CS 4700: Foundations of Artificial Intelligence

Local search

Instructor: Kevin Ellis

Cornell University

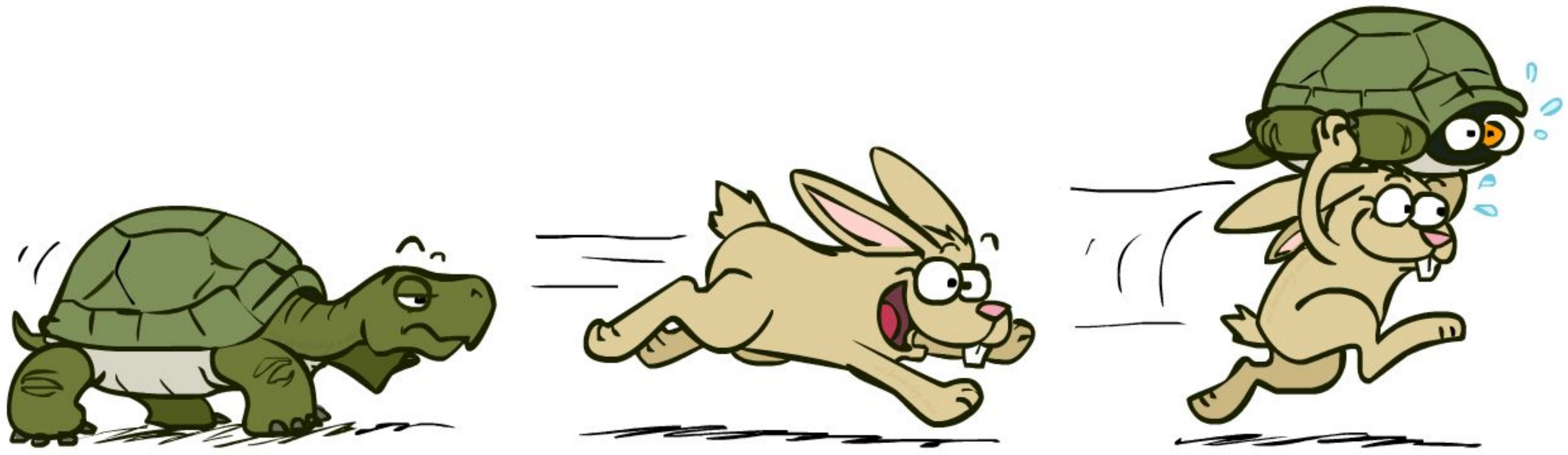
[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Informed Search Recap

A*: Summary

A*: Summary

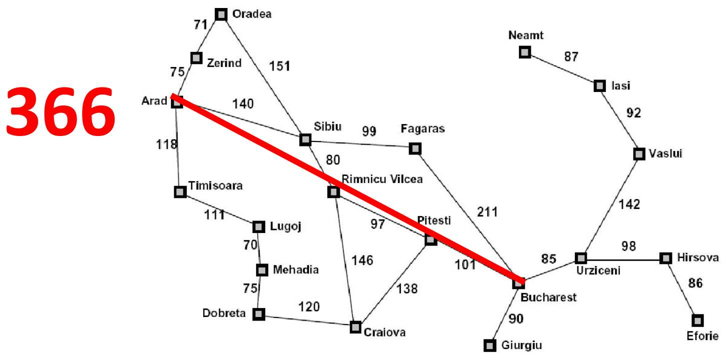
- A* uses both backward costs and (estimates of) forward costs
  - f(n) = g(n)   + h(n)
  - = cost-so-far + heuristic-estimate-of-distance-to-goal
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems

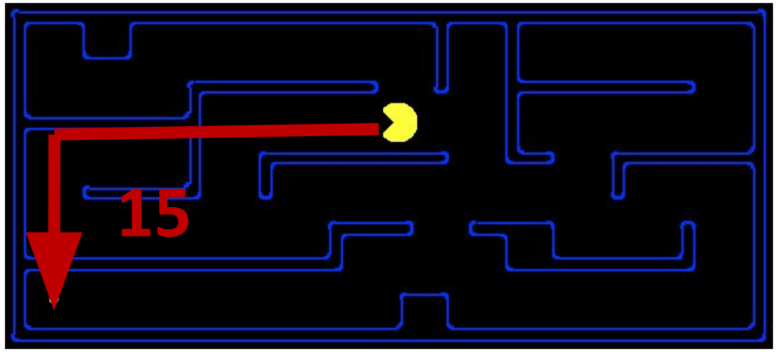


Creating Admissible Heuristics

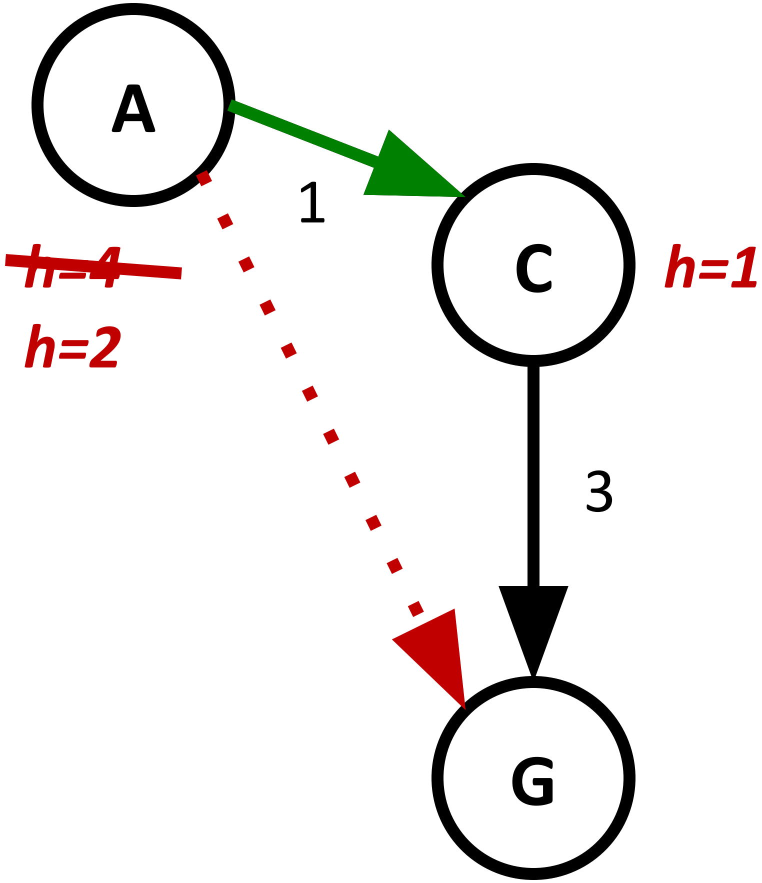
- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics

- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available





Creating Admissible+Consistent Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G

  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)

- Consequences of consistency:
  - A* graph search is optimal

Tree Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Graph Search Pseudo-Code

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Local search

Plan for today

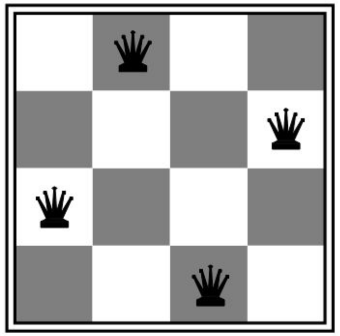
Overview of local search algorithms

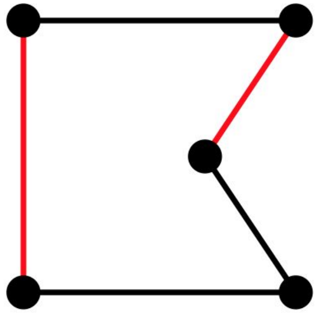
Gradient descent

Genetic algorithms at Cornell

Local search algorithms

- In many optimization problems, ***path*** is irrelevant; the goal state ***is*** the solution
- Then state space = set of "complete" configurations;
  find ***configuration satisfying constraints***, e.g., n-queens problem; or, find ***optimal configuration***, e.g., travelling salesperson problem





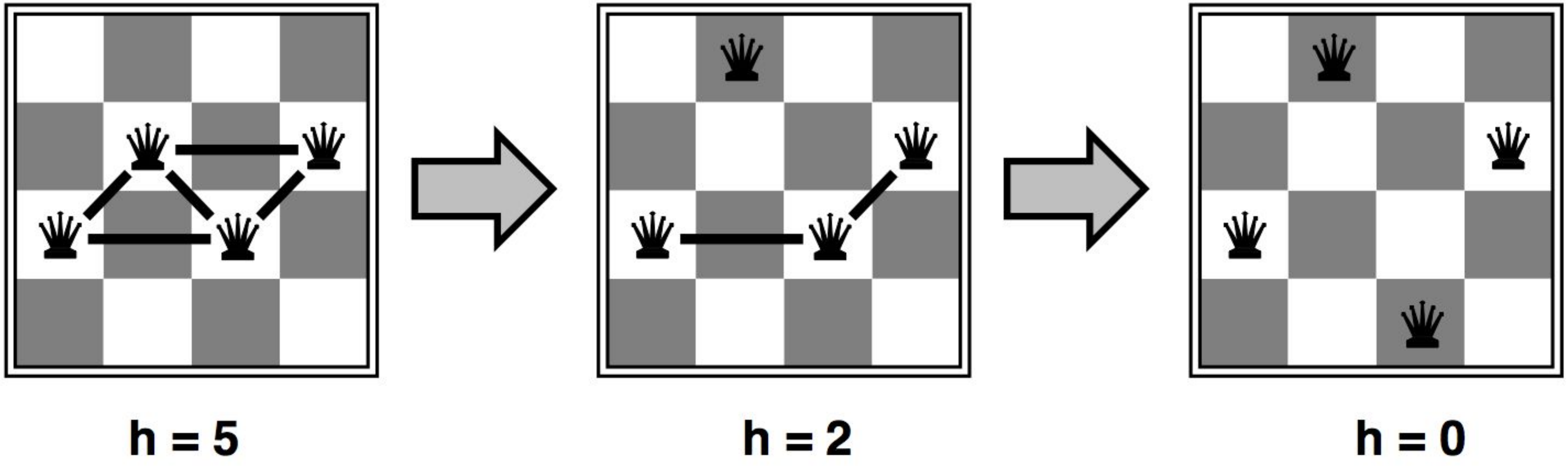
- In such cases, can use ***iterative improvement*** algorithms: keep a single "current" state, try to improve it
- Constant space, suitable for online as well as offline search
- More or less unavoidable if the "state" is yourself (i.e., learning)

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

Heuristic for *n*-queens problem

- Goal: n queens on board with no ***conflicts***, i.e., no queen attacking another
- States: n queens on board, one per column
- Actions: move a queen in its column
- Heuristic value function: number of conflicts



