Last update: September 18, 2012

Local search algorithms

CMSC 421: Chapter 4, Sections 1 and 2

Iterative improvement algorithms

♢ In many optimization problems, the **path** to a goal is irrelevant
- the goal state itself is the solution

♢ State space = a set of goal states
- find one that satisfies constraints (e.g., no two classes at same time)
- or find **optimal** one (e.g., highest possible value, least possible cost)

♢ *Iterative improvement* algorithms
  - ◇ keep a single "current" state, try to improve it
- Constant space
- Suitable for both offline and online search

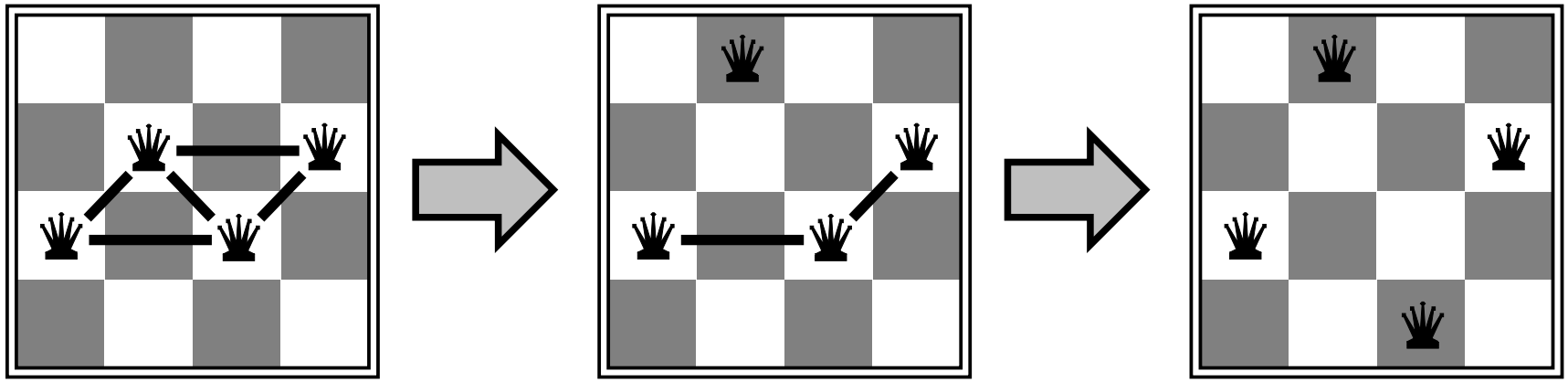
Example: the n-Queens Problem

♢ Put n queens on an $n \times n$ chessboard

- No two queens on the same row, column, or diagonal

♢ Iterative improvement:

- Start with one queen in each column
- move a queen to reduce number of conflicts



♢ Even for very large n (e.g., $n = 1\ million$), this usually finds a solution almost instantly

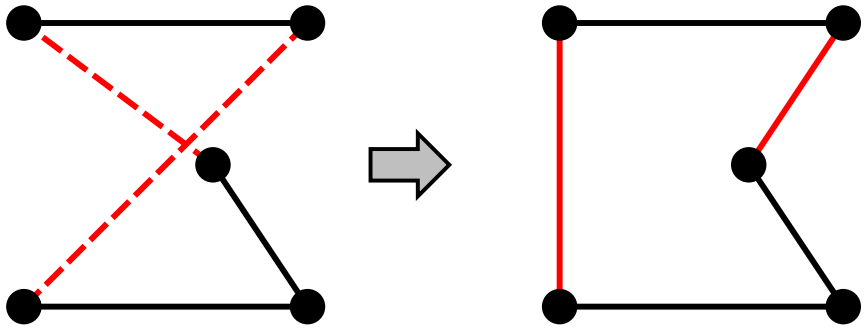
Example: Traveling Salesperson Problem

♢ Given a *complete* graph (edges between all pairs of nodes)

♢ Find a least-cost *tour* (simple cycle that visits each city exactly once)

♢ Iterative improvement:

- Start with any tour, perform pairwise exchanges



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

Outline

- ♢ Hill-climbing
- ♢ Simulated annealing
- ♢ Genetic algorithms (briefly)
- ♢ Local search in continuous spaces (very briefly)

Hill-climbing (or gradient ascent/descent)

