Local Search Algorithms

This lecture topic Chapter 4.1-4.2

Next lecture topic
Chapter 5

(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

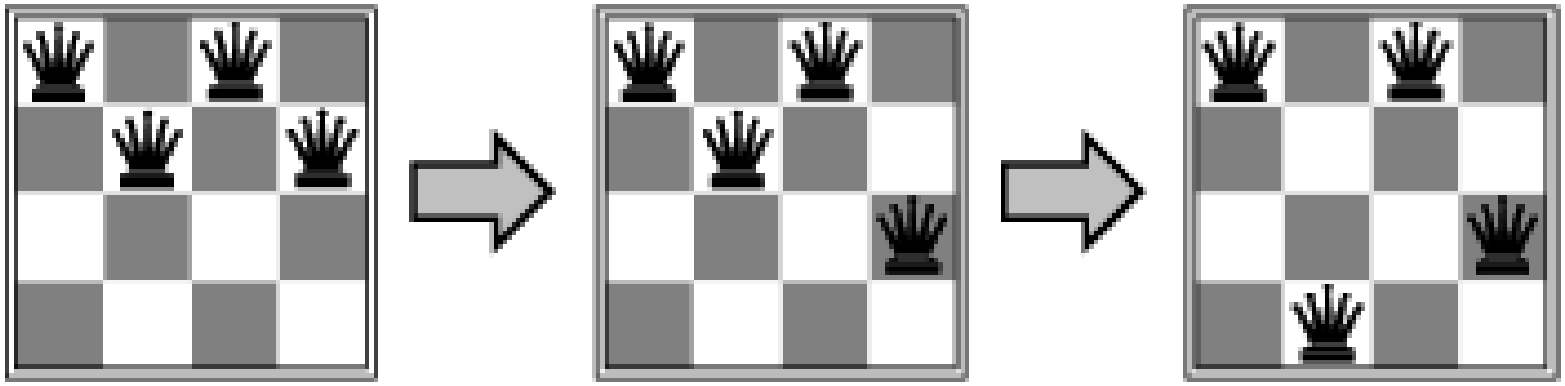
Local search algorithms

- 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* × *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

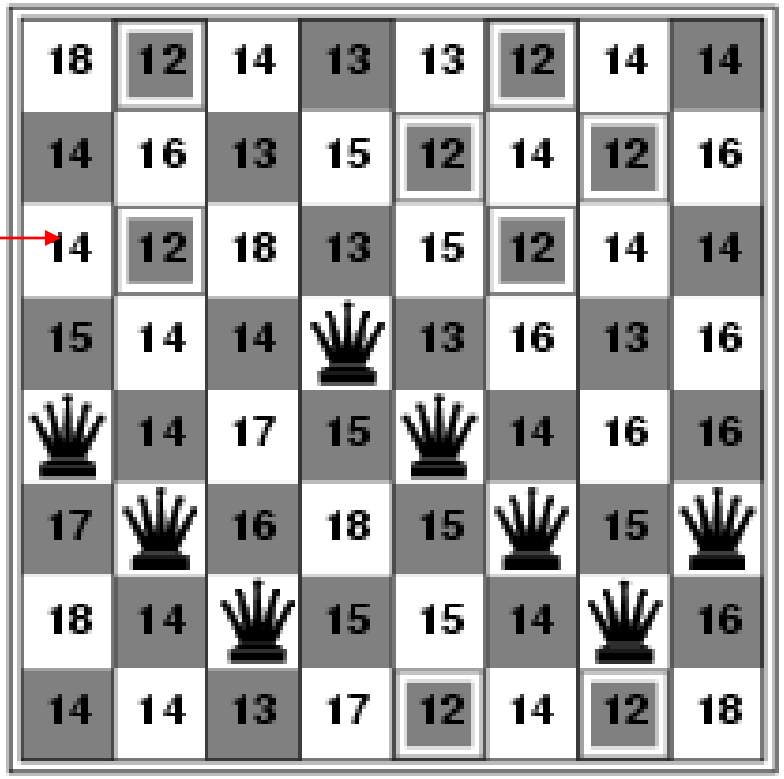
- "Like climbing Everest in thick fog with amnesia"
-

```
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
```

