CS 380: Artificial Intelligence

Lecture 5:
Local Search & CSPs

A General Method for Building Heuristics

- **Pattern Databases** – Store the exact optimal solution cost for every possible subproblem instance.
  - You care about some features of the problem, but not all
  - In this example, all the possible locations of [1,3,6,*blank*]
  - Identify the instance in the DB to get the h value

Start

	3	6
1	*	*
*	*	*

Goal

1	*	3
*	*	6
*	*	

Recall: Problem Solving

- Idea: represent the problem we want to solve as:
  - State space
  - Actions
  - Goal check
  - Heuristic / evaluation function

- If this is possible, we can use search methods to solve any problem:
  - BFS, DFS, ID, Greedy Search, A*

Systematic vs. Local Search

- All the algorithms we have seen so far are "systematic":
  - They will systematically explore ALL the states in the search space if needed
    - each of them just explores it in a different order
  - If the search space is finite, they are all complete
- Local Search:
  - Instead of systematically exploring the state space…
  - Start with one state (even a bad one), and iteratively improve it by exploring neighbor states ("local")
  - Search space is the set of all states visited
  - *Iterative improvement*

Iterative improvement algorithms

In many optimization problems, **path** is irrelevant;
the goal state itself is the solution

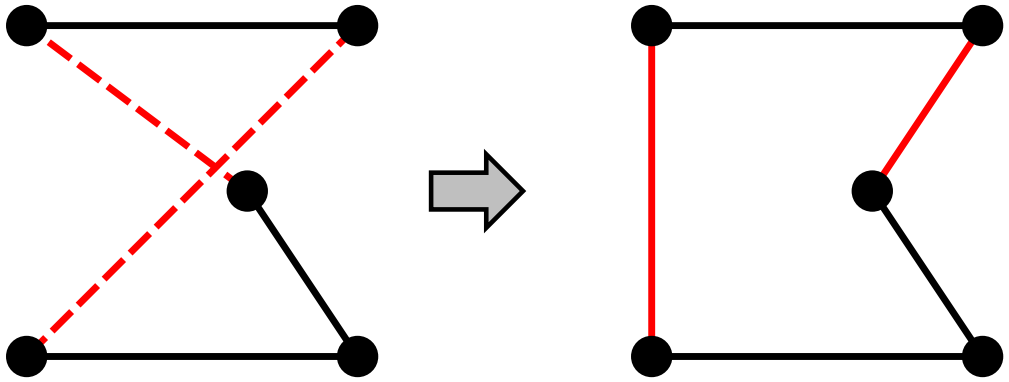
Then state space = set of "complete" configurations;
find **optimal** configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable

In such cases, can use iterative improvement algorithms;
keep a single "current" state, try to improve it

Constant space, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges

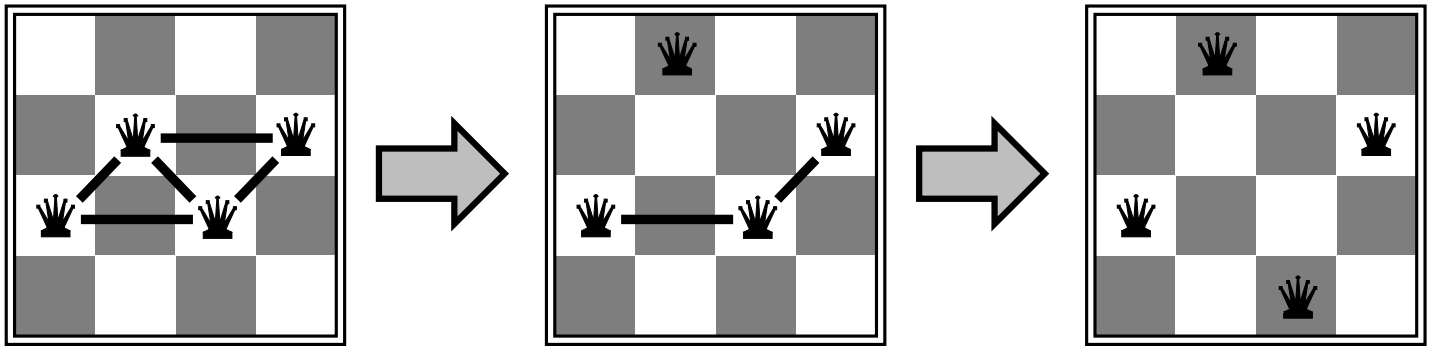


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1million$

Hill-climbing (or gradient ascent/descent)

