

# Local Search and Optimization

# Outline

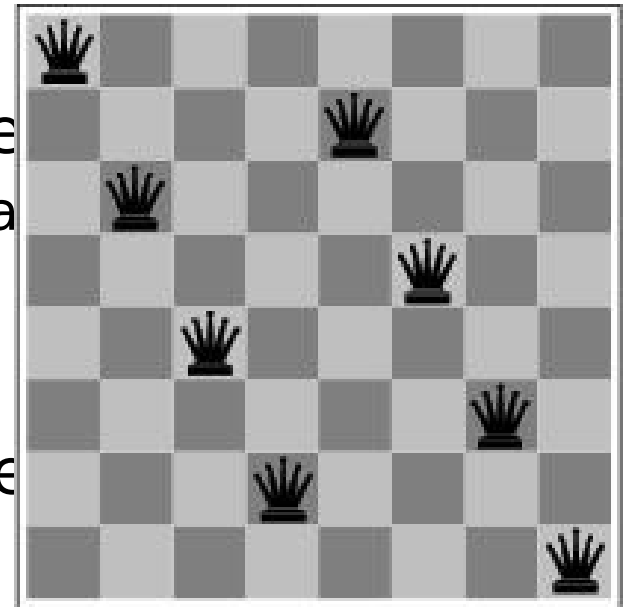
- Local Search Algorithms and Optimization
  - Hill-Climbing
  - Gradient Methods
  - Simulated Annealing
  - Genetic Algorithms
  - Issues with Local Search

# Local Search Algorithms

- In many optimization problems, the **path** to the goal is irrelevant; the goal state itself is the solution
  - Local Search: It is widely used for *very big* problems
  - Returns Good Solution 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

# Local Search and Optimization

- Path to goal is a solution to the problem
  - Systematic exploration of search space.
- State is a solution to the problem
  - for some problems path is irrelevant
  - E.g., 8-queens
- Different algorithms can be used
  - Depth First Branch and Bound
  - Local Search





Goal  
Satisfi  
on

Reach the Goal Node  
Constraint Satisfaction



Optimizati  
on

Optimize(Objective Fn)  
Constraint Optimization

We can go back and forth between the two problems  
Typically in the same complexity class

# Local Search and Optimization

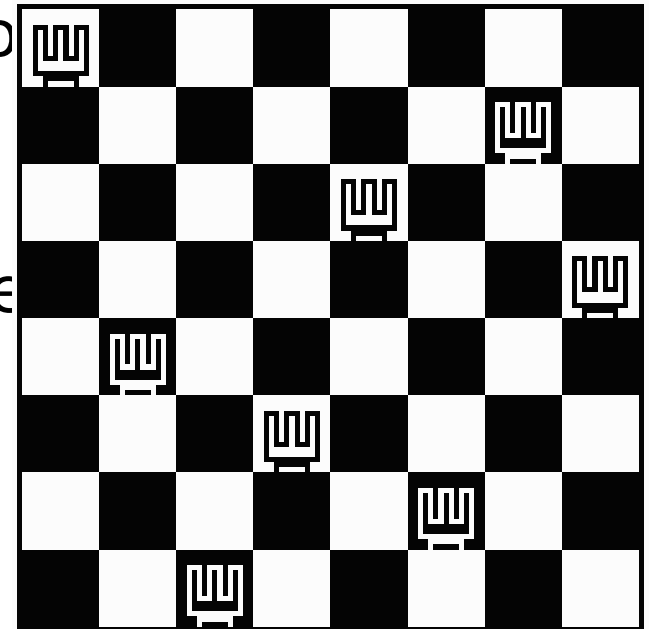
- Local Search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths
- Advantages:
  - Use very little memory (space)
  - Can often find Reasonable Solutions in large or infinite (continuous) state spaces for which the other algorithms are not suitable.
- “Pure Optimization” Problems
  - All states have an Objective Function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems.

