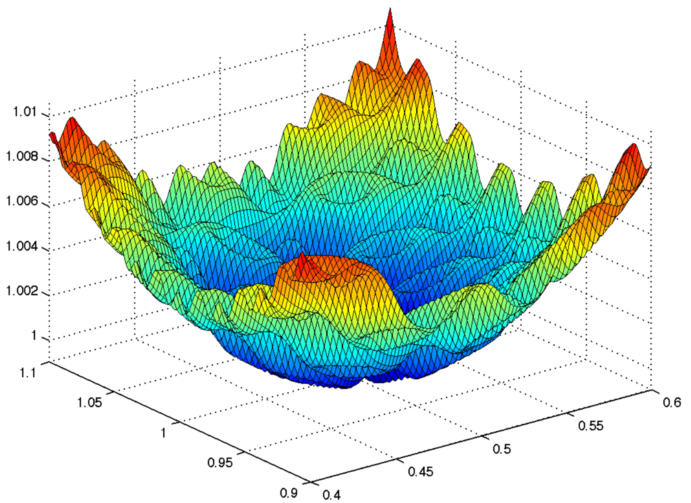
# ARTIFICIAL INTELLIGENCE

Russell & Norvig

Chapter 4: Local Search Algorithms and Optimization Problems

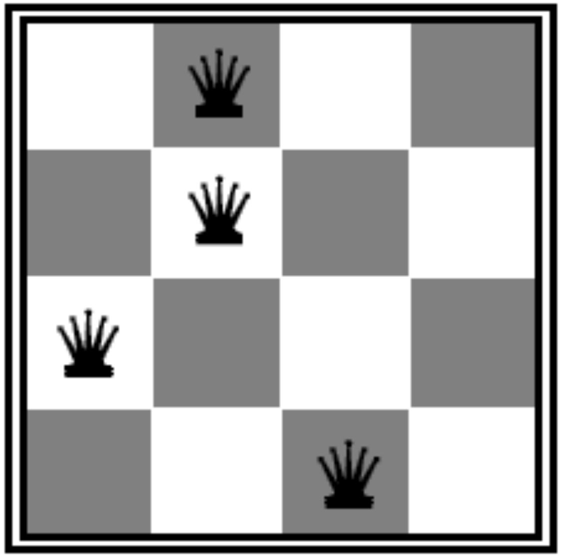
# Local search algorithms

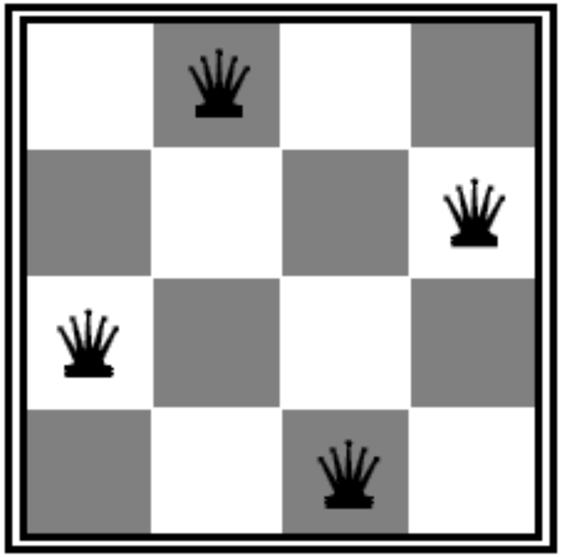
- Some types of search problems can be formulated in terms of **optimization**
  - We don't have a start state, don't care about the path to a solution
  - We have an **objective function** that tells us about the quality of a possible solution, and we want to find a good solution by minimizing or maximizing the value of this function



# Example: *n*-queens problem

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- **State space:** all possible $n$-queen configurations
- What's the **objective function**?
  - Number of pairwise conflicts

