Local Search Algorithms

This lecture topic Chapter 4.1-4.2

Next lecture topic
Chapter 6

(Please read lecture topic material before and after each lecture on that topic)

Outline

- Hill-climbing search
  - Gradient Descent in continuous spaces
- Simulated annealing search
- Tabu search
- Local beam search
- Genetic algorithms
- Linear Programming

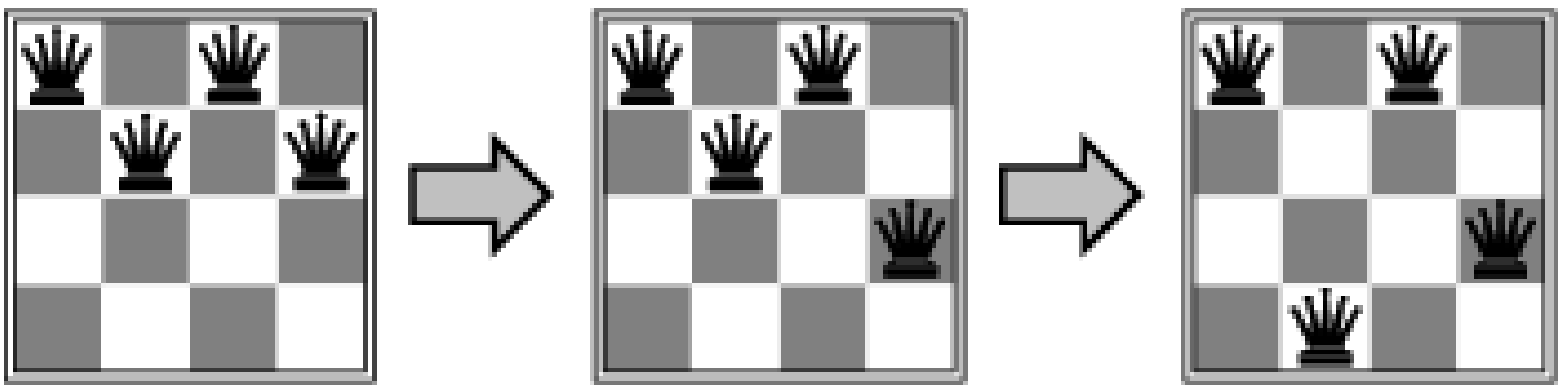
Local search algorithms

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

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

Example: *n*-queens

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



Note that a state cannot be an incomplete configuration with m<n queens

Hill-climbing search

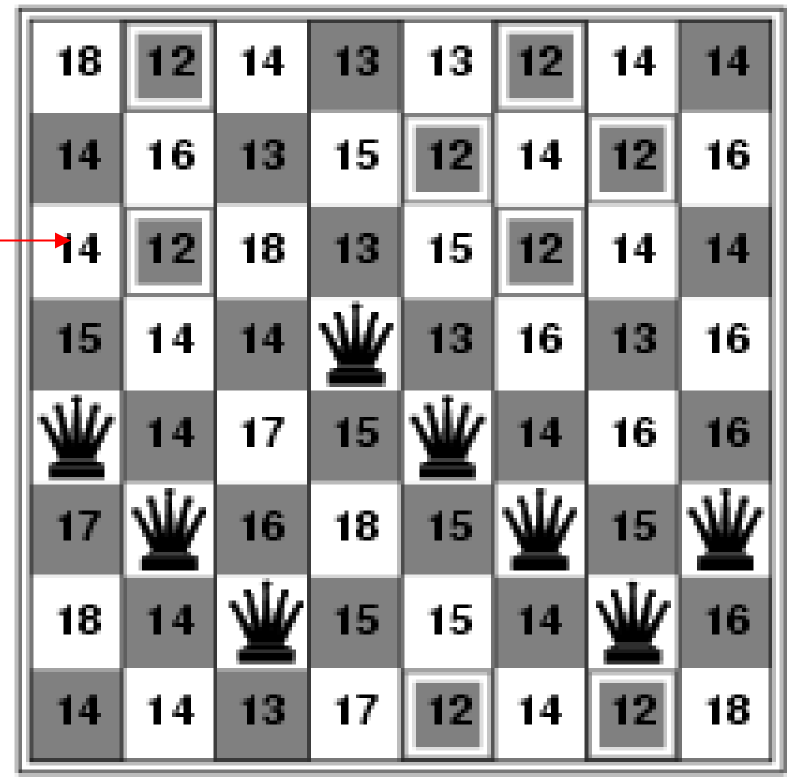
- "Like climbing Everest in thick fog with amnesia"
-

```
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
```