# Optimization

- Local search is often suitable for optimization problems. Search for best state by optimizing an **Objective Function**.
- **$F(\mathbf{x})$**  where  **$\mathbf{x}$**  is a vector of continuous or discrete values
- Begin with a complete configuration
- A successor of state  **$S$**  is  **$S$**  with a single element changed
- Move from the current state to a successor state
- Low memory requirements, because the search tree or graph is not maintained in memory (rather, one node at a time)

# Examples

- 8 queens: Find an arrangement of 8 queens on a chess board such that no two queens are attacking each other
- Start with some arrangement of queens, one per column
- $X[j]$ : row of queen in column  $j$
- Successors of a state: move one queen to another row in the same column
- $F$ : # pairs attacking each other





# Examples

- Traveling Salesperson Problem: Visit each city exactly once
- Start with some ordering of the cities
- State Representation – Order of the cities visited
- Successor State: A change to the current ordering
- $F$ : Length of the route

# Examples

- Flight Travel Problem
- Flight Schedule. The general problem is for all members of the family to travel to the same place.
- A state consists of flights for each member of the family.
- Successor States: All schedules that have one person on the next later or the earlier departing or returning flight.
- F: Sum of different types of costs (\$\$, time, etc)

# Examples

- Cryptosystems: Resistance to types of attacks is often discussed in terms of Boolean functions used in them
- Much work on constructing the Boolean functions with desired cryptographic properties (balancedness, high nonlinearity, etc.)
- One Approach: Local search, where states are represented with truth tables (that's a simplification); a successor results from a change to the truth table; objective functions have been devised to assess (estimate) relevant qualities of the Boolean functions

# Examples

- Racing yacht hull design
- Design representation has multiple components, including a vector of B-Spline surfaces.
- Successors: Modification of a B-Spline surface.
- Objective Function estimates the time the yacht would take to traverse a course given certain wind conditions

# Visualization

- States are laid out in a landscape
- Height corresponds to the objective function value
- Move around the landscape to find the highest  
(or lowest) peak
- Only keep track of the current states and immediate neighbors

# Algorithm Design Considerations

- How do we represent our problem?
- What is a “complete state”?
- What is our objective function?
  - How do we 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 we compute a neighbor or a step?
- Are there any constraints we can exploit?

# Trivial Algorithms

- Random Sampling
  - Generate a state randomly
- Random Walk
  - Randomly pick a neighbor of the current state
- Both algorithms asymptotically complete.

# 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.

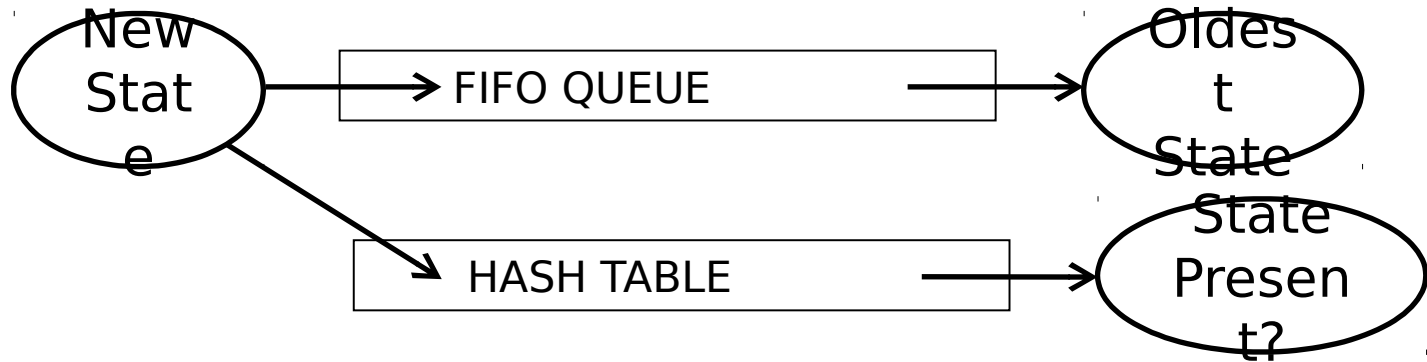
# Tabu Search

- Prevent returning quickly to the same state
- Keep fixed length queue (“Tabu list”)
- add most recent state to queue; drop oldest
- Never make the step that is currently Tabu’ed
- Properties:
  - As the size of the Tabu list grows, hill-climbing will asymptotically become “non-redundant” (won’t look at the same state twice)
  - In practice, a reasonable sized Tabu list (say 100 or so) improves the performance of hill climbing in many problems