♢ "Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem)
    current ← new node containing problem's initial state
    loop
        next ← a highest-valued neighbor of current
        if VALUE[next] ≤ VALUE[current] then return STATE[current]
        current ← next
    end
```

♢ VALUE$[x]$ is x's objective-function value

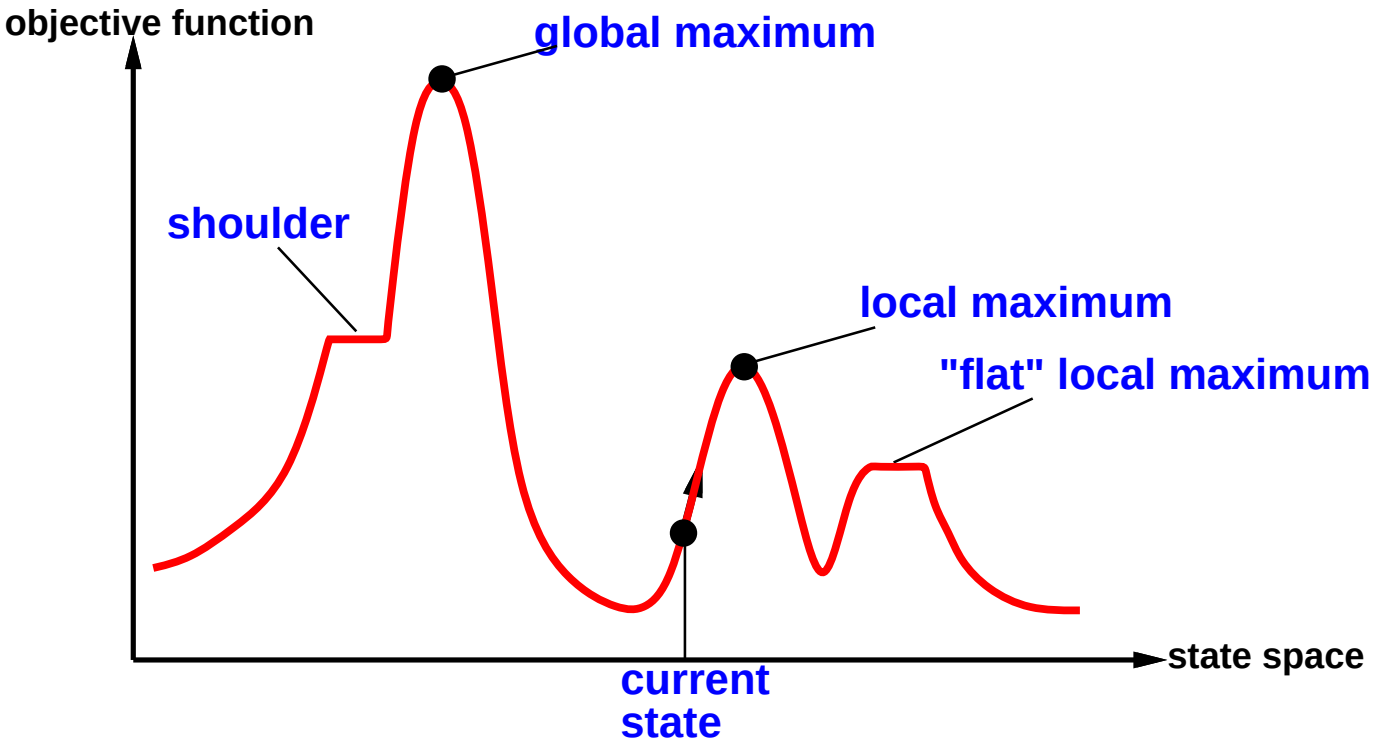
- how good we consider x to be

♢ At each step, move to a neighbor of higher value
in hopes of getting to a solution having the highest possible value

♢ Can easily modify this for problems where we want to
minimize rather than maximize

Hill-climbing, continued

♢ State space "landscape":



♢ *Random-restart hill climbing*:

- repeat with randomly chosen starting points

♢ If finitely many local maxima, then $\lim_{\text{restarts}\to\infty} P(\text{complete}) = 1$

Simulated annealing

