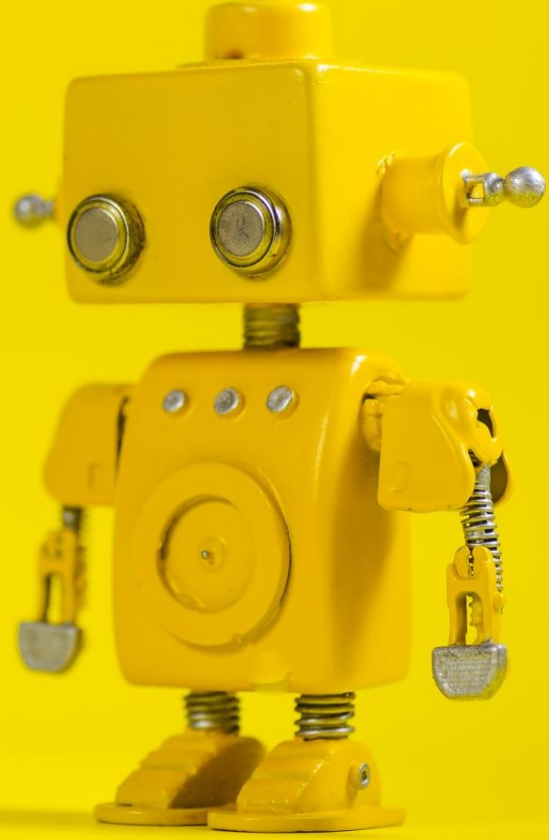


Lecture 05: CSP and Local Search

CS 445 – Artificial Intelligence Spring 2022

Instructor: Dr. Trung T. Nguyen

Agenda

- Lecture 05 – Constrain Satisfaction Problems (CSPs)
  - CSP definition and examples
  - Backtracking search for CSPs
  - Problem structure and problem decomposition
  - Local search for CSPs

- Recommended readings on search:
  – AIMA Ch 6.1-6.4 (required)
  – AIMA Ch 4.1 (required)

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 specialized for identification problems

Credit: ai.berkeley.edu

Constraint Satisfaction Problems (CSPs)

Standard search problem:
- state is a "black box" ----- arbitrary data structure
  - that supports goal test, eval, successor

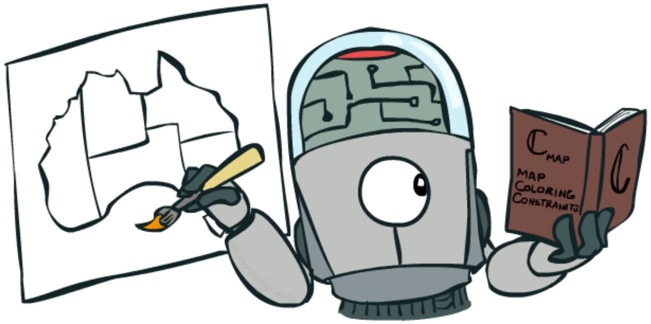
CSP:
- a special subset of search problems
- state is defined by variables X_i with values from domain D_i
- goal test is a set of constraints specifying allowable
  - combinations of values for subsets of variables

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

N variables

domain D

constraints



states

partial assignment

goal test

complete; satisfies constraints

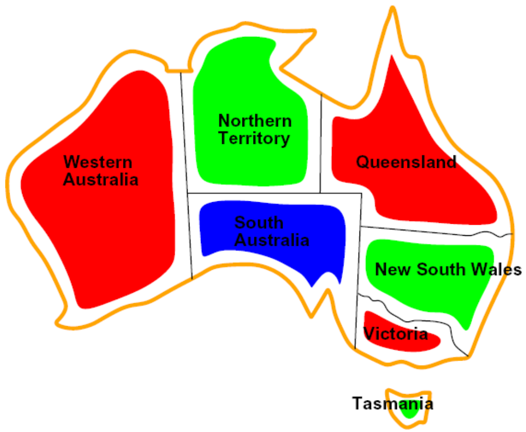
successor function

assign an unassigned variable

Credit: ai.berkeley.edu

Example: Map Coloring

- Variables: WA, NT, Q, NSW, V, SA, T

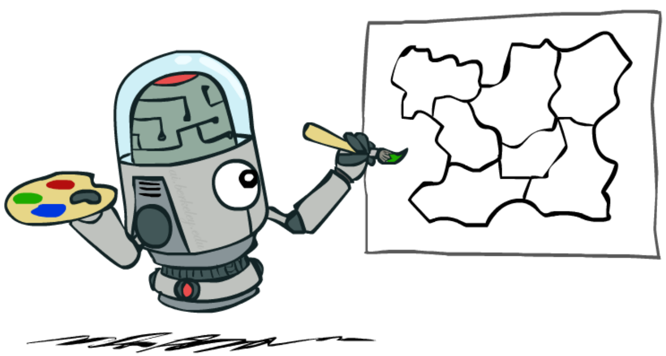


- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$

- Constraints: adjacent regions must have different colors

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

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



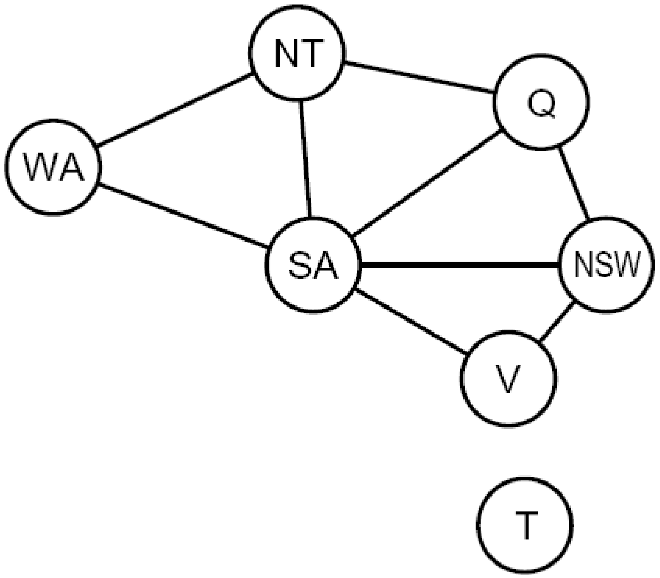
- Solutions are assignments satisfying all constraints, e.g.:

