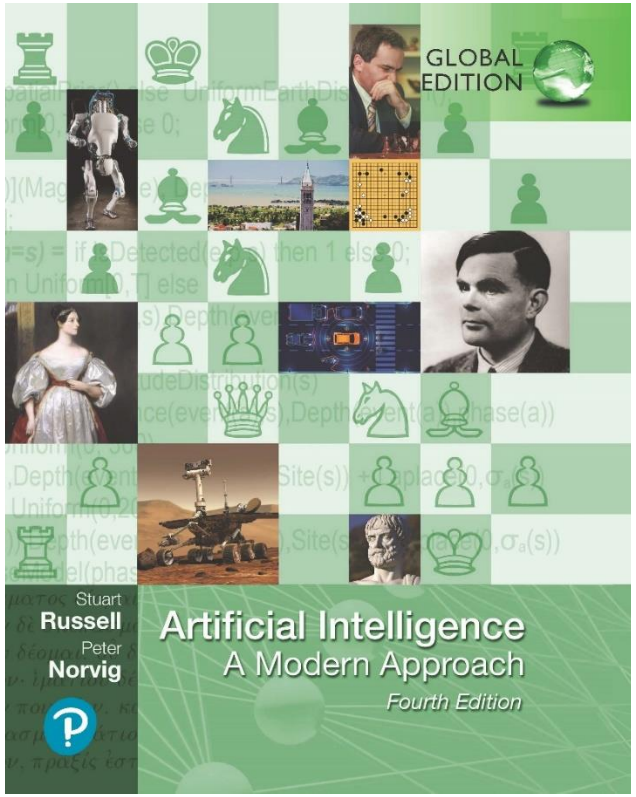
COE 4213564

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.



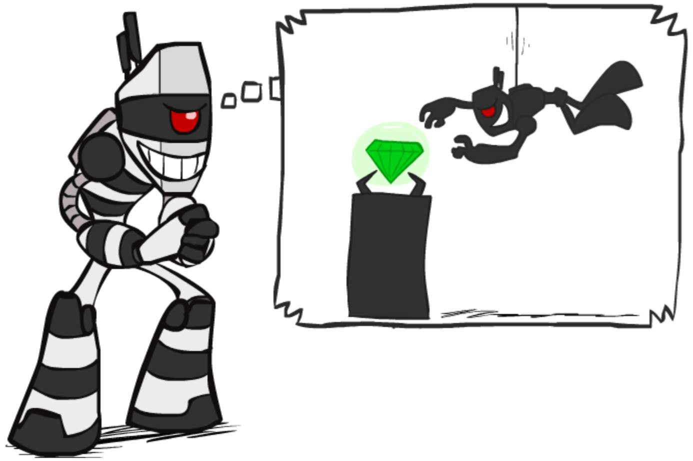
Chapter 5

Constraint Satisfaction Problems

What is Search For?

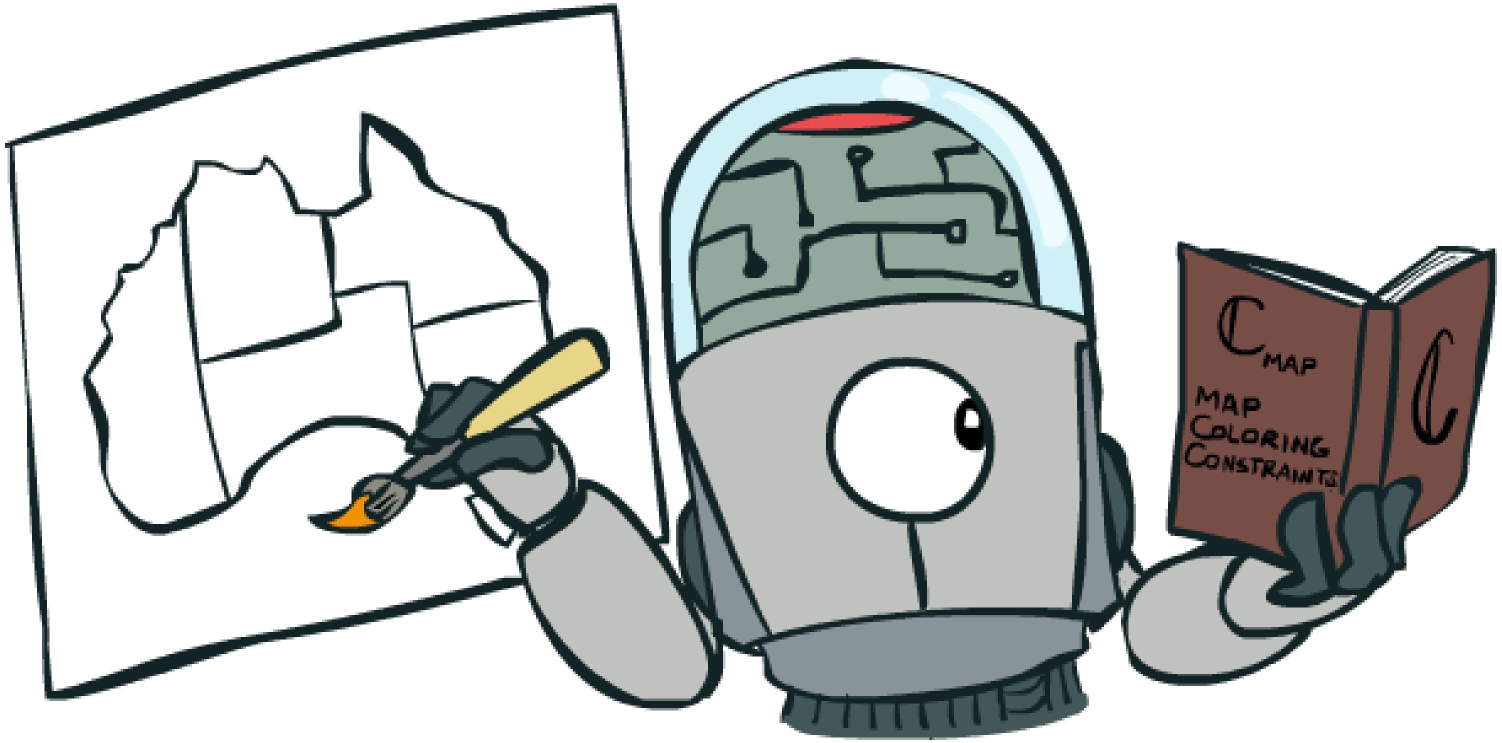
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- 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

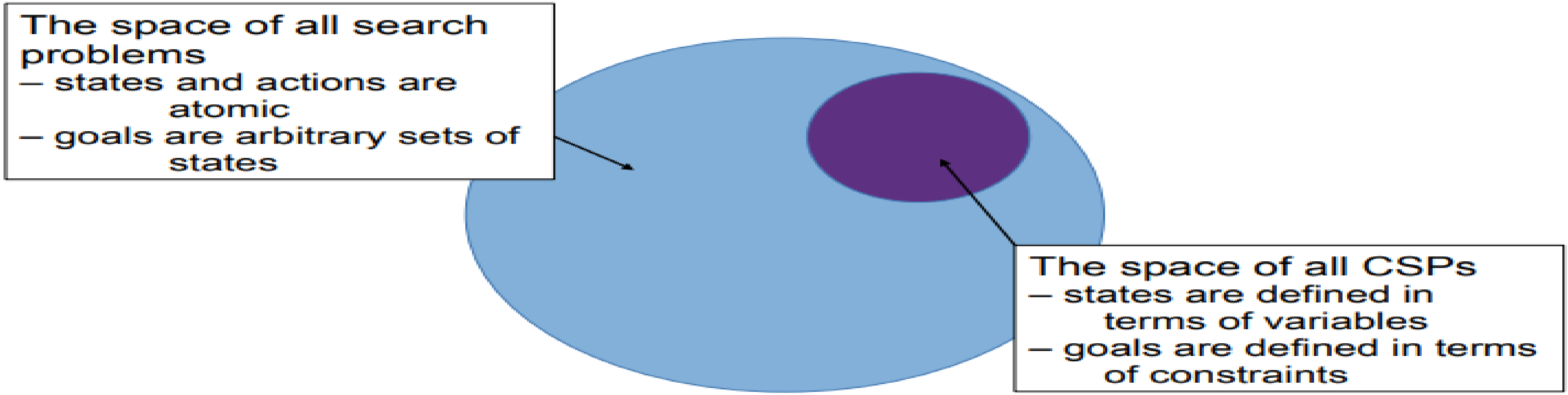
Constraint Satisfaction Problems



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.

CSPs $\subseteq$ 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

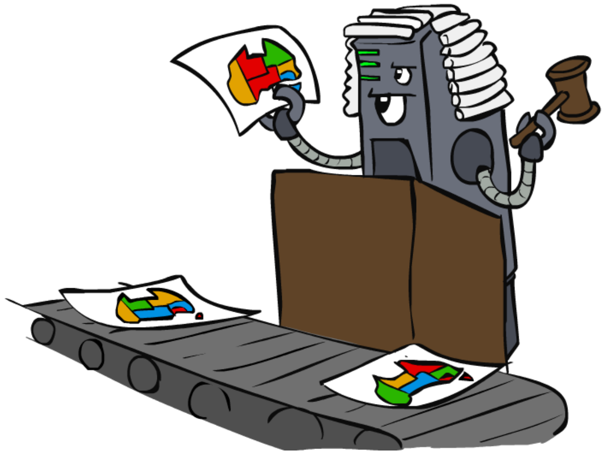
A constraint satisfaction problem (**CSP**) consists of three components, *X*, *D*, and *C:*

- *X* is a set of variables, $\{X_1 ..... X_n\}$.
- *D* is a set of domains, $\{D_1, ... , D_n\}$, 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

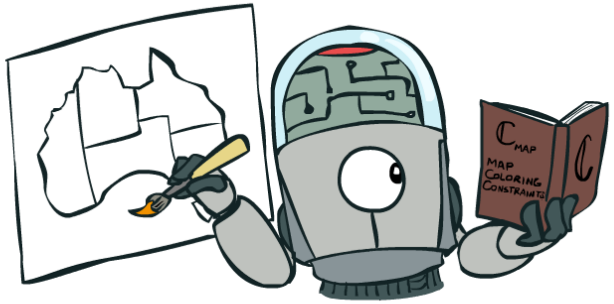
CSP:

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

Simple example:

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

Another example:

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

From: UBC CS 322 – CSP

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

- a set of variables $\mathcal{V}$
- a domain 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:

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

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

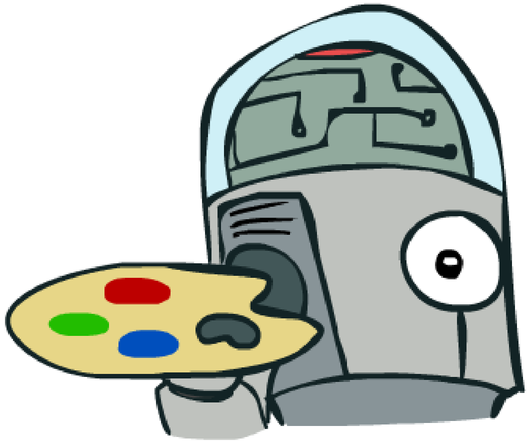
Possible worlds for this CSP:

$\{V_1=1, V_2=1\}$
$\{V_1=1, V_2=2\}$
$\{V_1=2, V_2=1\}$ (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

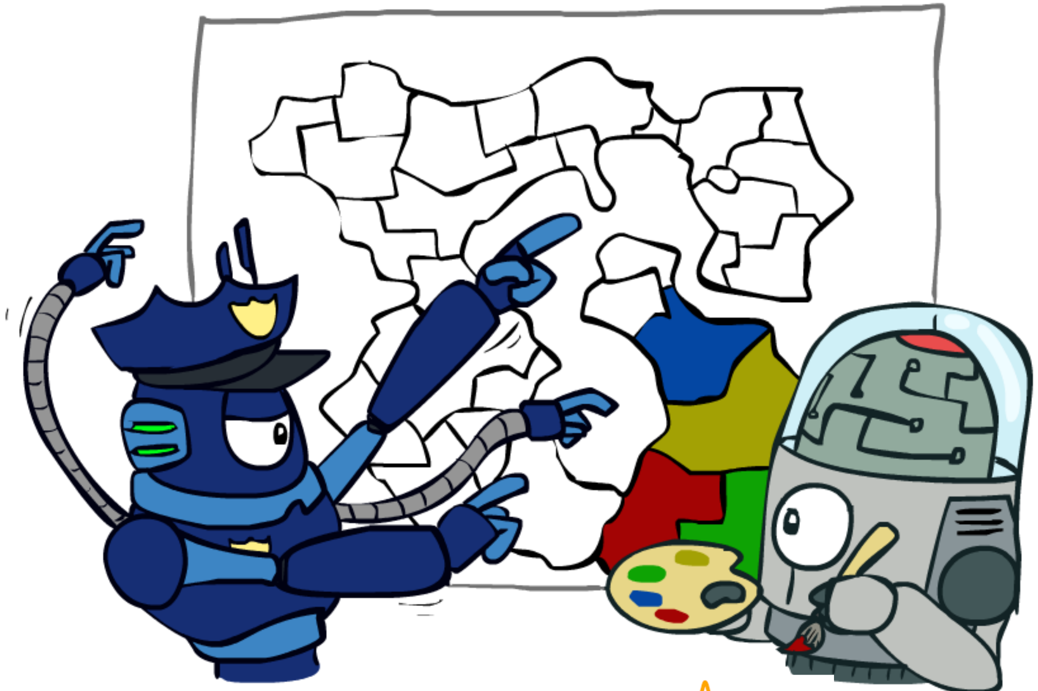
- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq green$$

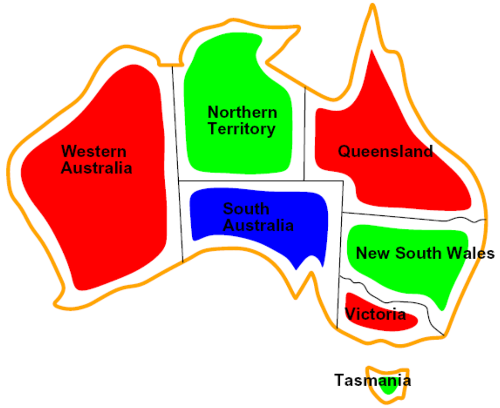
  - Binary constraints involve pairs of variables, e.g.:

$$SA \neq WA$$

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



- 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 $dom(V_1) = \{1,2,3\}$ and $dom(V_2) = \{1,2\}$:

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: $V_1 > V_2$

From: UBC CS 322 – CSP

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 $dom(V_1) = \{1,2,3\}$ and $dom(V_2) = \{1,2\}$:

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)

From: UBC CS 322 – CSP

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 ?

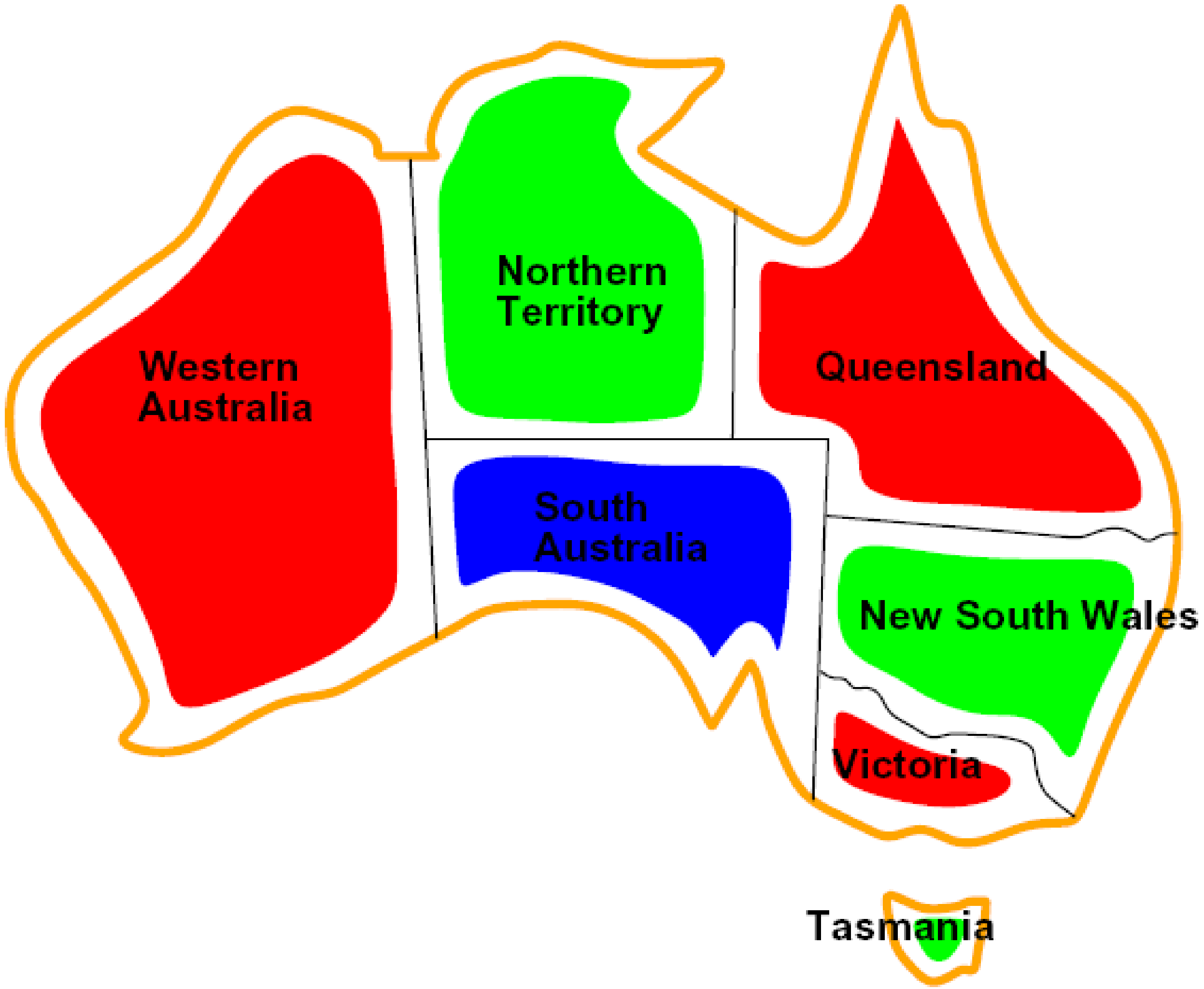
  k variables

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 problem: **Map coloring**

- We are looking at a map of Australia showing each of its states and territories
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions:
  X = {WA,NT,Q,NSW,V,SA,T}

Example: Map Coloring

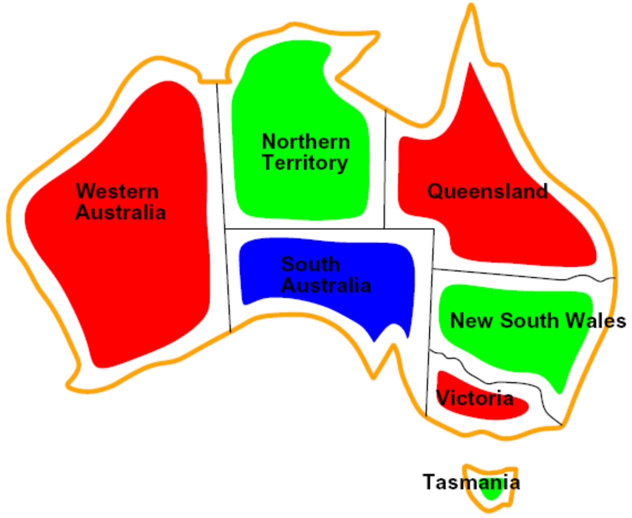
- Variables: $\text{WA, NT, Q, NSW, V, SA, T}$
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