```
function SIMULATED-ANNEALING( problem, temperatures)
    node ← a new node containing problem's initial state
    for i ← 1 to ∞ do
        T ← temperatures[i]
        if T = 0 then return STATE[node]
        next ← a randomly selected neighbor of node
        ΔE ← VALUE[next] – VALUE[node]      /* difference in desirability */
        if ΔE > 0 then node ← next
        else with probability e^(ΔE/T), set node ← next
```

♢ Idea: escape local maxima by allowing some "bad" moves

- but gradually decrease their frequency

♢ Next two slides: a simple example, in Python

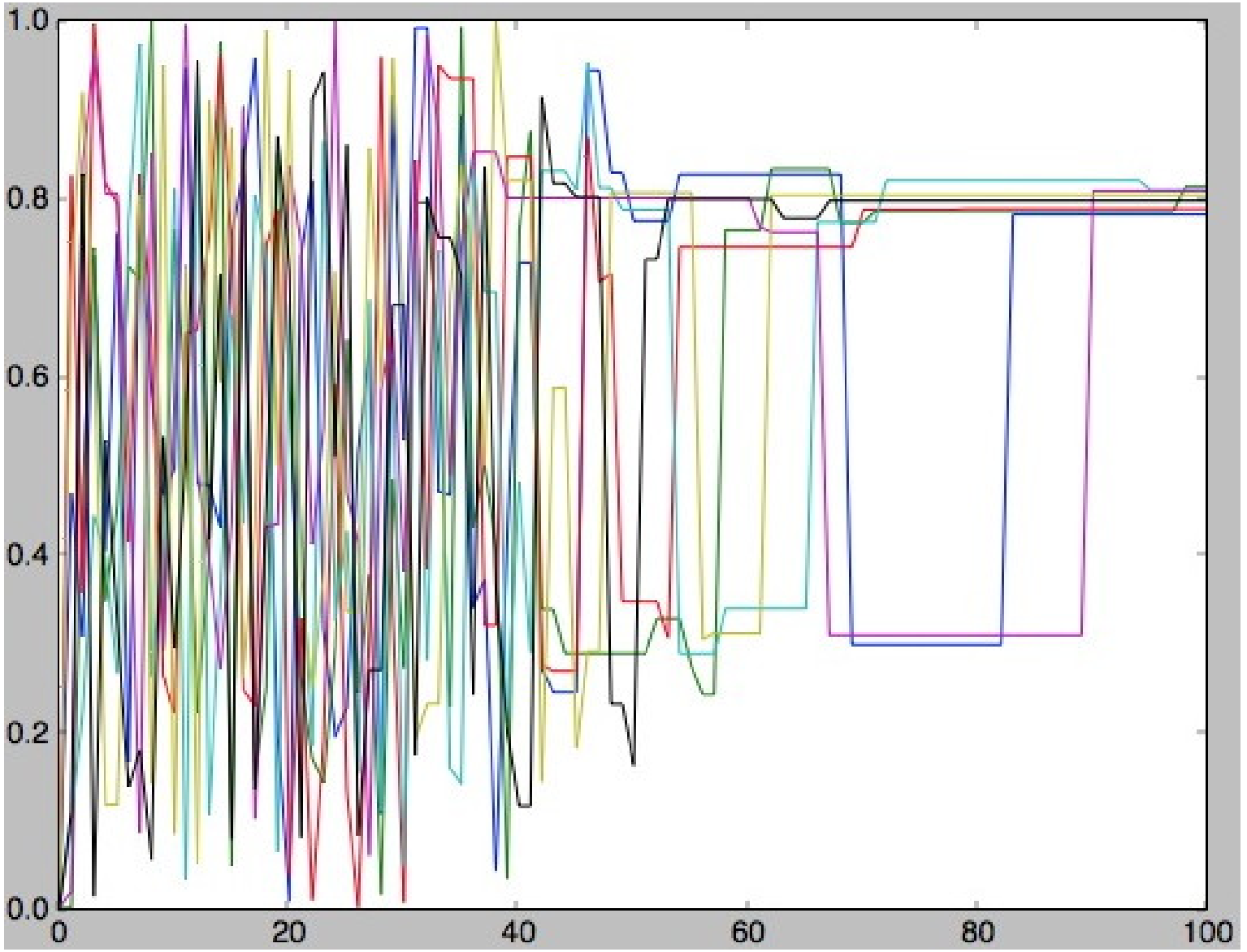
```
import cmath, random
def anneal_example(iterations):
    state = 0; history = [state]
    for i in range(iterations):
        T = 10 * (0.9 ** i)
        next = random.random()
        DeltaE = value(next) - value(state)
        target = cmath.e ** (DeltaE/T):
        if target > random.random():
            state = next
        history.append(round(state,3))
    return history

def value(state):
    return -abs(0.5-state)
```

function ANNEAL_EXAMPLE(*iterations*)
  state ← 0; *history* ← list containing *state*
  for *i* ← 0 **to** *iterations* − 1 **do**
    $T \leftarrow 10 \times 0.9^i$
    next ← a random number in [0,1)
    $\Delta E \leftarrow$ VALUE(*next*) – VALUE(*state*)
    $target \leftarrow e^{\Delta E/T}$
    if *target* > random no. in [0,1) **then**
      state ← *next*
    append *state* to *history*
  return *history*

function VALUE(*state*)
  return $-|0.5 - state|$

- ♢ A state is a number $s \in [0, 1]$. All states are neighbors
- ♢ Desirability = max{–(dist. from 0.3), .025 – (dist. from .8)}
- ♢ Start with *state* = 0; iterate for $i = 0$ to *iterations*–1
- ♢ On i'th iteration, temperature is 10×0.9^i
  - 10.0, 9.00, 8.10, 7.29, 6.56, 5.90, 5.31, 4.78, 4.30, 3.87, . . .

Simple example, continued



Properties of simulated annealing

♢ At fixed "temperature" T, probability of being in any given state x approaches a Boltzman distribution

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

♢ For every state x other than x^* and for small T,

$$p(x^*)/p(x) = e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$$

♢ It can be shown that if we decrease T slowly enough,

- Pr[reach x^*] approaches 1

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

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

Local beam search