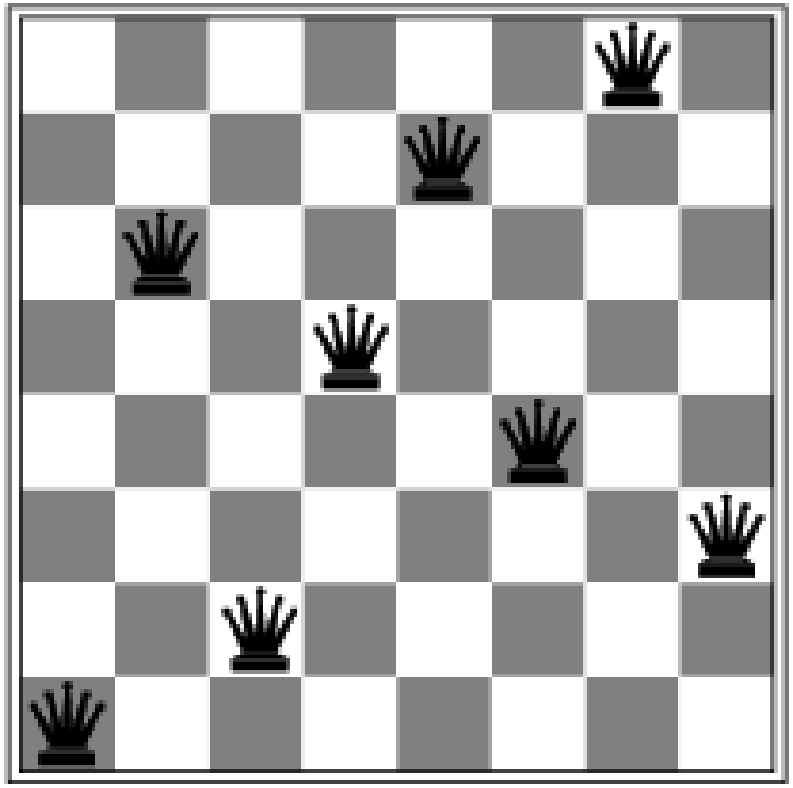
Hill-climbing search: 8-queens problem

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

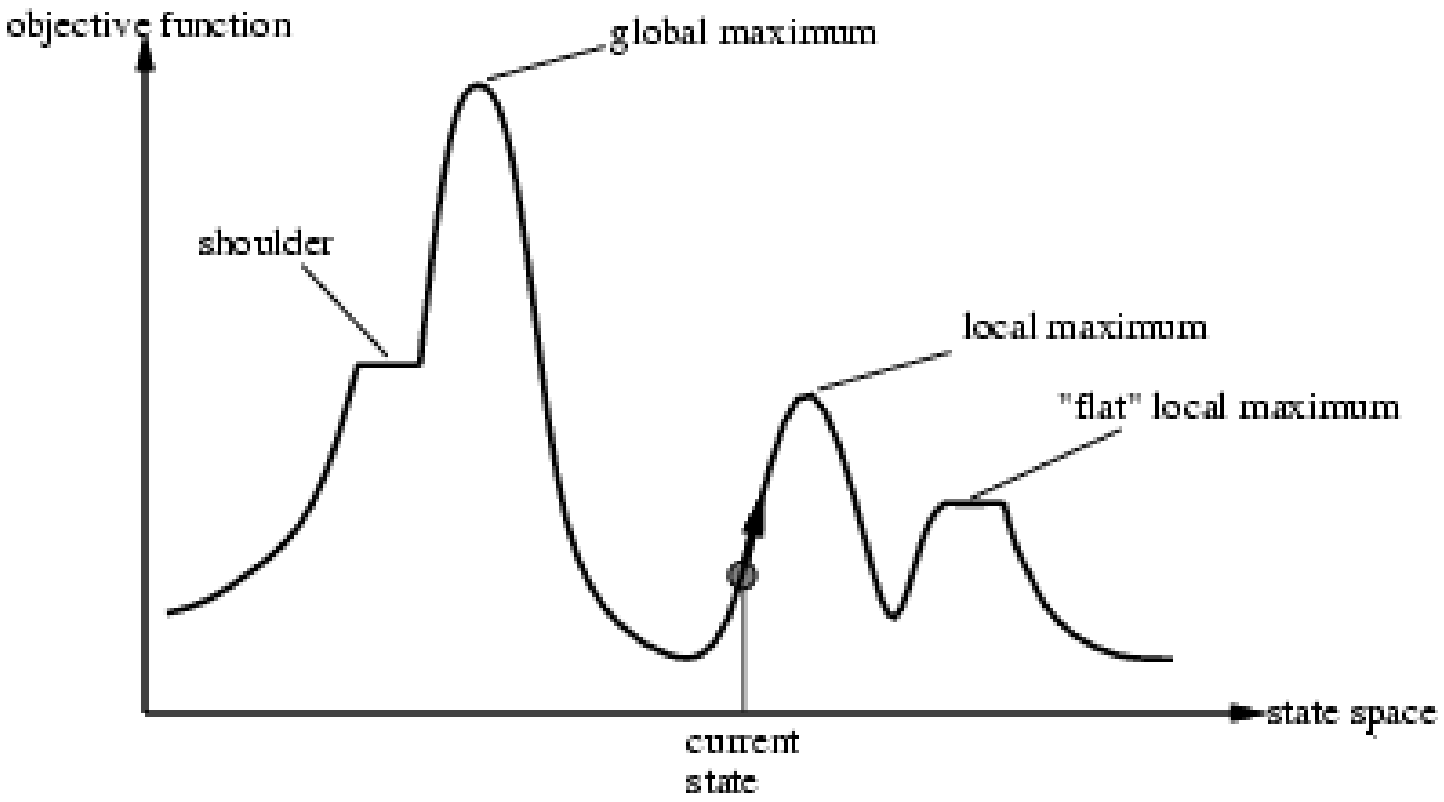


- A local minimum with $h = 1$

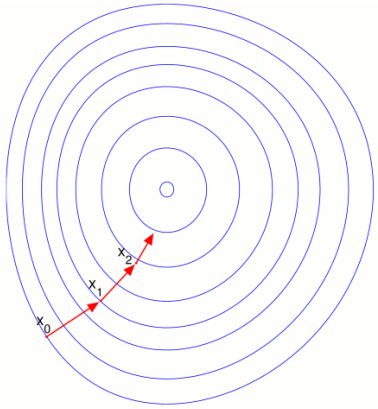
(what can you do to get out of this local minima?)

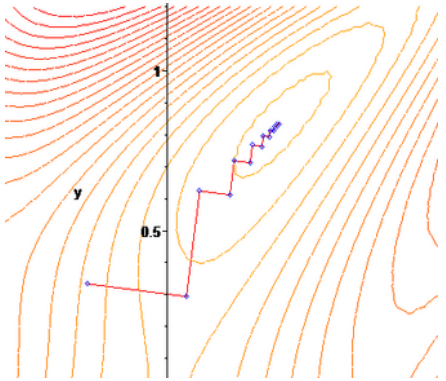
Hill-climbing Difficulties

- 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:

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

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:

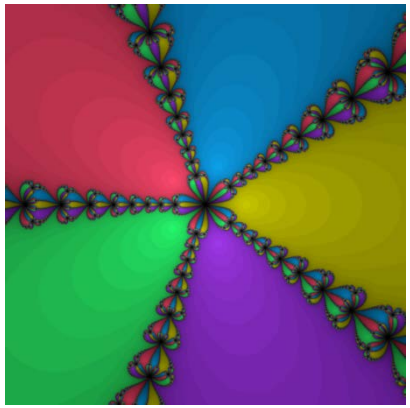
$$\eta^* = \arg\min\ C(x_t + \eta v_t)$$

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

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

- 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.