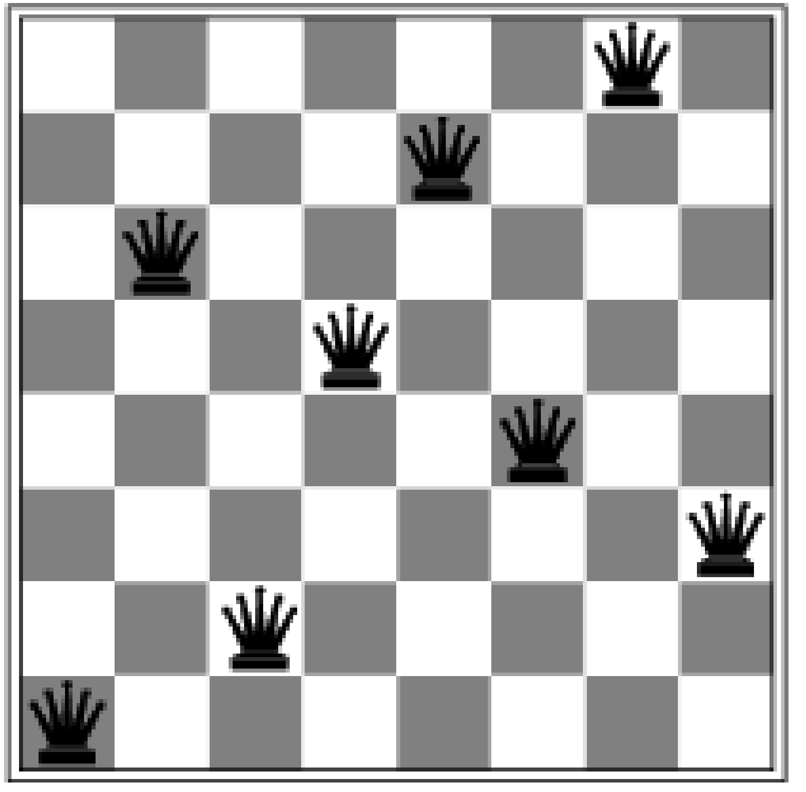
Hill-climbing search: 8-queens problem

Each number indicates h if we move a queen in its corresponding column



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly (*h = 17* for the above state)