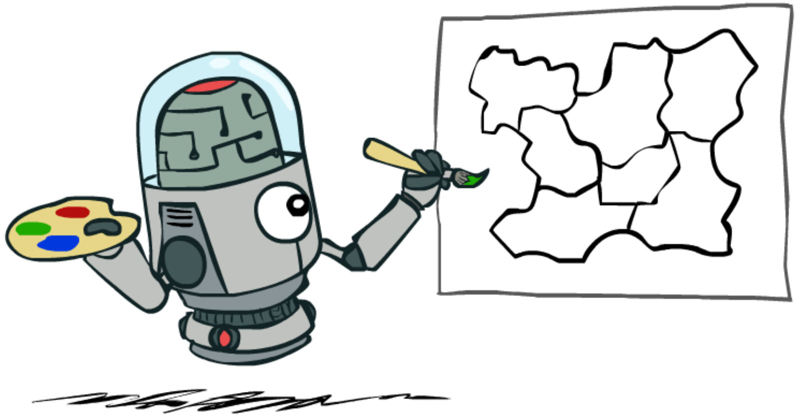
  Implicit: $\text{WA} \neq \text{NT}$

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

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

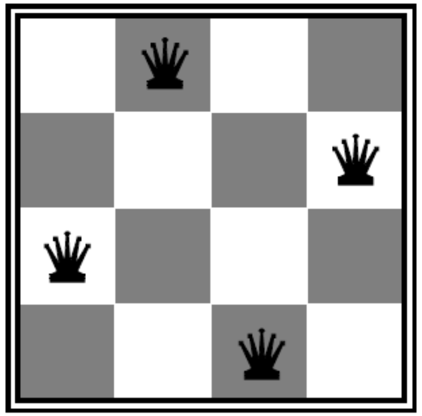
  $\{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$

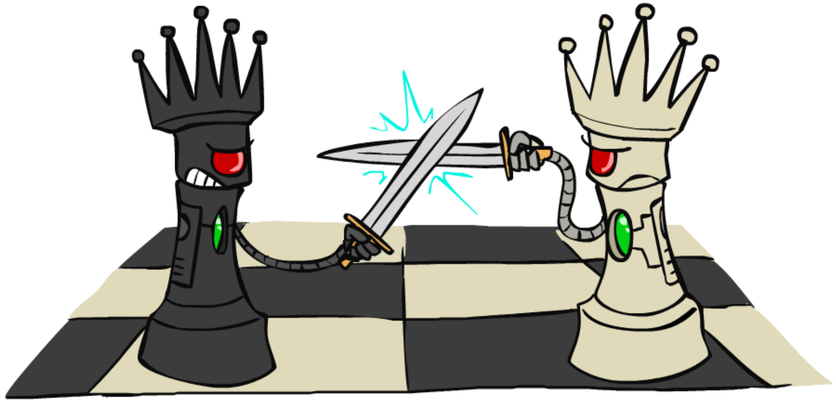




Example: N-Queens

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





Queens can not attack/threaten each other

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

Queens are not in the same row (No (1,1)),

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

Queens are not in the same columns

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

Queens are not in the same diagonals

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

Queens are not in the same oppsite diagonals

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

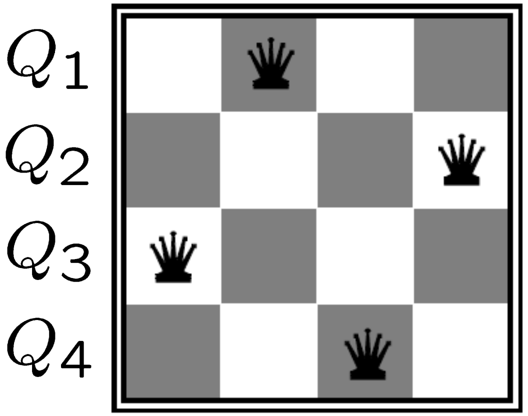
Example: N-Queens

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

Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

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

$\cdots$



Each row will one queen.
Assign a column to each queen.

Constraint Graphs

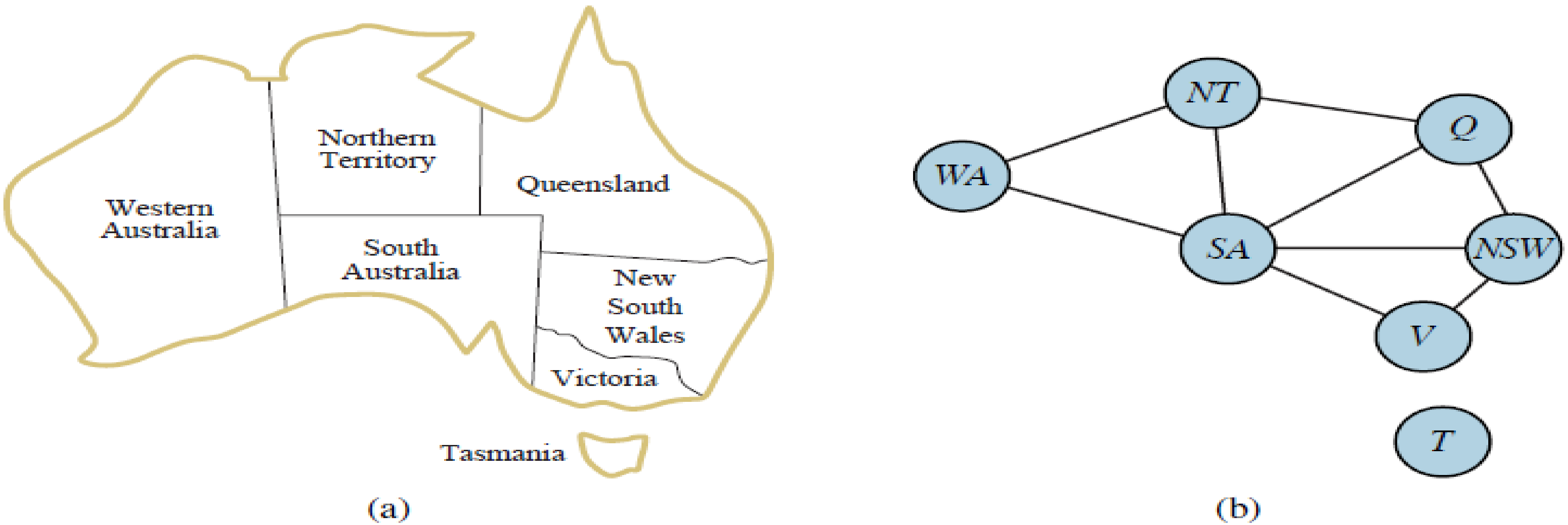
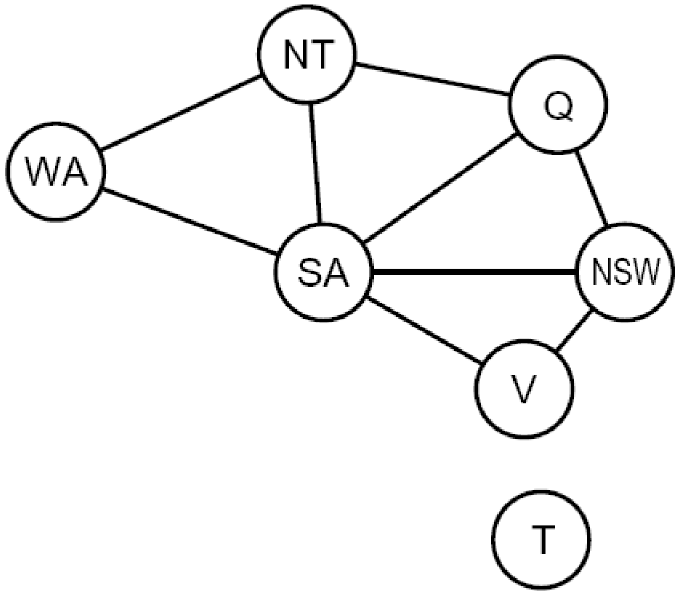


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

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!

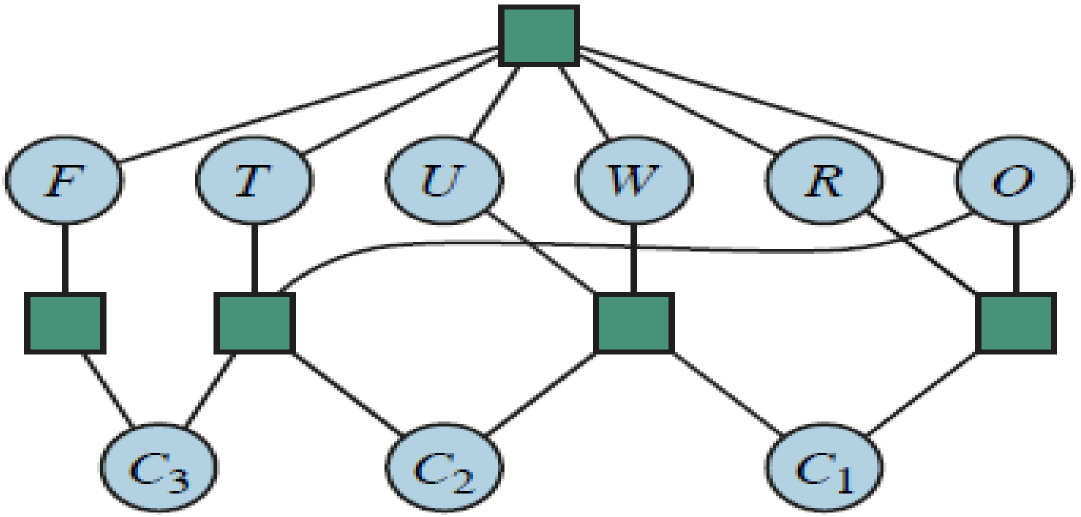


[Demo: CSP applet (made available by aispace.org) -- n-queens]

Example: cryptarithmetic puzzles

	T	*W*	*O*
+	*T*	*W*	*O*
F	*O*	*U*	*R*

(a)



(b)

Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns from right to left.

Example: Cryptarithmetic

- Variables:

  F T U W R O C_1 C_2 C_3

- Domains:

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

- Constraints:

C_3 C_2 C_1

```
    T W O
+   T W O
---------
  F O U R
```

- Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 5.2(a), this would be represented as the global constraint Alldiff (F;T;U;W;R;O).

- The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

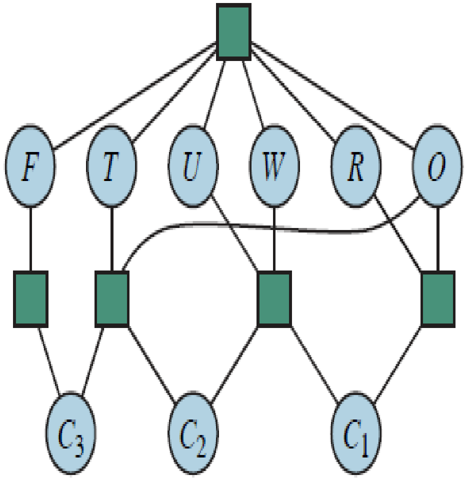
$O+O = R+10\,C_1$

$C_1+W+W = U+10\,C_2$

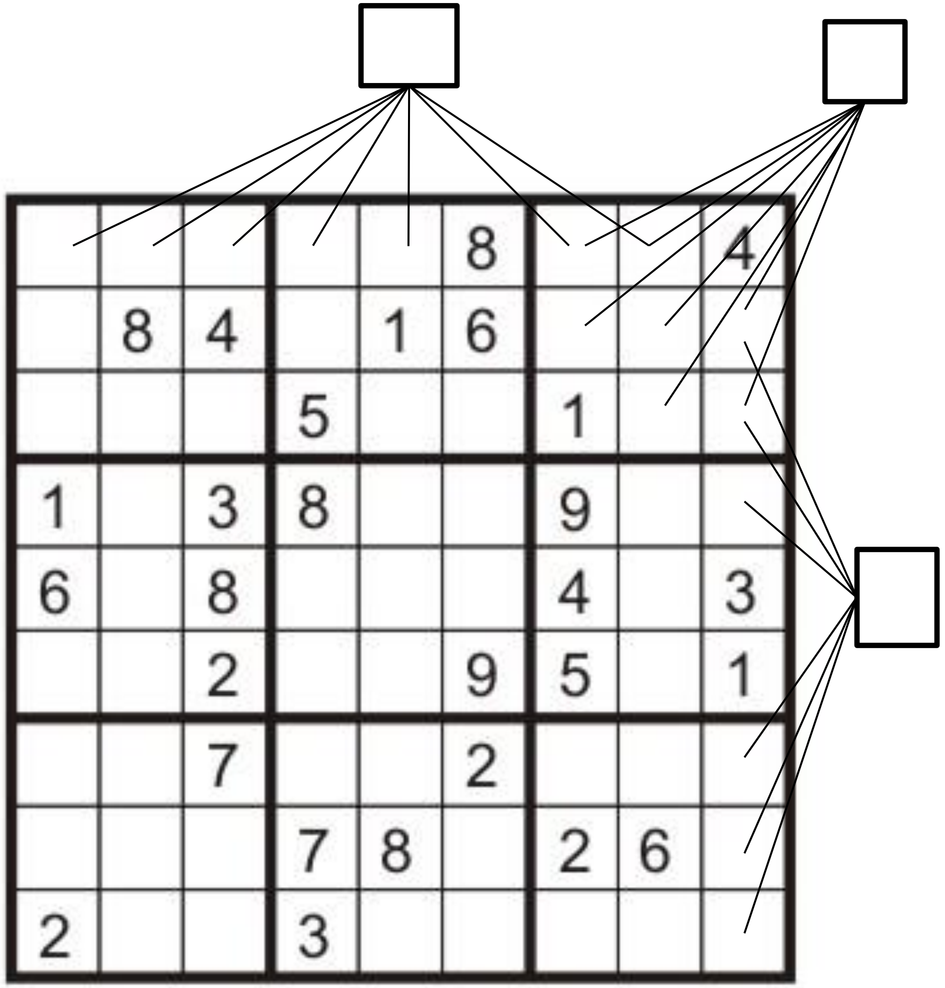
$C_2+T+T = O+10\,C_3$

$C_3 = F$;

where C_1, C_2, and C_3 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column.



Example: Sudoku



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

  9-way alldiff for each column

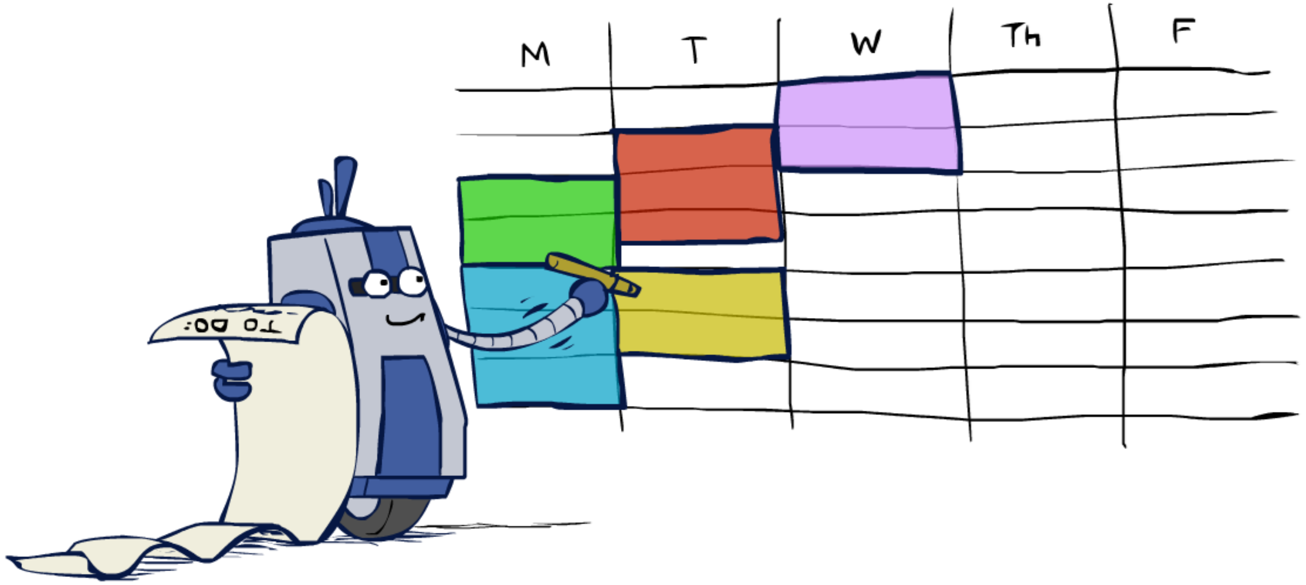
  9-way alldiff for each row

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

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 $dom(V_1) \times dom(V_2) \times \cdots \times dom(V_n)$

- 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

$O(ckd)$ $O(ck^d)$ $O(cd^k)$ $O(d^{ck})$

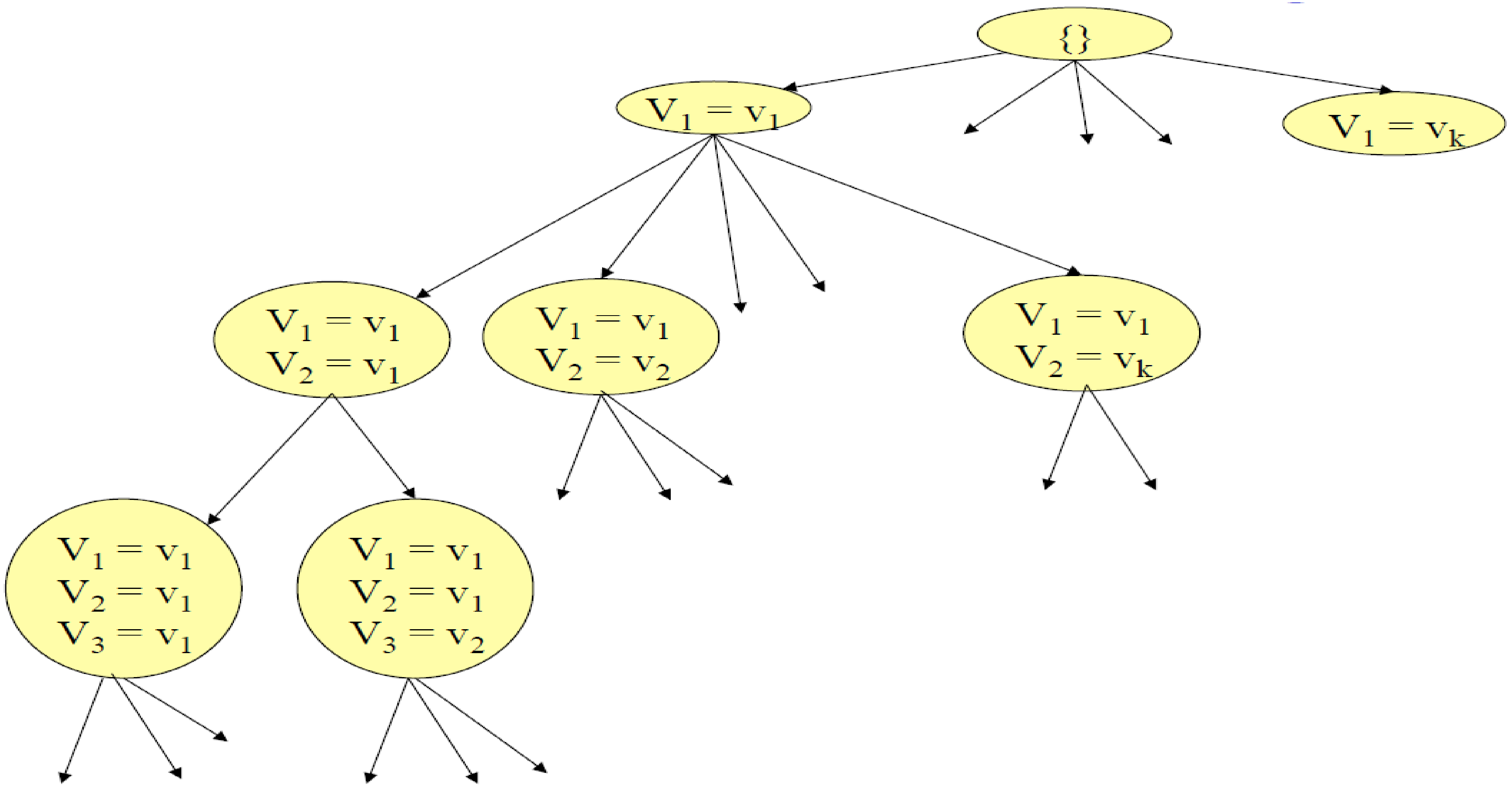
  - There are d^k possible worlds
  - For each one need to check c constraints

