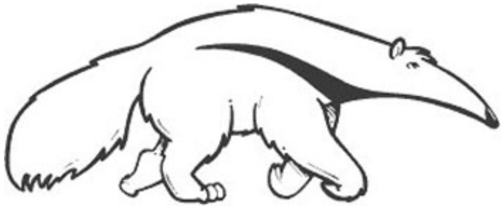
Local search algorithms

CS171, Fall 2016

Introduction to Artificial Intelligence

Prof. Alexander Ihler



Reading: R&N 4.1-4.2

Local search algorithms

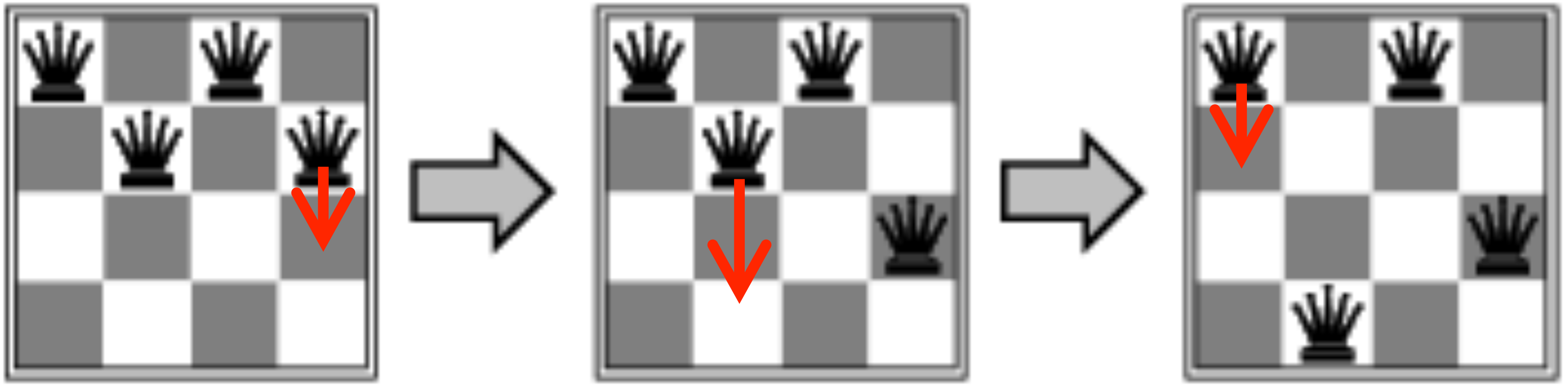
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: widely used for *very big* problems
  - Returns good but *not optimal* solutions

- State space = set of "complete" configurations
- Find configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, airline flight schedules

- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient
    - keeps only one or a few states
    - You control how much memory you use

Example: *n*-queens

- Goal: Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- Neighbor: move one queen to another row
- Search: go from one neighbor to the next…



Algorithm design considerations

- How do you represent your problem?
- What is a “complete state”?
- What is your objective function?
  - How do you measure cost or value of a state?
- What is a “neighbor” of a state?
  - Or, what is a “step” from one state to another?
  - How can you compute a neighbor or a step?
- Are there any constraints you can exploit?

Random restart wrapper

- We’ll use stochastic local search methods
  - Return different solution for each trial & initial state
- Almost every trial hits difficulties (see sequel)
  - Most trials will not yield a good result (sad!)
- Using many random restarts improves your chances
  - Many “shots at goal” may finally get a good one
- Restart a random initial state, *many times*
  - Report the best result found across *many* trials

Random restart wrapper

```
best_found ← RandomState()    // initialize to something

while not (tired of doing it):    // now do repeated local search
    result ← LocalSearch( RandomState() )
    if (Cost(result) < Cost(best_found)):
        best_found = result    // keep best result found so far

return best_found
```

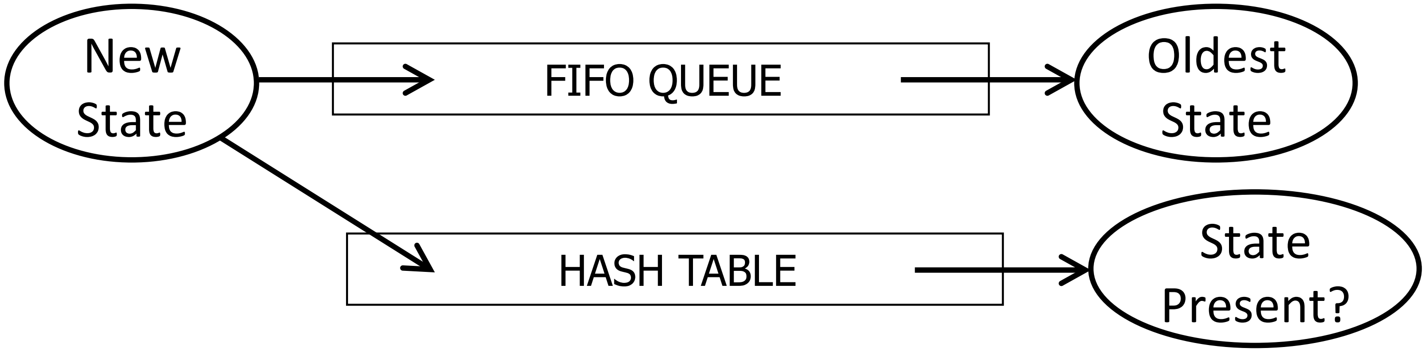
Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.

Tabu search wrapper

- Add recently visited states to a tabu-list
  - Temporarily excluded from being visited again
  - Forces solver away from explored regions
  - Avoid getting stuck in local minima (in principle)

- Implemented as a hash table + FIFO queue
  - Unit time cost per step; constant memory cost
  - You control how much memory is used

Tabu search wrapper



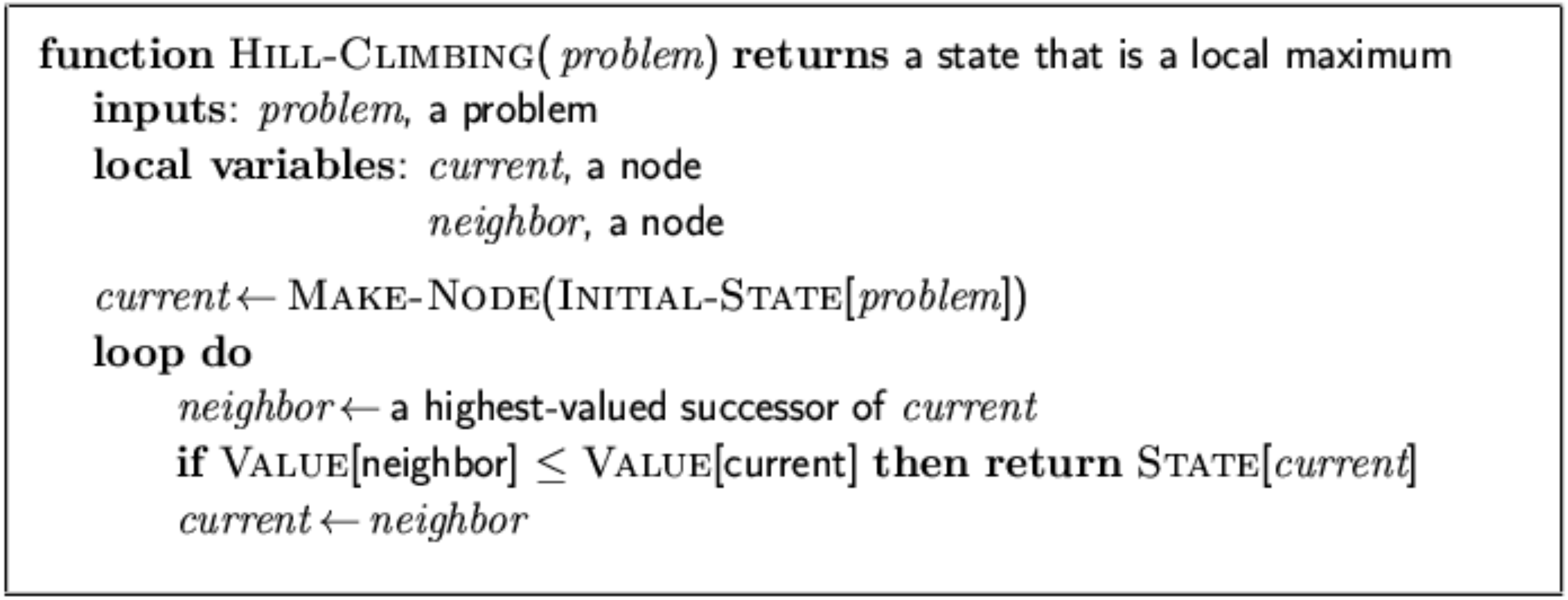
```
UNTIL ( you are tired of doing it ) DO {
    set Neighbor to makeNeighbor( CurrentState );
    IF ( Neighbor is in HASH ) THEN ( discard Neighbor );
        ELSE { push Neighbor onto FIFO, pop OldestState;
            remove OldestState from HASH, insert Neighbor;
            set CurrentState to Neighbor;
            run yourFavoriteLocalSearch on CurrentState; } }
```