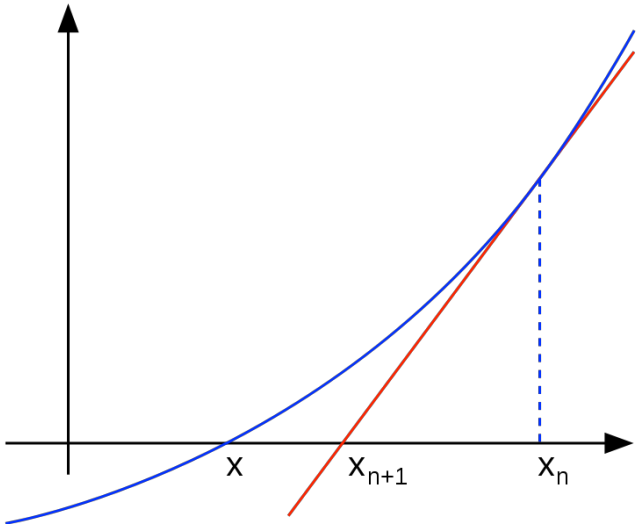
- Want to find the roots of f(x).
- To do that, we compute the tangent at Xn and compute where it crosses the x-axis.

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

- 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^(Δ E/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

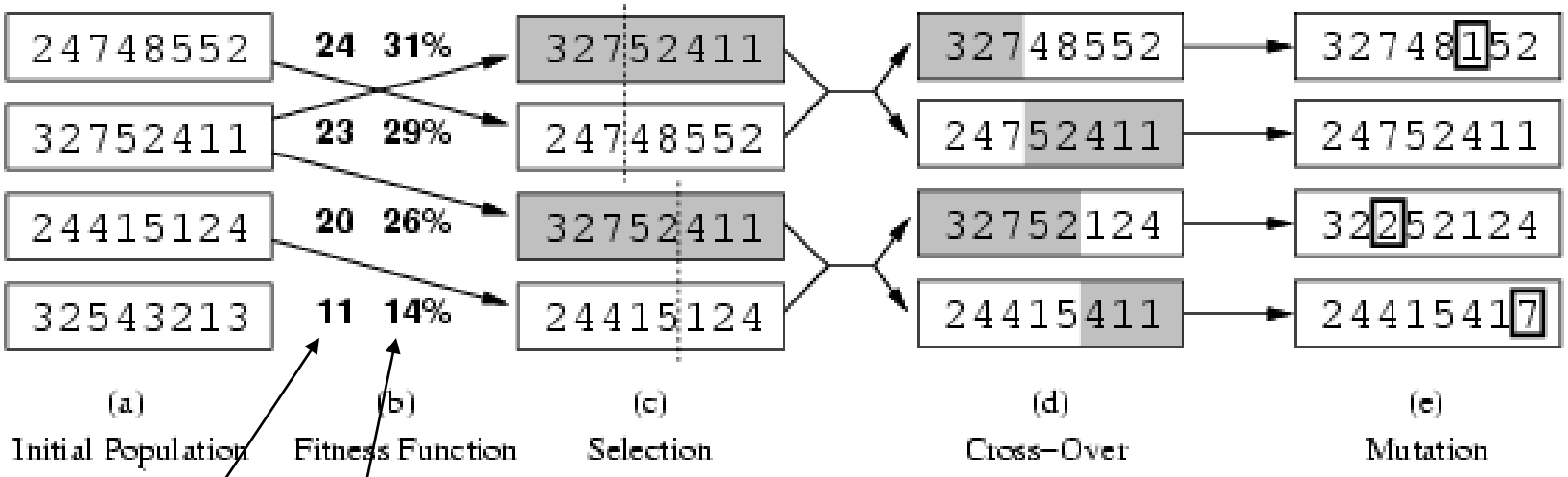
- A simple local search but with a memory.
- Recently visited states are added to a tabu-list and are temporarily excluded from being visited again.
- This way, the solver moves away from already explored regions and (in principle) avoids getting stuck in local minima.

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.

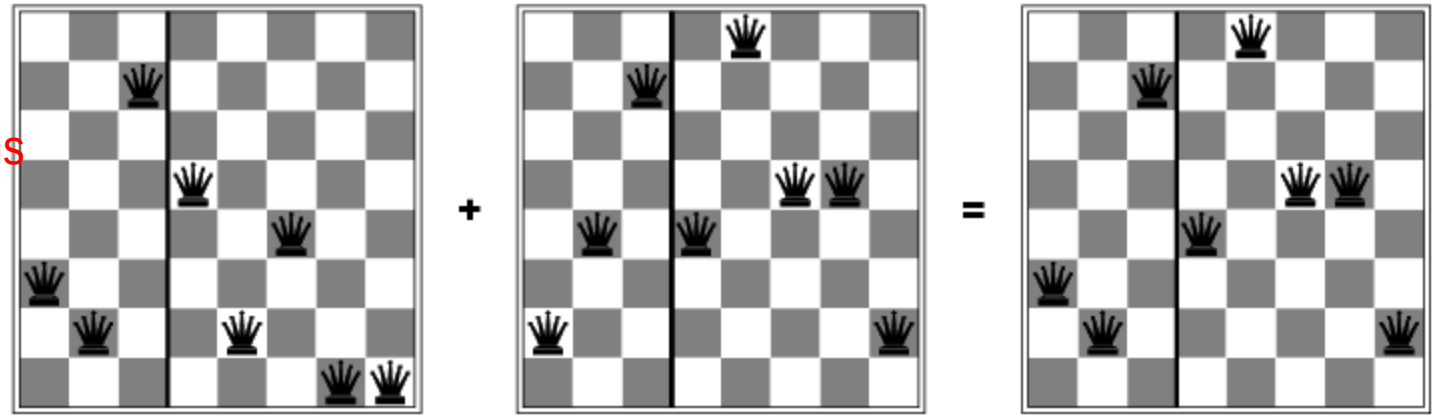
Genetic algorithms

- 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



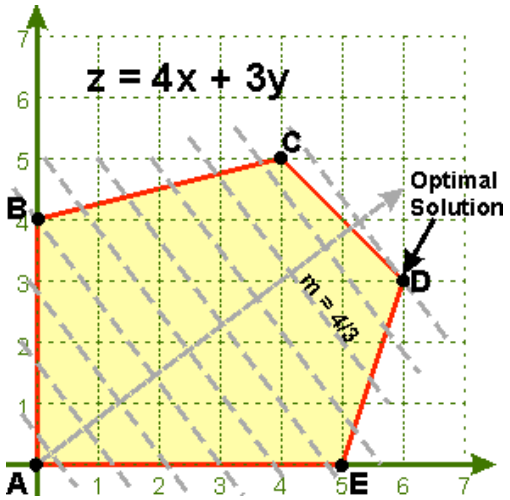
- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- P(child) = 24/(24+23+20+11) = 31%
- P(child) = 23/(24+23+20+11) = 29% etc

Linear Programming

Problems of the sort:

$$\text{maximize } c^T x$$
$$\text{subject to: } Ax \leq b;\ Bx = c$$

- Very efficient “off-the-shelves” solvers are available for LRs.
- They can solve large problems with thousands of variables.



