Iterative Improvement

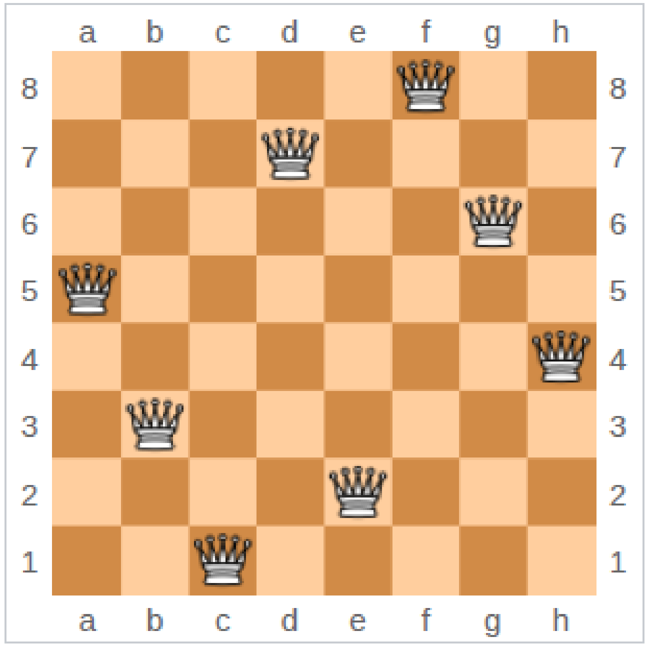
CSCE 420 – Fall 2023

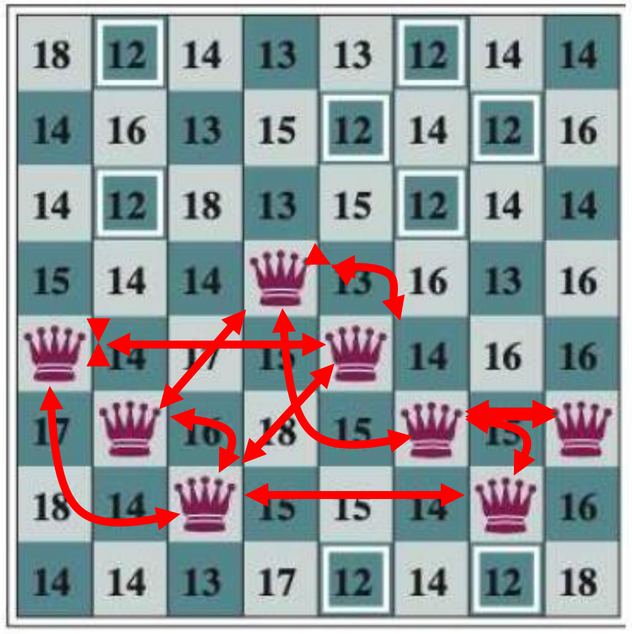
read: Sec. 4.1

Iterative Improvement Search

- also known as Local Search
- maximize the "quality" of states, q(s) or value(s)
    - note: this is different than path cost
- example: 8-queens
    - can you place 8 queens on chess board such that none can attack each other?
    - initial state: place all 8 queens randomly, one in each column
    - q(s) = -(number of pairs of queens that can attach each other)
        - use negative so higher is better; or modify algorithm to find state with minimum score (gradient descent)

q(s)=0

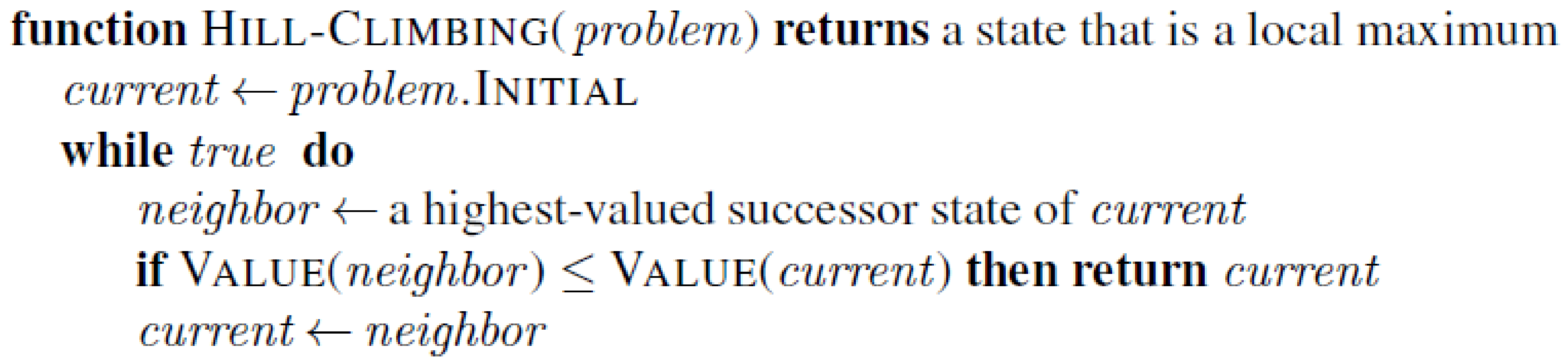




q(s)=-17

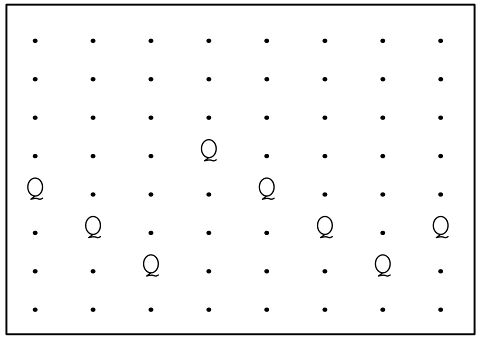
Hill Climbing

- maintain only a single current state
- generate successors using operator, and pick best

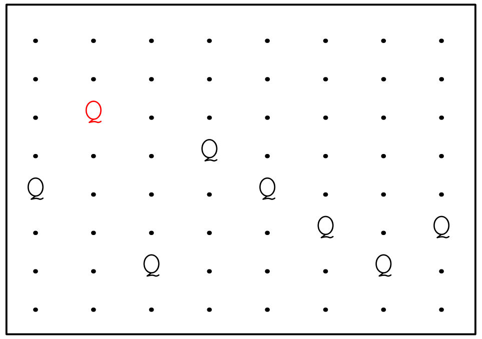


```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
     neighbor ← a highest-valued successor state of current
     if VALUE(neighbor) ≤ VALUE(current) then return current
     current ← neighbor
```

- operator for 8-queens: move any queen to another row in the same column

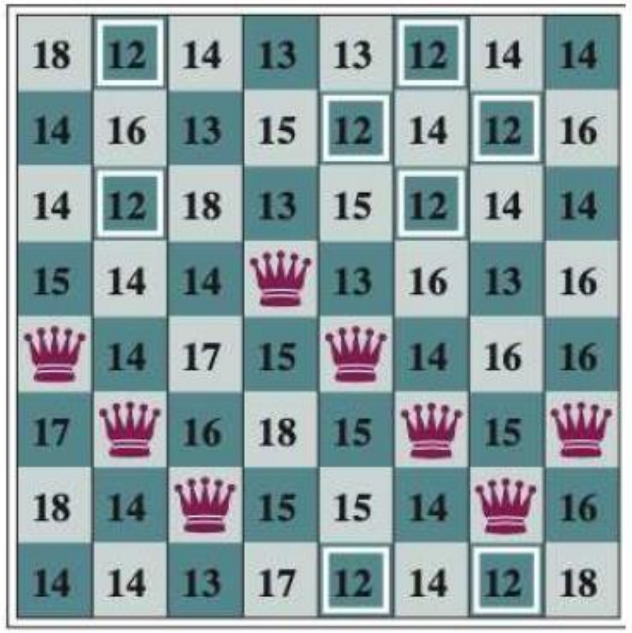


q=17

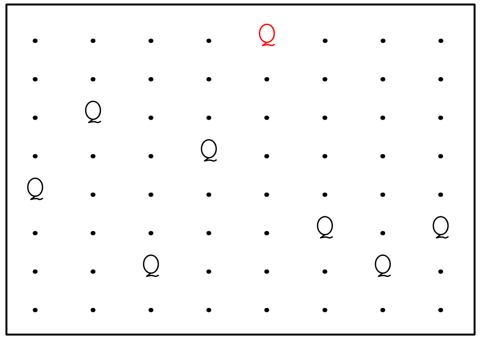


q=17-5+0=12

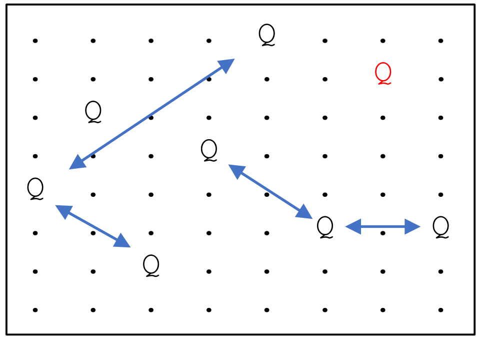
A sequence of iterations, where the best queen is moved to a new position in her column that most reduces the number of overall conflicts.