Local search algorithms

- Hill-climbing search
  - Gradient descent in continuous state spaces
  - Can use e.g. Newton’s method to find roots
- Simulated annealing search
- Local beam search
- Genetic algorithms

Hill-climbing search

"...like trying to find the top of Mount Everest in a thick fog while suffering from amnesia"



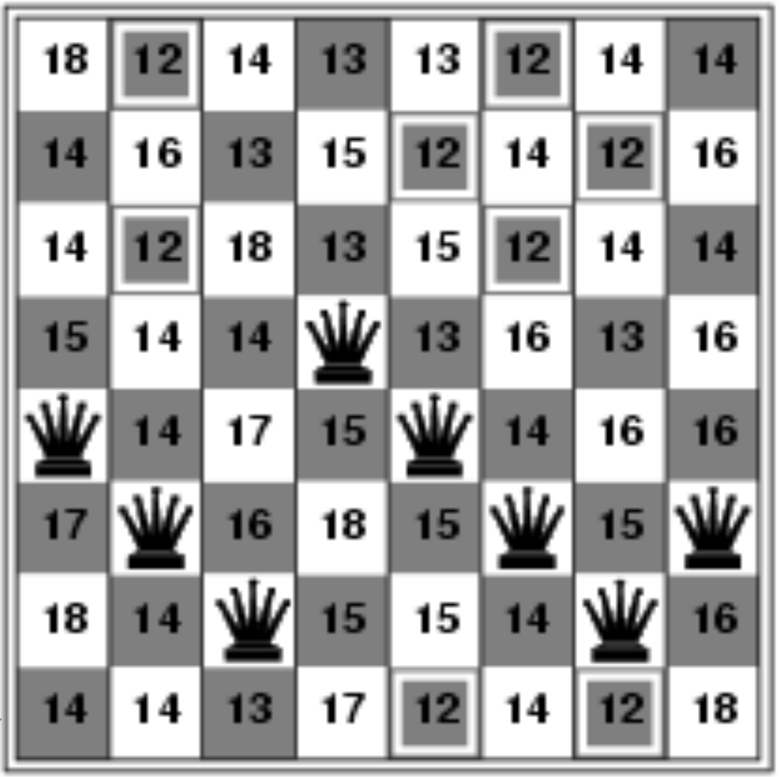
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] $\leq$ VALUE[current] then return STATE[current]
        current ← neighbor
```

Ex: Hill-climbing, 8-queens

h = # of pairs of queens that are attacking each other, either directly or indirectly

h=17 for this state



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

Each number indicates *h* if we move a queen in its column to that square

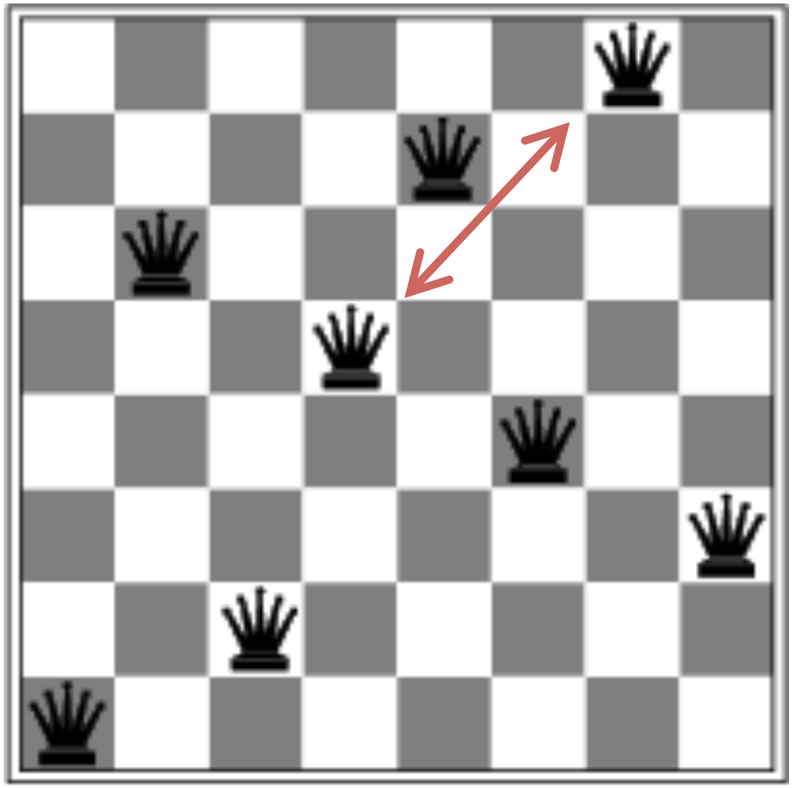
12 (boxed) = best *h* among all neighbors; select one randomly

Ex: Hill-climbing, 8-queens

- A local minimum with h=1

- All one-step neighbors have higher h values

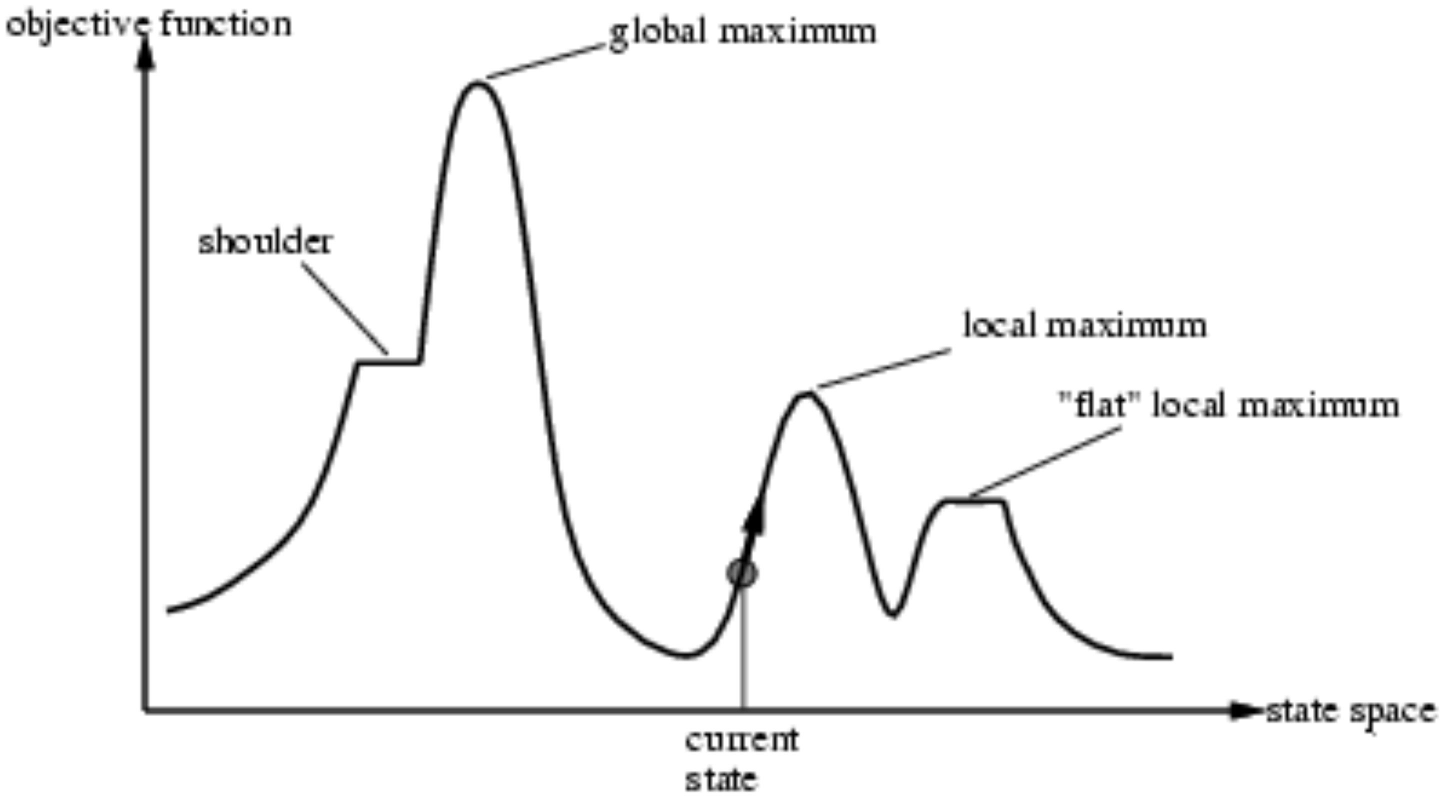
- What can you do to get out of this local minimum?



Hill-climbing difficulties

Note: these difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional

- Problem: depending on initial state, can get stuck in local maxima

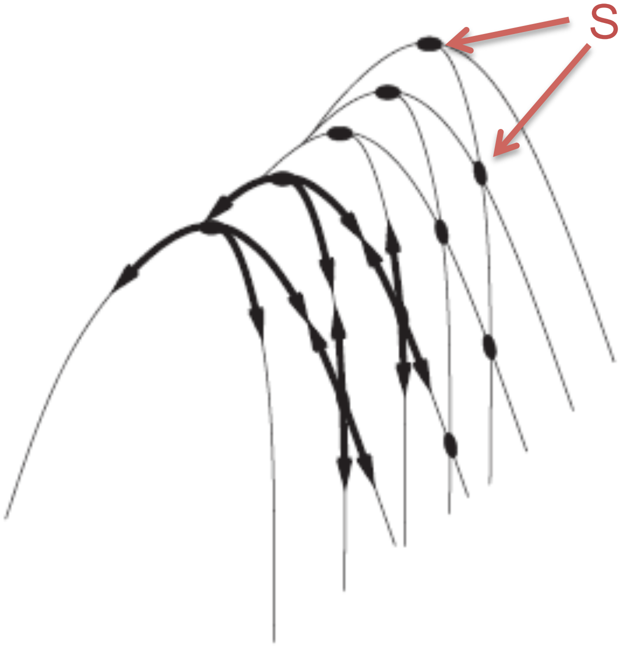


Hill-climbing difficulties