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)

The importance of a good representation

- **Properties of a good representation:**

- Reveals important features
- Hides irrelevant detail
- Exposes useful constraints
- Makes frequent operations easy-to-do
- Supports local inferences from local features
  - Called the “soda straw” principle or “locality” principle
  - Inference from features “through a soda straw”
- Rapidly or efficiently computable
  - It’s nice to be fast

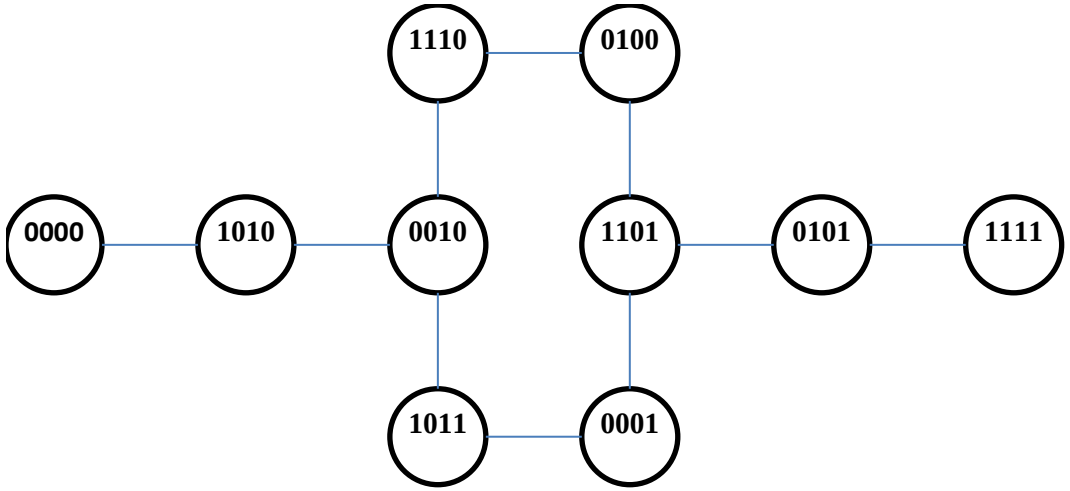
Reveals important features / Hides irrelevant detail

- **"You can't learn what you can't represent."** --- G. Sussman

- **In search:** *A man is traveling to market with a fox, a goose, and a bag of oats. He comes to a river. The only way across the river is a boat that can hold the man and exactly one of the fox, goose or bag of oats. The fox will eat the goose if left alone with it, and the goose will eat the oats if left alone with it.*

  How can the man get all his possessions safely across the river?

- **A good representation makes this problem easy:**



MFGO

M = man
F = fox
G = goose
O = oats
0 = starting side
1 = ending side

Exposes useful constraints

- **“You can’t learn what you can’t represent.”** --- G. Sussman

- **In logic:** *If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.*

  Prove that the unicorn is both magical and horned.

- **A good representation makes this problem easy:**

$$(\neg Y \vee \neg R) \wedge (Y \vee R) \wedge (Y \vee M) \wedge (R \vee H) \wedge (\neg M \vee H) \wedge (\neg H \vee G)$$

Makes frequent operations easy-to-do

Roman numerals

- M=1000, D=500, C=100, L=50, X=10, V=5, I=1
- 2000 = MM; 1776 = MDCCLXXVI

- Long division is **very tedious** (try MDCCLXXVI / XVI)
- Testing for N < 1000 is very easy (first letter is not “M”)

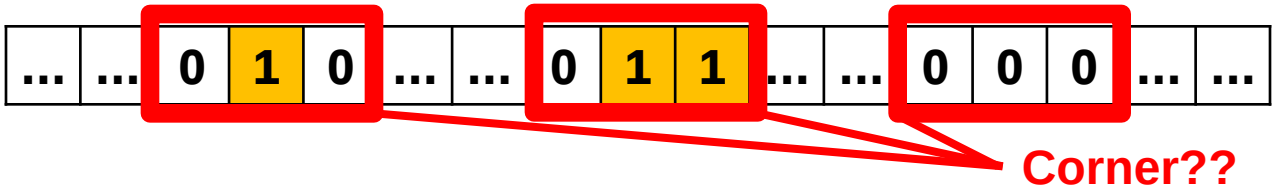
Arabic numerals

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, “.”

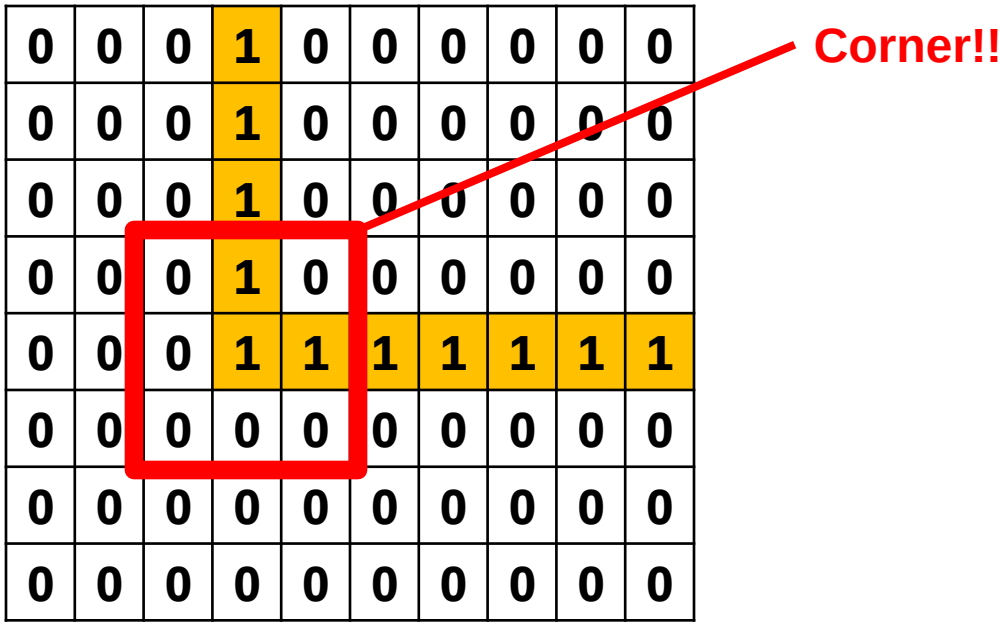
- Long division is **much easier** (try 1776 / 16)
- Testing for N < 1000 is slightly harder (have to scan the string)

