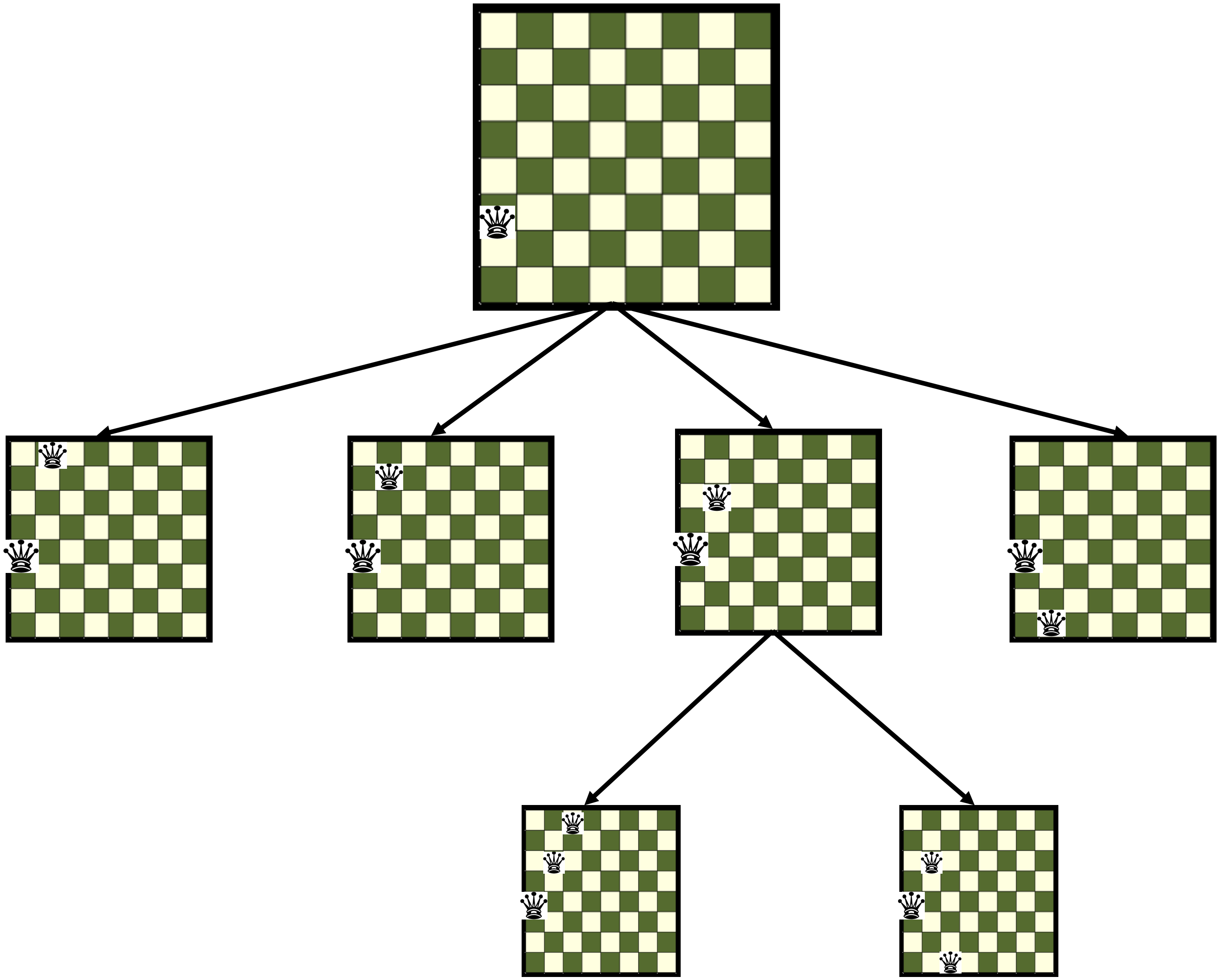
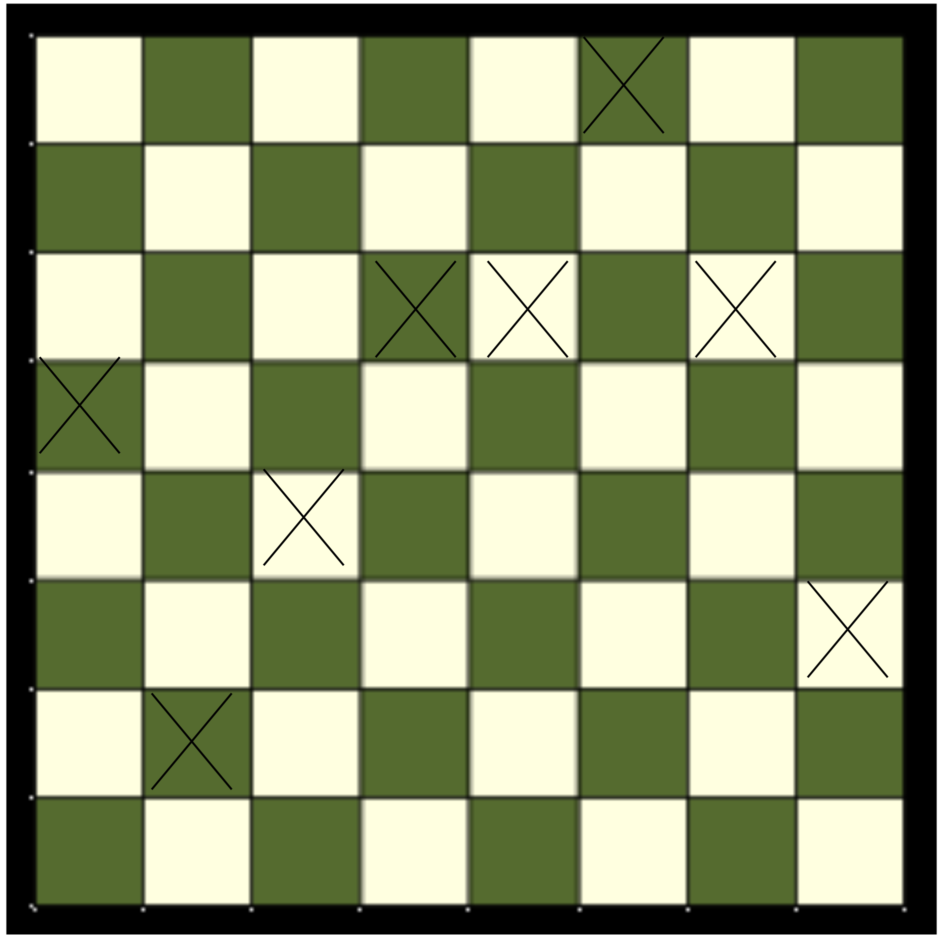
Local Search!