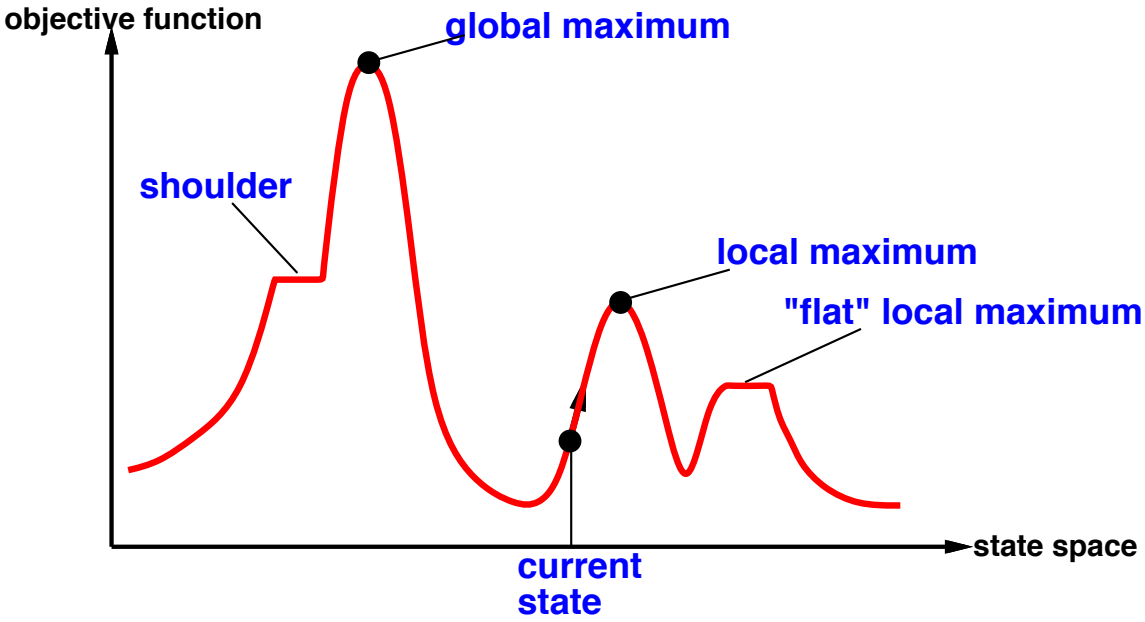
"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Hill-climbing contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves ☺ escape from shoulders ☹ loop on flat maxima

Simulated Annealing

- "Annealing" = heating and then controlled cooling (of metals to strengthen the material)
- At a fixed temperature *T*, the state occupation probability reaches the Boltzmann distribution.

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

- If T is decreased slowly enough you are guaranteed to reach the optimal state (if we need optimal!?)
- Visualization:
  - https://www.youtube.com/watch?v=iaq_Fpr4KZc

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

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 $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next $\leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
    else *current* $\leftarrow$ *next* only with probability $e^{\Delta\ E/T}$

Properties of simulated annealing

At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state x^*
because $e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.