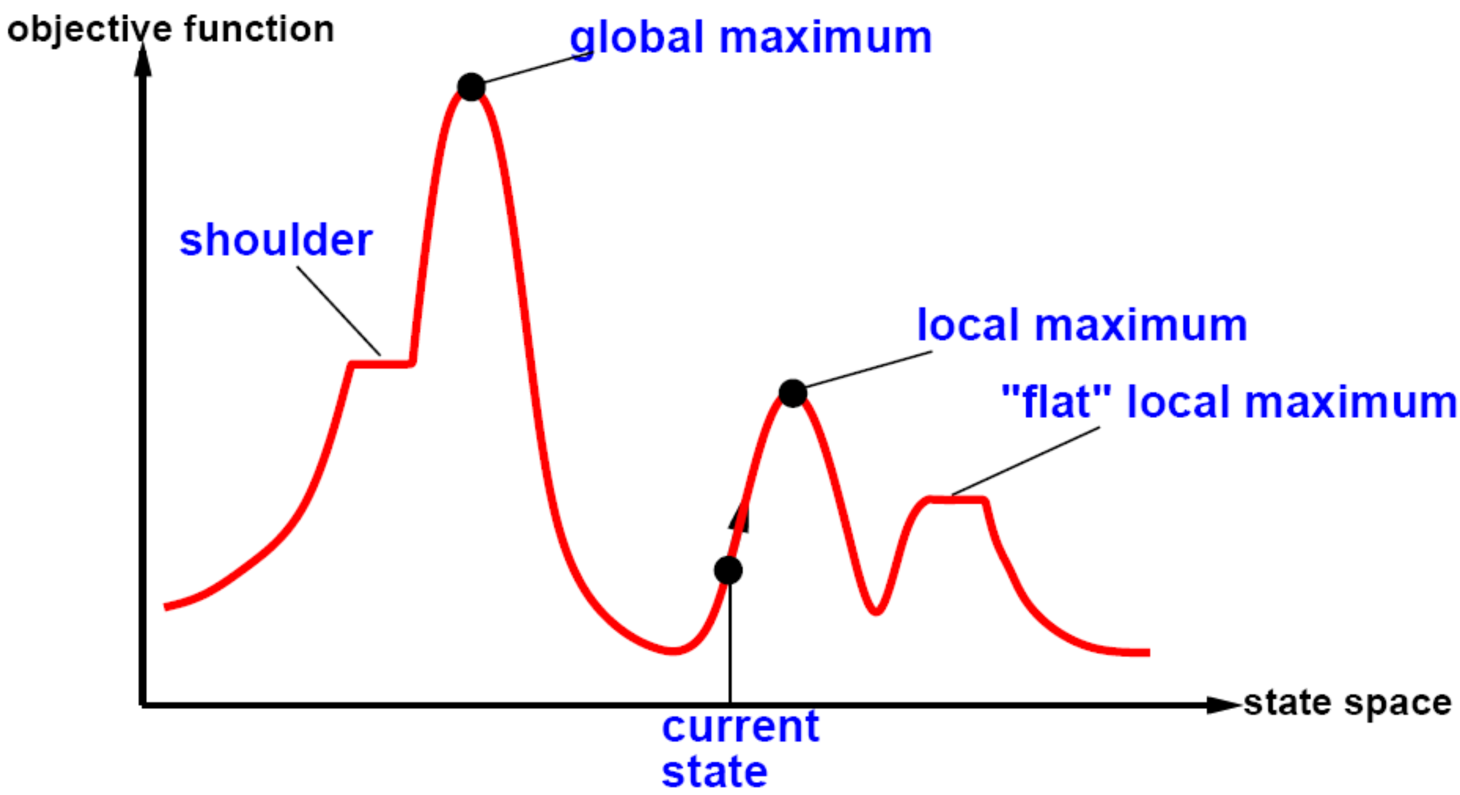




# Hill-climbing (greedy) search

- Idea: keep a single “current” state and try to locally improve it
- “Like climbing mount Everest in thick fog with amnesia”

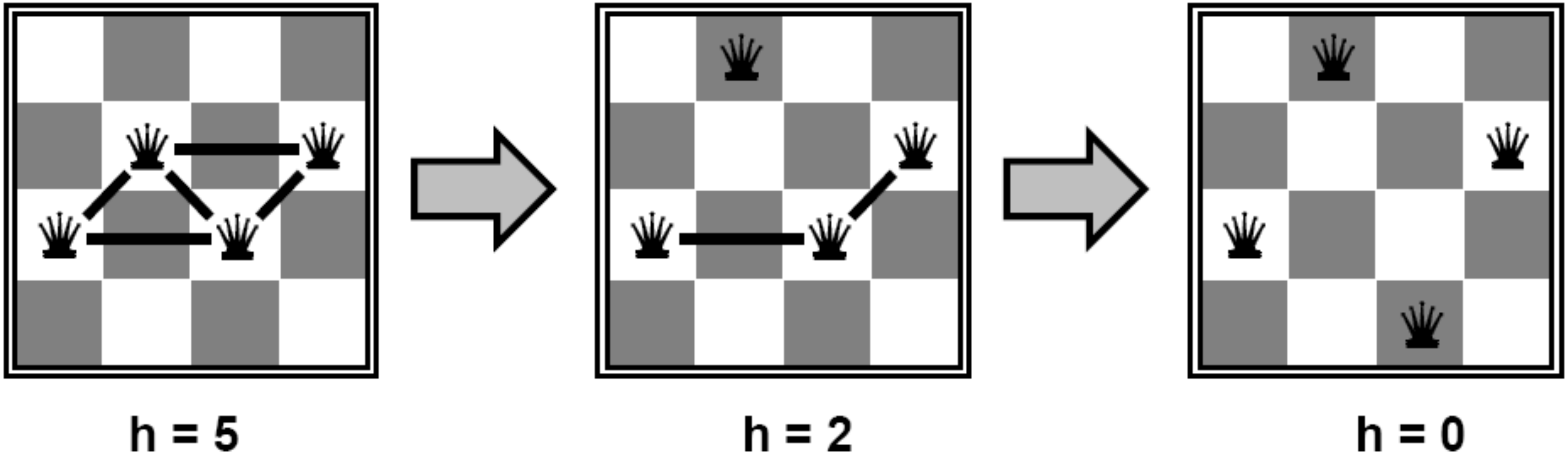
# The state space "landscape"