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

Credit: ai.berkeley.edu

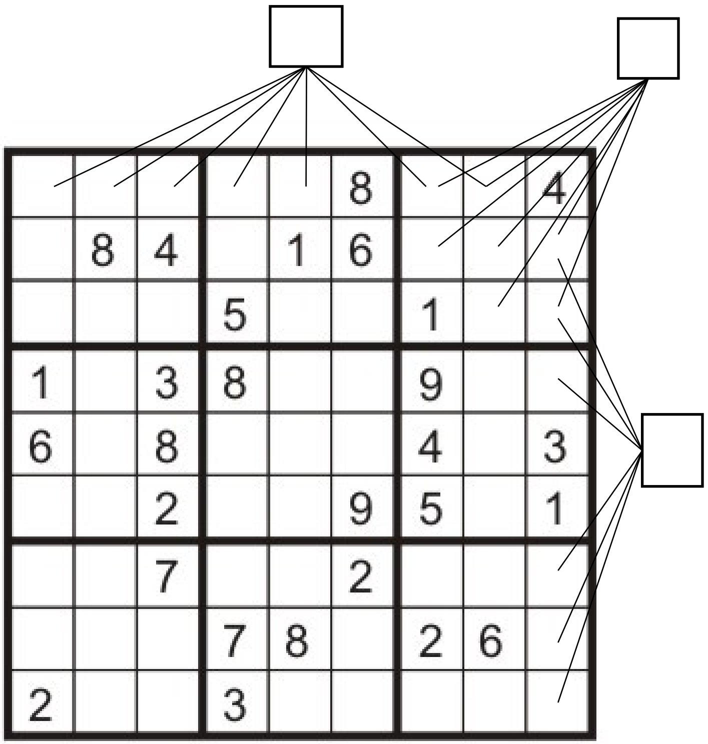
Constraint Graphs

- 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!



Credit: ai.berkeley.edu

Example: Sudoku



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

9-way alldiff for each column

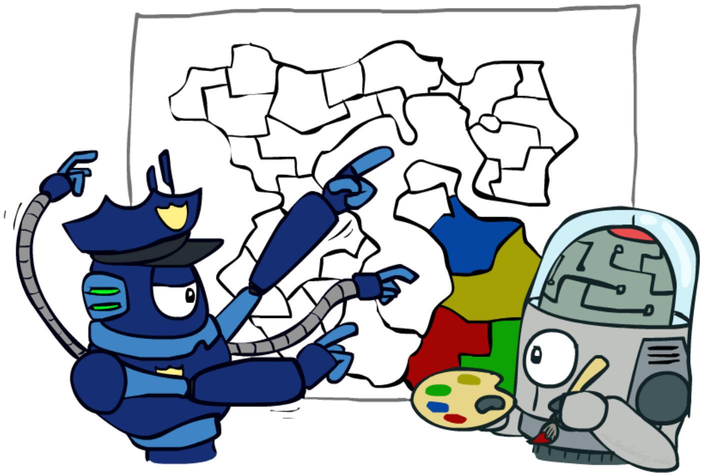
9-way alldiff for each row

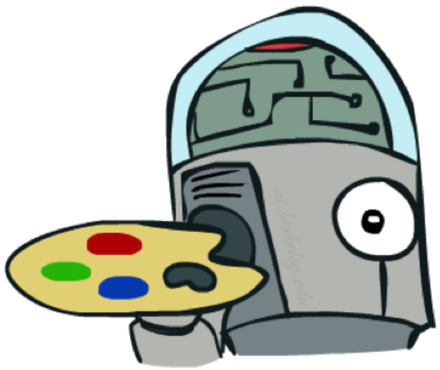
9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Varieties of CSPs and Constraints

- 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





Varieties of CSPs and Constraints

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

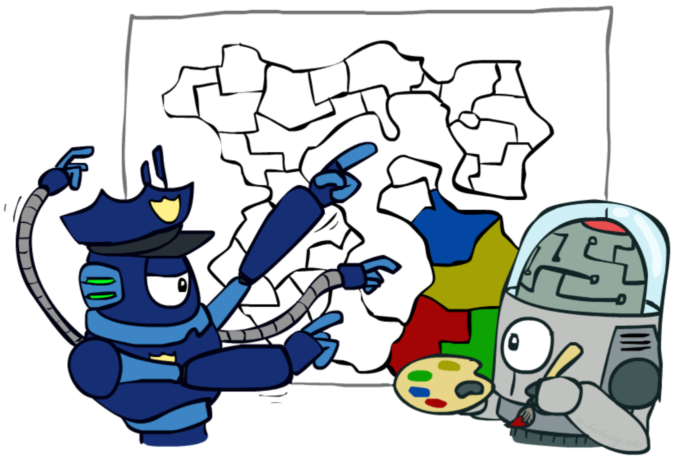
    $$SA \neq green$$

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

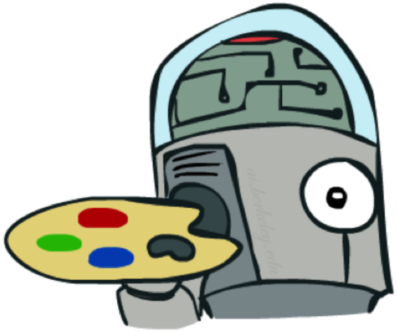
    $$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:

    e.g., cryptarithmetic column constraints

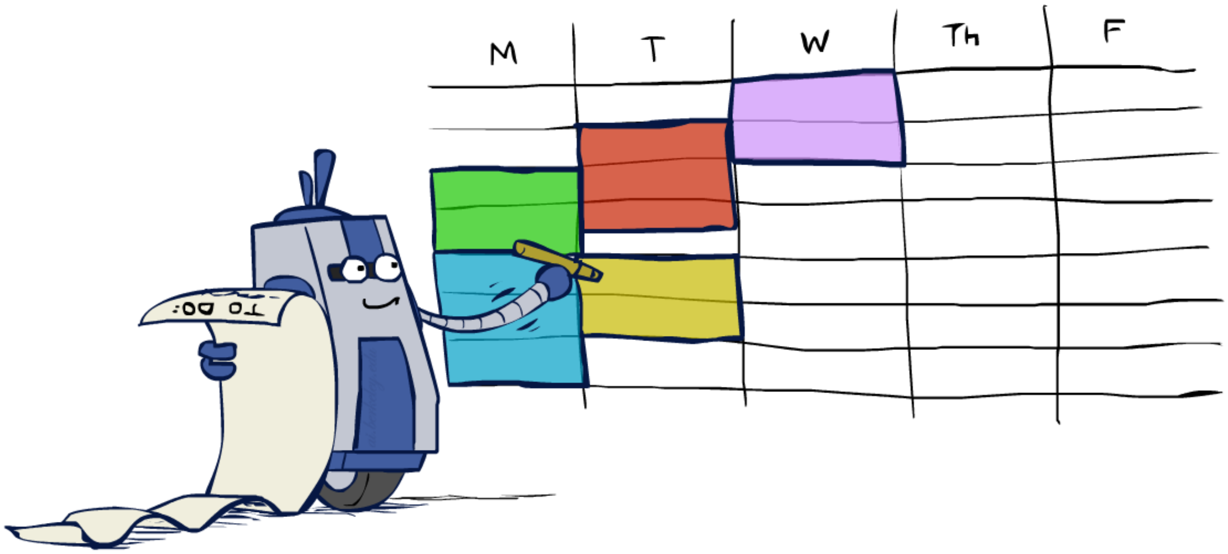


- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)



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 schedulir
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!