Note: these difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional

- Ridge problem: every neighbor appears to be downhill
  - But, search space has an uphill (just not in neighbors)

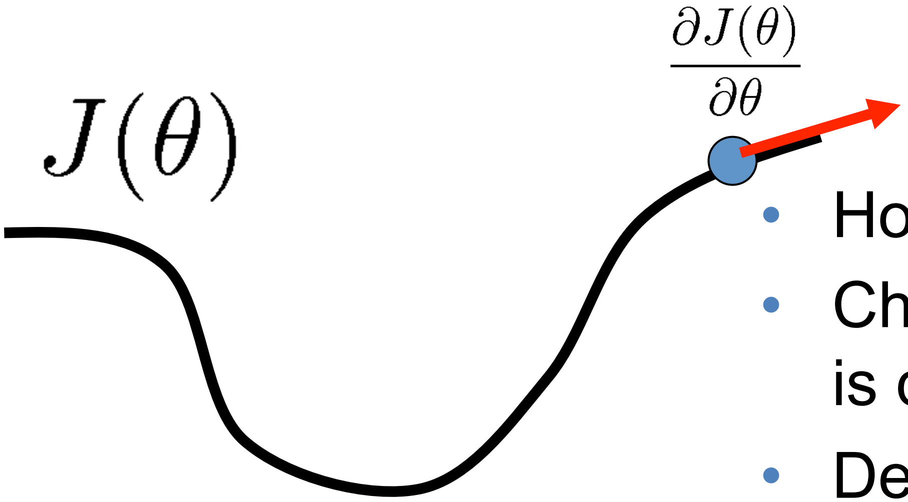
Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.



States / steps (discrete)

Gradient descent

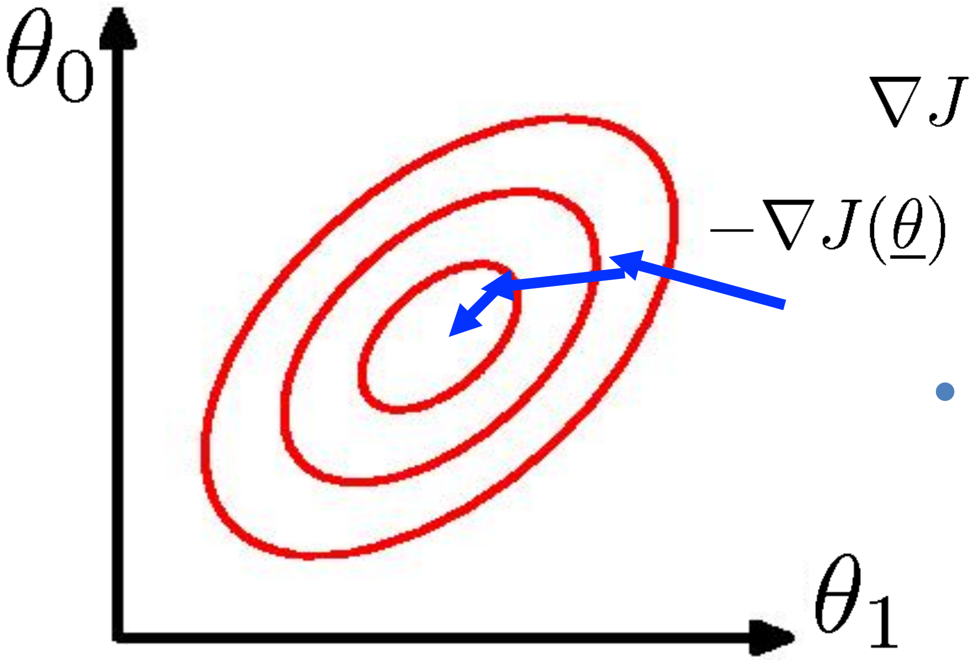
- Hill-climbing in continuous state spaces
- Denote “state” as θ; cost as J(θ)



- How to change θ to improve J(θ)?
- Choose a direction in which J(θ) is decreasing
- Derivative $\frac{\partial J(\theta)}{\partial \theta}$
  - Positive => increasing
  - Negative => decreasing

Gradient descent

Hill-climbing in continuous spaces



- Gradient vector

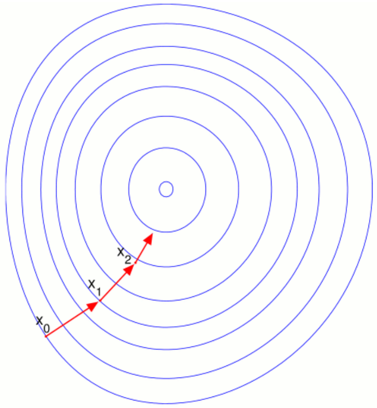
$$\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \ldots \right]$$

- Indicates direction of steepest ascent

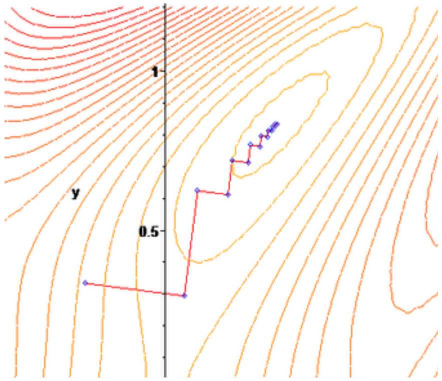
  (negative = steepest descent)

Gradient descent

Hill-climbing in continuous spaces



Gradient = the most direct direction up-hill in the objective (cost) function, so its negative minimizes the cost function.



* Assume we have some cost-function: $J(x_1, x_2, \ldots, x_n)$
and we want minimize over continuous variables $x_1,x_2,..,x_n$

1. Compute the *gradient :* $\frac{\partial}{\partial x_i} J(x_1, \ldots, x_n) \qquad \forall i$