- How to escape local maxima (minima)?
  - Random restart hill-climbing
- What about "shoulders"?
- What about "plateaus"?

# Example: *n*-queens problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- **State space:** all possible *n*-queen configurations
- **Objective function:** number of pairwise conflicts
- What's a possible local improvement strategy?
  - Move one queen within its column to reduce conflicts



# Example: *n*-queens problem (cont'd)


$h = 17$

# Hill-climbing (greedy) search

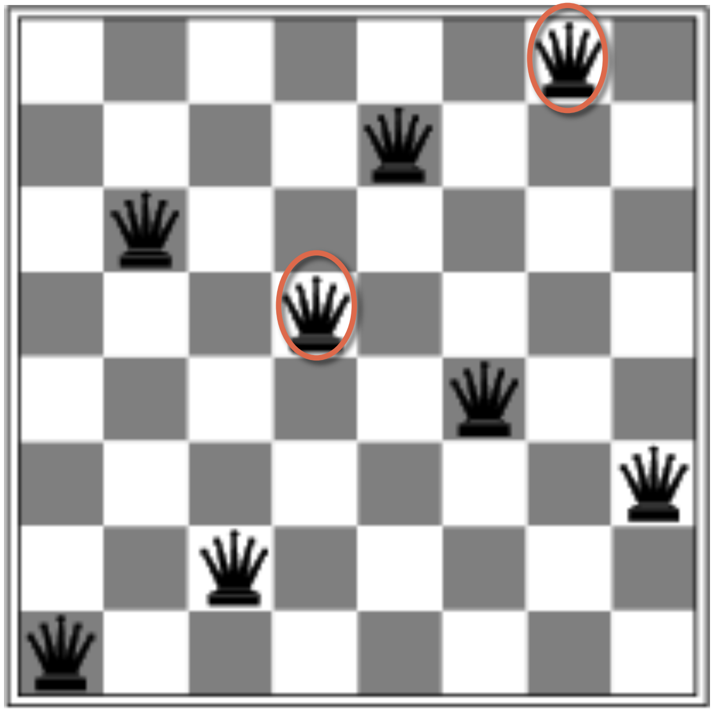
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

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

- Variants: choose first better successor, randomly choose among better successors
- Variants to avoid local maxima, plateaus, shoulders, ridges, etc.

# Hill-climbing search

- Is it complete/optimal?
  - No – can get stuck in local optima
  - Example: local optimum for the 8-queens problem



# Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
  - Probability of taking downhill move decreases with number of iterations, steepness of downhill move
  - Controlled by *annealing schedule*
- Inspired by tempering of glass, metal

# Simulated annealing search

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

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

# Simulated annealing search

- If temperature decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching one.

- However:
  - This usually takes impractically long
  - The more downhill steps you need to escape a local optimum, the less likely you are to make all of them in a row

