Artificial Intelligence

Chapter 5

Constraint Satisfaction Problems

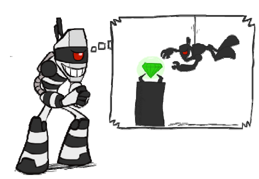
Updates and Additions: Dr. Siamak Sarmady

By: Dan Klein and Pieter Abbeel
University of California, Berkeley

What is Search For?

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

1. Planning: sequences of actions
   - The path to the goal is the important thing
   - Paths have various costs, depths
   - Heuristics give problem-specific guidance
   (measure of progress towards goal)

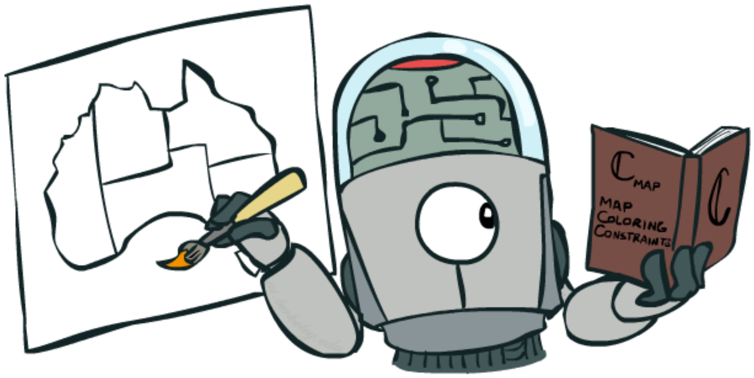


Goal State? How you do that (Planning)?

2. Identification: assignments to variables
   - Finding a well formed goal state
     Goal itself is important, not the path (8 queens, Coloring)
   - All paths are at the same depth (for **some** formulations)
   - CSPs are specialized for identification problems

Just wants to know where (in what state) it is
We are not interested in how to reach it

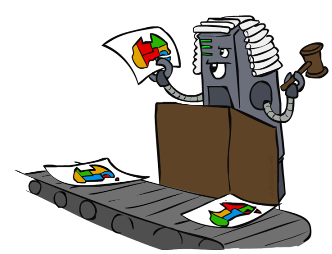
Constraint Satisfaction Problems



We will talk about map coloring problem… How to paint countries/states in a way that adjacent countries/states have different colors…

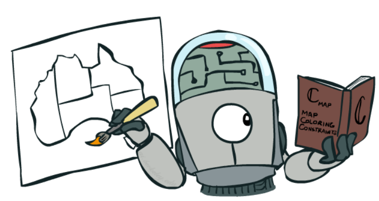
Constraint Satisfaction Problems

- Standard search problems:
  - **State:** might have arbitrary structure
    e.g.: where in Romania, x,y position on a board, positions of dots…
  - **Goal test:** a function over states, judges whether we are at a goal state
  - **Successor function:** takes states and actions, gives possible successor states (can have any mechanism i.e. a black box)



Check different possible states to see whether they are a Goal states or not

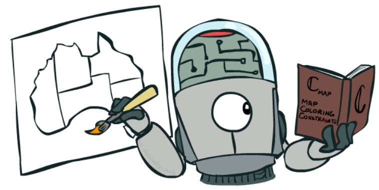
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - **State:** is defined by variables X_i with values from a domain D
    e.g. X_i: colors for states in Australia, D: RGB
  - **Domain:** Sometimes D depends on I (only some values allowable for specific variables, not all the possible values)
  - **Goal test:** is a set of constraints (rules) specifying allowable combinations of values for subsets of variables (e.g. not same colors for neighbor states)



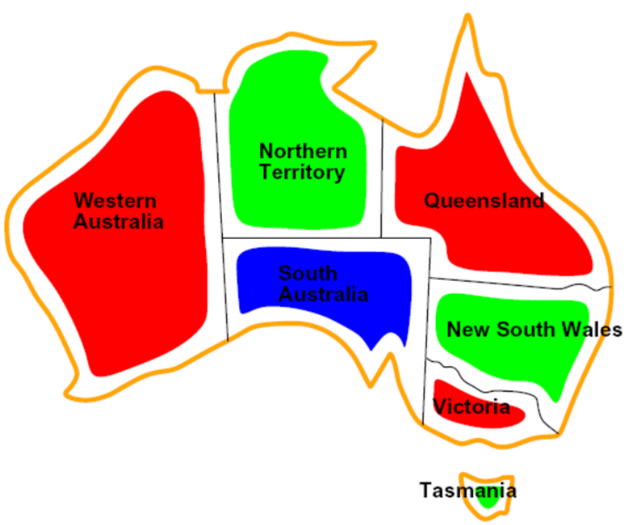
Now we just have a series of guides to judge whether a state is goal or not