Hill-climbing algorithm

function HILL-CLIMBING(problem) **returns** a state

  current ← make-node(problem.initial-state)

  loop do

    neighbor ← a highest-valued successor of current

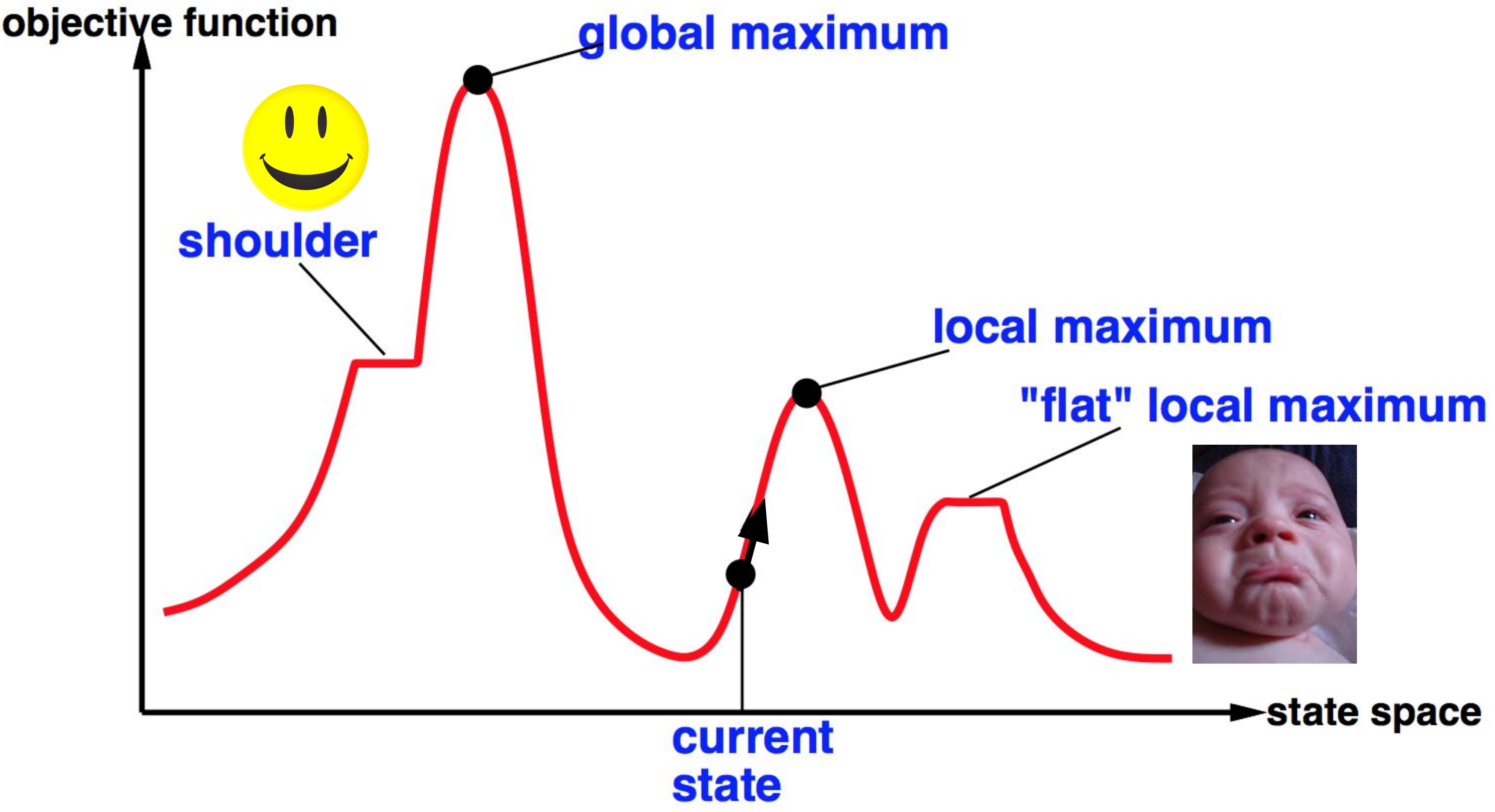
    if neighbor.value ≤ current.value **then**

      return current.state

    current ← neighbor

"Like climbing Everest in thick fog with amnesia"

Global and local maxima



Random restarts

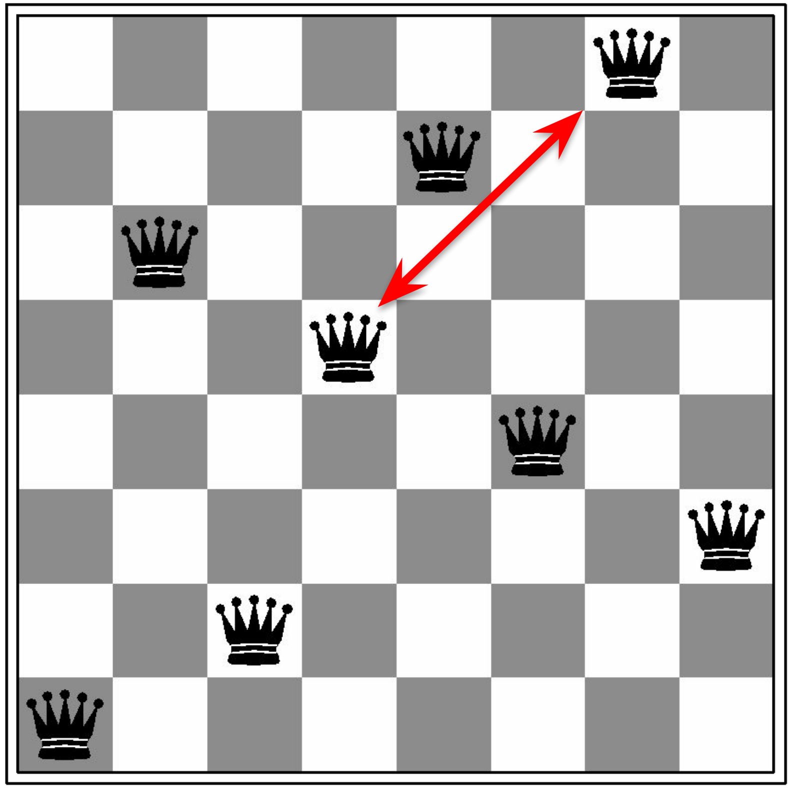
- find global optimum
- duh

Random sideways moves

- Escape from shoulders
- Loop forever on flat local maxima

Hill-climbing on the 8-queens problem

- No sideways moves:
  - Succeeds w/ prob. 0.14
  - Average number of moves per trial:
    - 4 when succeeding, 3 when getting stuck
  - Expected total number of moves needed:
    - 3(1-p)/p + 4 =~ 22 moves
- Allowing 100 sideways moves:
  - Succeeds w/ prob. 0.94
  - Average number of moves per trial:
    - 21 when succeeding, 65 when getting stuck
  - Expected total number of moves needed:
    - 65(1-p)/p + 21 =~ 25 moves



Moral: algorithms with knobs to twiddle are irritating

Simulated annealing

- Resembles the annealing process used to cool metals slowly to reach an ordered (low-energy) state
- Basic idea:
  - Allow “bad” moves occasionally, depending on “temperature”
  - High temperature => more bad moves allowed, shake the system out of its local minimum
  - Gradually reduce temperature according to some schedule
  - Sounds pretty flaky, doesn’t it?

Simulated annealing algorithm

function SIMULATED-ANNEALING(problem,schedule) **returns** a state

current $\leftarrow$ problem.initial-state

for t = 1 **to** ∞ **do**

    T $\leftarrow$schedule(t)

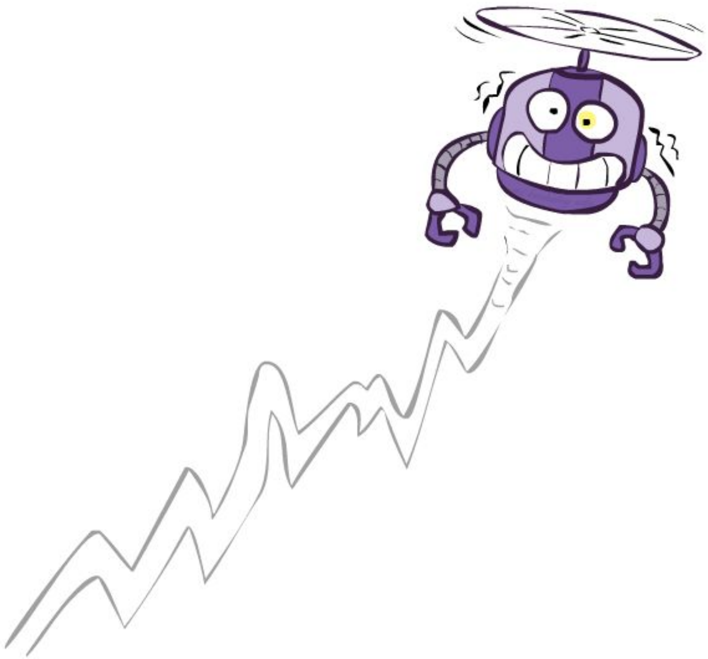
    if T = 0 **then return** current

    next $\leftarrow$ a randomly selected successor of current

    ΔE $\leftarrow$ next.value – current.value

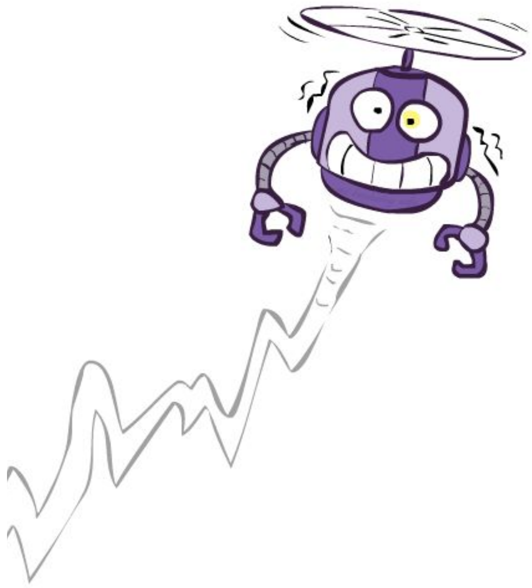
    if ΔE < 0 **then** current $\leftarrow$ next

        else current $\leftarrow$ next only with probability $e^{-\Delta E/T}$

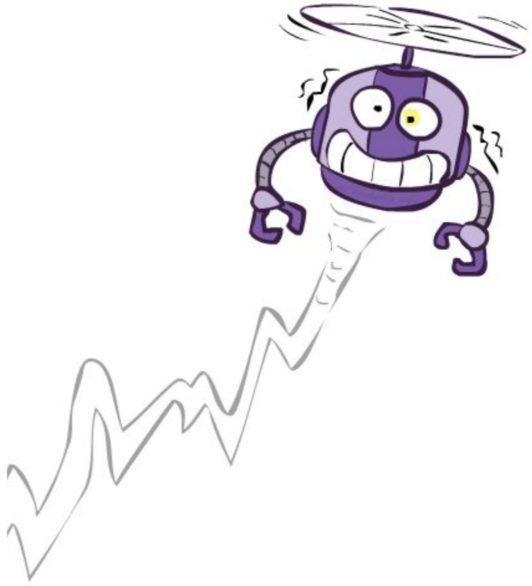


Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution (Boltzmann): $P(x) \propto e^{-E(x)/T}$
  - If T decreased slowly enough, will converge to optimal state!



