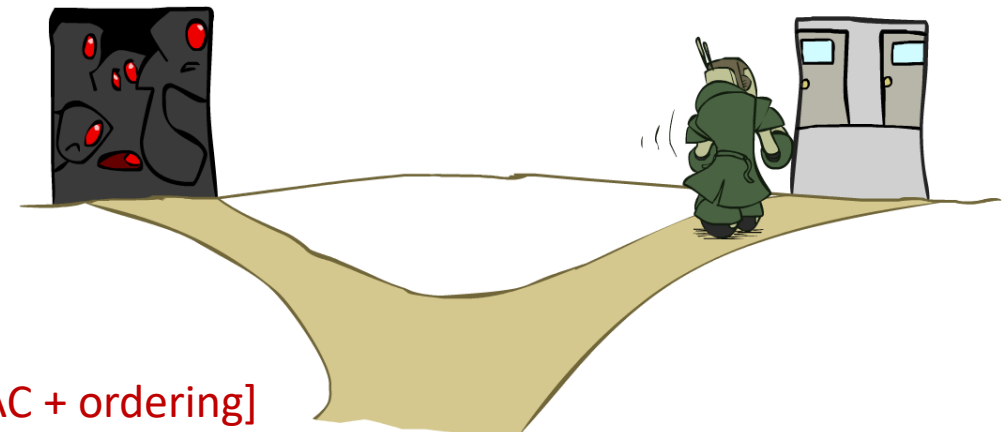
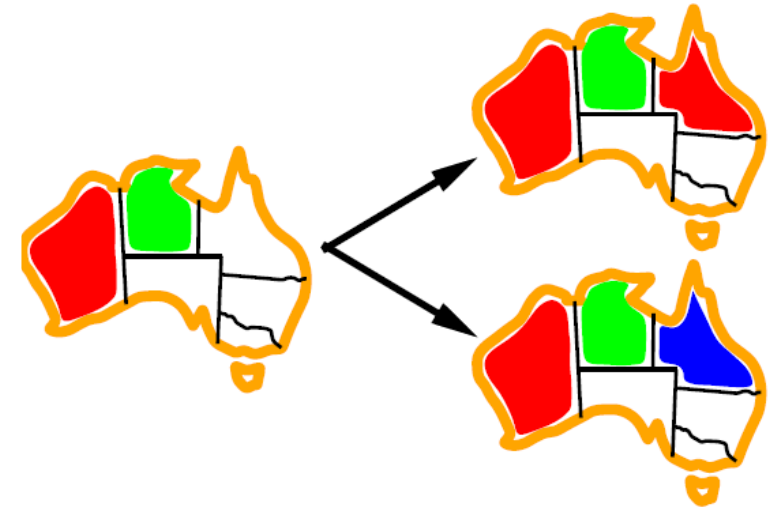


- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
 - Given a choice of variable, choose the *least constraining value*
 - I.e., the one that rules out the fewest values in the remaining variables
 - Note that it may take some computation to determine this! (E.g., rerunning filtering)
- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible

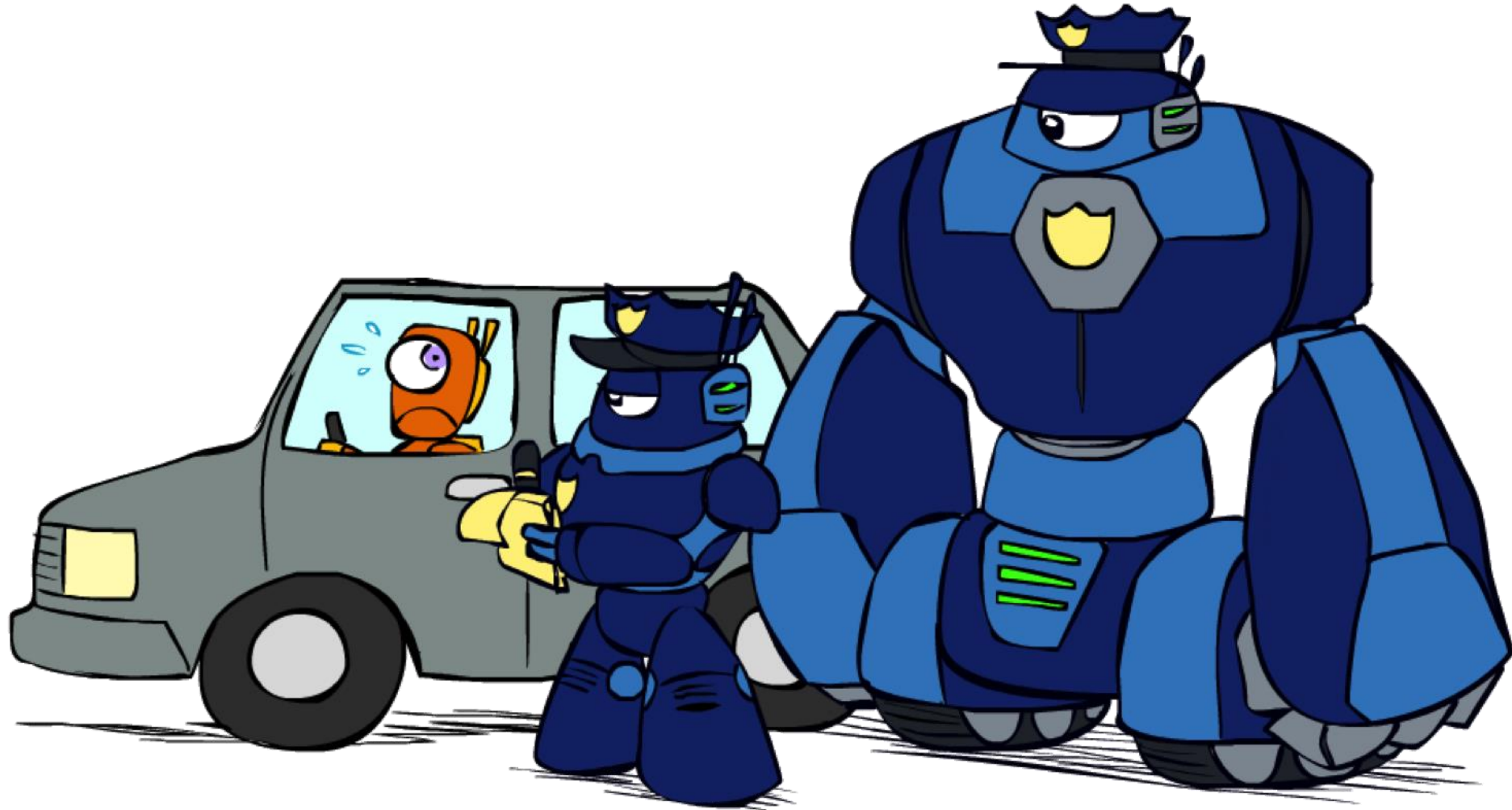


[Demo: coloring – backtracking + AC + ordering]