Supports local inferences from local features

- Linear vector of pixels = highly non-local inference for vision

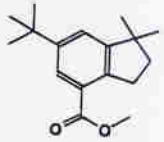


- Rectangular array of pixels = local inference for vision

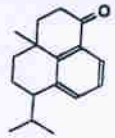


COLLECTION: 026160

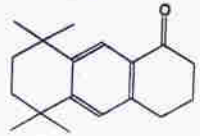
COLLECTION: 019486



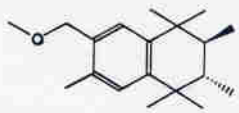
COLLECTION: 004285



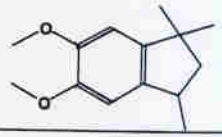
COLLECTION: 019449



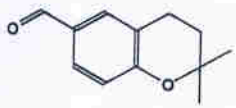
COLLECTION: 025684



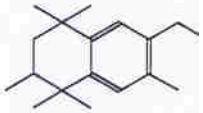
COLLECTION: 021409



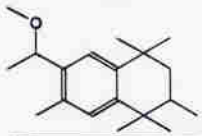
COLLECTION: 021004



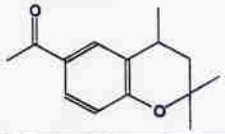
COLLECTION: 019491



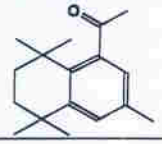
COLLECTION: 025685



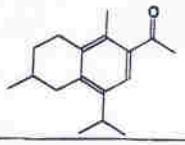
COLLECTION: 020830



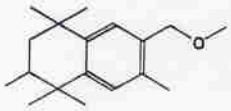