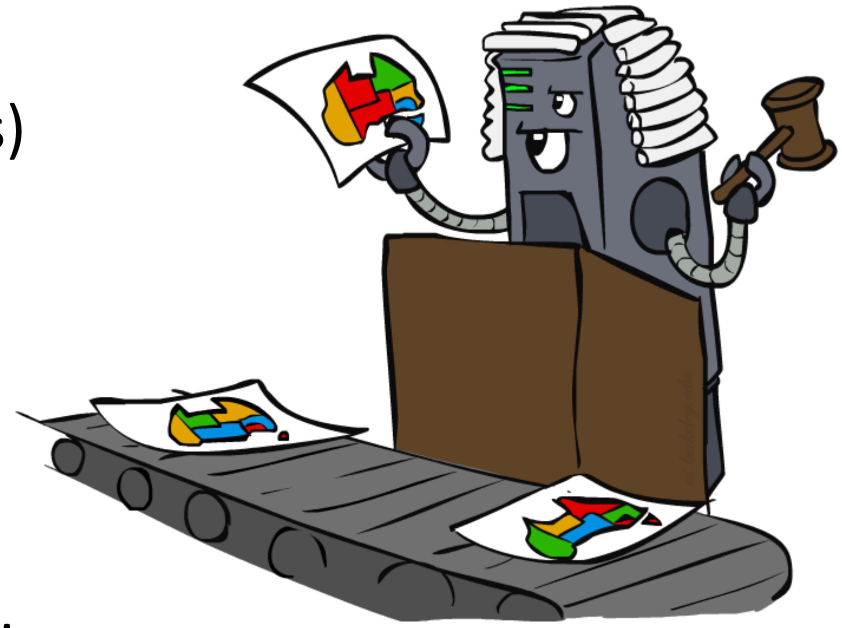


- Many real-world problems involve real-valued variables…

Solving CSPs

Standard Search Formulation

- Standard search formulation of CSPs
- 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

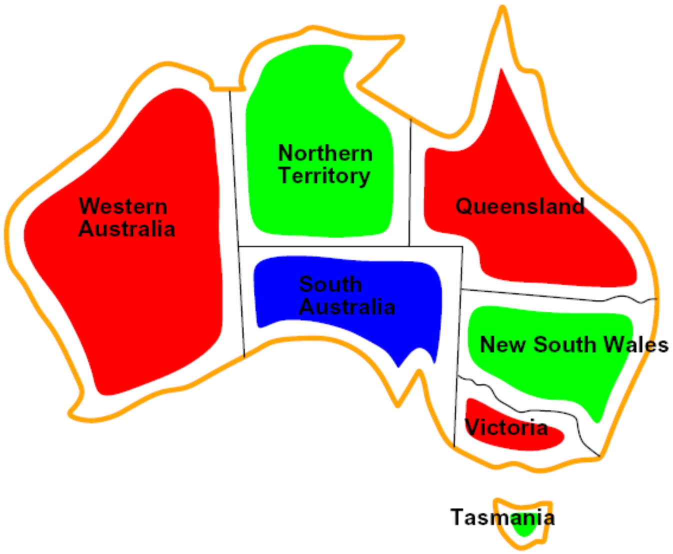


Search Methods

- What would BFS do?

{}

{WA=g}   {WA=r}   …   {NT=g}   …

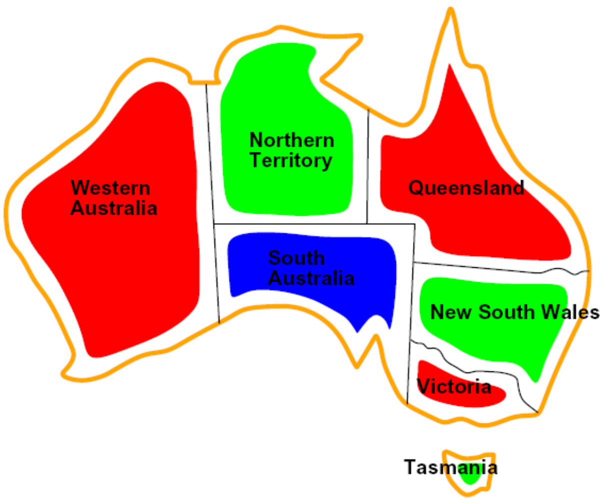


Search Methods

- What would BFS do?

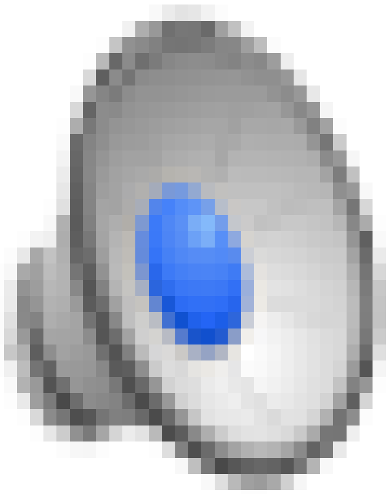
- What would DFS do?
  - let’s see!

- What problems does naïve search have?



