CS 4100 // artificial intelligence

instructor: byron wallace

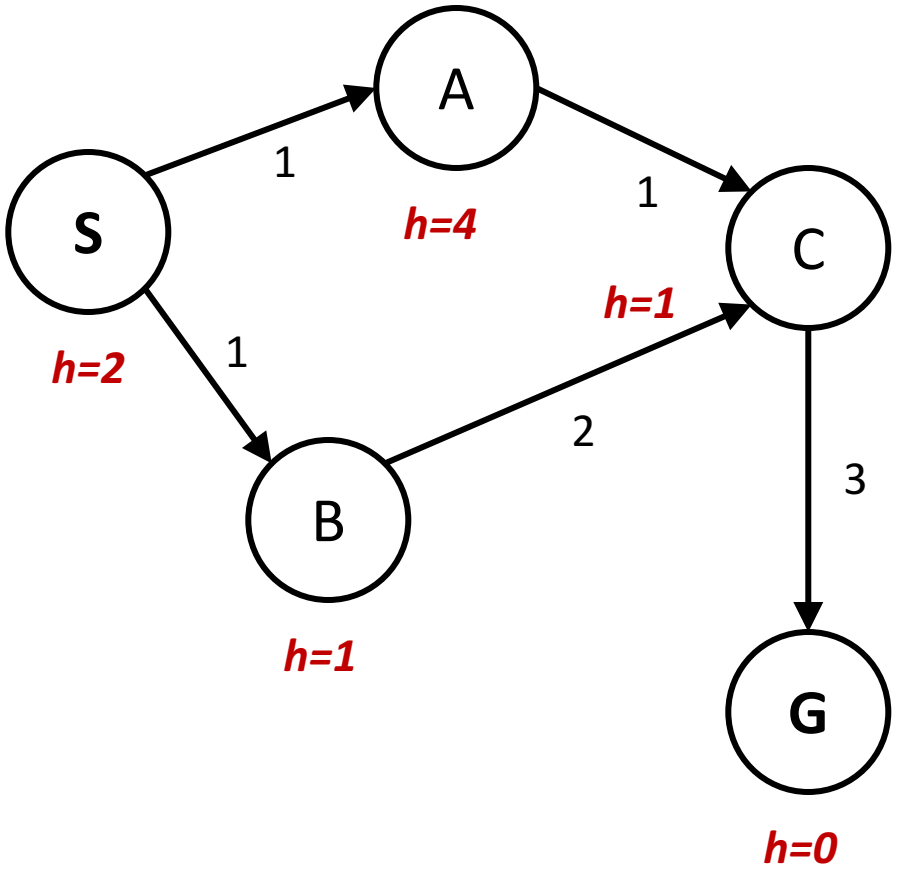
Constraint Satisfaction Problems

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

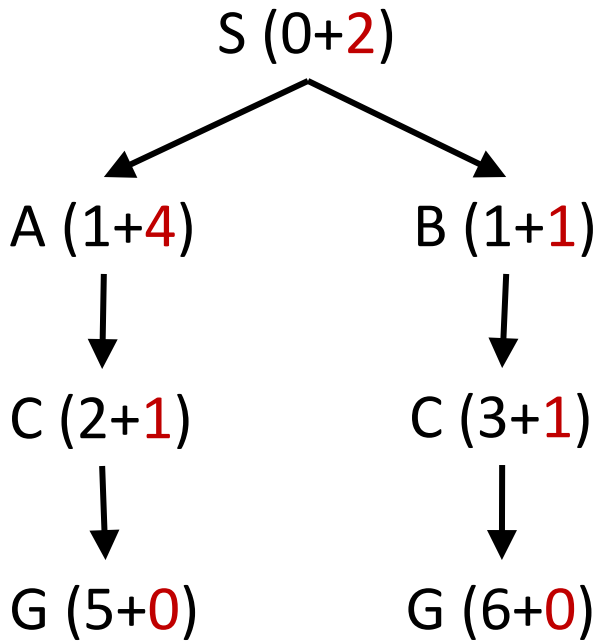
But first: wrapping up A* from last time

A* Graph search gone wrong?

State space graph

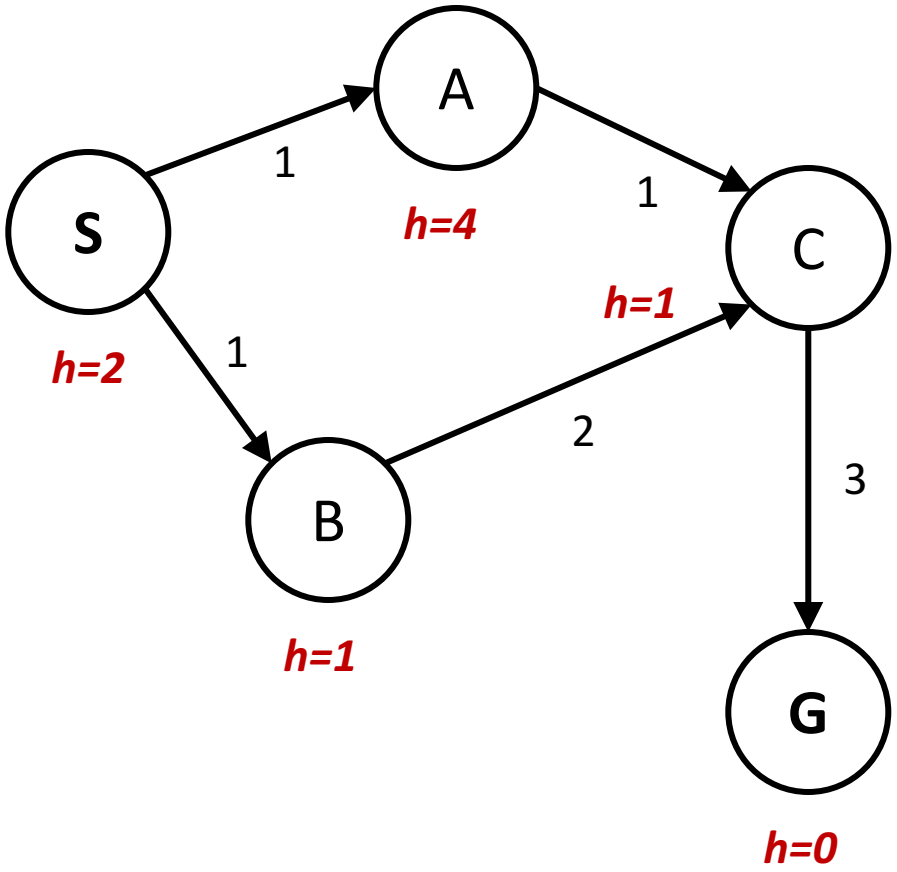


Search tree

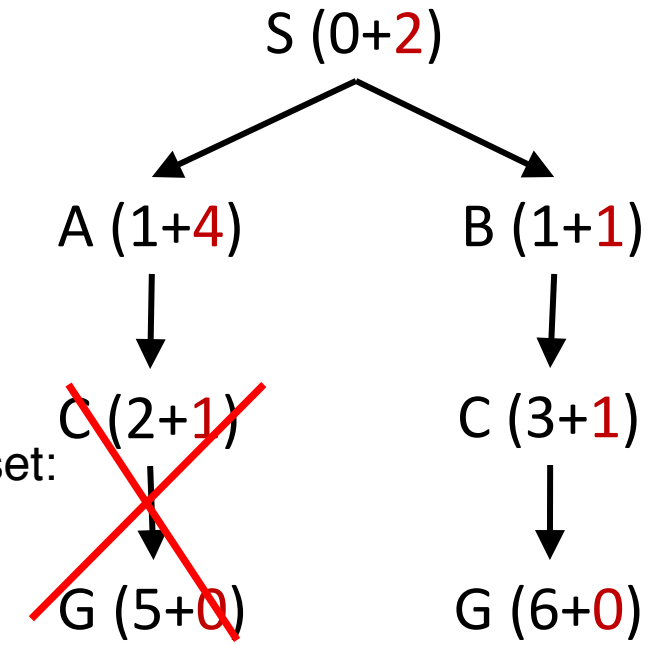


A* Graph search gone wrong?

State space graph



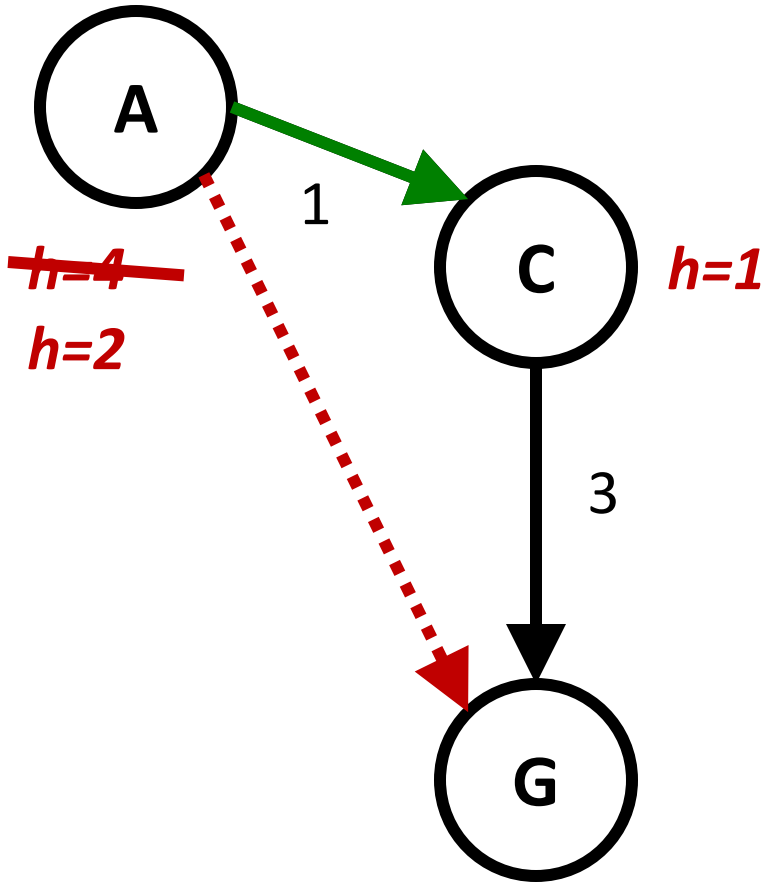
Search tree



Closed set:
{*SBC*}

Whoops! What went wrong??