2. Take a small step downhill in the direction of the gradient:

$$x'_i = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)$$

3. Check if $J(x'_1, \ldots, x'_n) < J(x_1, \ldots, x_n)$

(or, Armijo rule, etc.)

4. If true then accept move, if not "reject".

(decrease step size, etc.)

5. Repeat.

Gradient descent

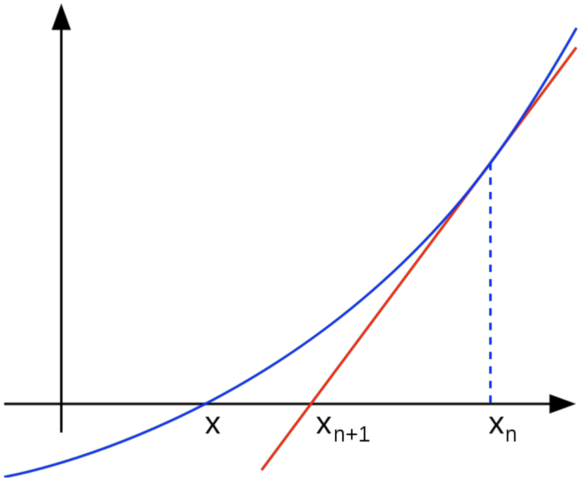
Hill-climbing in continuous spaces

- How do I determine the gradient?
  - Derive formula using multivariate calculus.
  - Ask a mathematician or a domain expert.
  - Do a literature search.

- Variations of gradient descent can improve performance for this or that special case.
  - See Numerical Recipes in C (and in other languages) by Press, Teukolsky, Vetterling, and Flannery.
  - Simulated Annealing, Linear Programming too

- Works well in smooth spaces; poorly in rough.

Newton's method



- Want to find the roots of f(x)
  - "Root": value of x for which f(x)=0

- Initialize to *some* point x
- Compute the tangent at x & compute where it crosses x-axis

$$\nabla f(x) = \frac{0 - f(x)}{x' - x} \quad \Rightarrow \quad x' = x - \frac{f(x)}{\nabla f(x)}$$

- Optimization: find roots of ∇f(x)

$$\nabla\nabla f(x) = \frac{0 - \nabla f(x)}{x' - x} \quad \Rightarrow \quad x' = x - \frac{\nabla f(x)}{\nabla\nabla f(x)}$$

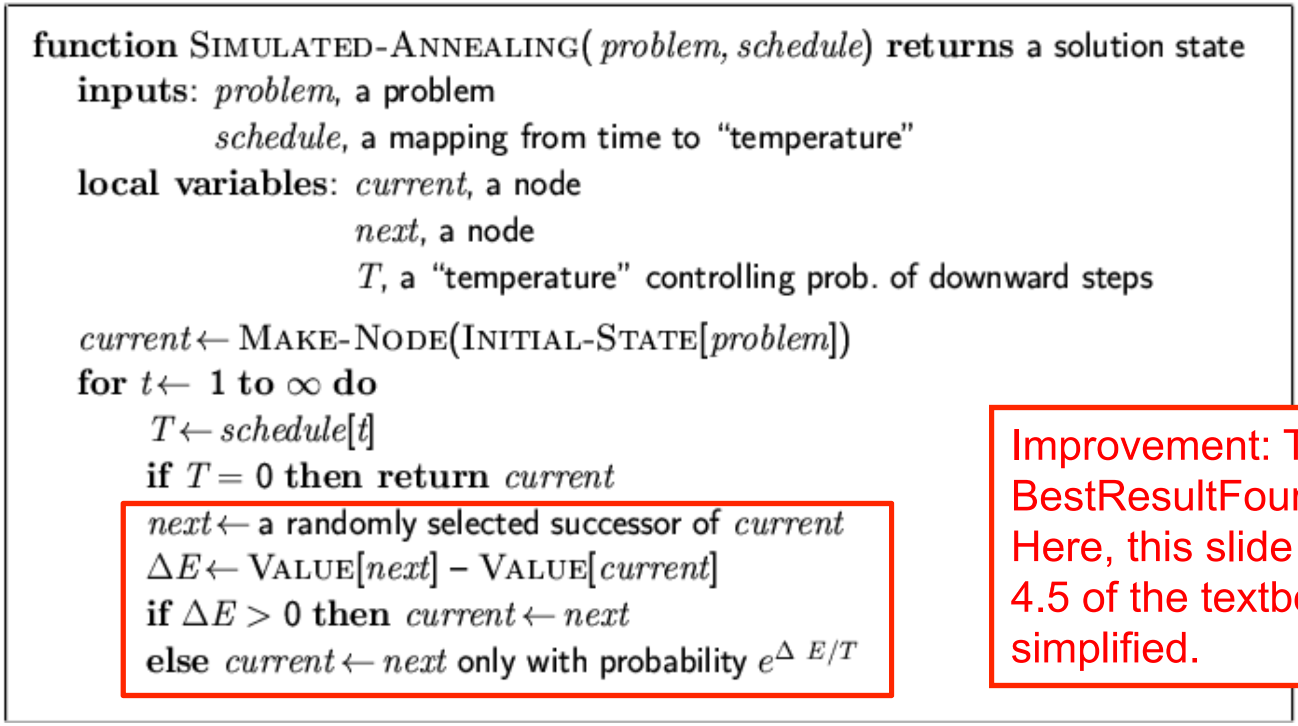