Constraint Satisfaction Problems

- Simple example of a *formal representation language*
  - a language with which we can express the (separate) constraints
  - Now, instead of writing a goal test function (with code), we can express the constraints with a series of conditions
  - So we have a constraint representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms
  - Since the constraints can now be easily changed, the algorithms become more general purpose
  - If it was inside a code (goal test), modifying would be more difficult



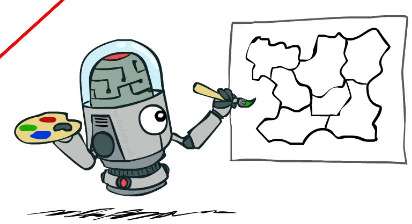
CSP Examples



Example: Map Coloring

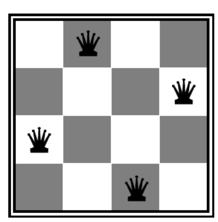
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: D= {red, green, blue}
- Constraints: adjacent regions must have different colors
  - Implicit: WA ≠ NT
    or even something like: if(a adjacent b) then color(a) ≠ color(b)
  - Explicit: (WA, NT) ϵ { (red, green), (red, blue), …}
- Solutions are assignments satisfying all constraints, e.g.:
  - {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

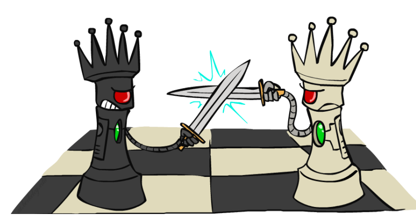
    A solution:
    1. Assigns a value to each variable
    2. satisfies every constraint



Example: N-Queens

- Formulation 1:
  - Have a variable for each cell telling whether it contains a queen
  - $\forall_{i,j}$, Variables: X_{ij} *(all cells)*
  - Domains: {0,1}
  - Constraints





The constraints are initially some rules ^^^ (implicit)

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

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

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

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

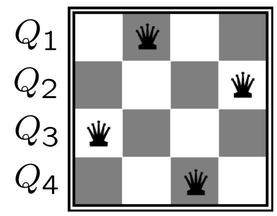
$$\sum_{i,j} X_{ij} = N$$

Also another constraint is that we actually have N queens on the board

Example: N-Queens

- Formulation 2:
  - **Variables:** Q_k
  - **Domains:** {1,2,3,…,N}
  - **Constraints:**
    - **Implicit:** $\forall$i,j non-threatening(Q_i,Q_j)
    - **Explicit:** (Q_1,Q_2) ∈ {(1,3),(1,4),…}

      …



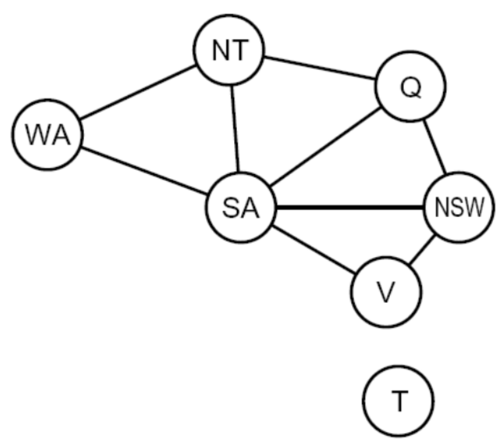
Now what if we include some knowledge of the problem:

- Since there should be only 1 queen in each row, we integrate that condition…
- More simple constraints
- We will have smaller search space

Constraint Graphs

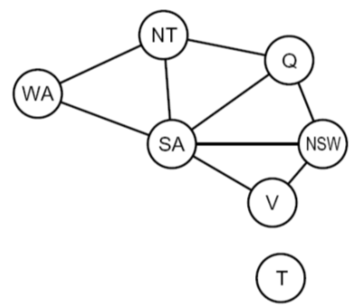
- If you have six variable constraints or seven variable constraints, If you want to show which variables have constraints between them, you may use such graphs…
- Each line shows a mutual constraint
- This graph is for Australia map coloring problem. How much does it make sense?

  Neighboring (adjacent) states have mutual constraints.



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 sub-problem!