```
function BEAM-SEARCH( problem, k) returns a solution state
    start with k randomly generated states
    loop
        generate all successors of all k states
        if any of them is a solution then return it
        else select the k best successors
```

♢ Not the same as k parallel searches

- Searches that find good states will recruit other searches to join them
- Problem: often all k states end up on same local hill

♢ Stochastic beam search:

- Choose k successors randomly, biased towards good ones
- Close analogy to natural selection

Genetic algorithms

♢ Genetic algorithms

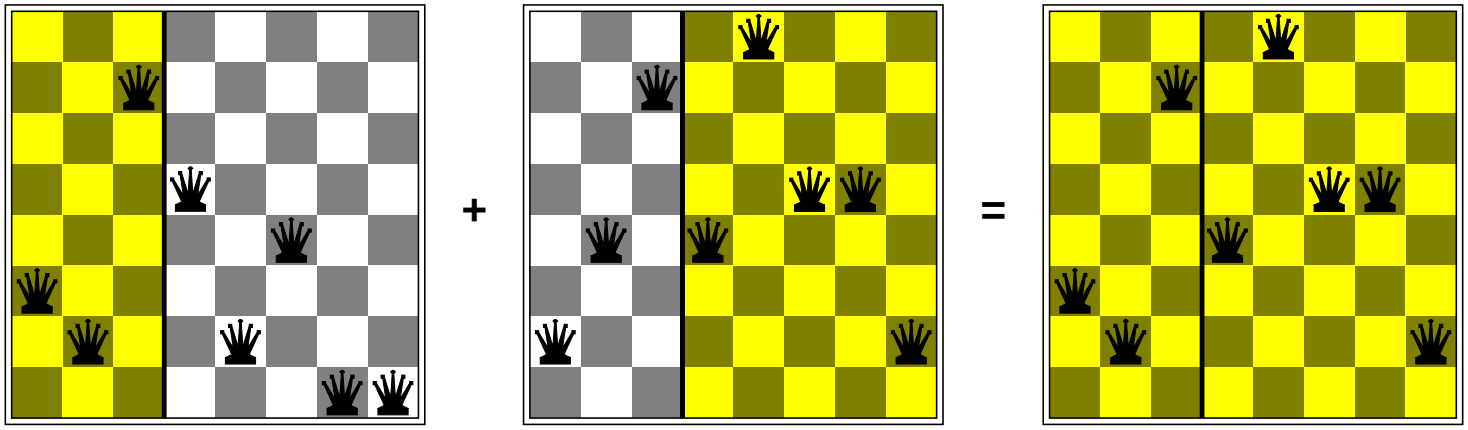
- ◇ stochastic local beam search