Simulated Annealing

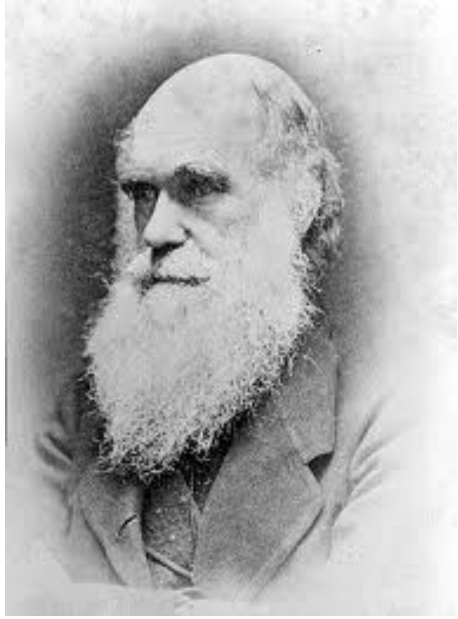


- Is this convergence an interesting guarantee?
- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - "Slowly enough" may mean exponentially slowly
  - Random restart hillclimbing also converges to optimal state…
- Simulated annealing and its relatives are a key workhorse in VLSI layout and other optimal configuration problems

Local beam search

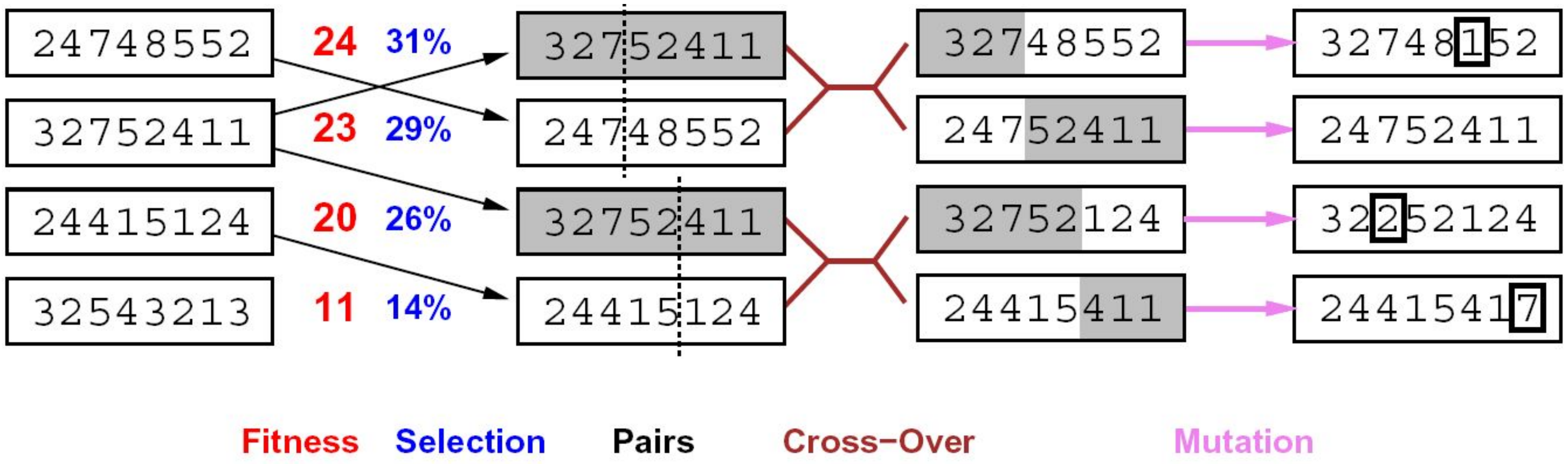
- Basic idea:
  - *K* copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from *K* current states
    - Choose best *K* of these to be the new current states

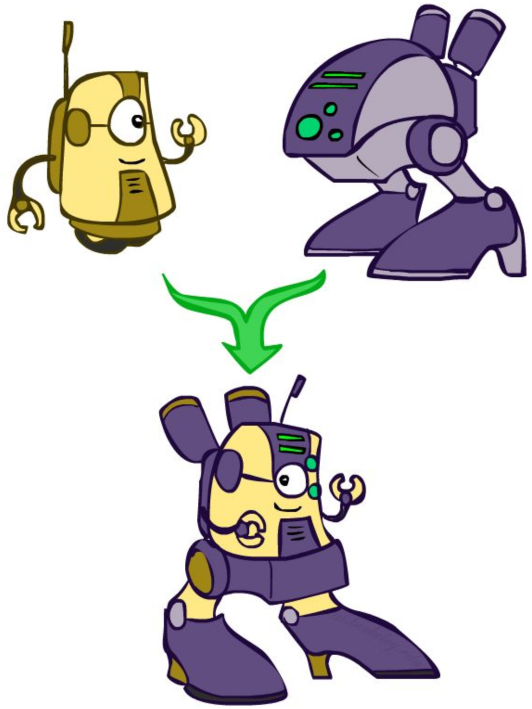
Or, K chosen randomly with a bias towards good ones



- Why is this different from *K* local searches in parallel?
  - The searches ***communicate***! “Come over here, the grass is greener!”
- What other well-known algorithm does this remind you of?
  - Evolution!

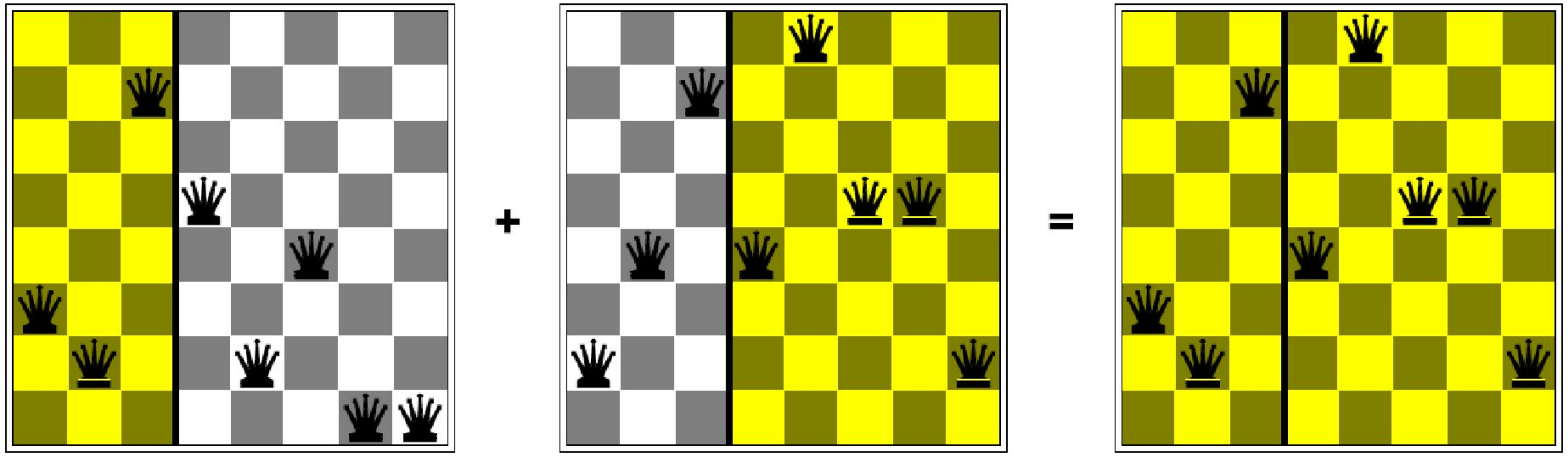
Genetic algorithms





- Genetic algorithms use a natural selection metaphor
  - Resample *K* individuals at each step (selection) weighted by fitness function
  - Combine by pairwise crossover operators, plus mutation to give variety

Example: N-Queens

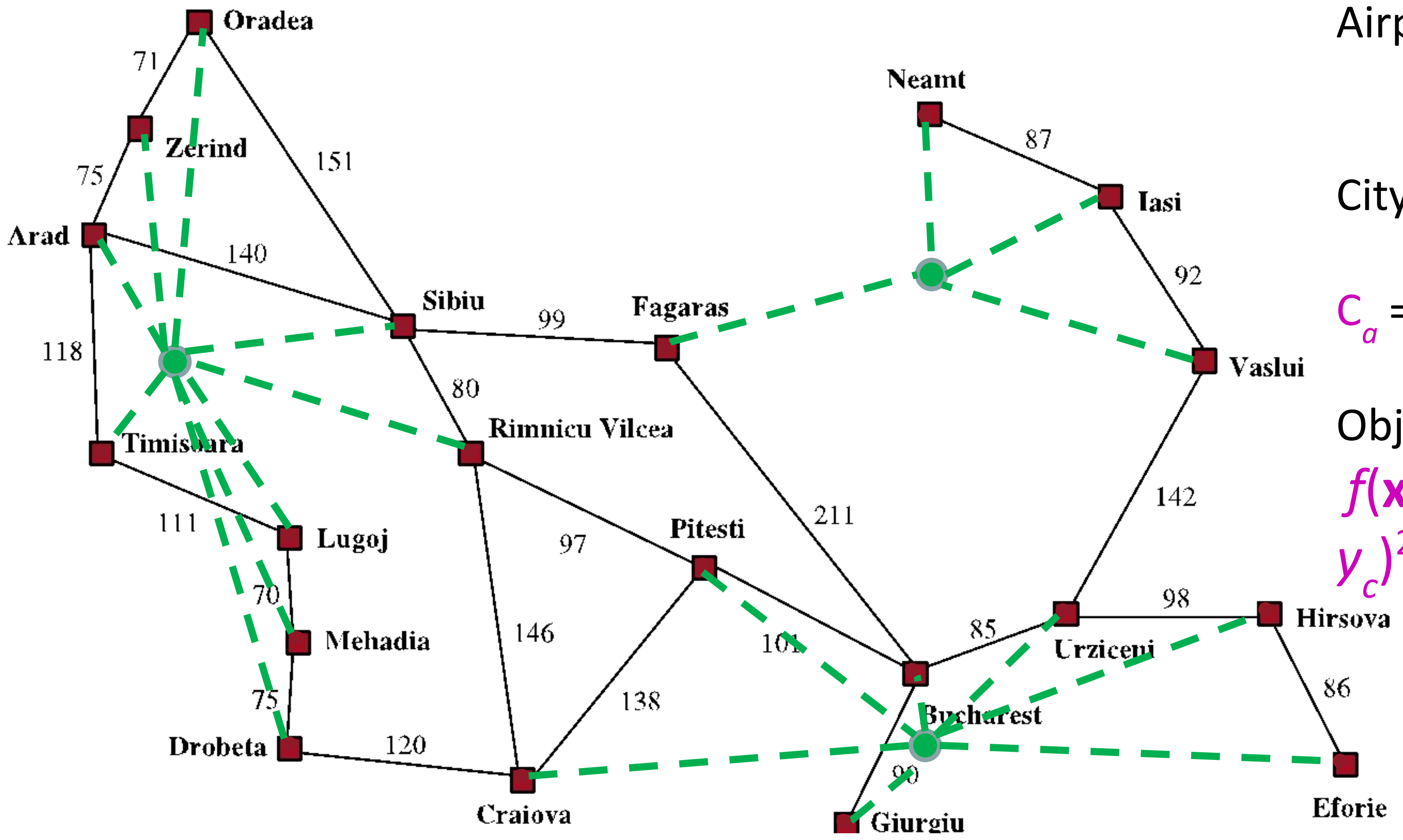


- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?