# 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; } }
```

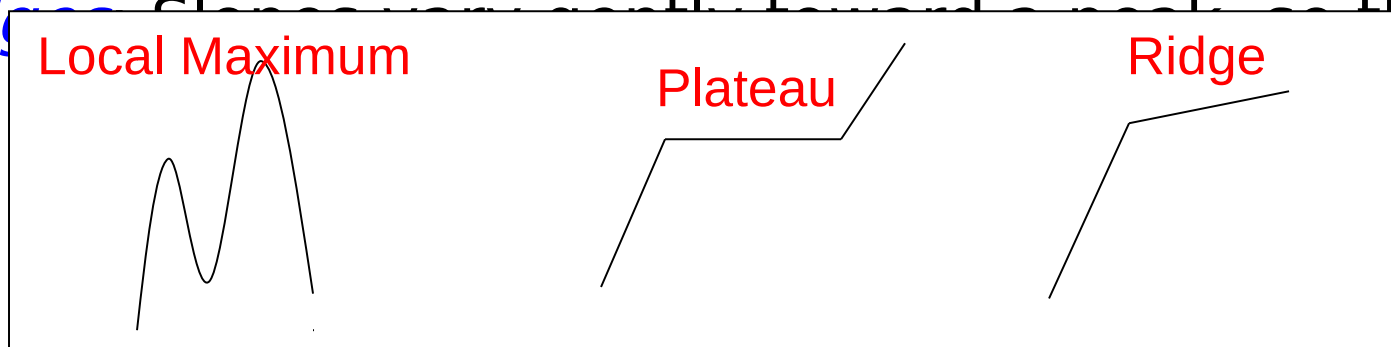
# Local Search Algorithms

- ❑ Two strategies for choosing the state to visit next
- Hill-Climbing Search
  - Gradient Descent in continuous state spaces
  - Can use e.g. Newton's method to find roots
- Simulated Annealing Search
- ❑ Then, an extension to multiple current states:
- Local Beam Search
- Genetic Algorithm

# Hill Climbing

## (Greedy Local Search)

- Generate nearby successor states to the current state
- Pick the best and replace the current state with that one.
- Loop
- *Local Maximum*: A peak that is lower than the highest peak, so a **Suboptimal Solution** is returned