Demo: Backtracking + AC + Ordering

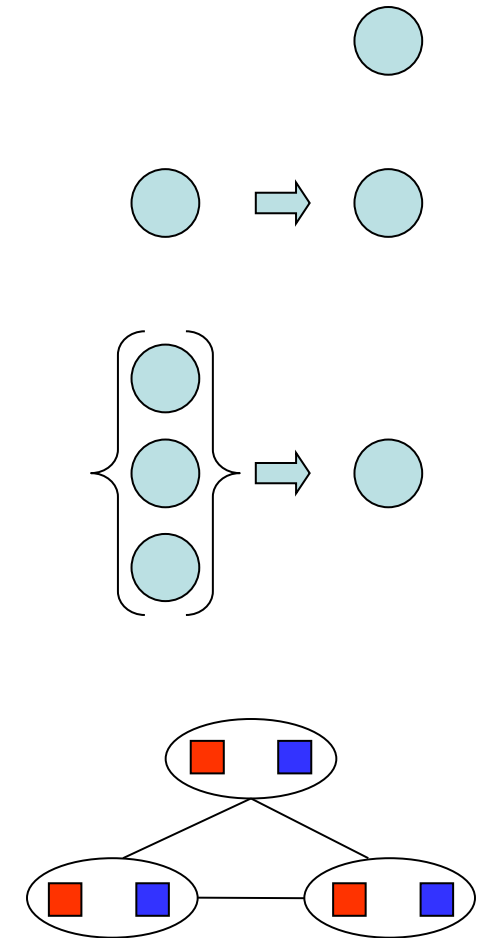
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

K-Consistency



K-Consistency

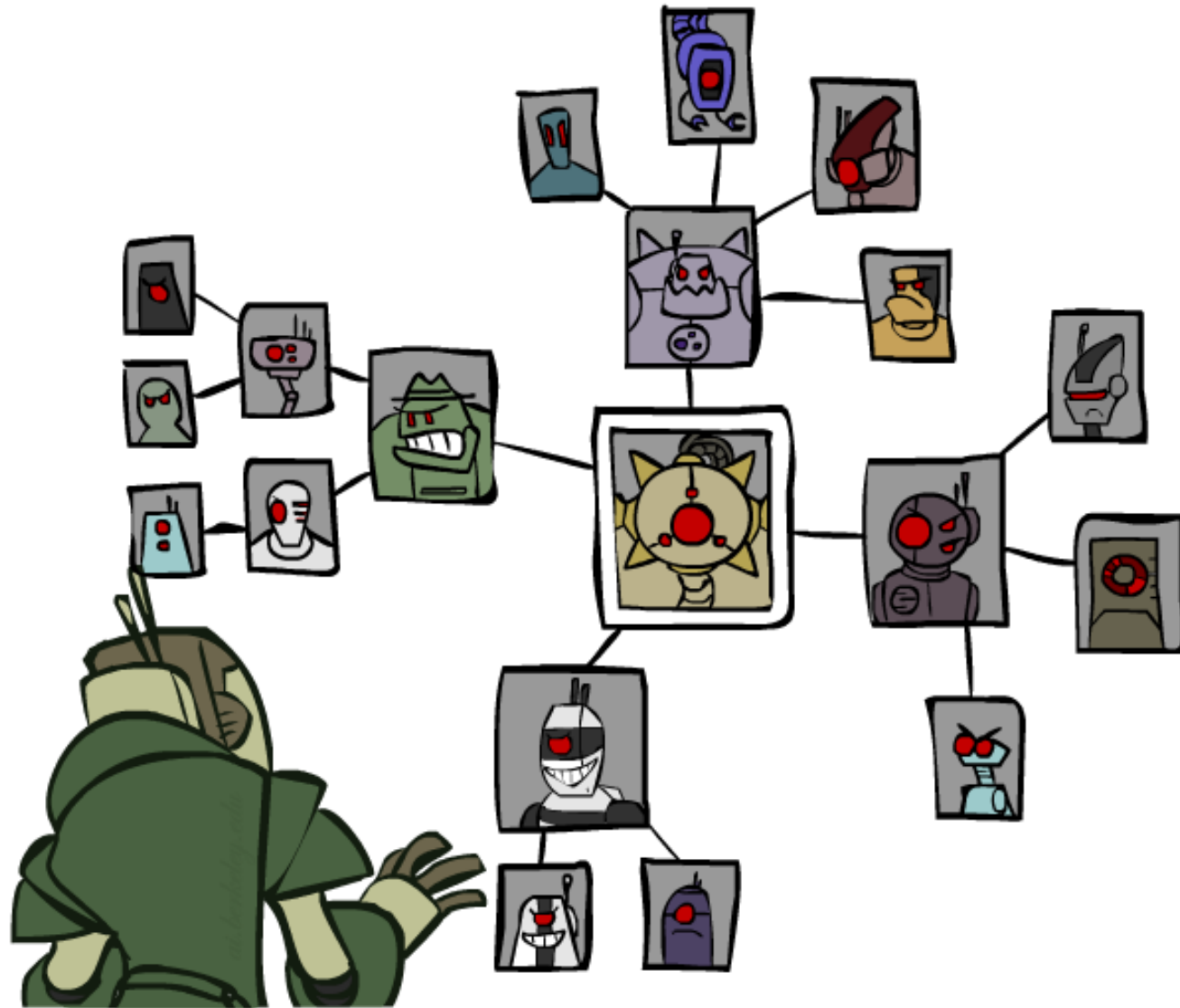
- Stronger forms of propagation can be defined with the notion of k-consistency
- Increasing degrees of consistency
 - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
 - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent 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
- In practice, determining the appropriate level of consistency checking is mostly an empirical science. Computing 2-consistency is common, and 3-consistency less common.
- (You need to know the k=2 case: arc consistency)