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

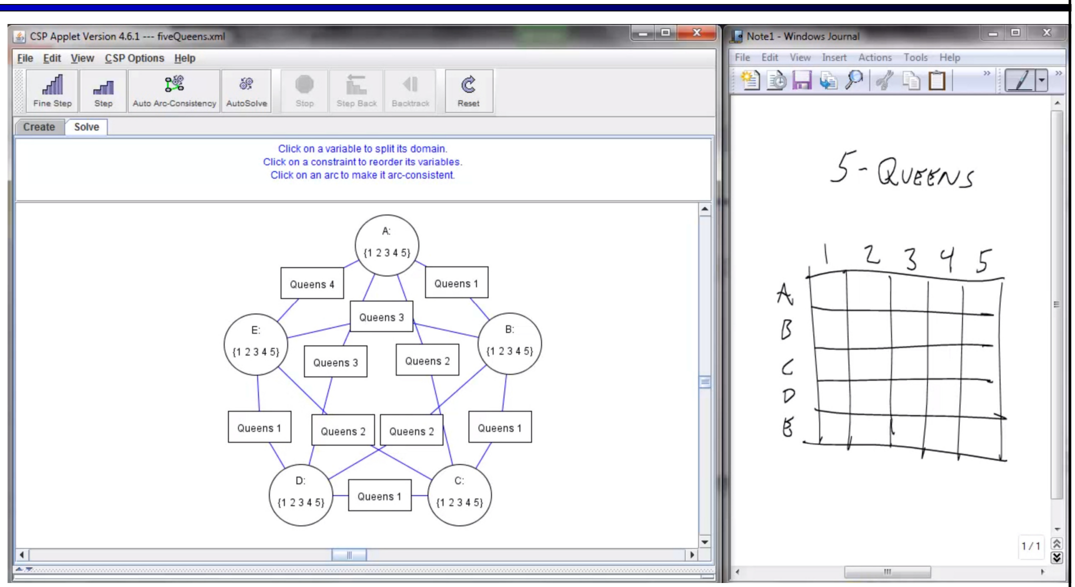
Screenshot of Demo N-Queens

A,B,C… are variables

They can have 1,2,3, …5 values which show the position of queen in that row.

The graph shows the constraints.

Constraint connects variables, and disallows them to have the same values…



Example: Cryptarithmetic

- Crypt-Arithmetic Problems are substitution problems where digits representing a mathematical operation are replaced by unique digits.
  Like :

  P L A Y S + W E L L = B E T T E R
  Where each unique alphabet represents a unique digit from among 0 to 9.

- So, if the solution to this puzzle is to be found, it would be (after a long computation) : **9 7 4 2 6 + 8 0 7 7 = 1 0 5 5 0 3**

- The basic rules are :
  - Each unique digit must be replaced by a unique character.
  - The number so formed cannot start with a ZERO.

See https://en.wikipedia.org/wiki/Verbal_arithmetic for more

Example: Cryptarithmetic

- Variables:

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

- Domains:

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

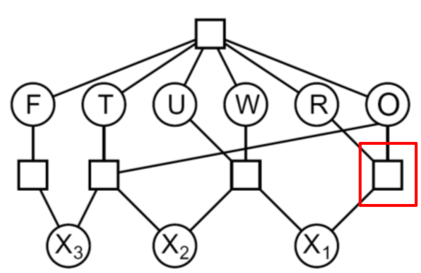
- Constraints:

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

  $$\boxed{O + O = R + 10 \cdot X_1}$$

  $$\cdots$$

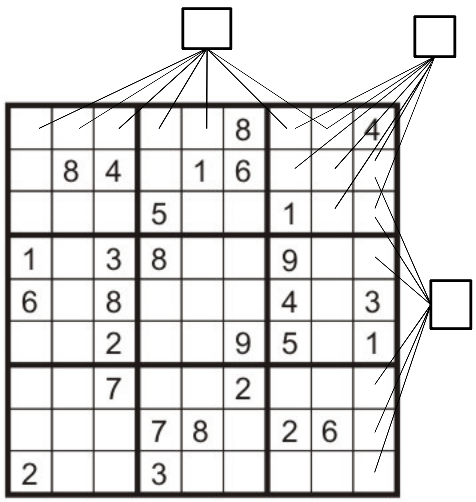
```
X3 X2 X1
   T W O
+  T W O
 -------
 F O U R
```



Solution: O=7, R=4, W=6, U=2, T=8, F=1; 867 + 867 = 1734

Constraints are no more binary: O-R-X1

Example: Sudoku



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

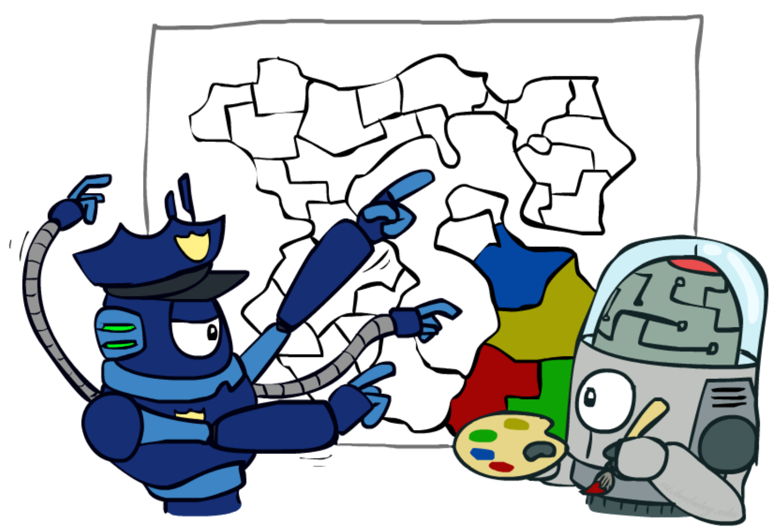
9-way all different for each column

9-way all different for each row

9-way all different for each region

(or can have a bunch of pairwise inequality constraints)