- *Plateau*: The evaluation function is flat, resulting in a random walk
- *Ridge*: A narrow path of local maxima that leads to the global maximum



# Hill-Climbing (Greedy Local Search)

max version

**function** HILL-CLIMBING(*problem*) **return** a state that is a local maximum

**input:** *problem*, a problem

**local variables:** *current*, a node.

*neighbor*, a node.

*current*  $\leftarrow$  MAKE-NODE(INITIAL-STATE[*problem*])

**loop do**

*neighbor*  $\leftarrow$  the highest valued successor of *current*

**if** VALUE [*neighbor*]  $\leq$  VALUE[*current*] **then return** STATE[*current*]

*current*  $\leftarrow$  *neighbor*

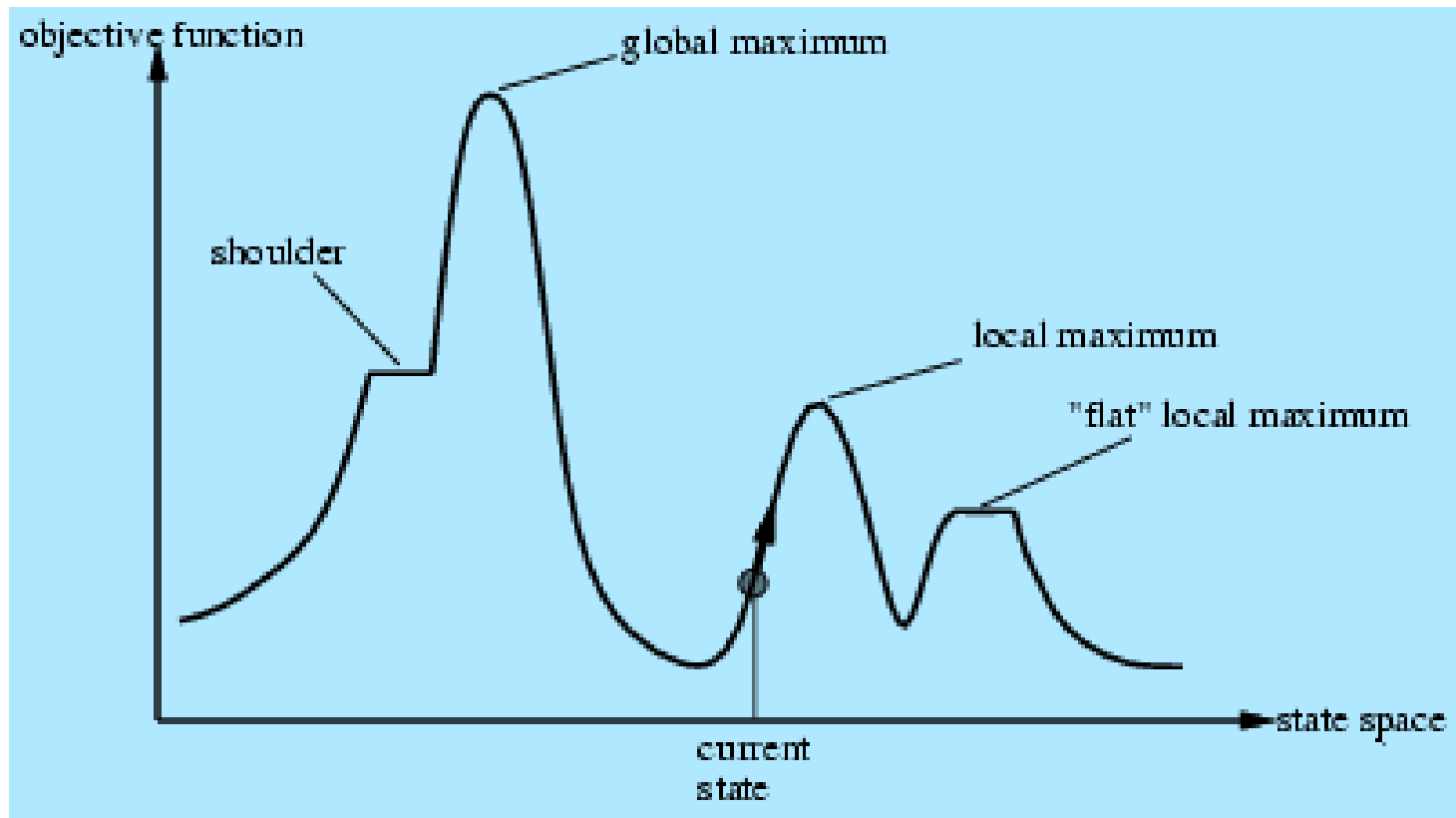
min version will reverse inequalities and look for the lowest valued successor