From: UBC CS 322 – CSP

CSP as a Search Problem: one formulation

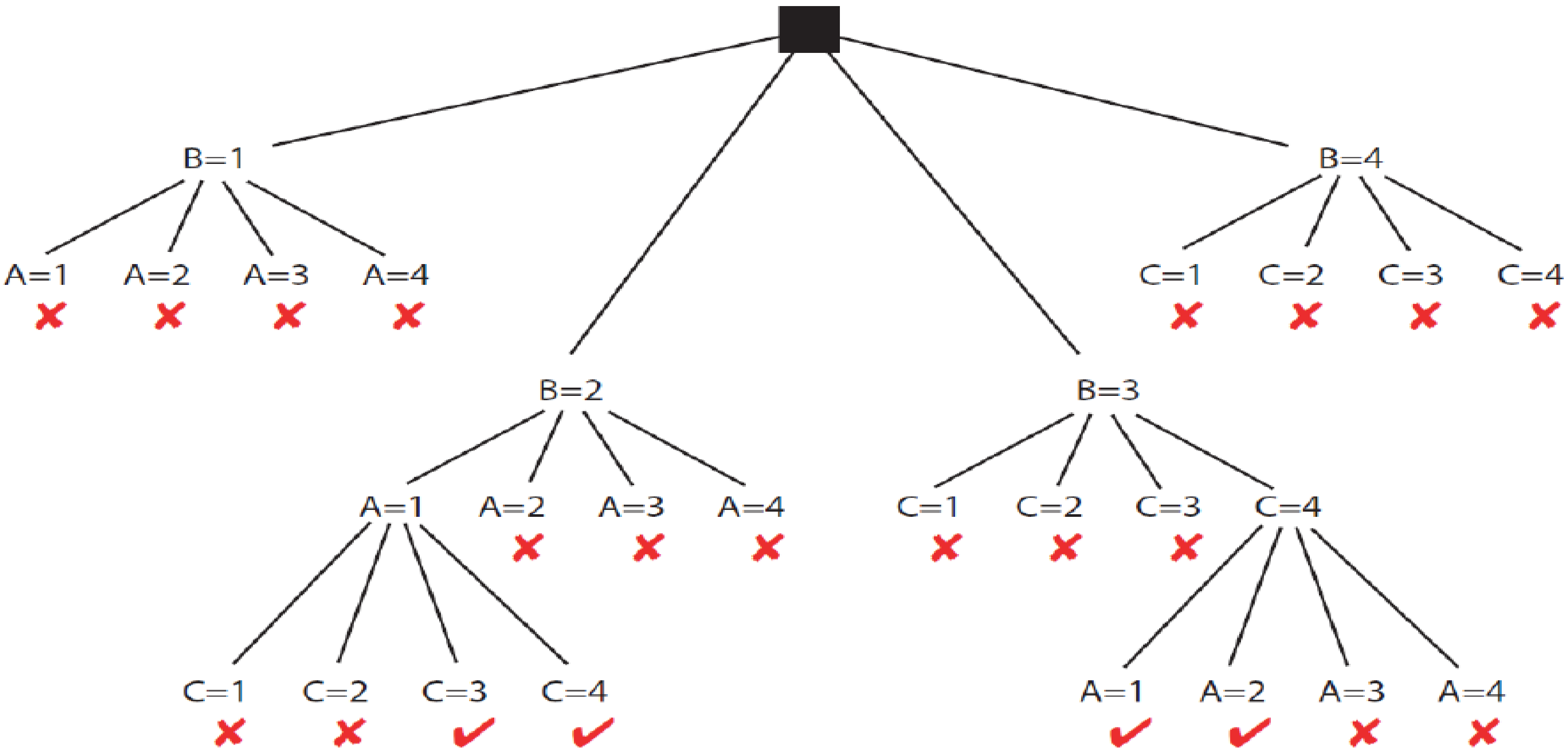
- States: partial assignment of values to variables
- Start state: empty assignment
- Successor function: states with the next variable assigned
  - E.g., follow a total order of the variables $V_1, \ldots, V_n$
  - A state assigns values to the first k variables:
    - $\{V_1 = v_1,\ldots,V_k = v_k\}$
    - Neighbors of node $\{V_1 = v_1,\ldots,V_k = v_k\}$:
      nodes $\{V_1 = v_1,\ldots,V_k = v_k, V_{k+1} = x\}$ for each $x \in dom(V_{k+1})$
- 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

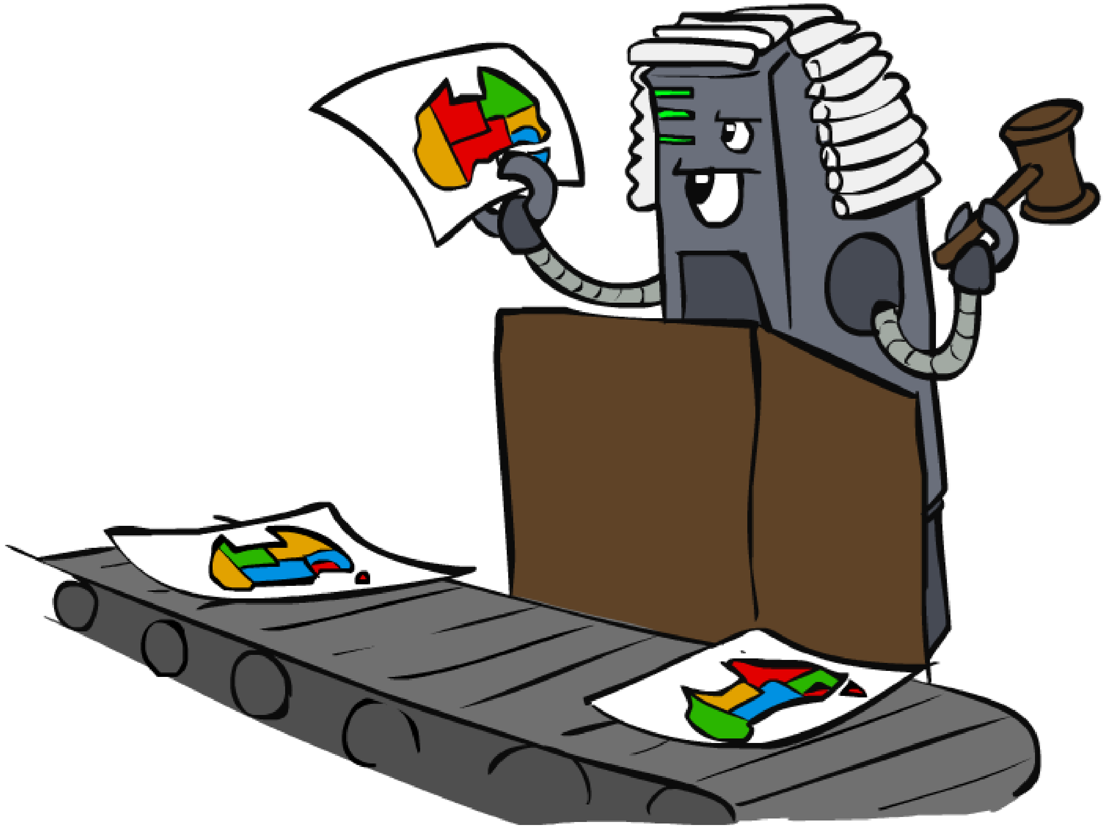


Standard Search Formulation

- Standard search formulation of CSPs

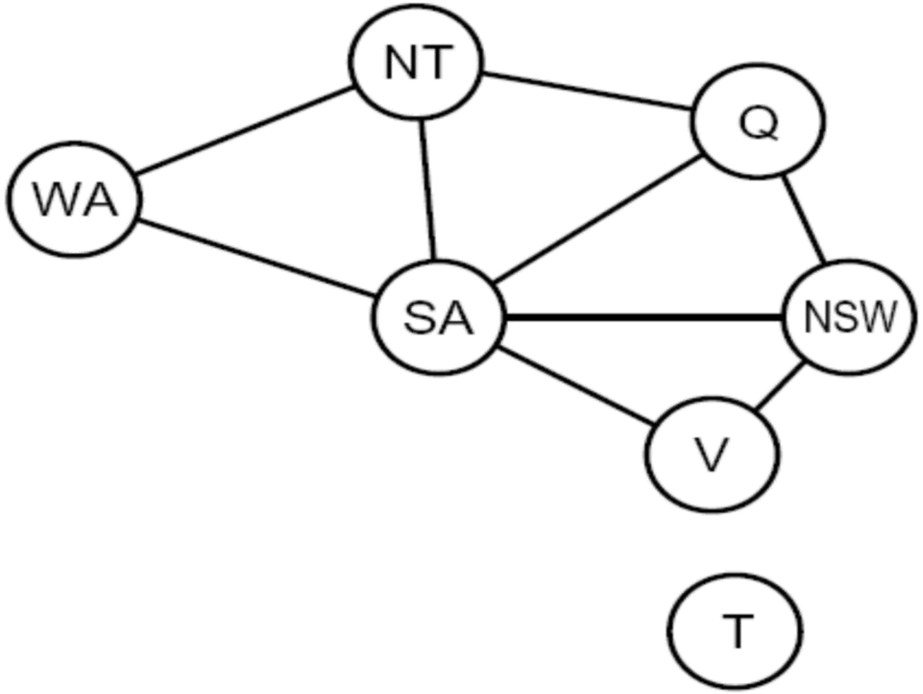
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

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



Search Methods

- What would BFS do?



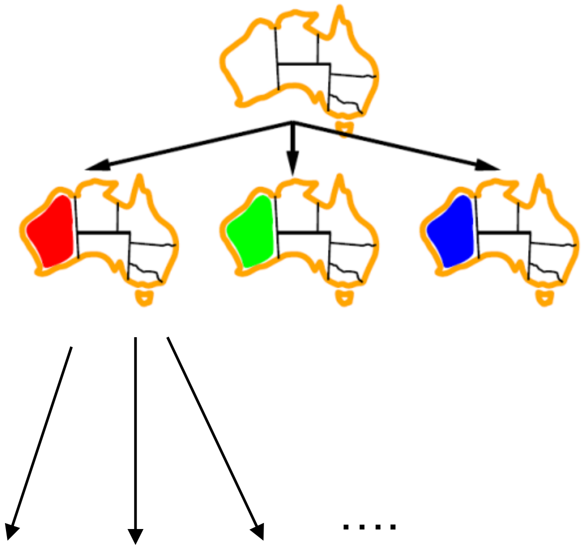
- What would DFS do?

- 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

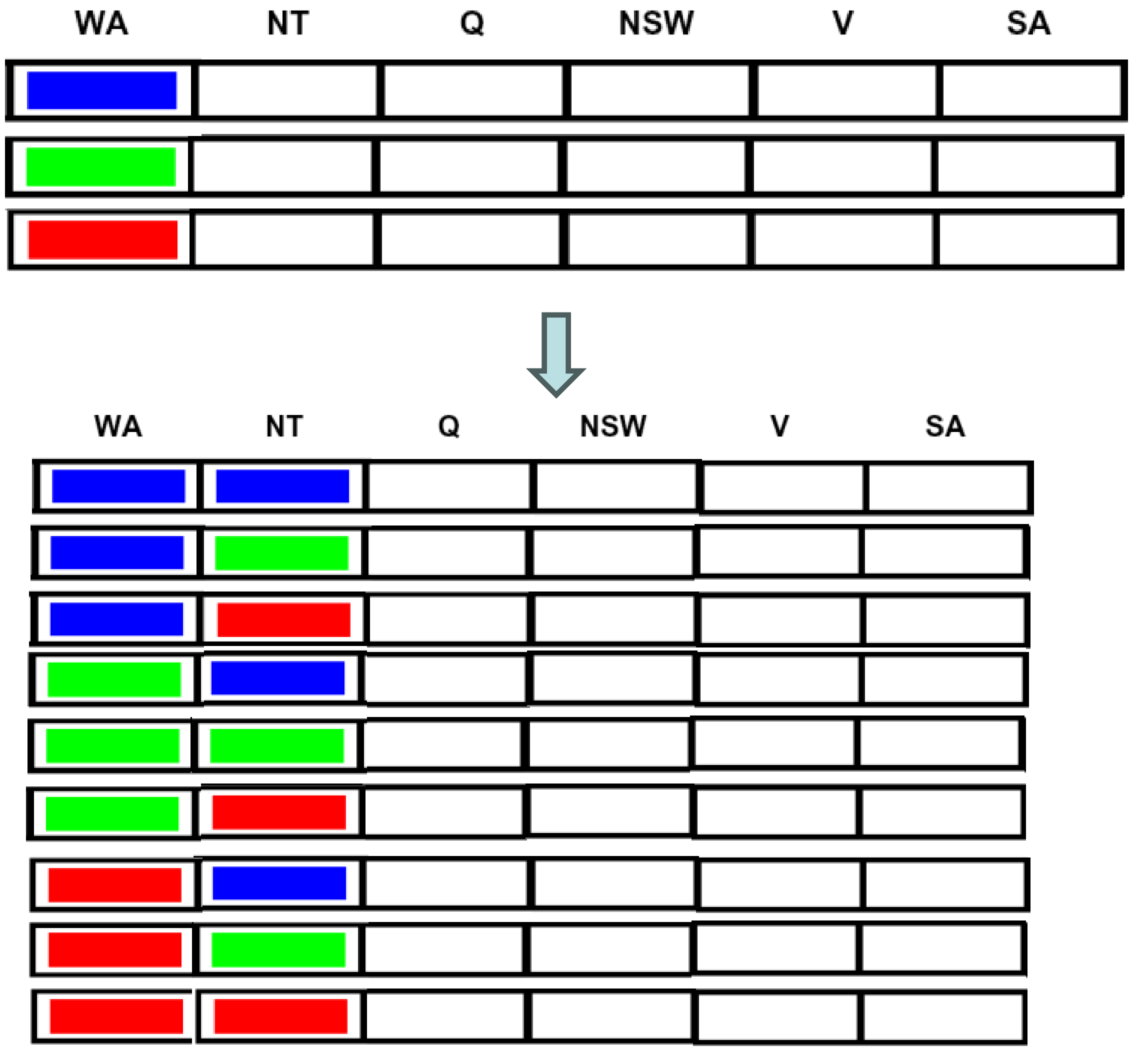
[Demo: coloring -- dfs]

BFS

Breadth First Search



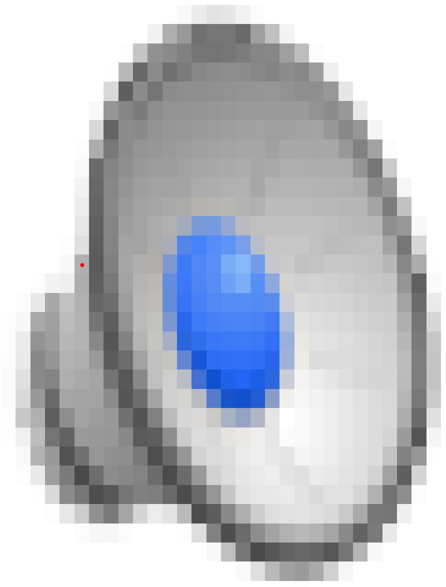
... All possible first variables
Check: Is there a solution?



BFS will take a long time to find a solution.
DFS (Naive search) seems to be better choice

Video of Demo Coloring – DFS

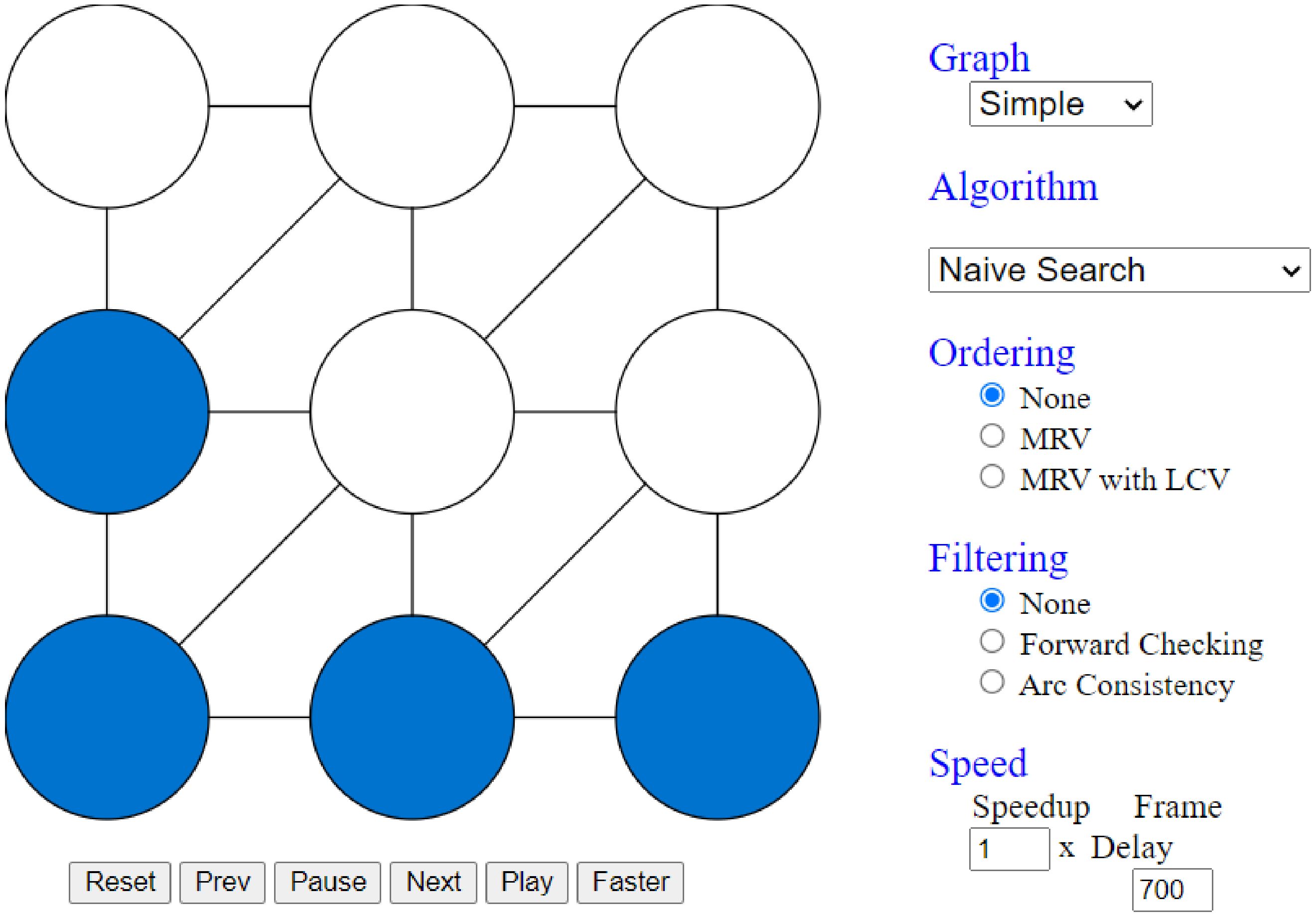
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html



- A map coloring problem is a type of CSP where each state can be assigned a color from the set (red,green,blue)

- The constraint involved says that no two neighbouring state is allowed to have the same color.