N-Queens problem



Alternative Approach



Random Search

1. Select (random) initial state (initial guess at solution)
2. If not goal state, make local modification to improve current state
3. Repeat Step 2 until goal state found (or out of time)

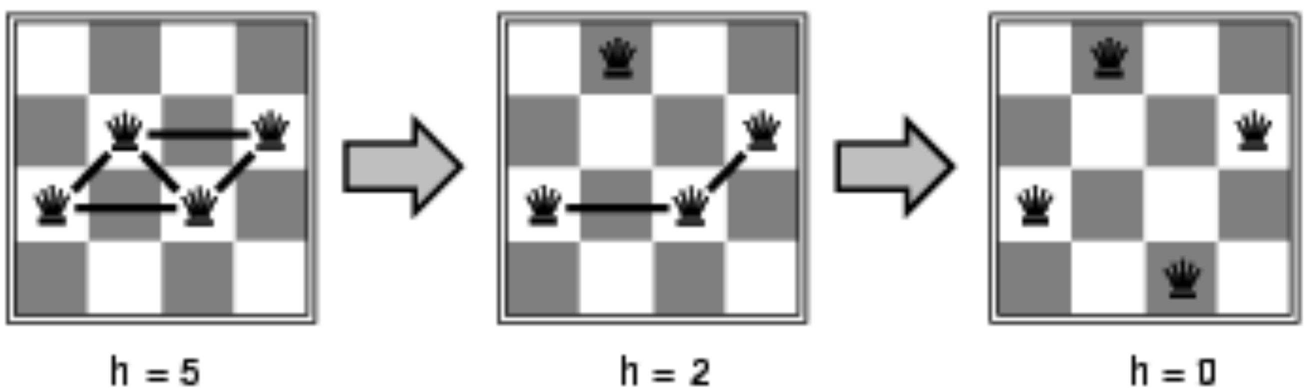
Requirements:

- generate a random (probably-not-optimal) guess
- evaluate quality of guess
- move to other states (well-defined neighborhood function)

. . . and do these operations quickly. . .

