COSC 4368
Fundamentals of Artificial Intelligence

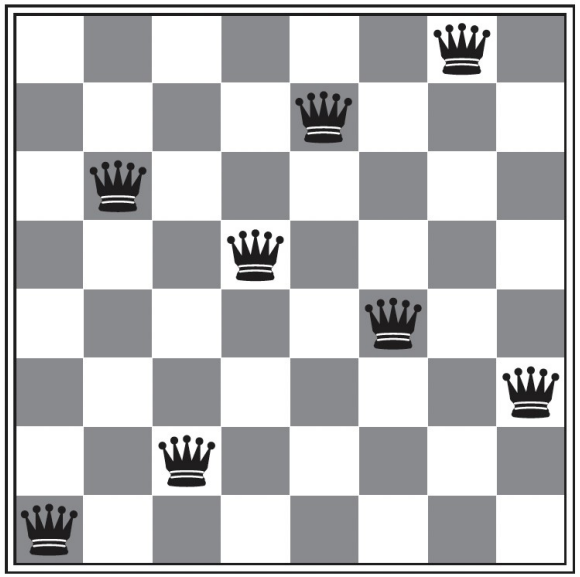
Search 4 – Local Search
September 11th, 2023

Path vs State Optimization

- Previous lectures: the path to the goal is also a solution to the problem
  - Systematic exploration of the search space

- This lecture: a state is a solution to the problem
  - The path to the goal is irrelevant in many problems, e.g., the 8-queens problem
  - State space = a set of "complete" configurations
  - Find a complete configuration satisfying constraints

- Local search algorithms can be used



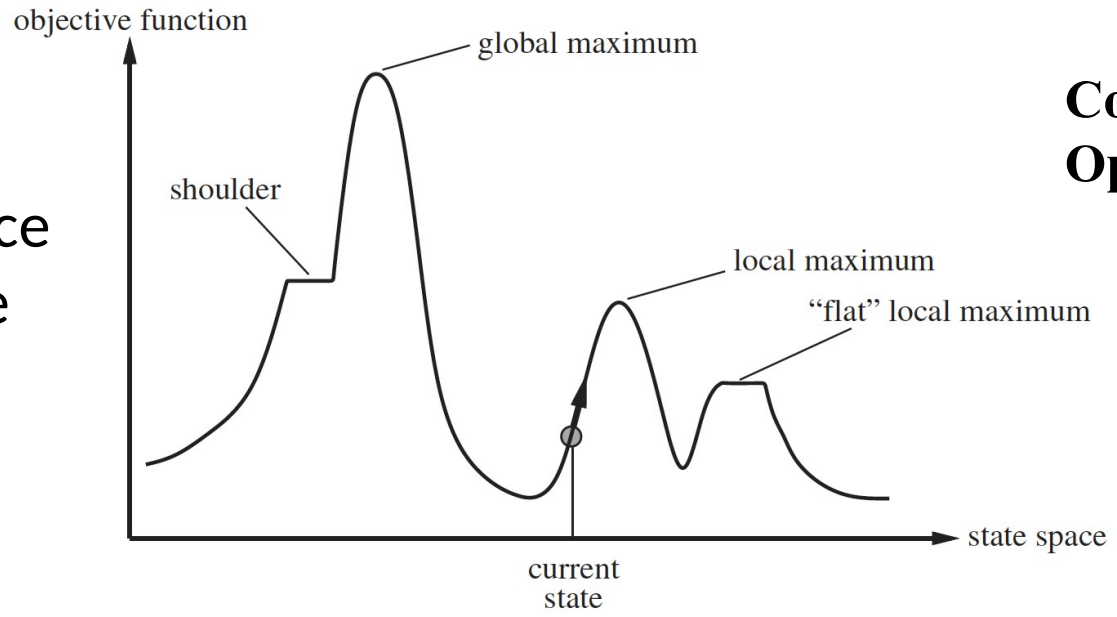
Local Search

- Local search
  - Keep track of a single current node (rather than multiple paths)
  - Move only to neighbors of that node
  - Paths are not retained
- Advantages
  - Use very little memory
    - Keeps only one or a few states; generally a constant amount
  - Can often find reasonable solutions in large or infinite (continuous) state spaces
- Pure optimization problems
  - Goal is to find the best state according to an objective function (max or min)
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems

Optimization Problem

- Pure optimization problems
  - Denote state value as S
  - An objective function f(S) which evaluates the value of the state S
  - Goal: find the best state S according to the objective function (max or min)

State-space landscape



Complete: always finds a goal if one exists
Optimal: always finds a global maximum/minimum

Basic Idea of Local Search (many variations)

```
// initialize to something, usually a random initial state
// alternatively, might pass in a human-generated initial state
best_found ← current_state ← RandomState()
// now do local search
loop do
    if (tired of doing it) then return best_found
    else
        current_state ← MakeNeighbor( current_state )
        if ( Cost(current_state) < Cost(best_found) ) then
            // keep best result found so far
            best_found ← current_state
```

You, as algorithm designer, write the functions named in red.

Typically, "tired of doing it" means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc.
It may also mean that result improvements are small and infrequent, e.g., less than 0.1% result improvement in the last week of run time.