Consistency of heuristics



Main idea: estimated heuristic costs ≤ actual costs

- **Admissibility**: heuristic cost ≤ actual cost to goal

  h(A) ≤ actual cost from A to G

- **Consistency**: heuristic "arc" cost ≤ actual cost for each arc

  h(A) – h(C) ≤ cost(A to C)

Consequences of consistency:

The f value along a path never decreases

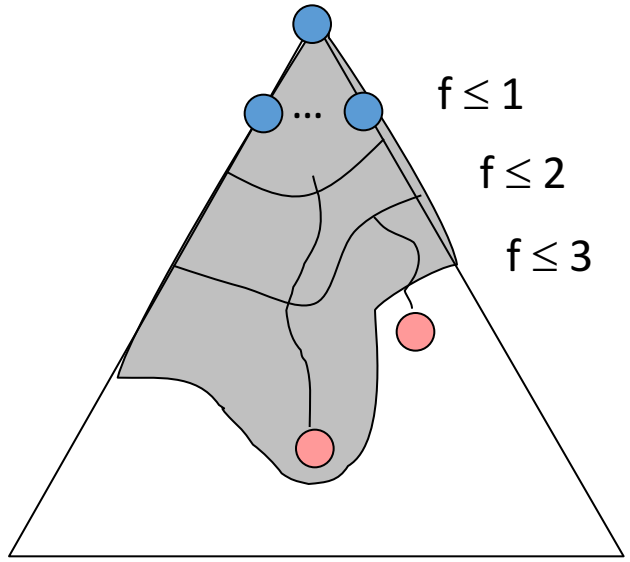
h(A) ≤ cost(A to C) + h(C)

A* graph search is optimal

Optimality of A* graph search

Sketch: consider what A* does with a consistent heuristic:

- Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
- Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
- Result: A* graph search is optimal



Optimality

Tree search:

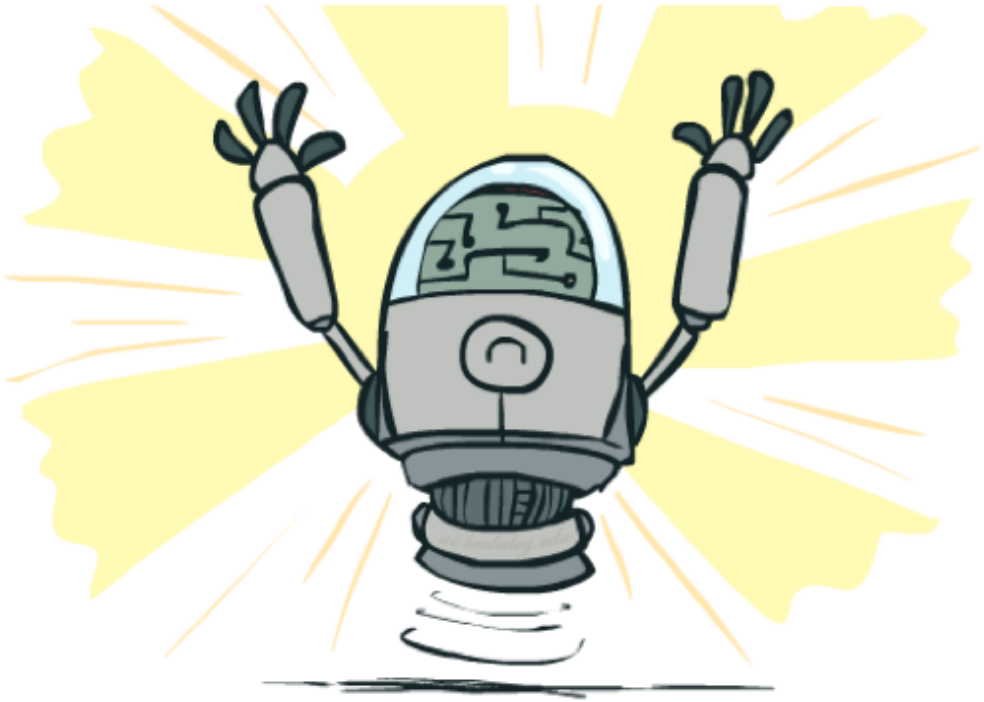
- A* is optimal if heuristic is admissible
- UCS is a special case (h = 0)

Graph search:

- A* optimal if heuristic is consistent
- UCS optimal (h=0 is consistent)

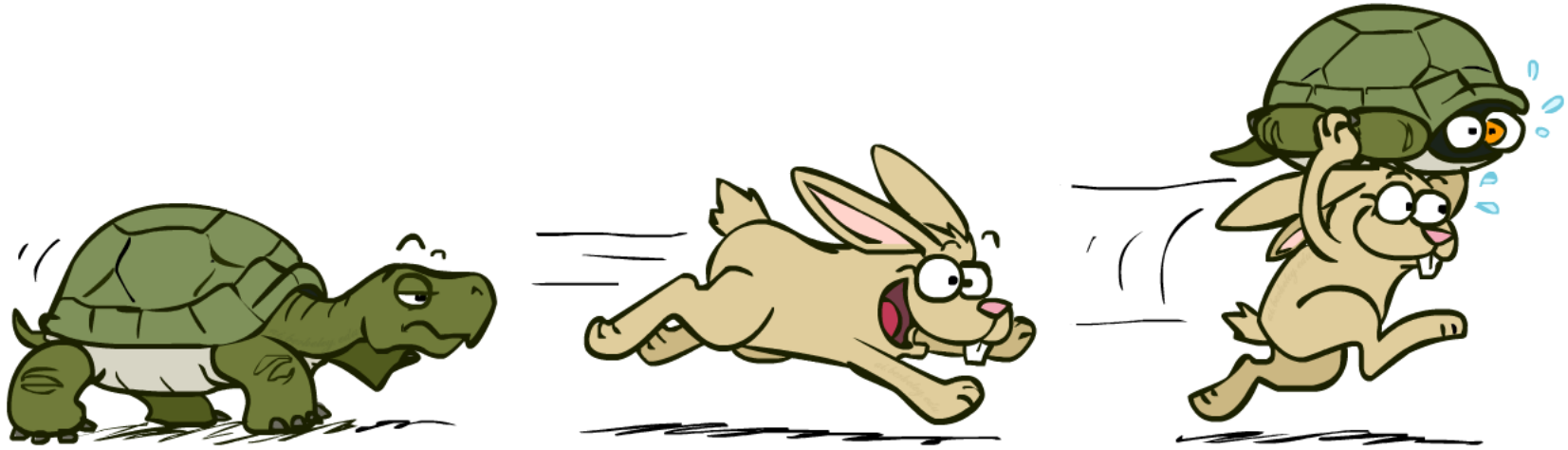
Consistency implies admissibility

In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A* Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



CS 4100 // artificial intelligence

instructor: byron wallace

Constraint Satisfaction Problems

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

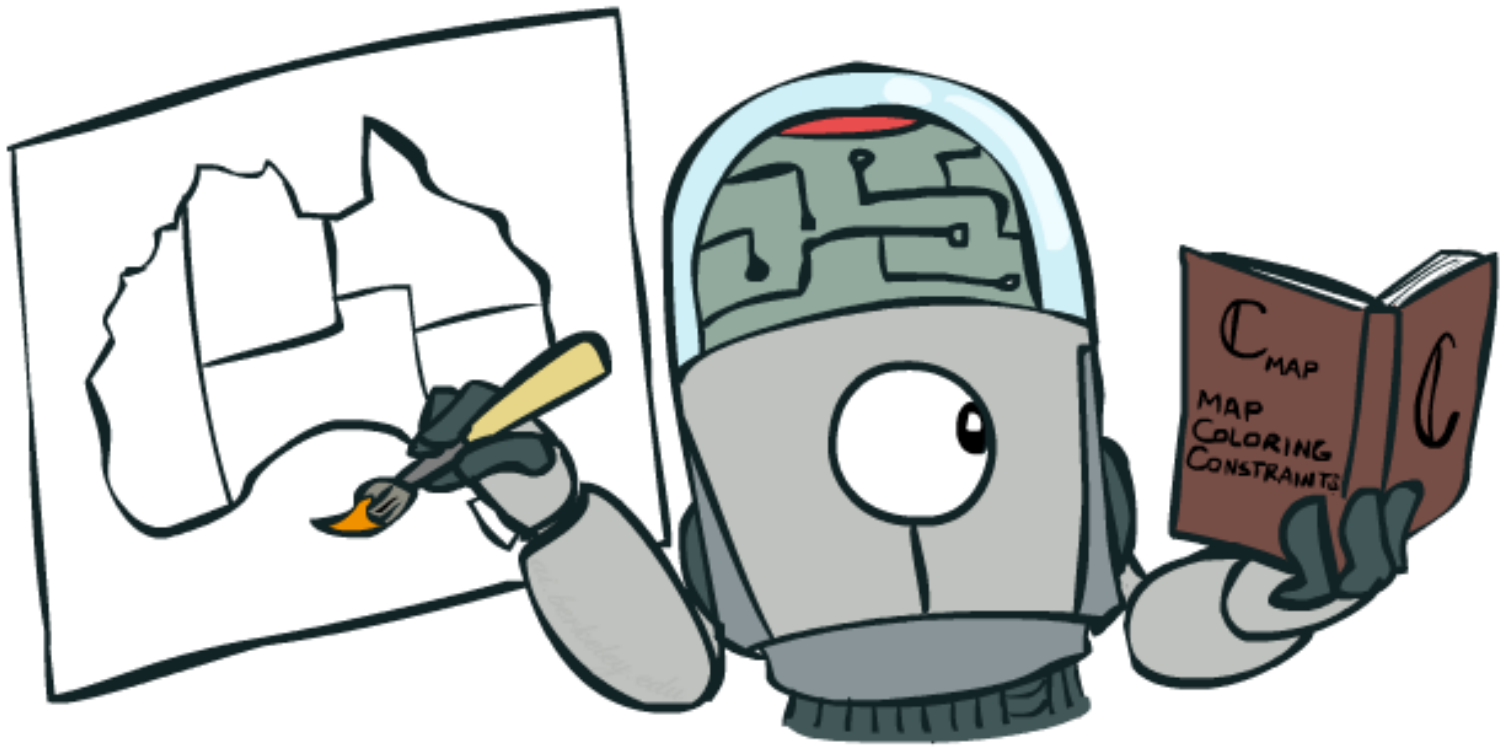
Search, so far

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

- So far we have mostly cared about *planning* or finding sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

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

Constraint satisfaction problems



Constraint satisfaction problems

Standard search problems:

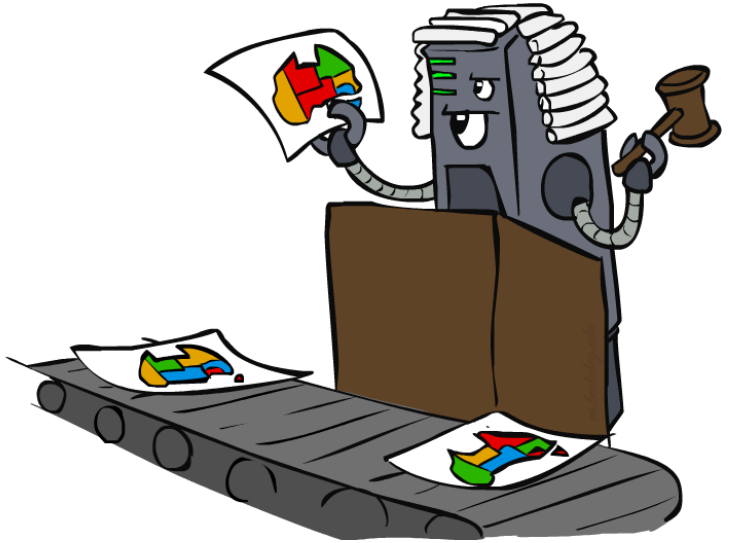
- State is a “black box”: arbitrary data structure
- Goal test can be any function over states
- Successor function can also be anything

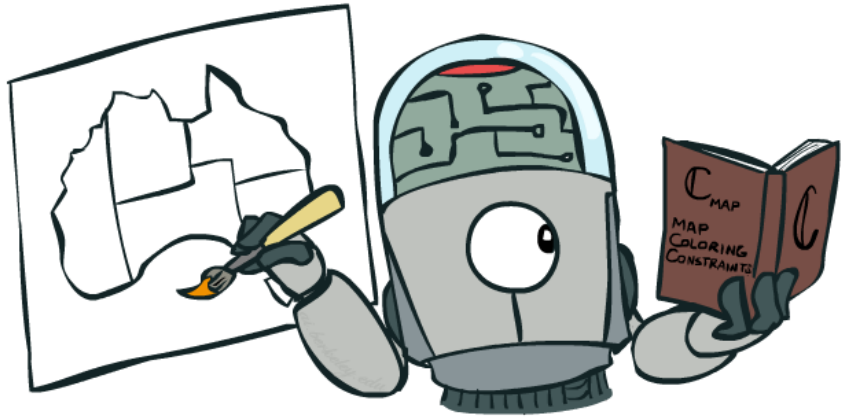
Constraint satisfaction problems (CSPs):

- A special subset of search problems
- State is defined by variables X_i with values from a domain D (sometimes D depends on i)
- *Goal test* is a set of constraints specifying allowable combinations of values for subsets of variables

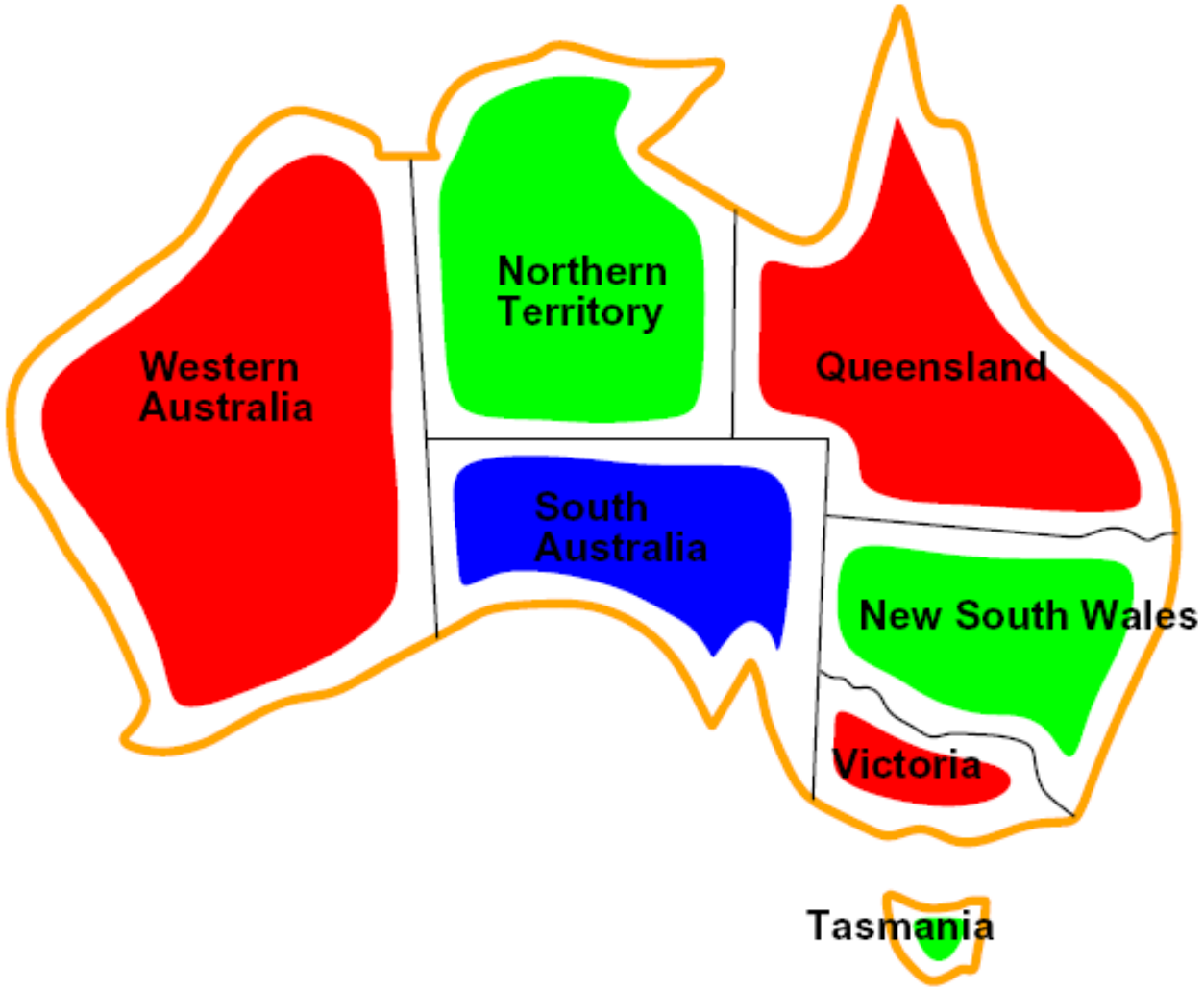
Simple example of a *formal representation language*

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





CSP Examples



Map coloring

Variables: $\mathrm{WA}, \mathrm{NT}, \mathrm{Q}, \mathrm{NSW}, \mathrm{V}, \mathrm{SA}, \mathrm{T}$

Domain: $D = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$

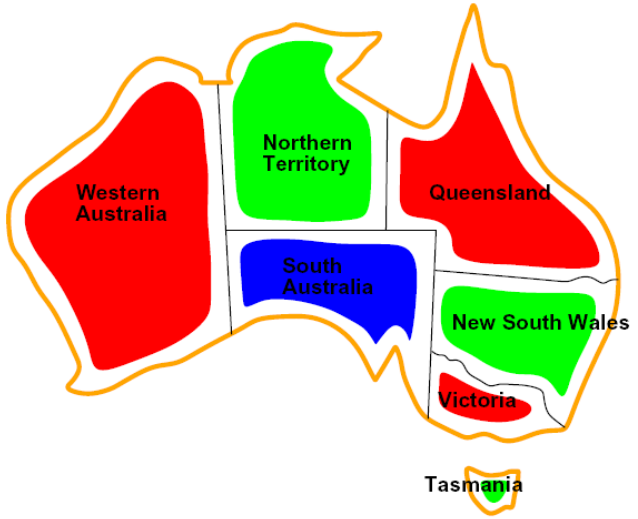
Constraints: adjacent regions must have different colors

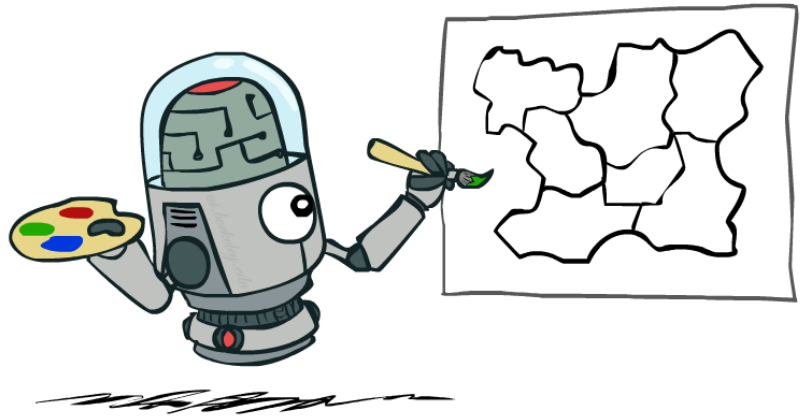
Implicit: $\mathrm{WA} \neq \mathrm{NT}$