- ◇ generate successors from **pairs** of states

• Each state should be a string of characters

• Substrings should be meaningful components

♢ **Example**: n-queens problem

• i'th character = row where i'th queen is located



Genetic algorithms

function GENETIC-ALGORITHM(*population*, FITNESS)

  loop

    new-population ← empty set

    for $i = 1$ to $|$*population*$|$

      choose x and y randomly* from *population*, using FITNESS

      child = REPRODUCE(x,y)

      if small random probability **then** *child* = MUTATE(*child*)

      add child to *new-population*

    population ←*new-population*

    if some individual is fit enough, or enough time has elapsed **then**

      return arg max $\{$FITNESS(x)$|x \in$ *population*$\}$

---

function REPRODUCE(x, y)

  c ←random integer in `range(len(`x`))`

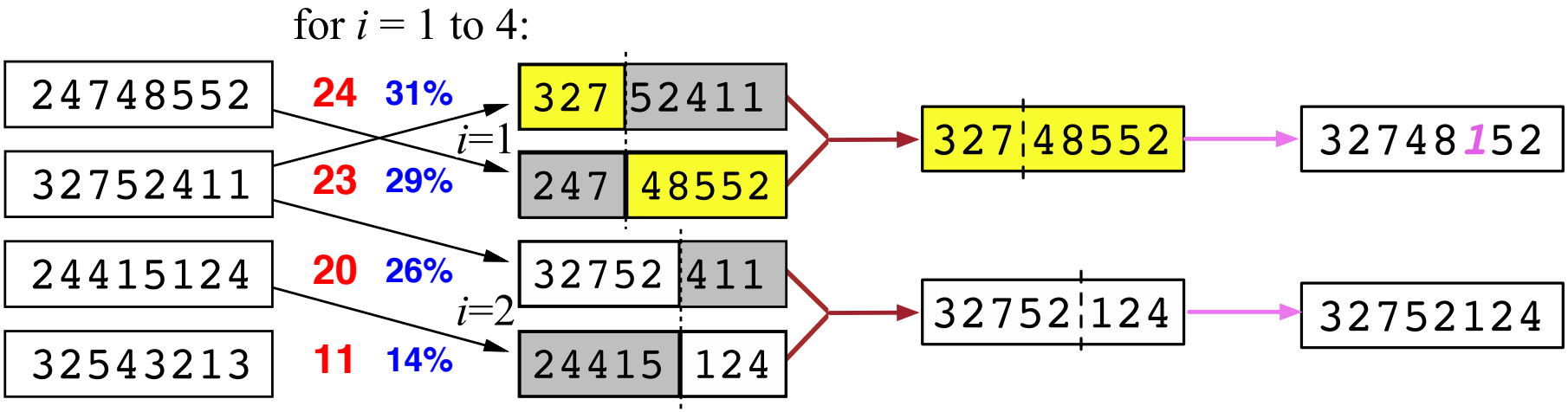
  return x`[:`c`] + `y`[`c`:]`

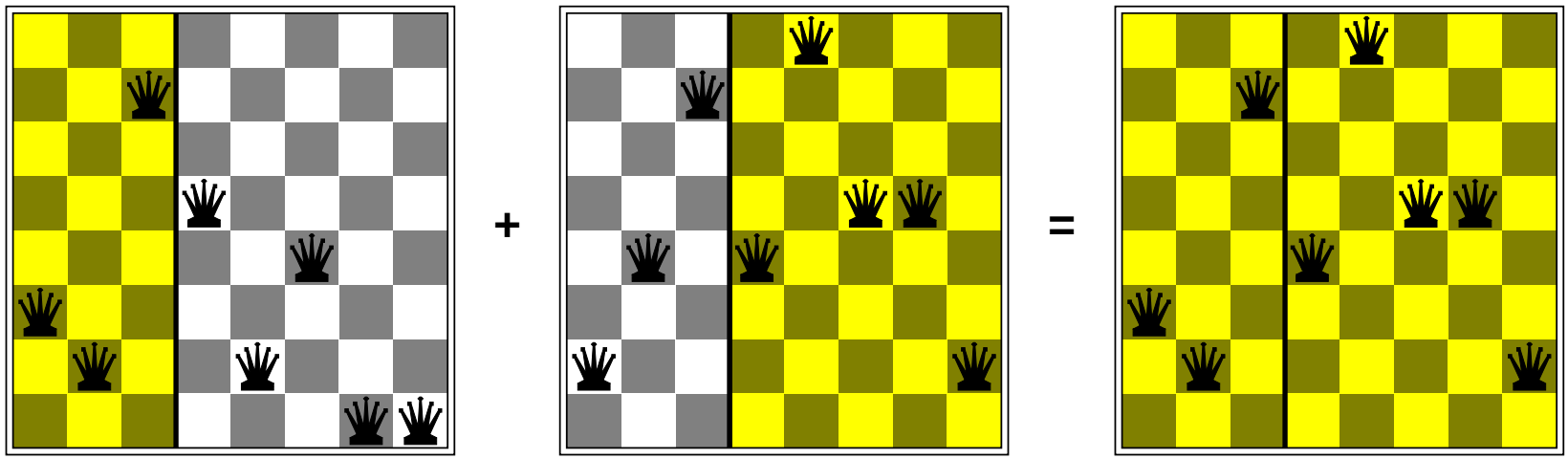
* To choose x, usually use *relative fitness*:

  for each $x \in$ *population*, Pr[choose x] = FITNESS(x)$/\sum_z$FITNESS(z).

To choose y, usually use relative fitness or a uniform distribution

Genetic algorithms

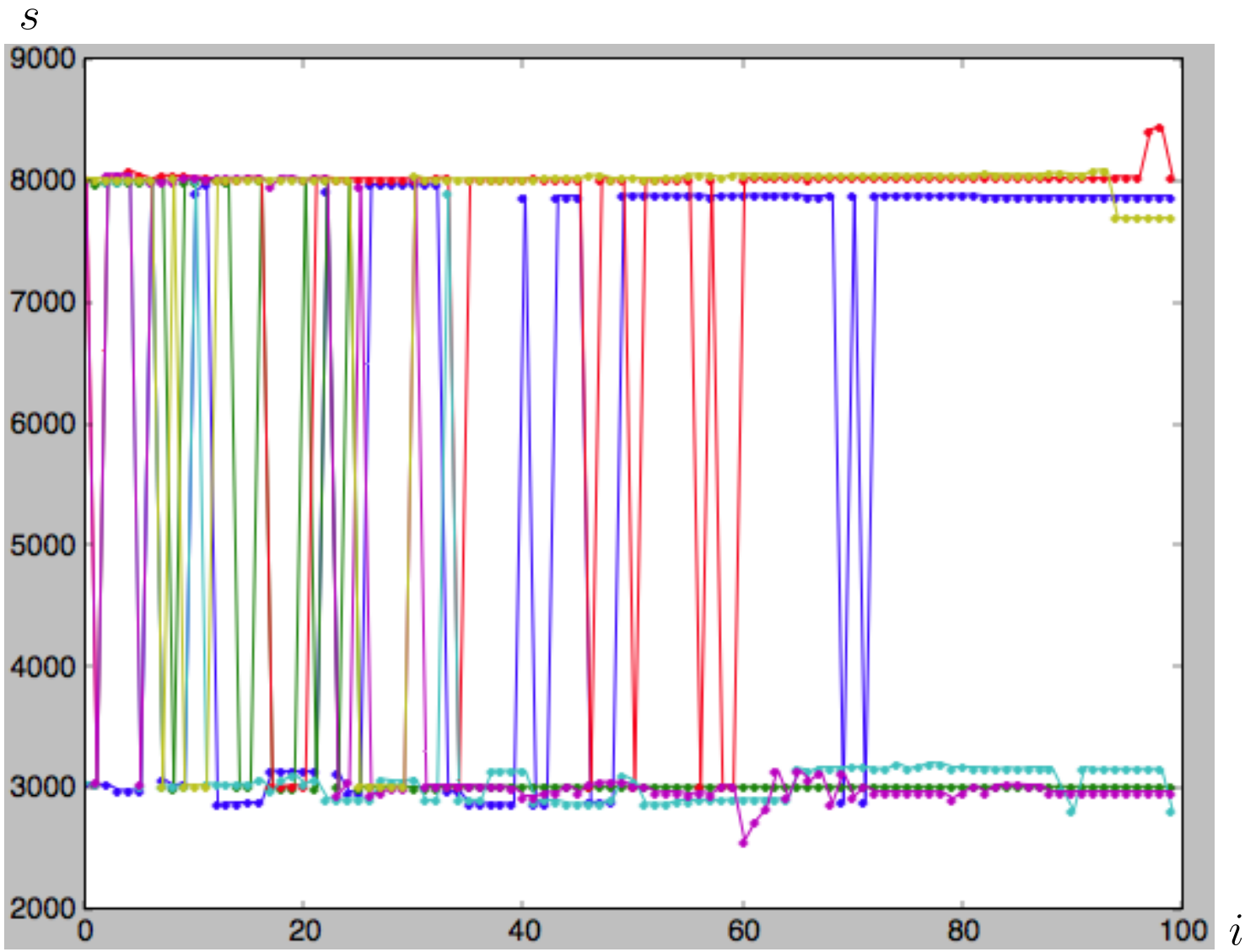




Same simple example as before

- ♢ A state is a number $s \in [0, 1]$. All states are neighbors
- ♢ Fitness = max{–(dist. from 0.3), .025 – (dist. from .8)}