("Step size" λ = 1/∇∇f ; inverse curvature)

  - Does not always converge; sometimes unstable
  - If converges, usually very fast
  - Works well for smooth, non-pathological functions, linearization accurate
  - Works poorly for wiggly, ill-behaved functions

(Multivariate:
∇ f(x) = gradient vector
∇^2 f(x) = matrix of 2nd derivatives
a/b = a b^{-1}, matrix inverse)

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency



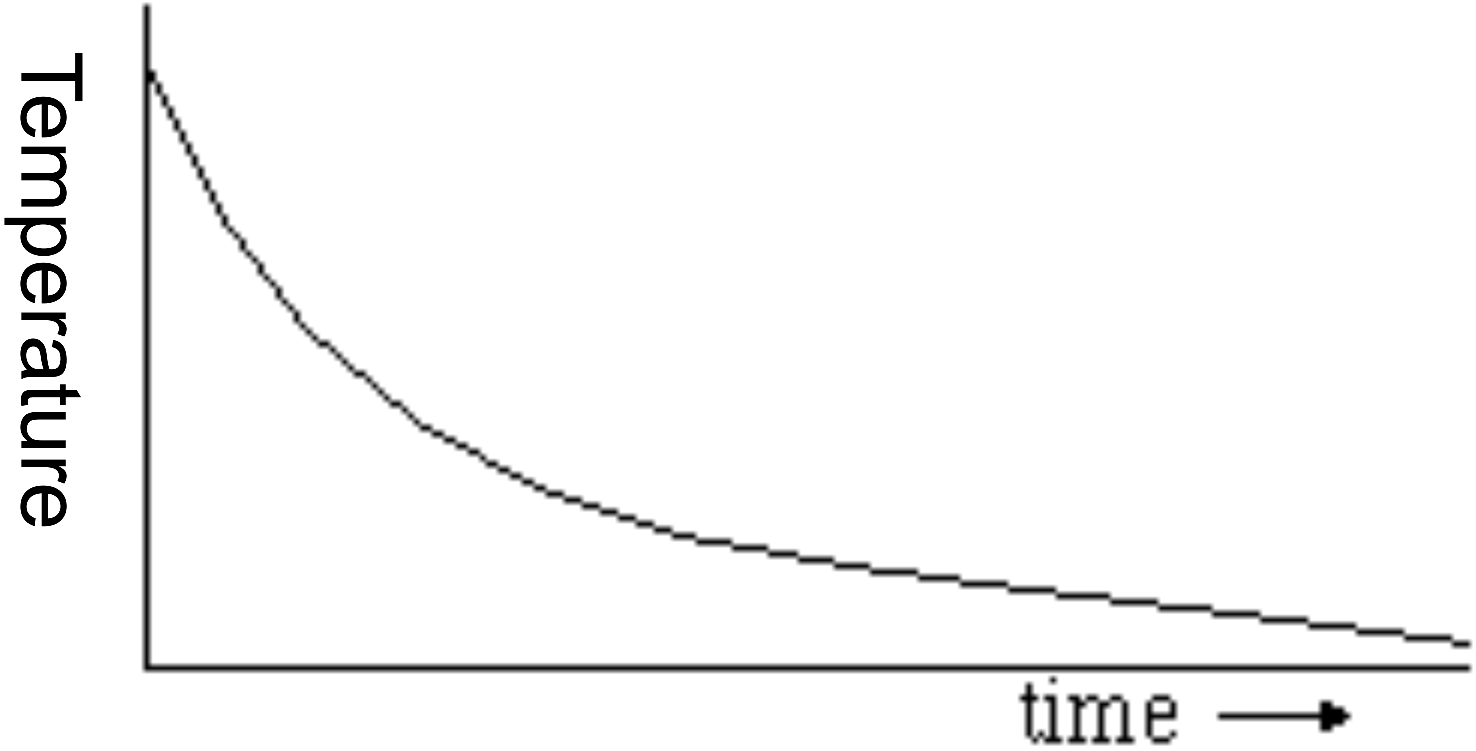
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to “temperature”
    local variables: current, a node
                     next, a node
                     $T$, a “temperature” controlling prob. of downward steps

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

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

Typical annealing schedule

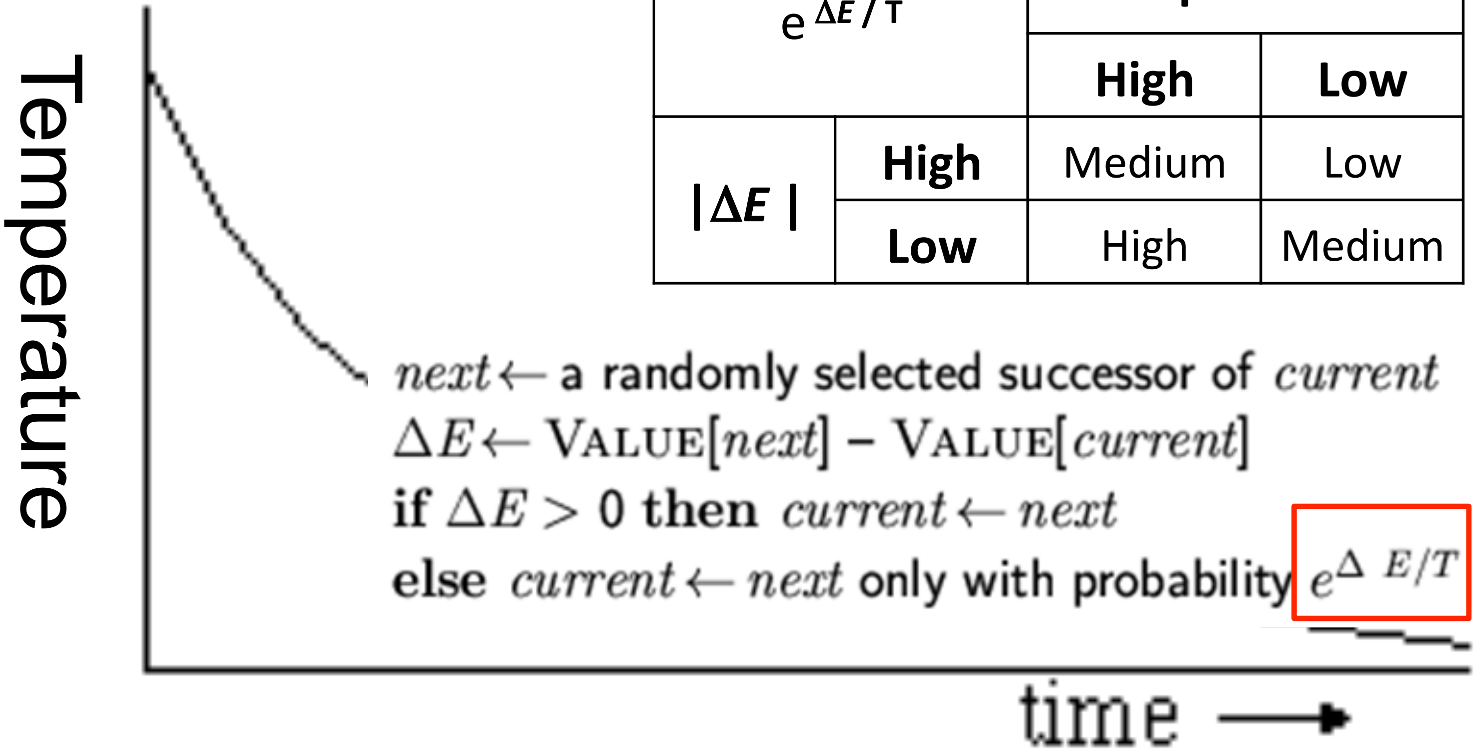
- Usually use a decaying exponential
- Axis values scaled to fit problem characteristics



Pr(accept worse successor)

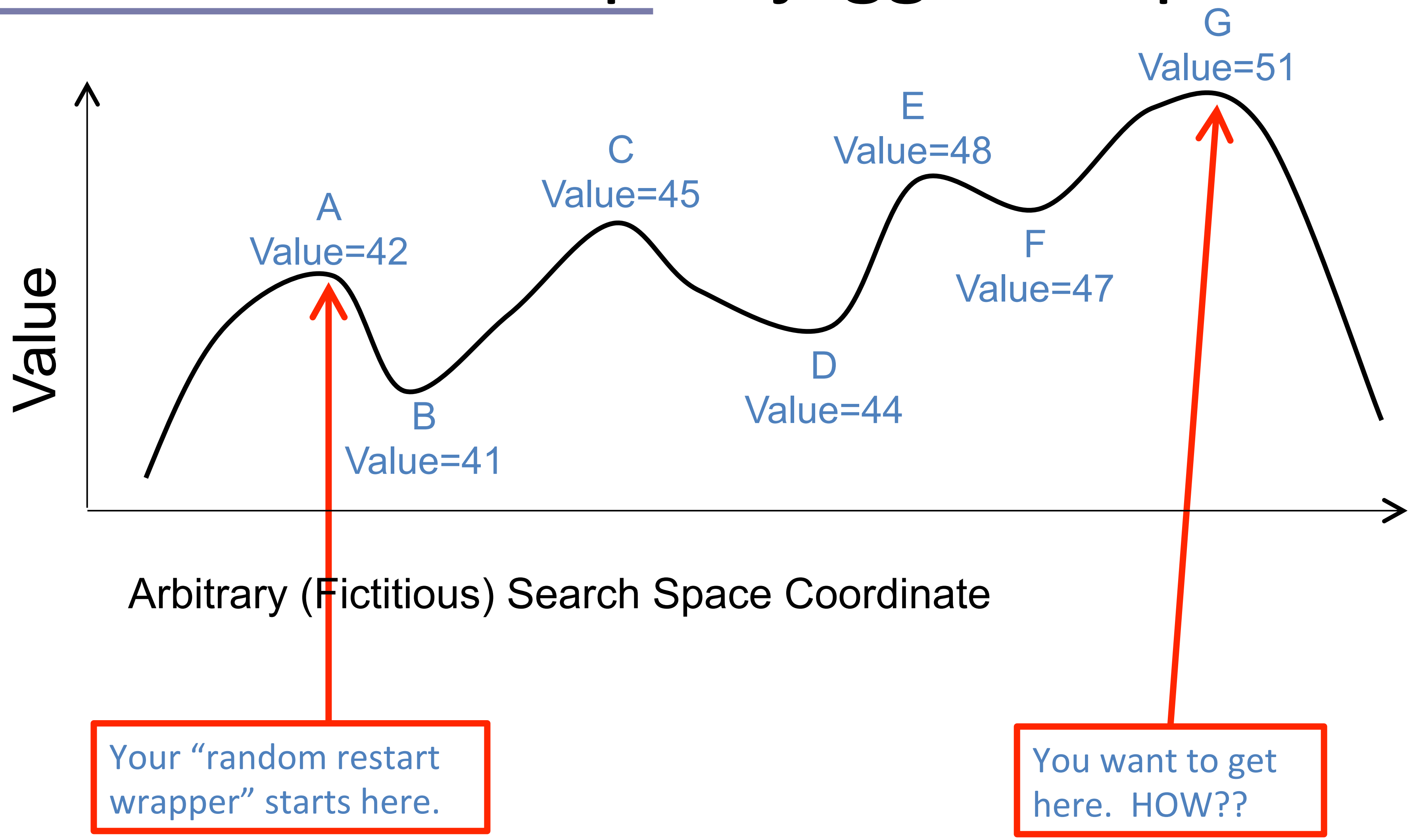
- Decreases as temperature T decreases (accept bad moves early on)
- Increases as $\lvert \Delta E \rvert$ decreases (accept not “much” worse)
- Sometimes, step size also decreases with T

$e^{\Delta E / T}$		**Temperature T**	
		High	**Low**
$\lvert \Delta E \rvert$	**High**	Medium	Low
	Low	High	Medium