Hill-climbing search: 8-queens problem

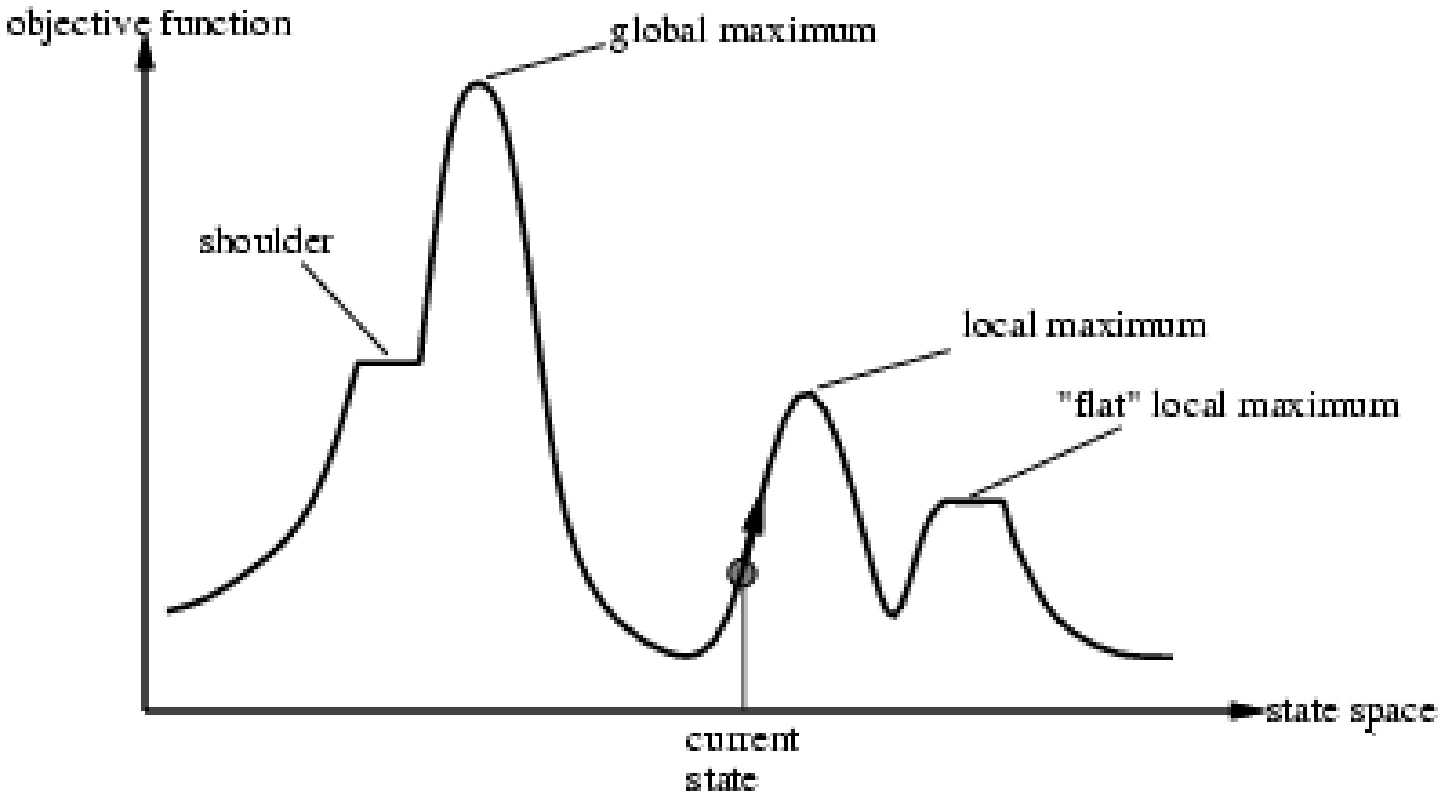


- A local minimum with $h = 1$

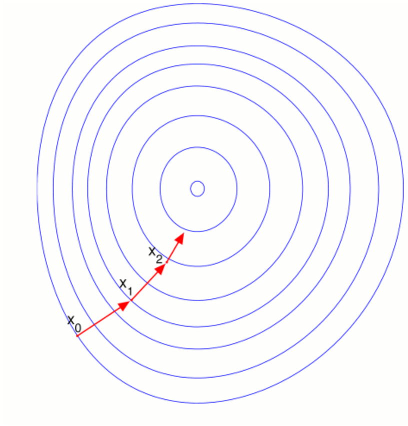
(what can you do to get out of this local minima?)

Hill-climbing Difficulties

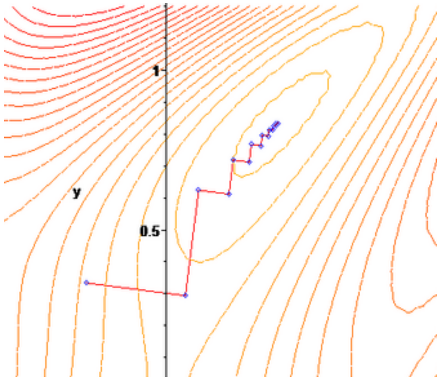
- Problem: depending on initial state, can get stuck in local maxima



Gradient Descent



- Assume we have some cost-function: $C(x_1,\ldots,x_n)$
 and we want minimize over continuous variables X1,X2,..,Xn



1. Compute the *gradient* : $\frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$

2. Take a small step downhill in the direction of the gradient:

$$x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$$

3. Check if $C(x_1,..,x'_i,..,x_n) < C(x_1,..,x_i,..,x_n)$

4. If true then accept move, if not reject.

5. Repeat.

Line Search

- In GD you need to choose a step-size.
- Line search picks a direction, v, (say the gradient direction) and searches along that direction for the optimal step:

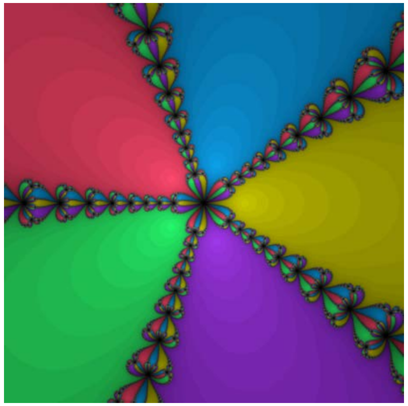
$$\eta^* = \arg\min \; C(x_t + \eta v_t)$$

- Repeated doubling can be used to effectively search for the optimal step:

$$\eta \to 2\eta \to 4\eta \to 8\eta \quad \text{(until cost increases)}$$

- There are many methods to pick search direction v. Very good method is “conjugate gradients”.

Newton's Method



Basins of attraction for x5 − 1 = 0;
darker means more iterations to converge.