# Local beam search

Start with *k* randomly generated states

Repeat

  Generate all the successors of all *k* states

  If a goal state is generated, stop

  Else select the *k* best successors from the complete list

Until some stopping condition

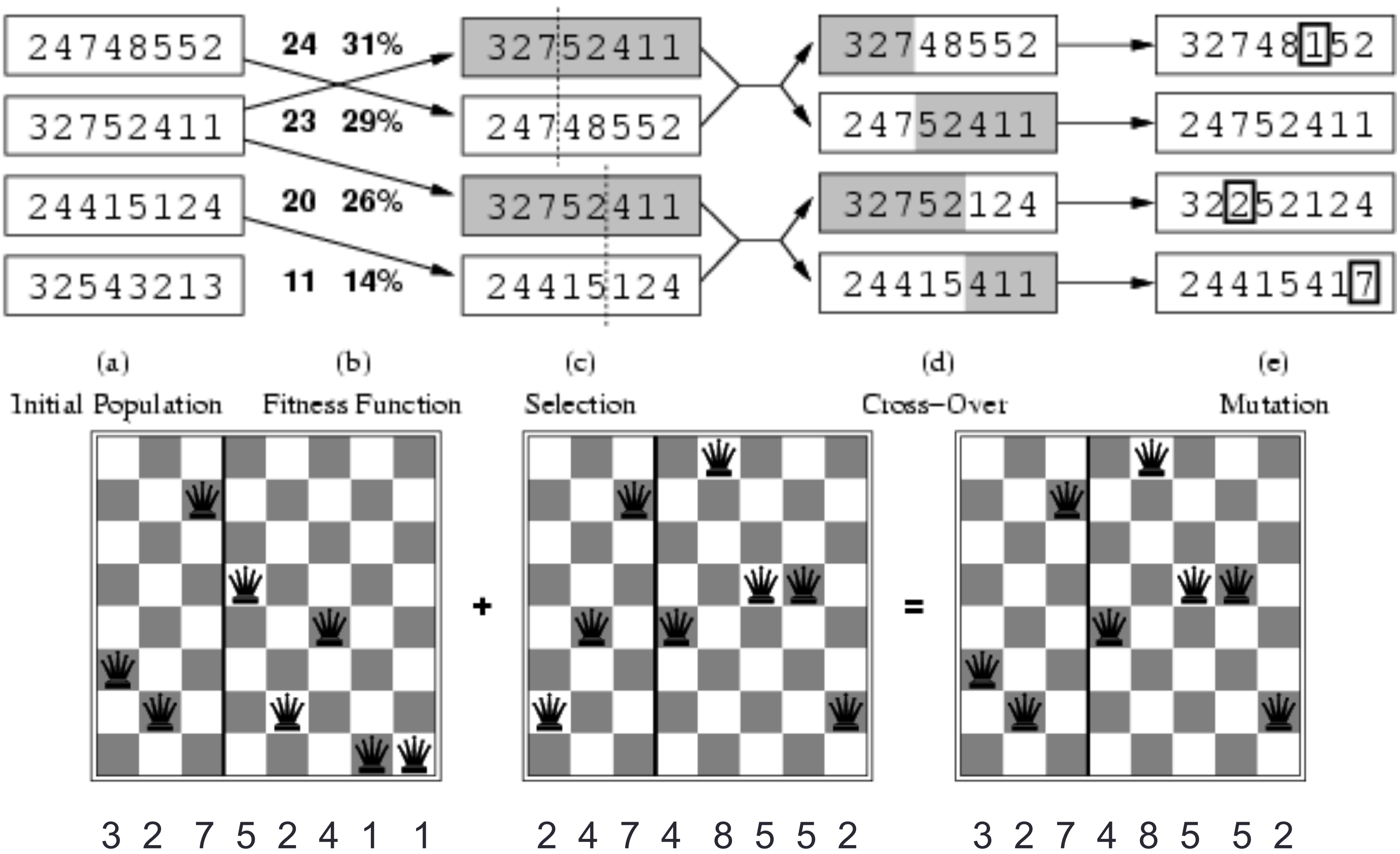
- Better than running *k* greedy searches in parallel.

- Stochastic beam search chooses *k* successors at random, proportional to the “goodness” of the state.

# Genetic algorithms (GA)

- Variant of stochastic beam search, inspired by “natural selection”
- 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

# Genetic algorithms



# Genetic algorithms

**function** GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  **inputs**: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  **repeat**
    *new_population* ← empty set
    **for** $i$ = 1 **to** SIZE(*population*) **do**
      $x$ ← RANDOM-SELECTION(*population*, FITNESS-FN)
      $y$ ← RANDOM-SELECTION(*population*, FITNESS-FN)
      *child* ← REPRODUCE($x$, $y$)
      **if** (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    *population* ← *new_population*
  **until** some individual is fit enough, or enough time has elapsed
  **return** the best individual in *population*, according to FITNESS-FN

---

**function** REPRODUCE($x$, $y$) **returns** an individual
  **inputs**: $x$, $y$, parent individuals

  $n$ ← LENGTH($x$); $c$ ← random number from 1 to $n$
  **return** APPEND(SUBSTRING($x$, 1, $c$), SUBSTRING($y$, $c+1$, $n$))