Hill-climbing 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
```

maximize value vs minimize cost

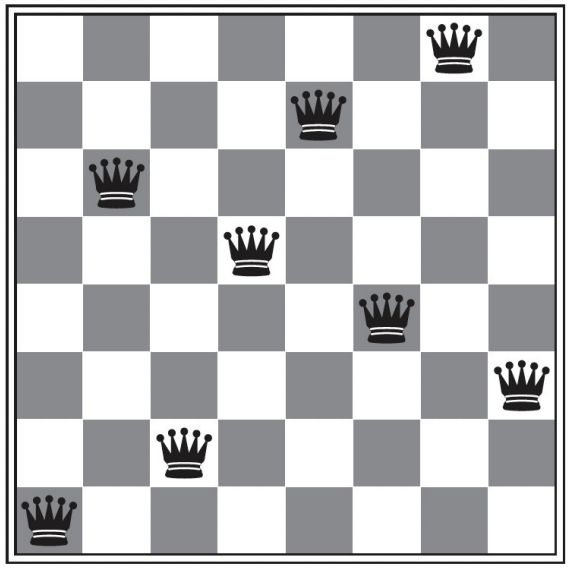
- A loop that continuously moves in the direction of increasing value (steepest-ascent version)
  - Objective function value or heuristic function value
- Terminates when it reaches a "peak" where no neighbor has a higher value
- No search tree, only record the state and the value of the objective function
- Aka Greedy local search
  - Does not look beyond immediate neighbors of the current state

Example: the 8-queens problem

- State: complete-state formulation
  - All 8 queens on the board in some configuration, one per column

- Successor function
  - Move a single queen to another square in the same column

- Example of a heuristic cost function :
  - The number of pairs of queens that are attacking each other
  - Choose the best successors to minimize the value of



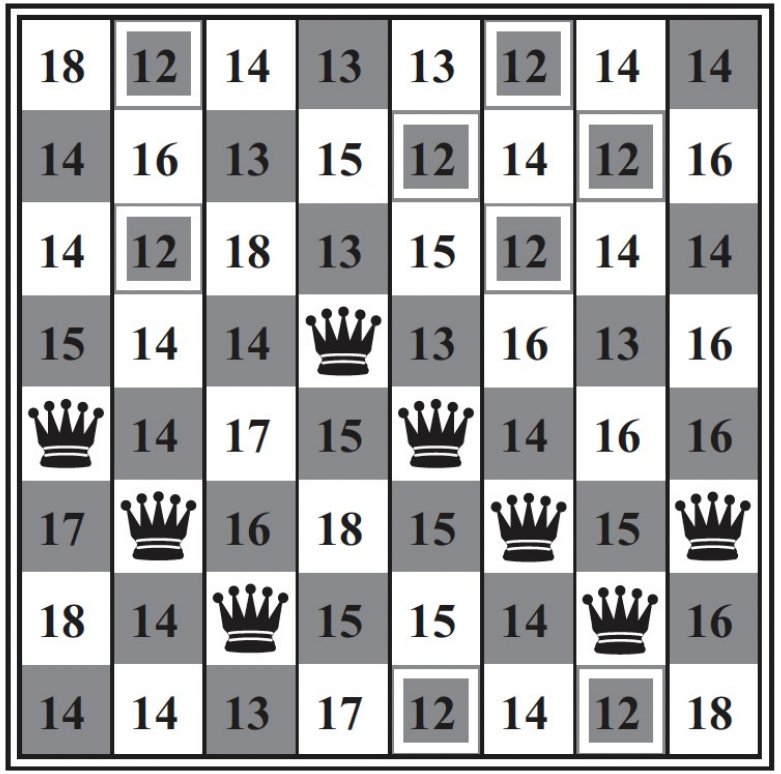
Example: the 8-queens problem

- : # of pairs of queens that are attacking each other

- for the current state

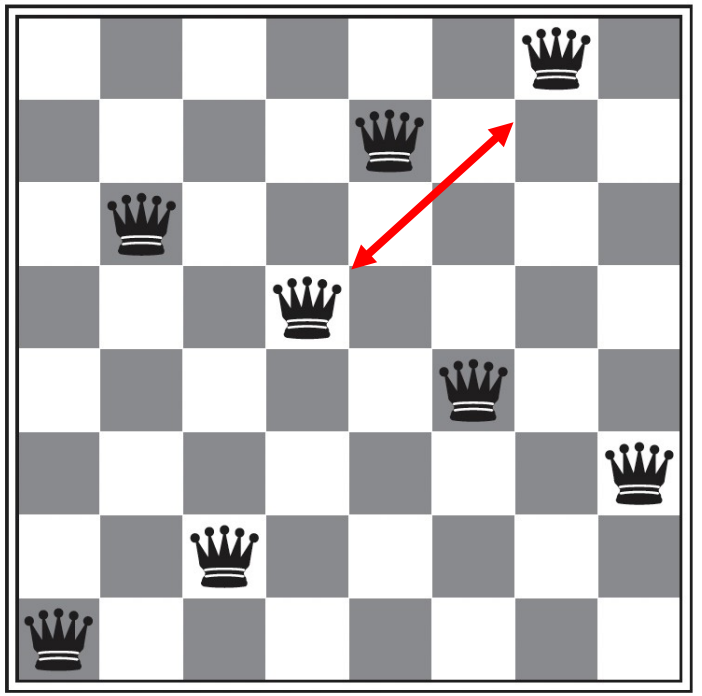
- Each number indicates the value of if we move a queen in its column to that square

- : best among all successors, select one randomly



Example: the 8-queens problem

- Takes just 5 steps to reach from the previous state with
- A local minimum with , nearly a solution
- All one-step neighbors have higher values



Hill-climbing on 8-queens

- Start from a randomly generated 8-queen state

- Pro: works quickly
  - Takes only 4 steps on average when it succeeds
  - 3 steps on average when it gets stuck
  - (for a state space with around 17 million states)

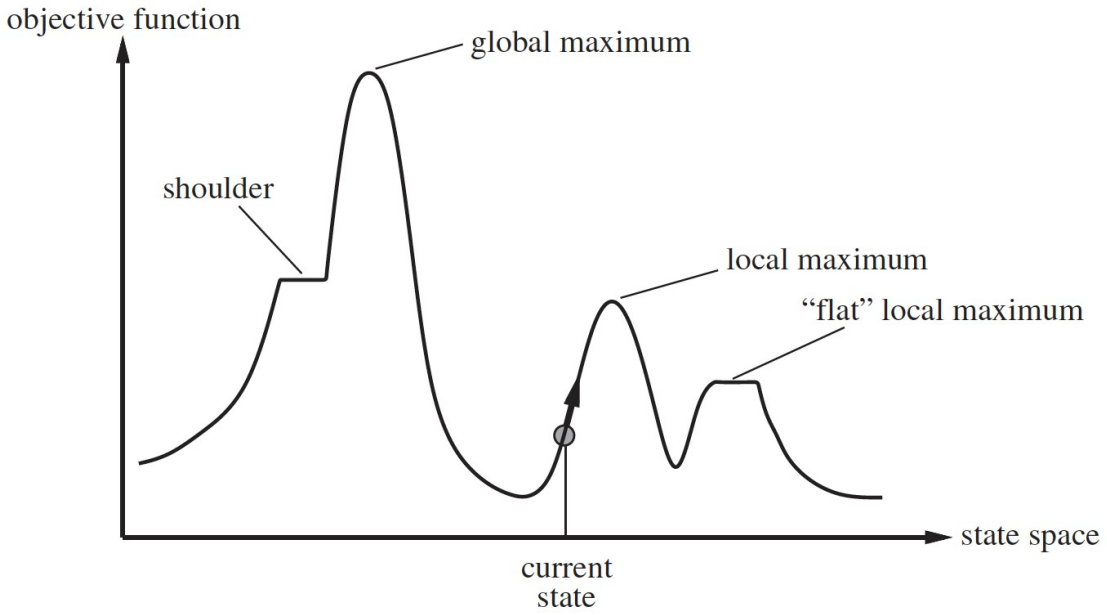
- Con: gets stuck easily
  - 14% of the time it solves the problem
  - 86% of the time it gets stuck

Why it gets stuck?

Local Search Difficulties

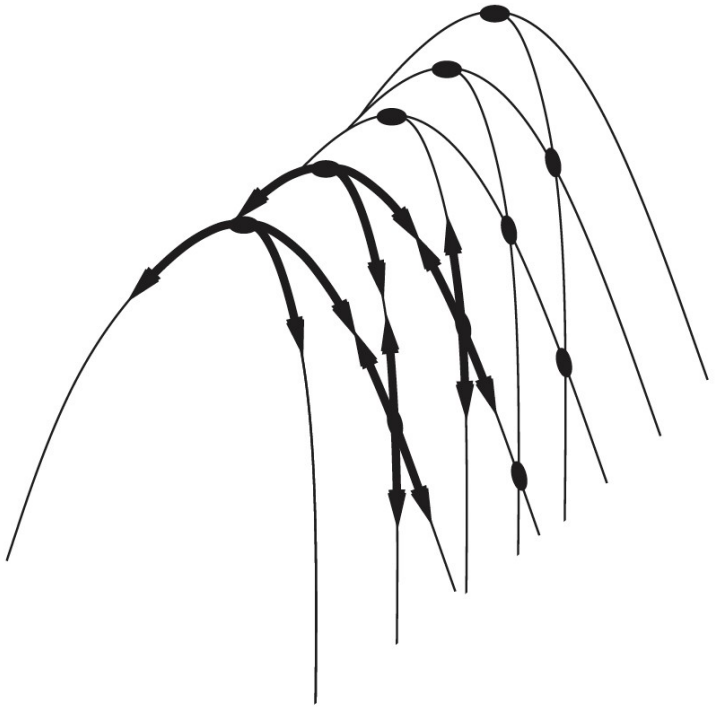
- Local optima
  - E.g., local maximum, a peak that is higher than each of its neighbors but lower than the global maximum

- Plateau: A flat area of the state-space landscape
  - E.g., flat local maximum, shoulder



Local Search Difficulties

- Ridges: a sequence of local maxima that is very difficult for greedy algorithms to navigate