Local beam search

Idea: keep k states instead of 1; choose top k of all their successors

Not the same as k searches run in parallel!
Searches that find good states recruit other searches to join them

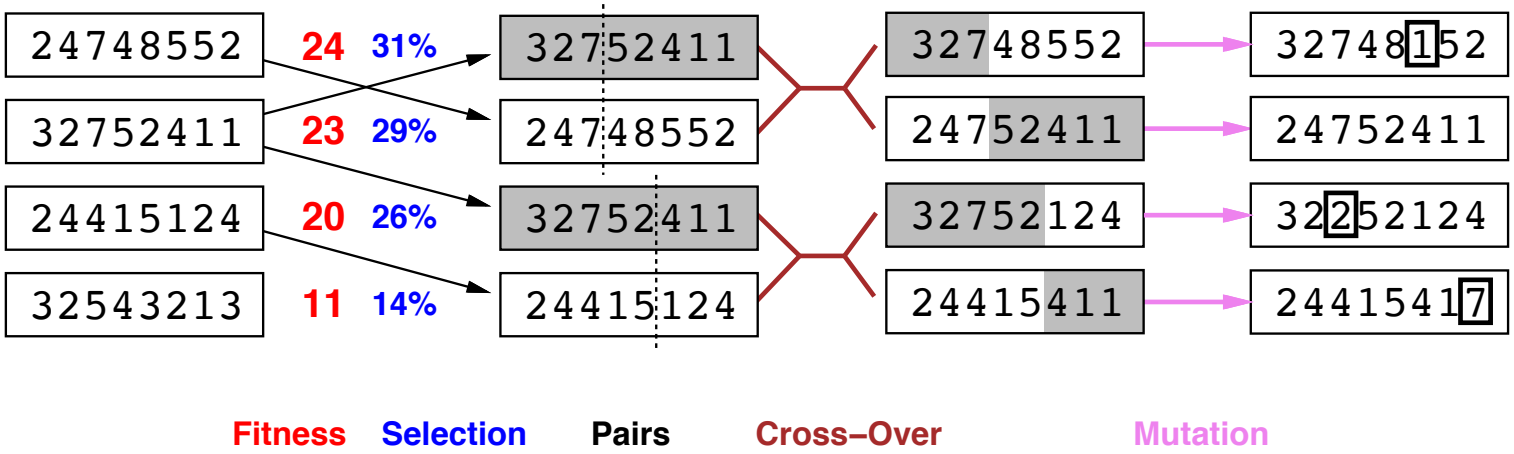
Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic algorithms

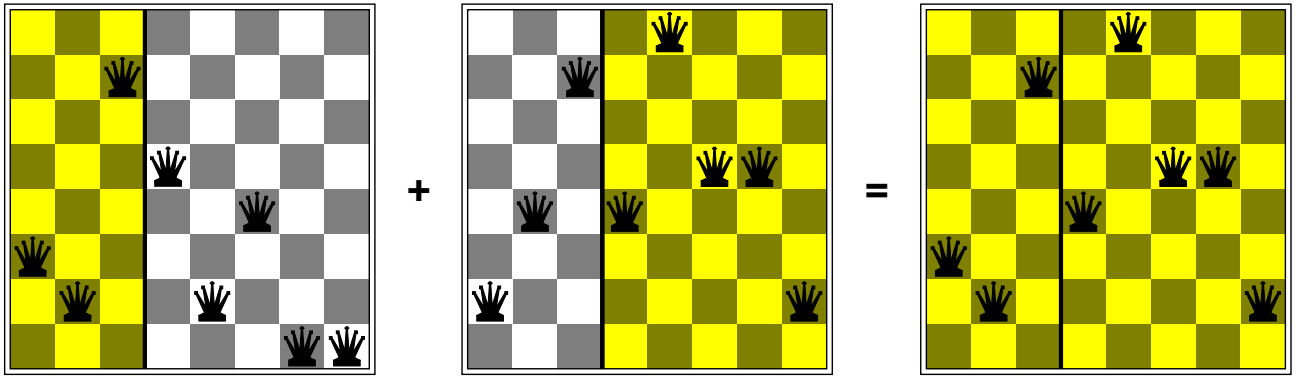
= stochastic local beam search + generate successors from **pairs** of states



Genetic algorithms contd.