Strong K-Consistency

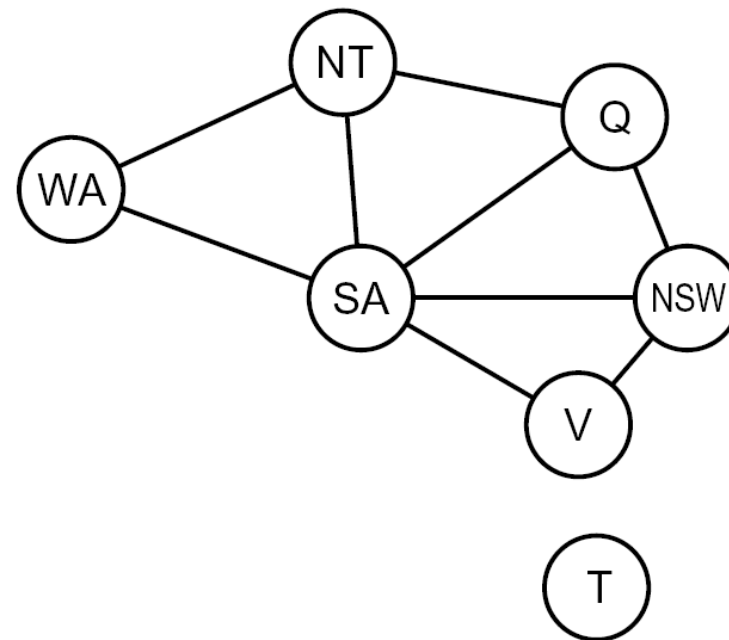
- Strong k-consistency: also k-1, k-2, ... 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
 - Choose any assignment to any variable
 - Choose a new variable
 - By 2-consistency, there is a choice consistent with the first
 - Choose a new variable
 - By 3-consistency, 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

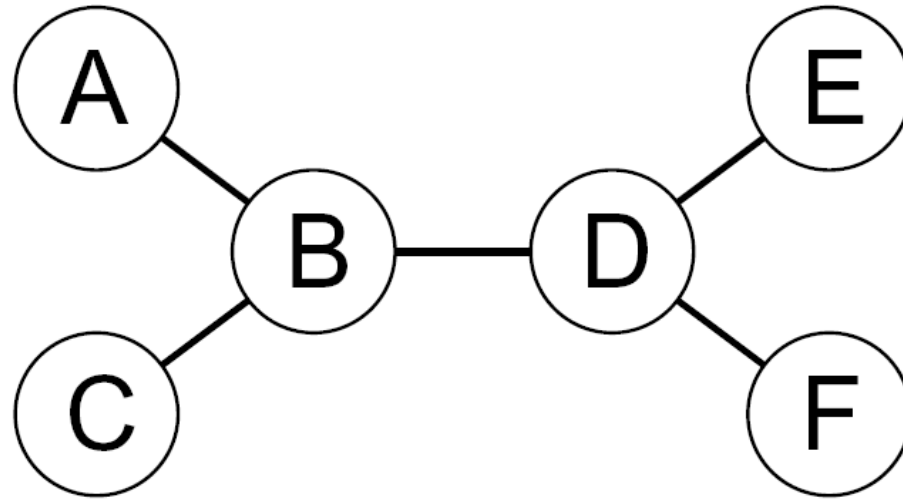


Problem Structure

- 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



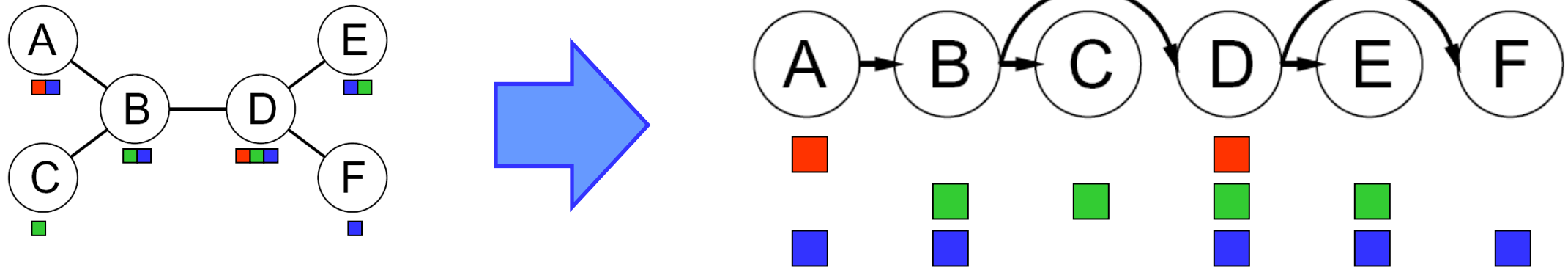
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time where n is the number of tree nodes and d is the size of the largest domain.
 - 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

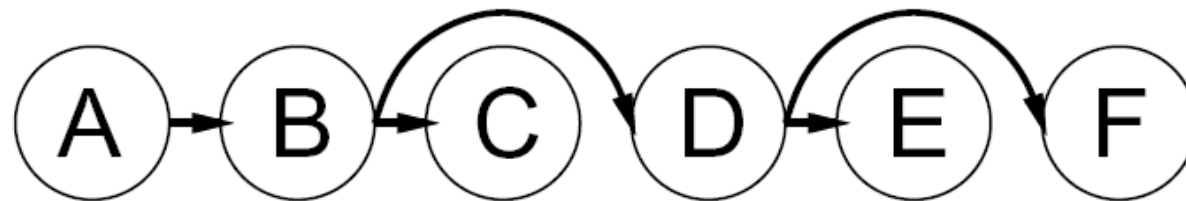


- Remove backward: For $i = n : 2$, apply $\text{RemoveInconsistent}(\text{Parent}(X_i), X_i)$
 - Assign forward: For $i = 1 : n$, assign X_i consistently with $\text{Parent}(X_i)$
- Runtime: $O(n d^2)$ (why?)