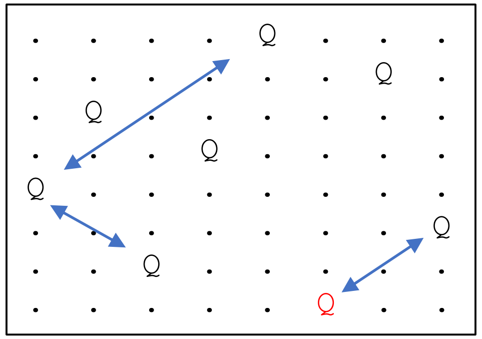
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18



q=12-5+1=8

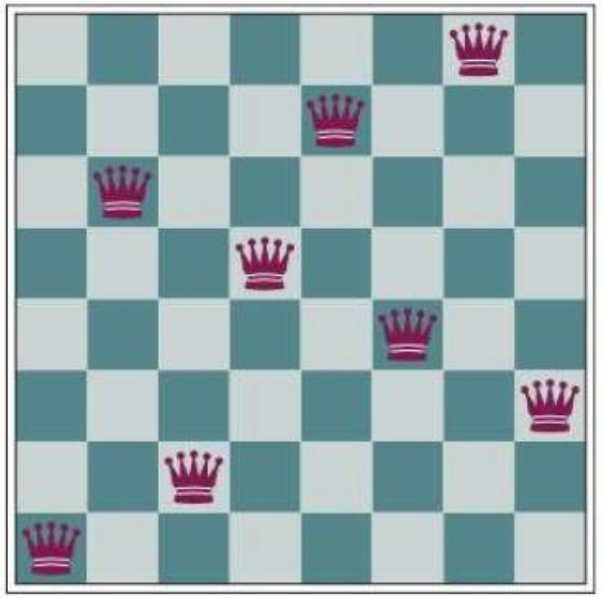


q=8-4+0=4

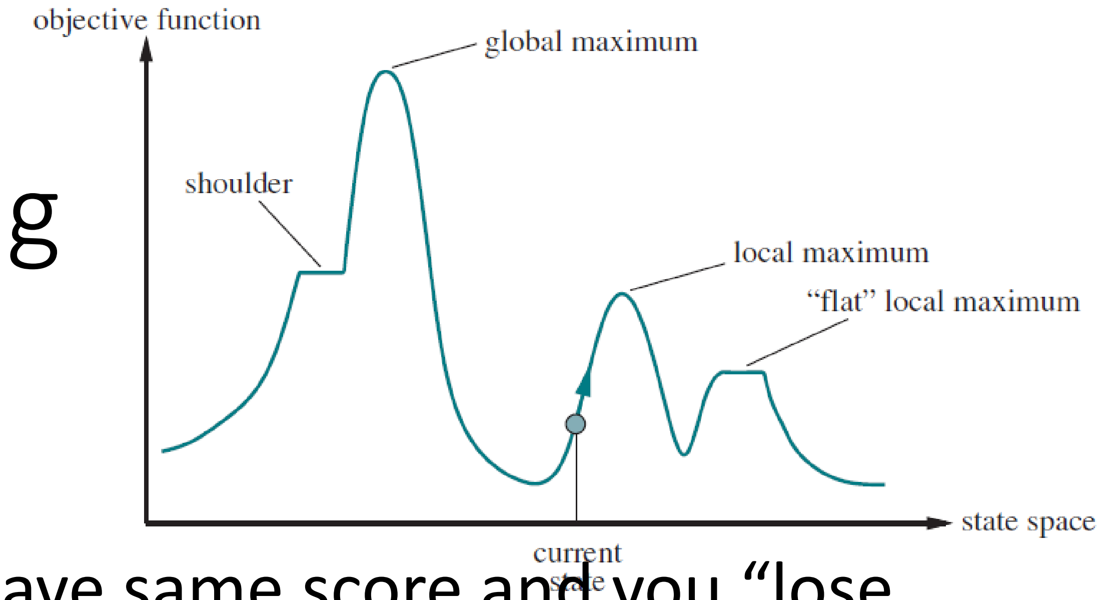


q=4-2+1=3

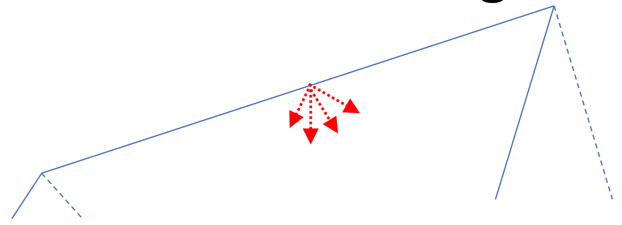
No further improvements can be made.

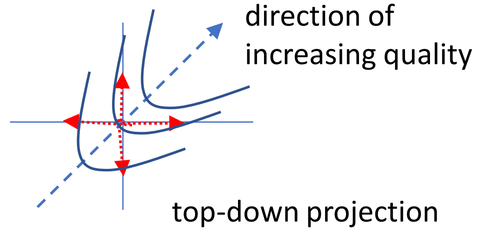


Problems with Hill-Climbing

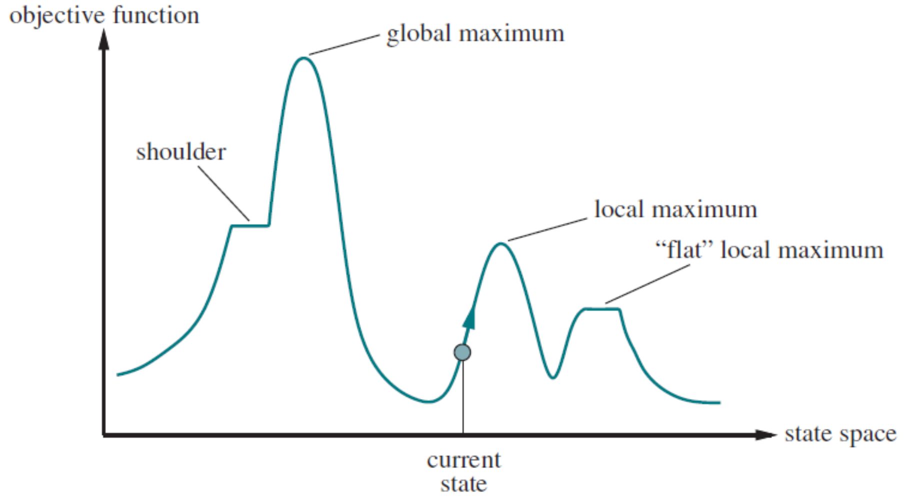


1. **local maxima**
2. **plateau effect** – when all neighbors have same score and you "lose the gradient", even if not at top of hill
3. **ridge effect** – all neighbors have same or lower score, even then there might be other close states that are better
   - suppose only choices are to go N, S, E, or W, but ridge goes up NE; hence all steps go down sides of the ridge
   - often related to limitations of the successor function; consider expanding it to generate more successors in neighborhood (e.g. combinations of 2 steps)





Possible Solutions



- random restart HC
- stochastic HC – choose any successor that is better than current state, not always the best
  - you can't just choose any random successor; must still bias the search upward
  - can this strategy really reduce risk of local minima?
  - this idea leads to **Simulated Annealing**...
- provide memory of previous states
  - HC only maintains 1 state: the current state
  - perhaps we could remember previously expanded-but-not-explored nodes to allow some "back-tracking" if we get stuck at a local maximum
  - this idea leads to **Beam Search**...
- allow multiple steps – also consider successors of successors; create new operators (macro operators) from combinations of 2 or 3 actions, expanding the number of successors; similar to look-ahead, or "gradient sampling"