Local search in continuous spaces

Example: Siting airports in Romania

Place 3 airports to minimize the sum of squared distances from each city to its nearest airport



Airport locations

$\mathbf{x} = (x_1,y_1), (x_2,y_2), (x_3,y_3)$

City locations (x_c,y_c)

C_a = cities closest to airport a

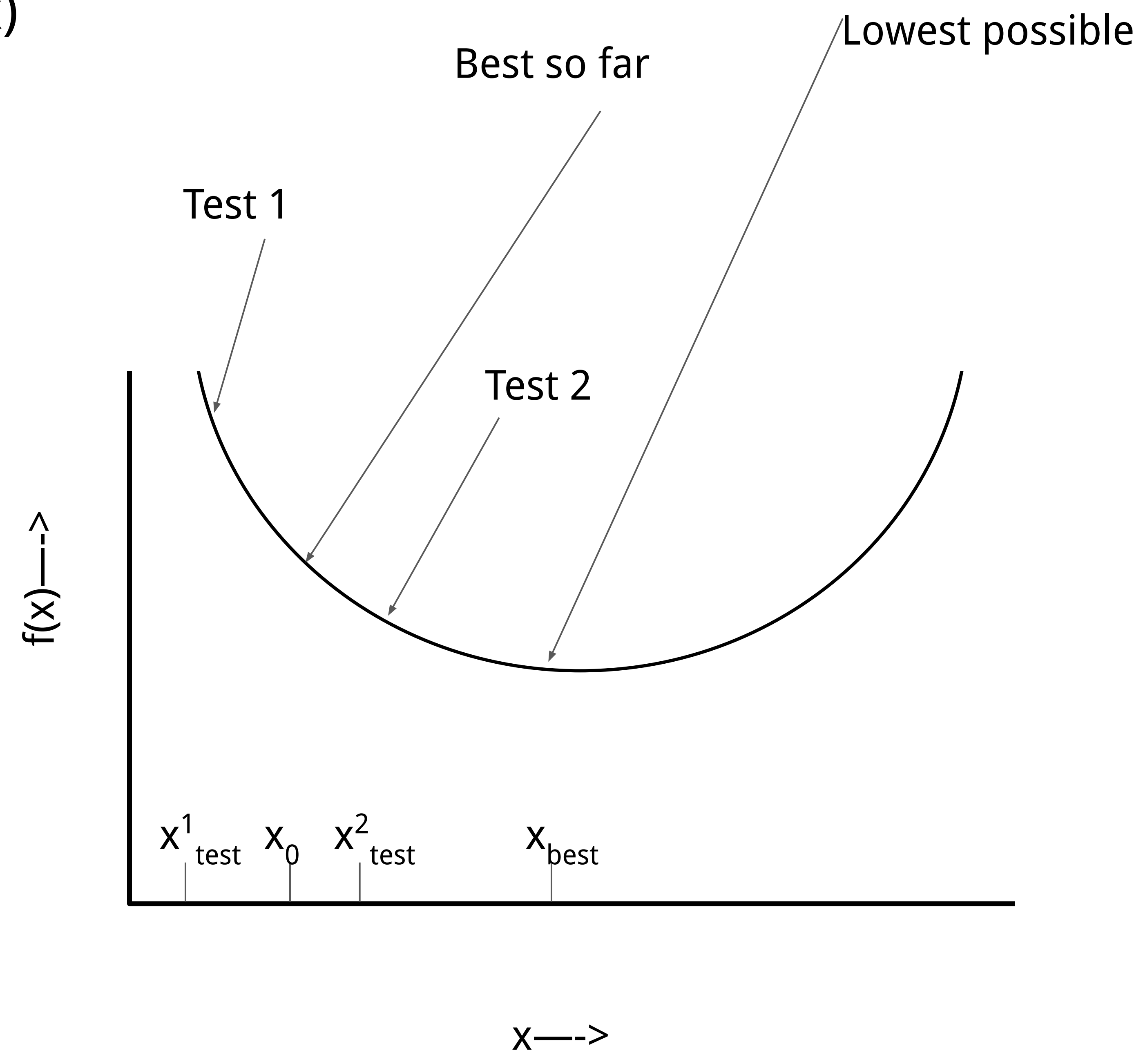
Objective: minimize

$f(\mathbf{x}) = \Sigma_a \, \Sigma_{c \in C_a} \, (x_a - x_c)^2 + (y_a - y_c)^2$

Handling a continuous state/action space

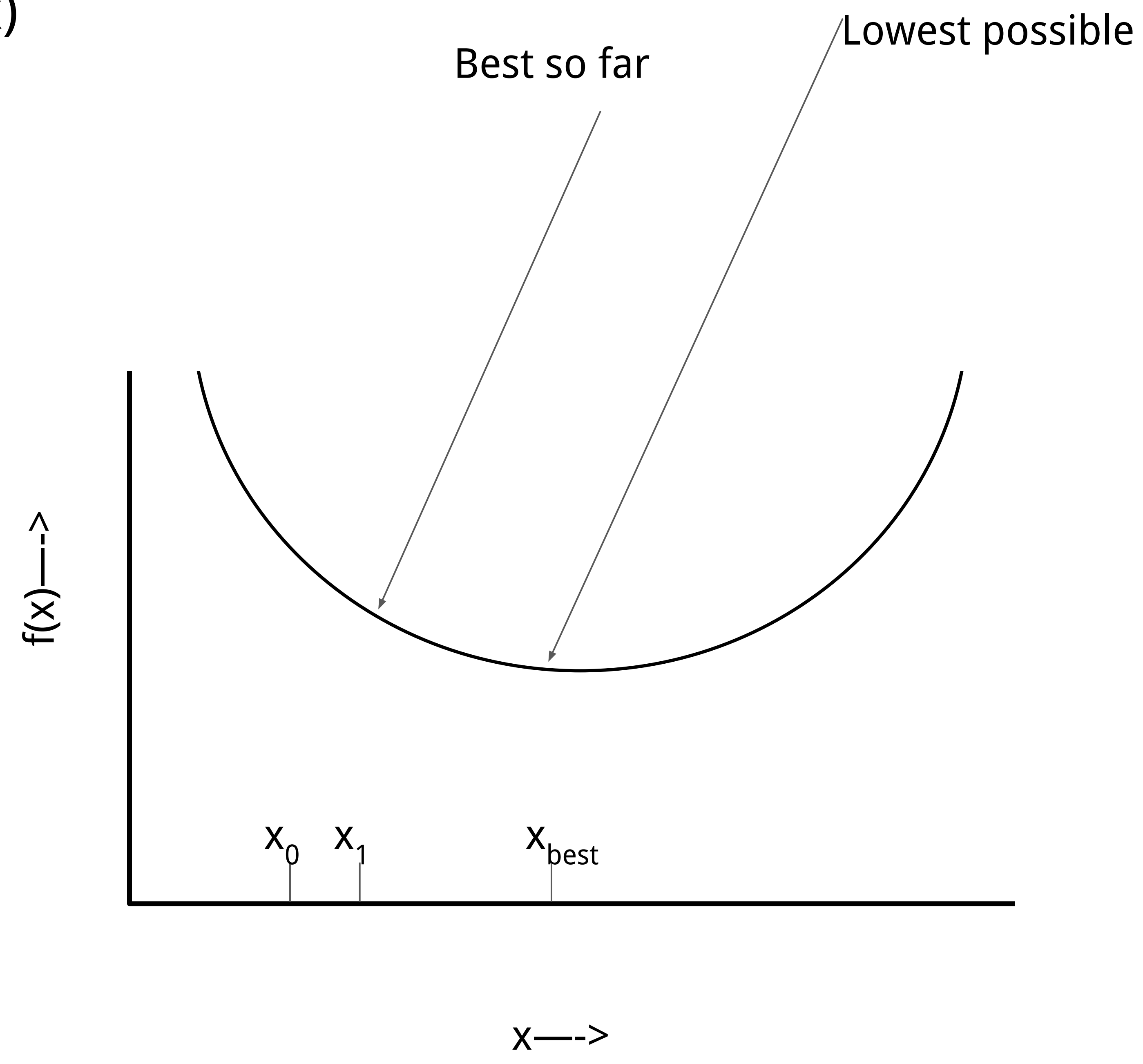
1. Discretize it!
   - Define a grid with increment δ , use any of the discrete algorithms
2. Choose random perturbations to the state
   a. First-choice hill-climbing: keep trying until something improves the state
   b. Simulated annealing
4. Compute derivatives of $f(\mathbf{x})$ analytically

$$x = \underset{x \in \mathbb{R}^D}{\operatorname{argmin}} f(x)$$



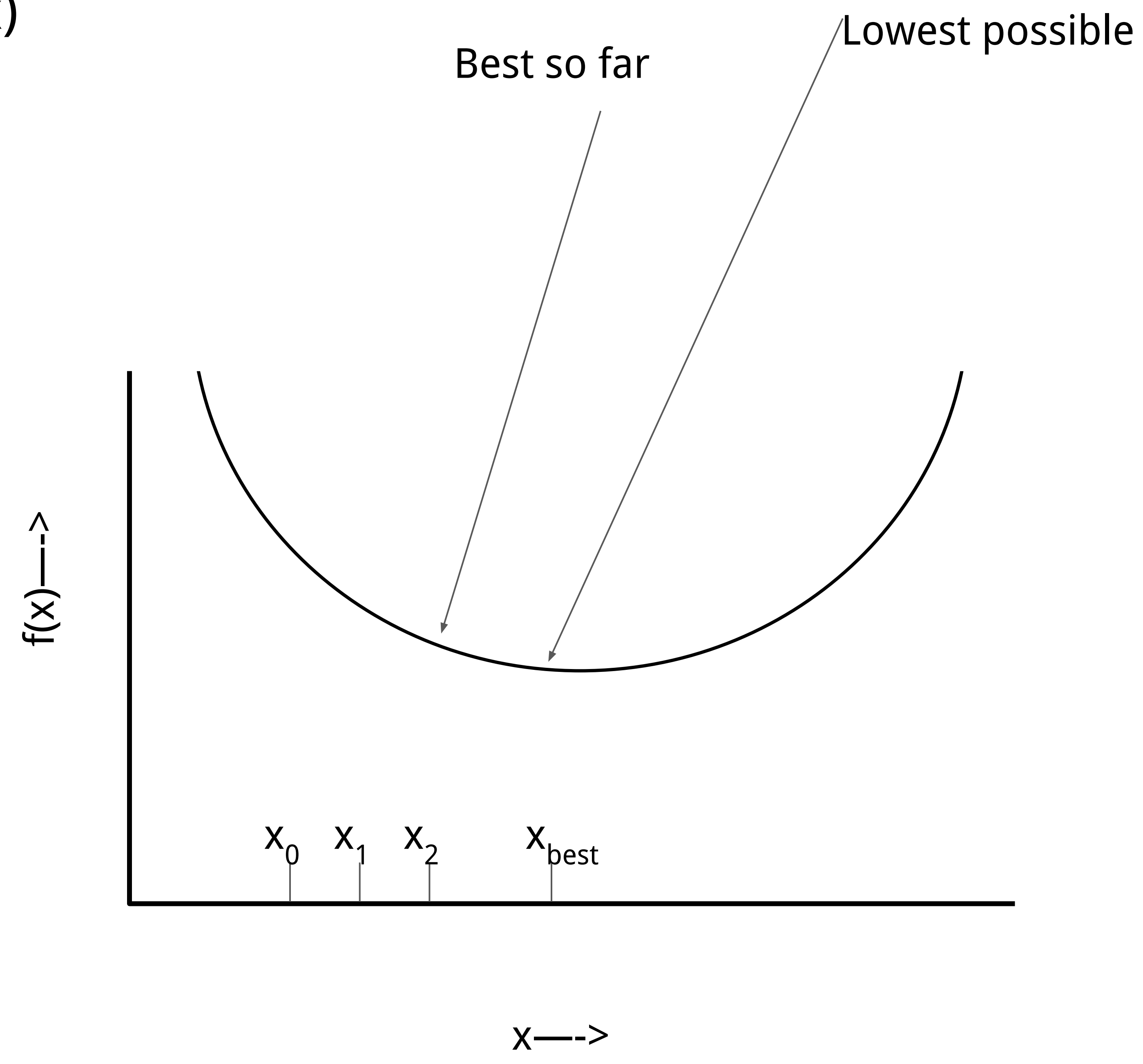
Test 2 gives lower f(x)

$$x = \underset{x \in \mathbb{R}^D}{\operatorname{argmin}} f(x)$$



Test 2 gives lower

$$x = \underset{x \in \mathbb{R}^D}{\operatorname{argmin}} f(x)$$



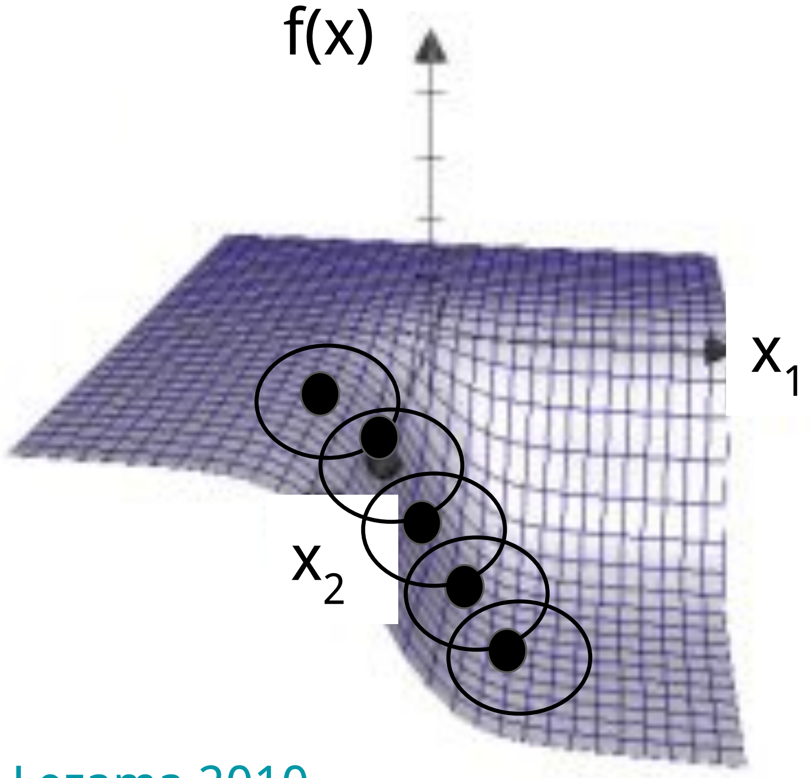


Figure from [Chaudhuri & Solar-Lezama 2010]