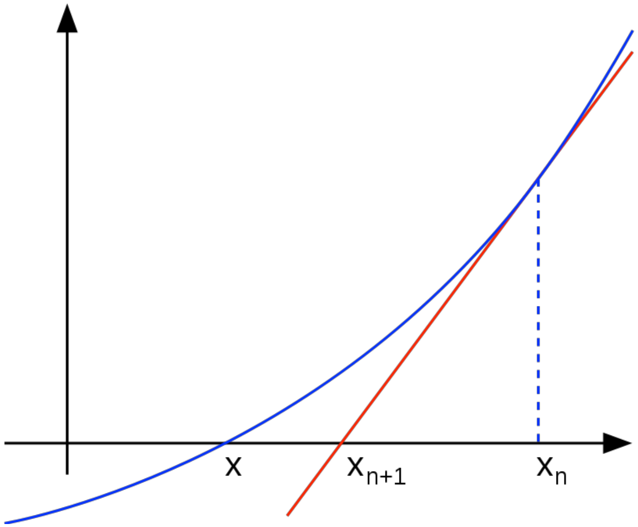
- Want to find the roots of f(x).
- To do that, we compute the tangent at Xn and compute where it crosses the x-axis.

$$\nabla f(x_n) = \frac{0 - f(x_n)}{x_{n+1} - x_n} \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{\nabla f(x_n)}$$

- Optimization: find roots of $\nabla f(x_n)$

$$\nabla\nabla f(x_n) = \frac{0 - \nabla f(x_n)}{x_{n+1} - x_n} \Rightarrow x_{n+1} = x_n - \frac{\nabla f(x_n)}{[\nabla\nabla f(x_n)]}$$

- Does not always converge & sometimes unstable.
- If it converges, it converges very fast



Simulated annealing search

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

- This is like smoothing the cost landscape.