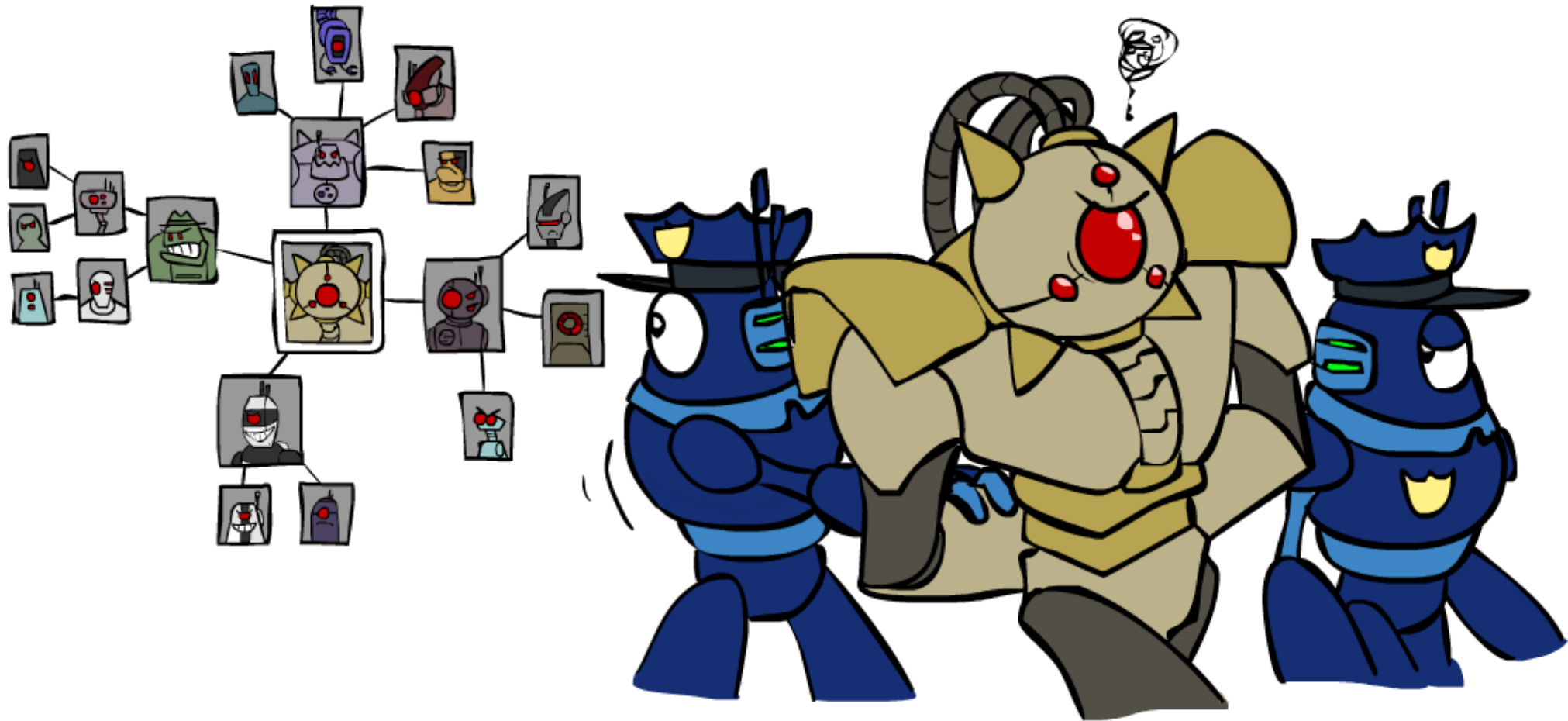
Tree-Structured CSPs

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

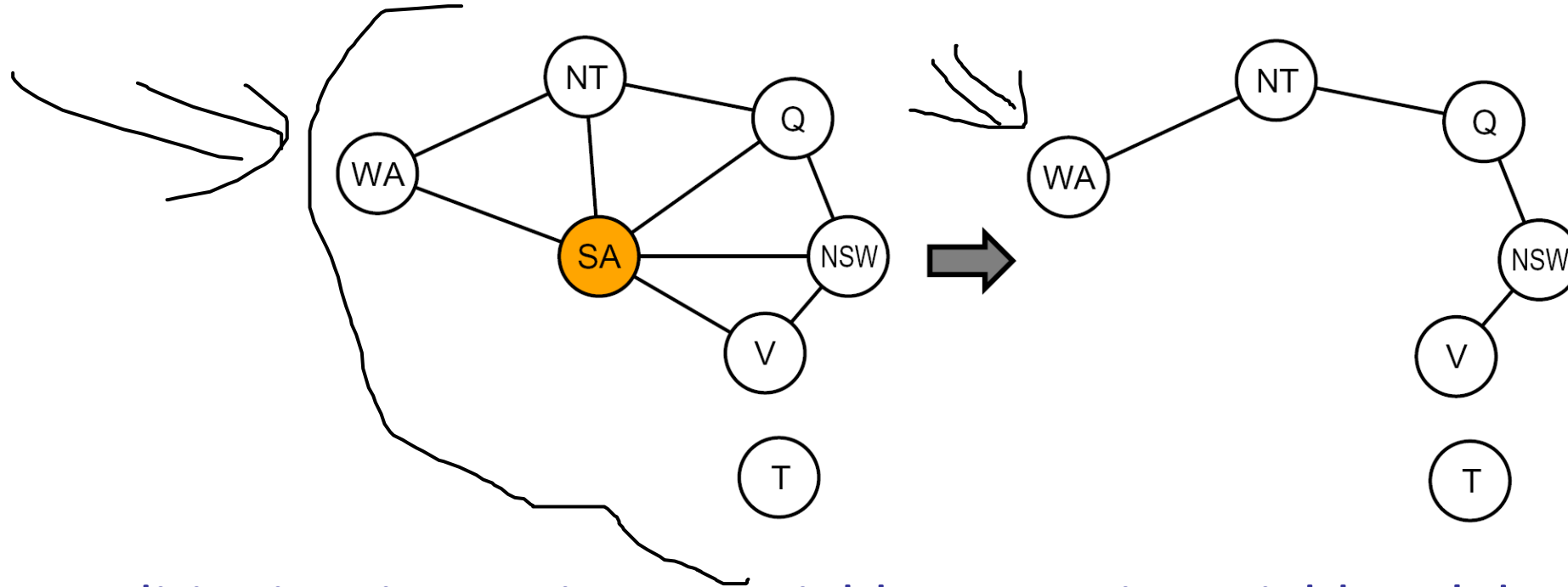


- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?
- Note: we'll see this basic idea again with Bayes' nets