Continuous Local Search

x = a random vector in $\mathbb{R}^D$

Loop:

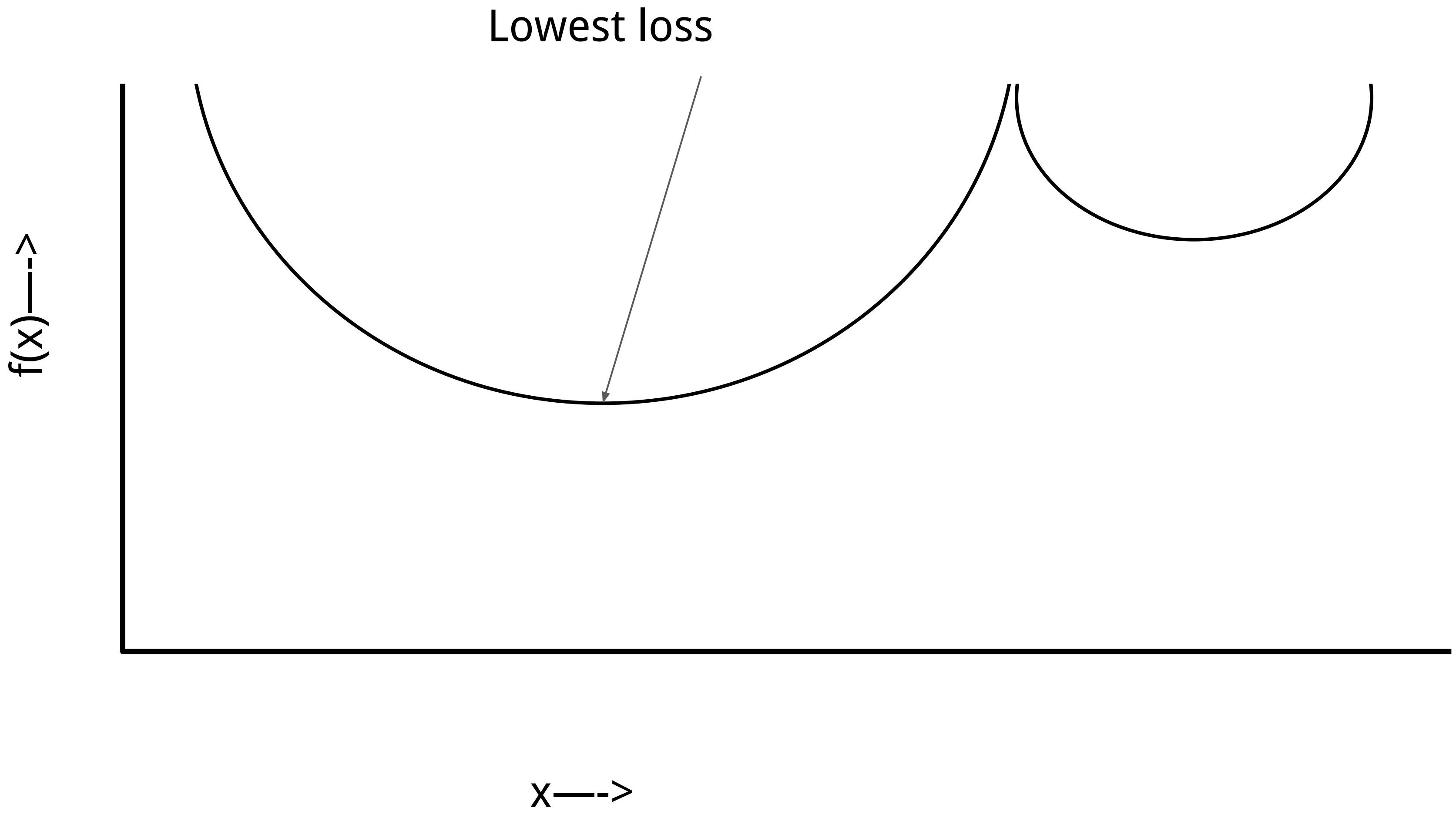
Make small perturbations to x

Call these perturbations x^1_{test} x^2_{test} x^2_{test} …

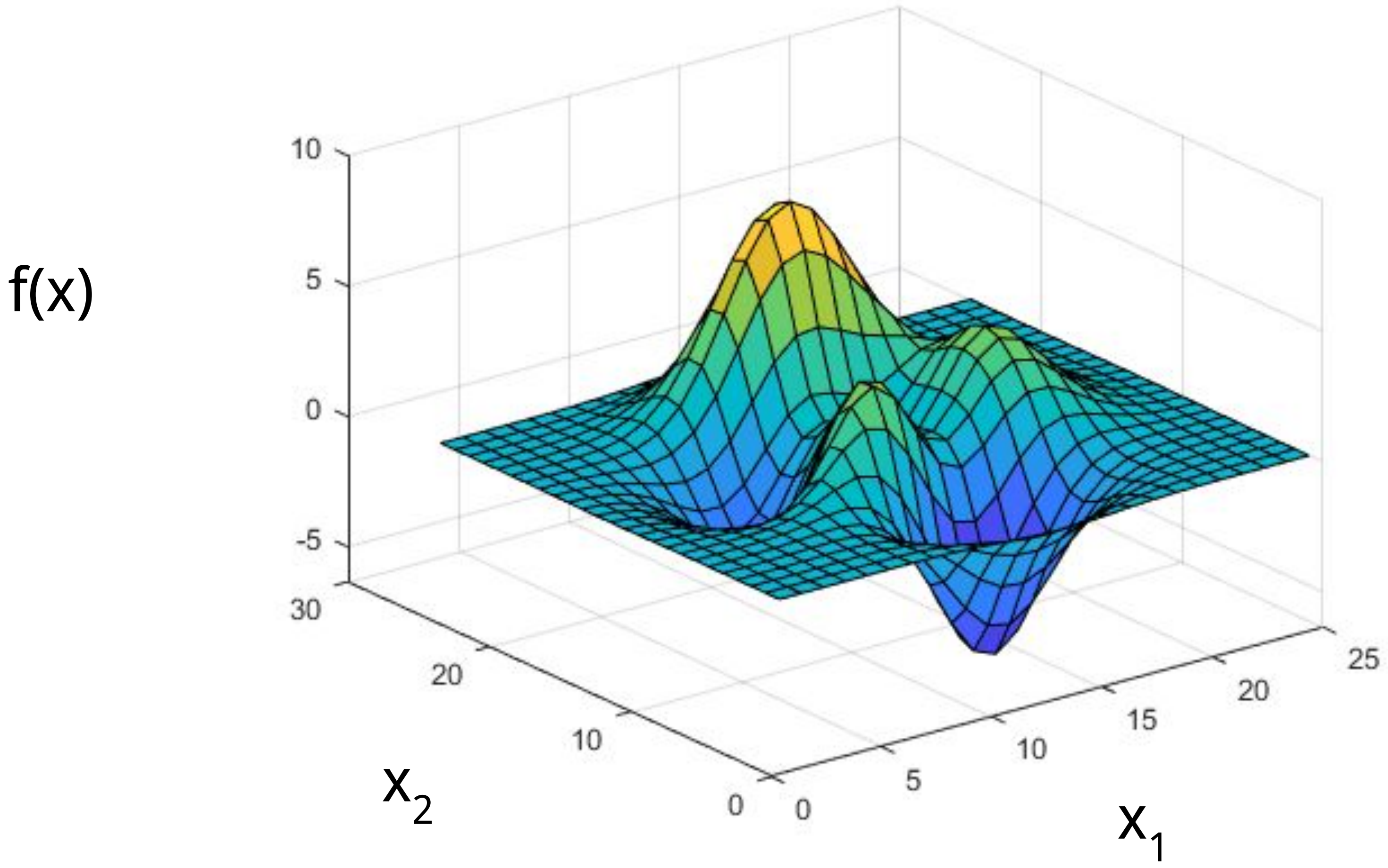
Compute $f(x^1_{test}), f(x^2_{test})$, …

x = the x^i_{test} with lowest $f(x^i_{test})$

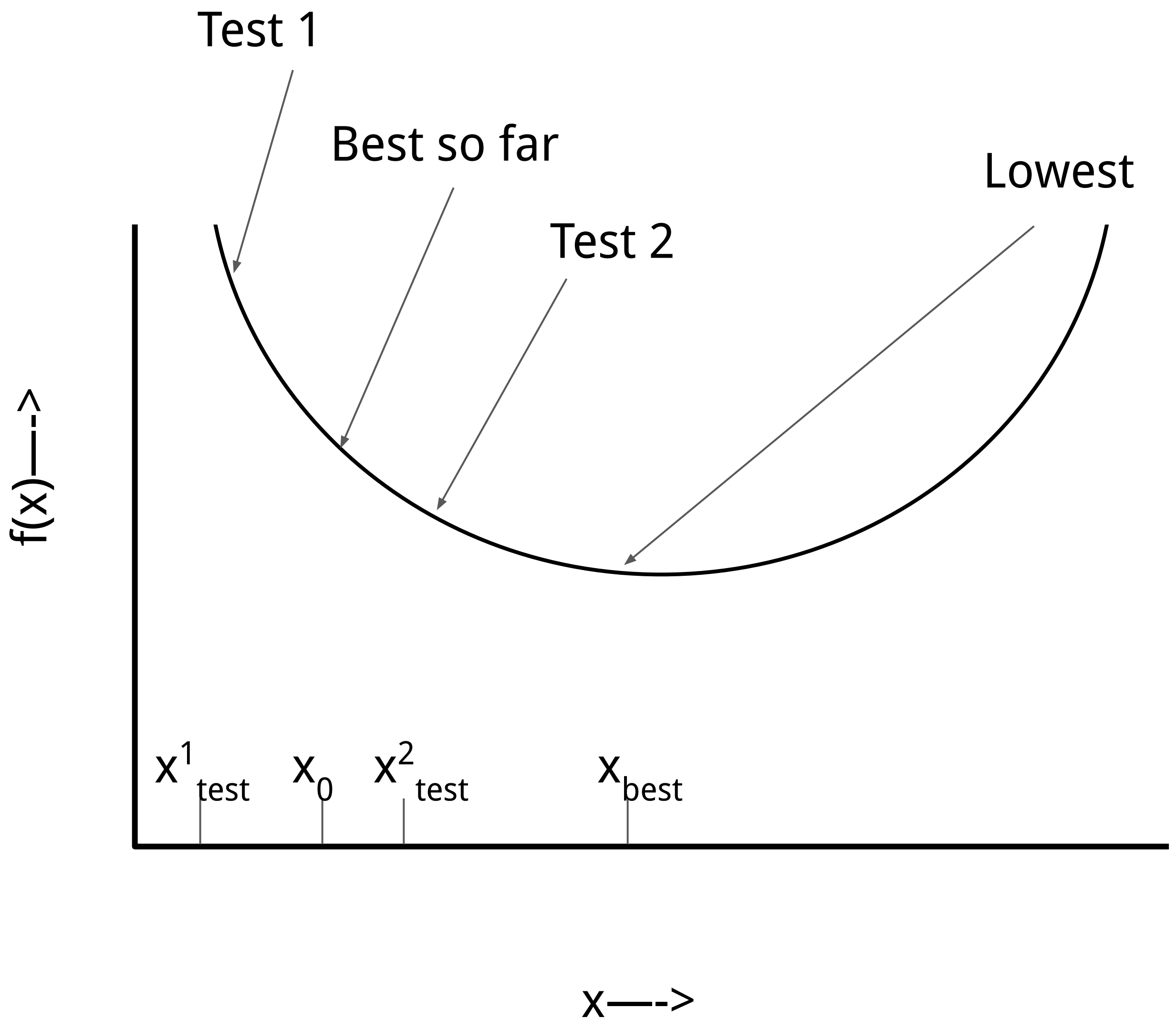
Does this local search always work?