GAs require states encoded as strings (GPs use programs)

Crossover helps **iff substrings are meaningful components**



GAs $\neq$ evolution: e.g., real genes encode replication machinery!

Sample Visualization

- Greedy search, local search, simulated annealing…

https://www.youtube.com/watch?v=SC5CX8drAtU

Continuous state spaces

Suppose we want to site three airports in Romania:

- 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
- objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$ sum of squared distances from each city to nearest airport

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers $\pm\delta$ change in each coordinate

Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).
Newton–Raphson (1664, 1690) iterates $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$
to solve $\nabla f(\mathbf{x}) = 0$, where $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

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* V_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

Example: 4-Queens as a CSP

Assume one queen in each column. Which row does each one go in?

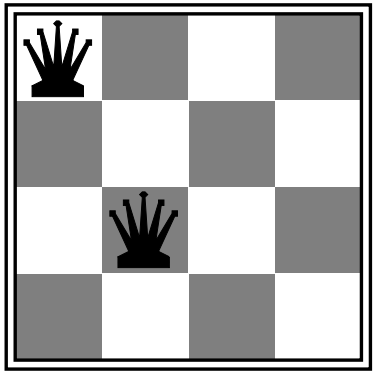
Variables Q_1, Q_2, Q_3, Q_4

Domains $D_i = \{1, 2, 3, 4\}$

Constraints

$Q_i \neq Q_j$ (cannot be in same row)

$|Q_i - Q_j| \neq |i - j|$ (or same diagonal)



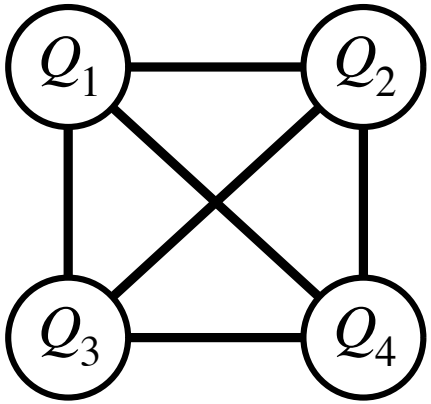
Translate each constraint into set of allowable values for its variables

E.g., values for (Q_1, Q_2) are $(1, 3)$ $(1, 4)$ $(2, 4)$ $(3, 1)$ $(4, 1)$ $(4, 2)$

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



Example: Cryptarithmetic

Variables

$D\ E\ M\ N\ O\ R\ S\ Y$

Domains

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

Constraints

$M \neq 0$, $S \neq 0$ ($unary$ constraints)

$Y = D + E$ or $Y = D + E - 10$, etc.

$D \neq E$, $D \neq M$, $D \neq N$, etc.

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Example: Map coloring

Color a map so that no adjacant countries have the same color

Variables

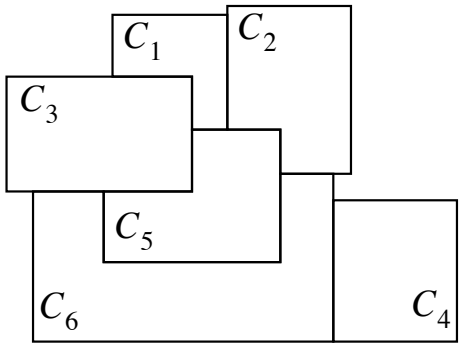
Countries C_i

Domains

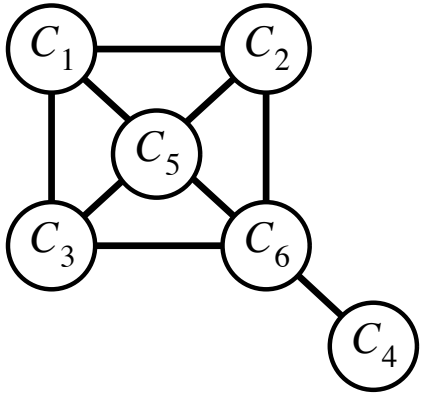
$\{Red, Blue, Green\}$

Constraints

$C_1 \neq C_2$, $C_1 \neq C_5$, etc.