Varieties of CSPs and Constraints



Varieties of Constraints

- Types 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.:

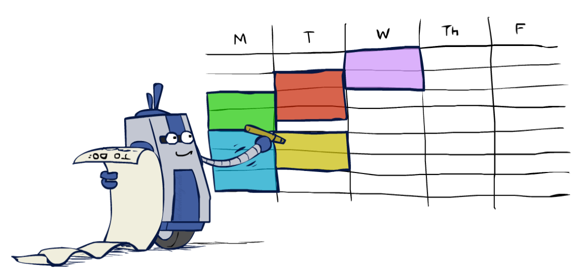
    $$SA \neq WA$$

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

- Preferences (soft constraints):
  - E.g., red is better than green (use red more than green as much as possible? For whatever reason…)
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - We'll ignore these for now

Real-World CSPs

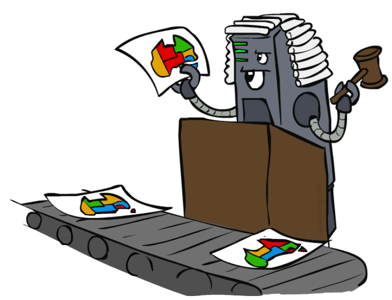
- Assignment problems: e.g., who teaches what class (one lecturer's classes cannot be in the same time)
- Timetabling problems: e.g., which class is offered when and where? (no two classes in the same place and time)
- Hardware configuration (parts will fit and work together?)
- Transportation scheduling (ticket, times, stays)
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



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

Solving CSPs

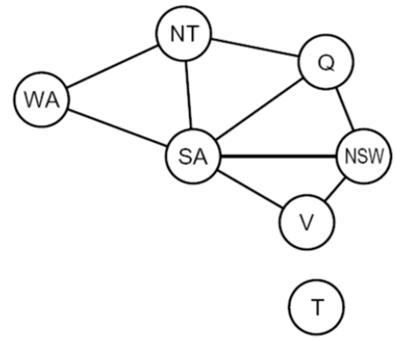
Standard Search Formulation

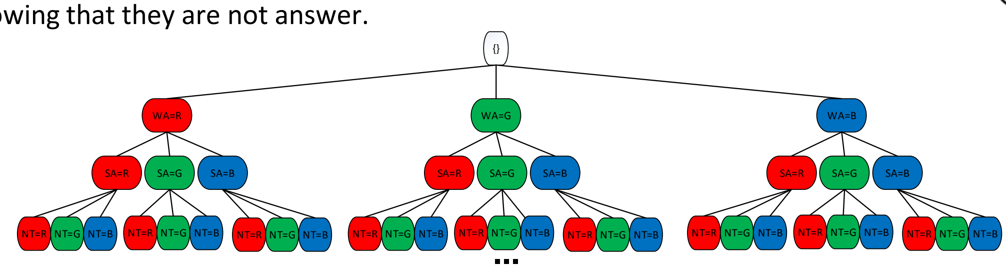


- Standard search formulation of CSPs
  - We first start by threating CSP as a normal search problem with some considerations
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {} i.e. no value is assigned to any of the variables.
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment (values for vars.) is complete and satisfies all constraints
- We'll start with the straightforward, naïve (i.e. uninformed) approach, then improve it

Search Methods

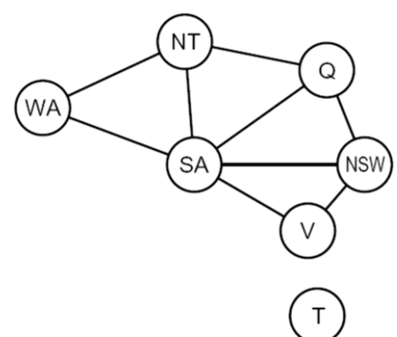
- What would BFS do?
  - Remember that each var. can take red, green and blue
  - Create trees of possible assignments to each var, for example starting from WA
  - In 1st depth we assign values to only one var, in 2nd depth to two vars only, only in d_{th} depth all vars have values.
  - We will go down and down until we have values for all variables (i.e. all in the same d_{th} depth) and that is bad news for BFS because you are checking shallow levels knowing that they are not answer.