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their 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 ← 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^(ΔE/T)
```

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1 (however, this may take VERY long)
  - However, in any finite search space RANDOM GUESSING also will find a global optimum with probability approaching 1 .

- Widely used in VLSI layout, airline scheduling, etc.

Tabu Search

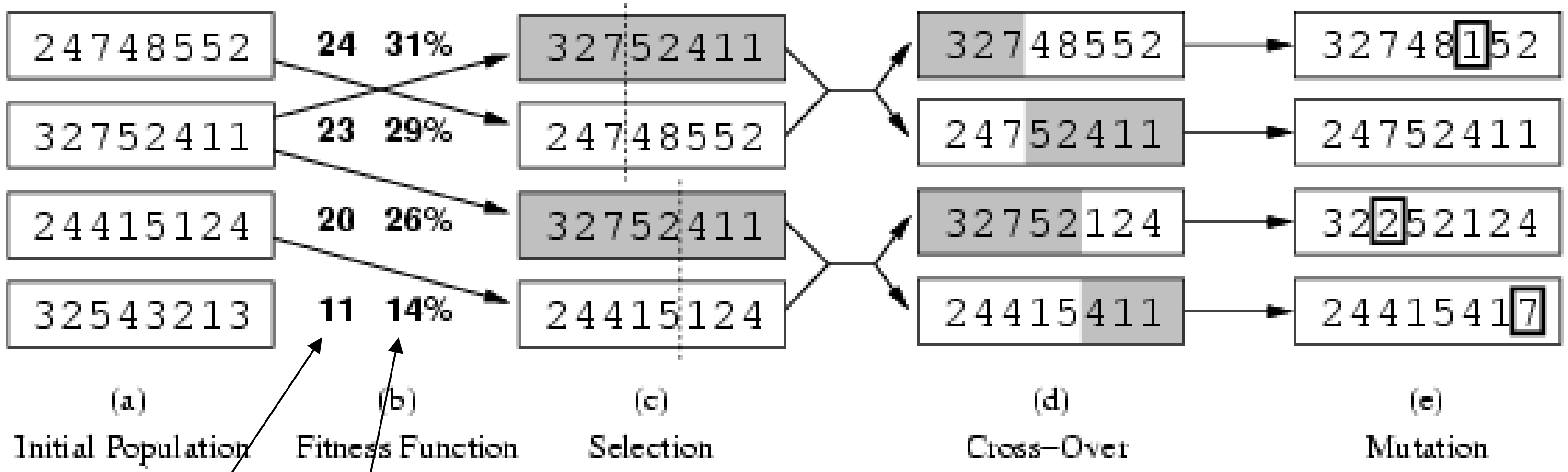
- Almost any simple local search method, but with a memory.
- Recently visited states are added to a tabu-list and are temporarily excluded from being visited again.
- This way, the solver moves away from already explored regions and (in principle) avoids getting stuck in local minima.
- Tabu search can be added to most other local search methods to obtain a variant method that avoids recently visited states.
- Tabu-list is usually implemented as a hash table for rapid access. Can also add a LIFO queue to keep track of oldest node.
- Unit time cost per step for tabu test and tabu-list maintenance.

Local beam search

- Keep track of *k* states rather than just one.
- Start with *k* randomly generated states.
- At each iteration, all the successors of all *k* states are generated.
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.
- Concentrates search effort in areas believed to be fruitful.
  - May lose diversity as search progresses, resulting in wasted effort.

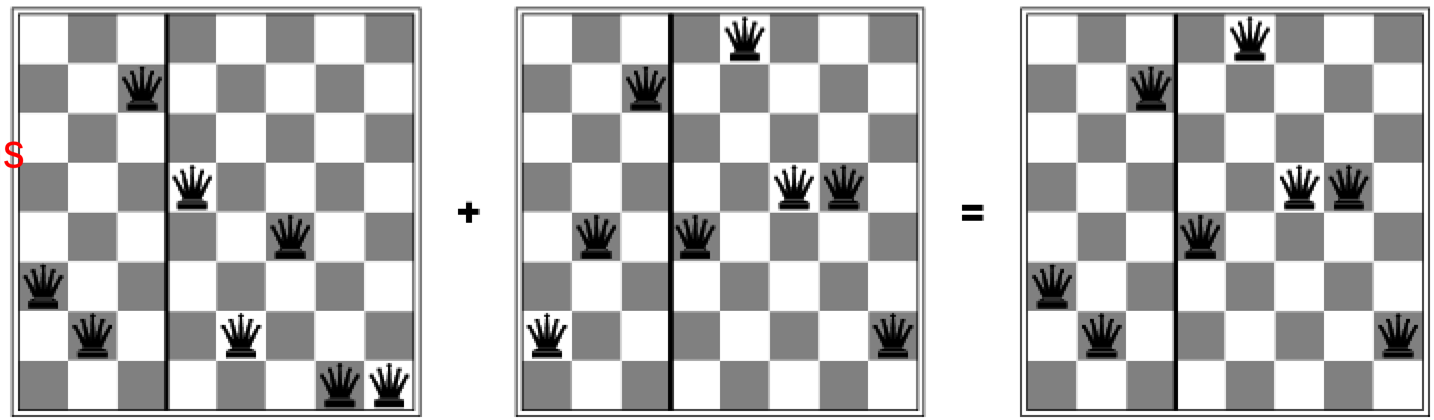
Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation



fitness:
#non-attacking queens

probability of being
regenerated
in next generation



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- P(child) = 24/(24+23+20+11) = 31%
- P(child) = 23/(24+23+20+11) = 29% etc

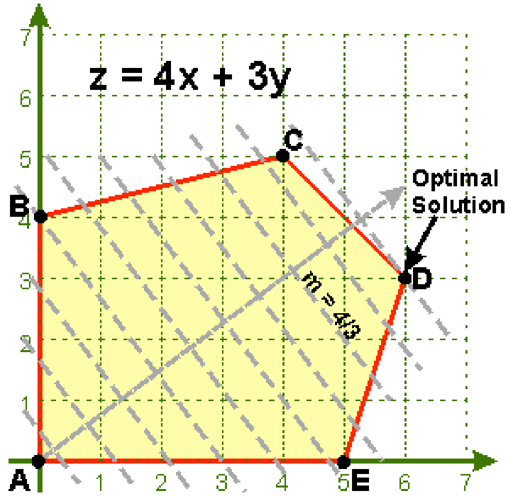
Linear Programming

Problems of the sort:

$$\text{maximize}\; c^T x$$
$$\text{subject to}: \mathrm{Ax} \le \mathrm{a};\, \mathrm{Bx} = \mathrm{b}$$

- Very efficient "off-the-shelves" solvers are available for LRs.

- They can solve large problems with thousands of variables.



Linear Programming Constraints

- Maximize: $z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$
- Primary constraints: $x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0$
- Additional constraints:
- $a_{i1} x_1 + a_{i2} x_2 + \ldots + a_{in} x_n \leq a_i, (a_i \geq 0)$
- $a_{j1} x_1 + a_{j2} x_2 + \ldots + a_{jn} x_n \geq a_j \geq 0$
- $b_{k1} x_1 + b_{k2} x_2 + \ldots + b_{kn} x_n = b_k \geq 0$

Summary

- Local search maintains a complete solution
  - Seeks to find a consistent solution (also complete)
- Path search maintains a consistent solution
  - Seeks to find a complete solution (also consistent)
- Goal of both: complete and consistent solution
  - Strategy: maintain one condition, seek other
- Local search often works well on large problems
  - Abandons optimality
  - Always has some answer available (best found so far)