```
$next \leftarrow$ a randomly selected successor of $current$
$\Delta E \leftarrow \text{VALUE}[next] - \text{VALUE}[current]$
if $\Delta E > 0$ then $current \leftarrow next$
else $current \leftarrow next$ only with probability $\boxed{e^{\Delta E/T}}$
```

Goal: “ratchet up” a jagged slope

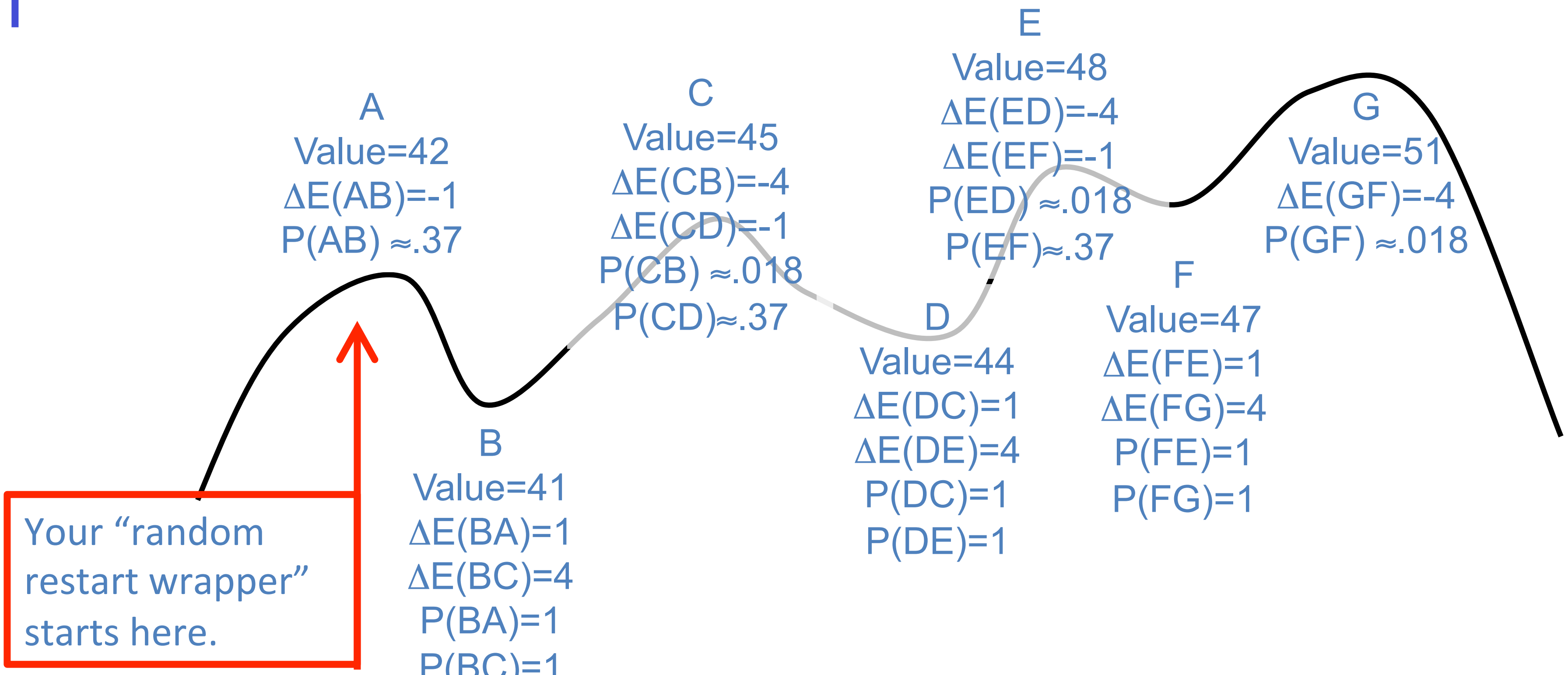


Your “random restart wrapper” starts here.

You want to get here. HOW??

This is an illustrative *cartoon*…

Goal: “ratchet up” a jagged slope



A
Value=42
ΔE(AB)=-1
P(AB) ≈.37

B
Value=41
ΔE(BA)=1
ΔE(BC)=4
P(BA)=1
P(BC)=1

C
Value=45
ΔE(CB)=-4
ΔE(CD)=-1
P(CB) ≈.018
P(CD)≈.37

D
Value=44
ΔE(DC)=1
ΔE(DE)=4
P(DC)=1
P(DE)=1

E
Value=48
ΔE(ED)=-4
ΔE(EF)=-1
P(ED) ≈.018
P(EF)≈.37

F
Value=47
ΔE(FE)=1
ΔE(FG)=4
P(FE)=1
P(FG)=1

G
Value=51
ΔE(GF)=-4
P(GF) ≈.018

Your “random restart wrapper” starts here.

x	-1	-4
e^x	≈.37	≈.018

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?).

This is an illustrative *cartoon*…

Properties of simulated annealing

- One can prove:
  - If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - Unfortunately this can take a VERY VERY long time
  - Note: in any finite search space, random guessing also will find a global optimum with probability approaching 1
  - So, ultimately this is a very weak claim

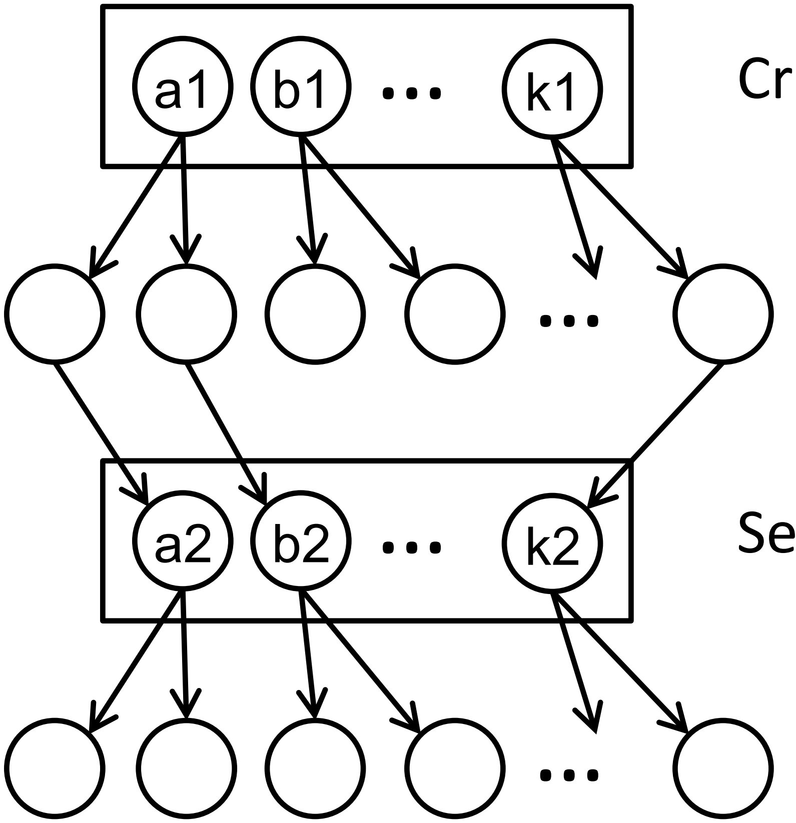
- Often works very well in practice
  - But usually VERY VERY slow

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

Local beam search

- 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.
- Concentrates search effort in areas believed to be fruitful
  - May lose diversity as search progresses, resulting in wasted effort