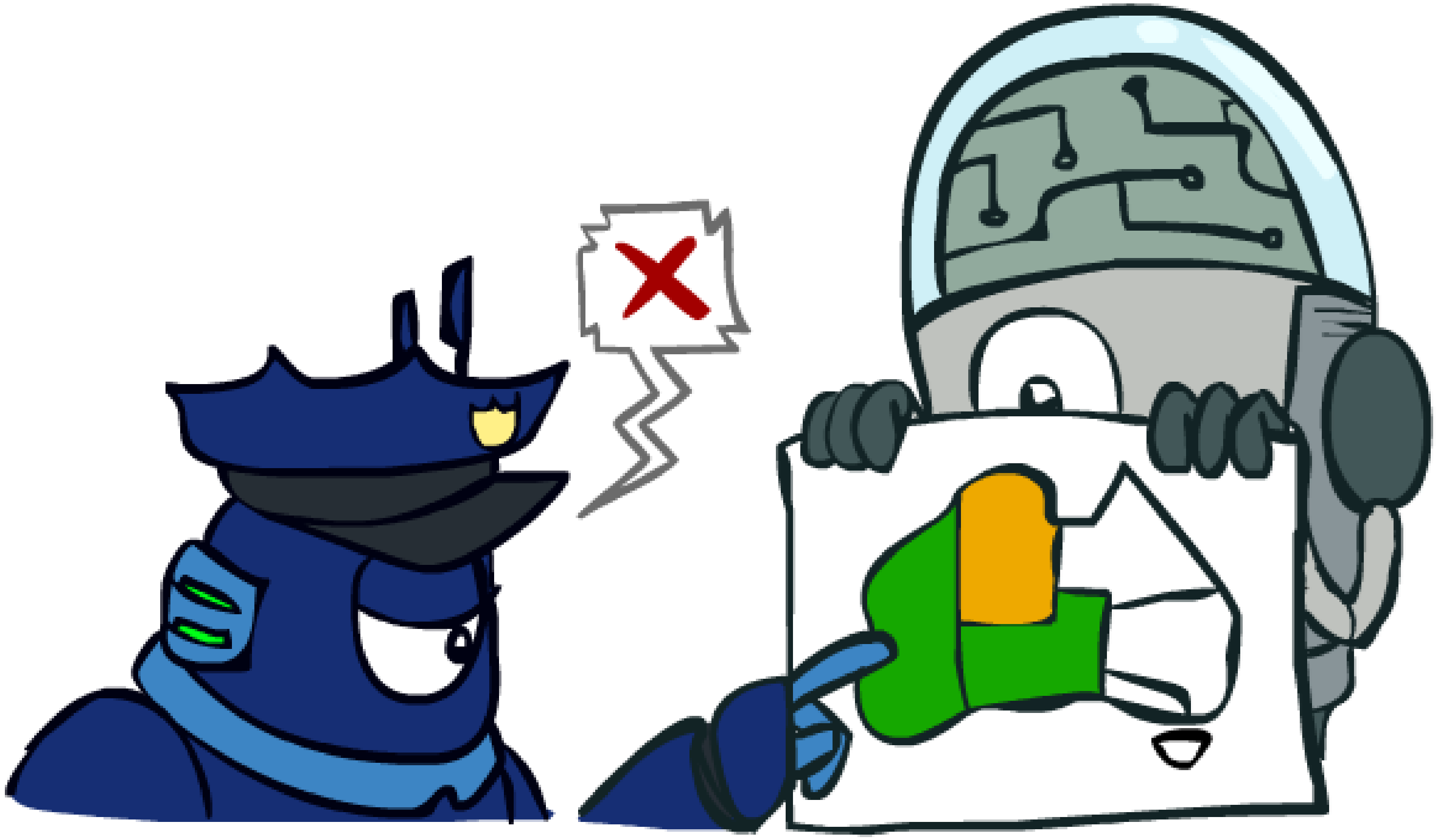
Demo Coloring – DFS



- A map coloring problem is a type of CSP where each state can be assigned a color from the set (red,green,blue)
- The constraint involved says that no two neighbouring state is allowed to have the same color.

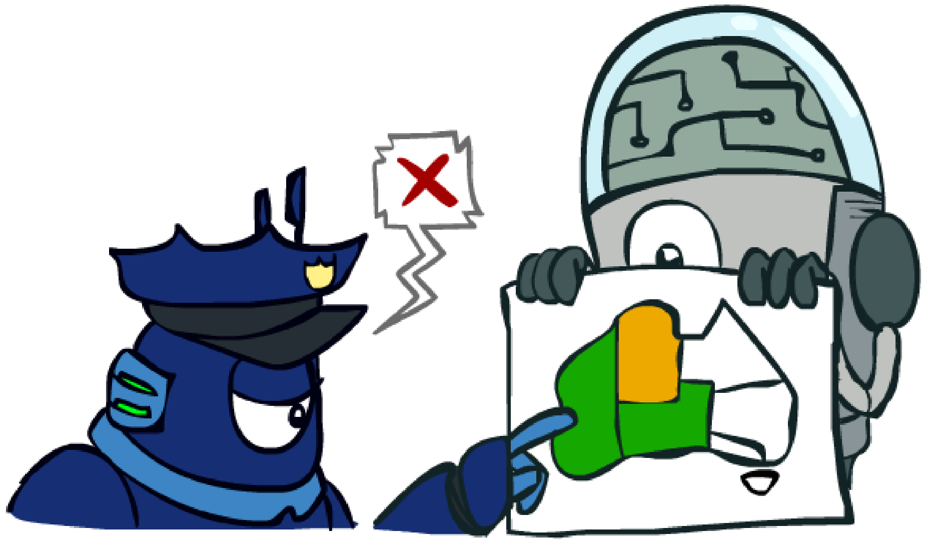
https://inst.eecs.berkeley.edu/~cs188/fa21/assets/demos/csp/csp_demos.html

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 $n \approx 25$

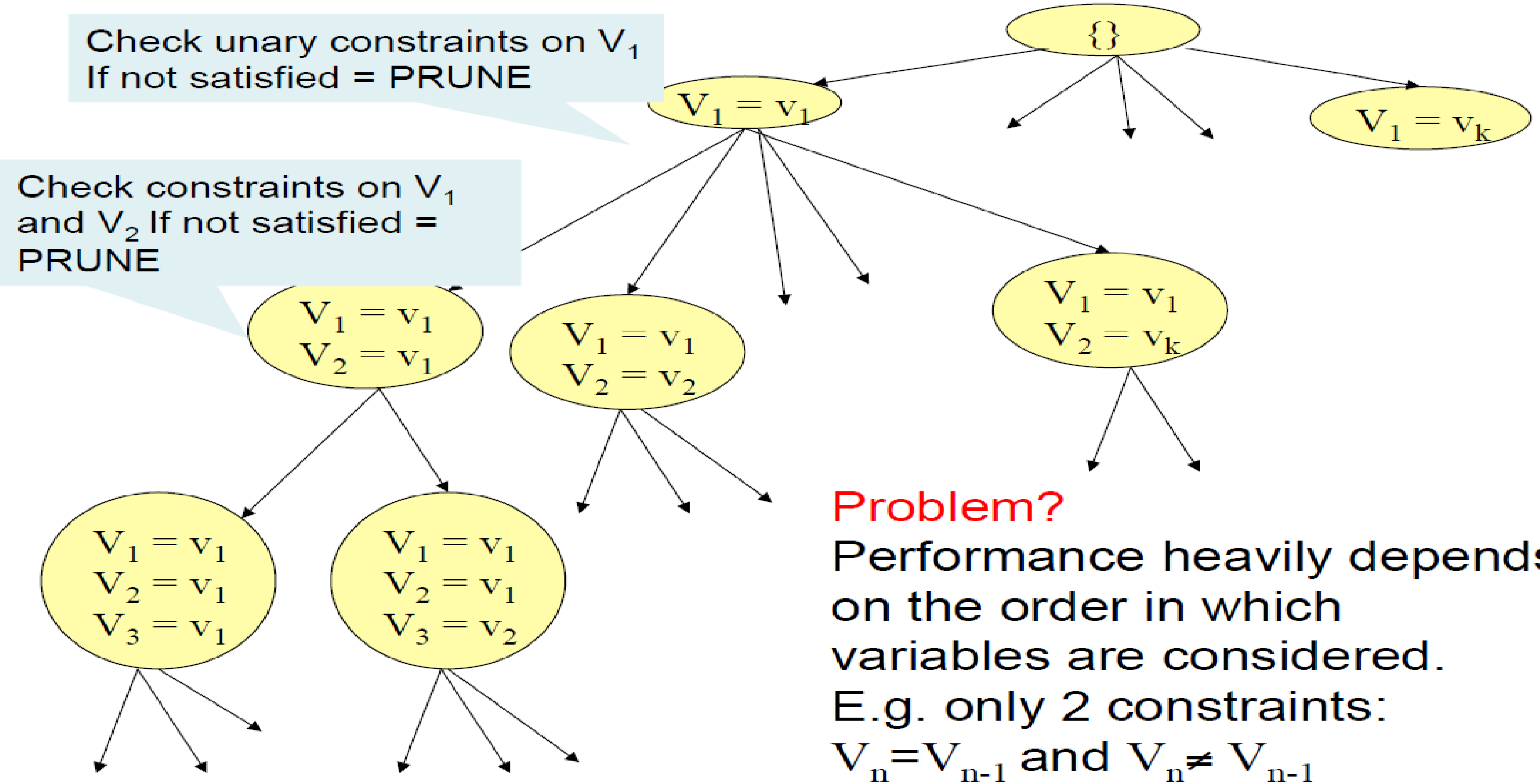


Backtracking Search

- 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

    $\Rightarrow$ Fail! Prune!

CSP as Graph Searching



Problem?

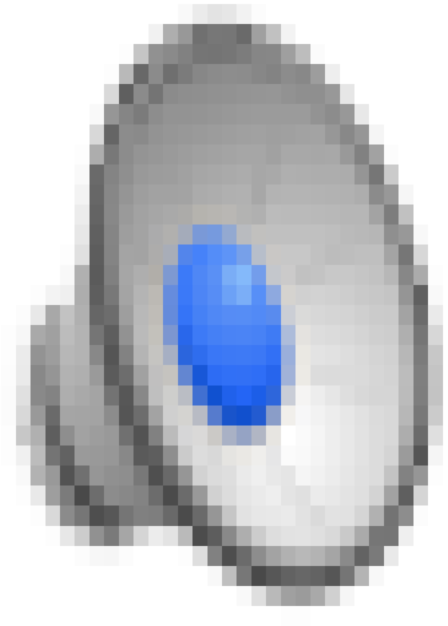
Performance heavily depends on the order in which variables are considered.

E.g. only 2 constraints:

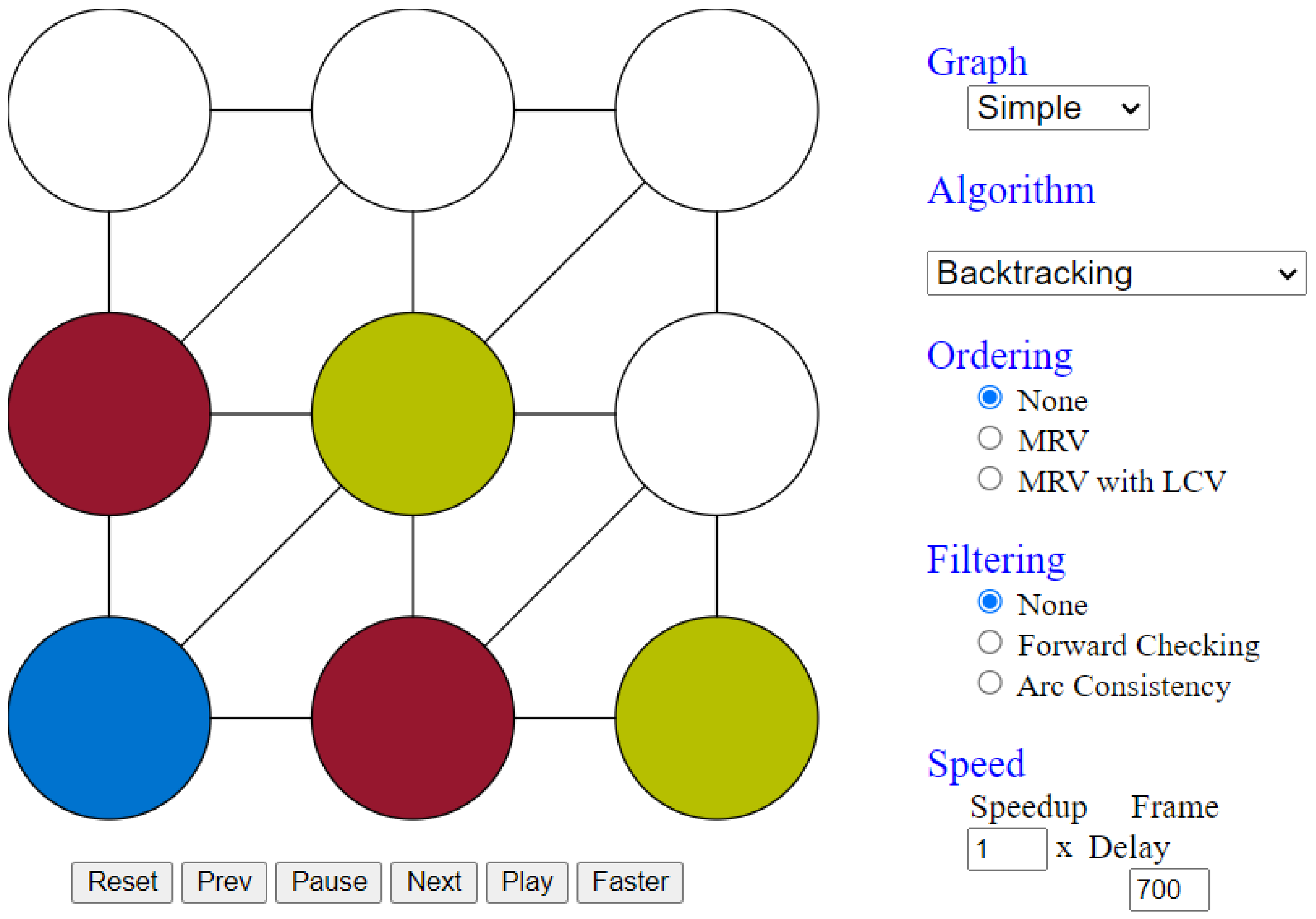
$V_n = V_{n-1}$ and $V_n \neq V_{n-1}$

Video of Demo Coloring – Backtracking

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

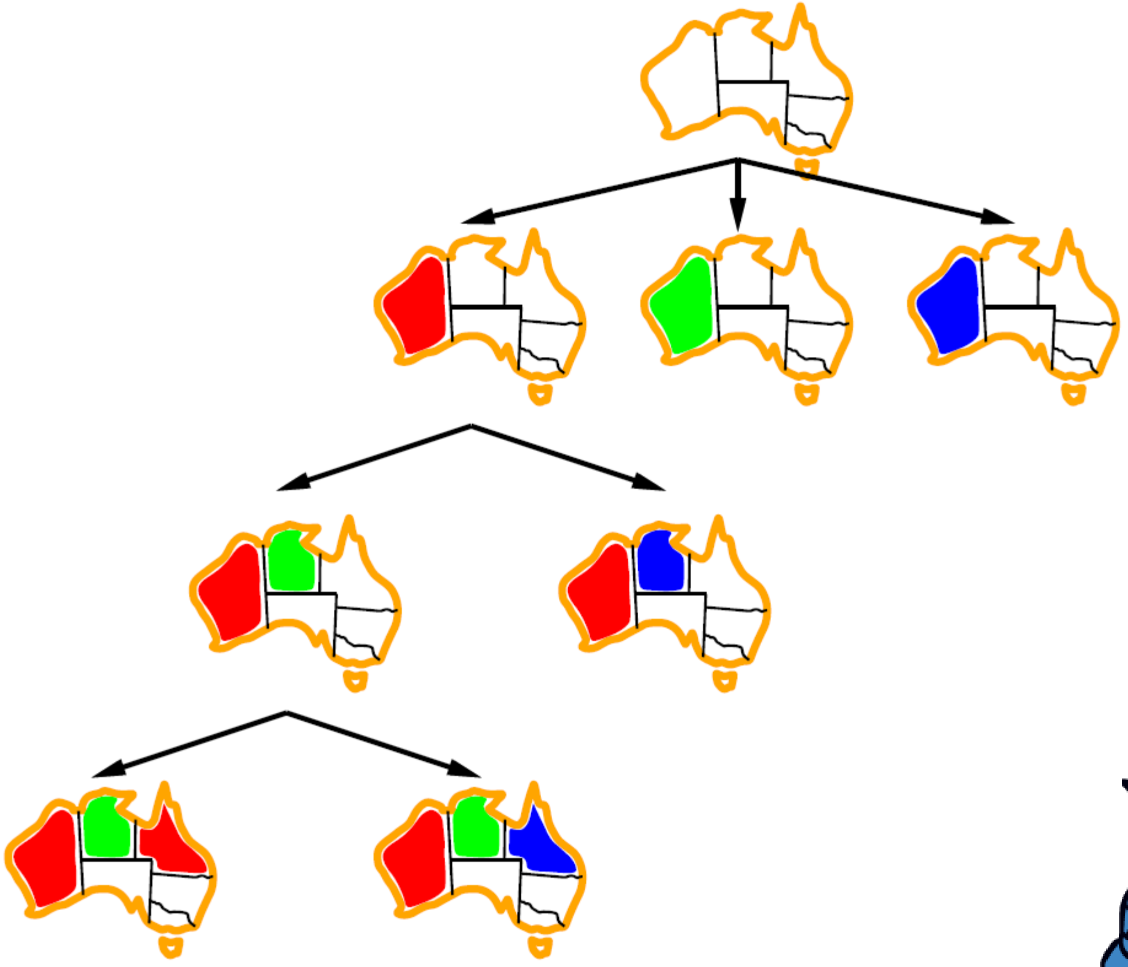


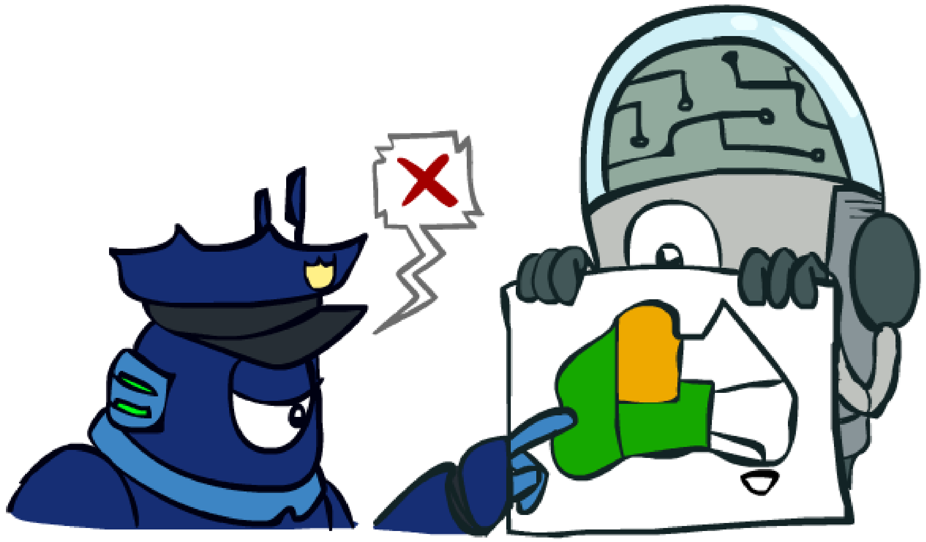
Demo Coloring – Backtracking



https://inst.eecs.berkeley.edu/~cs188/fa21/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.

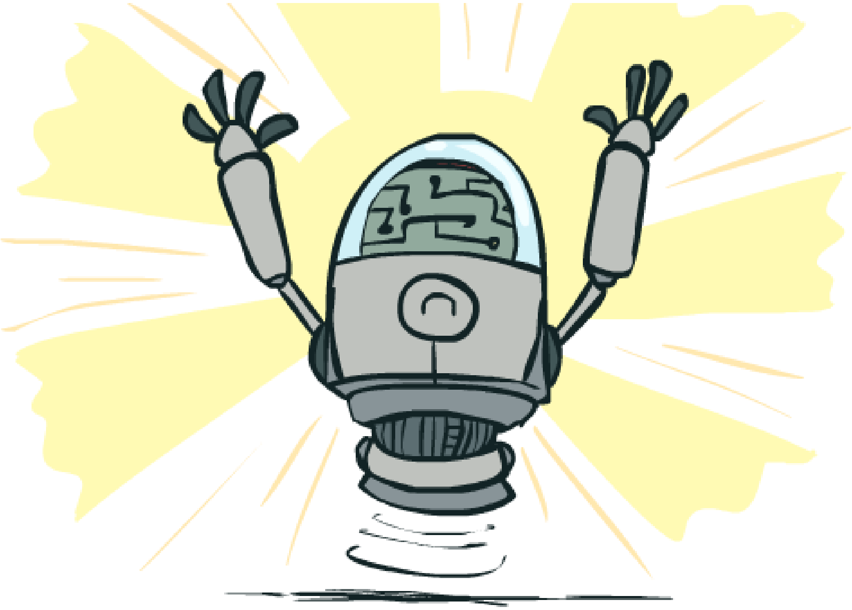
Improving Backtracking

- 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?