[Demo: coloring -- dfs]

Video of Demo Coloring -- DFS

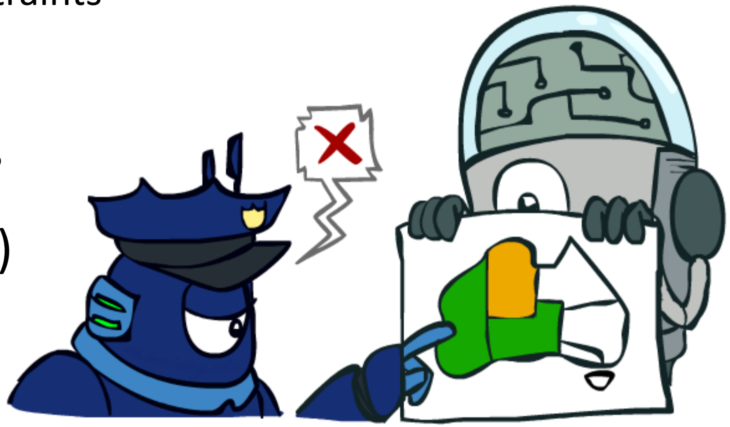


Credit: ai.berkeley.edu

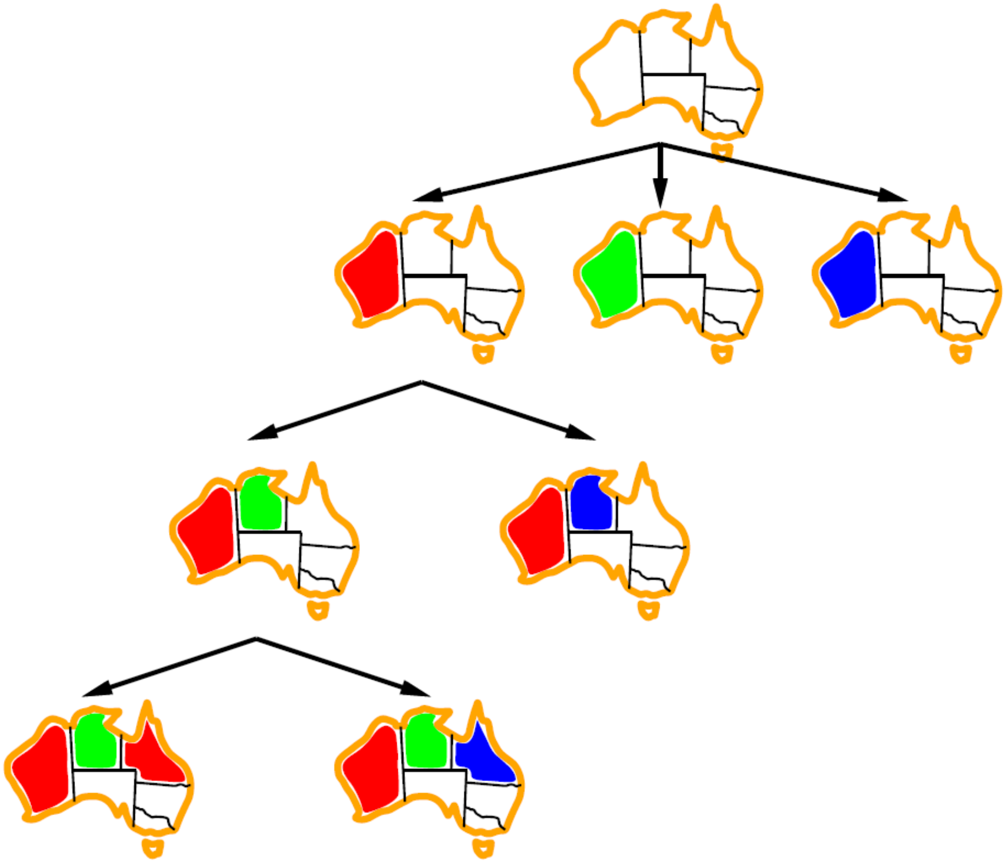
Backtracking Search

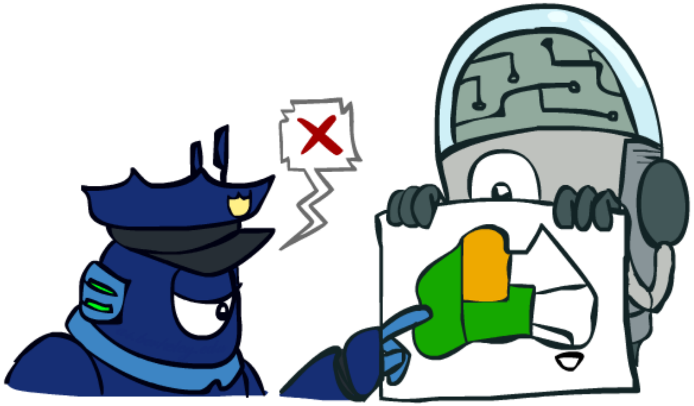
- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments 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 $n \approx 25$



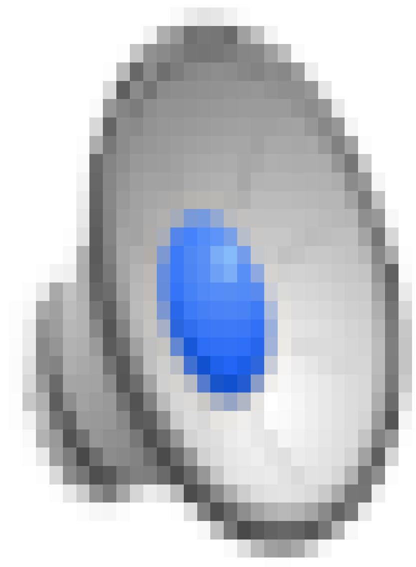
Backtracking Example





[Demo: coloring -- backtracking]

Video of Demo Coloring – Backtracking



Credit: ai.berkeley.edu

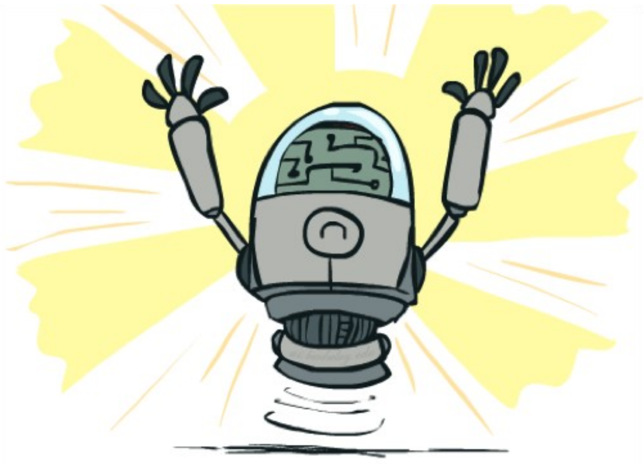
Backtracking Search

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

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

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

Improving Backtracking

- 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?