Constraint graph:



Real-world CSPs

Assignment problems
  e.g., who teaches what class

Timetabling problems
  e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

Applying standard search

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

Initial state: all variables unassigned

Operators: assign a value to an unassigned variable

Goal test: all variables assigned, no constraints violated

Notice that this is the same for all CSPs!

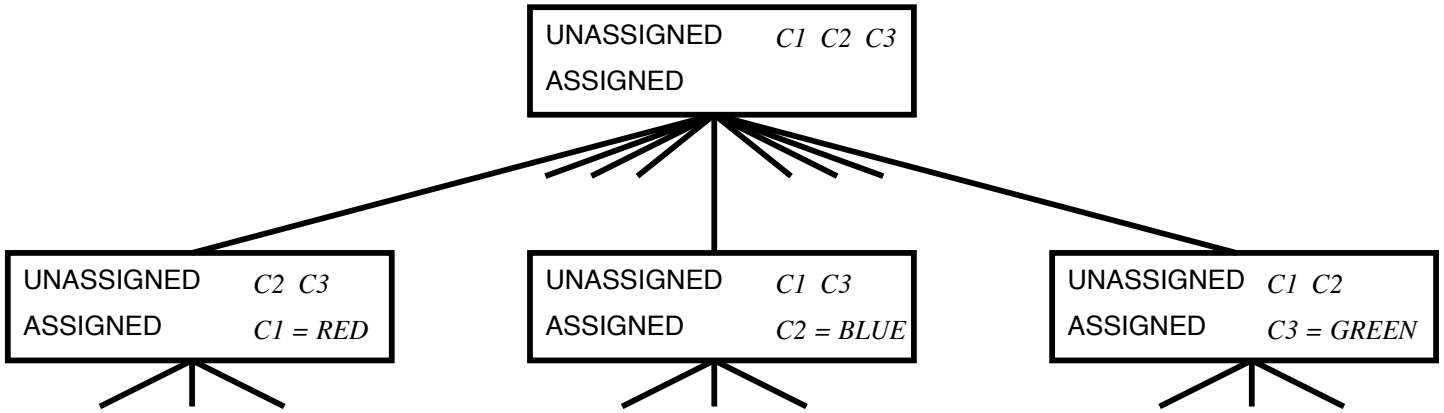
Implementation

CSP state keeps track of which variables have values so far
Each variable has a domain and a current value

datatype CSP-STATE
  components: UNASSIGNED, a list of variables not yet assigned
    ASSIGNED, a list of variables that have values
datatype CSP-VAR
  components: NAME, for i/o purposes
    DOMAIN, a list of possible values
    VALUE, current value (if any)

Constraints can be represented
  explicitly as sets of allowable values, or
  implicitly by a function that tests for satisfaction of the constraint

Standard search applied to map-coloring



Complexity of the dumb approach

Max. depth of space $m =$??

Depth of solution state $d =$??

Search algorithm to use??

Branching factor $b =$??

This can be improved dramatically by noting the following:

1) Order of assignment is irrelevant, hence many paths are equivalent
2) Adding assignments cannot correct a violated constraint

Complexity of the dumb approach

Max. depth of space $m =$?? n (number of variables)

Depth of solution state $d =$?? n (all vars assigned)

Search algorithm to use?? depth-first

Branching factor $b =$?? $\sum_i |D_i|$ (at top of tree)

$|D_i|$ = size of Domain *i*

This can be improved dramatically by noting the following:

1) Order of assignment is irrelevant so many paths are equivalent
2) Adding assignments cannot correct a violated constraint

Backtracking search

Use depth-first search, but

1) fix the order of assignment, $\Rightarrow b = |D_i|$
   (can be done in the SUCCESSORS function)
2) check for constraint violations

The constraint violation check can be implemented in two ways:

1) modify SUCCESSORS to assign only values that
   are allowed, given the values already assigned
or 2) check constraints are satisfied before expanding a state