  - A sequence of local maxima that are not directly connected to each other
  - Every neighbor appears to be downhill
  - But the search space has an uphill



These difficulties apply to all local search algorithms, and become much more difficult as the search space increases to high dimensionality

How to Escape Local Optima

- **Sideways move:** if no uphill moves, allow sideways moves in hope that the algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops

- For 8-queens
  - Allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14% to 94%

  - However…
    - Average 21 steps for every successful instance
    - 64 steps for each failure

How to Escape Local Optima

- **Stochastic (randomized) hill climbing:**
  - Iterate the process of randomly selecting a neighbor for a candidate solution
  - Accept it only if it results in an improvement
    - Can generate a set of neighbors for the current state and pick the best one among the selected neighbors

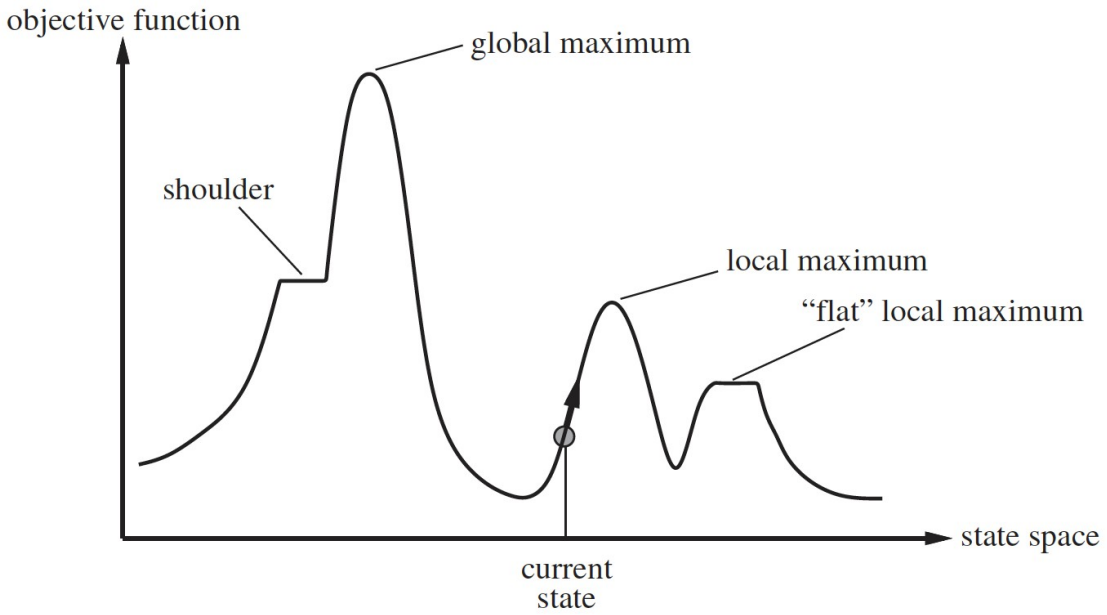
```
Input: Iter_max, ProblemSize
Output: Current
Current ← RandomSolution(ProblemSize);
foreach iter_i ∈ Iter_max do
    Candidate ← RandomNeighbor(Current);
    if Cost(Candidate) ≥ Cost(Current) then
        Current ← Candidate;
    end
end
return Current;
```

How to Escape Local Optima

- **Random restart:**
  - If at first you don’t succeed, try, try again
  - Launch multiple hill-climbing searches from randomly generated initial states
  - Different variations
    - For each restart: run until termination vs. run for a fixed time
    - Run a fixed number of restarts or run indefinitely until a goal is found
  - A meta-algorithm that can encapsulate any local search algorithm, not just hill-climbing
  - Effective on many difficult optimization problems

Simulated Annealing

- When the state-space landscape has local minima, any search that moves only in the greedy direction cannot be complete, e.g., hill-climbing
- On the other hand, a purely random walk is complete but extremely inefficient
  - Randomly select a successor from the set of successors
- Idea: combine hill-climbing with a random walk



Simulated Annealing

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)
```