# Hill-Climbing Search

- “A loop that continuously moves towards increasing value”
  - terminates when a peak is reached
  - Aka **Greedy Local Search**
- Value can be either
  - Objective Function Value
  - Heuristic Function Value (Minimized)
- Hill Climbing does not look ahead of the immediate neighbors
- Can randomly choose among the set of best successors
  - **NOTE: Amnesia refers to the Loss of Memories, such as Facts, Information and Experiences.**
- **“Climbing Mount Everest in a Thick Fog with Amnesia”**



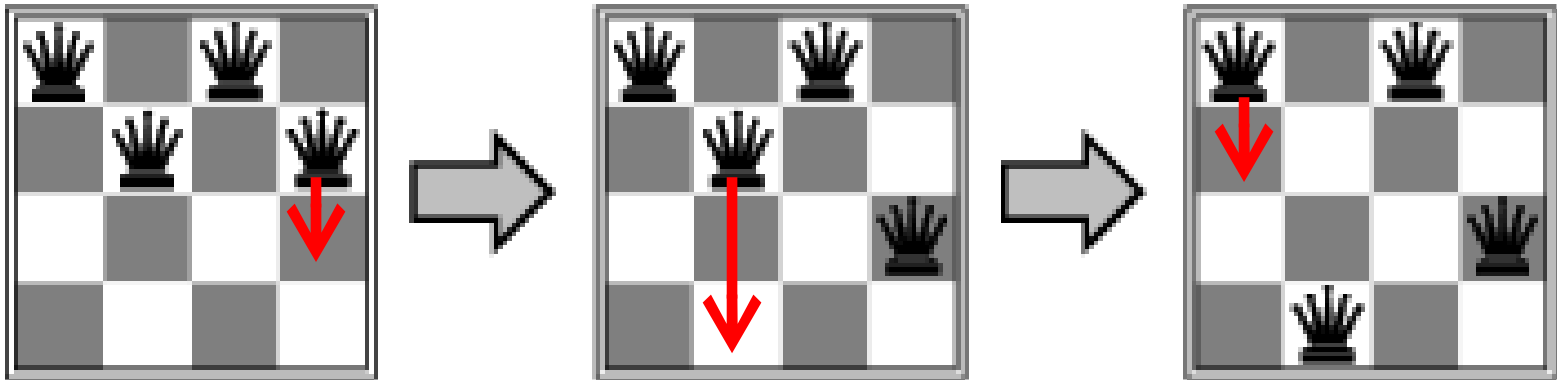
# “Landscape” of Search



Hill Climbing gets stuck in Local Minima  
depending on?