```
import random
stringsize = 4; top = 10**stringsize

def example(times=100, popsize=200, mutation_prob=.001):
    population = [int(random.random() * top) for j in range(popsize)]
    history = []
    for i in range(times):
        fitness = [value(population[j]) for j in range(popsize)]
        fitmax = -float('inf')
        for j in range(popsize):
            if fitness[j] > fitmax:
                fitmax = fitness[j]; jbest = j
        history.append(population[jbest])
        fit_sum = sum(fitness)
        probs = [x/fit_sum for x in fitness]
        cutoff = [sum(probs[:j+1]) for j in range(popsize)]
        children = []
        for j in range(popsize):
            r = random.random()
            for k in range(popsize):
                if r < cutoff[k]: break
            par1 = population[k-1];
            par2 = population[int(random.random() * popsize)]
            split = int(random.random() * (stringsize+1))
            child = str(par1)[:split] + str(par2)[split:]
            if random.random() < mutation_prob:
                where = int(random.random() * stringsize)
                what = str(int(random.random() * 10))
                child = child[0:where] + what + child[where+1:]
            children.append(int(child))
        population = children
    return history

def value(s):
    return max(-abs(round(0.3*top) - s), .025 - abs(round(0.8*top) - s))
```



- 100 iterations, population size = 200, mutation probability = .001
- Running time is much worse than for simulated annealing

Discussion

♢ Genetic algorithms $\neq$ biological evolution

- for example, real genes encode replication machinery

♢ Genetic algorithms are widely used by engineers for optimization problems

  ◇ circuit layout, job-shop scheduling

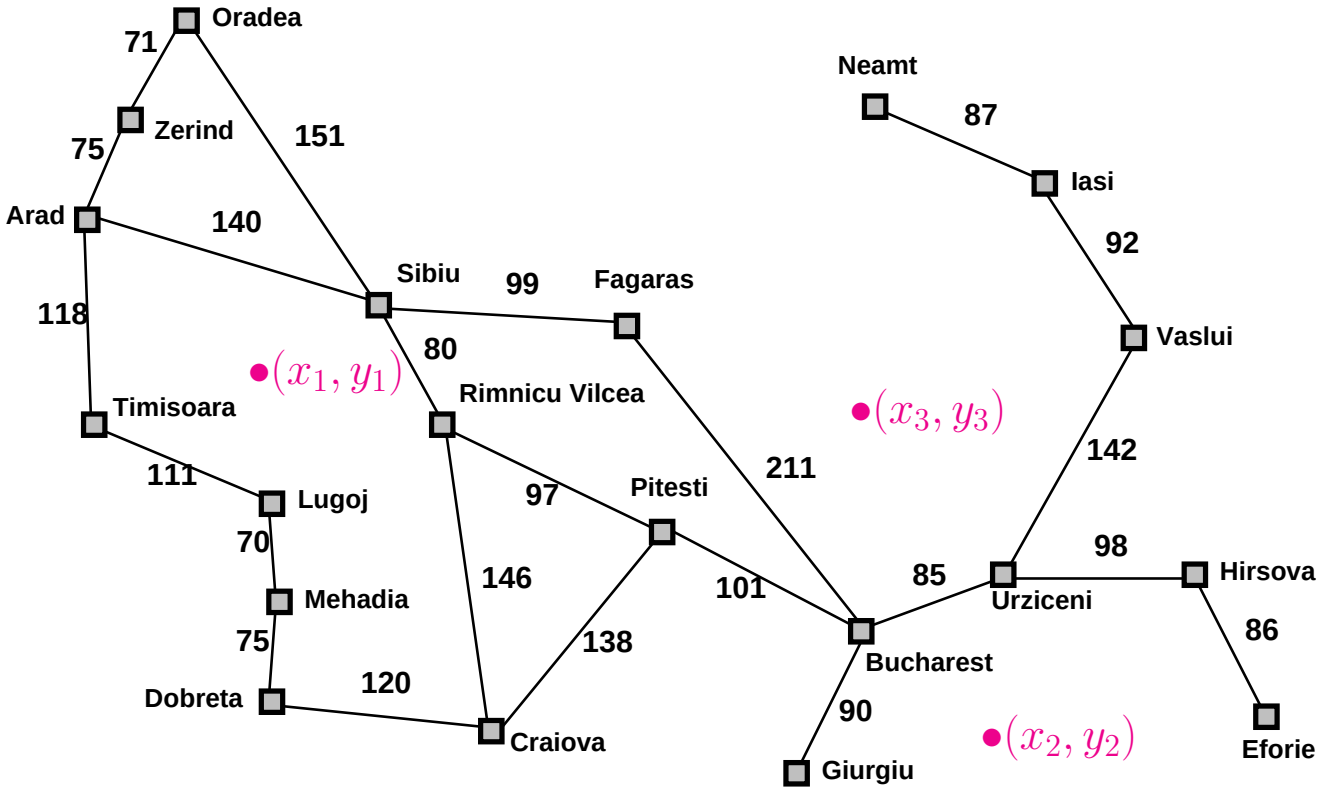
- Not clear whether this is due to performance or intuitive appeal

♢ More work is needed to identify conditions under which they perform well