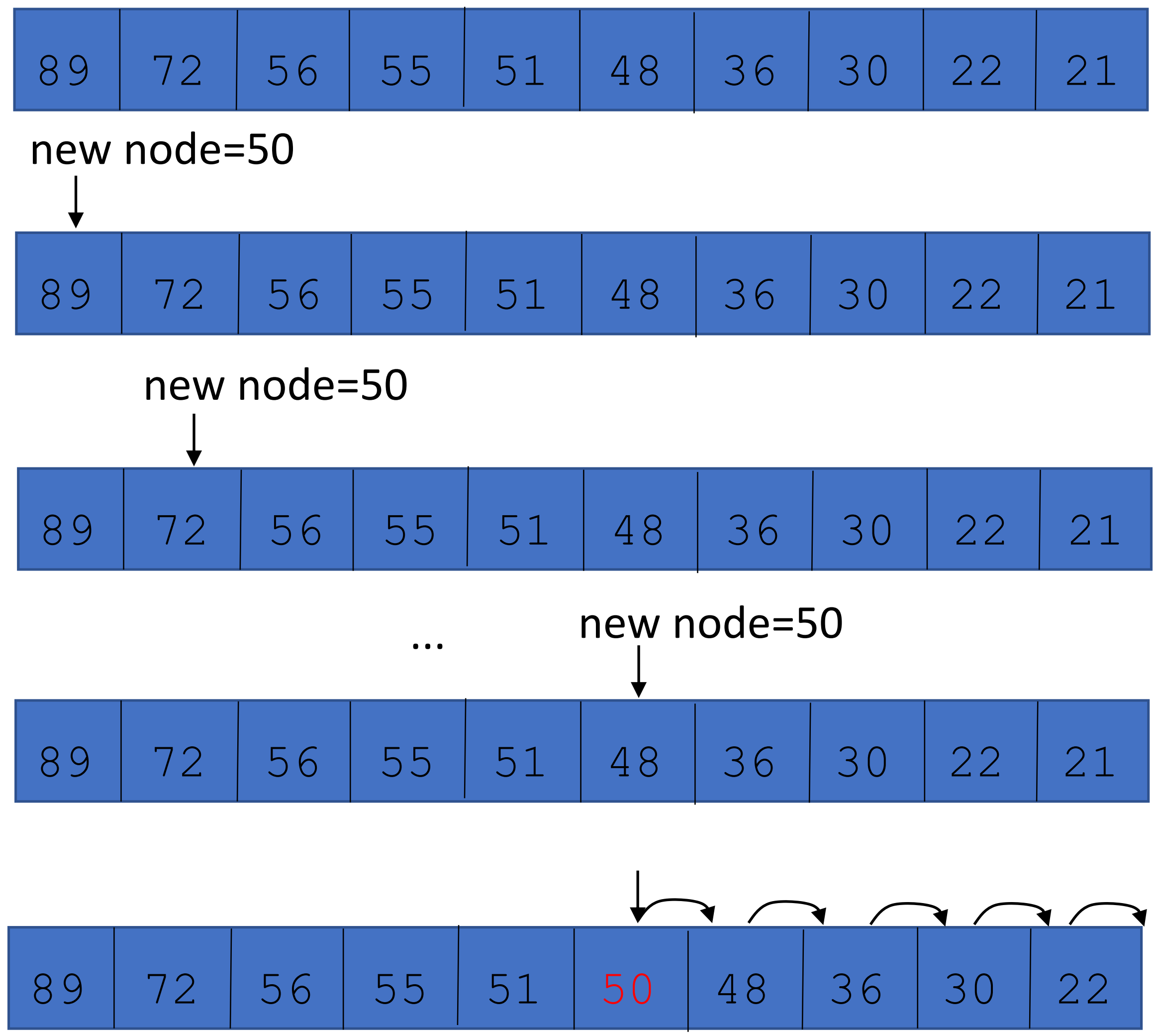
Beam Search

- adding "memory" to Hill-Climbing
  - comparison to Greedy Search: we used a frontier to keep track of *all* previously expanded-but-not-yet-explored states
  - (another difference: Greedy sorts PQ by h(n), and HC chooses successors by q(n))
  - however, this potentially has high space-complexity (exponential frontier size)
  - is there a compromise?
  - yes – keep track of the K best previous nodes (based on state quality q(n))
  - this fixed-size array allows *some* back-tracking, even if not complete enough to explore the whole space (typically, beam size=10-100 nodes)
  - thus, Beam Search could possibly back-track off of one hill and get onto another with a higher local maximum, even though it might fail to find the global max.

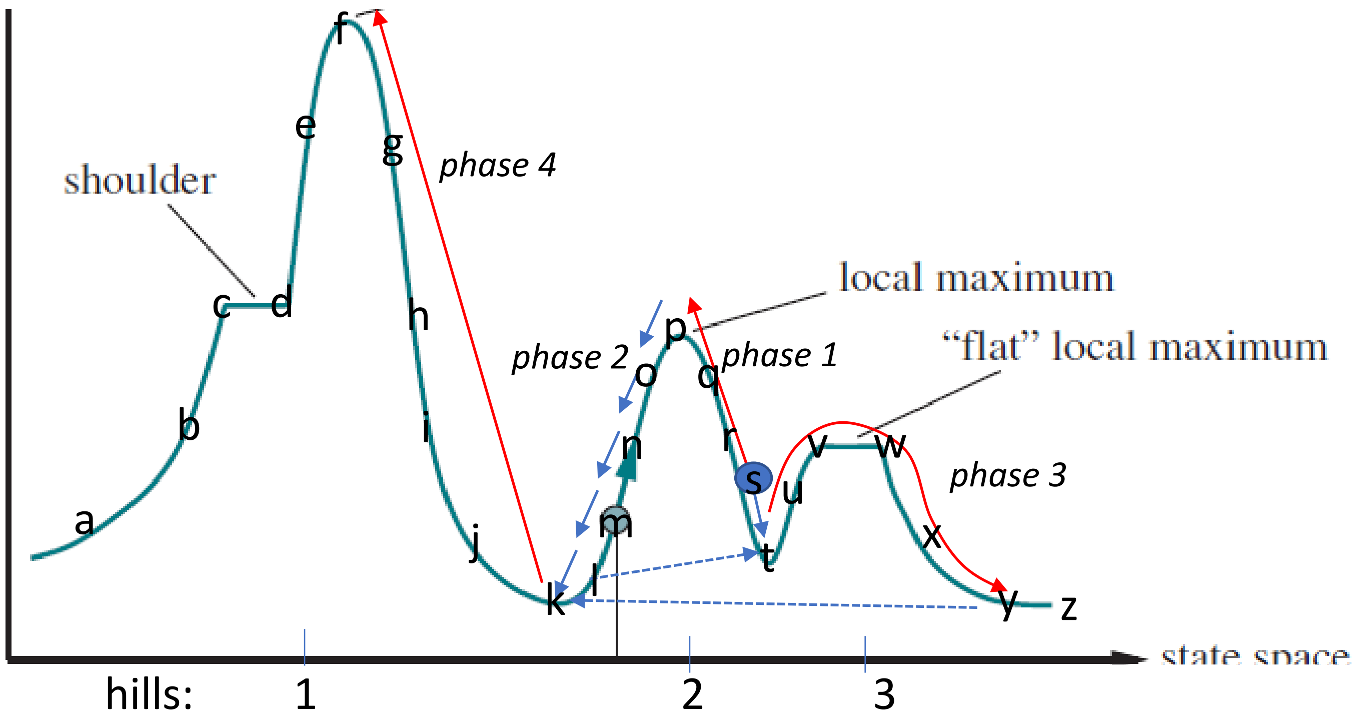
```
BeamSearch(init,k)
    beam←array[k]
    beam.insert(init)
    while True // or until beam empty, or reach max iterations...
         // pop best node in beam (highest score at front)
        curr←beam[0]
        for each child c∈operator(curr):
            beam.insert(c)

beam.insert(node n)
    do insertion sort
    since beam is always presorted, scan the list to find where n
fits, and shift the rest of the nodes down (the last one falls out
of the beam)
```

Quality q(n) of top K nodes:



note: something falls off end of beam (with lowest score) - it could be the kth item in beam, or the new node if it is worse than everything currently in beam (e.g. score<21)



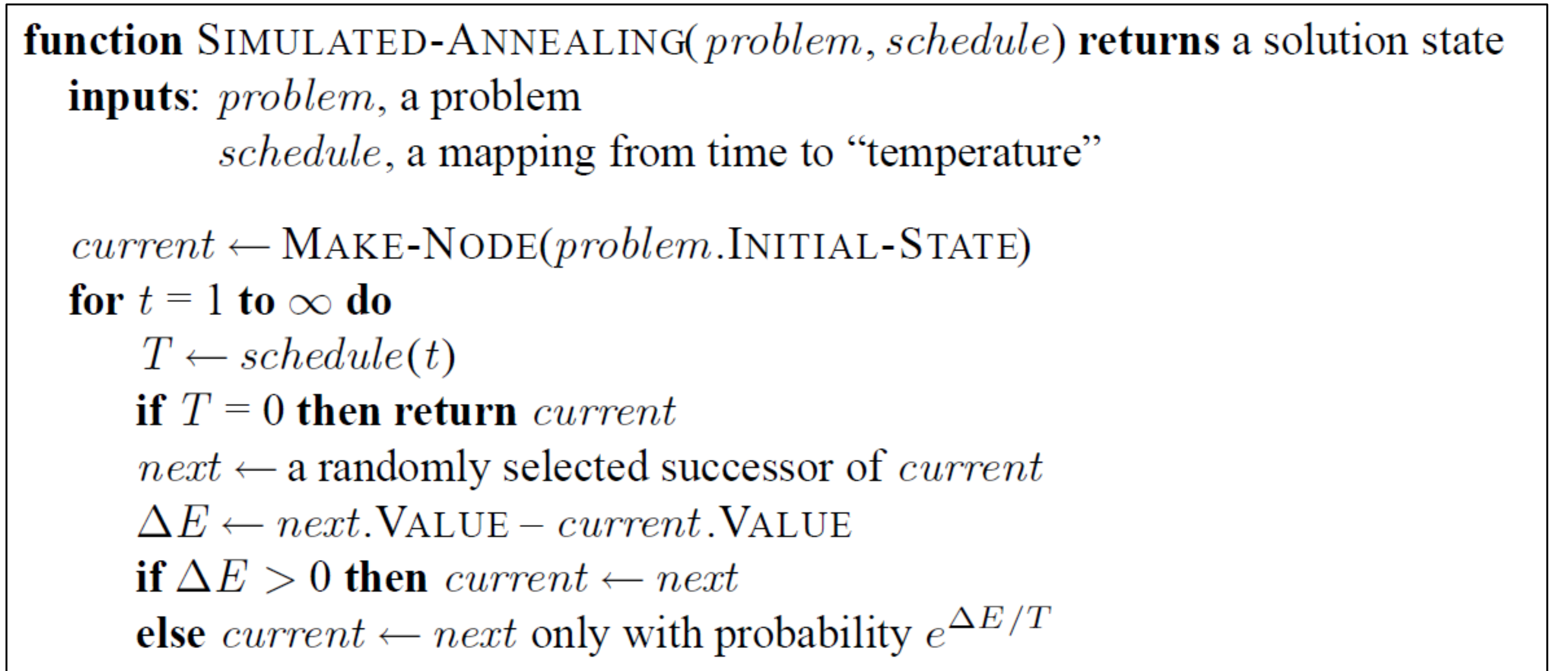
beam(size=2):
s
r, t
q, t
p, t
o, t
n, t
m, t
t, l
u, l
v, l
w, l
x, l
l, y
k, y
j, y
…

phase 1
phase 2
phase 3

- because this is a simple 1D space, all states have two neighbors
- usually, one of the neighbors has been recently visited, so we discard it
- phase 1: start at 's'; climb to local max at 'p' (hill 2)
- phase 2: when reach top of hill, the beam remembers these 2 nodes: [o,t]; start descending left slope of hill 2
- phase 3: there is a point where beam has both 'l' and 't' in it [l,t]; start ascending hill 3 since 't' is better
- phase 4: eventually, when reach 'y', resume search at 'l' and climb hill 1