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

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

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

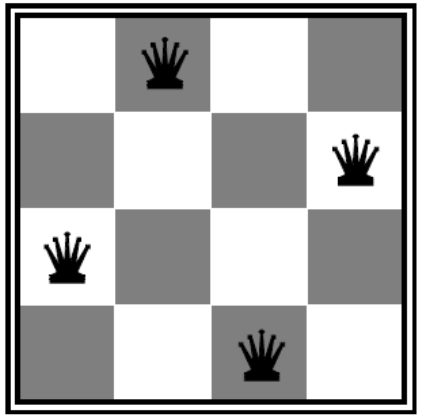


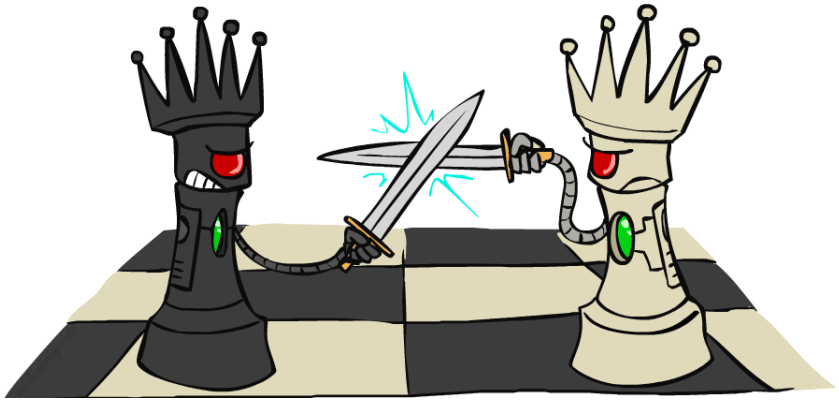


N-Queens

Formulation 1

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





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

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

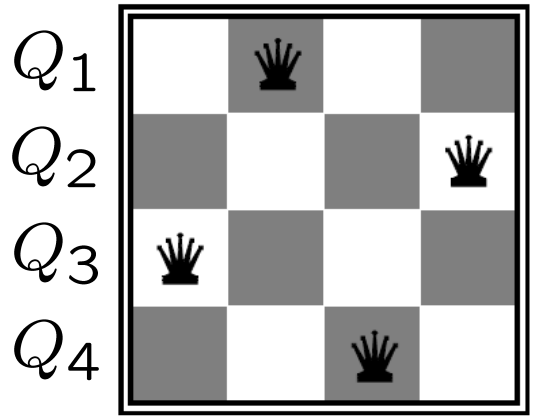
N-Queens

Formulation 2

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

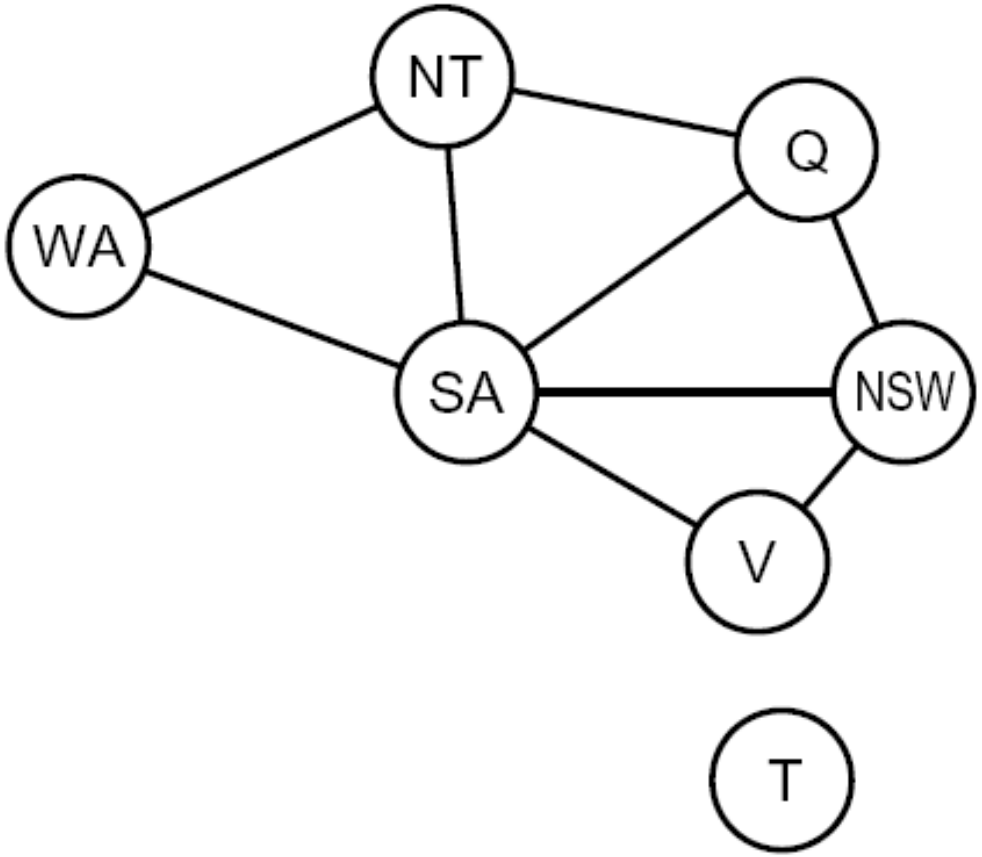
  Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

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



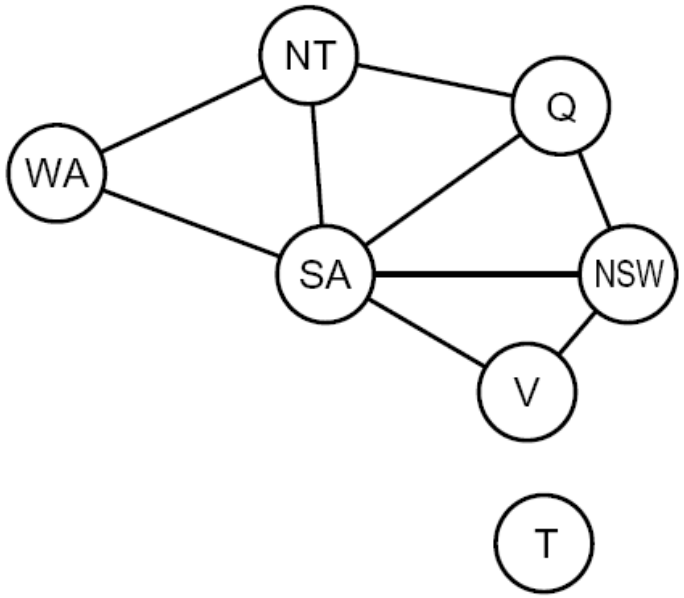
. . .

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!



Cryptarithmetic

Variables:

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

Domains:

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

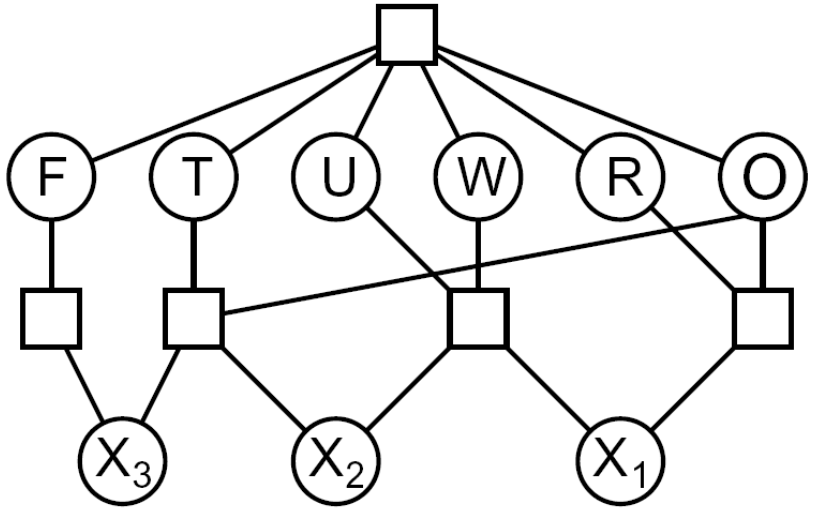
Constraints:

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

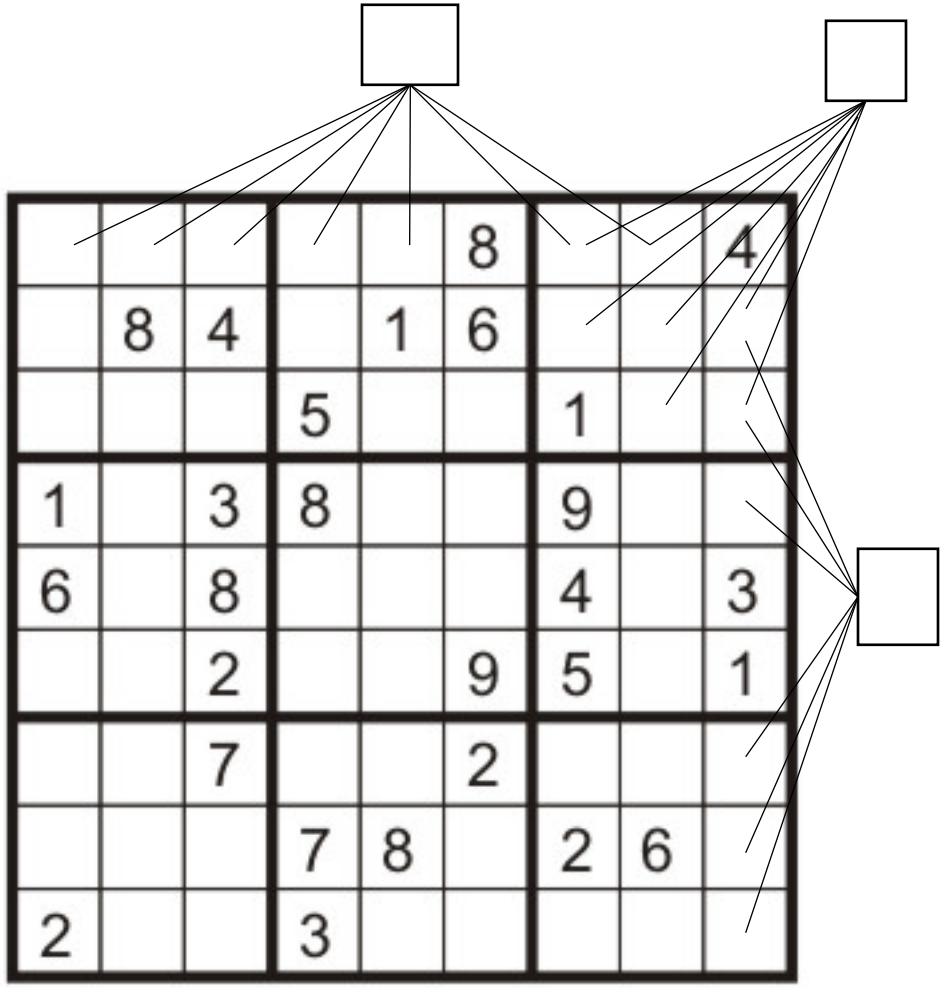
$$O + O = R + 10 \cdot X_1$$

$$\cdots$$

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



Sudoku



Variables:

Each (open) square

Domains:

{1,2,…,9}

Constraints:

9-way all-diff for each column

9-way all-diff for each row

9-way all-diff for each *region*

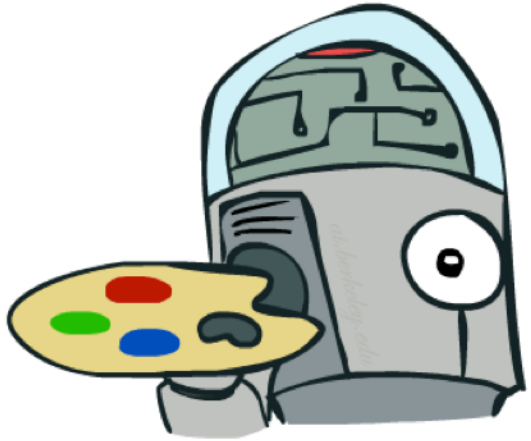
(or can have a bunch of pairwise inequality constraints)

Varieties of CSPs and constraints

Varieties of CSPs

Discrete Variables

- Finite domains
  - n variables with domain 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 Linear Programming methods (we won't cover this here, but a huge and important topic)