Hill-climbing in continuous state spaces

♢ Suppose we want to put three airports in Romania – what locations?

- 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
- Objective function $f(x_1, y_2, x_2, y_2, x_3, y_3)$ measures desirability
  - ♢ e.g., sum of squared distances from each city to nearest airport



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Hill-climbing in continuous state spaces

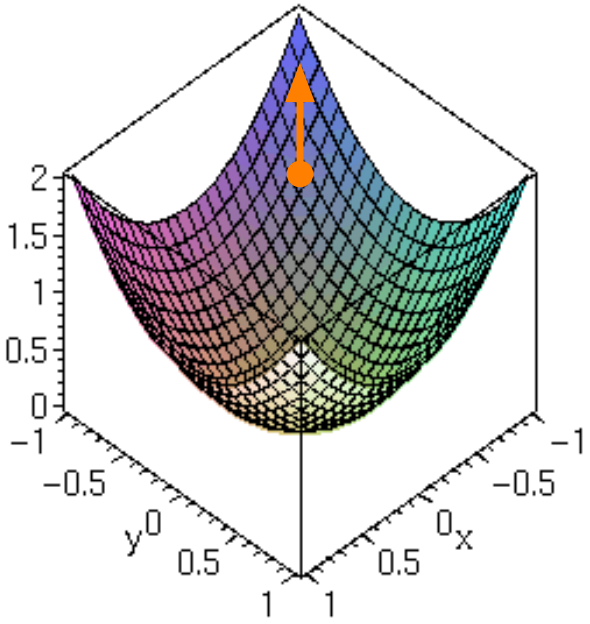
♢ A technique from numerical analysis:

♢ Given a surface $z = f(x, y)$ and a point (x, y)

♢ A **gradient** is a vector

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

- points in the direction of the steepest slope
- length is proportional to the slope



♢ Gradient methods compute ∇f and use it to increase/reduce f

◇ e.g., $\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla f(\mathbf{x})$

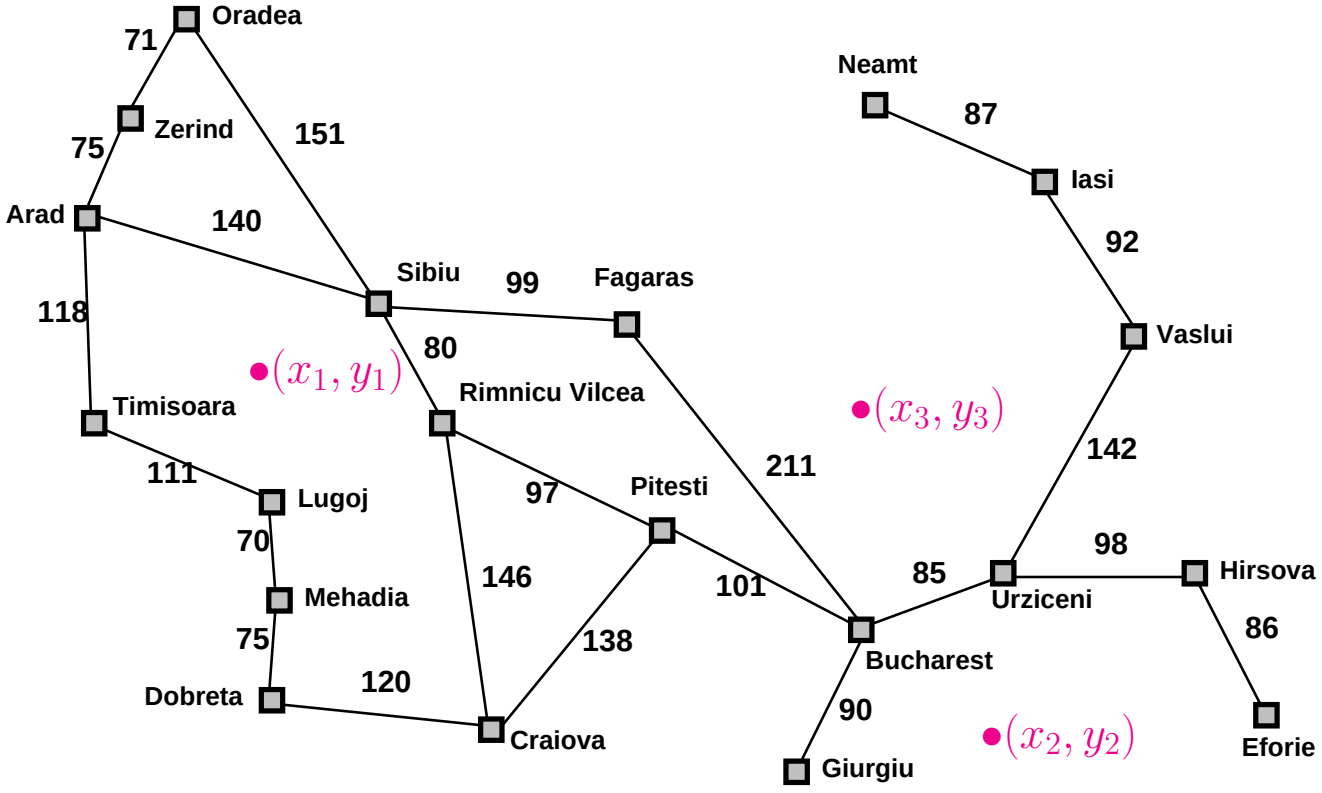
- If $\nabla f = 0$ then you've reached a local maximum/minimum

Hill-climbing in continuous state spaces

♢ Suppose we want to put three airports in Romania – what locations?

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

♢ Look for $x_1, y_1, x_2, y_2, x_3, y_3$ such that $\nabla f(x_1, y_1, x_2, y_2, x_3, y_3) = 0$



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Continuous state spaces, continued

♢ Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city)

♢ *Newton–Raphson* method (1664, 1690)

- solve $\nabla f(\mathbf{x}) = 0$ by iterating $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}^{-1}(\mathbf{x})\nabla f(\mathbf{x})$
  - ◇ $\mathbf{H}$ is an $n \times n$ matrix with $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$
- I used to know this stuff, but that was a **long** time ago 😕

♢ *Discretization* methods turn continuous space into discrete space

- e.g., *empirical gradient* considers $\pm\delta$ change in each coordinate