Simulated Annealing

- stochastic search
- choose next child randomly, but “bias it upward”
- always accept better states, and accept worse states probabilistically, proportional to how much lower the quality is

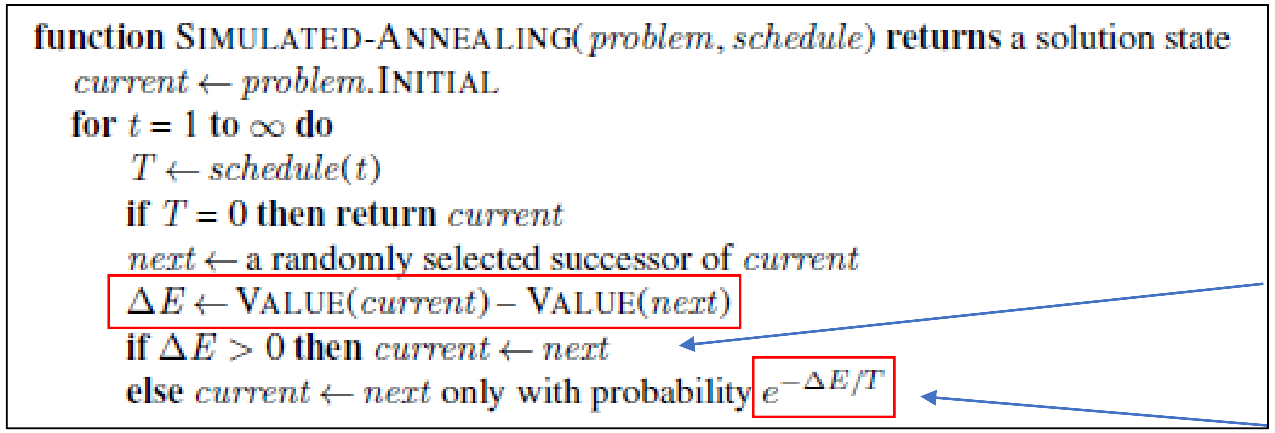


```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to “temperature”

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

The algorithm in the 4th ed. of the textbook has 2 errors...

AIMA, 4th ed.
(Fig 4.5)

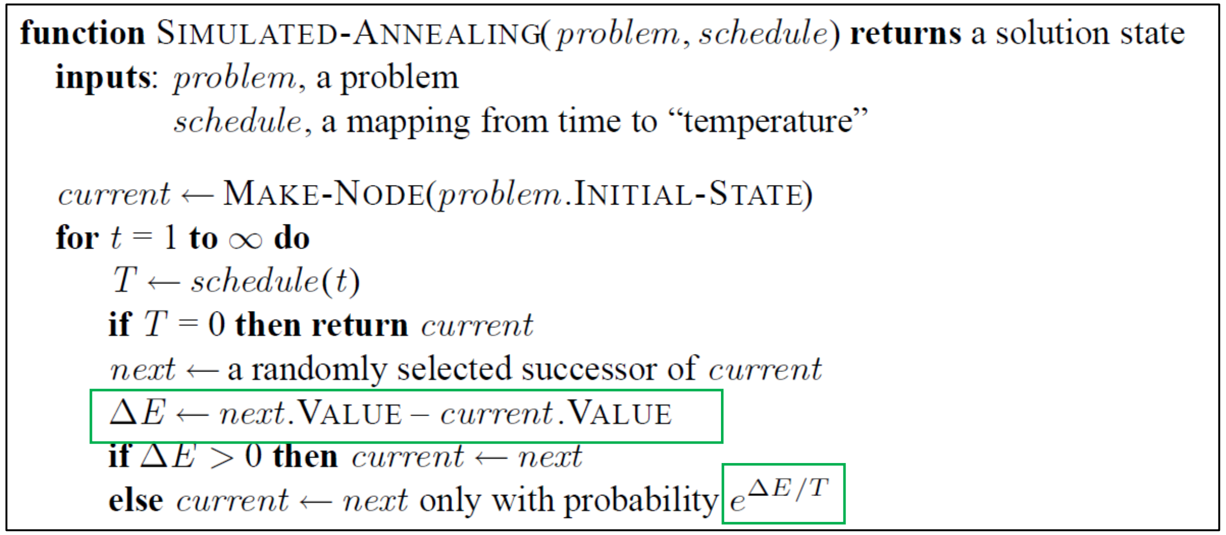


```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      $\boxed{\Delta E \leftarrow \text{VALUE}(current) - \text{VALUE}(next)}$
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $\boxed{e^{-\Delta E/T}}$
```

(want to accept if next is <u>higher</u> than curr)

(the exponent should be negative, but $-\Delta E/T>0$ since $\Delta E<0$)

AIMA, 3rd ed.
(correct)



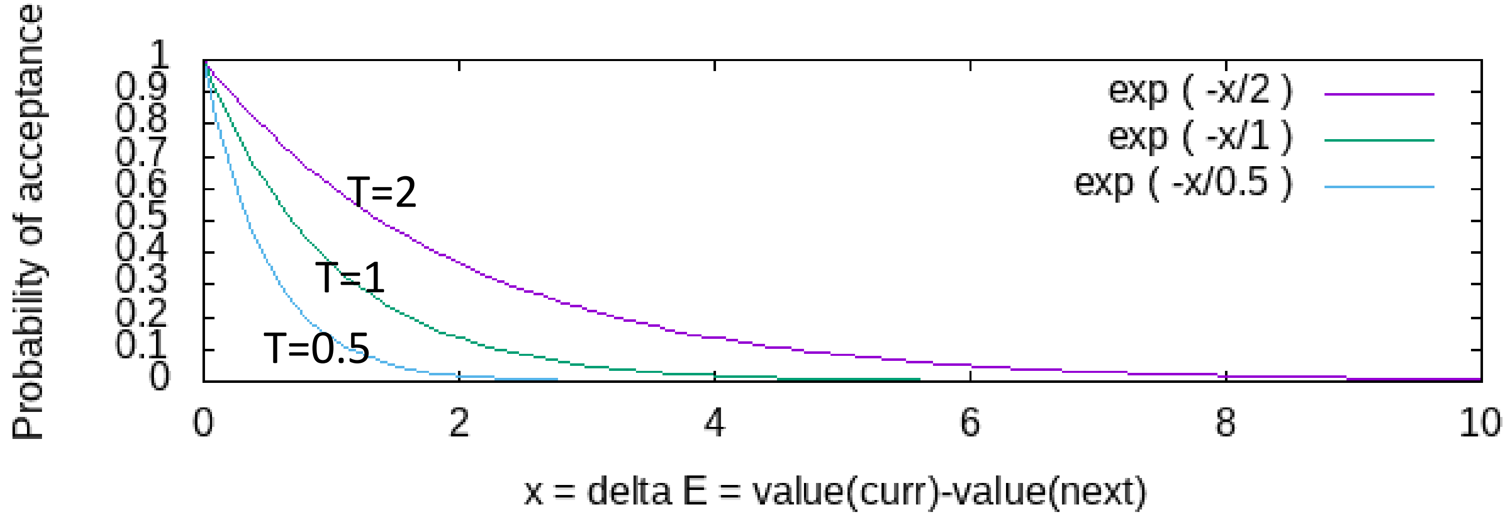
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  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
      $\boxed{\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}}$
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $\boxed{e^{\Delta E/T}}$
```

Simulated Annealing

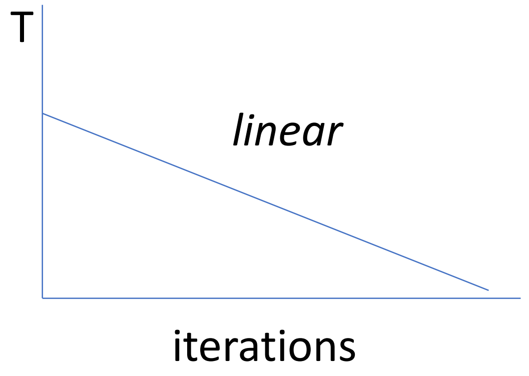
- accept with prob = $e^{-\Delta E/T}$ where ΔE=value(curr)-value(child)
  - if child is only a little worse, ΔE is small, so accept with high prob
  - if child is much worse, ΔE is large, and acceptance is less likely
- T ("temperature" controls) how loose or stringent we are
  - in the limit $T \to \infty$: all backward steps are allowed
  - in the limit T=0: no backward steps are allowed

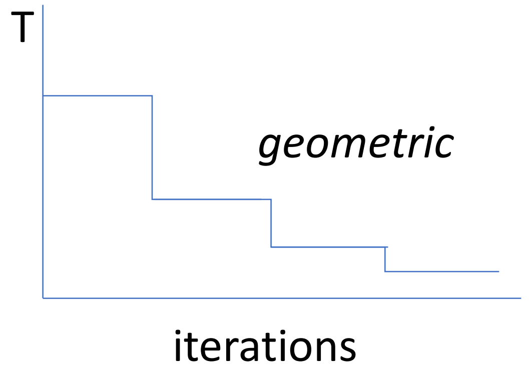
this is analogous to "cooling" in materials like metal; malleable at high temperatures, but gets locked into a lattice structure at low temperatures

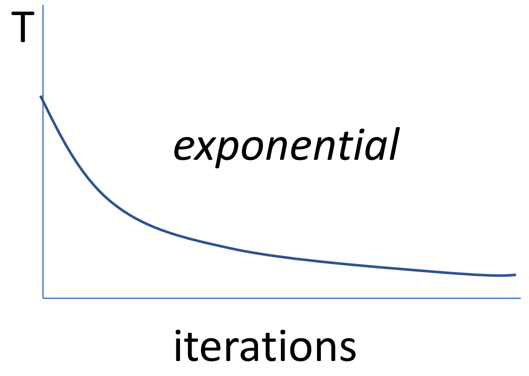


Temperature schedules

- a critical part of SA is to start with a high temperature and gradually lower it
- this allows the search to sample many local maxima initially, but over time, it becomes more selective and climbs up the best hill it it can find
- linear, geometric (e.g. halving every 1000 iterations), exponential...