Varieties of constraints

Varieties of Constraints

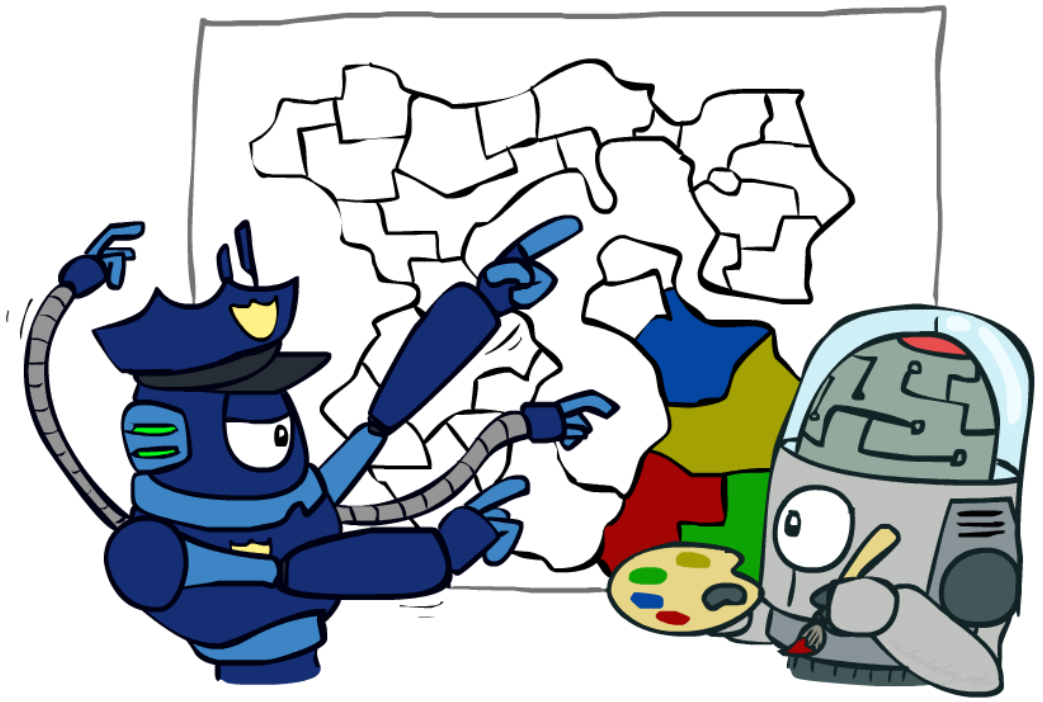
- Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq green$$

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

$$SA \neq WA$$

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

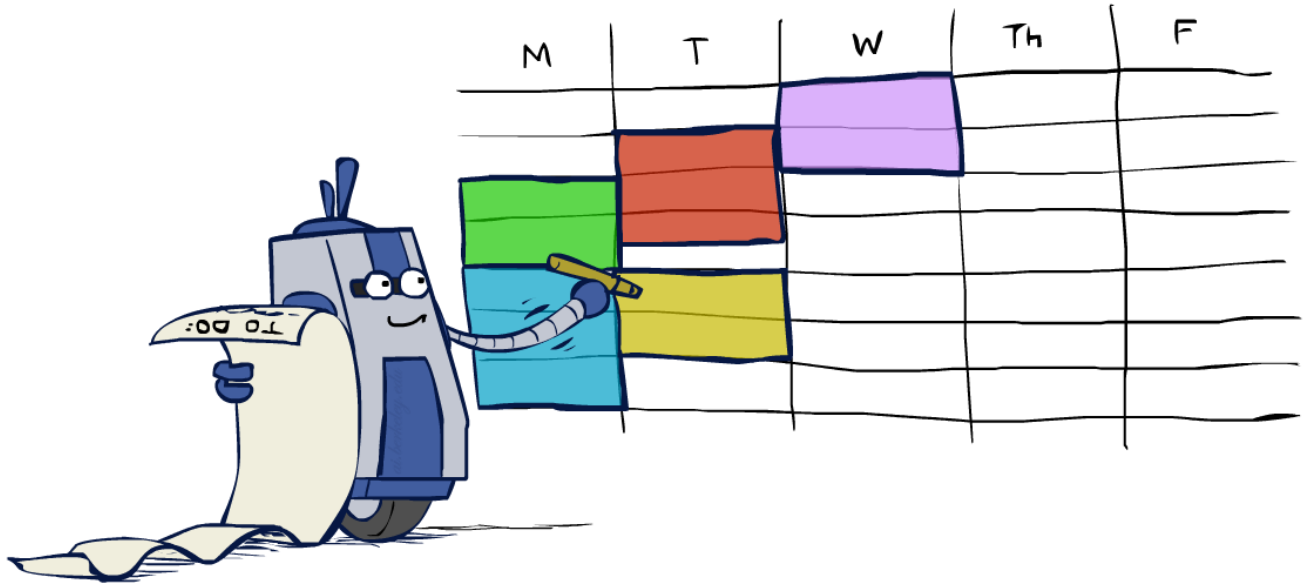


Preferences (soft constraints):

- 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)

Real-world CSPs

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



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

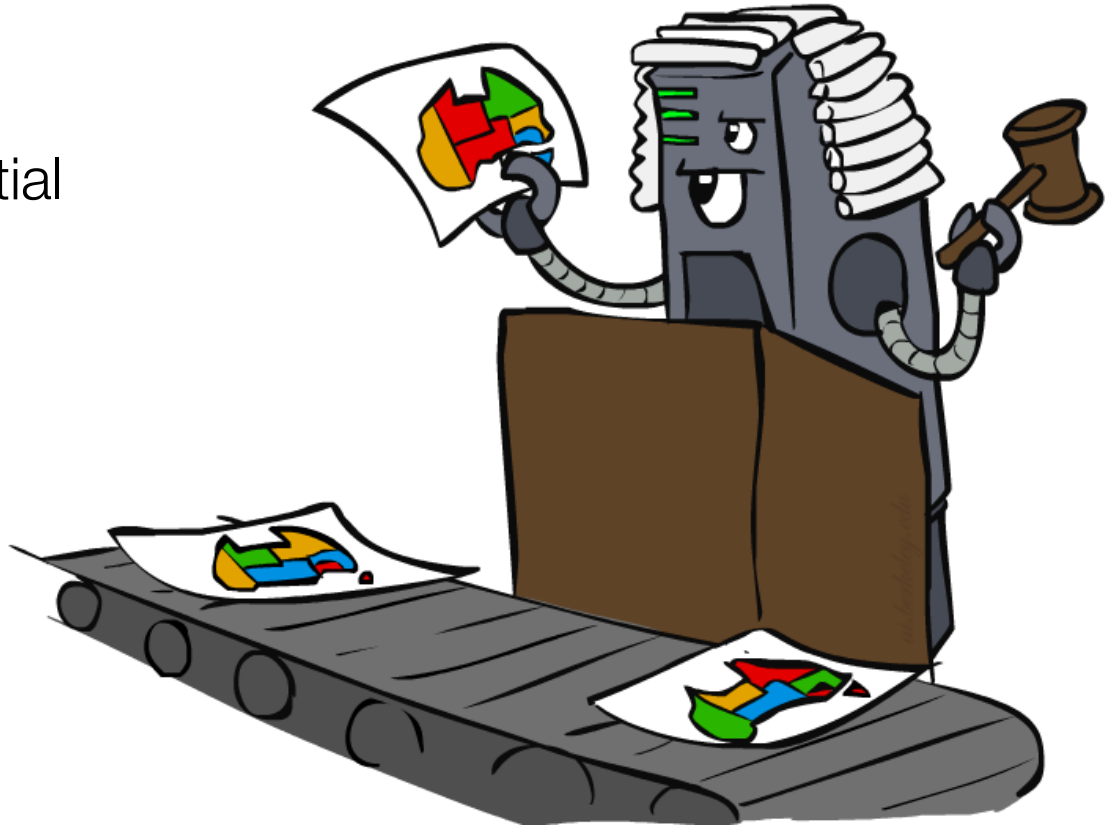
Great, but how do we solve these things?

Standard search formulation

Standard search formulation of CSPs

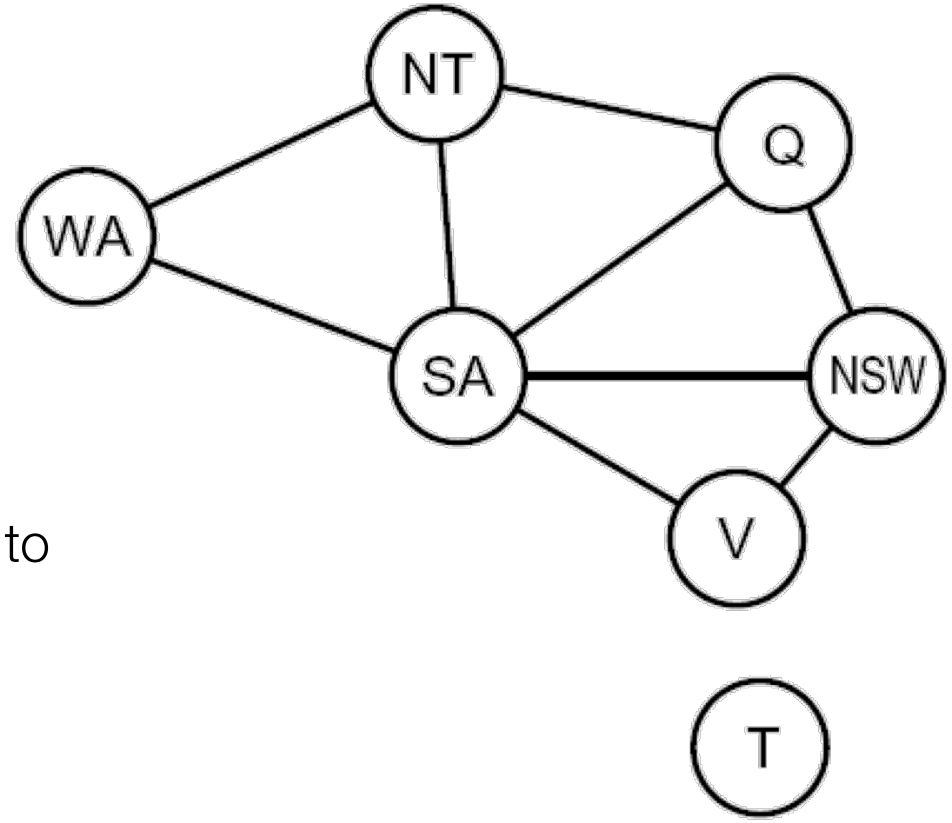
States defined by the values assigned so far (partial assignments)

- 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



In-class exercise

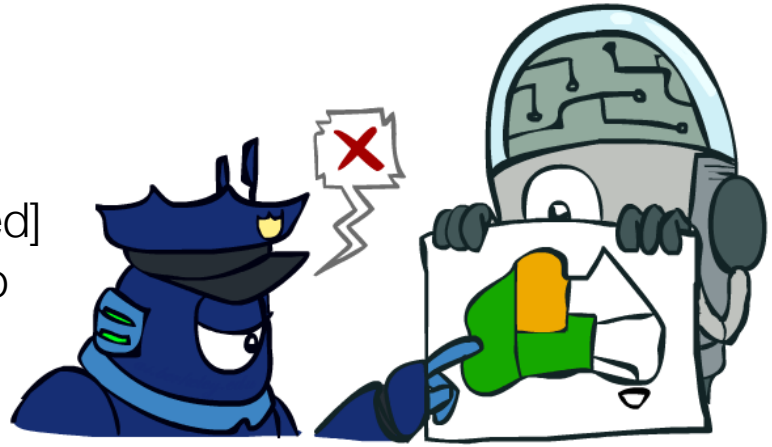
- Work on your own or in small groups
- Be sure to write everyone's name *legibly*!
- Oops! Hand-out should read "*successor function* to generate children is



How’d we do?