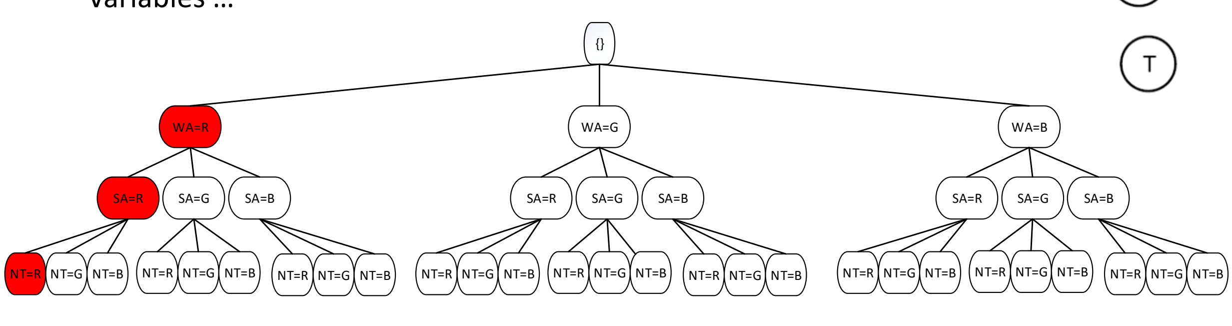




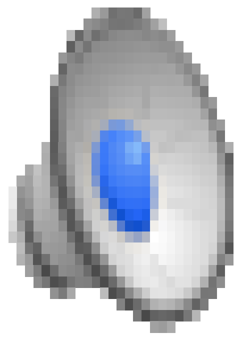
Search Methods

- What would DFS do?
  - Will go to depth (i.e. one complete assignment) and then will check other assignments. In the coloring example the first will possibly assign red, red, red… to every variable.
  - Then it goes back and changes value for one of the variables …



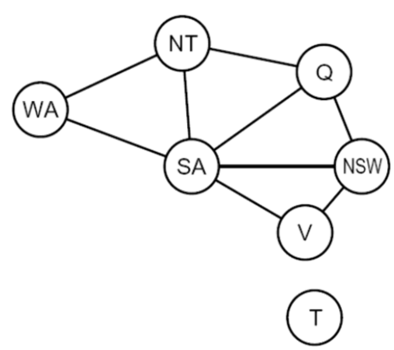


Video of Demo Coloring -- DFS



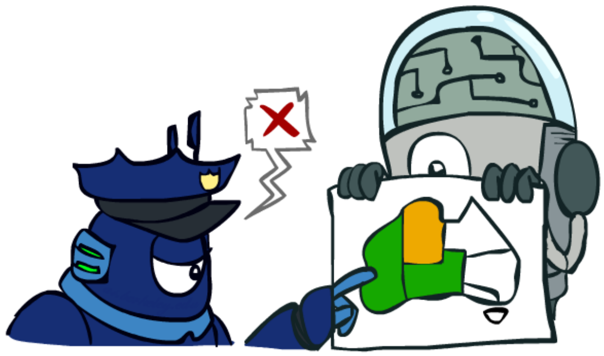
Search Methods

- What problems does naïve search have?
  - By searching all options we are disregarding constraints we had set…
  - As we saw it tries combinations that obviously cannot be the answer (e.g. blue, blue, …)



[Demo: coloring -- dfs]

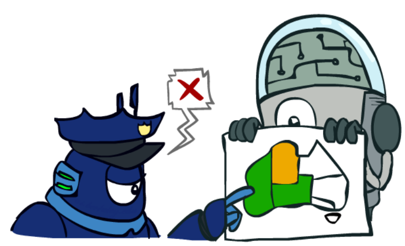
Backtracking Search



- **The idea:** in addition to "enumerating successors" and "goal check" we also check whether we have broken a constraint… since constraints are separate, we can check them separately
- Backtracking incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *it* cannot possibly be completed to a valid solution

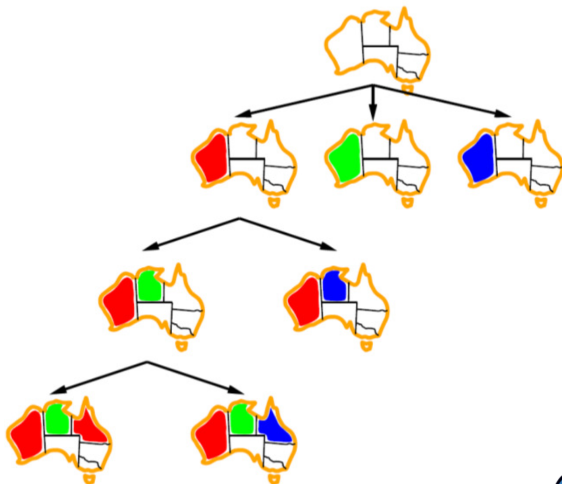
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- **Idea 1:** One variable at a time (instead of all at once and checking then)
  - Only need to consider assignments to a single variable at each step
  - Variable assignments are commutative (order of assignment is not important)
    i.e. [WA = red then NT = green] same as [NT = green then WA = red], don't repeat (by fixing the order)
- **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", as soon as it breaks const. it is not goal.
- Depth-first search with these two improvements is called ***backtracking search*** (not the best name)
  You can implement without actual backtracking
- Can solve n-queens for n ≈ 25