COLLECTION: 021961



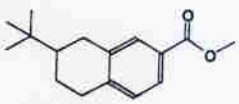
COLLECTION: 012774



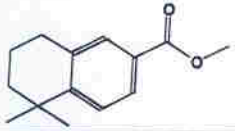
COLLECTION: 019492



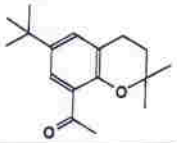
COLLECTION: 022001



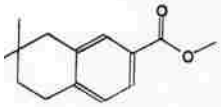
COLLECTION: 021999



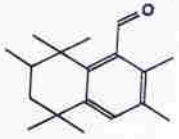
COLLECTION: 019194



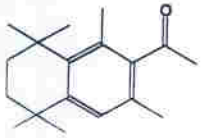
COLLECTION: 022000



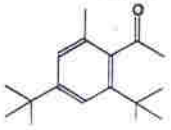
COLLECTION: 021107



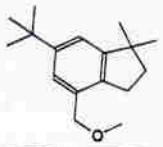
COLLECTION: 021111



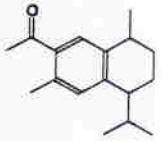
COLLECTION: 019726



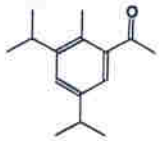
COLLECTION: 019484



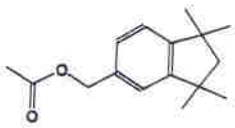
COLLECTION: 012773



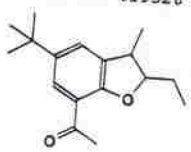
COLLECTION: 020069



COLLECTION: 020074

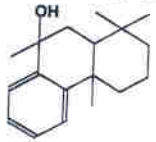


COLLECTION: 019326