Local beam search



Create *k* random initial states

Generate their children

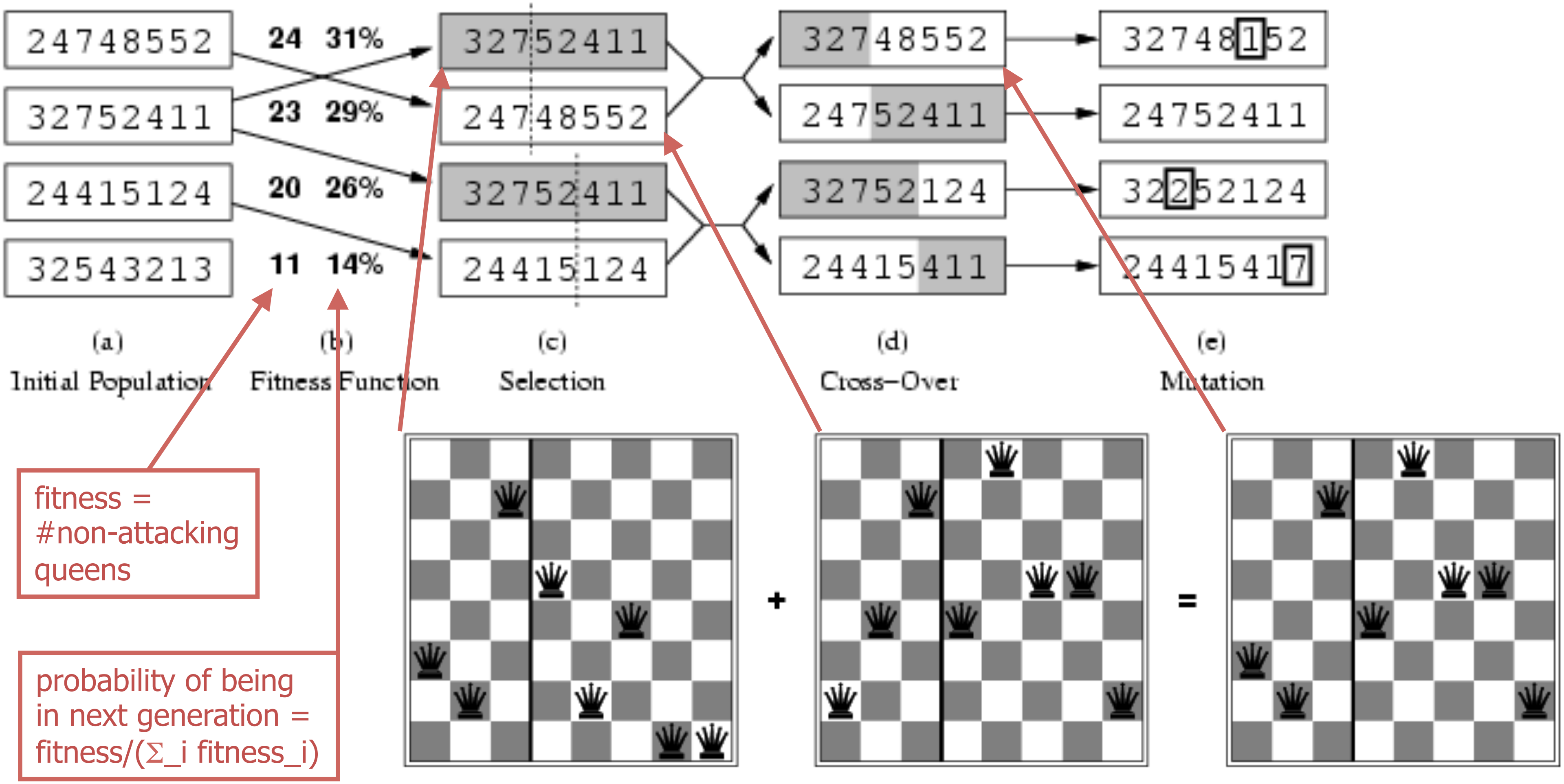
Select the *k* best children

Repeat indefinitely…

Is it better than simply running *k* searches? Maybe…??

Genetic algorithms

- State = a string over a finite alphabet (an individual)
  - A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- Evaluation function (fitness function).
  - Higher values for better states.
- Select individuals for next generation based on fitness
  - P(indiv. in next gen) = indiv. fitness / total population fitness
- Crossover: fit parents to yield next generation (offspring)
- Mutate the offspring randomly with some low probability



fitness = #non-attacking queens

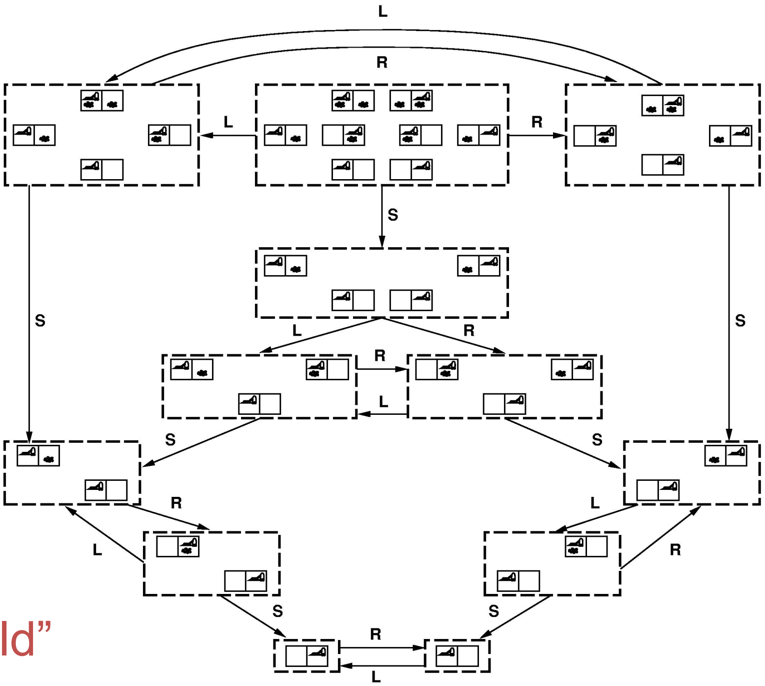
probability of being in next generation = fitness/(Σ_i fitness_i)

How to convert a fitness value into a probability of being in the next generation.

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(pick child_1 for next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(pick child_2 for next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

Partially observable systems

- What if we don't even know what state we're in?
- Can reason over "belief states"
  - What worlds *might* we be in? "State estimation" or "filtering" task
  - Typical for probabilistic reasoning
  - May become hard to represent ("state" is now very large!)

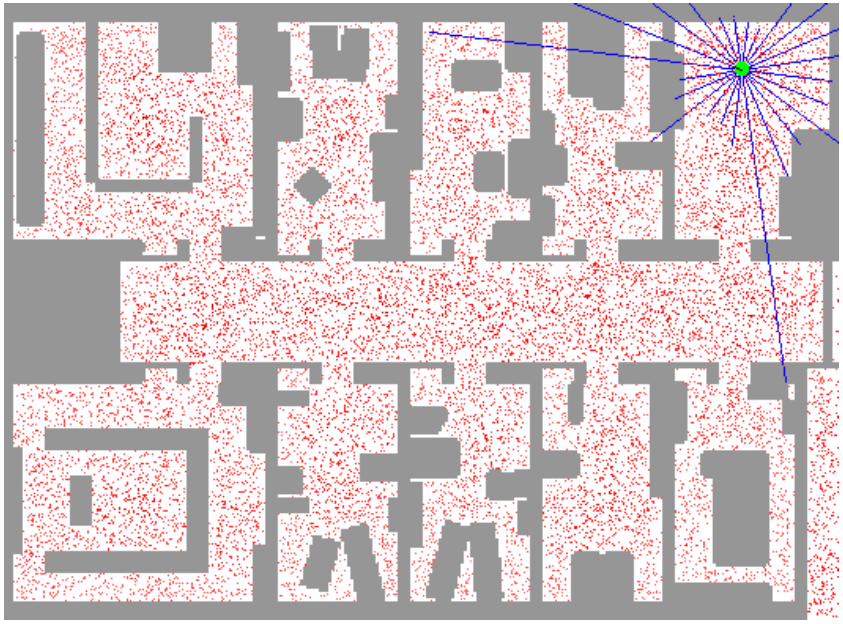


Recall:
"vacuum world"

Partially observable systems

- What if we don’t even know what state we’re in?
- Can reason over “belief states”