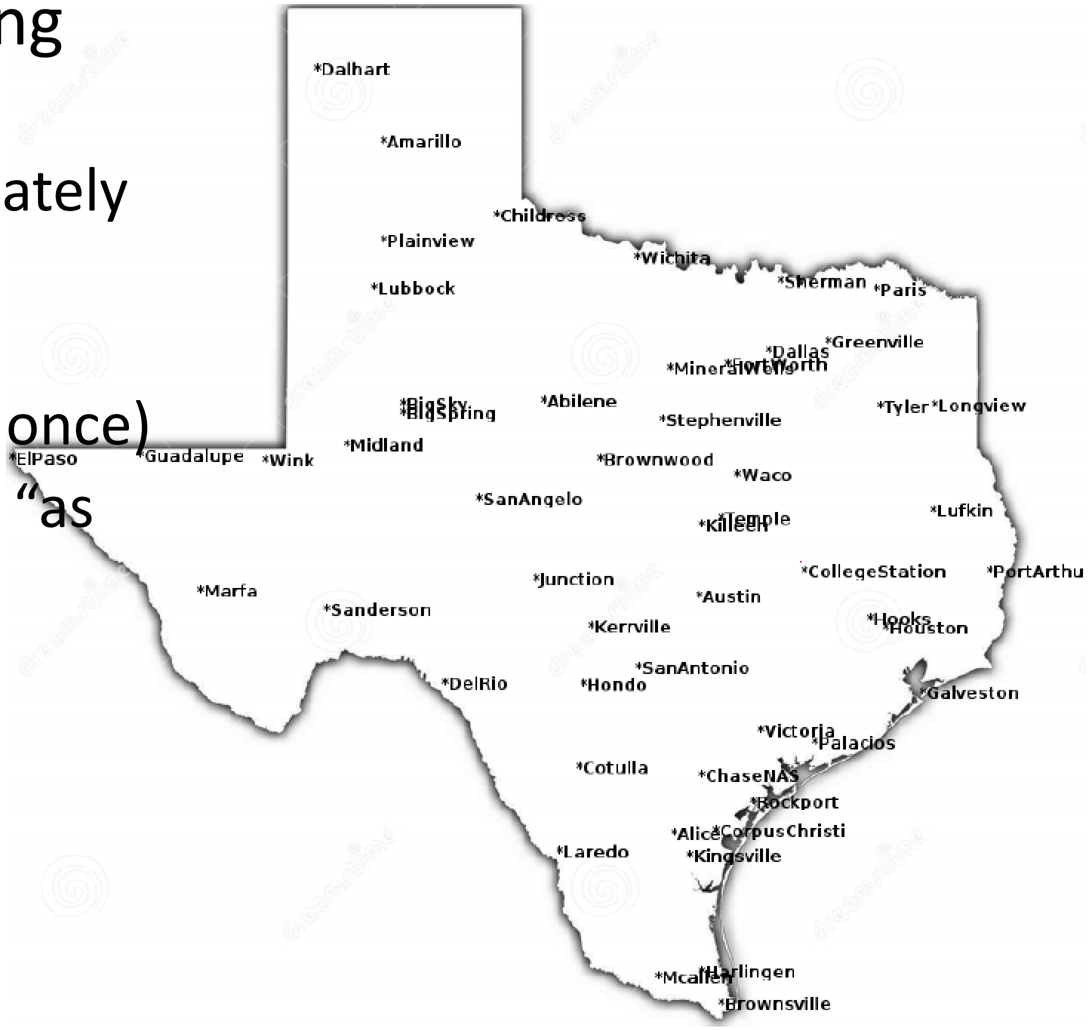


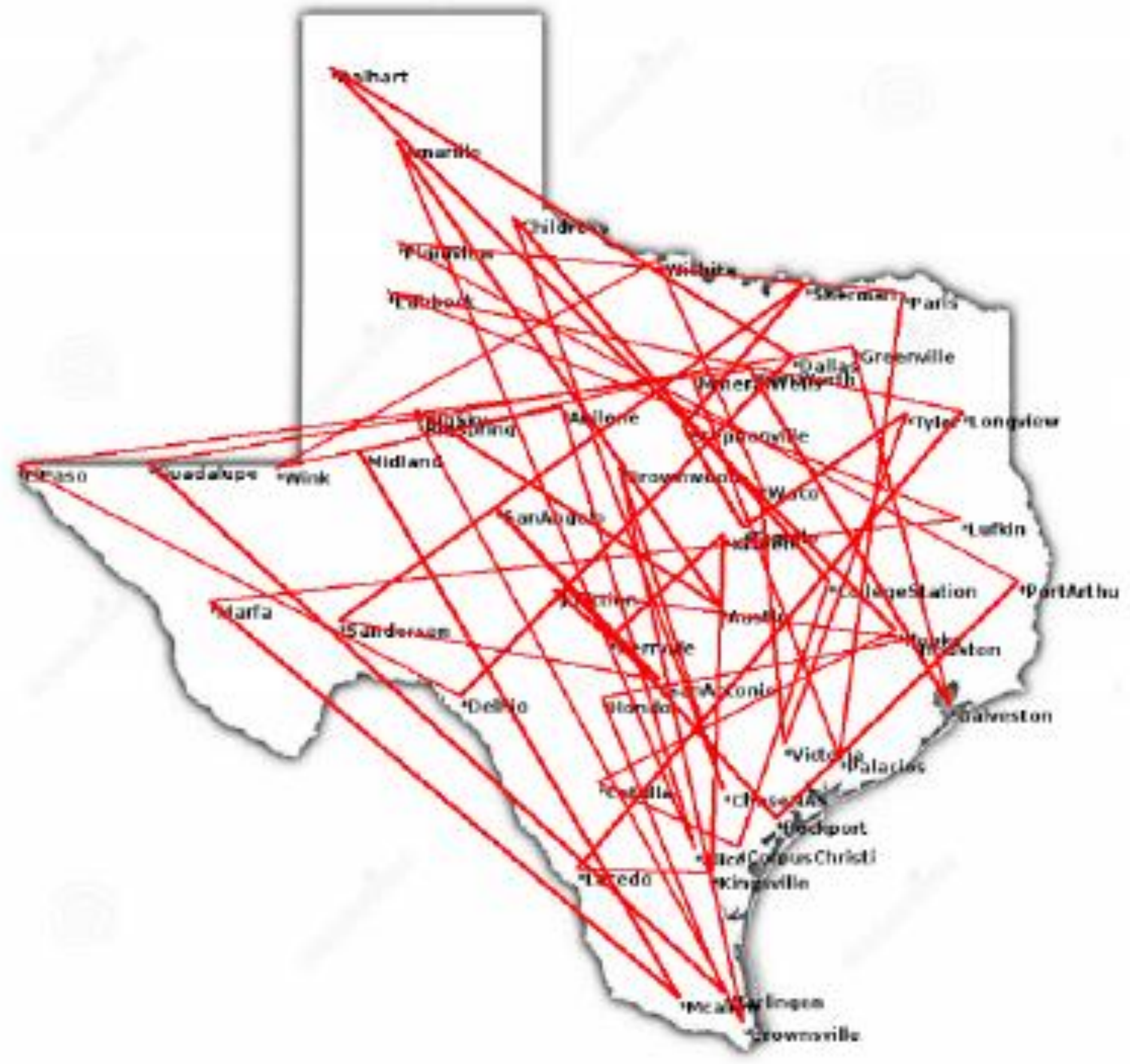




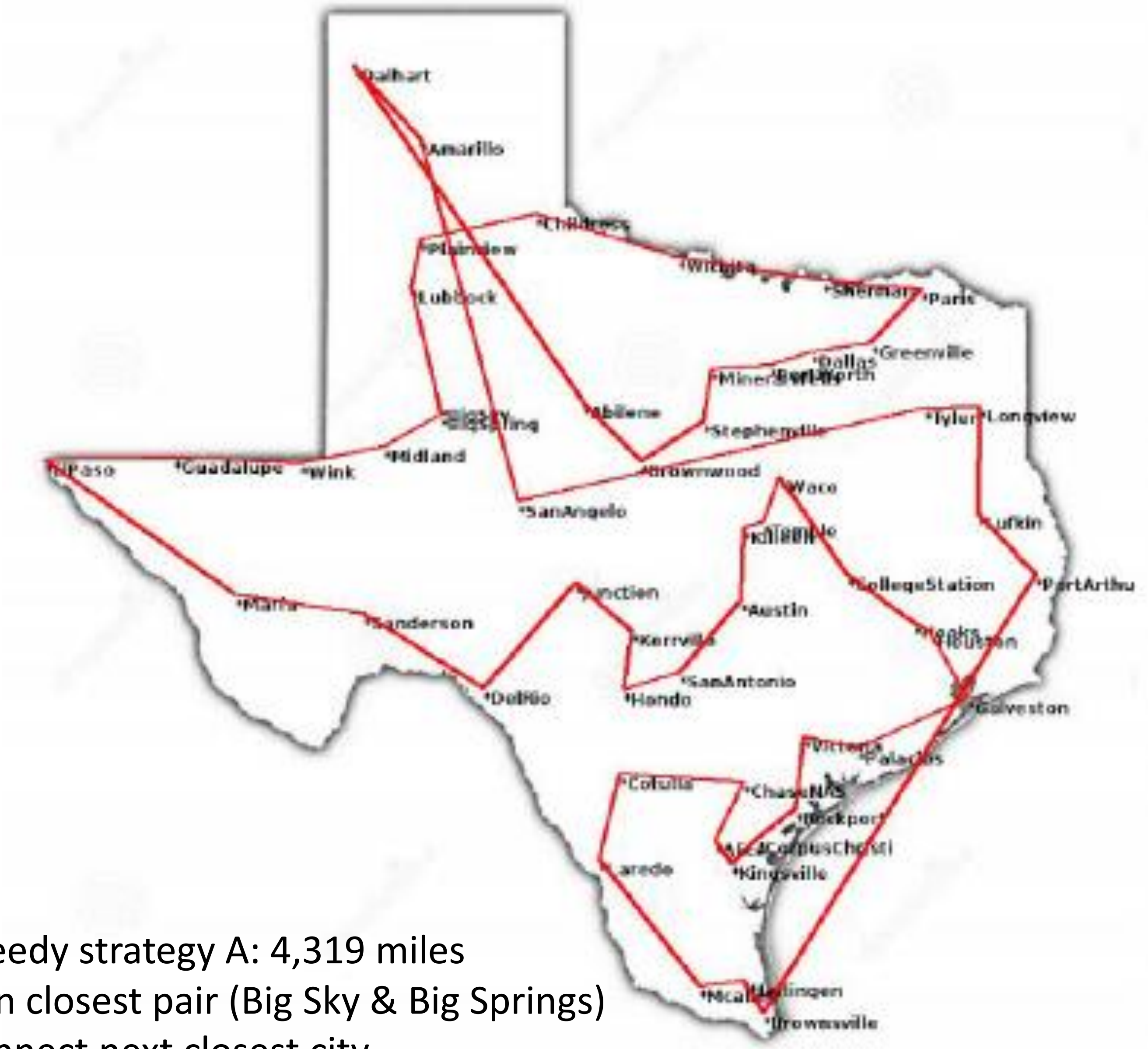
Simulated Annealing

- application of SA to “solving” the Traveling Salesman Problem (TSP)
  - actually, we can only hope to find approximately optimal solutions, because TSP is NP-hard
  - tour with minimum total length
  - (Hamiltonian cycle: visit every node exactly once)
  - example: Texas cities (pairwise connectable “as the crow flies”)

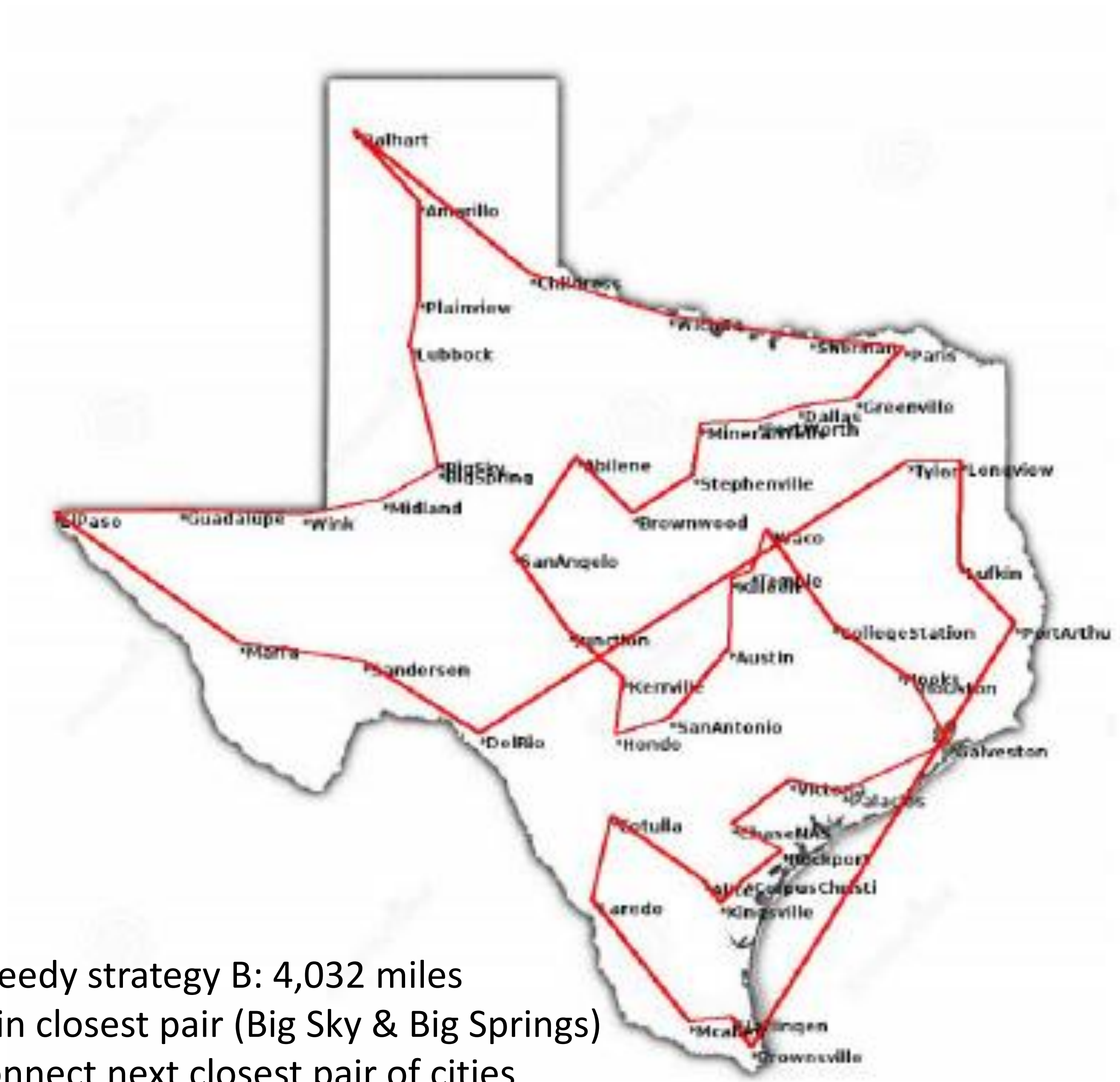




random tour, length=16,697 miles



greedy strategy A: 4,319 miles
join closest pair (Big Sky & Big Springs)
connect next closest city
repeat till all connected