Instead of picking the best move, pick one randomly

: change in objective function

If is positive, move to that state (like hill-climbing)

Otherwise, move to this state with probability proportional to

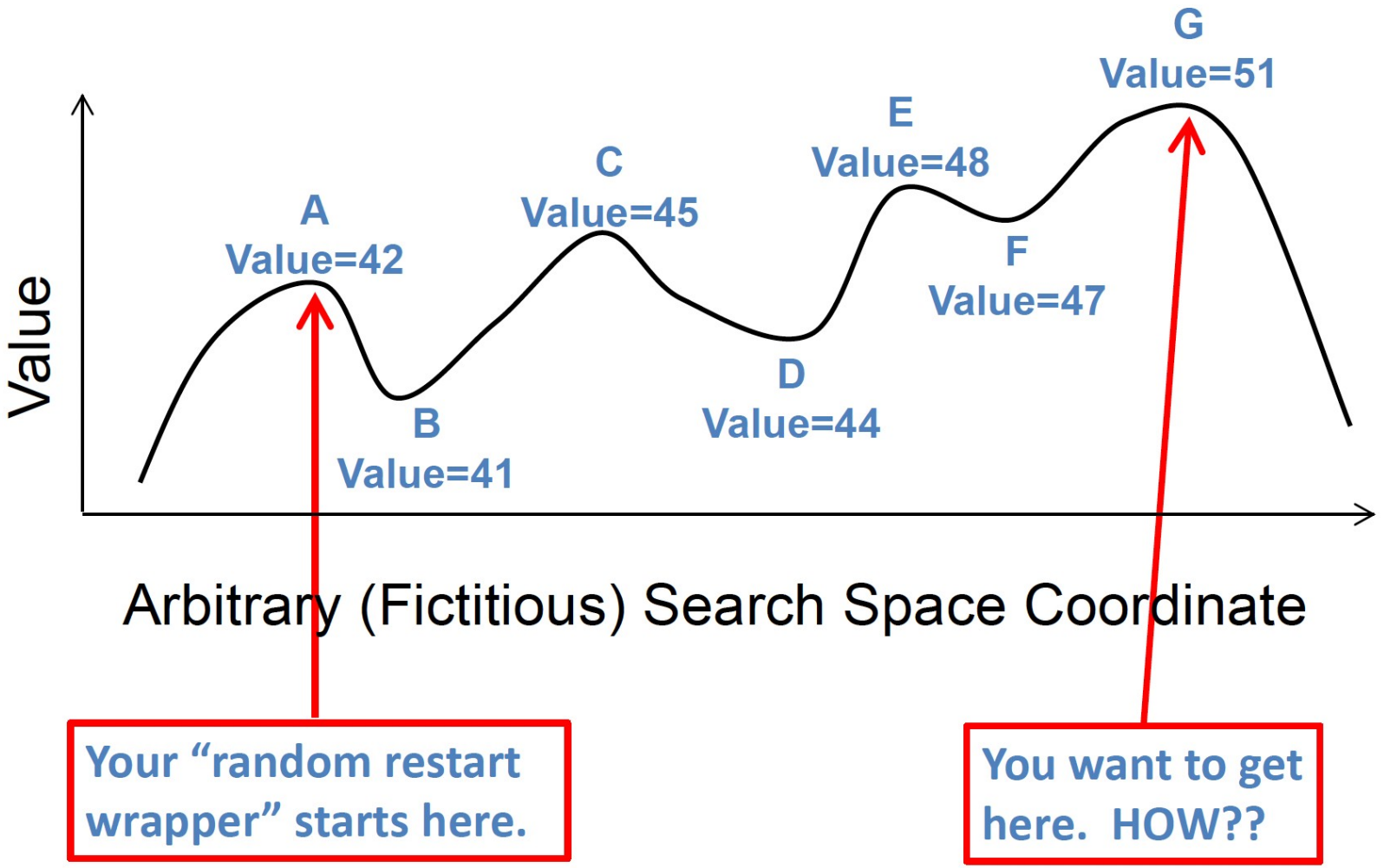
Gradually decrease

Example of Annealing Schedule

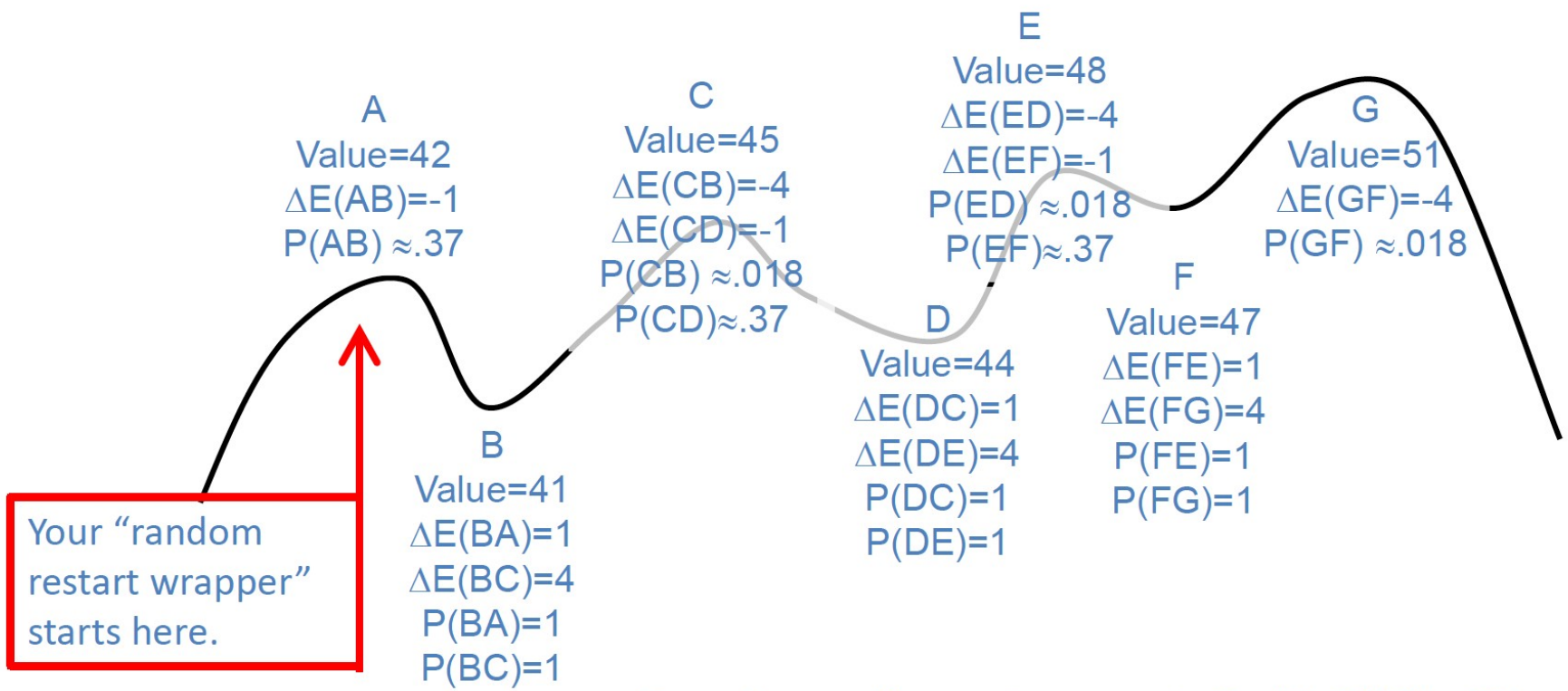
- Assume
- ward move will be accepted with prob
- ward move will be accepted with prob
- ward move will be accepted with prob
- ward move will be accepted with prob

- Usually use a decaying exponential

Example of Annealing Schedule



Example of Annealing Schedule



Fix T=1

x	-1	-4
e^x	≈.37	≈.018

This is an illustrative *cartoon*…

From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

Simulated Annealing in Practice

- Method proposed in 1983 by IBM for solving VLSI layout problems
  - Theoretically will always find the global optimum

- Other applications: traveling salesman, graph partitioning, graph coloring, scheduling,…

- Useful for some problems, but can be very very slow
  - Because T must be decreased slowly enough to retain optimality