Example: 4 Queen

- States: 4 queens in 4 columns
- Operations: move queen in column
- Goal test: no attacks
- Evaluation: h(n) = number of attacks



Example: Graph Coloring

1. Start with random coloring of nodes
2. If not goal state, change color of one node to reduce # of conflicts
3. Repeat 2

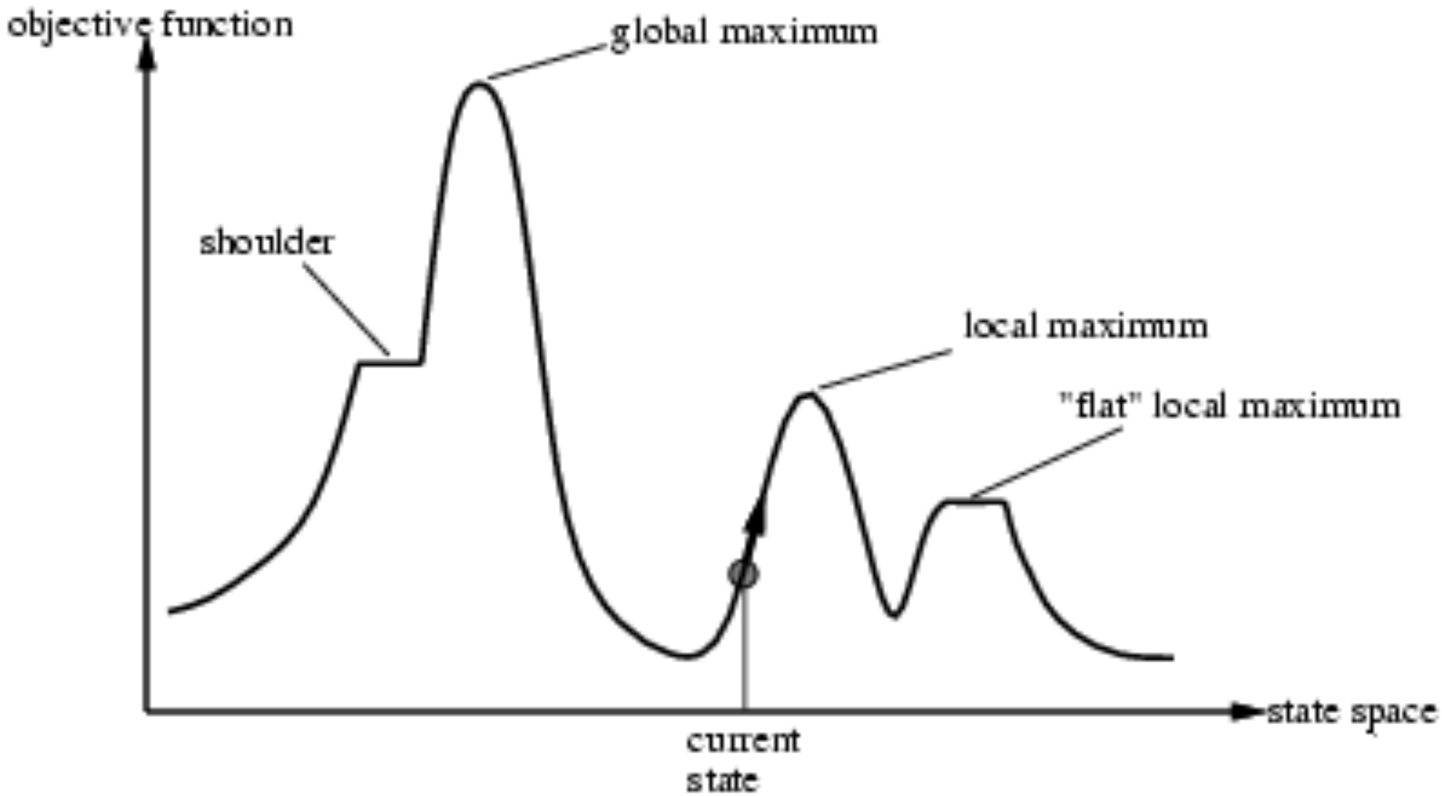
Local Search algorithms