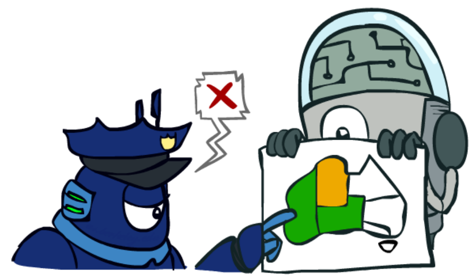
Backtracking Example

Note that it assigns values one at a time i.e. in first layer we only assign values to one variable instead of all



Just like depth first but if a node breaks the law (constraint rules) it is removed (not added to fringe)…

That is the cop came and said you are out!



Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
         add {var = value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value} from assignment
   return failure
```

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

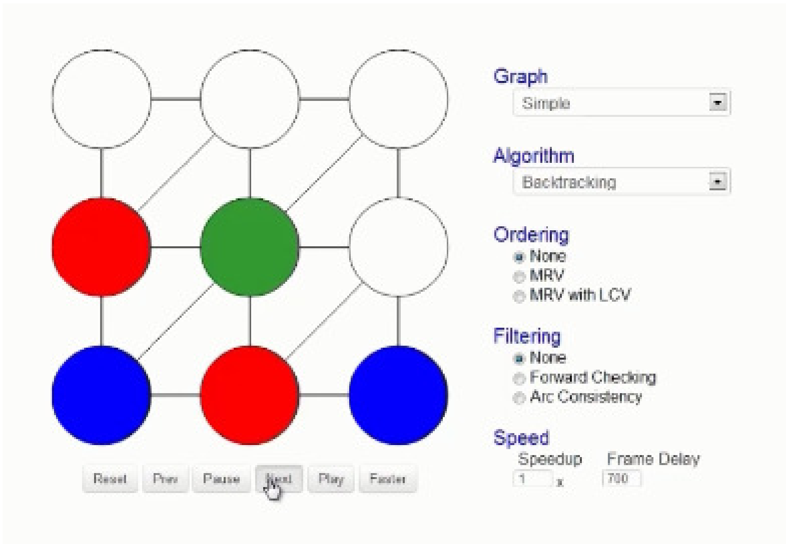
- Recursive impl. but can do it in non-recursive methods…
- Start with empty assignment

- If managed to finish all return.
- Peek an unassigned var.
- Loop & Consider a new value
- If it does not break constraints
- add to assignment
- Recurse & If fine(call can find consistent values) , return result
- Else remove last assignment and return failure

- If cannot find anything after all just return failure

[Demo: coloring -- backtracking]

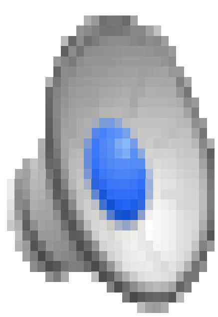
Video of Demo Coloring – Backtracking



- At first assigns blue
- In next step, it won't do ~~blue~~ since it breaks the constraint

- Then it selects red
- Then it selects blue again. Is that good? We don't know… otherwise we would not use CSP search!
- Then red again and then green…(cannot ~~blue~~ and ~~red~~)
- Now we don't have any color for the 6th place since all 3 colors will break the constraint… so now we need to backtrack
- And with just a bit of backtracking we can finish…
- Note: we backtrack early and don't go deep to backtrack

Video of Demo Coloring – Backtracking



- At first assigns blue
- In next step, it won't do ~~blue~~ since it breaks the constraint

- Then it selects red
- Then it selects blue again. Is that good? We don't know… otherwise we would not use CSP search!
- Then red again and then green…(cannot ~~blue~~ and ~~red~~)
- Now we don't have any color for the 6th place since all 3 colors will break the constraint… so now we need to backtrack
- And with just a bit of backtracking we can finish…
- Note: we backtrack early and don't go deep to backtrack

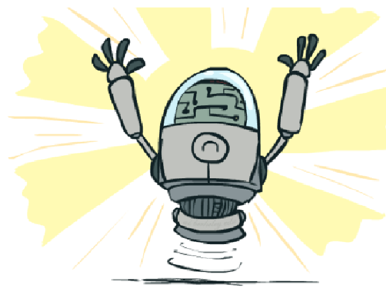
Improving Backtracking

- General-purpose ideas give huge gains in speed
  - unlike heuristic which is problem dependent

- Filtering: Can we detect inevitable failure early?
  - Without actually doing the rest of the CSP?

- Ordering:
  - Which variable (part of graph) should be assigned next?
  - What value and In what order should the values be tried?

- Structure:
  - Can we exploit the problem structure?



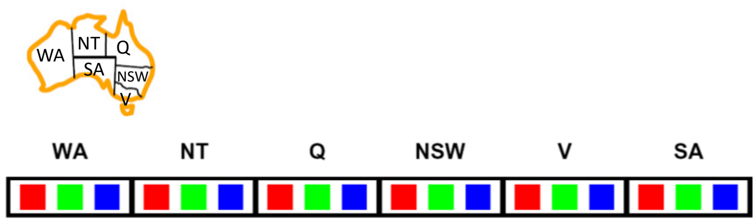
Filtering

Filtering is about ruling out suspects …and focusing on the more likely candidates

Note that we still run backtracking search, but with new ideas…

Filtering: Forward Checking