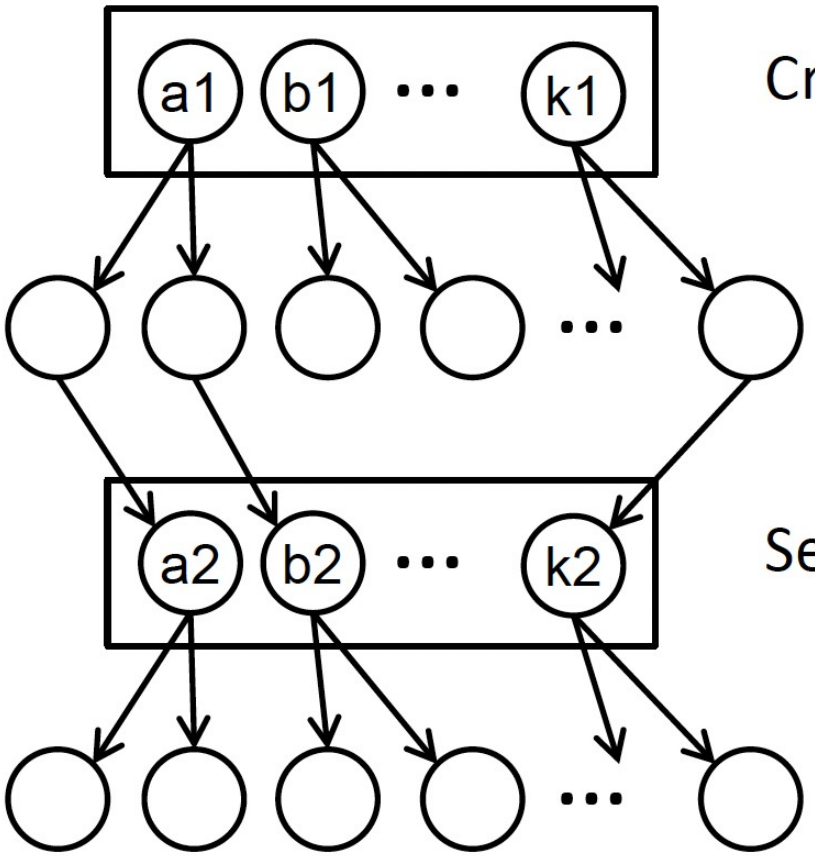
Local Beam Search

- Idea: keeping only one node in memory is an extreme reaction to memory problems

- 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

Local Beam Search

- Idea: keeping only one node in memory is an extreme reaction to memory problems



Create *k* random initial states

Generate their children

Select the *k* best children

Repeat indefinitely…

Local Beam Search

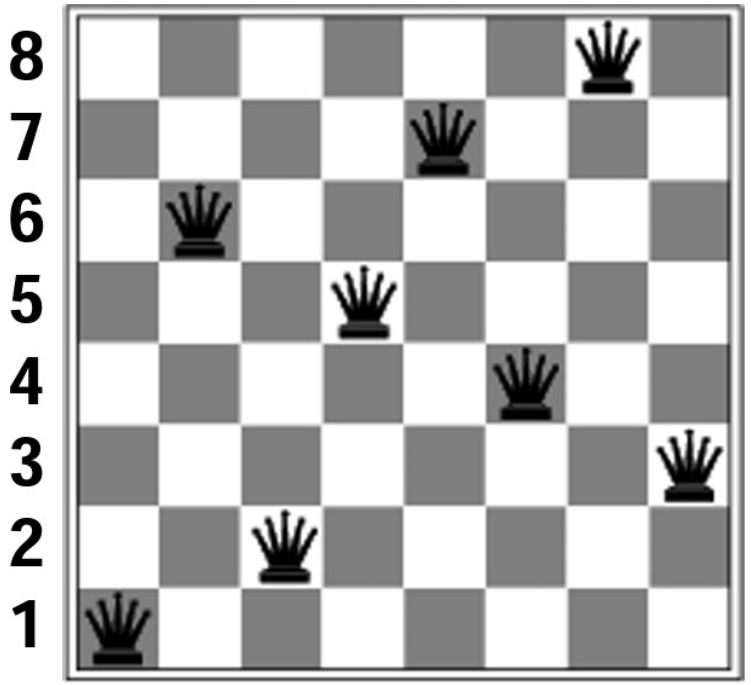
- Not the same as k random-start searches run in parallel
- Searches that find good states recruit other searches to join them

- **Problem:** concentrates search effort in areas believed to be fruitful
  - May lose diversity as search progresses, resulting in wasted effort

- **To improve:** stochastic beam search
  - Chooses k successors randomly, biased towards good ones
  - The probability of choosing a given successor is an increasing function of its value
  - Close analogy to *natural selection*

Genetic Algorithms

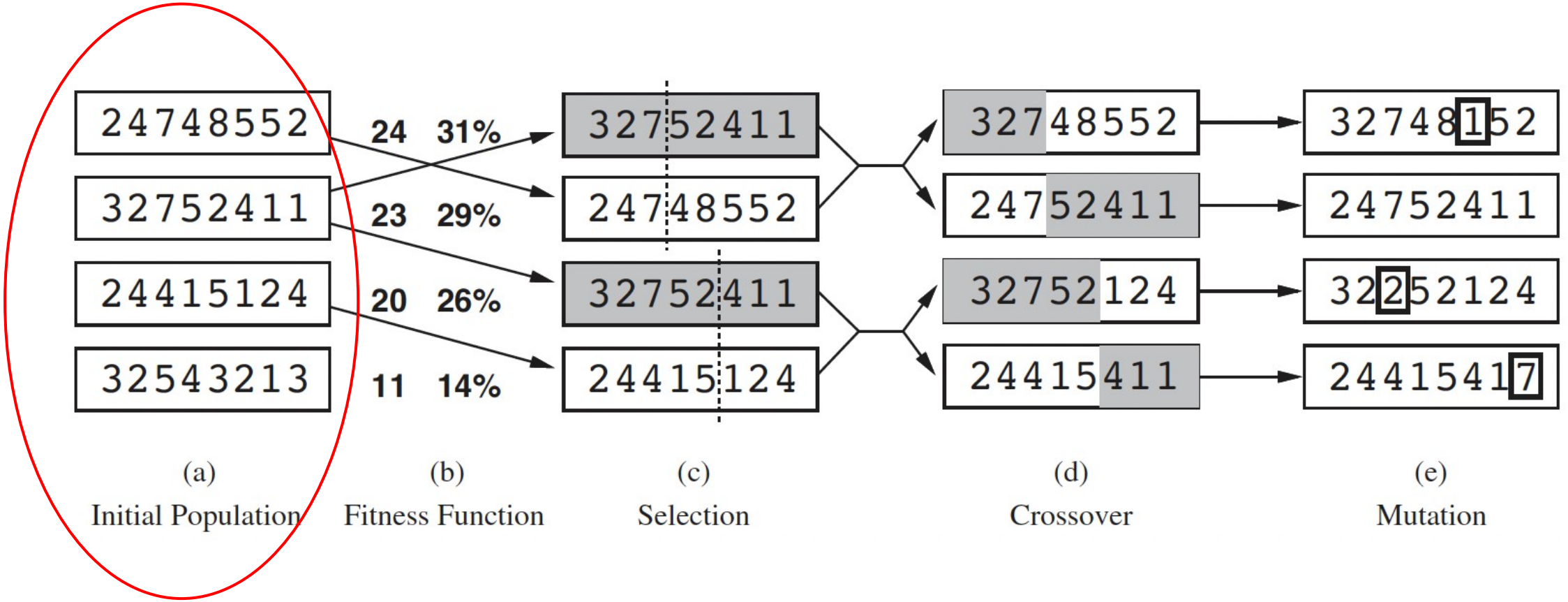
- Variant of stochastic beam search: successor is generated by combining two parent states
- A state: a string over a finite alphabet
  - E.g., 8-queens, state = position of 8 queens, each in a column