# Example: $n$ -queens

- Goal: Put  $n$  queens on an  $n \times n$  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...











- Is it a Satisfaction or Optimization Problem?

# Ex: Hill-Climbing Search, 8-queens Problem

Need to convert this to an Optimization Problem

$h$  = # 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

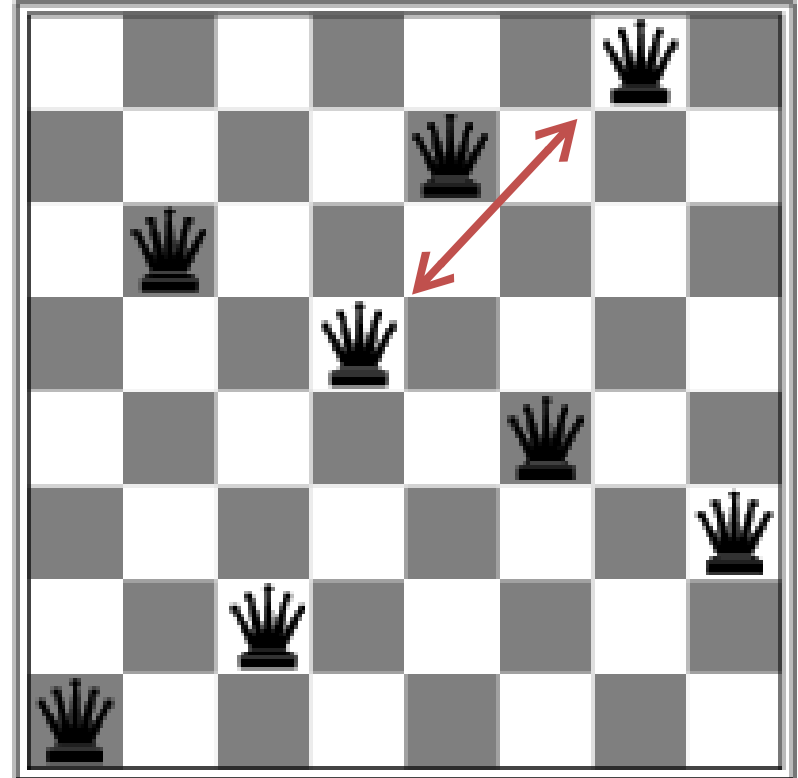
12 (boxed) = best  $h$  among all neighbors; select one randomly

# Search Space

- State
  - All 8 queens on the chess board in some configuration
- Successor Function
  - move a queen to another square in the same column.
- Example of a Heuristic Function  $h(n)$ :
  - The # of pairs of queens that are attacking each other
  - (so we want to minimize this)