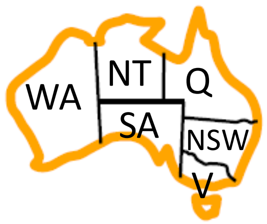
- Structure: Can we exploit the problem structure?

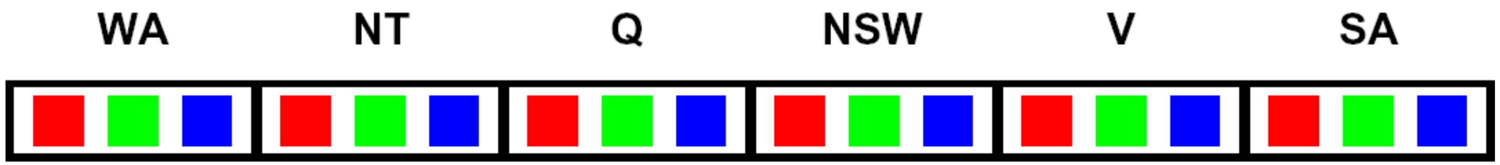


Filtering

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

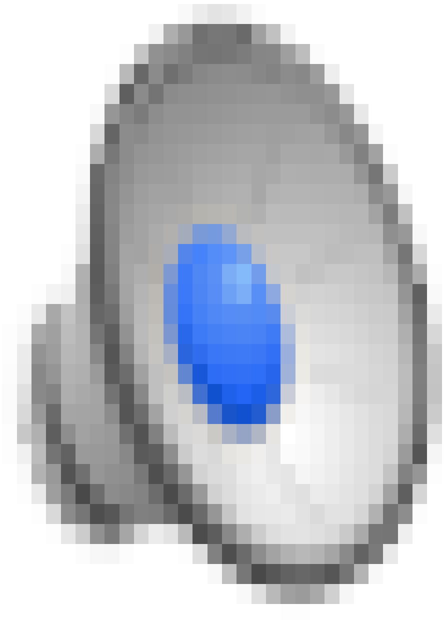




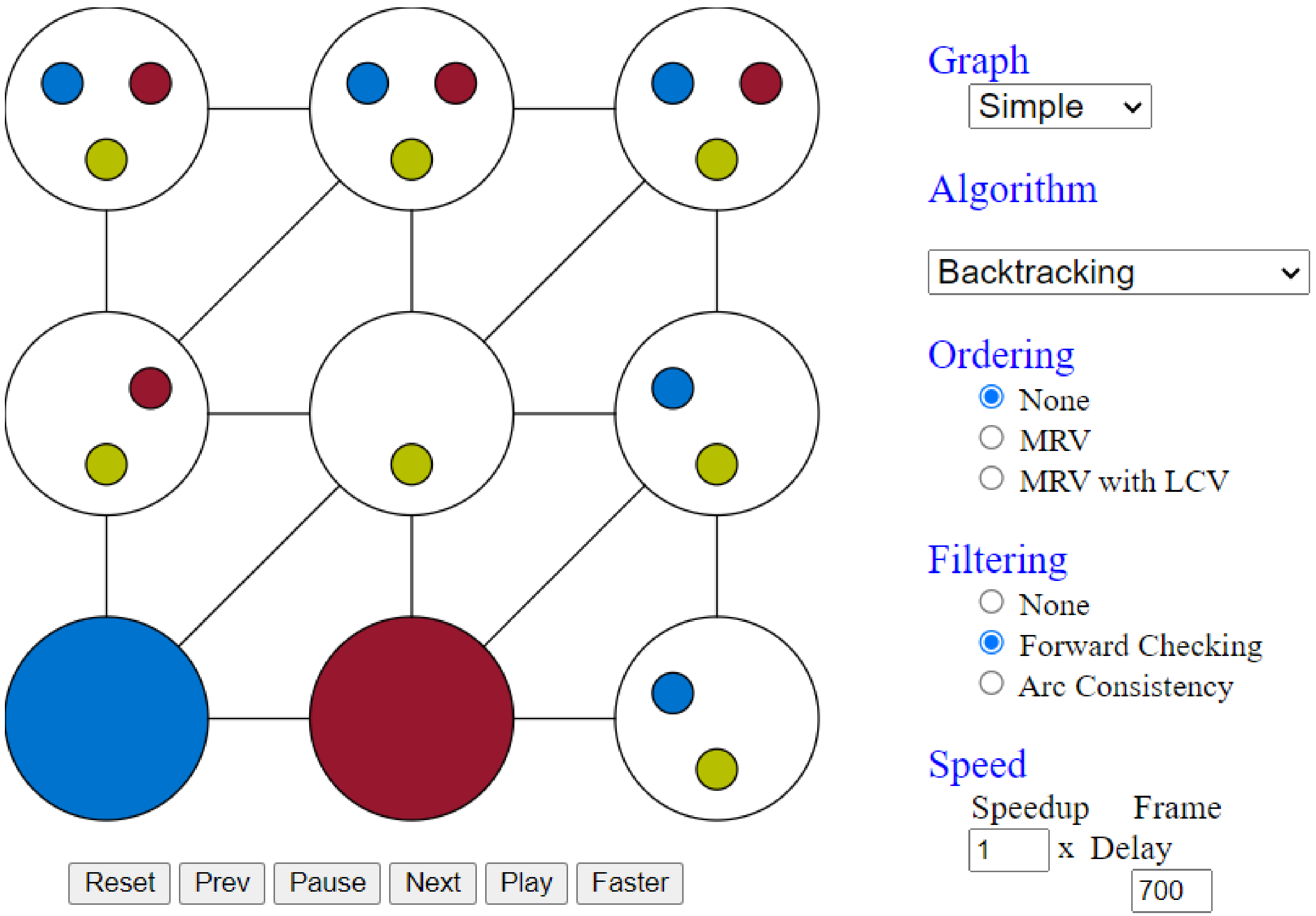
[Demo: coloring -- forward checking]

Video of Demo Coloring – Backtracking with Forward Checking

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



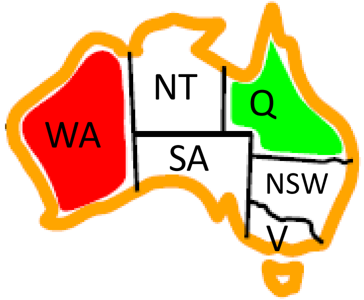
Backtracking with Forward Checking

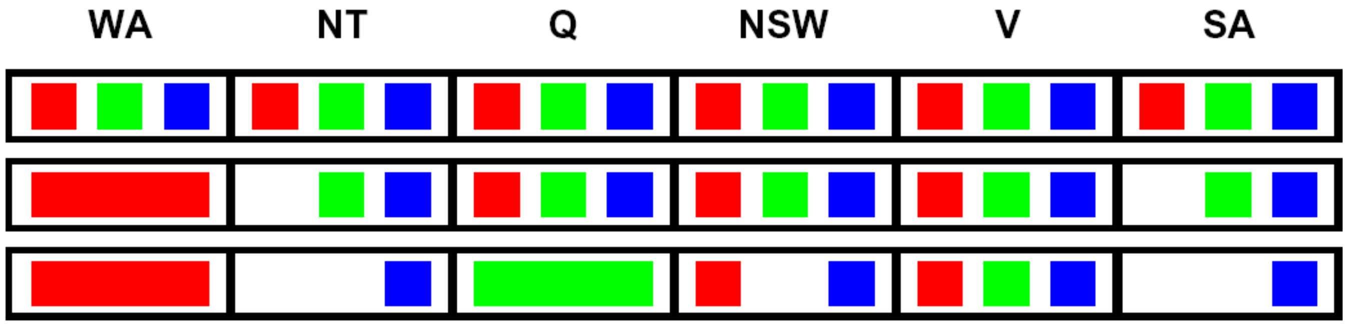


https://inst.eecs.berkeley.edu/~cs188/fa21/assets/demos/csp/csp_demos.html

Filtering: Constraint Propagation

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

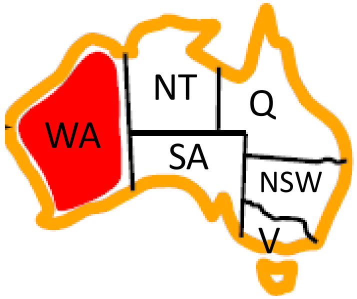


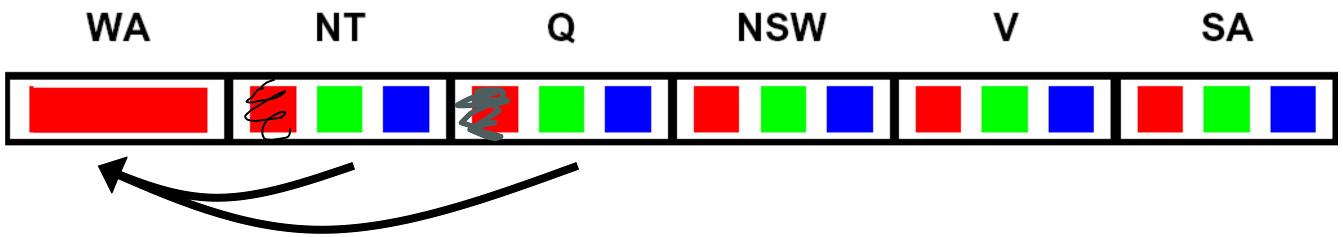


- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

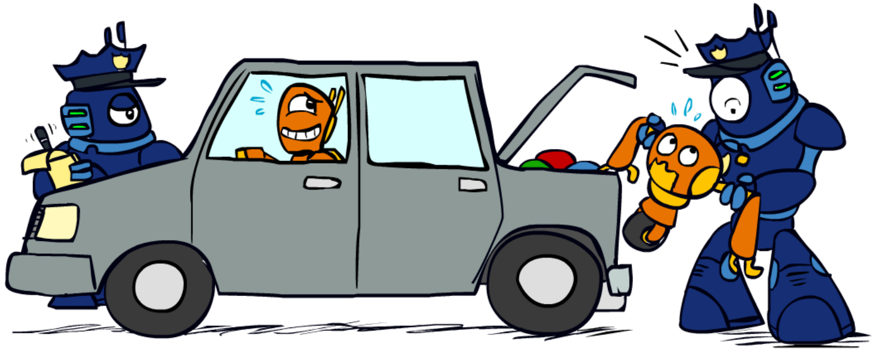
Consistency of A Single Arc

- 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





- Remove values in the domain of X if there isn't a corresponding legal Y



Delete from the tail!

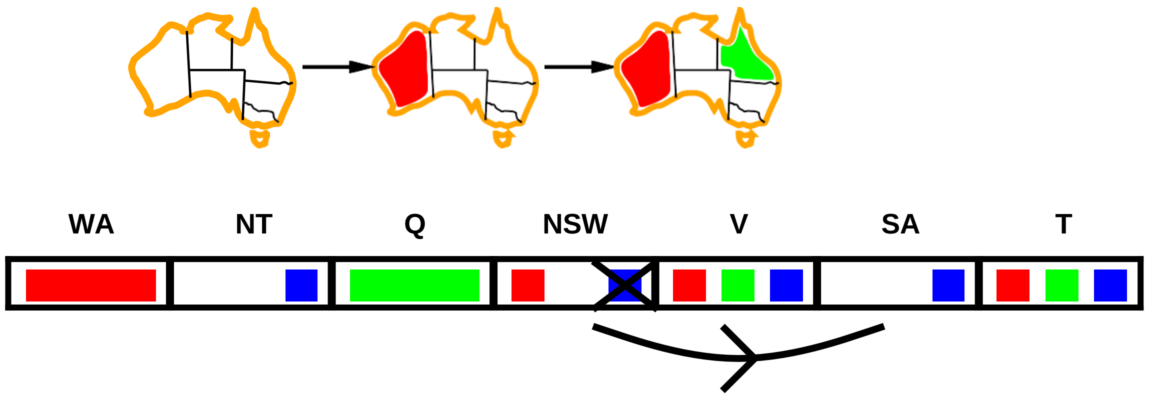
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

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

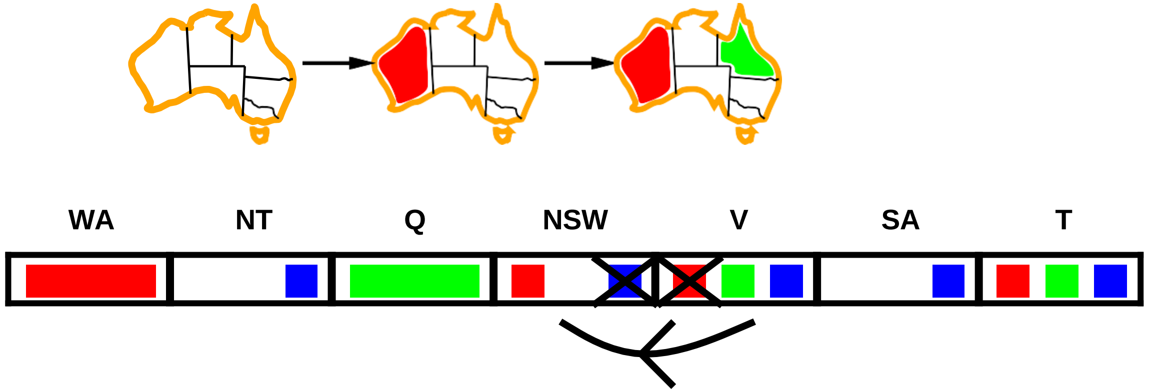


Arc consistency

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



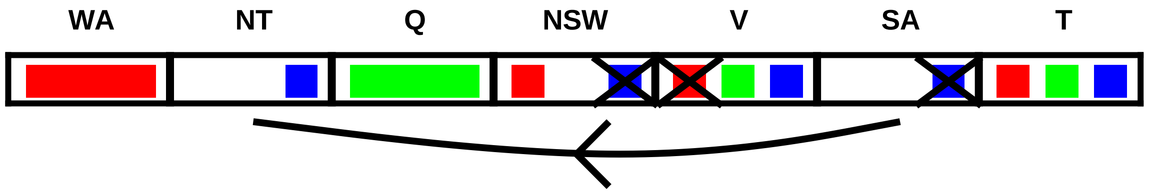
If X loses a value, neighbors of X need to be rechecked

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



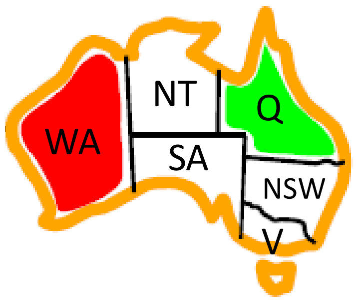
If X loses a value, neighbors of X need to be rechecked

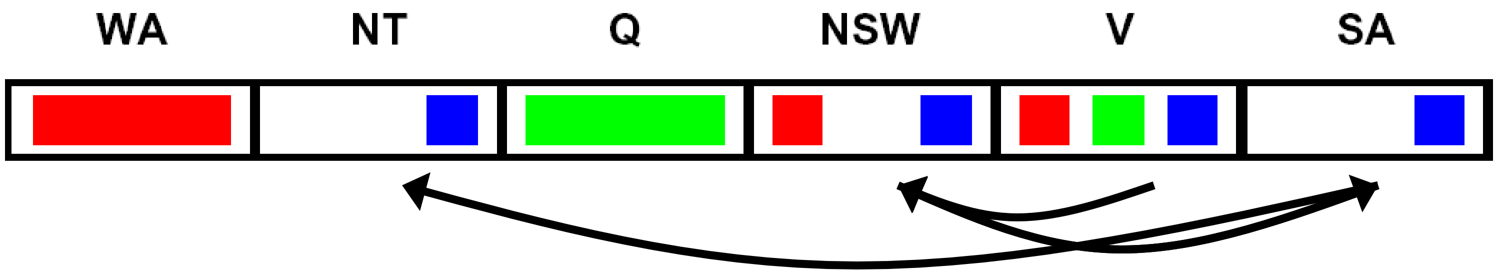
Arc consistency detects failure earlier than forward checking