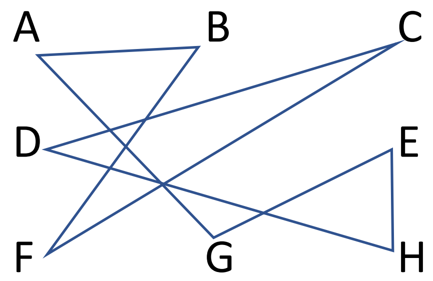


greedy strategy B: 4,032 miles
join closest pair (Big Sky & Big Springs)
connect next closest pair of cities
  (except cities already connected to 2 others)
repeat till all connected

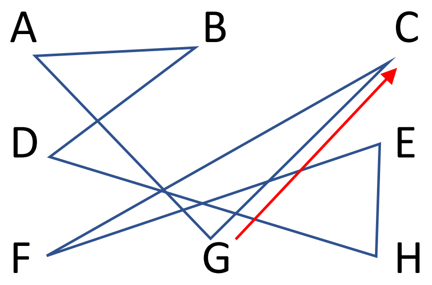
Simulated Annealing

- representation for TSP (for complete graphs): a list of the nodes in any order (permutation)
- operators for TSP (for complete graphs)
  - how to generate “variants” of any given tour (successor states)?
  - there are many ways to do this
  - choose a random subsequence and move it to another position
  - choose a random subsequence and reverse it

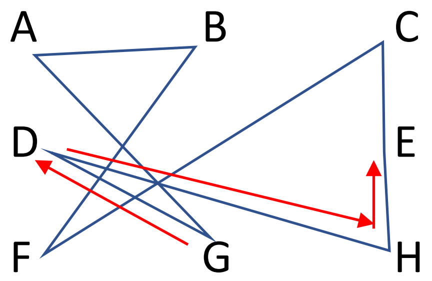
[A G E H D C F B]