Does This Local Search Always Work?



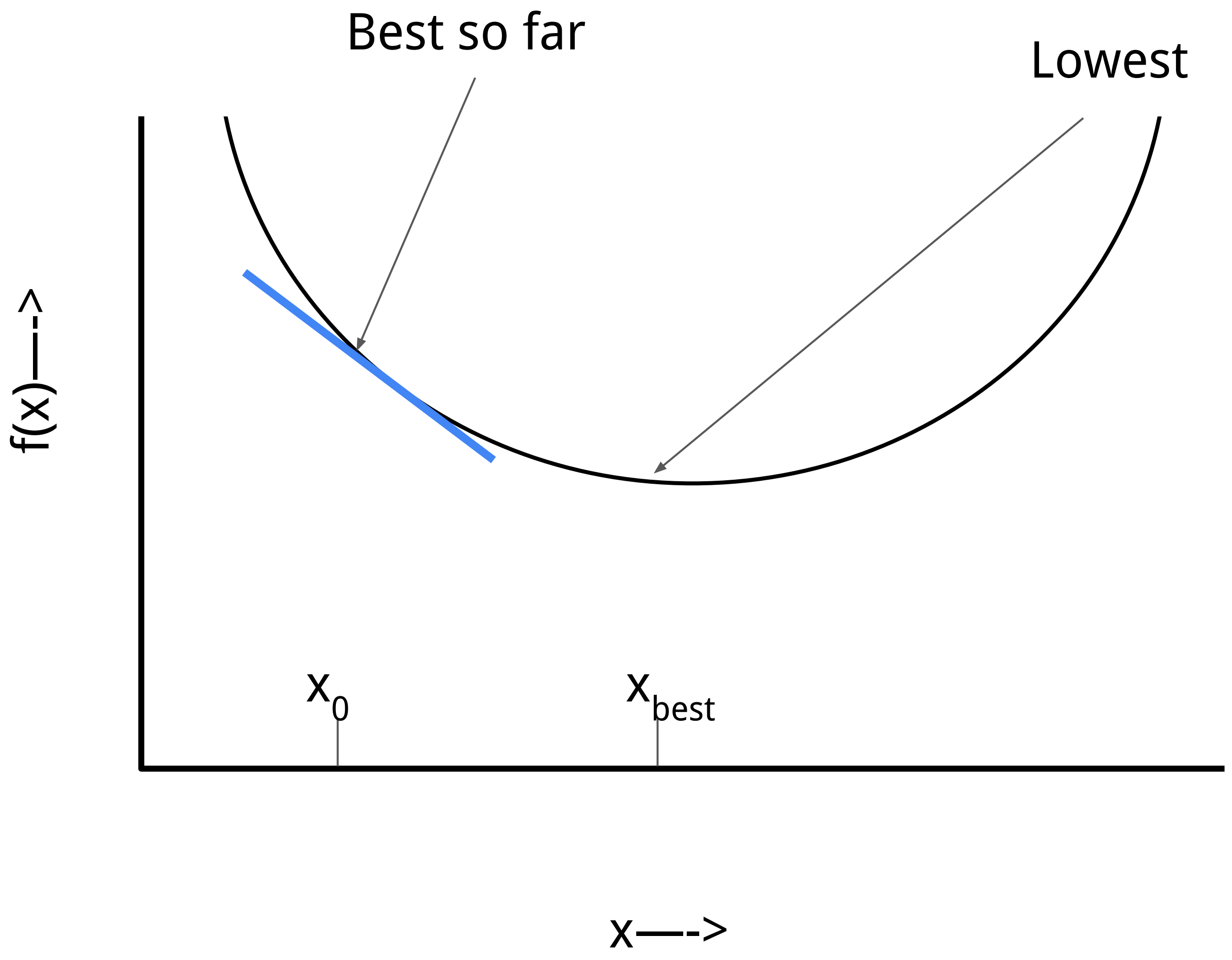
From here to gradient descent



We decided to move right (increase x)

Did we need to make the test points to figure out that we should move right?

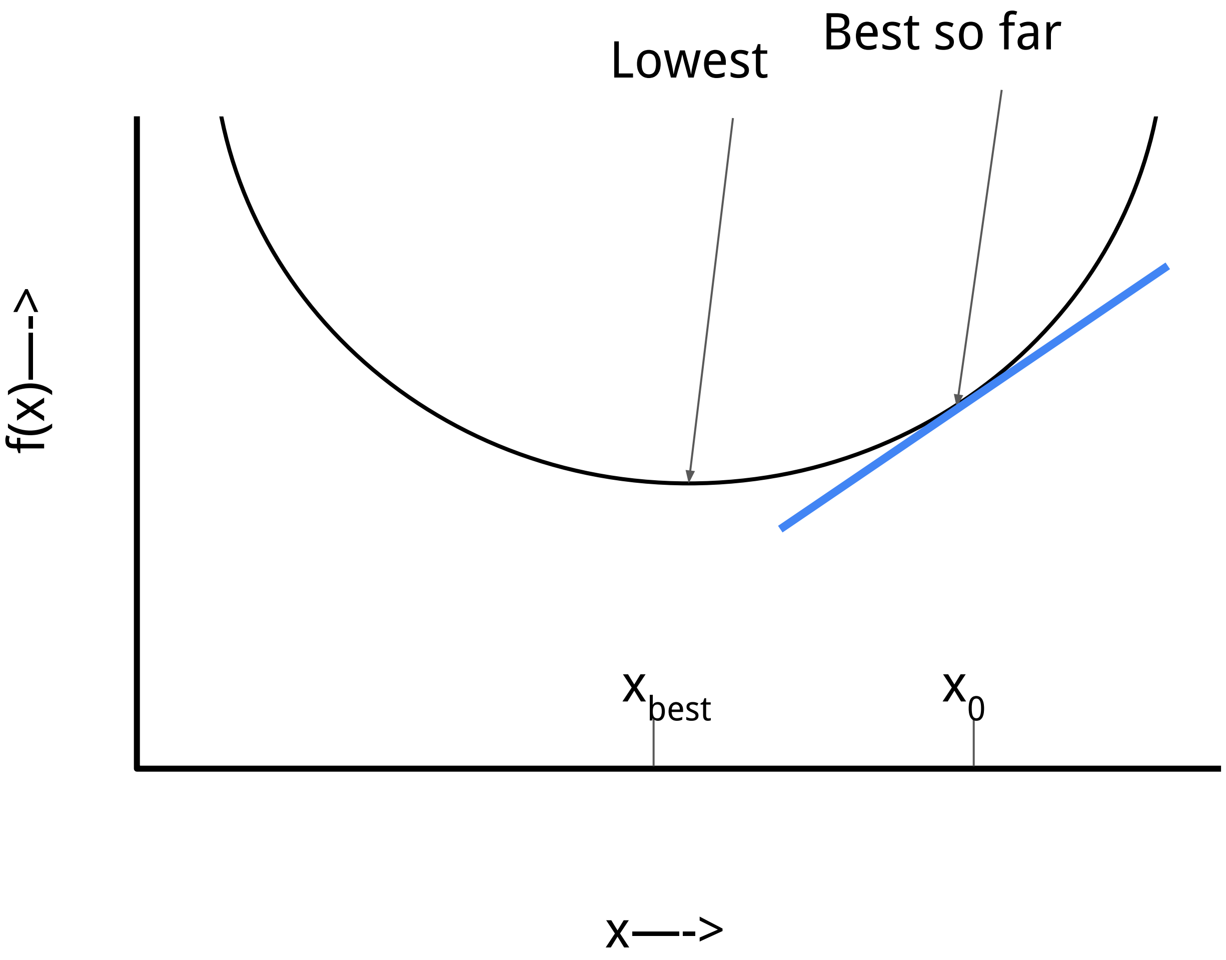
From here to gradient descent



$$\frac{d}{dx} f(x)\Big|_{x=x_0} < 0$$

Negative derivative, increase x to decrease f(x)

From here to gradient descent



$$\frac{d}{dx} f(x)\Big|_{x=x_0} > 0$$

Positive derivative, decrease x to decrease f(x)