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

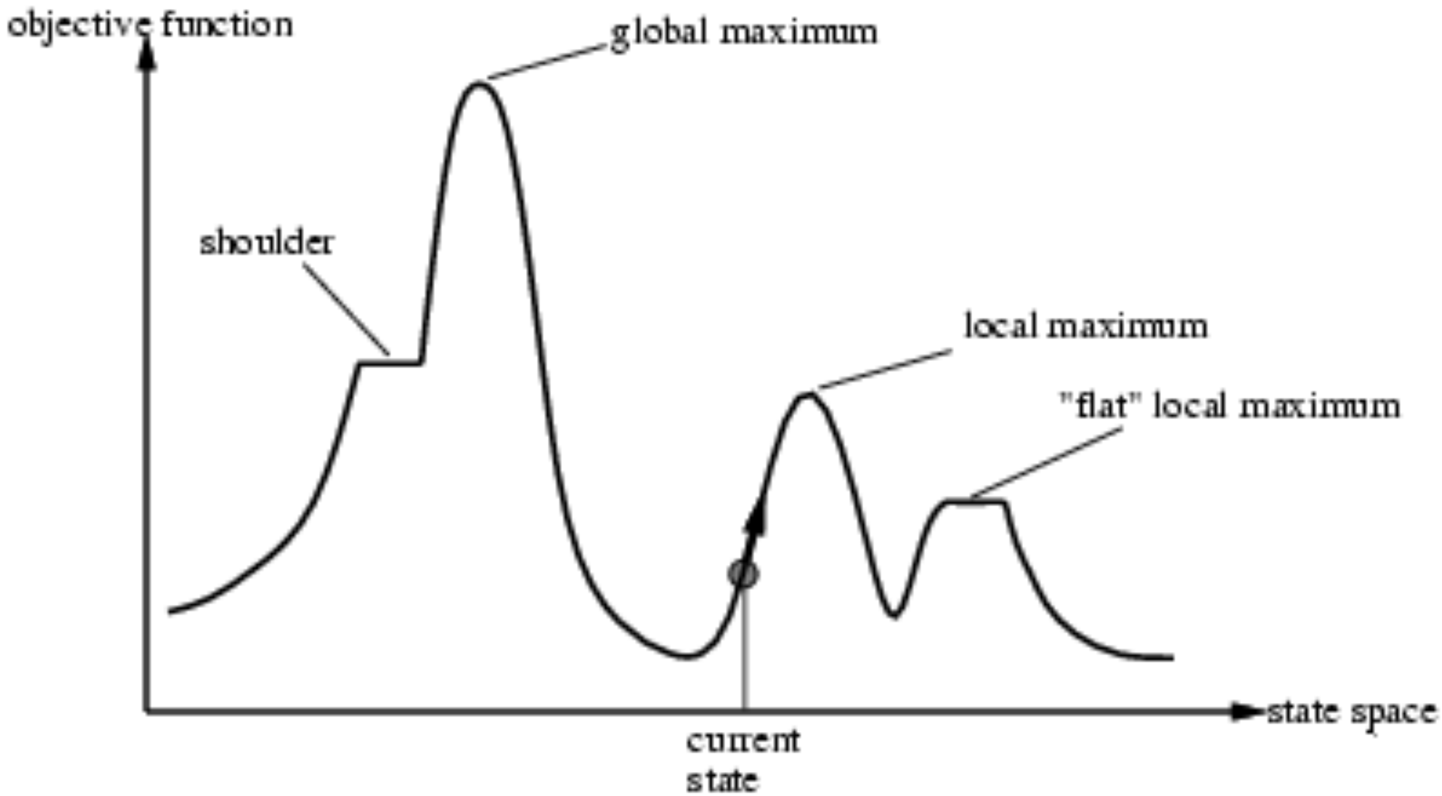
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

•Hill-climbing
•Simulated annealing
•Local Beam Search
•Stochastic Beam Search
•Genetic Algorithms

Local Search Algorithms



Hill-climbing Search



Hill-climbing Search

- "Like climbing Everest in thick fog with amnesia"