- **Filtering:** As we recurse in the code, we maintain a data structure that keeps track of value assignments and crosses off bad options from the domains (possible values) of unassigned variables
- **Forward checking:** Cross off values that violate a constraint when added to the existing assignment



Now we backtrack less, but instead we need to spend time to forward check and prepare a list of possible/impossible answers…

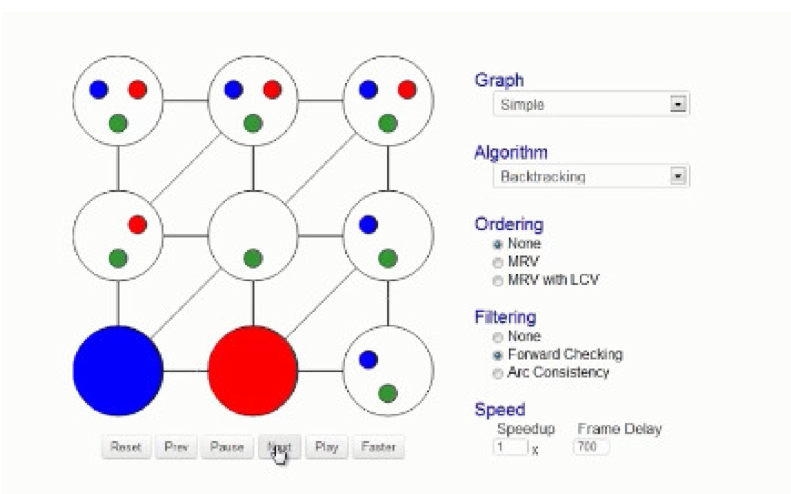
It is like A*, you do more calculations for each node, but then you have less nodes to search…

Still cannot backtrack, everyone has a legal move

Now we have a cell with no options so we backtrack

Notice: as you backtrack(undo an assignment), you put the domain (possible options) back with the same order..

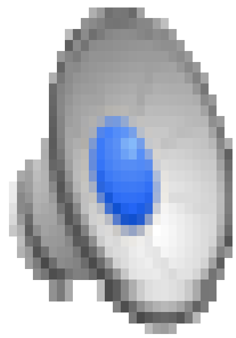
Video of Demo Coloring – Backtracking with Forward Checking



- When filtering is enabled, we will keep track of possible values of each cell (domain).

- As we assign blue, the possible values of neighboring cells are filtered.

Video of Demo Coloring – Backtracking with Forward Checking

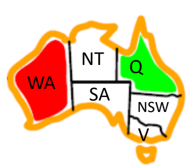


- When filtering is enabled, we will keep track of possible values of each cell (domain).

- As we assign blue, the possible values of neighboring cells are filtered.

Filtering: Constraint Propagation

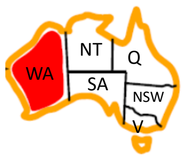
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

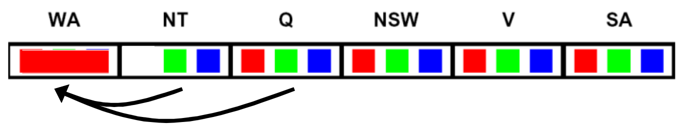


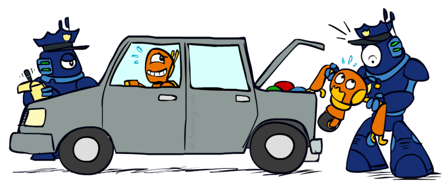
- NT and SA cannot both be blue!
- Why didn't we detect this yet? Filtering (forward checking) cannot check the interactions between unassigned variables
- Filtering cannot do it but ***constraint propagation*** *can:* reason from constraint to constraint
- It need more work on each node, but it allows to backtrack even earlier

Consistency of A Single Arc

- **Checking single Arcs:** An arc X → Y is consistent iff for *every* x (value) in the tail there is *some* y in the head which could be assigned without violating a constraint (otherwise to make consistent, remove x from tail)







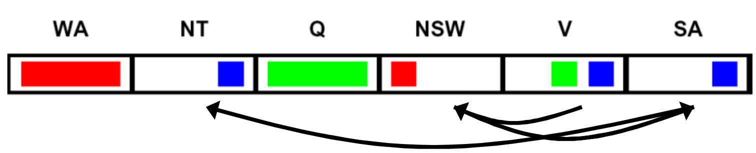
Delete from the tail!

1. For every value in tail, check whether there is a value in head that is ok
2. Is the Arc consistent?
3. If no, make it consistent (by removing from tail)

- **Forward checking (filtering):** Is in fact, enforcing consistency of arcs pointing to only new assignment
- We are free to enhance and check every Arc though …

Arc Consistency of an Entire CSP

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



- **Important:** If X loses a value, neighbors of X need to be rechecked! (worse case, recheck all)
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or (usually) after each assignment
- What's the downside of enforcing arc consistency?
  Now it takes a long time (check every arc) to do each step

Remember: Delete from the tail!

If empty domain, backtrack

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
```