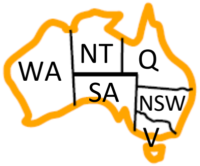


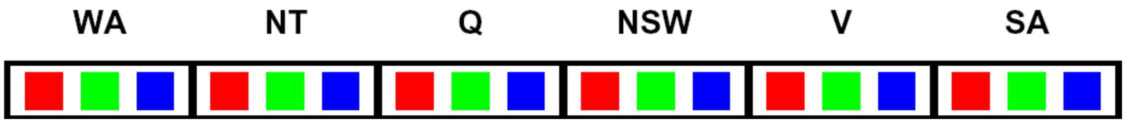
Filtering

Keep track of domains for unassigned variables and **cross off bad options**

Filtering: Forward Checking

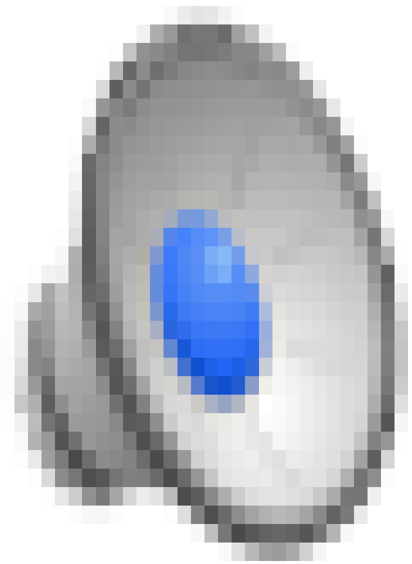
- **Filtering**: Keep track of domains for unassigned variables and cross off bad options
- **Forward checking**: Cross off values that violate a constraint when added to the existing assignment





[Demo: coloring -- forward checking]

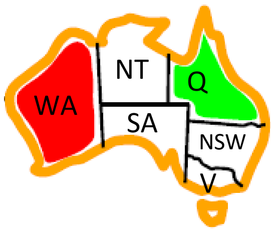
Video of Demo Coloring – Backtracking with Forward Checking



Credit: ai.berkeley.edu

Filtering: Constraint Propagation

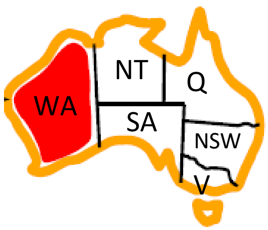
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

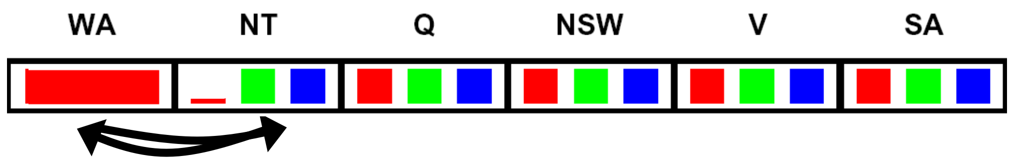


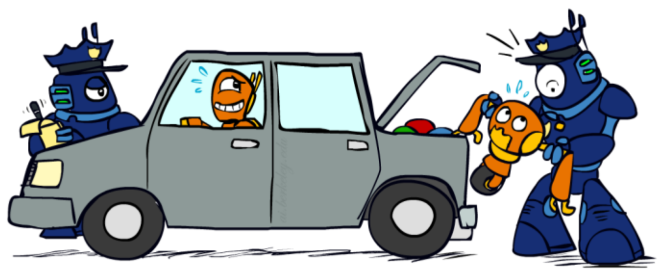
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

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







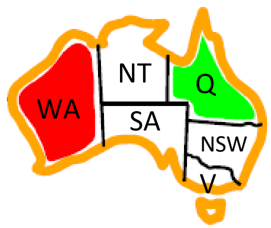
Delete from the tail!

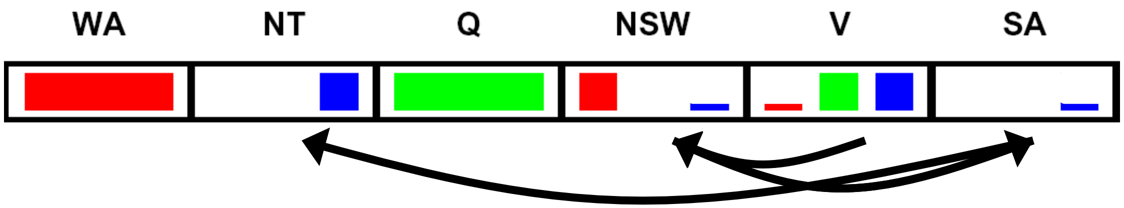
Forward checking?

Enforcing consistency of arcs pointing to each new assignment

Arc Consistency 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
```

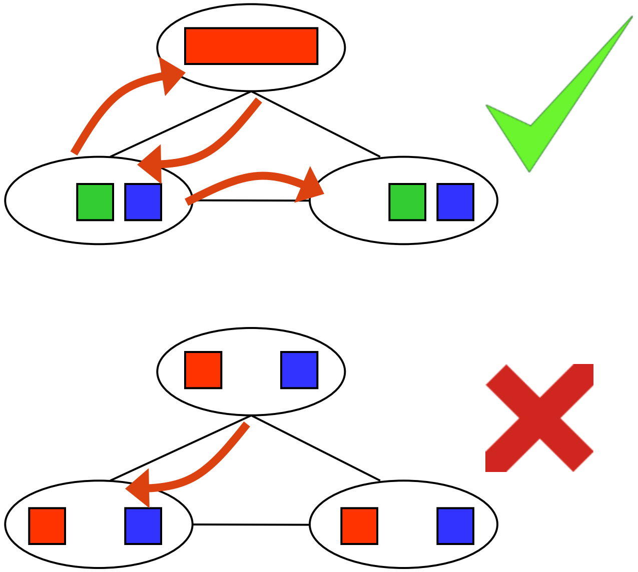
```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

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

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)

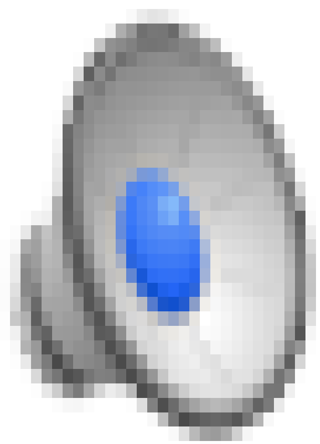
- Arc consistency still runs inside a backtracking search!