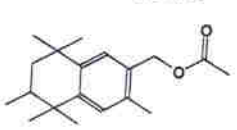


COLLECTION: 020075

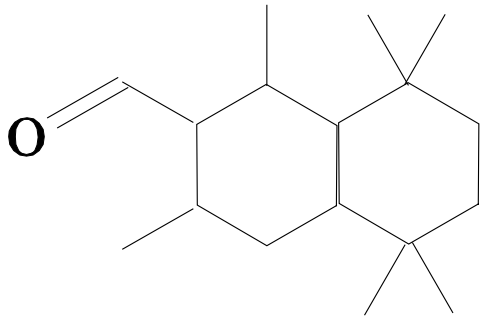
COLLECTION: 019713

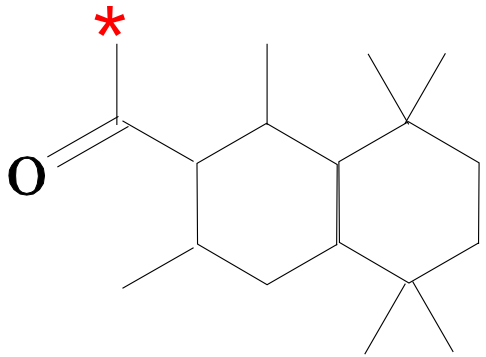


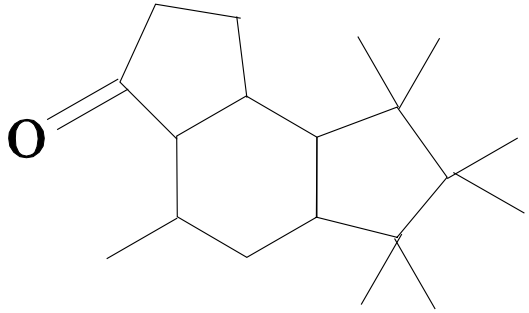
COLLECTION: 019494

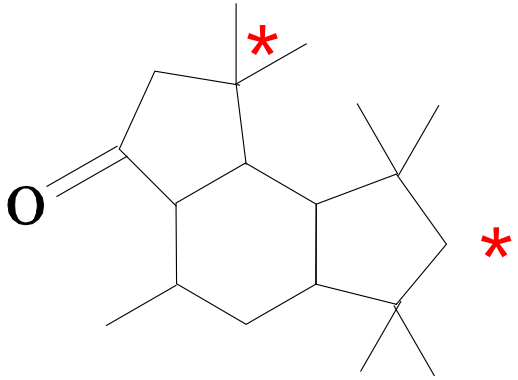


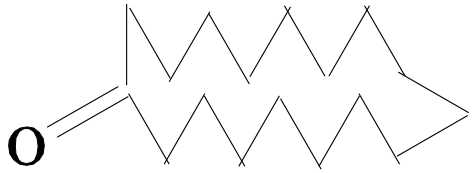
Positive Examples | Negative Examples



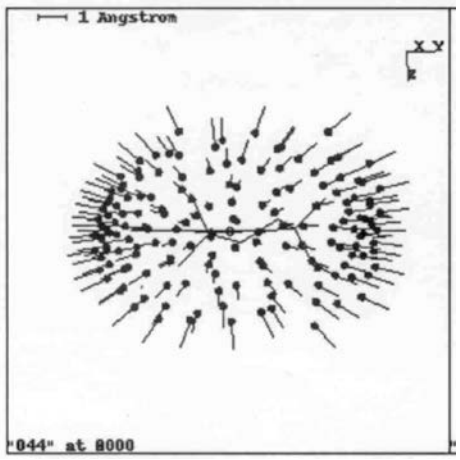


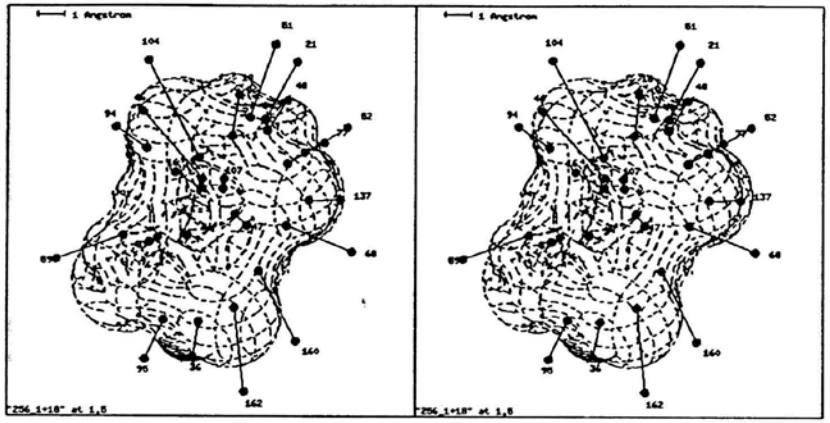


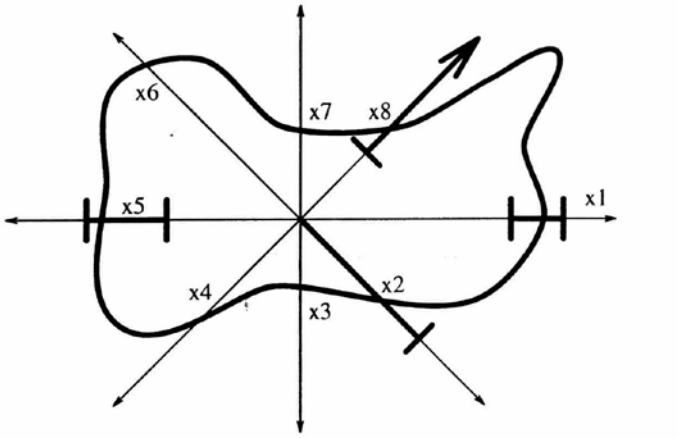


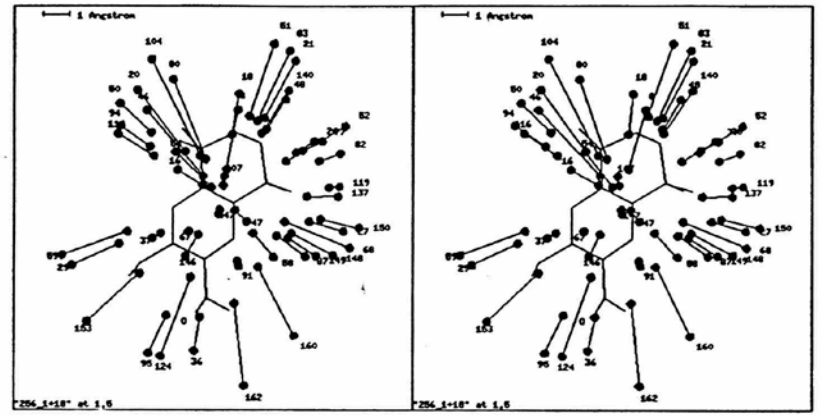


Digital 3D Shape Representation

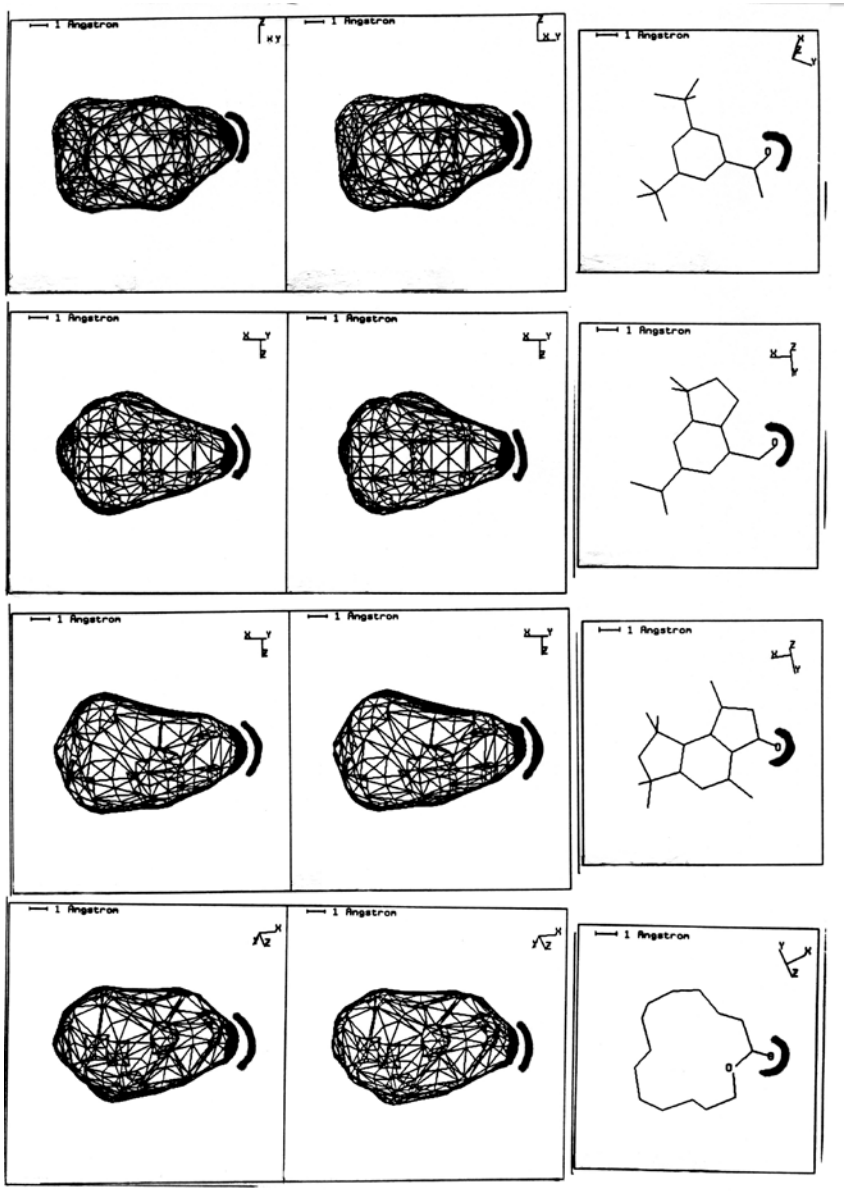




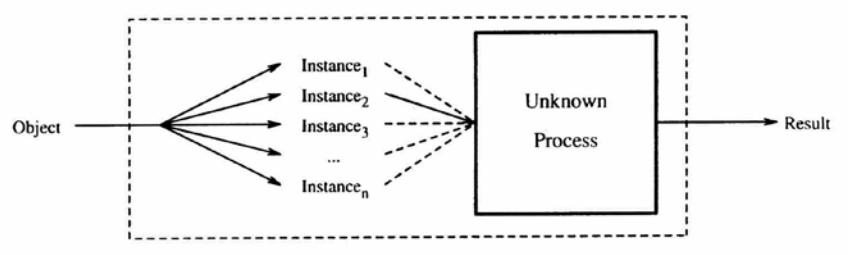


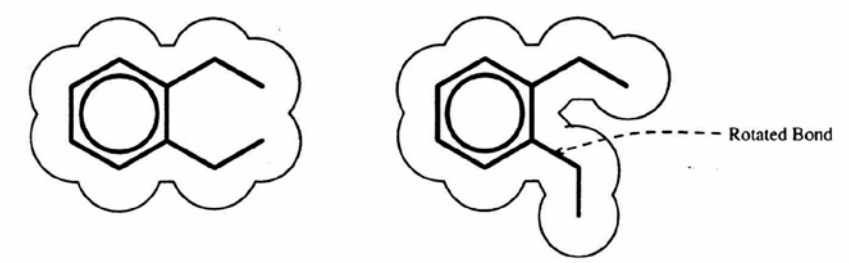


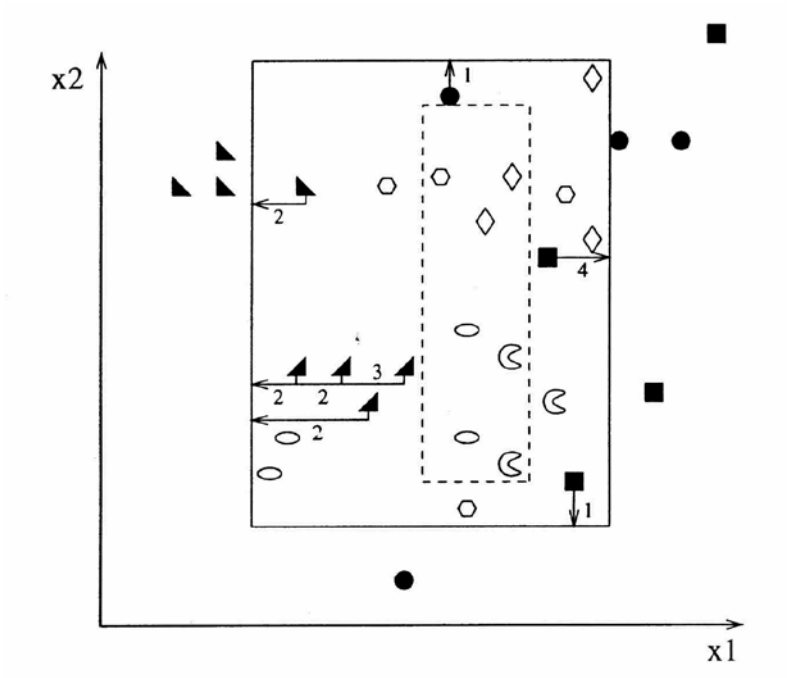
The Power of a Good Representation



Learning the "Multiple Instance" Problem







"Solving the multiple instance problem with axis-parallel rectangles"