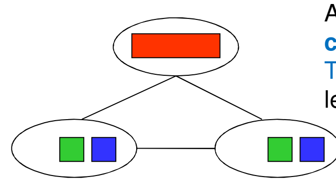
- Called arc consistency
- Input is a CSP
- Build a queue of suspect arcs (all initially)
- If there is any suspect
- Remove from queue and
- Check (and remove inconsistencies)
- And add all neighbors back to the queue for recheck if any inconsistent value was removed

- Looks at each value x in tail
- If no value y in head that allows consistency
- Remove the value x from tail

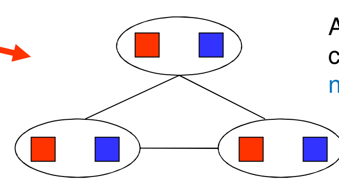
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
  - For applying consistency (by removing inconsistent) takes d^2 (d: number of values in domains)
  - Upper section takes n^2 in best case, might be repeated d times (if values are rechecked), that's n^2d
  - So best case: n^2d^2, worse case: n^2d^3
- … but detecting all possible future problems is NP-hard – why?

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have no solution left
  - Can have one solution left
  - Can have multiple solutions left
  - Also might have no solutions left but in a way that is not obvious
- Arc consistency still runs inside a backtracking search!



Arcs **consistent**, Two solutions left



Arcs are consistent but no solution

What went wrong here?

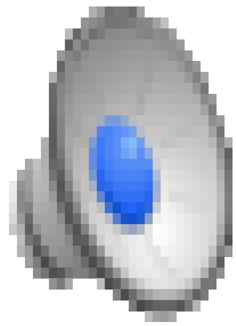
Cannot track **three way interactions**

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

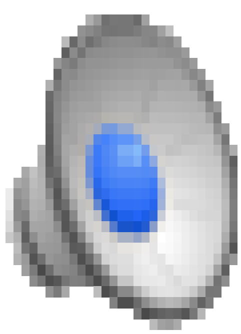
Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph

Forward checking:

We see only green left for two variables, but we still continue our search and do not back track



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



- As soon as we assign 2nd value, all arcs are checked and inconsistent values are removed and we are quickly done.

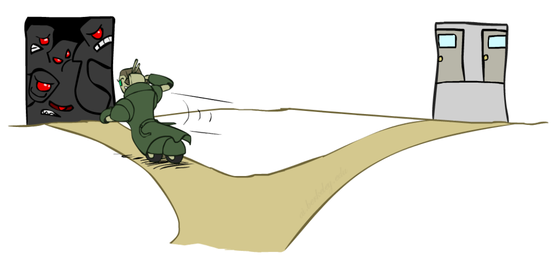
Ordering

Ordering: Minimum Remaining Values

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

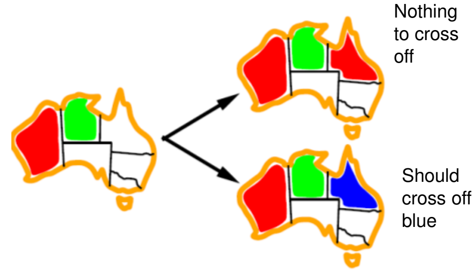
- Also called "most constrained variable"
- Why min rather than max?
  - "Fail-fast" ordering

    Do hard things first, if you are going to fail, you fail earlier and backtrack earlier



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?
  We want to save our options as much as possible

- Combining these ordering ideas makes 1000 queens feasible