[Demo: coloring -- forward checking]

[Demo: coloring -- arc consistency]

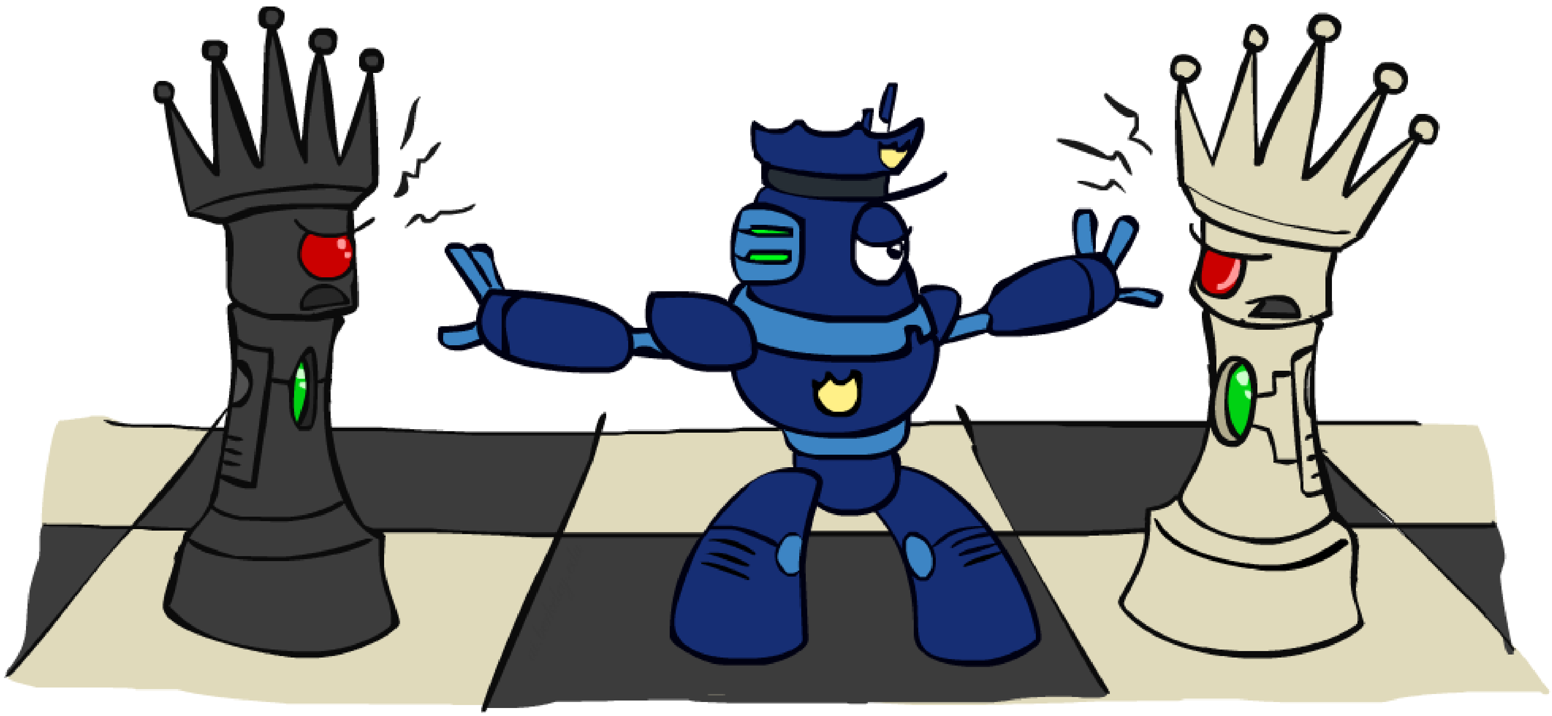
Demo – Backtracking with Forward Checking – Complex Graph



Credit: ai.berkeley.edu

Demo – Backtracking with Arc Consistency – Complex Graph

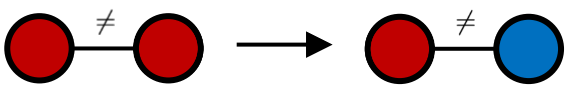
Iterative Improvement

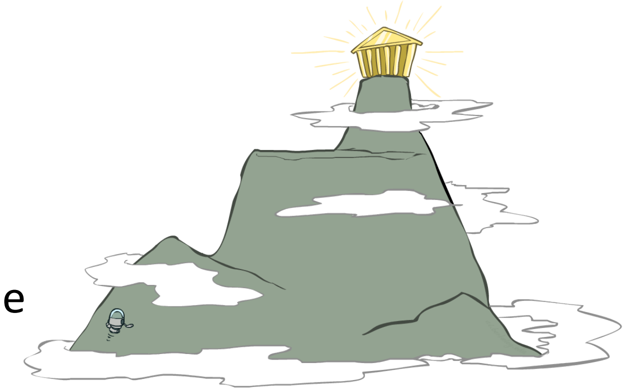


Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned

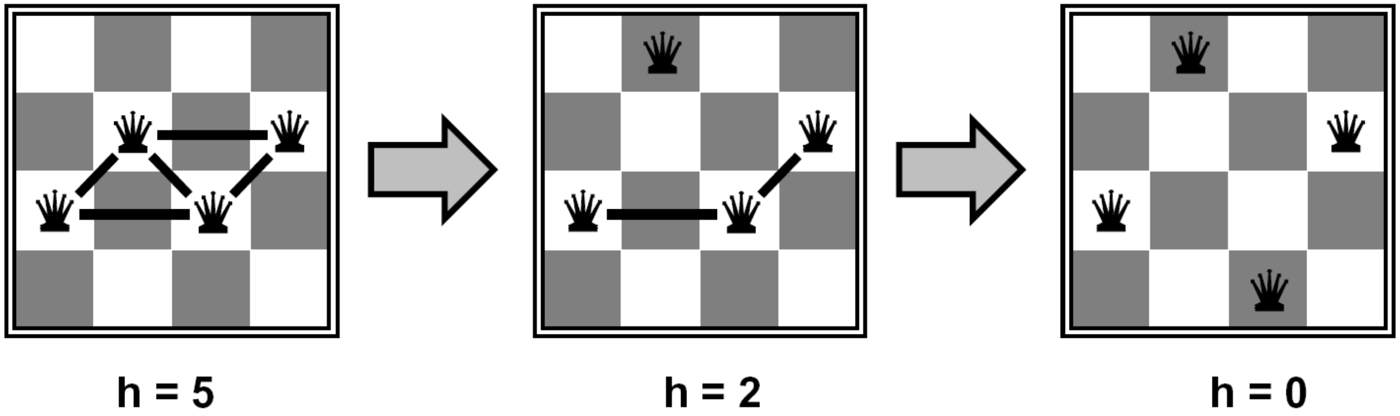
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.