Backtracking search is the basic uninformed algorithm for CSPs

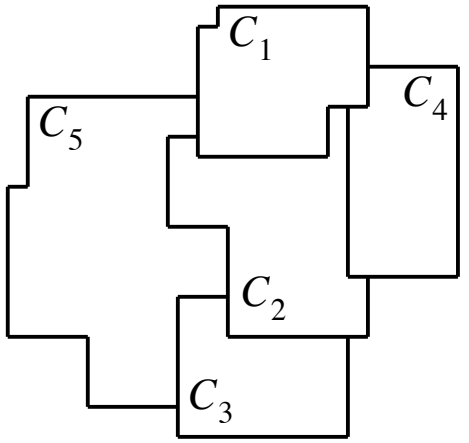
Can solve n-queens for $n \approx 15$

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1			
C_2			
C_3			
C_4			
C_5			



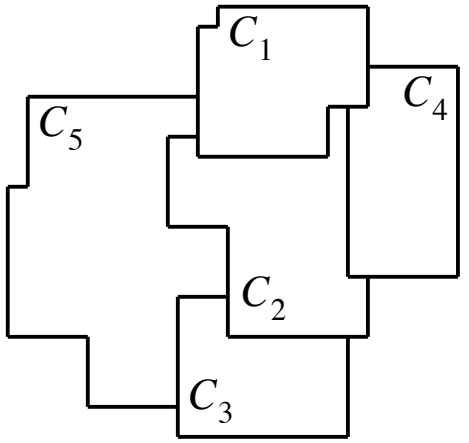
Can solve n-queens up to $n \approx 30$

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1	✓		
C_2	x		
C_3			
C_4	x		
C_5	x		



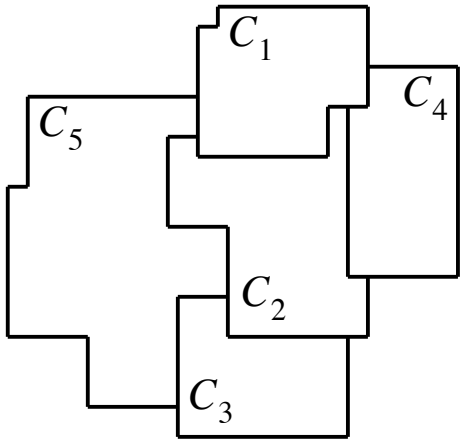
Can solve n-queens up to $n \approx 30$

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1	✓		
C_2	x	✓	
C_3		x	
C_4	x	x	
C_5	x	x	



Can solve n-queens up to $n \approx 30$

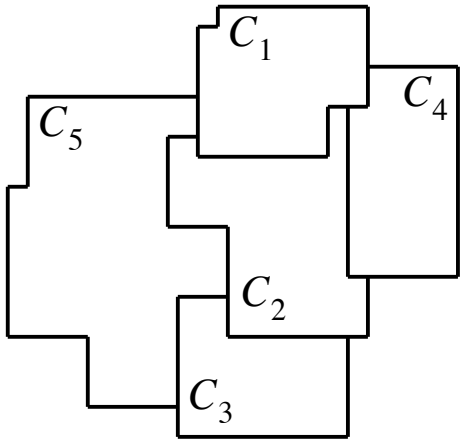
Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1	✓		
C_2	x	✓	
C_3		x	✓
C_4	x	x	
C_5	x	x	x

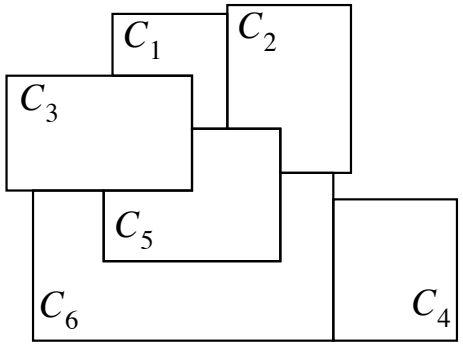
← problem!