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 {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
```

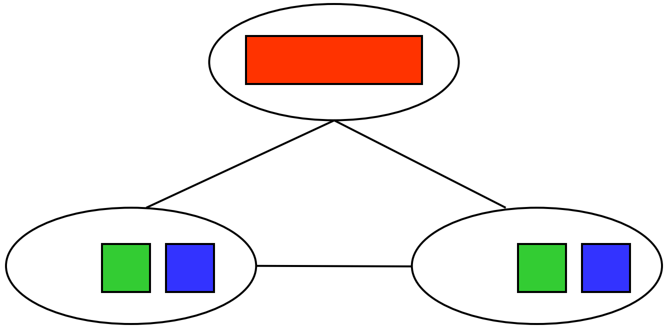
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

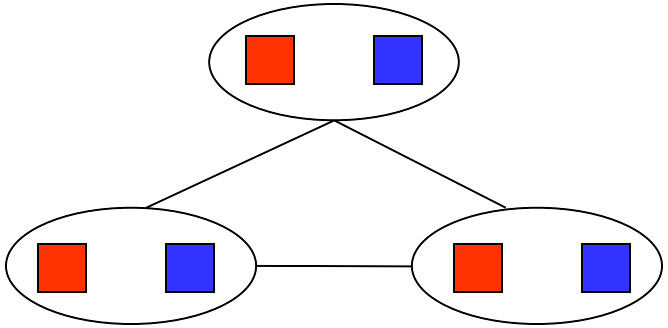
[Demo: CSP applet (made available by aispace.org) -- n-queens]

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!

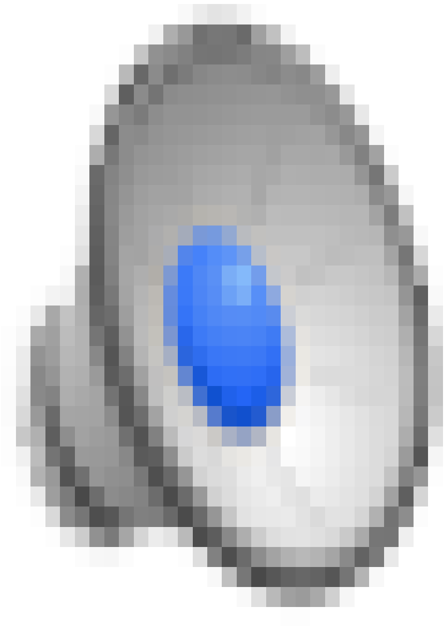




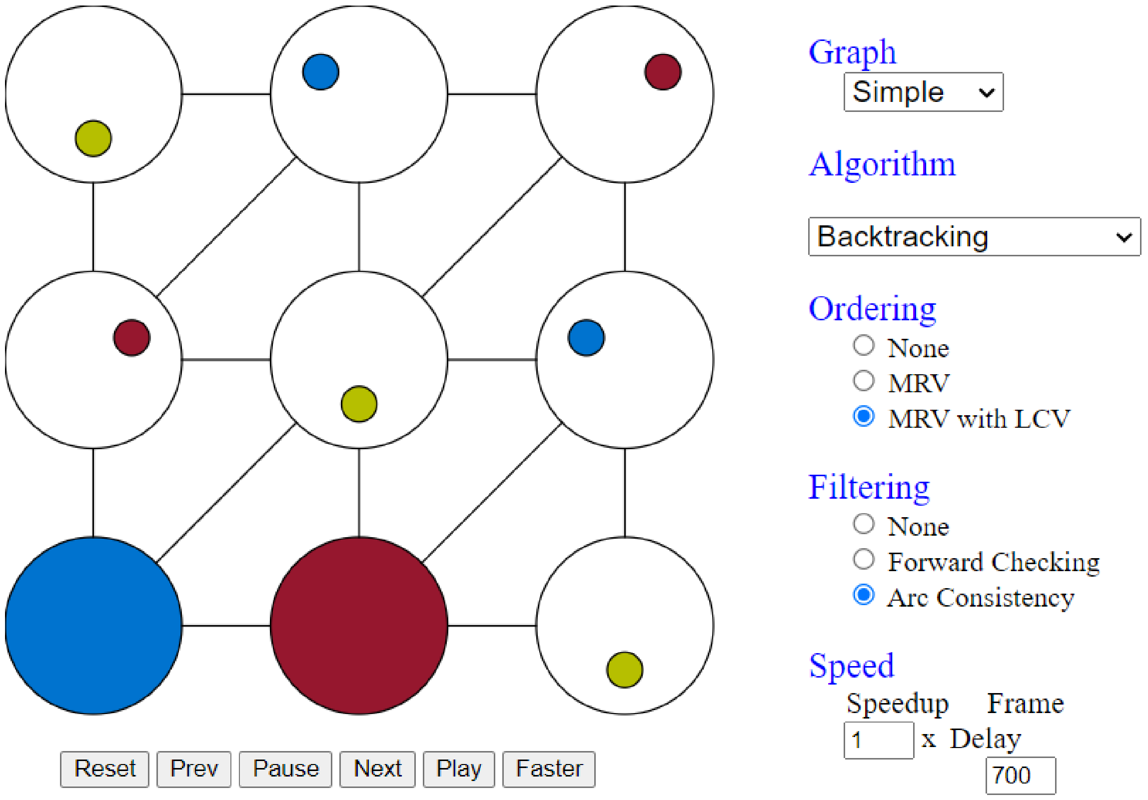
What went wrong here?

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph



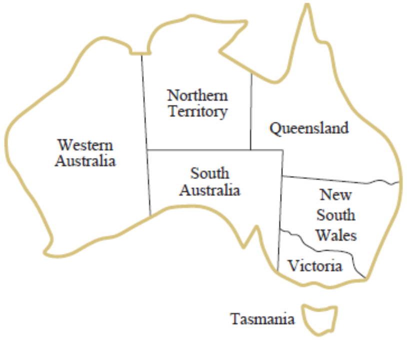
Backtracking with Arc Consistency



https://inst.eecs.berkeley.edu/~cs188/fa21/assets/demos/csp/csp_demos.html

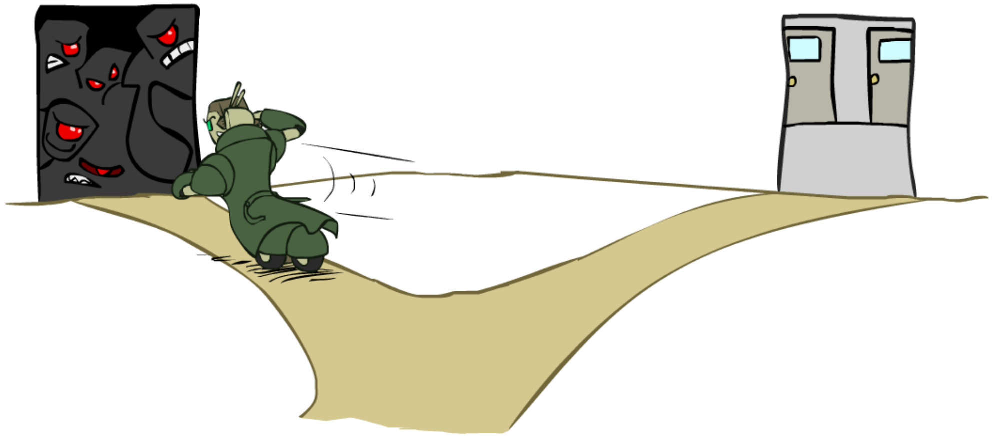
Ordering

Ordering: Minimum Remaining Values



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

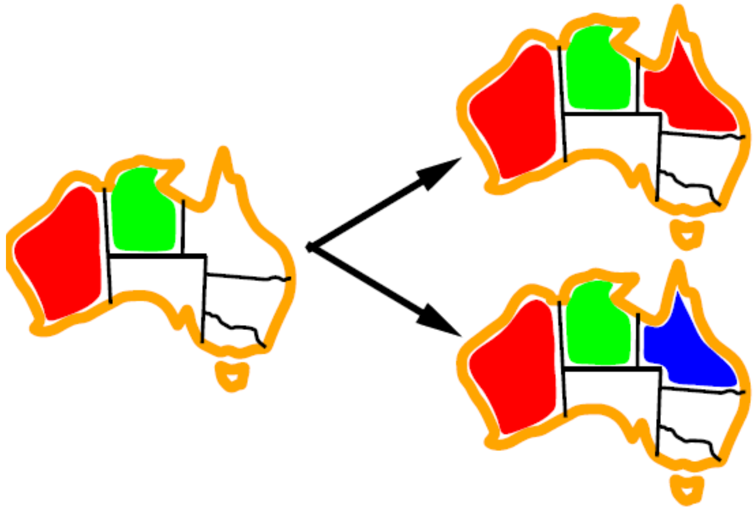
  - There is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.



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

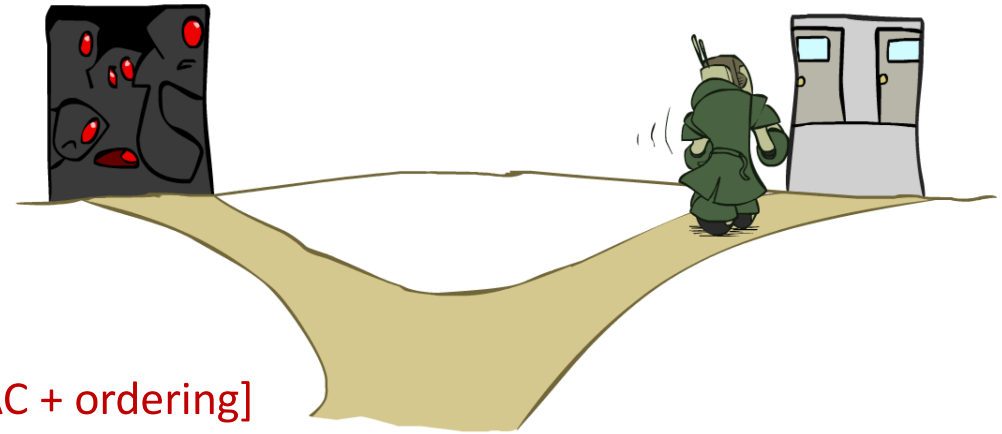
Ordering: Least Constraining Value

- 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)



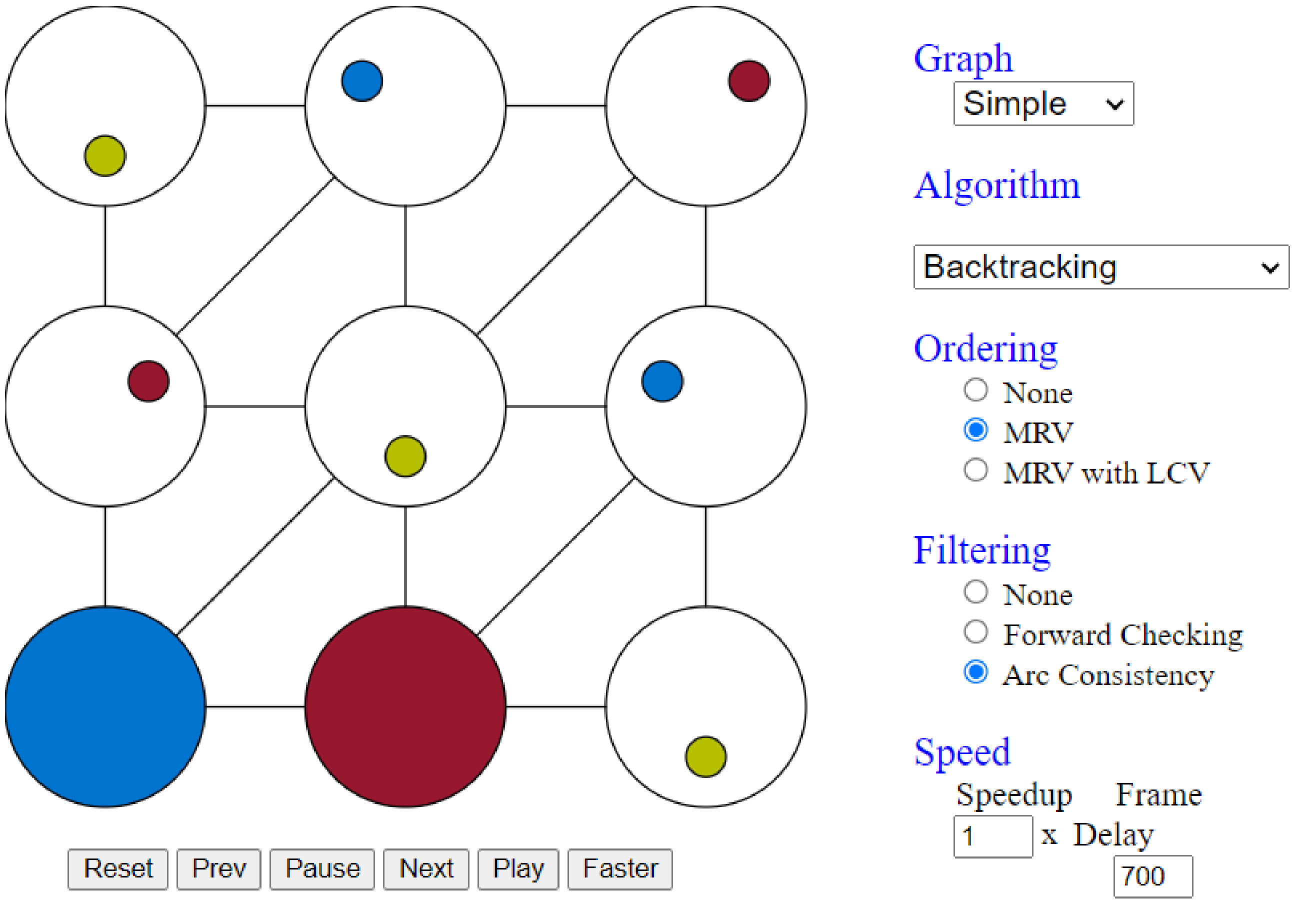
- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible



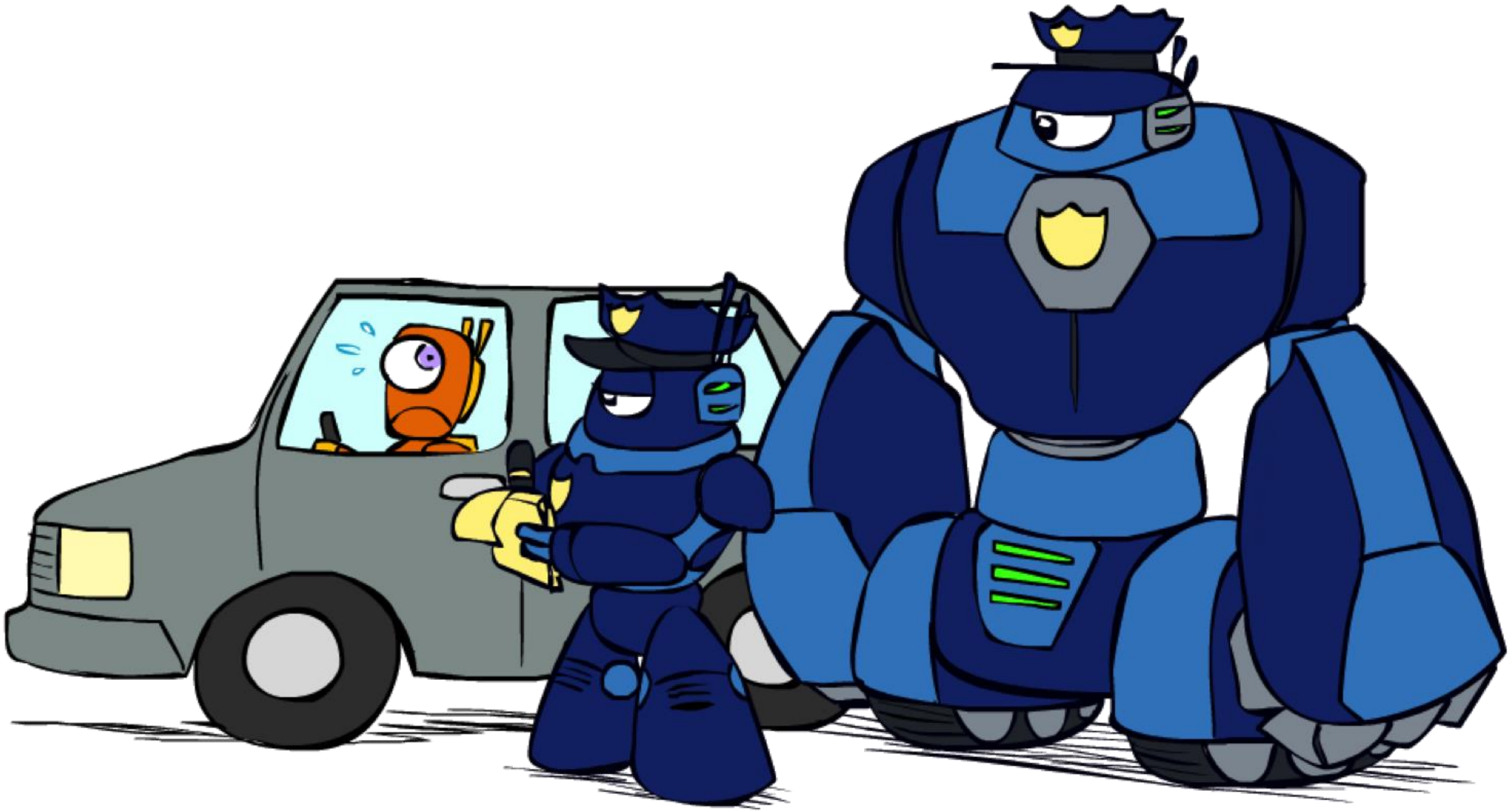
[Demo: coloring – backtracking + AC + ordering]

Demo: Backtracking + AC + Ordering



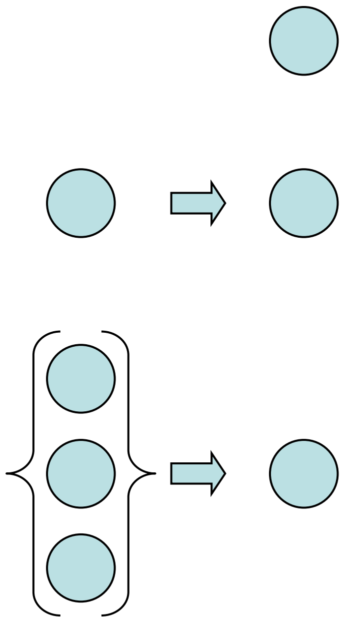
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

K-Consistency

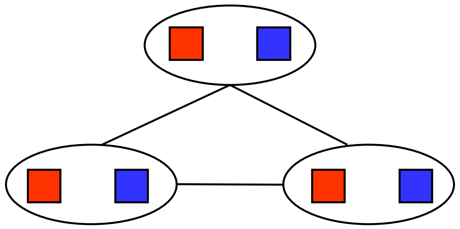


K-Consistency

- 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 k^{th} node.



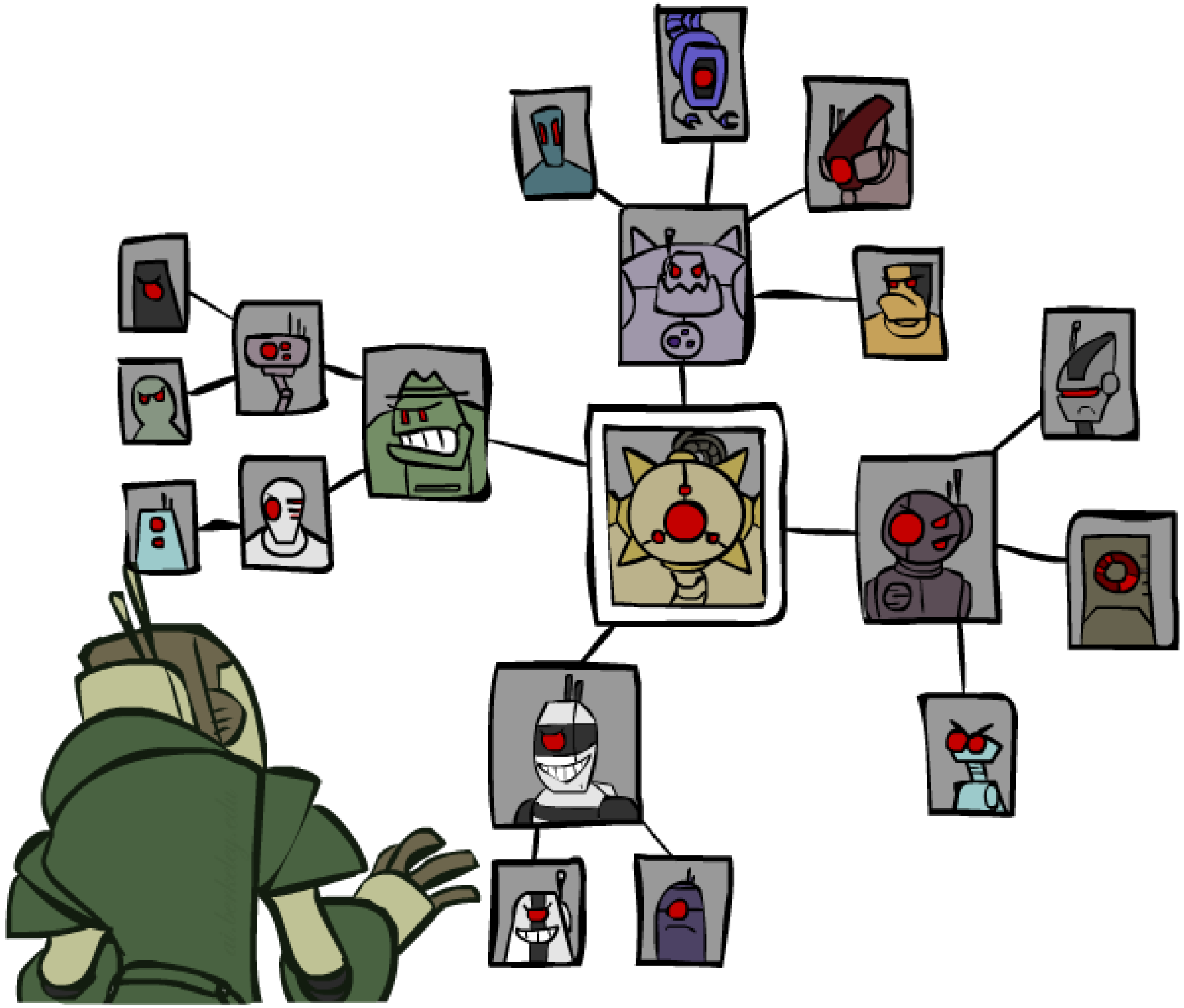
- Higher k --> more expensive to compute
- In practice, determining the appropriate level of consistency checking is mostly an empirical science. Computing 2-consistency is common, and 3-consistency less common.
- (You need to know the k=2 case: arc consistency)



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

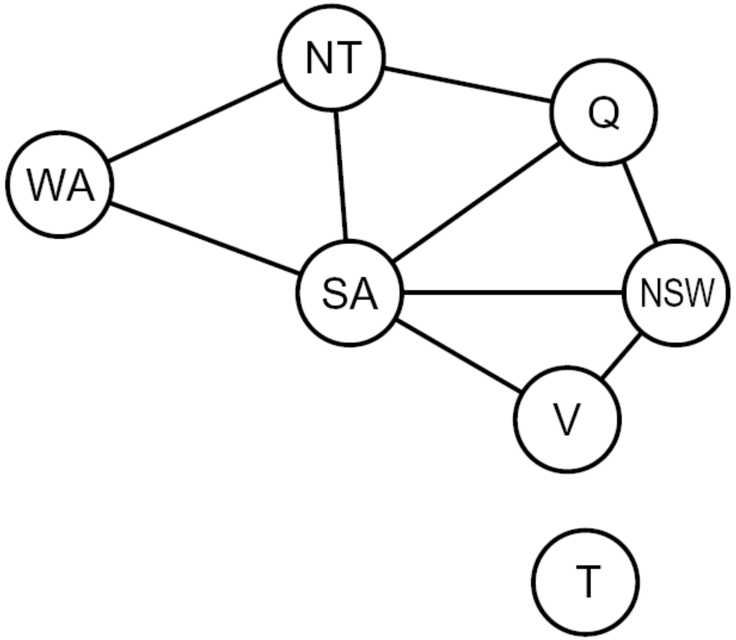


Problem Structure

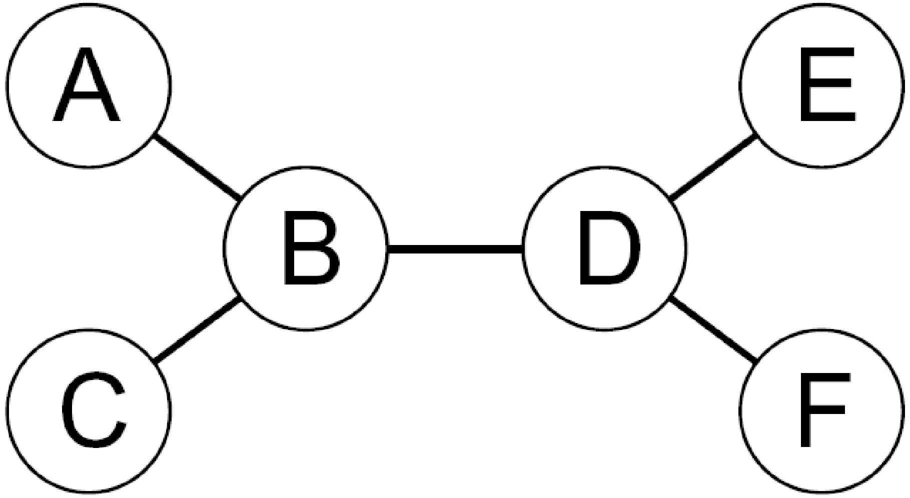
- We examine ways in which the structure of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning.
- The only way we can possibly hope to deal with the vast real world is to decompose it into subproblems. Looking again at the constraint graph for Australia (Figure 5.1(b), repeated as Figure 5.12(a)), one fact stands out: Tasmania is not connected to the mainland.3 Intuitively, it is obvious that coloring Tasmania and coloring the mainland are independent subproblems

- 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



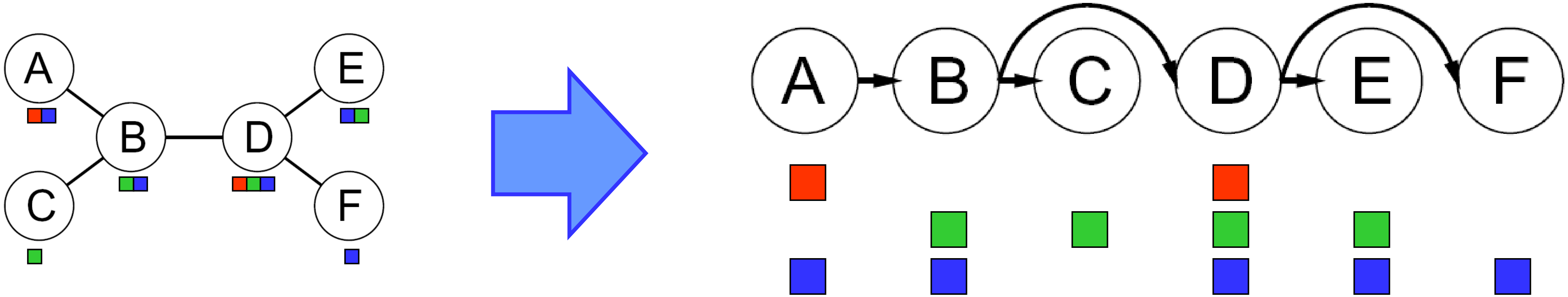
Tree-Structured CSPs



- 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:
  - To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering s called a topological sort.

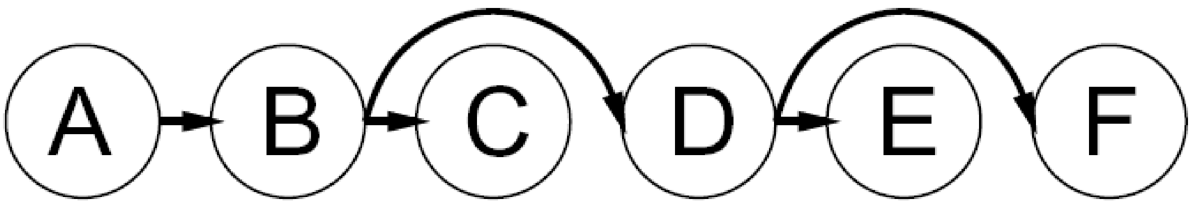


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Paren

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

Tree-Structured CSPs

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



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

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

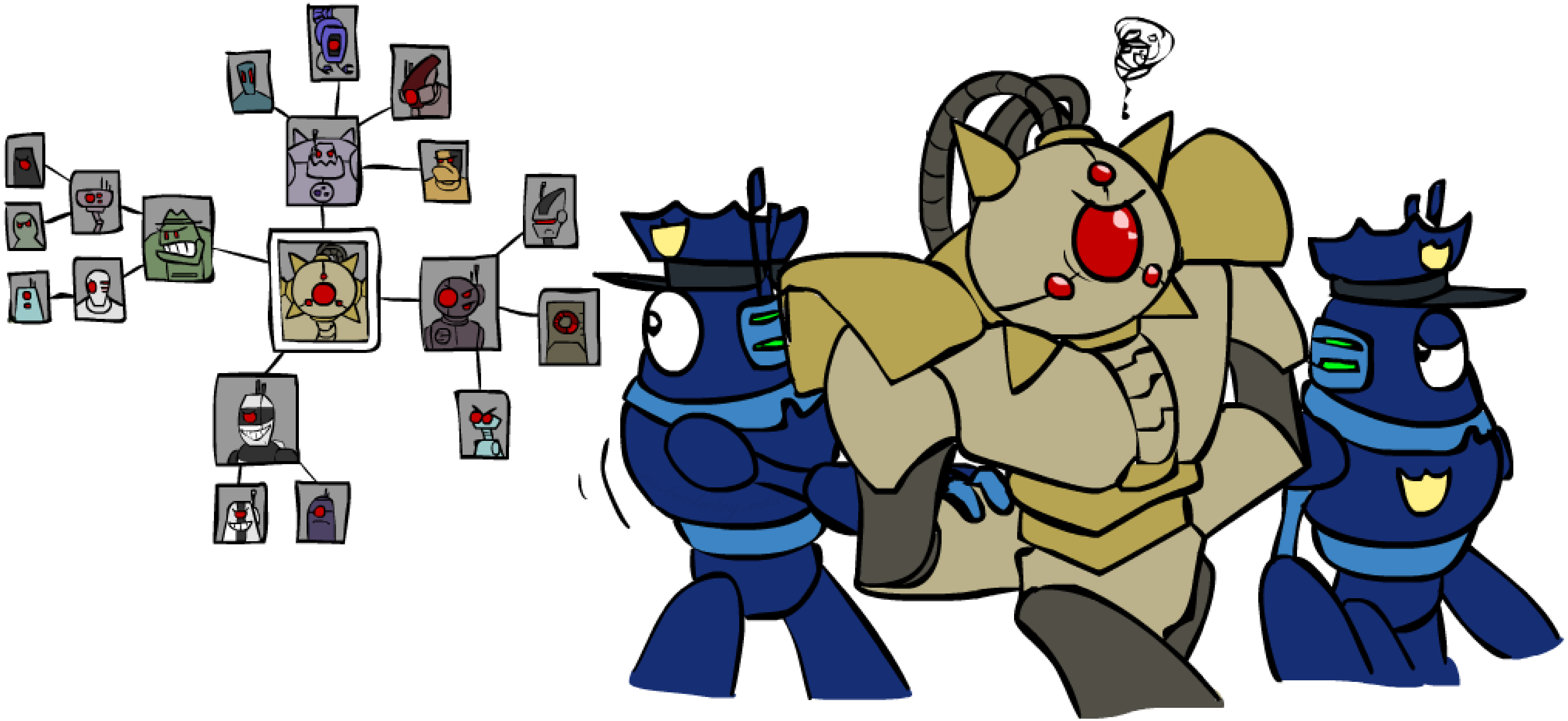
- Note: we'll see this basic idea again with Bayes' nets

The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs.

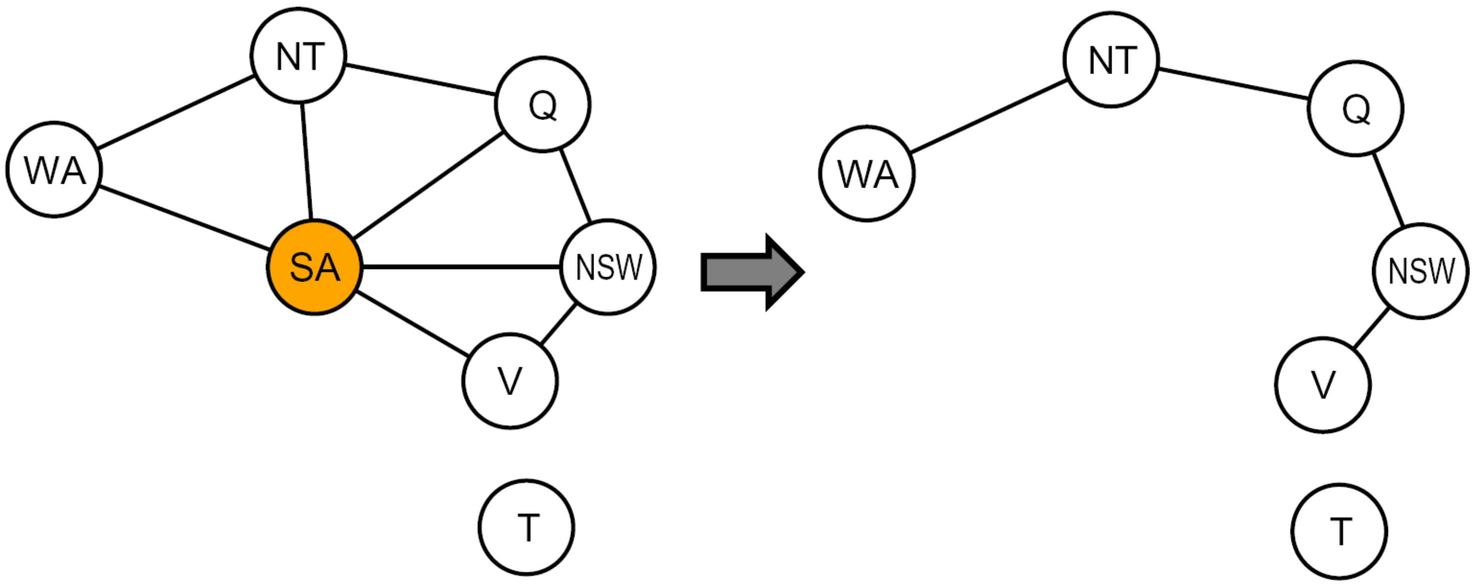
```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← 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] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 5.11 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.