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

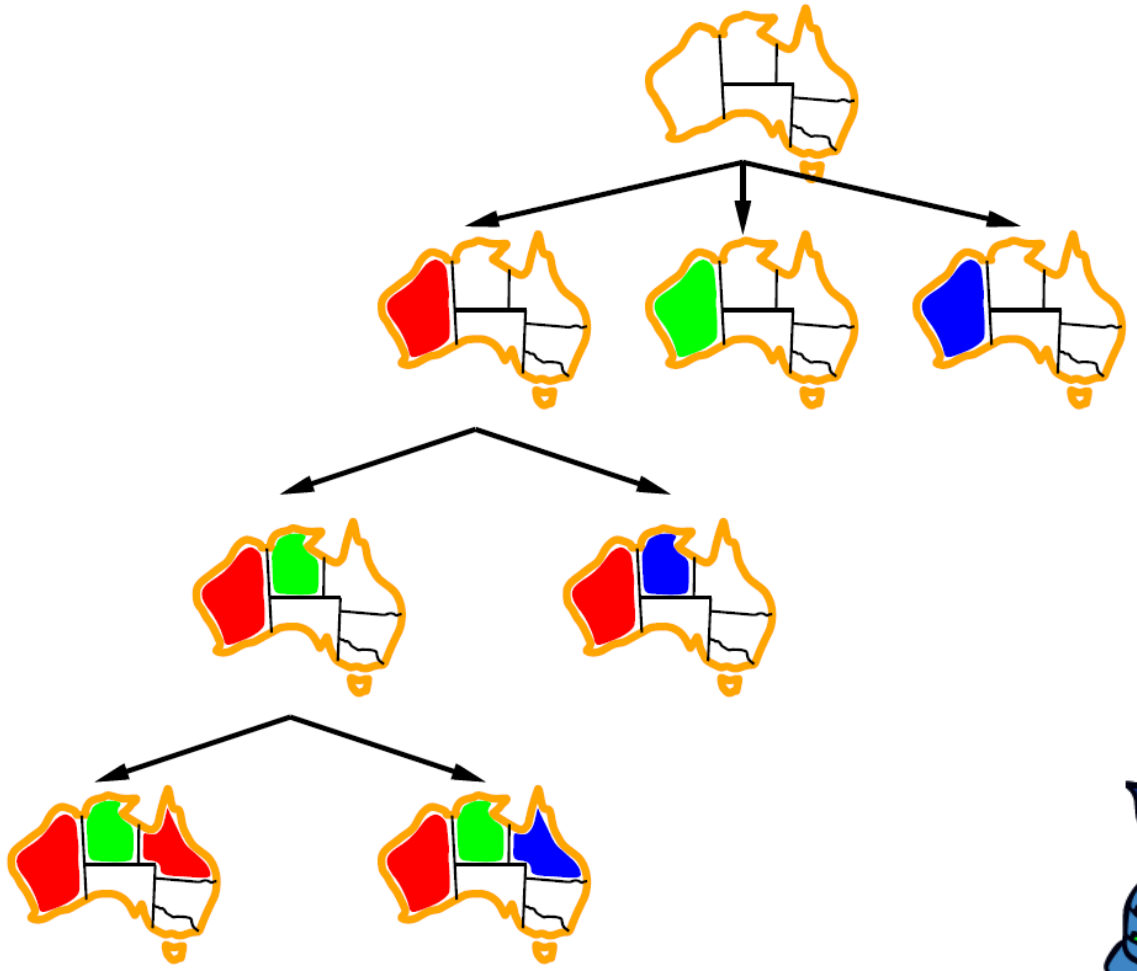
Idea 2: *Check constraints as you go*

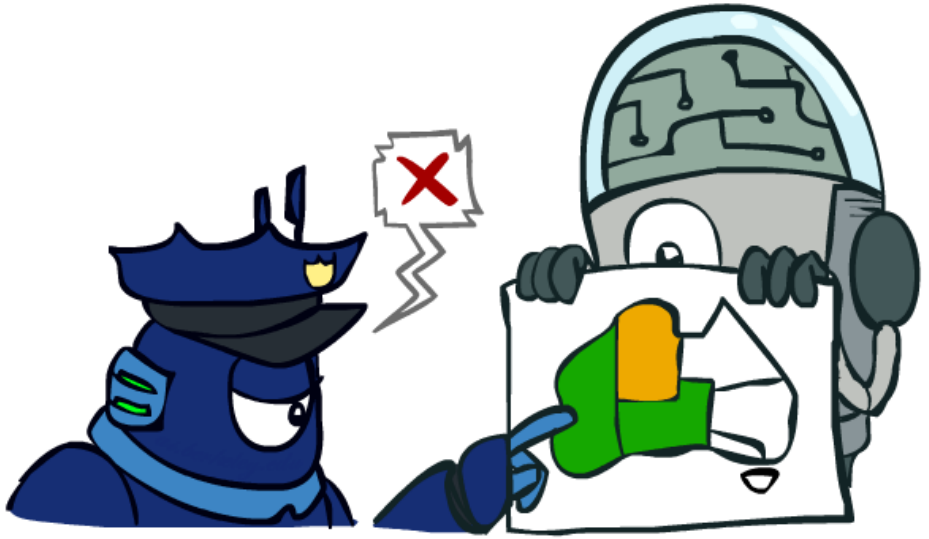
- I.e. consider only values which do not conflict previous assignments
- Might have to do some computation to check the constraints
- “Incremental goal test”

Depth-first search with these two improvements is called *backtracking search* (not the best name)