Can solve n-queens up to $n \approx 30$

Heuristics for CSPs

More intelligent decisions on
- which value to choose for each variable
- which variable to assign next



Given $C_1 = Red$, $C_2 = Green$, choose $C_3 =$??

.

Given $C_1 = Red$, $C_2 = Green$, what next??

.

Can solve n-queens for $n \approx 1000$

Heuristics for CSPs

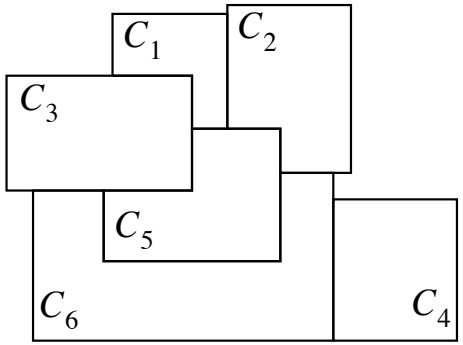
More intelligent decisions on
- which value to choose for each variable
- which variable to assign next

Given $C_1 = Red$, $C_2 = Green$, choose $C_3 =$??

$C_3 = Green$: *least-constraining-value*

Given $C_1 = Red$, $C_2 = Green$, what next??

C_5: *most-constrained-variable*



Can solve n-queens for $n \approx 1000$

Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

To apply to CSPs:
- allow states with unsatisfied constraints
- operators *reassign* variable values

Variable selection: randomly select any conflicted variable

min-conflicts heuristic:
- choose value that violates the fewest constraints
- i.e., hillclimb with $h(n)$ = total number of violated constraints

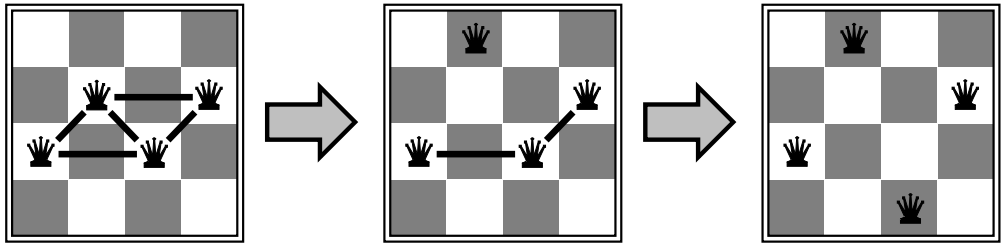
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n)$ = number of attacks

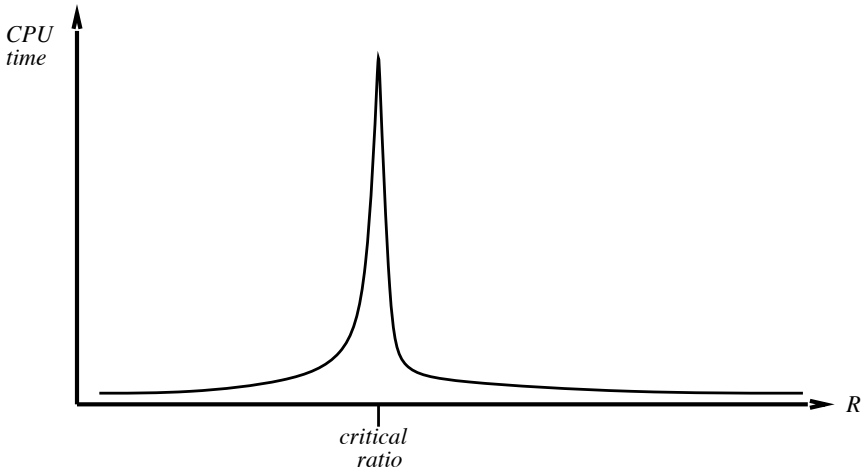


Performance of min-conflicts

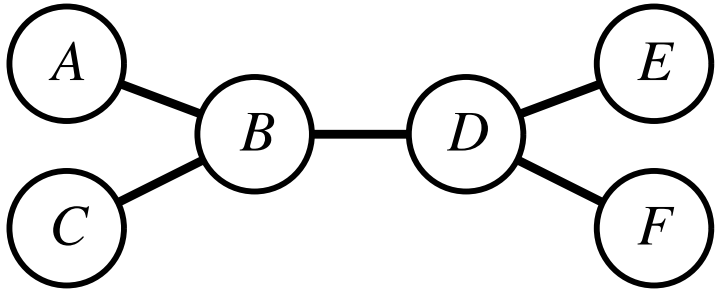
Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