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

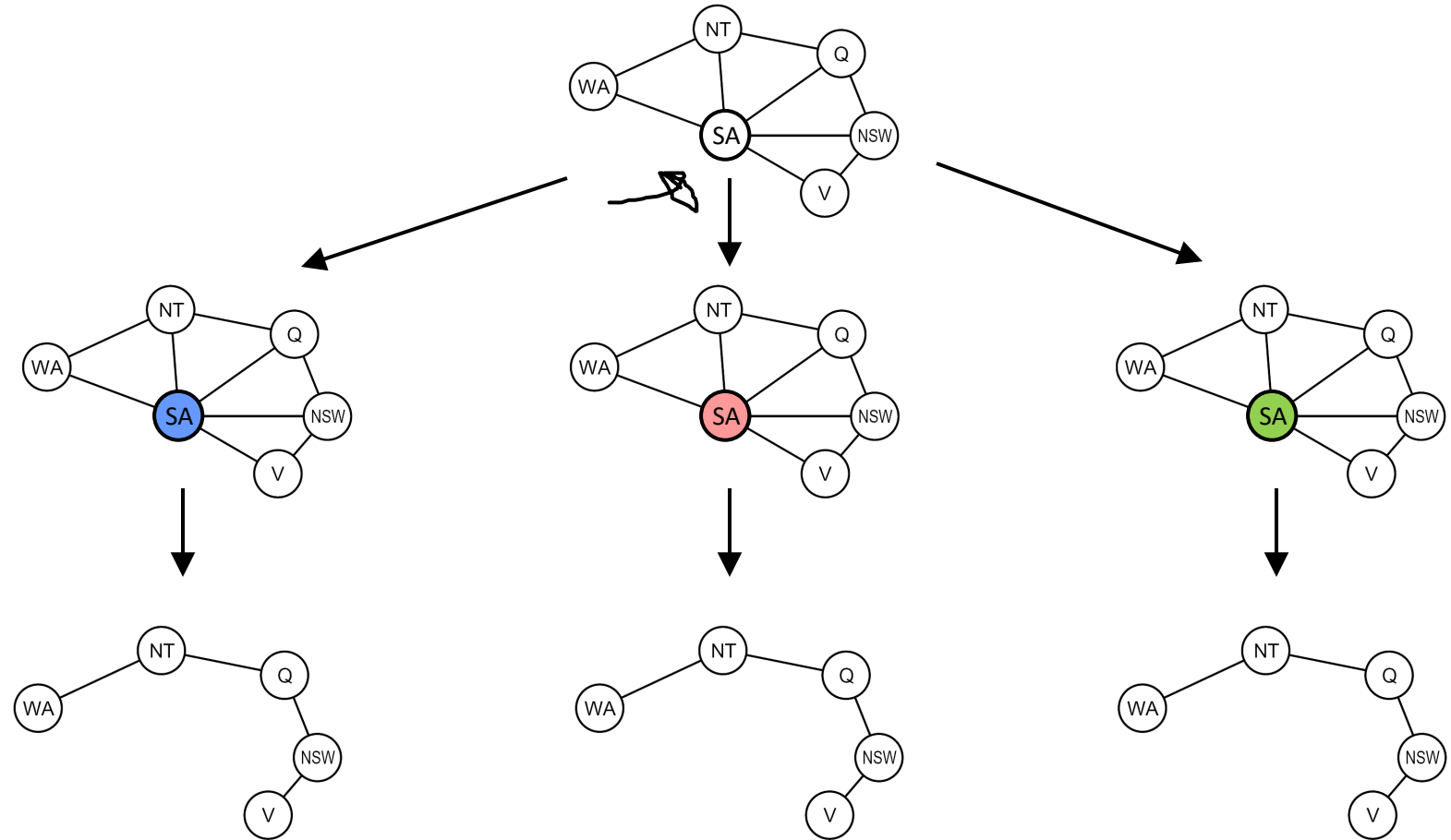
Cutset Conditioning

Choose a cutset

Instantiate the cutset
(all possible ways)

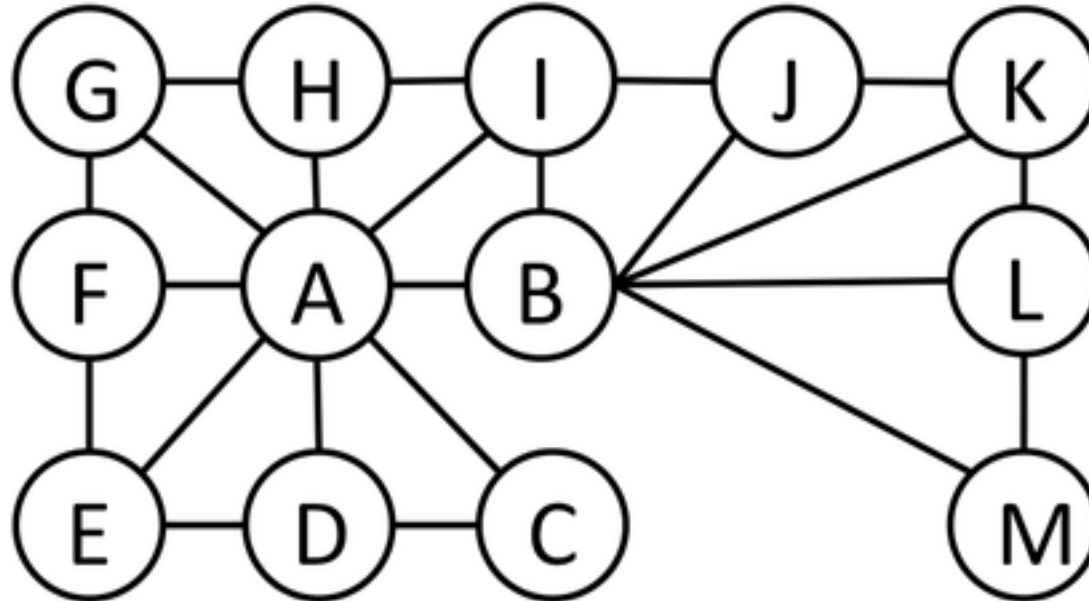
Compute residual CSP
for each assignment

Solve the residual CSPs
(tree structured)