- 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(x) = total number of violated constraints

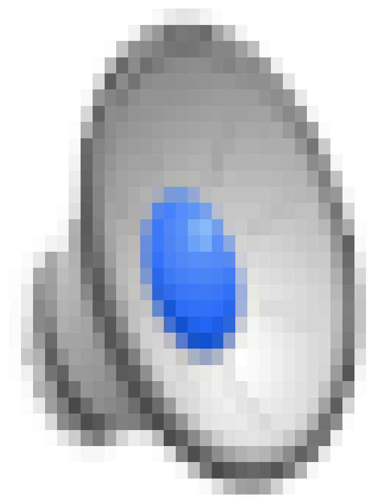
Example: 4-Queens with Min-Conflict heuristics



- 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

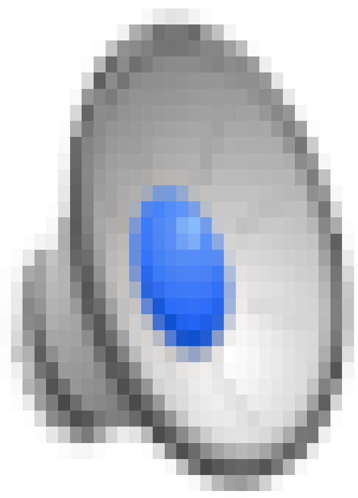
[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

Video of Demo Iterative Improvement – n Queens



Credit: ai.berkeley.edu

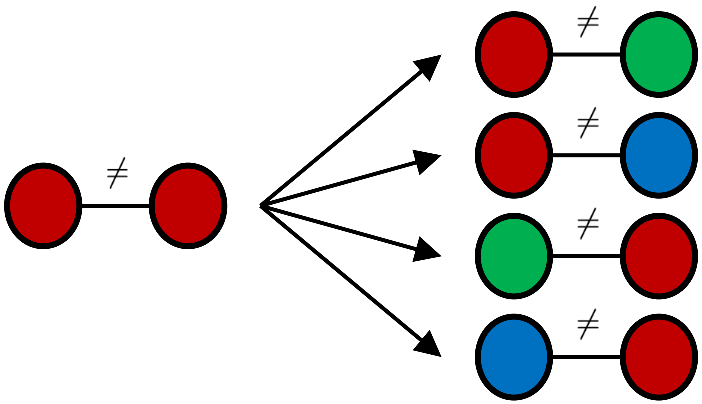
Video of Demo Iterative Improvement – Coloring



Local Search (AIMA 4.1)

Local Search (AIMA 4.1)

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes

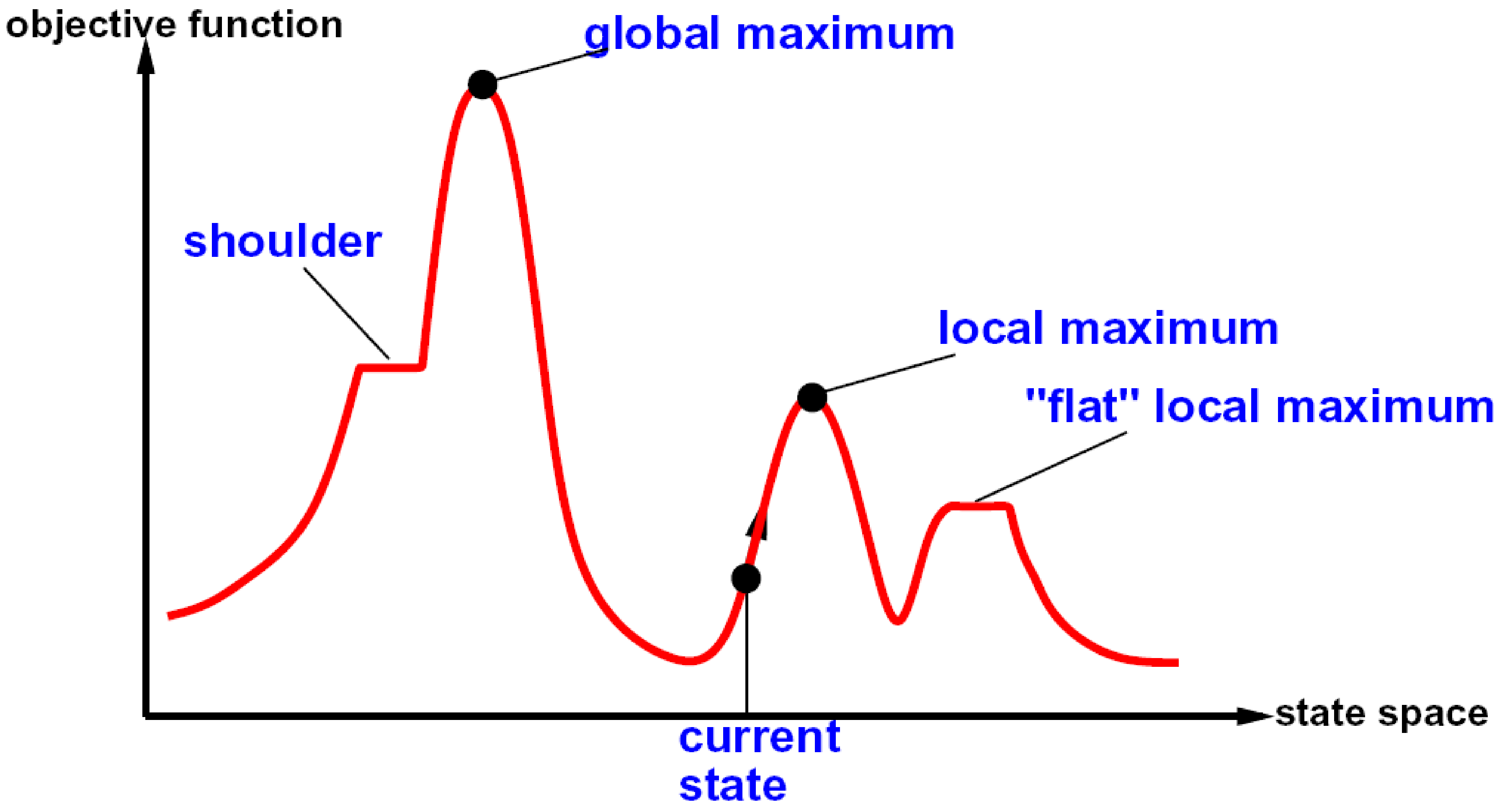


- Generally, much faster and more memory efficient (but incomplete and suboptimal)