State=16257483

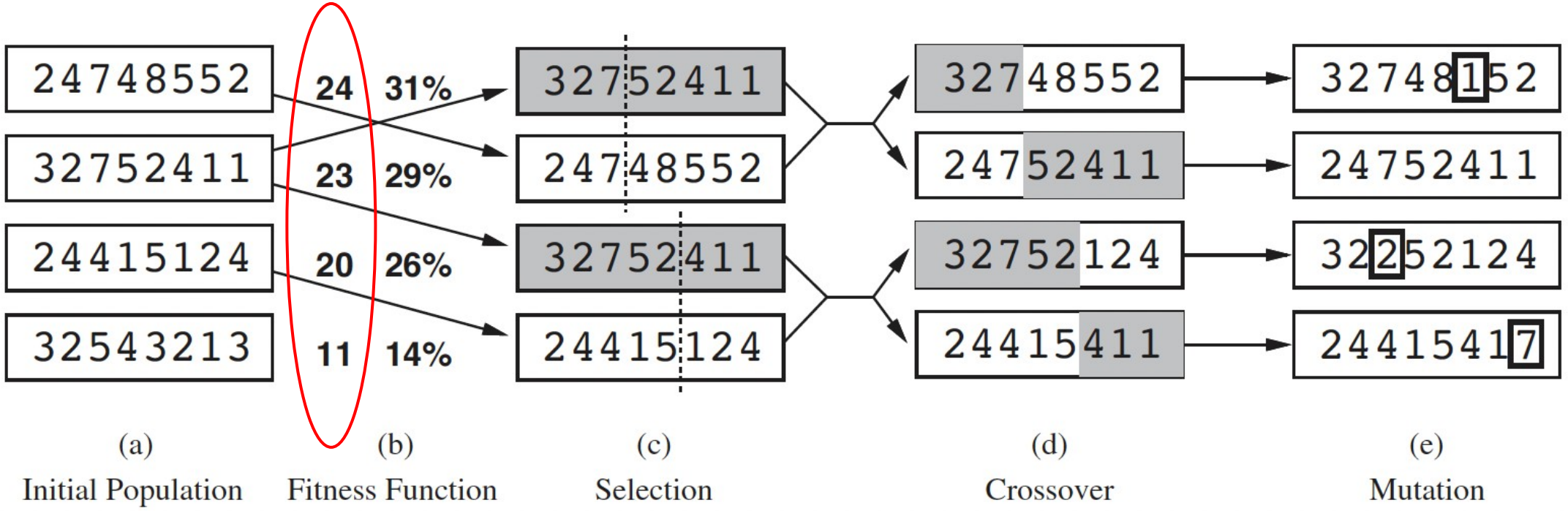
Genetic Algorithms

- Start with a set of k randomly generated states: called the **population**
  - **{24748552, 32752411, 24415124, 32543213}**



Genetic Algorithms

- Each state is rated by the **fitness function** (= our objective function)
  - Higher values for better states
  - For 8-queens: the number of nonattacking pairs of queens (=28 for a solution)



Genetic Algorithms

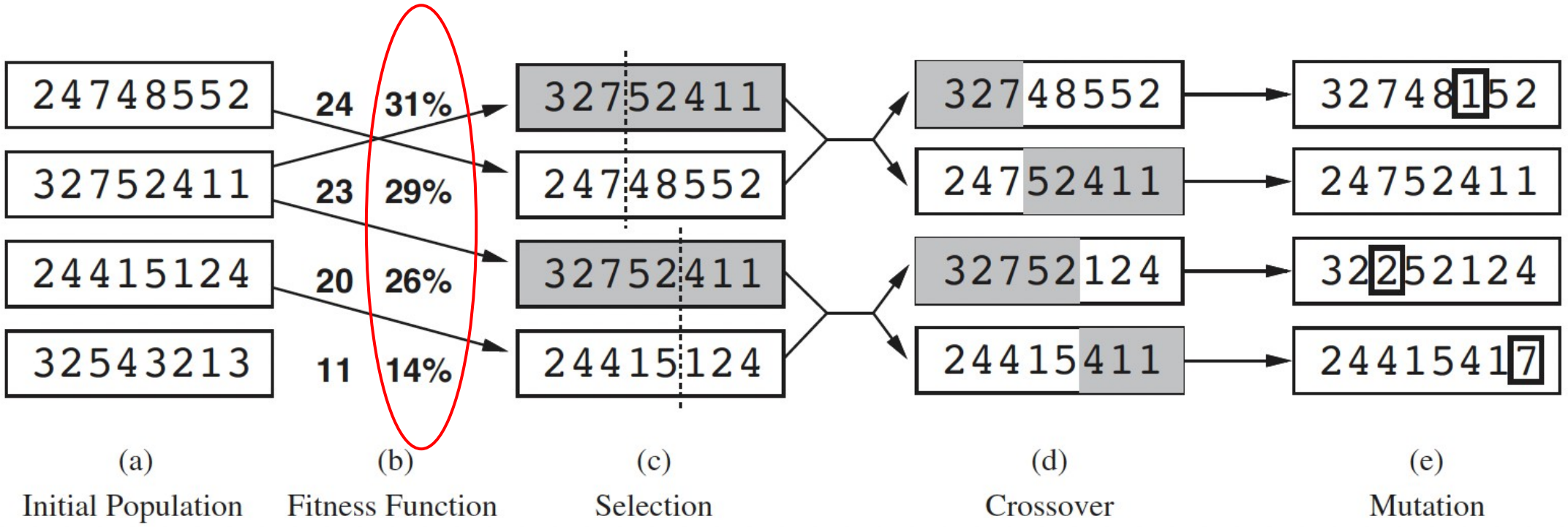
- Produce the next generation of states by "simulated evolution"
  - **Random selection:** select individuals for next generation based on fitness

```
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
```

Probability of being in next generation for individual i:
fitness_i/sum_i(fitness_i)