Can solve n-queens for n ≈ 25

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
```

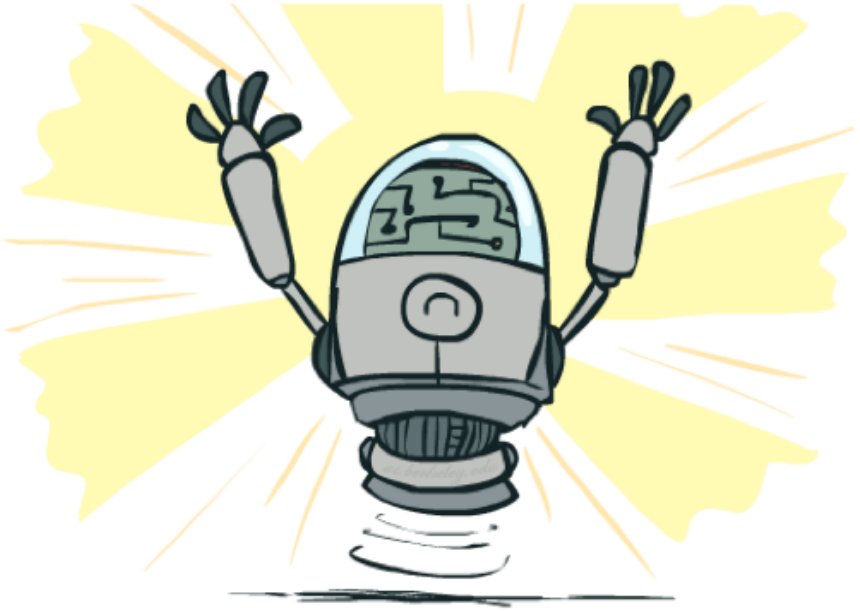
Backtracking = DFS + variable-ordering + fail-on-violation

Improving backtracking

Ordering:

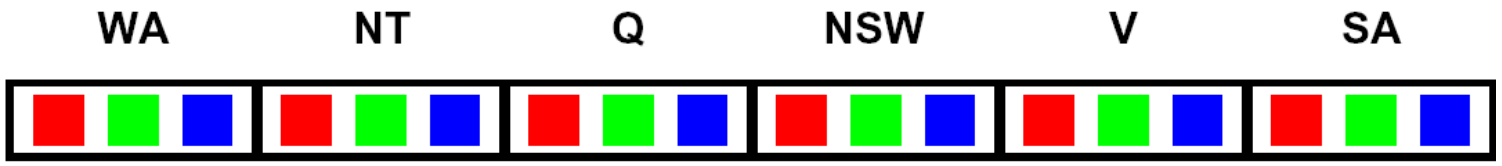
- Which variable should be assigned (expanded) next?
- In what order should its values be tried?

Filtering: Can we detect inevitable failure early?

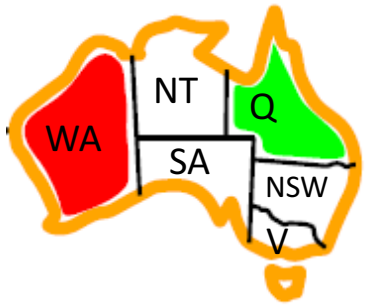


Filtering: forward checking

Filtering: Keep track of domains for unassigned variables and cross off bad option



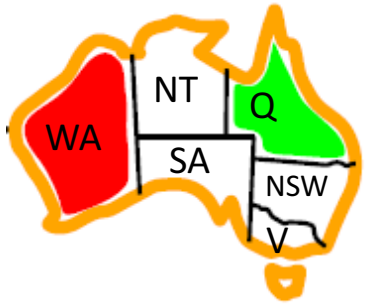
Forward checking

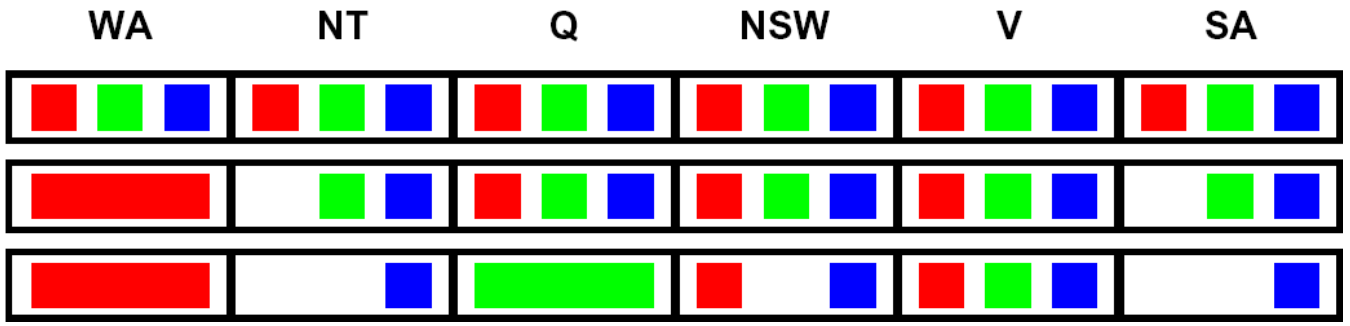


Assume we have assigned WA red. Let's think about the remaining vars.

Filtering: constraint propagation

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

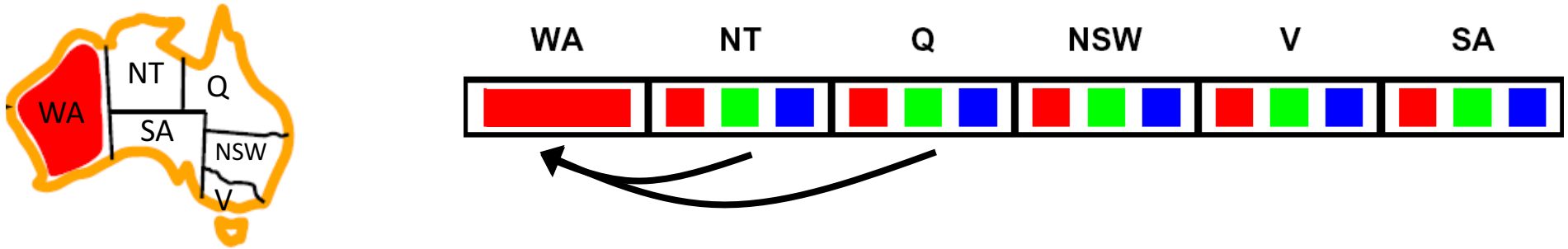




- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of a single arc

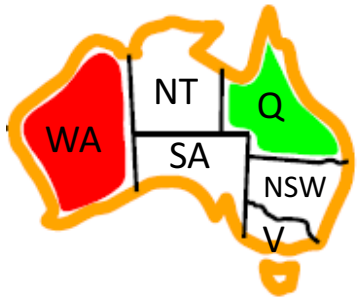
An arc X → Y is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

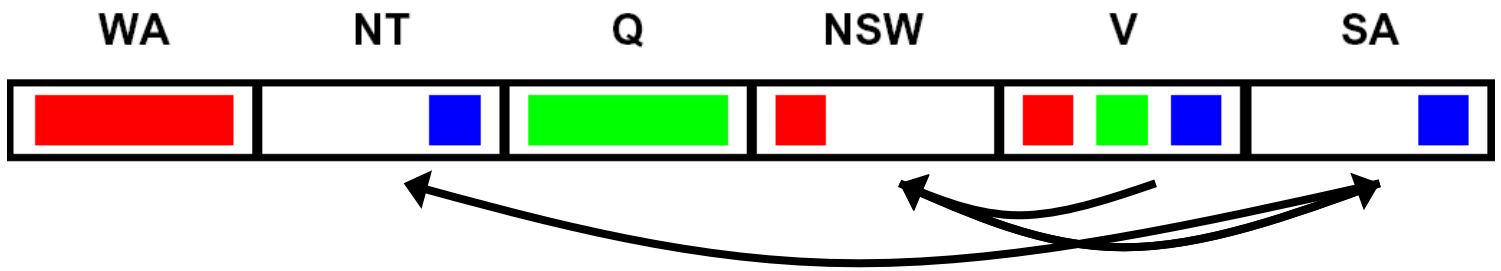


Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc consistency of an entire CSP

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





- **Important**: *If X loses a value, neighbors of X need to be rechecked*!
  - i.e., you need to reconsider all arcs going into!
- 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 $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*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*

Enforcing arc consistency in a CSP

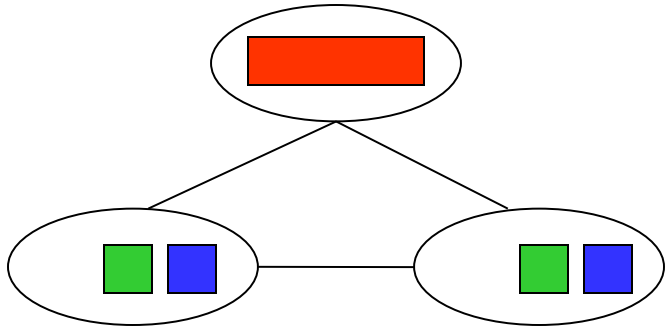
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(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 ← 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 ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

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

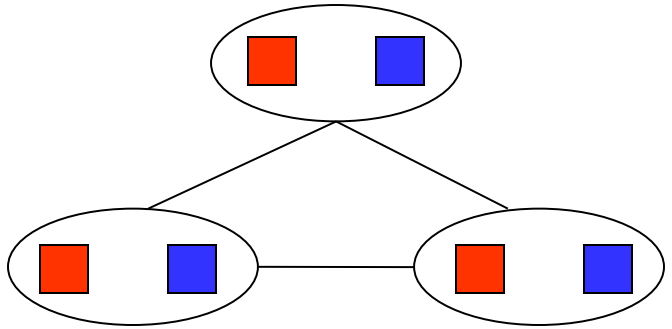
Limitations of arc consistency

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?

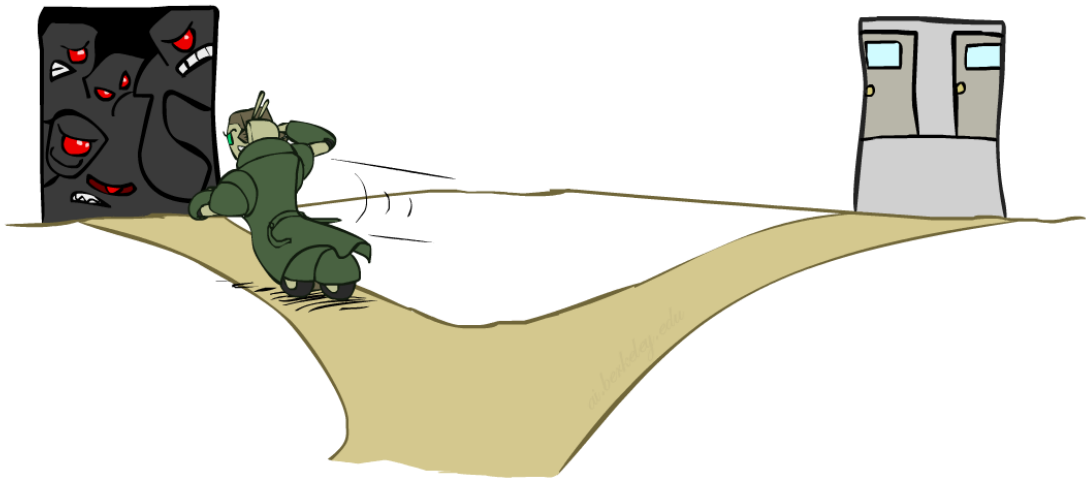
Ordering

Ordering: minimum remaining values

Variable Ordering: Minimum remaining values (MRV):

- Choose the variable with the fewest legal left values in its domain