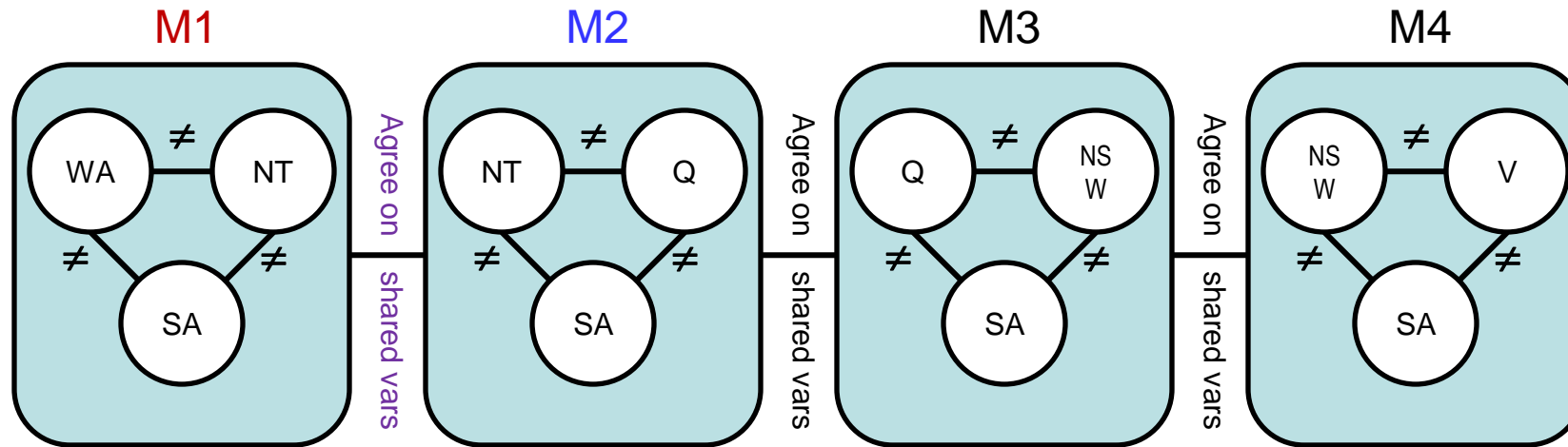
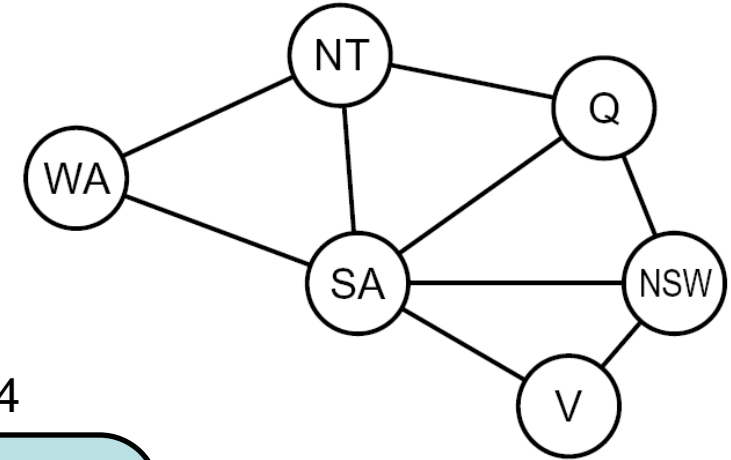
Cutset Quiz