Genetic Algorithms

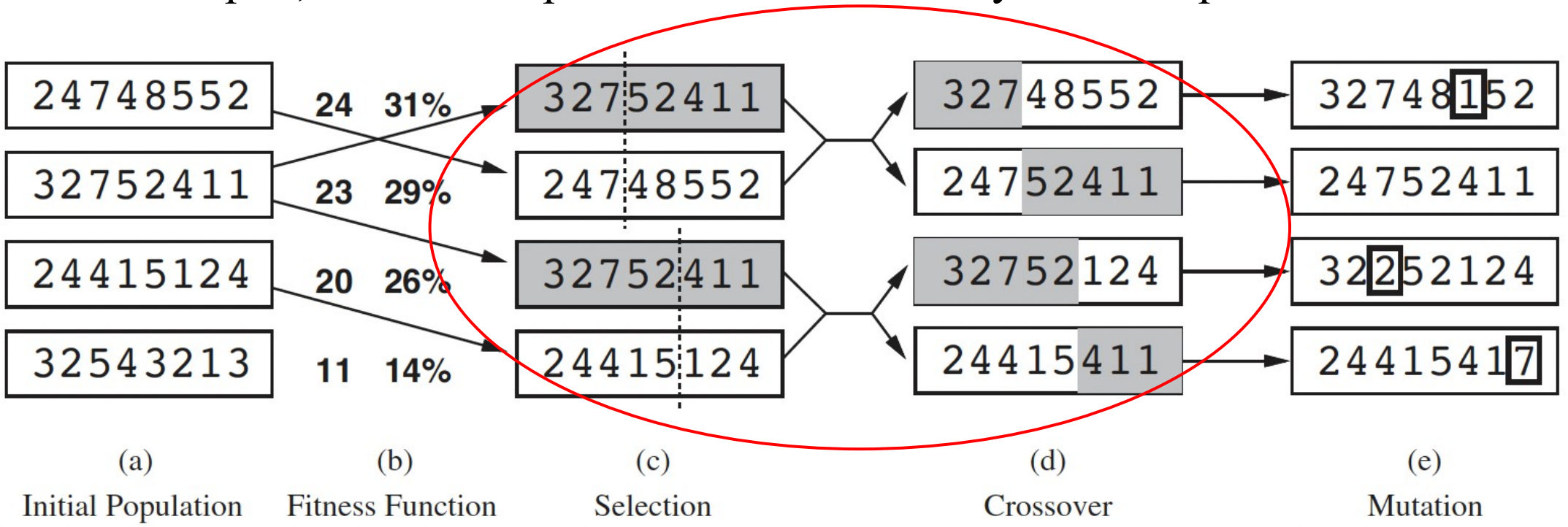
- Produce the next generation of states by "simulated evolution"
  - **Random selection:** select individuals for next generation based on fitness



Probability of being in next generation: fitness_i/sum_i(fitness_i)
24/(24+23+20+11)=31%, 23/(24+23+20+11)=29%, etc.

Genetic Algorithms

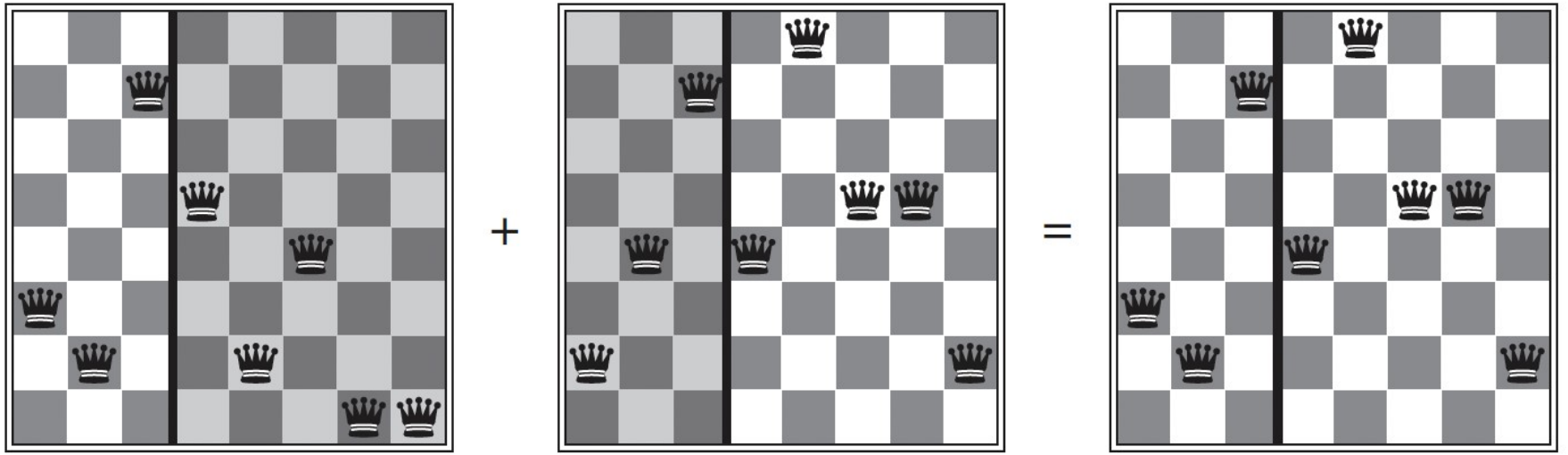
- Produce the next generation of states by "simulated evolution"
  - **Crossover:** fit parents to yield next generation
  - For each pair, a crossover point is chosen randomly from the positions in the string



The offspring are created by crossing over the parent strings at the crossover point

Genetic Algorithms

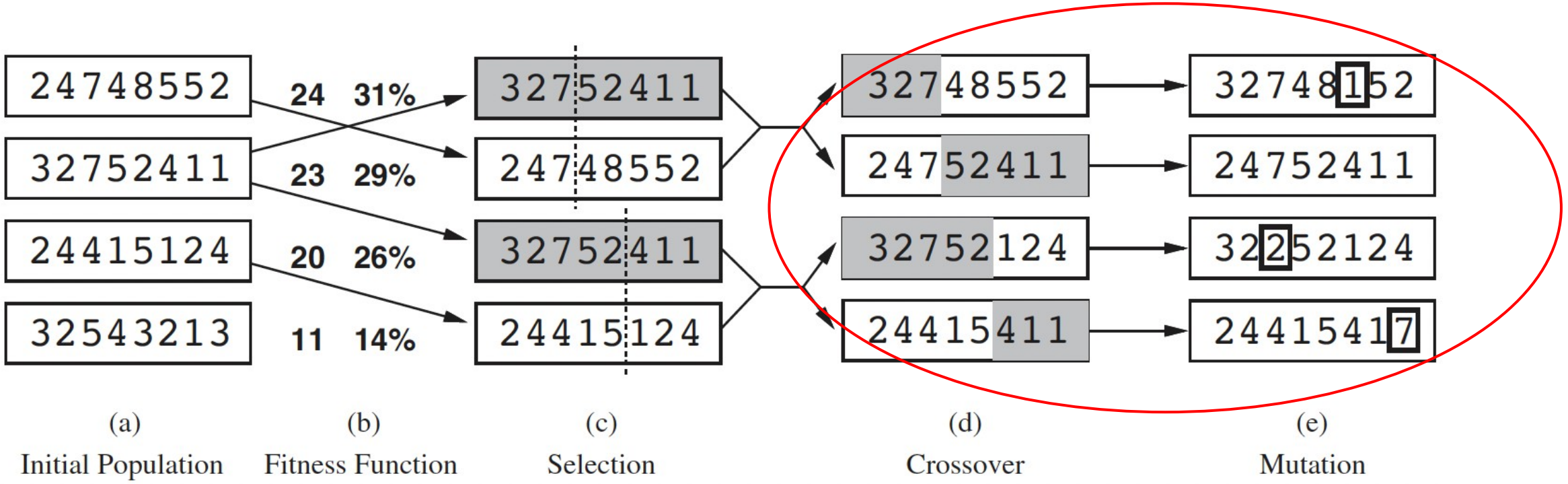
- Produce the next generation of states by "simulated evolution"
  - **Crossover:** fit parents to yield next generation