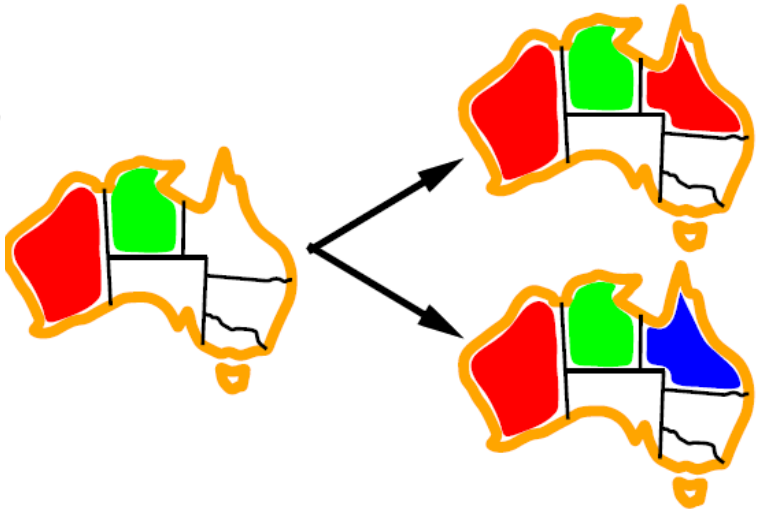
- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



Ordering: least constraining value

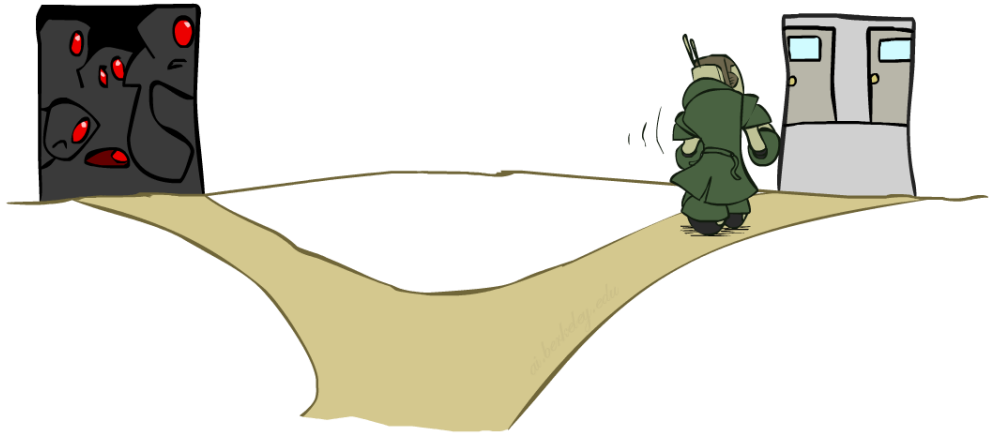
Value Ordering: *Least Constraining Value*

- 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?

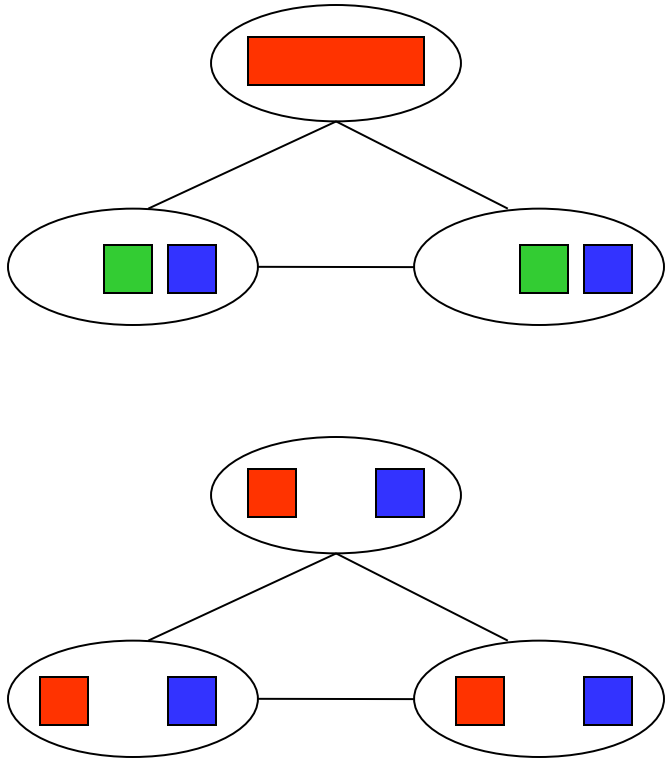
Combining these ordering ideas makes 1000 queens feasible



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)

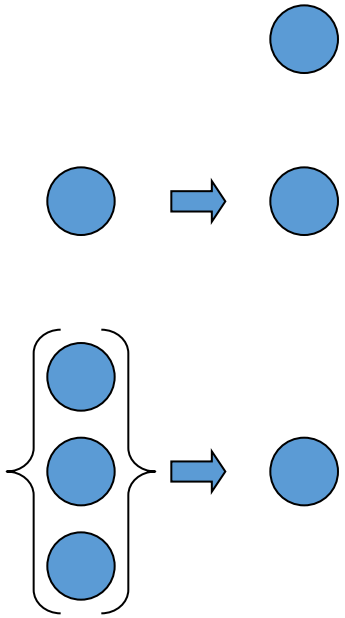


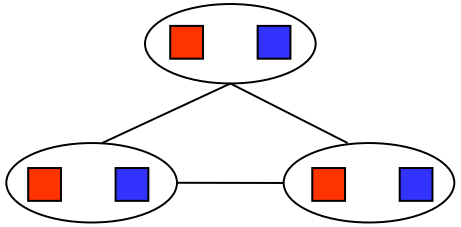
What went wrong here?

K-consistency: beyond pairs

Increasing degrees of consistency

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





Higher *k* more expensive to compute

(You need to know the *k*=2 case: arc consistency)

Q

Can a CSP be *k*-consistent but not *k*-1 consistent?

Strong *K*-consistency

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

Exploiting structure

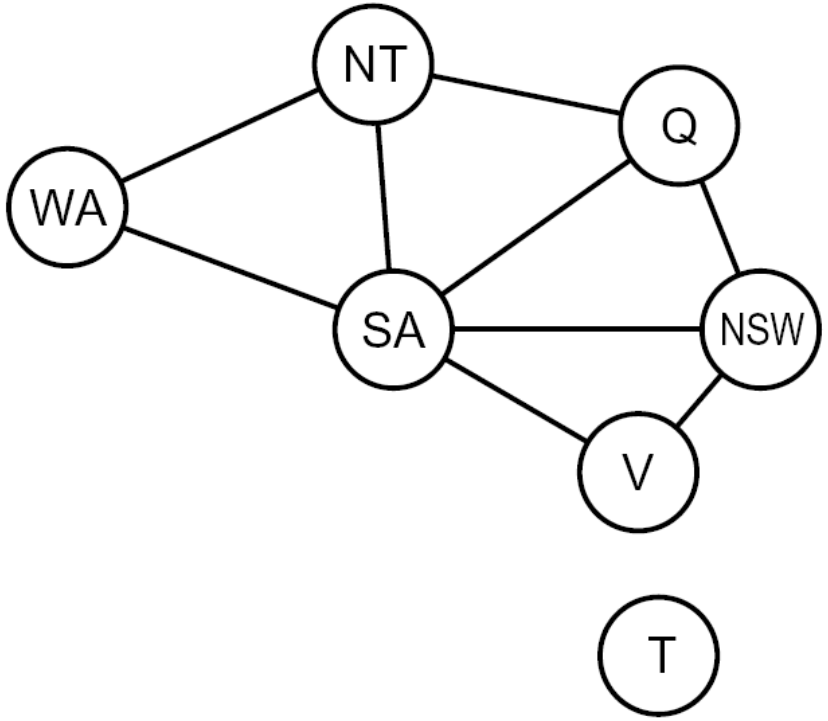
Extreme case: independent subproblems

- Example: Tasmania and mainland do not interact

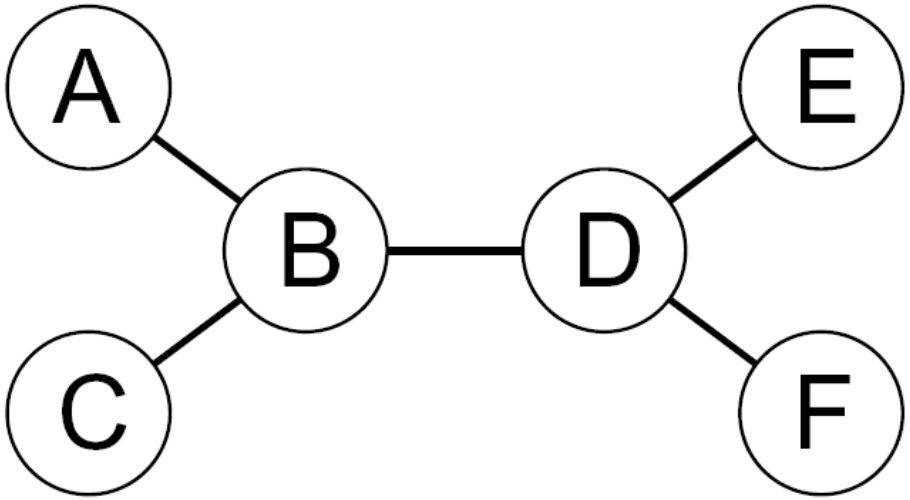
Independent subproblems are identifiable as connected components of constraint graph

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 (so 4 problems of size 20)
- 2^{80} = 4 billion years at 10 million nodes/sec
- $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



Tree-structured CSPs



Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time

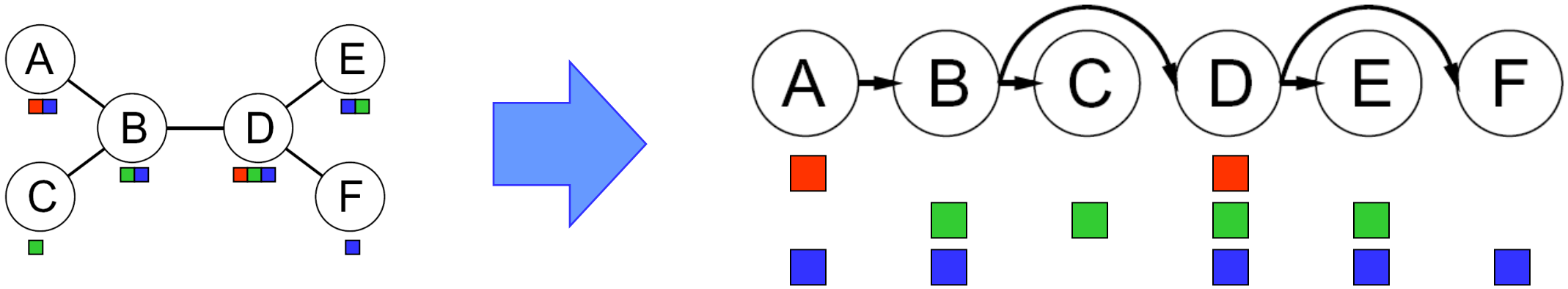
- Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-structured CSPs

Algorithm for tree-structured CSPs:

Order: Choose a root variable, order variables so that parents precede children



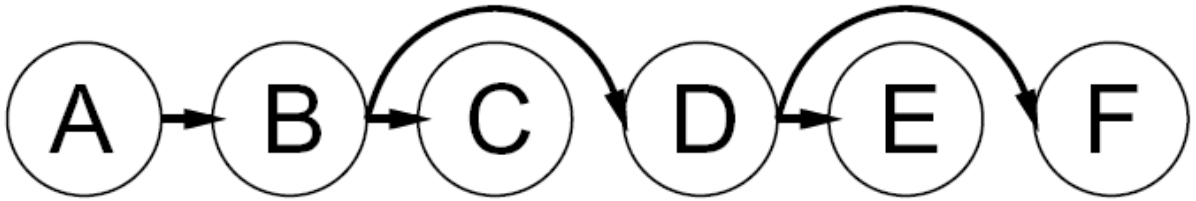
Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

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

Tree-structured CSPs

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

How does this rely on the tree structure again?

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

Summary: CSPs

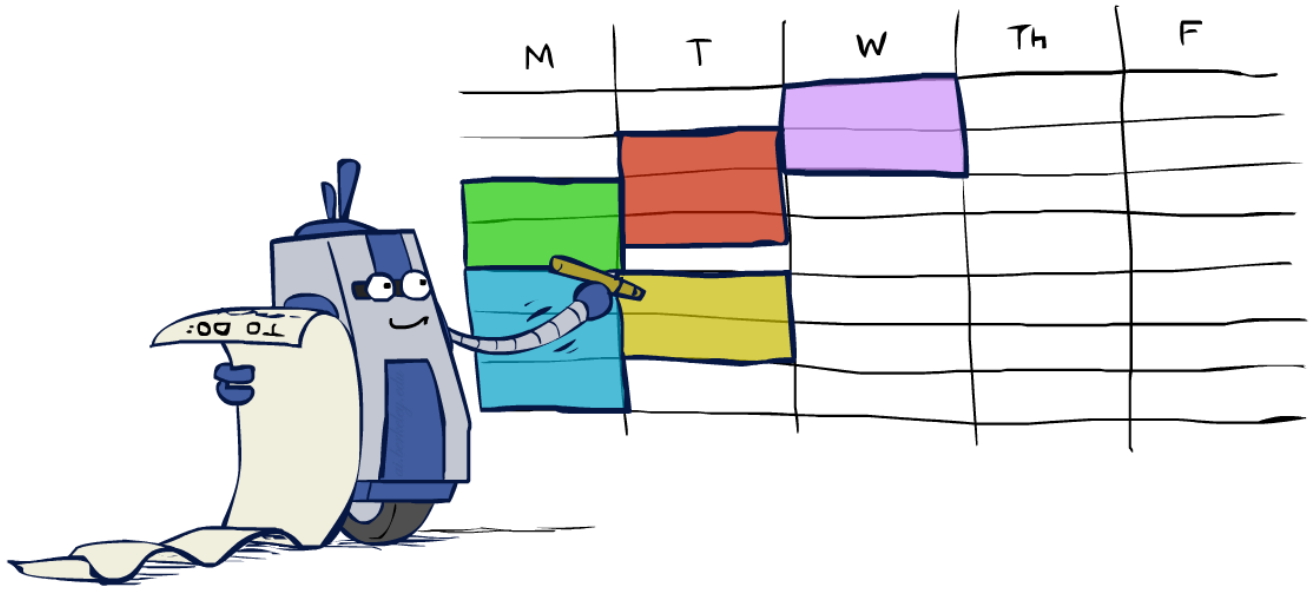
CSPs are a special kind of search problem:

- 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

That's it for today!

Next time: *adversarial search*