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



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 $O(|D|^n)$

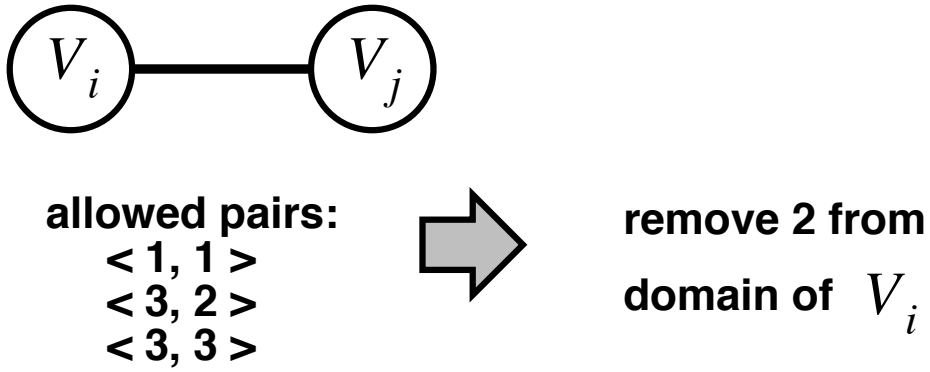
This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions and complexity of reasoning.

Algorithm for tree-structured CSPs

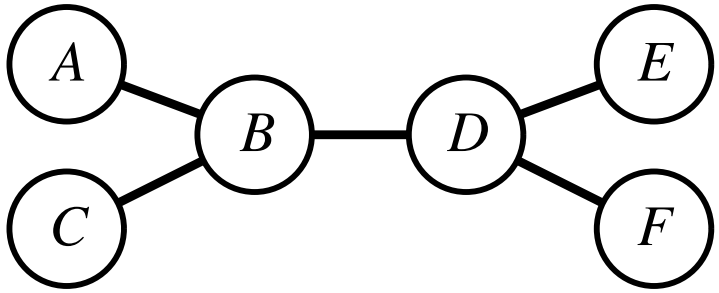
Basic step is called *filtering*:

FILTER(V_i, V_j)
removes values of V_i that are inconsistent with ALL values of V_j

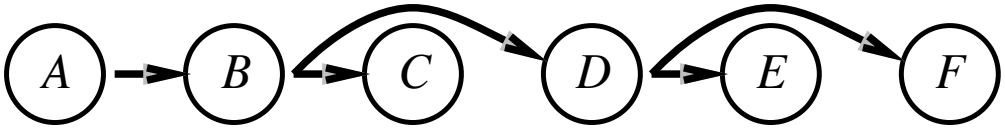
Filtering example:



Algorithm contd.



1) Order nodes breadth-first starting from any leaf:



2) For $j = n$ to 1, apply $\textsc{Filter}(V_i, V_j)$ where V_i is a parent of V_j

3) For $j = 1$ to n, pick legal value for V_j given parent value

Summary

CSPs are a special kind of problem:
- states defined by values of a fixed set of variables
- goal test defined by *constraints* on variable values

Backtracking = depth-first search with
1) fixed variable order
2) only legal successors

Forward checking prevents assignments that guarantee later failure

Variable ordering and value selection heuristics help significantly

Iterative min-conflicts is usually effective in practice

Tree-structured CSPs can always be solved very efficiently

Coming up…

- We will start applying ideas of search to adversarial domains, where one player is playing another…