Dietterich, Lathrop, Lozano-Perez, Artificial Intelligence 89(1997) 31-71

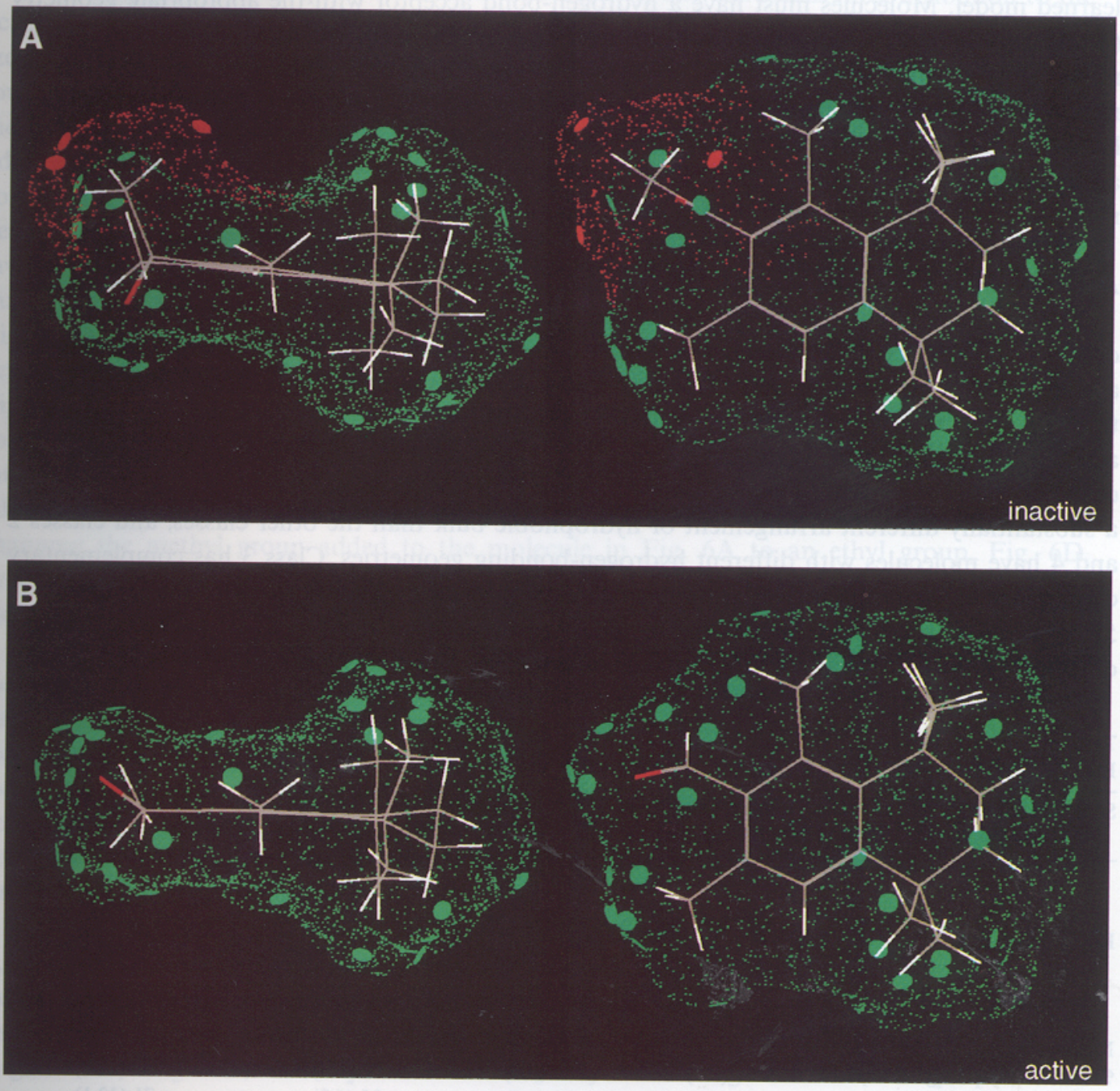


Fig. 7. Remediation of unfavorable steric interaction. (A)

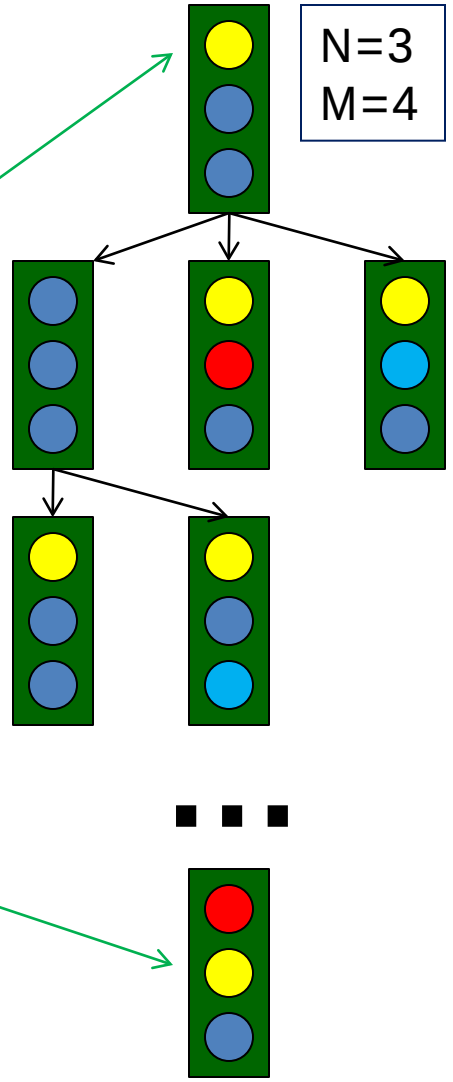
"Compass: A shape-based machine learning tool for drug design," Jain, Dietterich, Lathrop, Chapman, Critchlow, Bauer, Webster, Lozano-Perez,

J. Of Computer-Aided Molecular Design, 8(1994) 635-652

From last class meeting – a "non-distance" heuristic

- The "N Colored Lights" search problem.
  - You have N lights that can change colors.
    - Each light is one of M different colors.
  - Initial state: Each light is a given color.
  - Actions: Change the color of a specific light.
    - You don't know what action changes which light.
    - You don't know to what color the light changes.
    - Not all actions are available in all states.
  - Transition Model: RESULT(s,a) = s'

    where s' differs from s by exactly one light's color.
  - Goal test: A desired color for each light.
- Find: Shortest action sequence to goal.



From last class meeting – a "non-distance" heuristic

- The "N Colored Lights" search problem.
  - Find: Shortest action sequence to goal.
- h(n) = number of lights the wrong color
- f(n) = g(n) + h(n)
  - f(n) = (under-) estimate of total path cost
  - g(n) = path cost so far = number of actions so far
- Is h(n) admissible?
  - Admissible = never overestimates the cost to the goal.
  - Yes, because: (a) each light that is the wrong color must change and (b) only one light changes at each action.
- Is h(n) consistent?
  - Consistent = h(n) ≤ c(n,a,n') + h(n'), for n' a successor of n.
  - Yes, because: (a) c(n,a,n')=1; and (b) h(n) ≤ h(n')+1
- Is A* search with heuristic h(n) optimal?