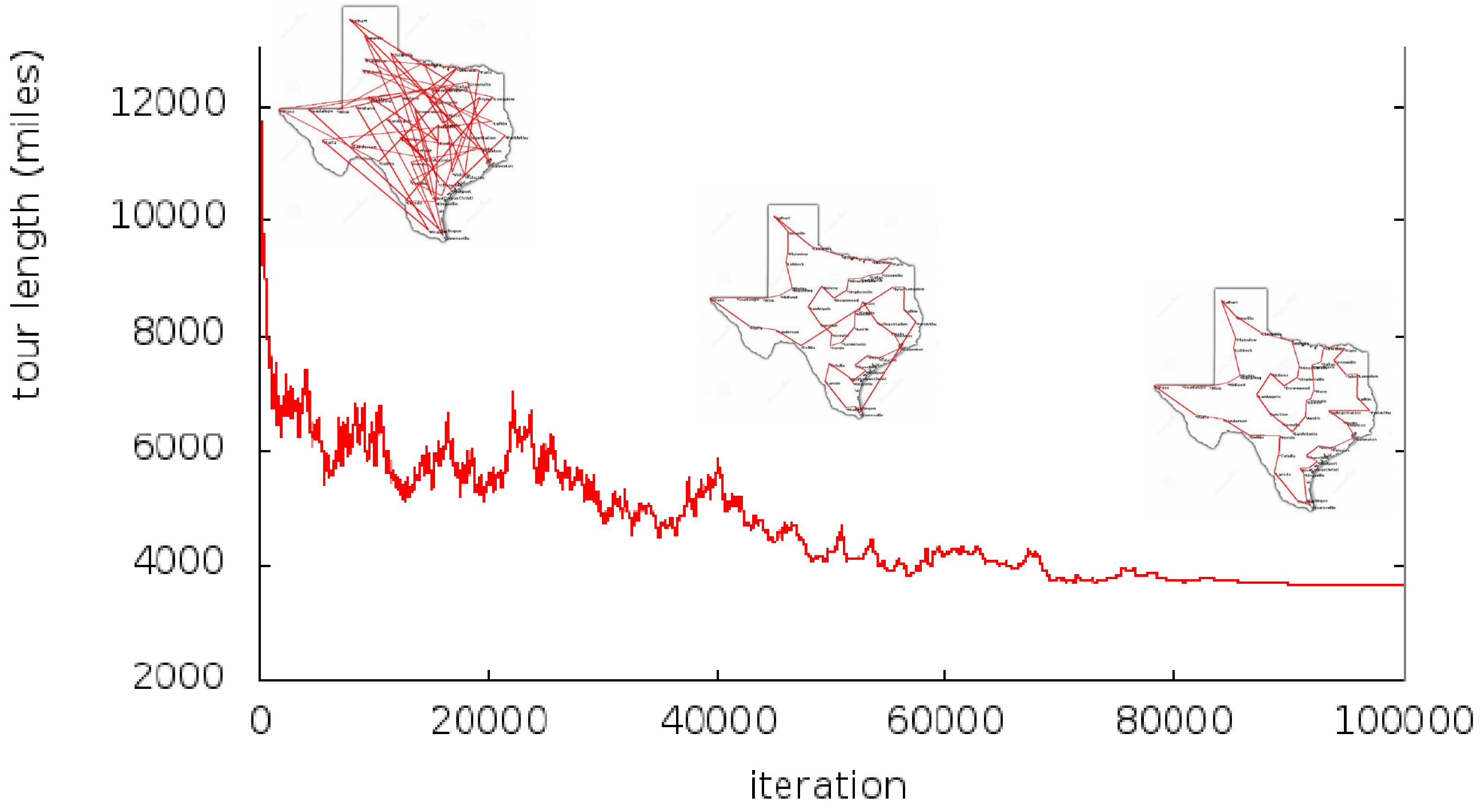


[A G C F E H D B]

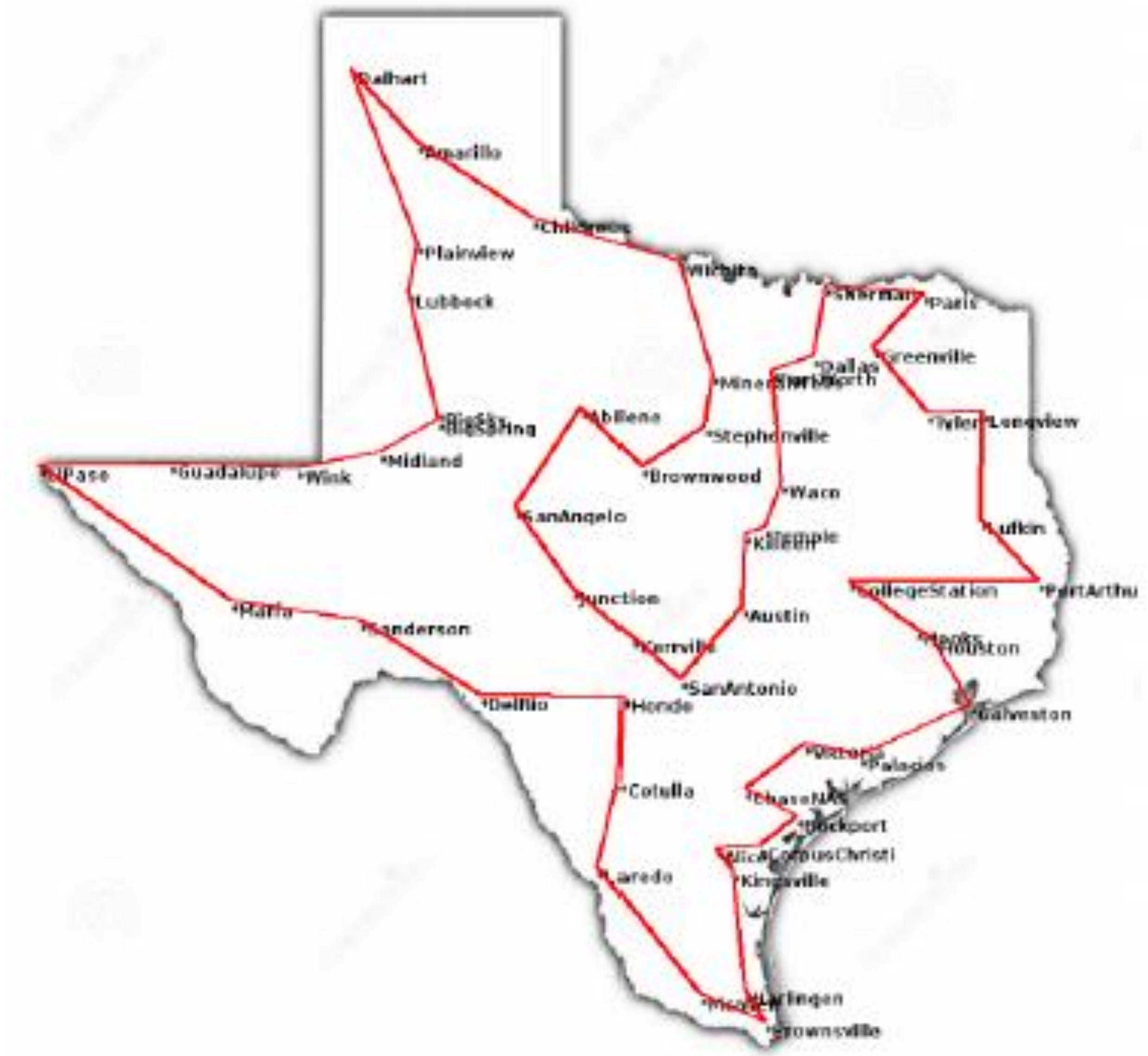


[A G D H E C F B]





state = list of cities (complete tour) (state space size = ?)
operators:
A) pick 2 cities and swap them
B) pick a subsequence and reverse it
C) pick a subsequence and move to a new position in list