Gradient Descent, 1 Dimension

$$\frac{d}{dx}f(x) > 0$$

DECREASE x

$$\boxed{x \mathrel{+}= -\lambda \frac{d}{dx}f(x)}$$

$$\frac{d}{dx}f(x) < 0$$

INCREASE x

Gradient Descent, 1-dimension

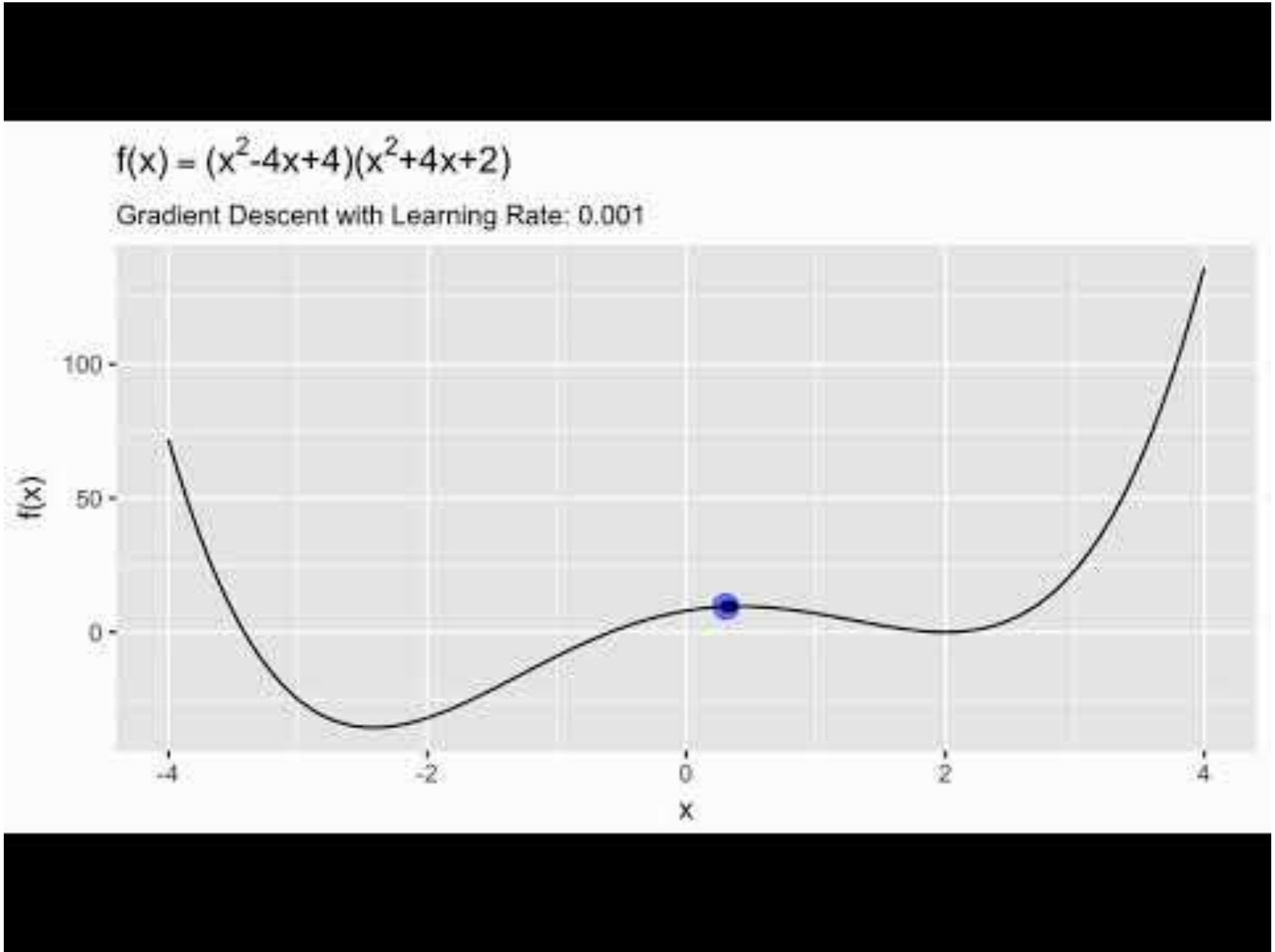
λ = a small positive number (like 0.001), called the “learning rate”

x = a random number

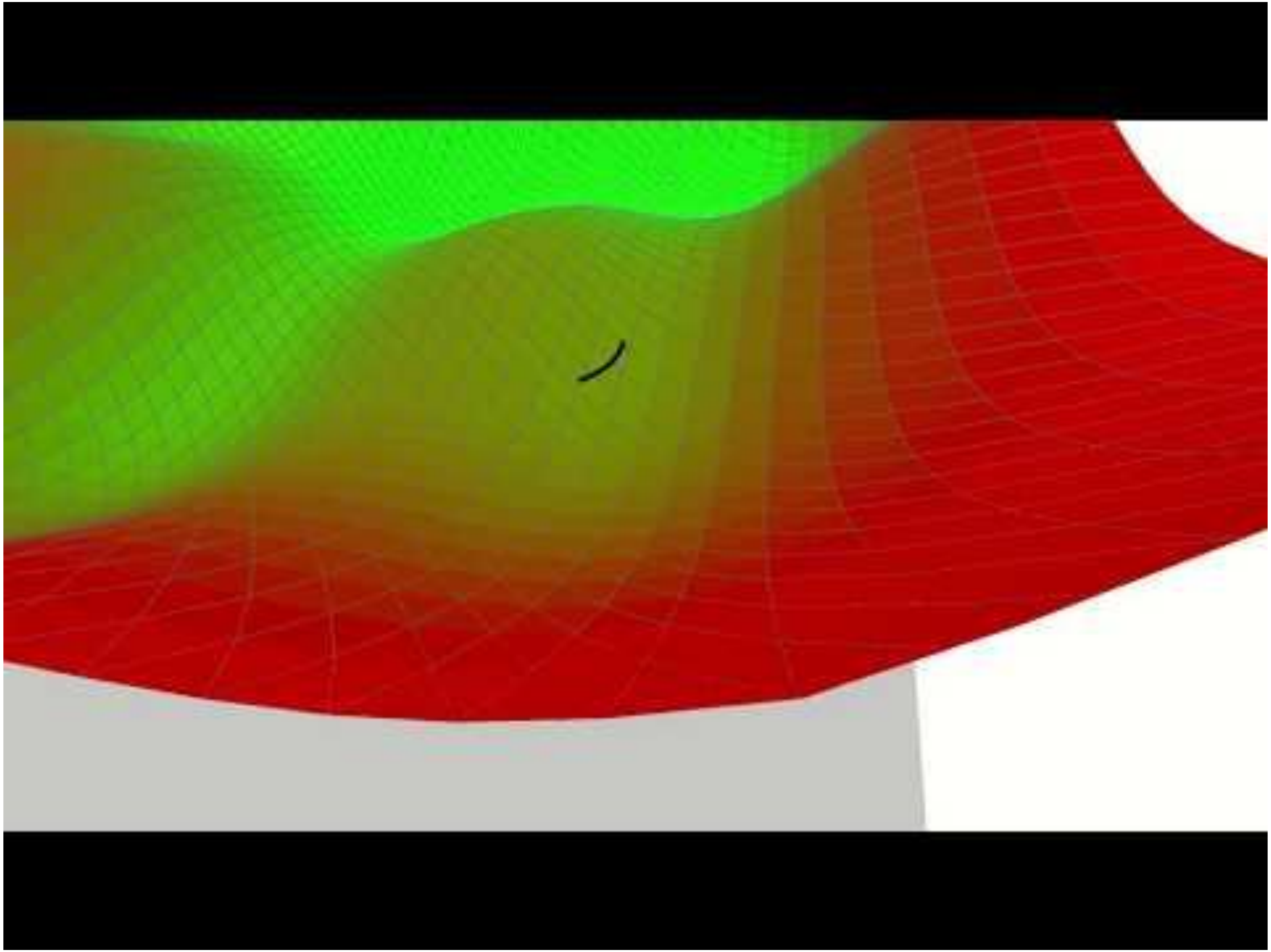
Loop:

$$x \leftarrow x - \lambda \times \frac{d}{dx} f(x)$$

Gradient Descent ~ Falling Downhill



Higher dimensions?



Gradient Descent, 2 Dimensions

x in $\mathbb{R}^2$

$x = (x_1, x_2)$

$f(x) = f(x_1, x_2)$

$$\frac{d}{dx_1} f(x_1, x_2) > 0$$

DECREASE x_1

$$\frac{d}{dx_1} f(x_1, x_2) < 0$$

INCREASE x_1

$$\frac{d}{dx_2} f(x_1, x_2) > 0$$

DECREASE x_2

$$\frac{d}{dx_2} f(x_1, x_2) < 0$$

INCREASE x_2

Gradient Descent

λ = a small positive number (like 0.001), called the "learning rate"

x = a random vector in $\mathbb{R}^D$

Loop:

Compute:

$$g_i = \frac{d}{dx_i} f(x), \text{ for } i \text{ from 1 to } D$$

For each dimension *i* ranging from 1 to D:

$$x_i \leftarrow x_i - \lambda \times g_i$$

Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
  - Hill-climbing, continuous optimization
  - Simulated annealing (and other stochastic methods)
  - Local beam search: multiple interaction searches
  - Genetic algorithms: break and recombine states

Many machine learning algorithms are local searches

Example: Discovering Natural Laws

Done at Cornell!

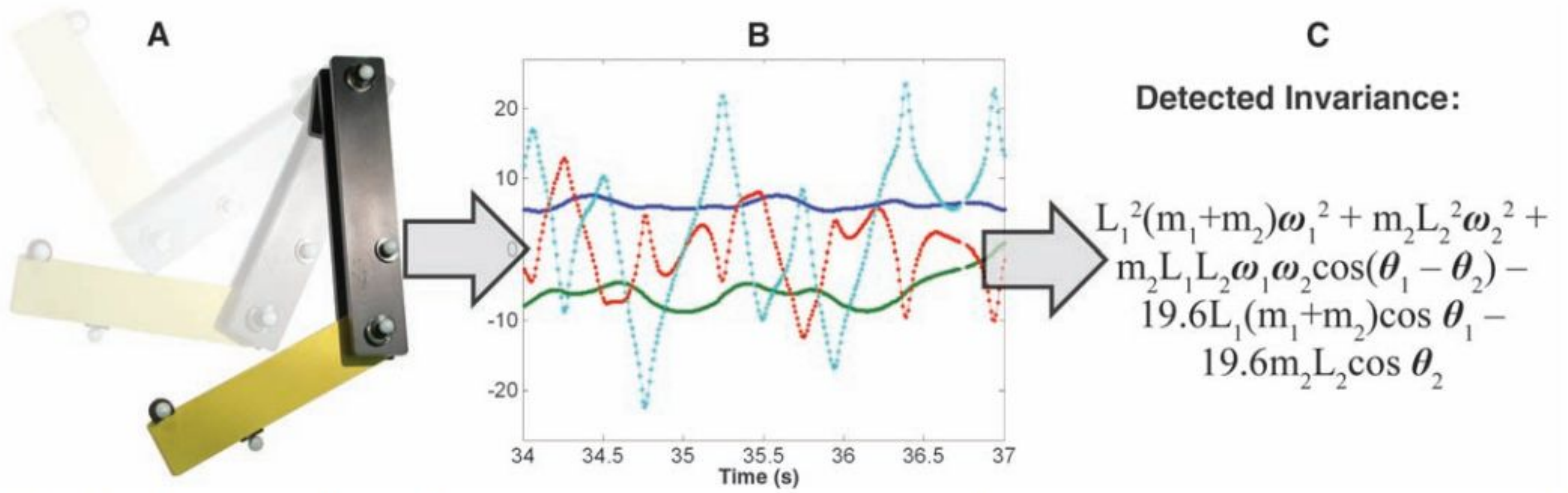


Fig. 1. Mining physical systems. We captured the angles and angular velocities of a chaotic double-pendulum (**A**) over time using motion tracking (**B**), then we automatically searched for equations that describe a single natural law relating these variables. Without any prior knowledge about physics or geometry, the algorithm found the conservation law (**C**), which turns out to be the double pendulum's Hamiltonian. Actual pendulum, data, and results are shown.

Example: Discovering Natural Laws
(Schmidt & Lipson 2009)

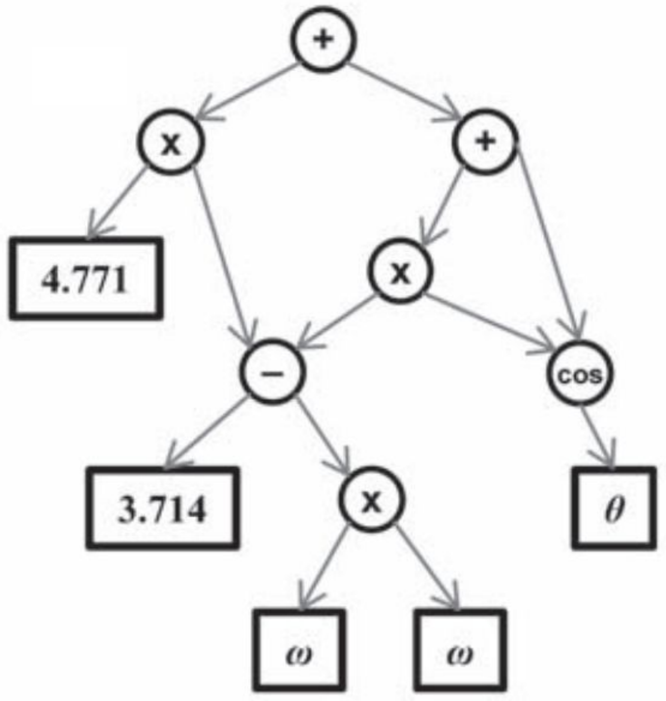
Done at Cornell!