- 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

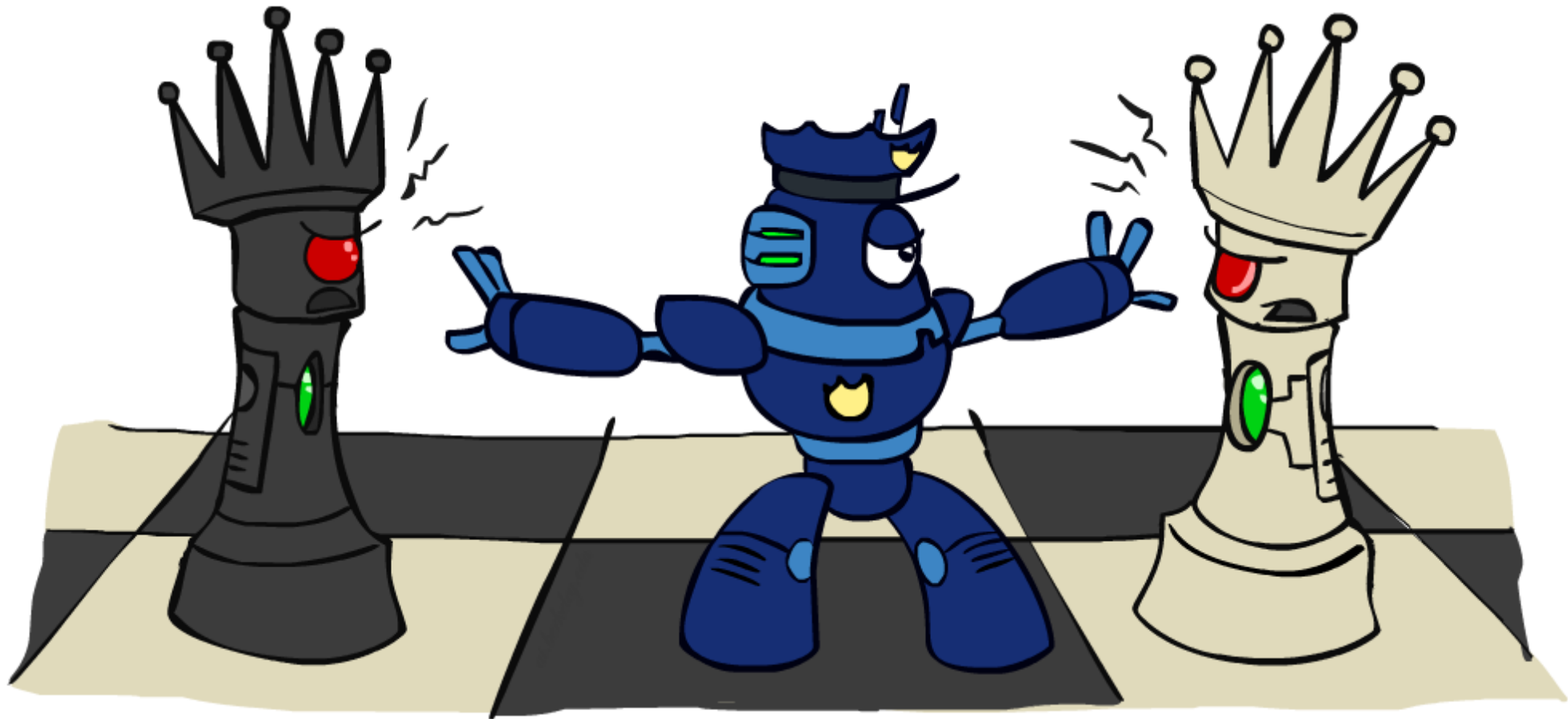


{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
...}

{(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.
- Local search algorithms turn out to be very effective in solving many CSPs. They use a complete-state formulation (as introduced in Section 4.1.1) where each state assigns a value to every variable, and the search changes the value of one variable at a time.
- To apply to CSPs:
 - Take an assignment with unsatisfied constraints
 - Operators *reassign* variable values
 - No fringe! Live on the edge.
- We then randomly choose a conflicted variable, we’d like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables—the min-conflicts heuristic.
- 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
- Min-conflicts is surprisingly effective for many CSPs, amazingly, on the n-queens problem.



8-queens problem

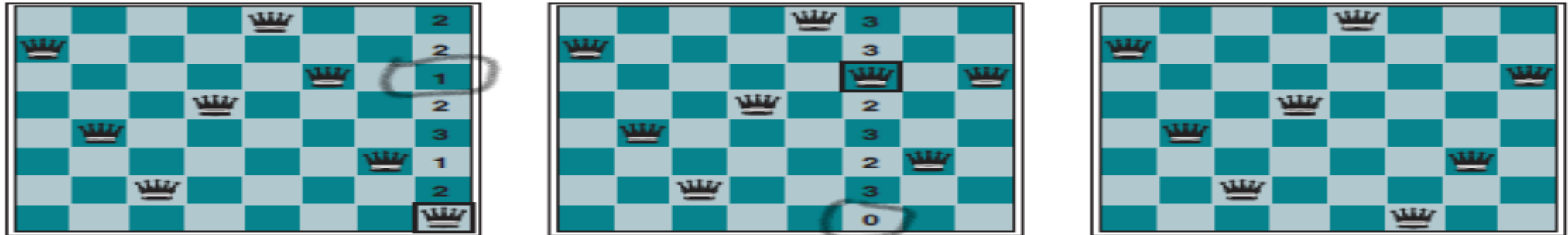
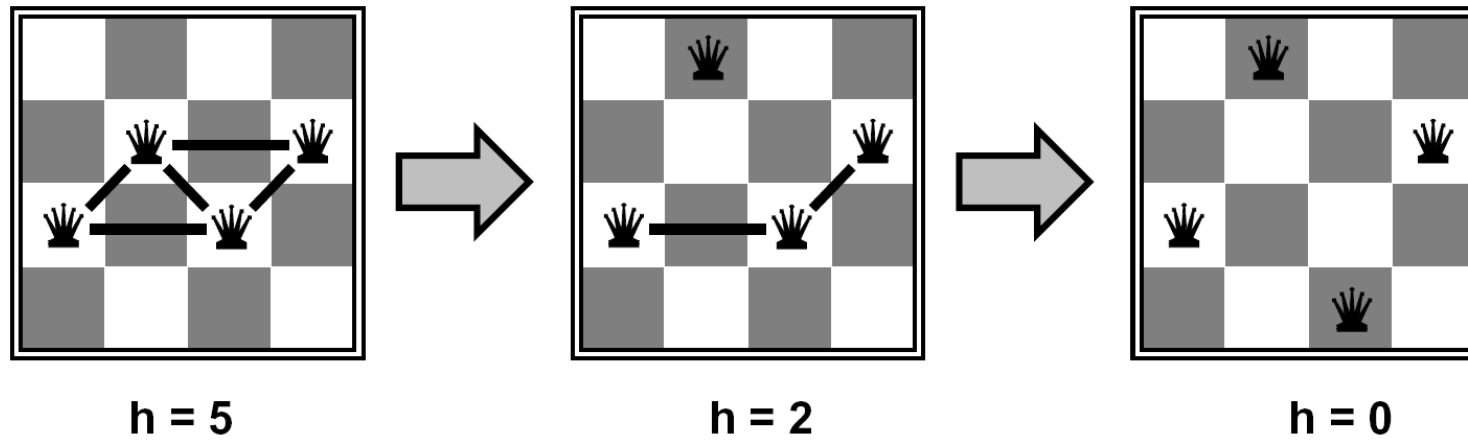


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

```
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 ← an initial complete assignment for csp  
for i = 1 to max_steps do  
    if current is a solution for csp then return current  
    var ← a randomly chosen conflicted variable from csp.VARIABLES  
    value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)  
    set var = value in current  
return failure
```

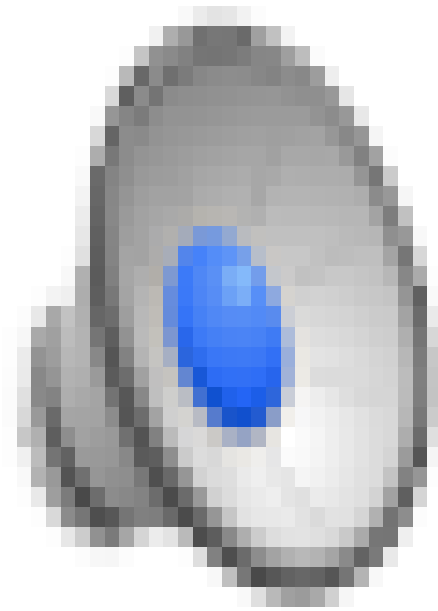
Figure 5.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Example: 4-Queens