- Has the effect of "jumping" to a completely different new part of the search space (quite non-local)
- Large steps in early stages (diverse population) and smaller steps later (similar population)

Genetic Algorithms

- Produce the next generation of states by "simulated evolution"
  - **Random mutation:** mutate each location in each offspring with a small independent probability



In 8-queens, correspond to choosing a queen at random and moving it to a random square in its column

Genetic Algorithms

- Positive points
  - Random exploration (via crossover) can find solutions that previous local strategies can’t
  - Appealing connection to human evolution, like “neural” networks

- Negative points
  - Difficult to replicate performance from one problem to another
    - The conditions under which genetic algorithms perform well are still not clear
  - Useful on some set of problems but no convincing evidence that genetic algorithms are better than hill-climbing with random restarts in general
  - Crossover are helpful if the substrings are meaningful components

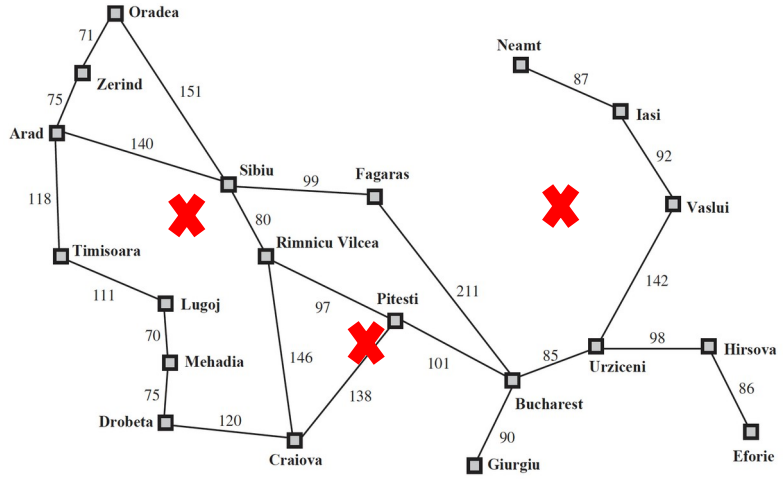
Overview

- Local search
  - Keep track of a single current node (or multiple nodes) rather than multiple paths
  - Three components:
    - Initial state selection, neighbor selection, objective function evaluation
  - Hill-climbing search
    - Move in the direction of increasing value (maximum version)
    - Local optima
  - Sideways move, stochastic hill-climbing, random restart
  - Simulated annealing: allow some bad moves but gradually decrease the frequency
  - Local beam search
  - Genetic algorithms
- Today
  - Local search in continuous space
  - Constraints satisfaction problem

Local Search in Continuous Spaces

- Example: place three new airports anywhere in Romania, such that the sum of squared distances from each city to its nearest airport is minimized
- **State space:** coordinates of the airports:
  - States are defined by six variables
- Given the set of cities whose closet airport is airport , the objective function is

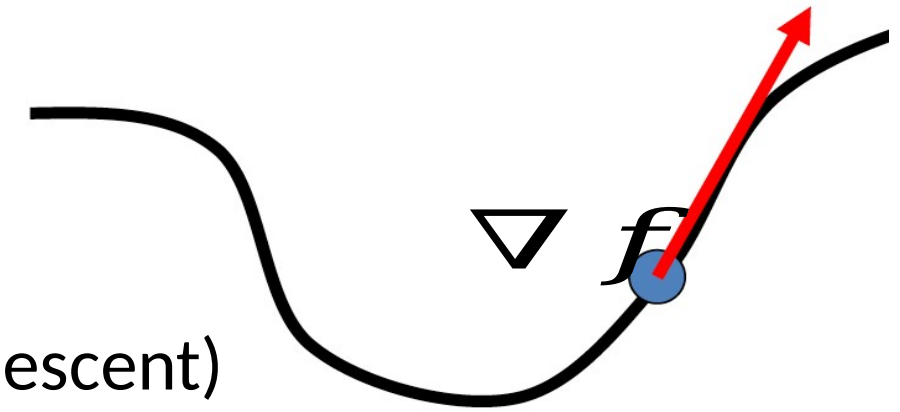
$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$



Local Search in Continuous Spaces

- Solutions:
  - Discretize the state space
    - Any local search algorithms described previously can be used
  - Use stochastic hill-climbing and simulated annealing directly
    - Choose successors randomly ---- generate random vectors with some length
  - Gradient-based approaches
    - The gradient of the objective function is a vector that gives the magnitude and direction of the steepest slope

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



Maximize f, use gradient (steepest ascent)

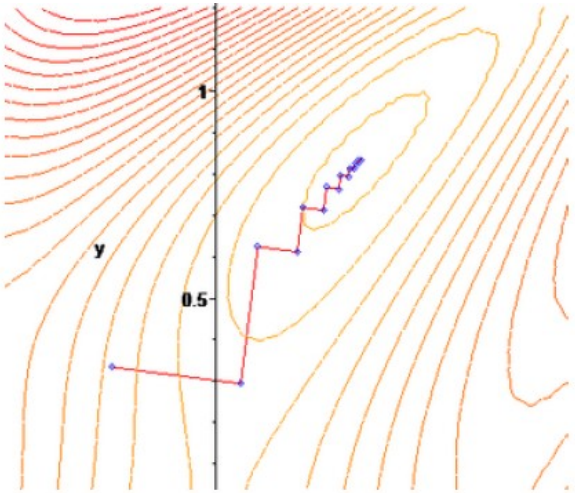
Minimize f, use negative gradient (steepest descent)

Gradient Descent

- Assume we have a continuous cost function
- We want to minimize it over continuous variables

1. Compute the gradient with respect to every :
2. Take a small step downhill in the opposite direction of gradient with step size :

3. Repeat



Summary

- Hill climbing is a steady monotonous ascent to better nodes
- Simulated annealing, local beam search, and genetic algorithms are “random” searches with a bias towards better nodes
- Gradient descent works well for continuous space
- All need very little memory
- None guarantees to find the globally optimal solution