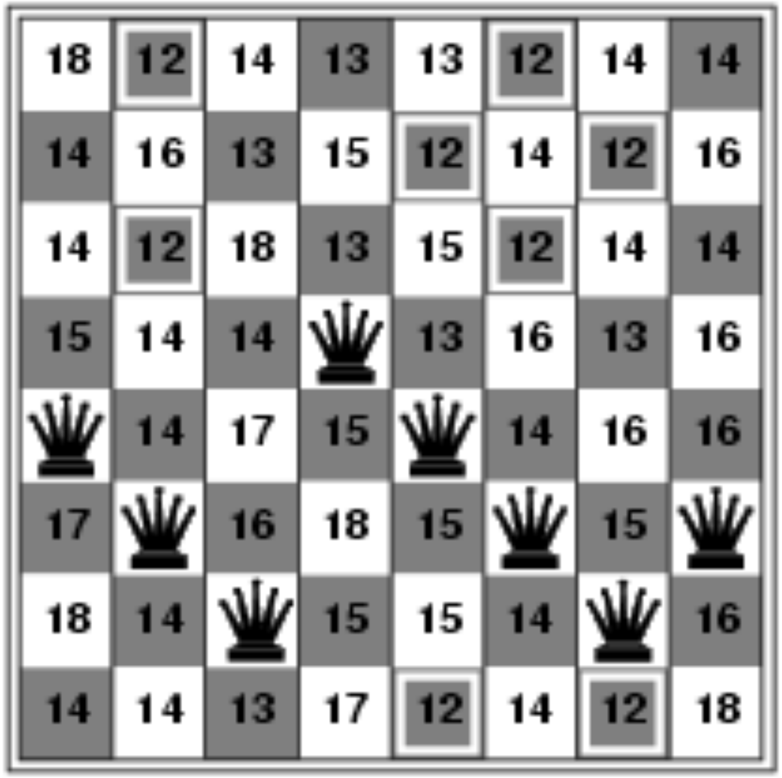
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   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
```

Example: *n*-queens

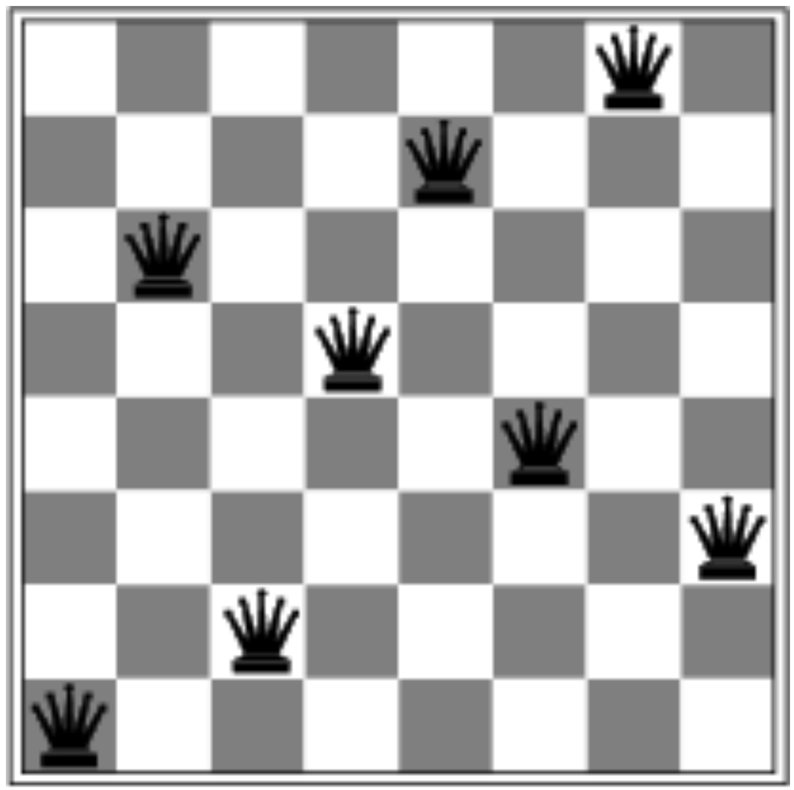
- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Hill-climbing Search: 8-queens problem



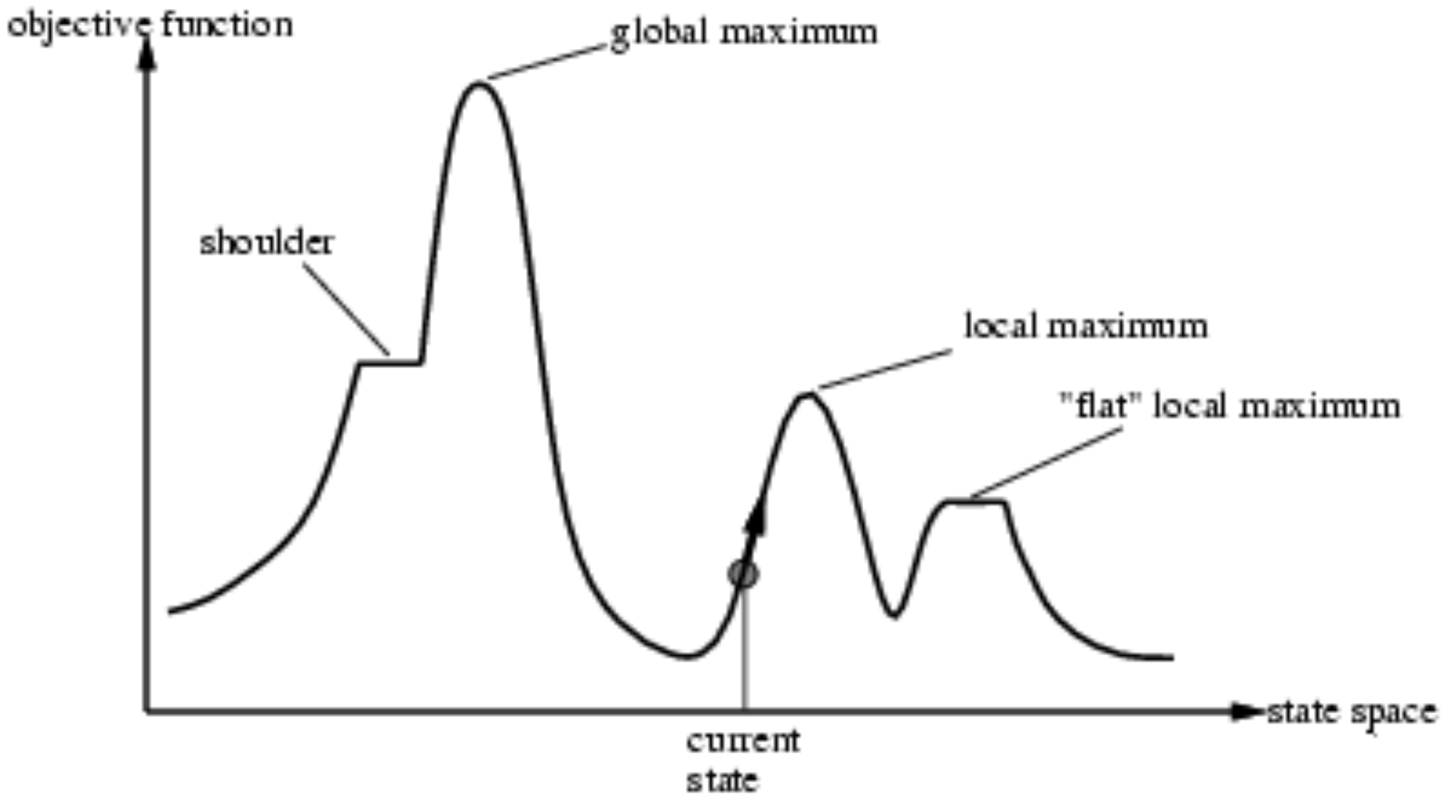
- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h = 17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with *h = 1*

Problems with hill-climbing?



Hill-climbing Performance

- Complete?
- Optimal?
- Time Complexity
- Space Complexity

Hill-climbing Variants

- Stochastic Hill Climbing
- First-choice hill climbing
- Random-restart hill climbing

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)
```