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.
- We can solve the remaining tree with the TREE-CSP-SOLVER
- Without South Australia, the graph would become a tree
- Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c

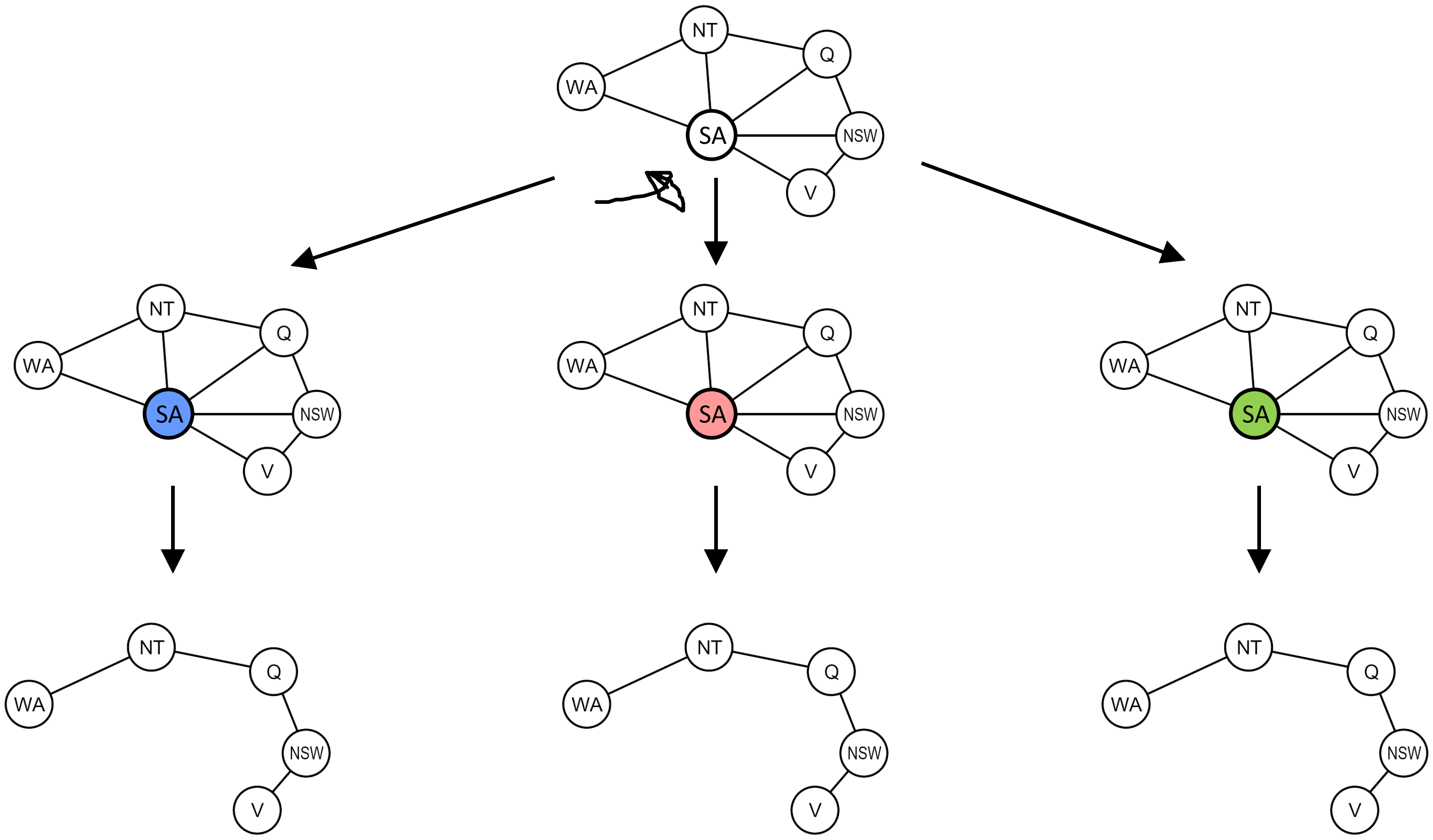
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

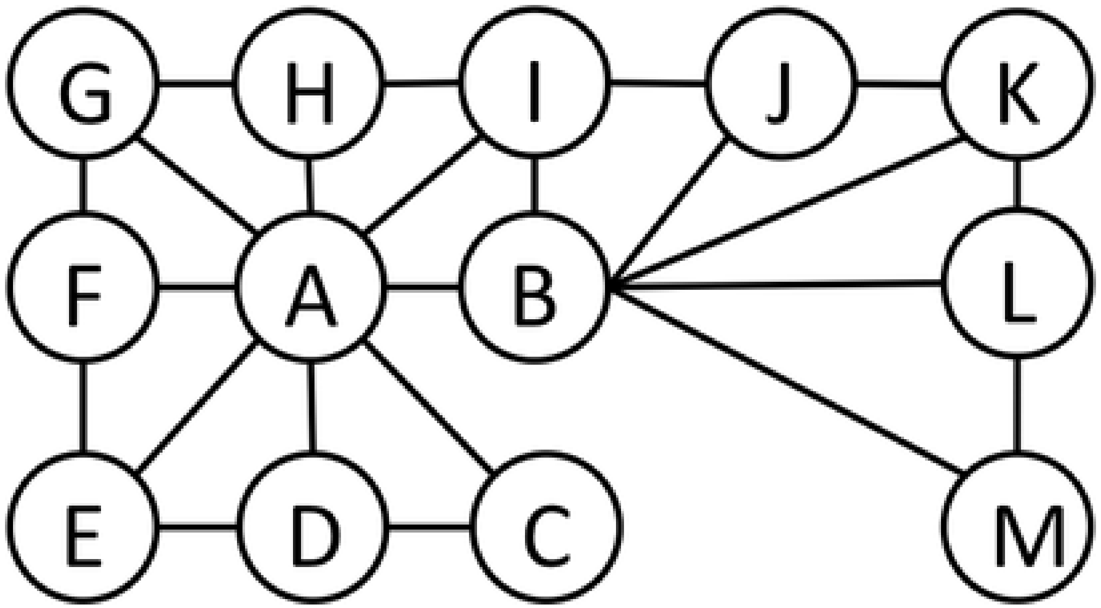
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



Cutset Quiz

- Find the smallest cutset for the graph below.



- The second way to reduce a constraint graph to a tree is based on constructing a tree decomposition of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables, as in Figure 5.13. A tree decomposition must satisfy these three requirements:
  - Every variable in the original problem appears in at least one of the tree nodes.
  - If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
  - If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

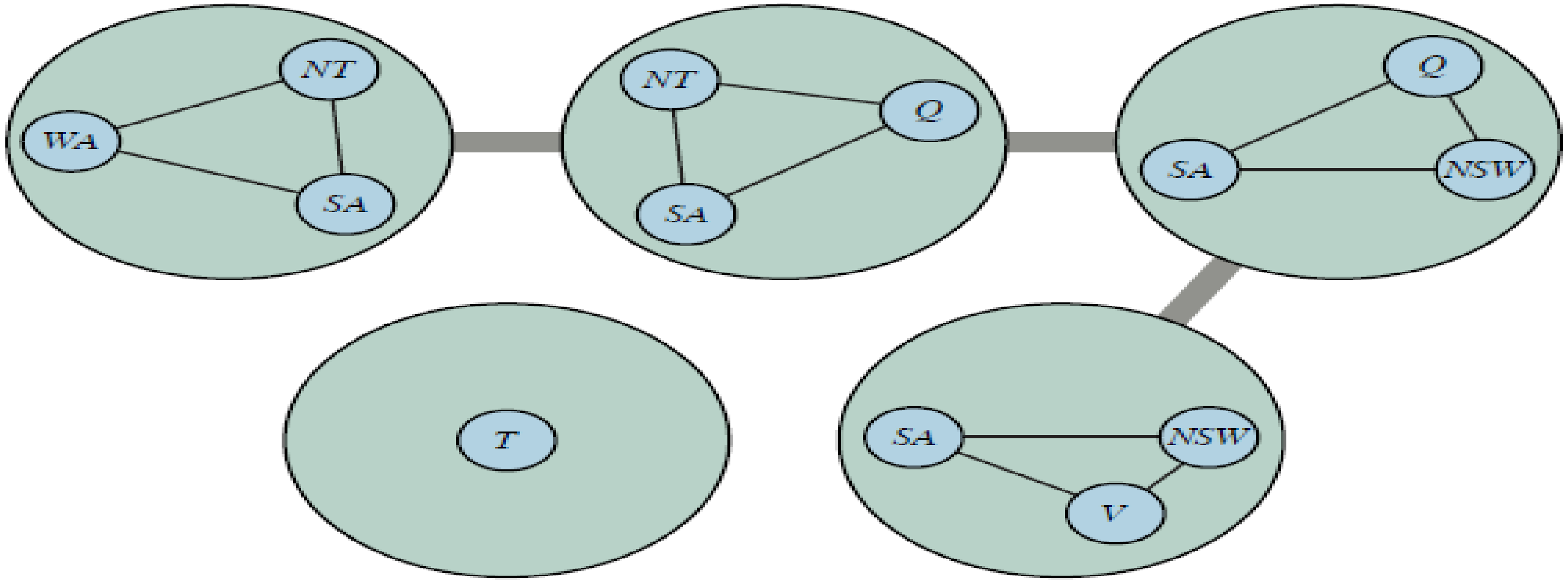
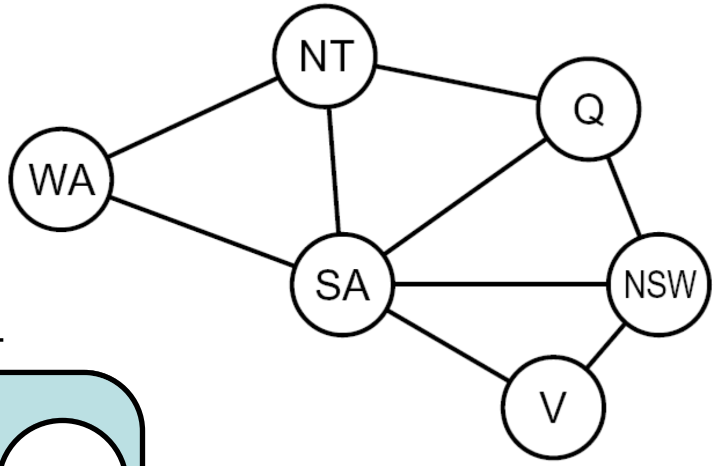
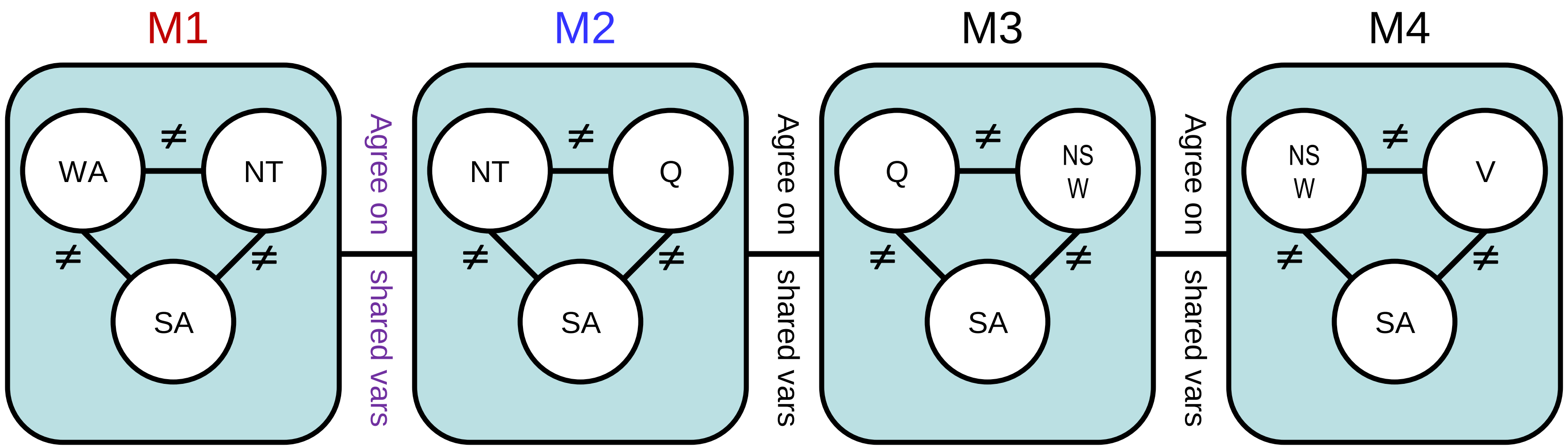


Figure 5.13 A tree decomposition of the constraint graph in Figure 5.12(a).