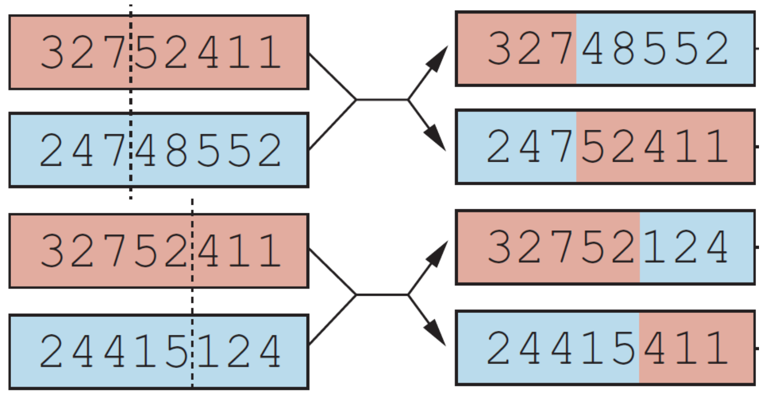
Simulated Annealing: 3,679 miles

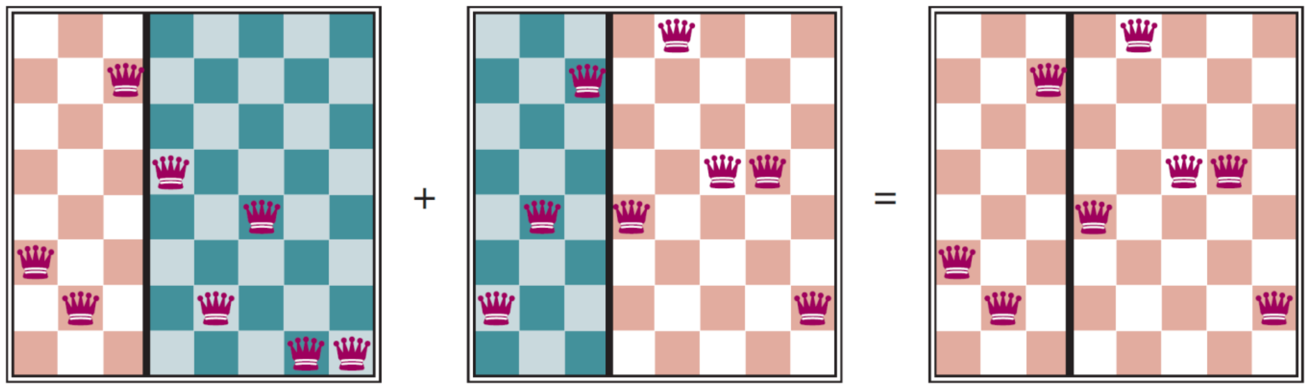
Genetic Algorithms

- also known as Evolutionary Programming
- the unique aspects of GA Search are:
  - maintain a *population* of **multiple candidate states** (parallel search, not just curr)
  - mix-and-match states by *recombination*
  - use *fitness* to select winners each round, akin to ‘natural selection’
- fitness(state) is a synonym for value(s) or quality(s)
- some GAs use ‘chromosomes’, which represent state as a bit string
  - example: state of 8-queens is given by a list of 8 integers (0-7), which can be converted to a 24-bit string: 5,1,7,2,3,6,4,0 → 101001111010011110100000
  - but chromosomes are not necessary, as long as states can be recombined

Recombination or 'cross-over'

- instead of an 'operator' to generate successors from states, use 'recombination' to combine parts of existing members of population





- for chromosomes, splice their strings at a random locations
- for other data types and states representation, use must define 'cross-over'
- by selecting parents at random and recombining them, you sometimes get the *best of both* and produce and improved state
- food for thought: How would you perform recombination between 2 tours for the TSP to generate a child state?

```
function GENETIC-ALGORITHM($population$, $fitness$) returns an individual
  repeat
     $weights \leftarrow$ WEIGHTED-BY($population$, $fitness$)
     $population2 \leftarrow$ empty list
     for $i$ = 1 to SIZE($population$) do
        $parent1$, $parent2 \leftarrow$ WEIGHTED-RANDOM-CHOICES($population$, $weights$, 2)        based on fitness
        $child \leftarrow$ REPRODUCE($parent1$, $parent2$)
        if (small random probability) then $child \leftarrow$ MUTATE($child$)
        add $child$ to $population2$
     $population \leftarrow population2$
  until some individual is fit enough, or enough time has elapsed
  return the best individual in $population$, according to $fitness$

function REPRODUCE($parent1$, $parent2$) returns an individual
  $n \leftarrow$ LENGTH($parent1$)
  $c \leftarrow$ random number from 1 to $n$
  return APPEND(SUBSTRING($parent1$, 1, $c$), SUBSTRING($parent2$, $c + 1$, $n$))
```

Genetic Algorithms

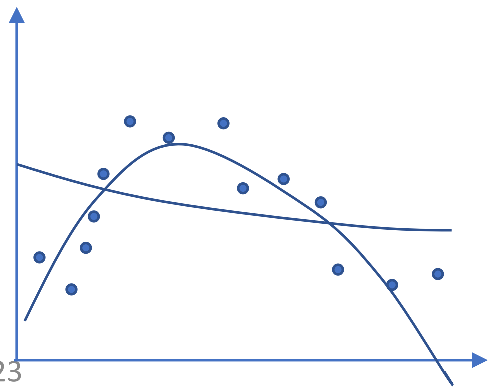
- there are many variations on GAs
- some include *mutation*
  - make random changes to state (like operator) at low frequency
- Lamarckian evolution – improvements/adaptation acquired during lifetime of individual can be passed on to offspring
- ‘loss of diversity’ is a problem for GAs, where population becomes homogeneous (everybody on the same hill)

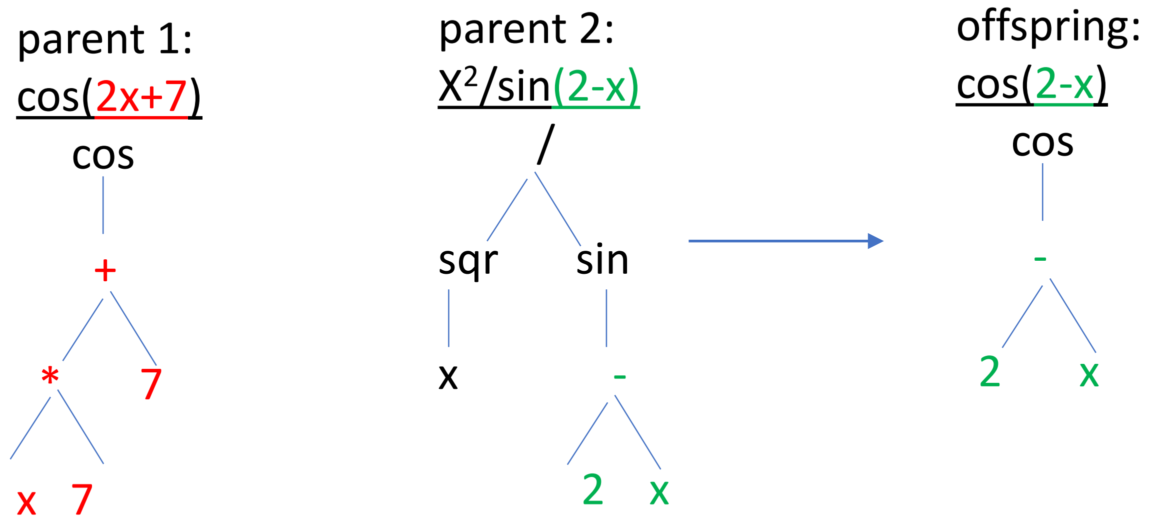
Genetic Algorithms

- many applications of GAs to search problems,
    - from airfoil design (airplane wings)
    - to automatic program synthesis (random computation trees)
- optimization:
    - the power comes not from mutation, but from **competition**
    - survival of the fittest drives the population as a whole to gradually improve
    - weaker/less fit individuals do not get selected to reproduce and are effectively dropped from the population

Automatic Program Synthesis

- given a set of example inputs and outputs, generate a function that does the same
  - objective function: number of "errors" on outputs (or mean-squared error for numerics)
- represent "functions" or "programs" as abstract syntax trees
  - algebraic functions for mathematical expressions
  - pseudocode for programs
- cross-over operator generates an offspring from 2 parents by <u>swapping subtrees</u>
- for numeric problems, GA search is an alternative to non-linear regression or neural networks

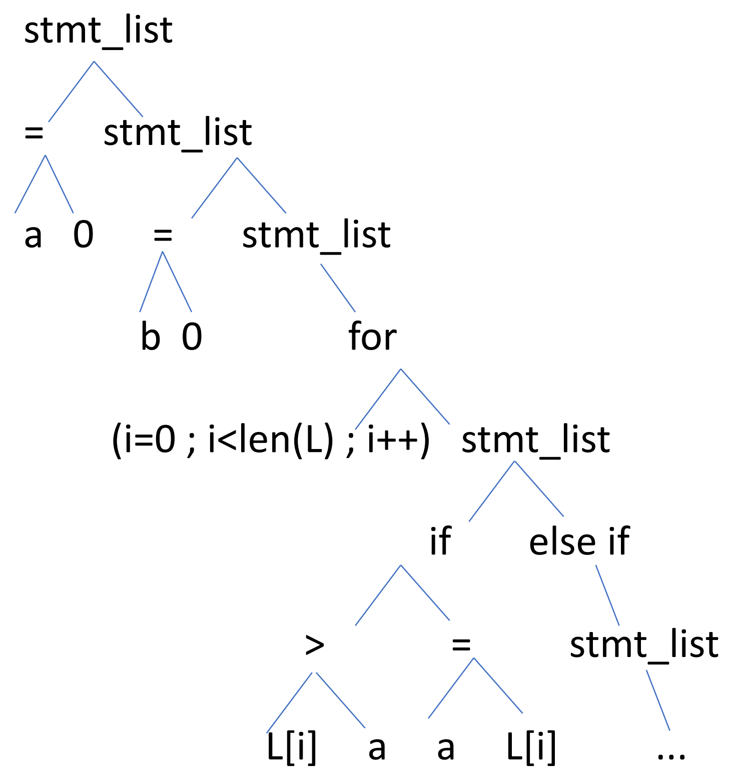




Programs can be synthesized by GA search too...

- Search the space of functions (as syntax trees) to find one that reproduce outputs from example inputs.
- For example, what function finds the 2nd highest value in a list?
- In a population of "random" functions, many will be bad. But can we build better functions incrementally by swapping blocks of statements?

```
f(list L):
  a=0; b=0
  for (i=0 ; i<len(L) ; i++) {
    if L[i]>a: a = L[i]
    else if L[i]>b: b = L[i] }
  return b
```



Summary of Iterative Improvement Algorithms

- Uninformed (Weak) Search
  - Breadth-first (BFS)
  - Depth-first (DFS)
  - Iterative Deepening (ID)
  - Uniform-cost (UC) – optimal (finds a goal with minimum path cost)
- Informed Search – uses a heuristic h(n)
  - Greedy (Best-first) search
  - A* - optimal (provided heuristic is admissible)
- Iterative Improvement
  - Hill-Climbing
  - Beam search
  - Simulated Annealing – stochastic search
  - Genetic Algorithms – parallel search (with a population of candidate solutions)