  - What worlds *might* we be in? “State estimation” or “filtering” task
  - Typical for probabilistic reasoning
  - May become hard to represent (“state” is now very large!)
- Often use approximate belief state
- Particle filters
  - Population approach to state estimation
  - Keep list of (many) possible states
  - Observations: increase/decrease weights
  - Resampling improves density of samples in high-probability regions

Animation: Deiter Fox, UW



Linear Programming

CS171, Fall 2016

Introduction to Artificial Intelligence

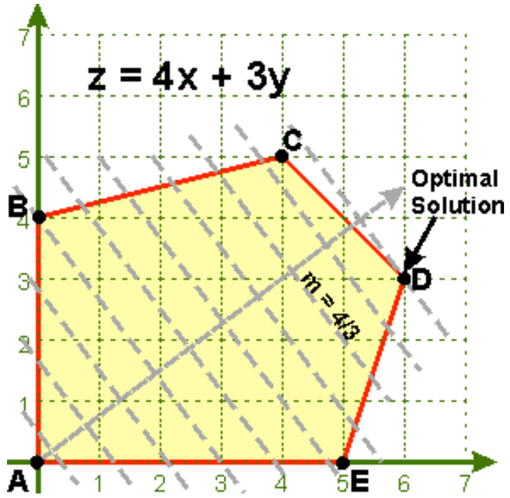
Prof. Alexander Ihler

BREN:ICS
INFORMATION AND COMPUTER SCIENCES

UNIVERSITY of CALIFORNIA IRVINE

Linear programming

- Restricted type of problem, but
- Efficient, **optimal** solutions



- Problems of the form:

Maximize: $v^T x$ (linear objective)

Subject to: $A x \leq b$ (linear constraints)

$C x = d$

  - Very efficient, "off the shelf" solvers available for LPs
  - Can quickly solve large problems (1000s of variables)
    - Problems with additional special structure ⇒ solve *very large* systems!

Summary

- Local search maintains a complete solution
  - Seeks consistent (also complete) solution
  - vs: path search maintains a consistent solution; seeks complete
  - Goal of both: consistent & complete solution

- Types:
  - hill climbing, gradient ascent
  - simulated annealing, Monte Carlo methods
  - Population methods: beam search; genetic / evolutionary algorithms
  - Wrappers: random restart; tabu search

- Local search often works well on large problems
  - Abandons optimality
  - Always has some answer available (best found so far)