Cornell University

Genotype:
Mutation and crossover operate over this

```
(0)  <- load [3.714]
(1)  <- load [ω]
(2)  <- mul   (1), (1)
(3)  <- sub   (0), (2)
(4)  <- load [θ]
(5)  <- cos   (4)
(6)  <- mul   (3), (5)
(7)  <- load [4.771]
(8)  <- mul   (7), (3)
(9)  <- add   (8), (5)
(10) <- add   (9), (6)
```

Phenotype:
Fitness operates over this



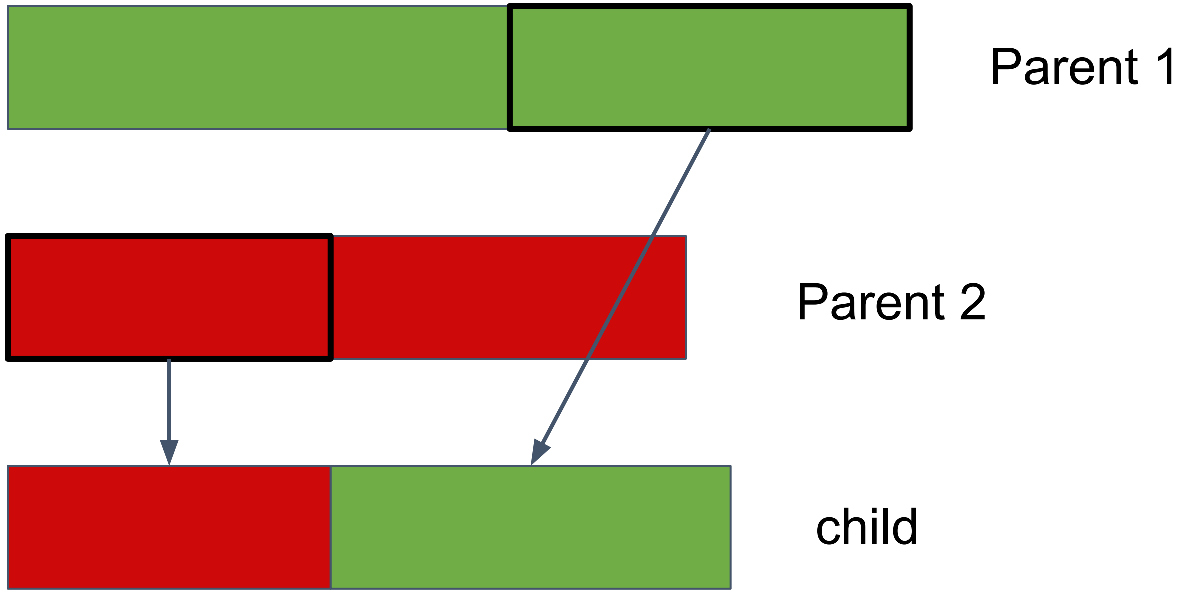
Example: Discovering Natural Laws
(Schmidt & Lipson 2009)

Done at Cornell!

Cornell University

Crossover randomly exchanges prefix/suffix of parents

```
(0)  <- load [3.714]
(1)  <- load [ω]
(2)  <- mul  (1), (1)
(3)  <- sub  (0), (2)
(4)  <- load [θ]
(5)  <- cos  (4)
(6)  <- mul  (3), (5)
(7)  <- load [4.771]
(8)  <- mul  (7), (3)
(9)  <- add  (8), (5)
(10) <- add  (9), (6)
```



Fitness for physical laws

Very subtle:

Want to discover equations that capture conserved invariants,

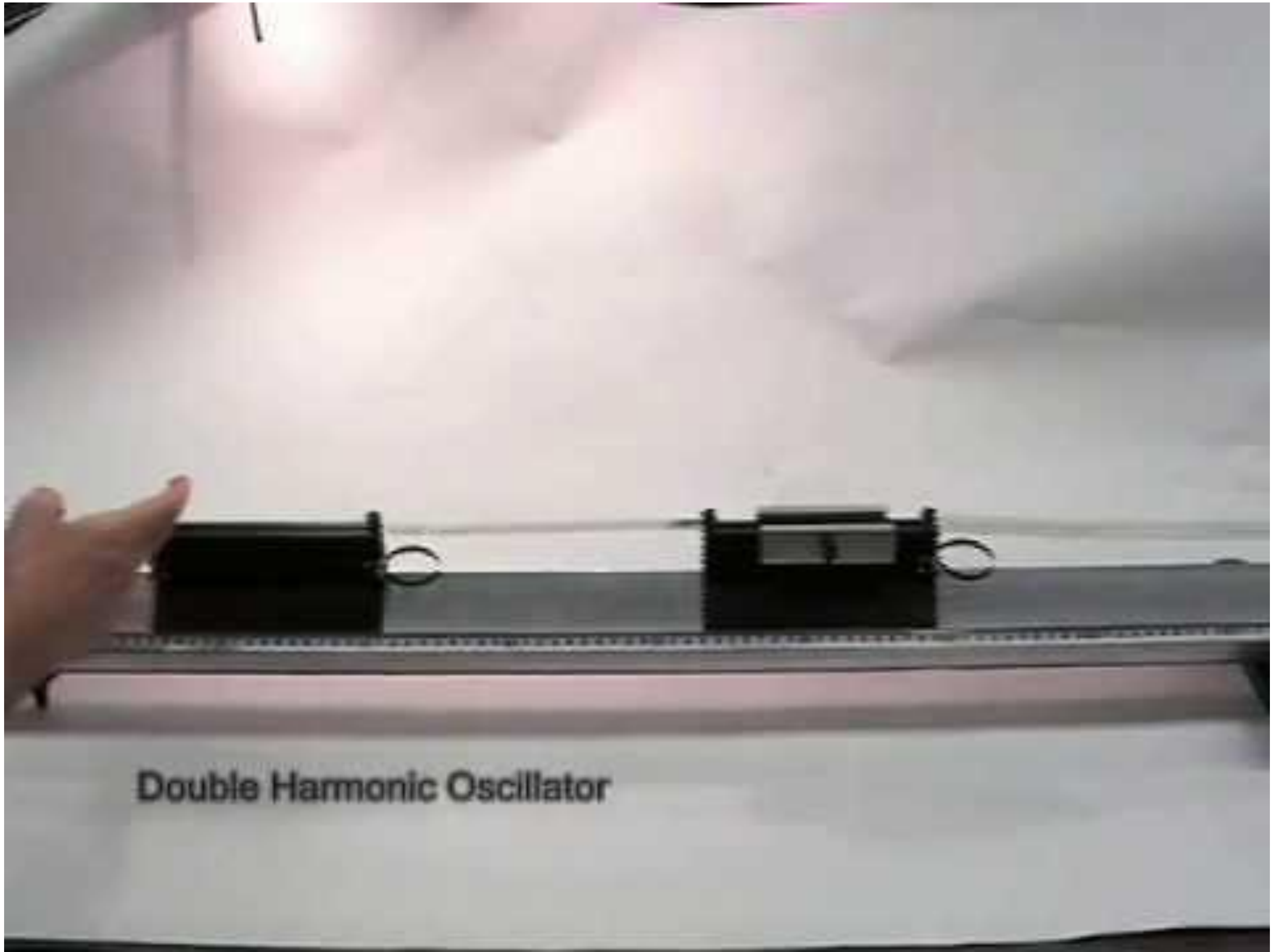
like energy or momentum

Main physics insight goes into design of “fitness” for a candidate invariant that is being evolved

Example: Discovering Natural Laws [(Schmidt & Lipson 2009)]

Done at Cornell!

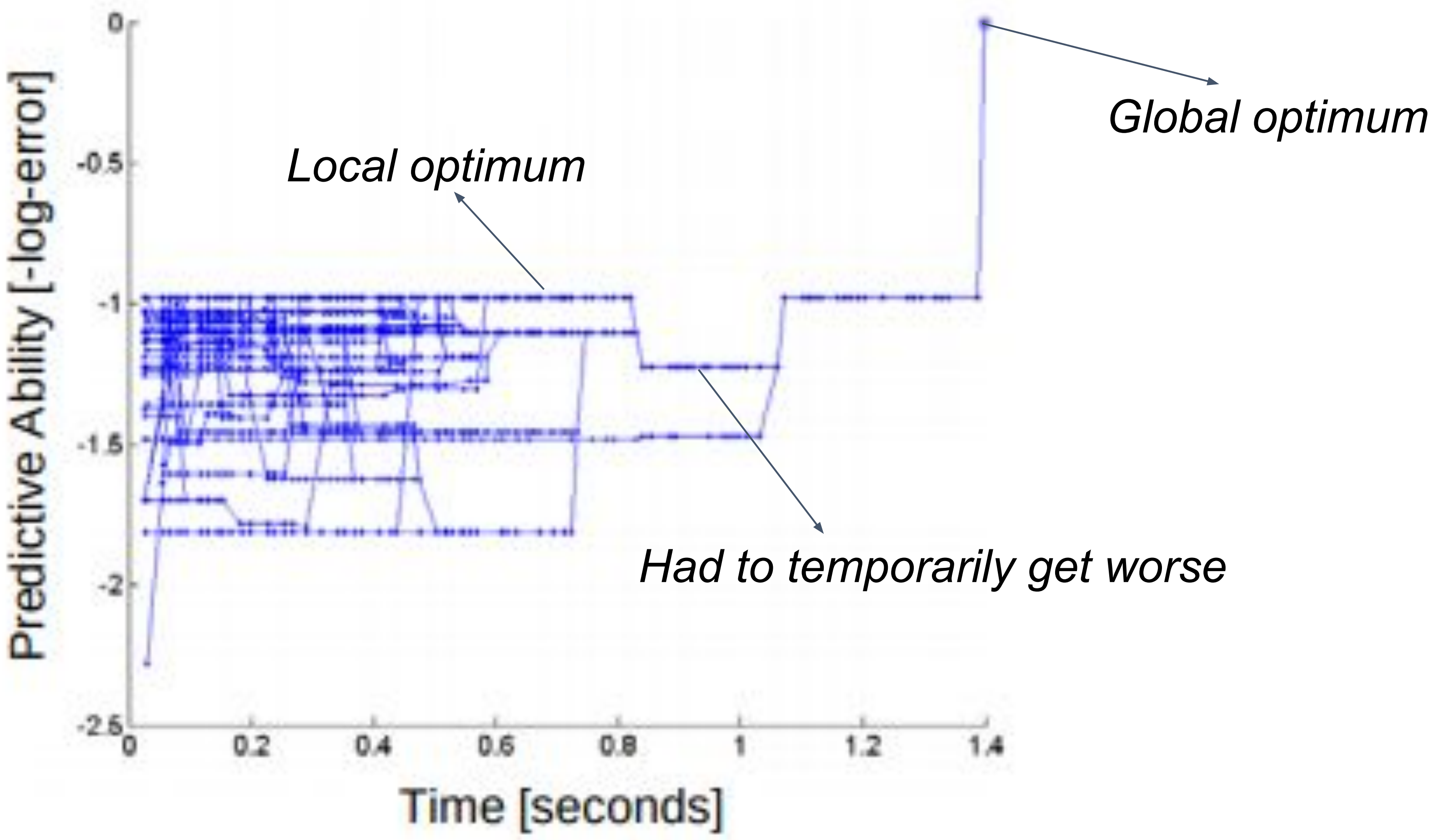
Cornell University



Example: Discovering Natural Laws [(Schmidt & Lipson 2009)]

Done at Cornell!

Cornell University



Example: Discovering Natural Laws (Schmidt & Lipson 2009)

Done at Cornell!

Cornell University

“Island” evolution model:
Separate evolving populations that occasionally exchange solutions
Promotes diversity, more parallel