Simulated Annealing Search

- Widely used in:
  - VLSI layout,
  - airline scheduling,
  - Factory layout
  - etc

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.

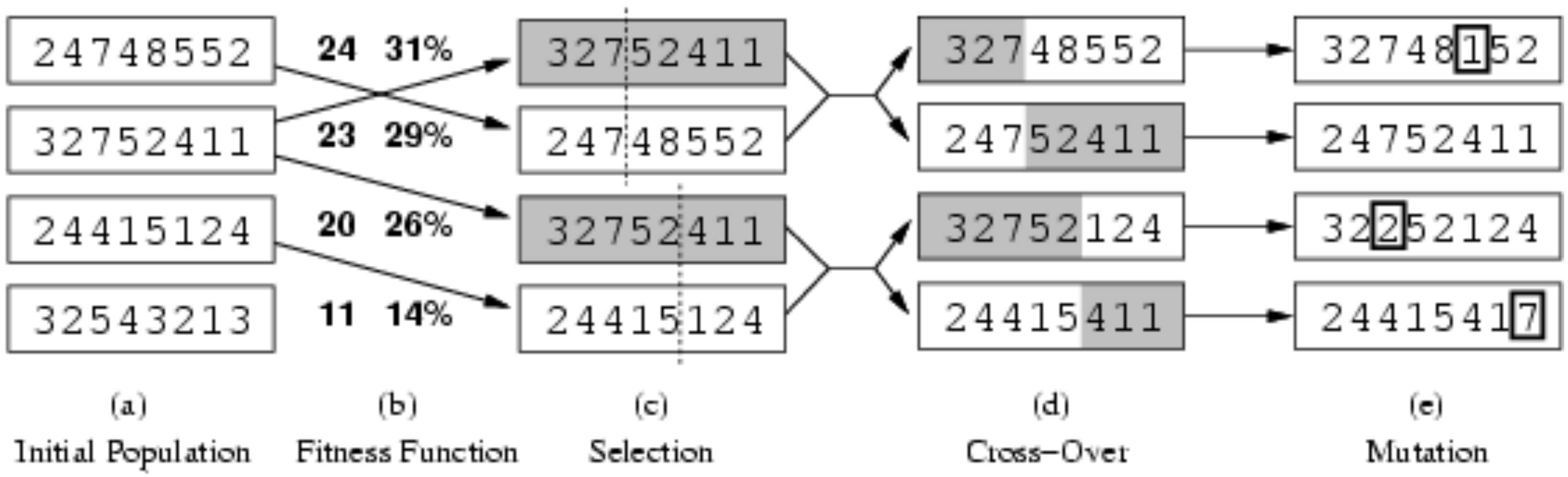
Stochastic Beam Search

- Instead of choosing the *k* best from pool, choose *k* at "random"
- Like natural selection
  - Successors = offspring
  - State = organism
  - Value = fitness

Genetic algorithms

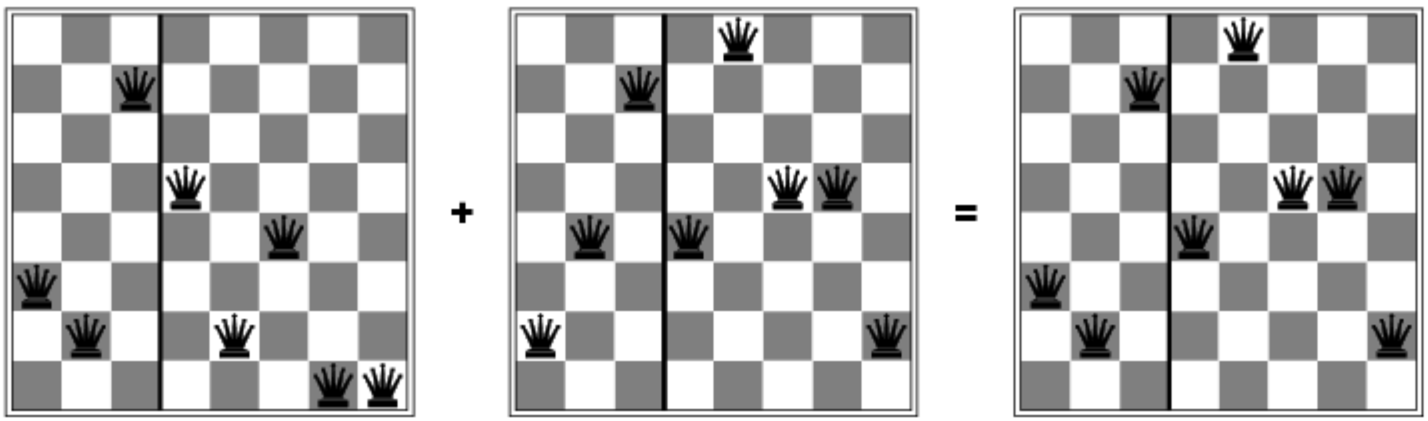
- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce (breed)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

Genetic algorithms



- Fitness function: number of non-attacking pairs of queens
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms



Genetic Algorithms Continued…

1. Choose initial population
2. Evaluate fitness of each in population
3. Repeat the following until we hit a terminating condition:
    1. Select best-ranking to reproduce
    2. Breed using crossover and mutation
    3. Evaluate the fitnesses of the offspring
    4. Replace worst ranked part of population with offspring

Anatomy of a Genetic Algorithm