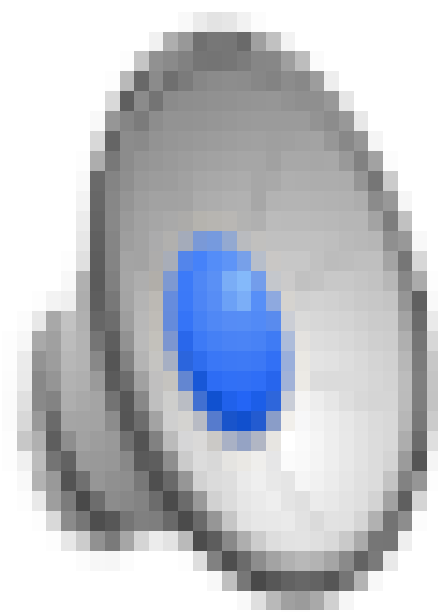


- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $c(n) =$ number of attacks

Video of Demo Iterative Improvement – n Queens



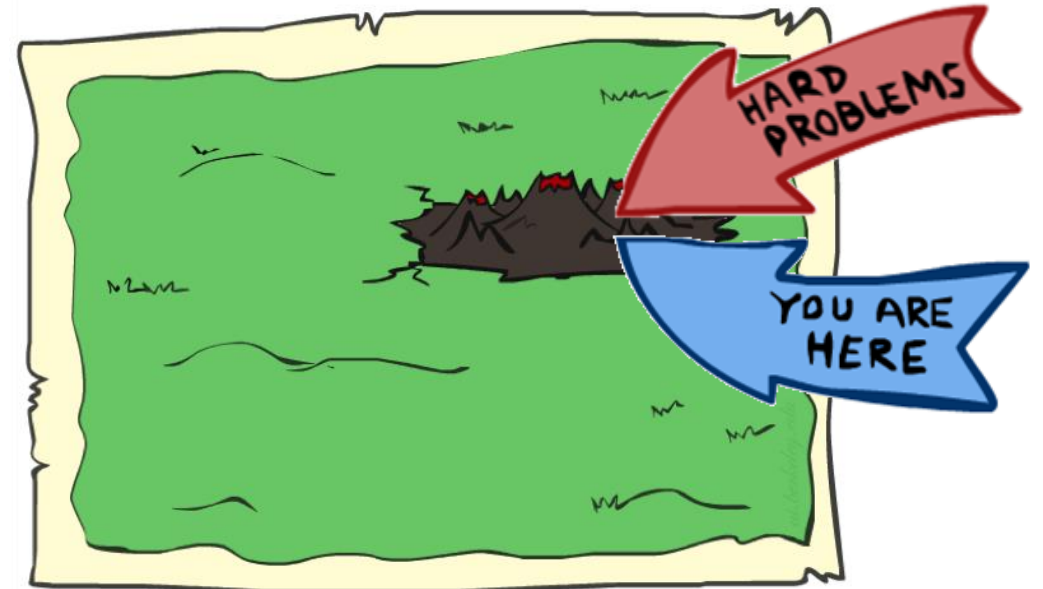
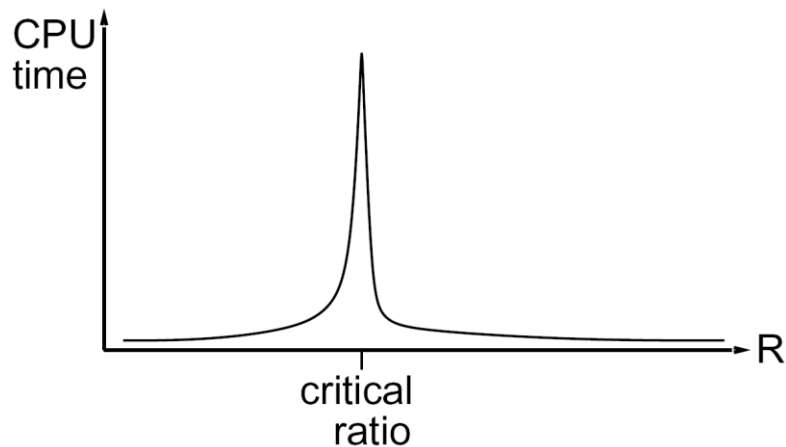
Video of Demo Iterative Improvement – Coloring



Performance of Min-Conflicts

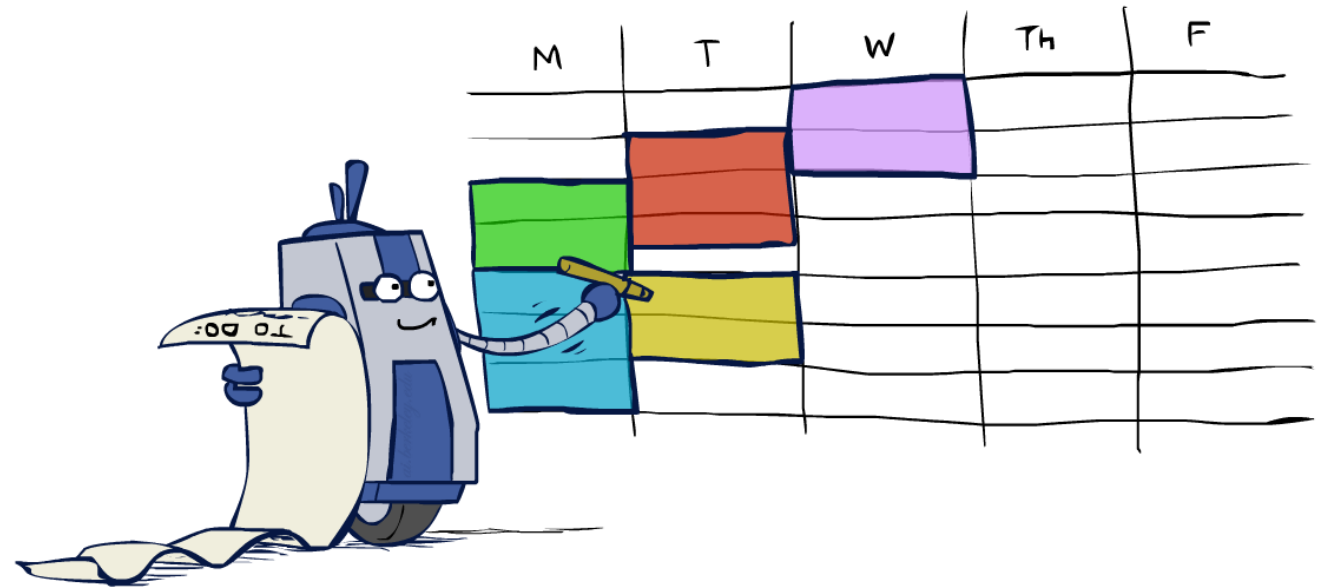
- **Min-conflicts** is surprisingly **effective for many CSPs**. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly independent of problem size.
- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!
- It solves even the million-queens problem in an average of 50 steps (after the initial assignment).
- Min-conflicts also works well for hard problems.
- For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.
- 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



CSPLib: A problem library for constraints:

<https://www.csplib.org/>