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





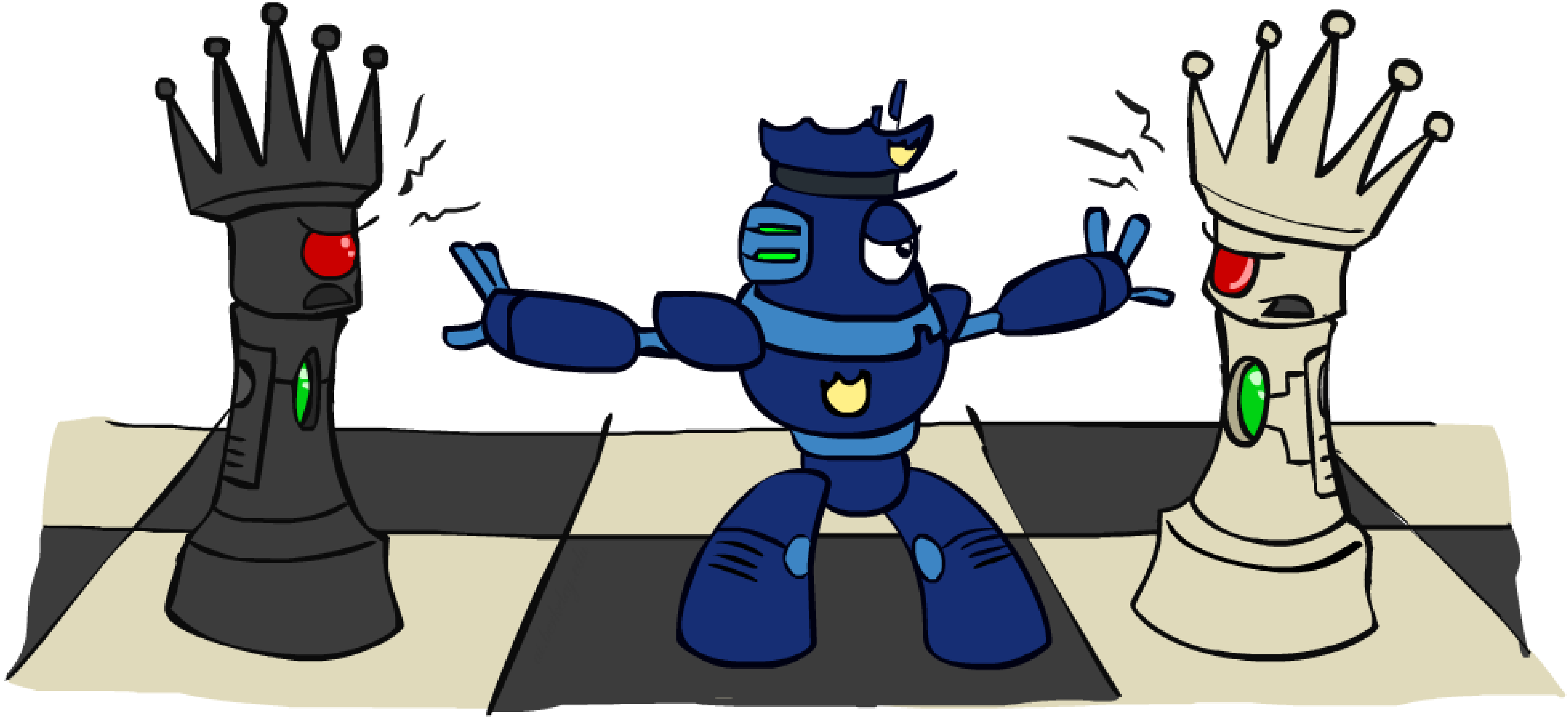
{(WA=r,SA=g,NT=b), (WA=b,SA=r,NT=g), …}

{(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)), …}

Once we have a tree-structured graph, we can apply Tree-CSP-Solver to get a solution in $O(nd^2)$ time, where n is the number of tree nodes and d is the size of the largest domain.

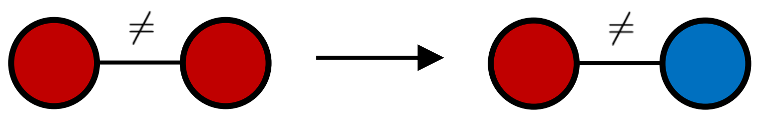
Iterative Improvement



Local Search 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

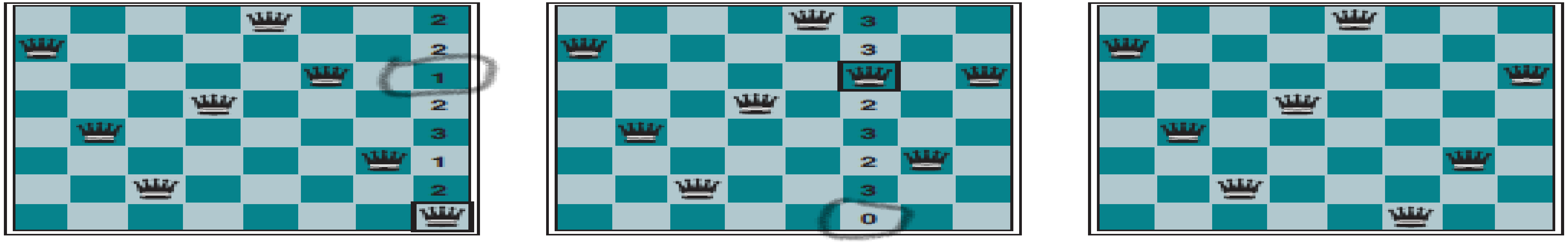
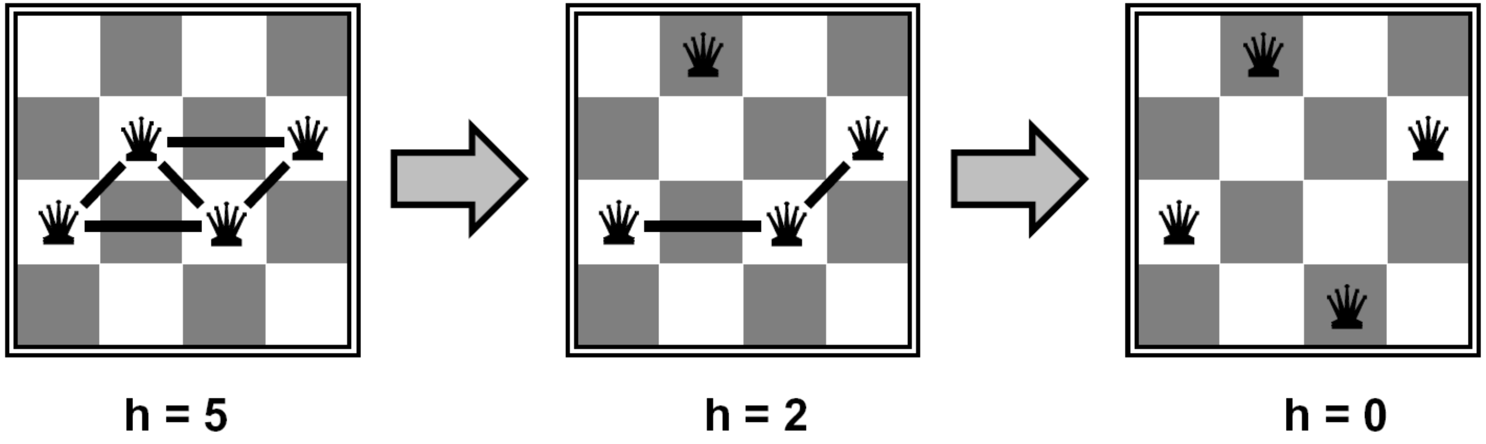


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking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

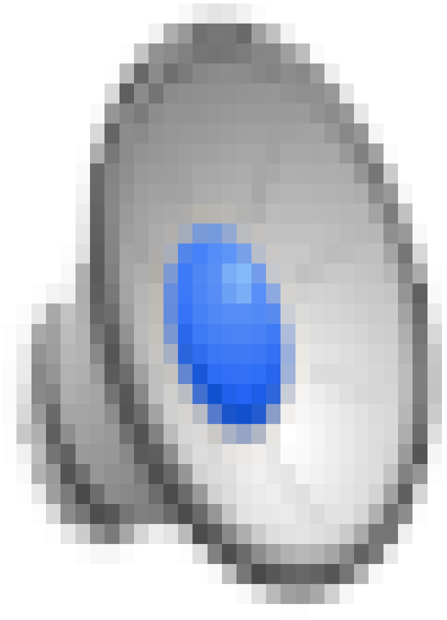
Example: 4-Queens



- 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

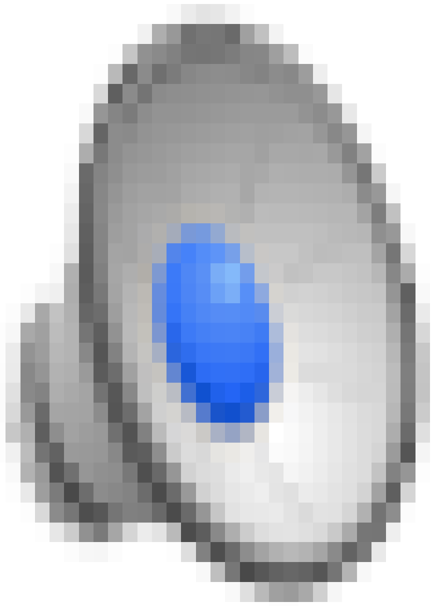
[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

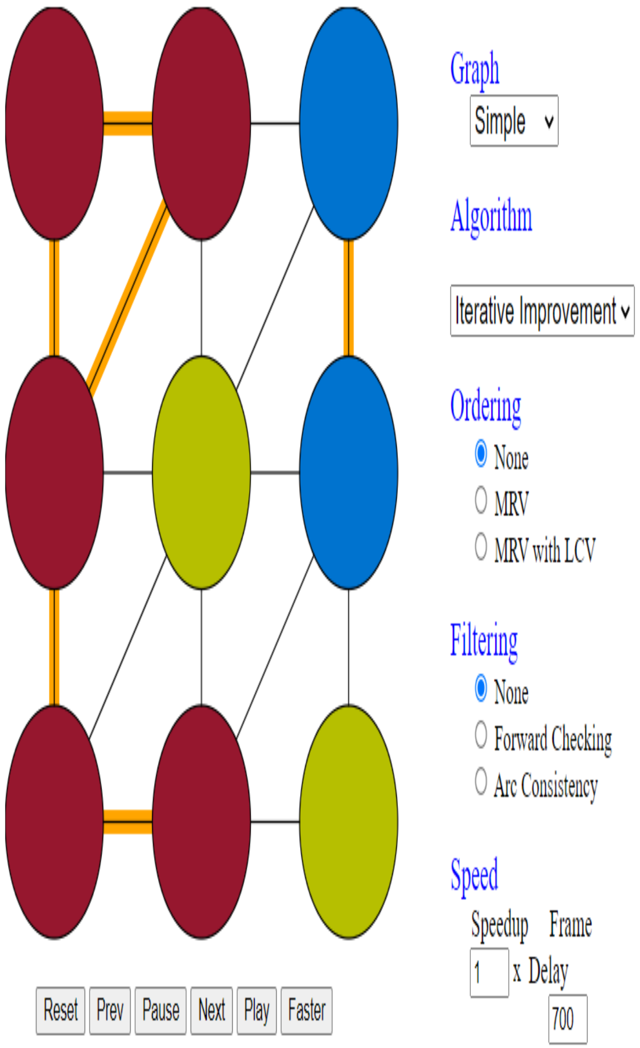
Video of Demo Iterative Improvement – n Queens



Video of Demo Iterative Improvement – Coloring

https://inst.eecs.berkeley.edu/~cs188/fa21/assets/demos/csp/csp_demos.html

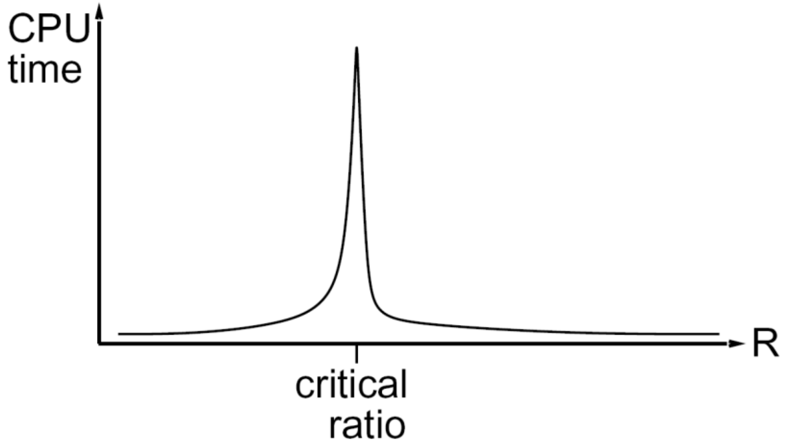


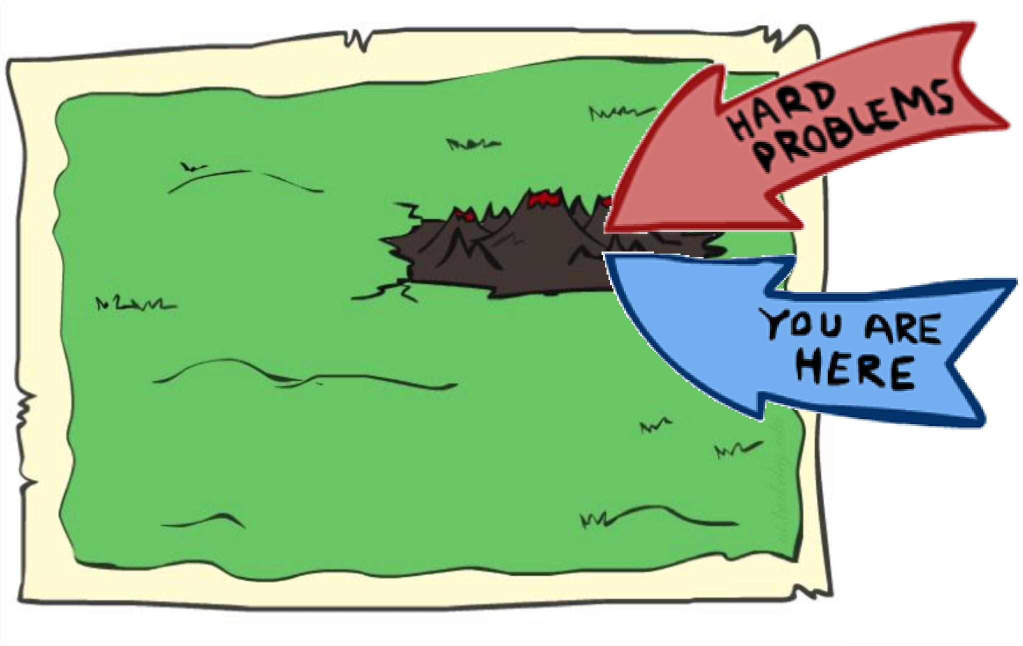


Performance of Min-Conflicts

- 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.

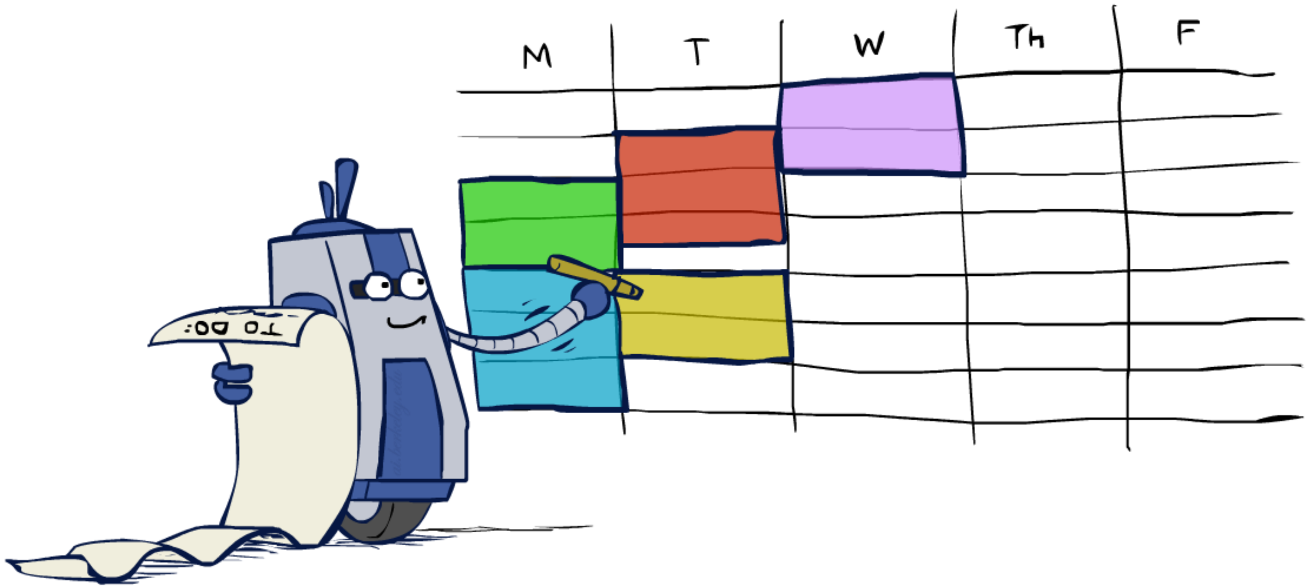
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





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/