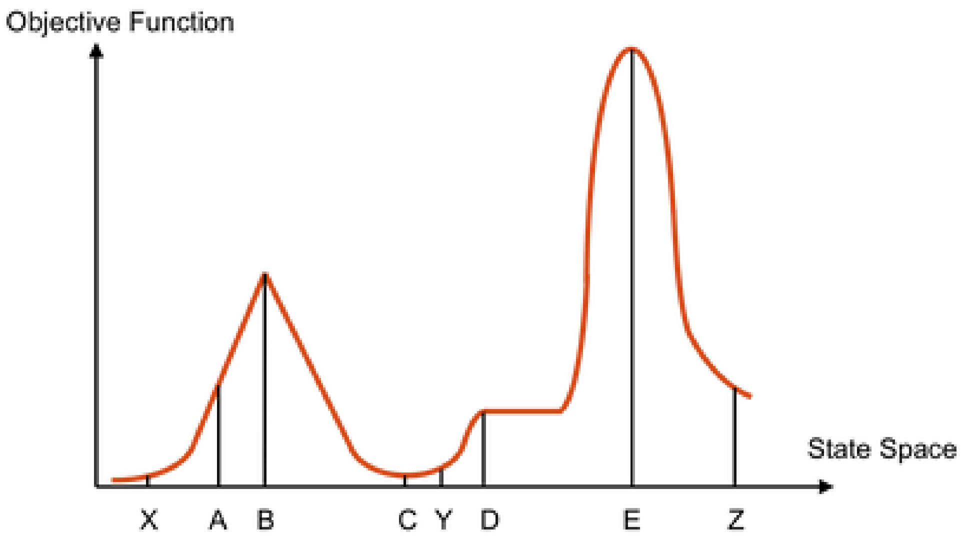
Hill Climbing (AIMA 4.1)

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What's bad about this approach?
- What's good about it?

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

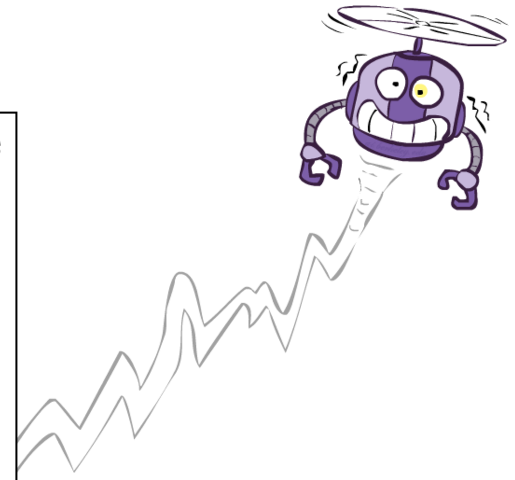
Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Simulated Annealing

- Idea: Escape local maxima by allowing downhill move
  - But make them rarer as time goes on

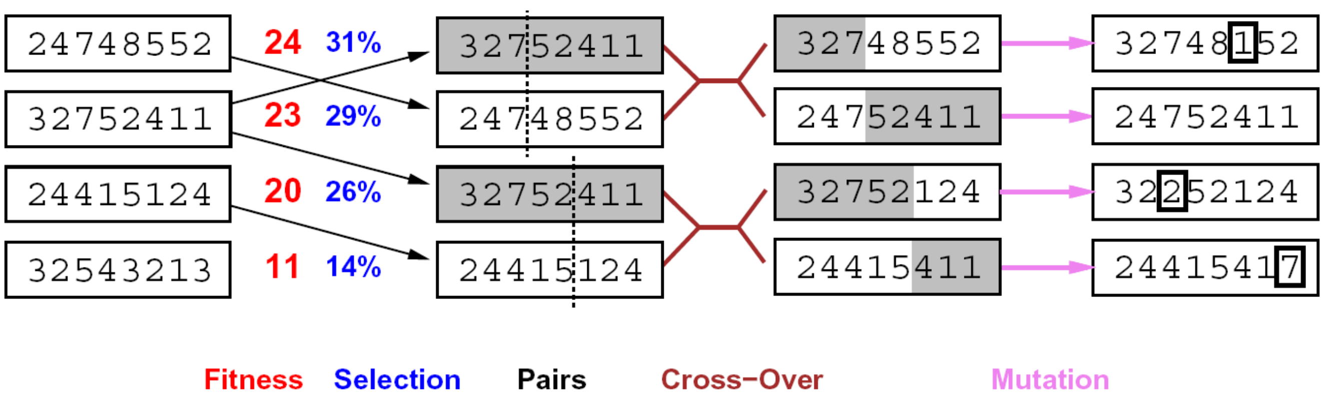
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

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

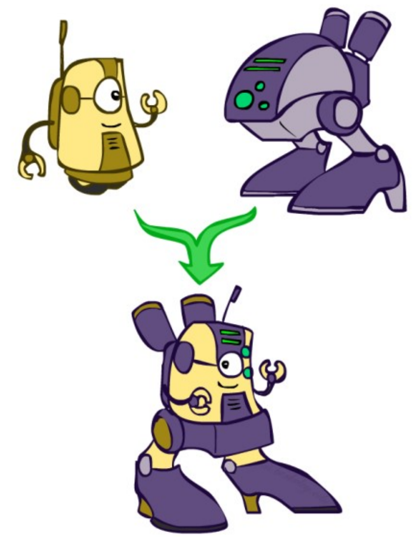


Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!
- Is this an interesting guarantee?
- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

Genetic Algorithms

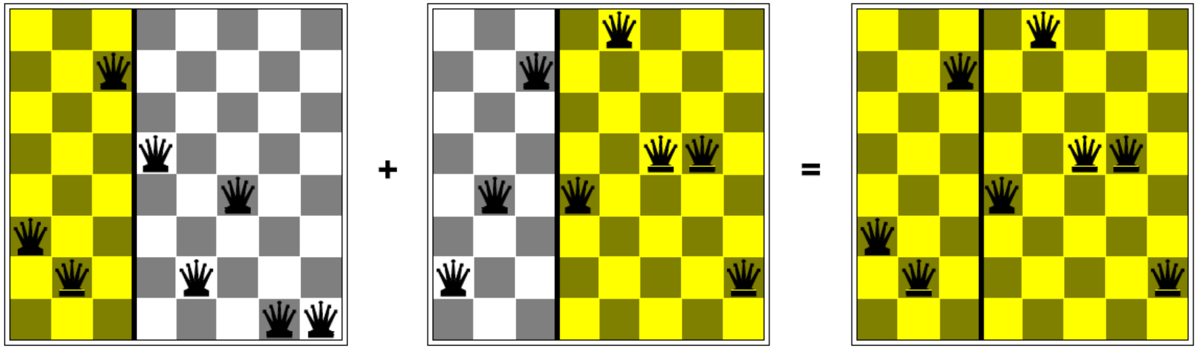




- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety

- Possibly the most misunderstood, misapplied (and even maligned) technique around

Example: N-Queens



- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?