# Hill-Climbing Search: 8-queens Problem

- Is this a solution?
- What is the value of  $h$ ?
- A local minimum with  $h=1$
- All one-step neighbors have higher  $h$  values
- What can we do to get out of this local minimum?

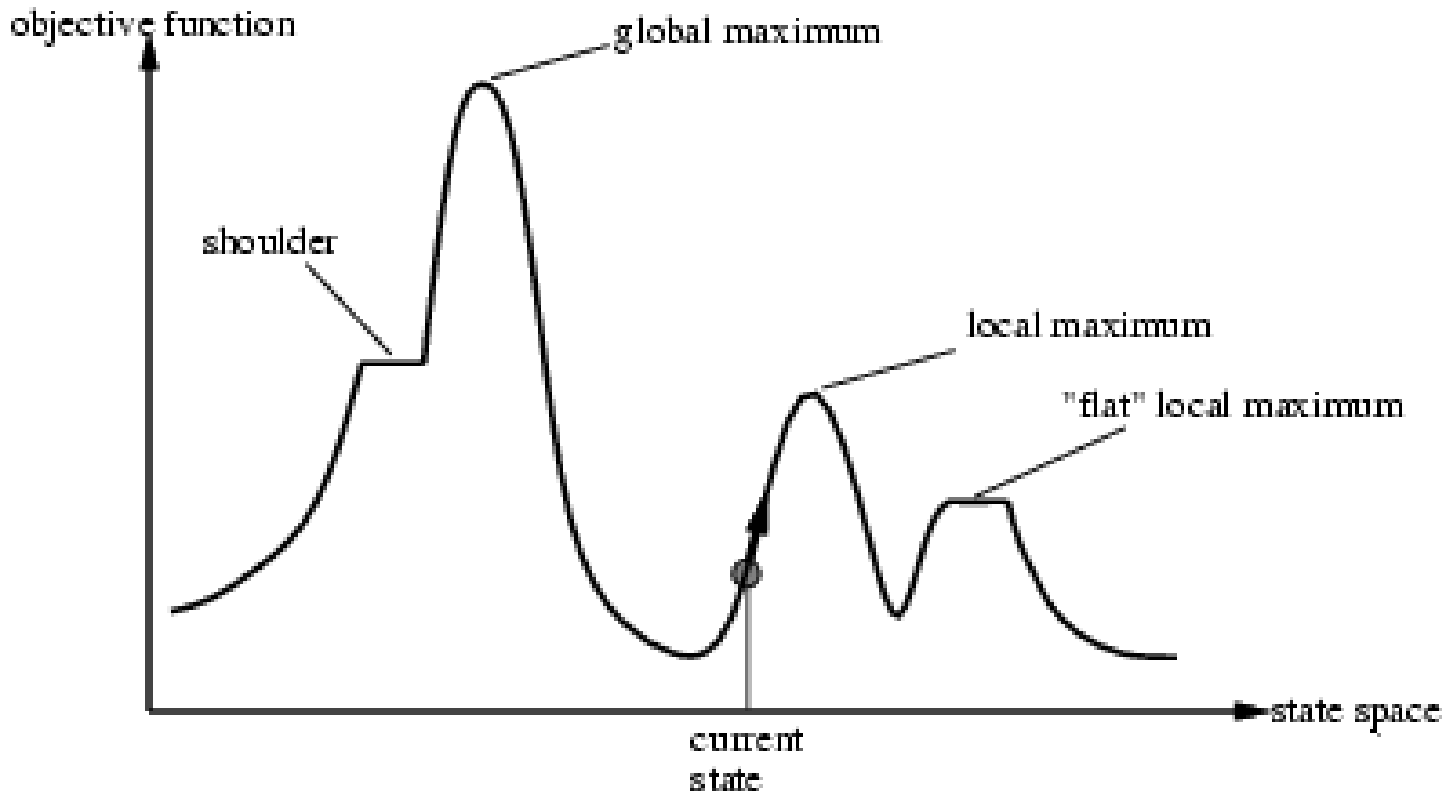


# Hill-Climbing Difficulties

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, it can stick in local

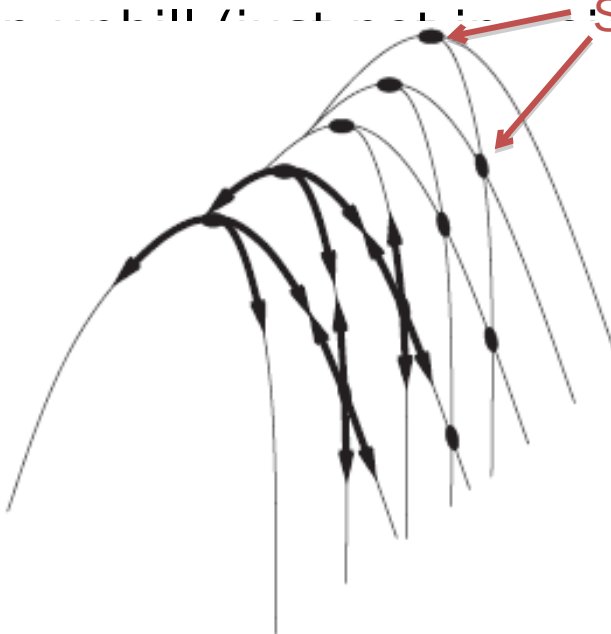
- 



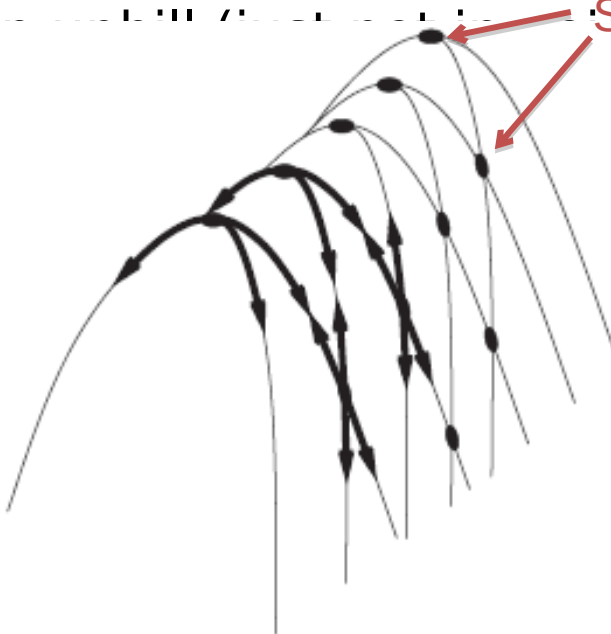
# Hill-Climbing Difficulties

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 Ridge:  States / steps (discrete 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.



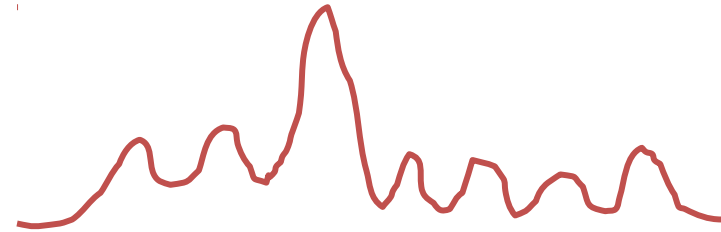
# Hill-Climbing on 8-queens

- Randomly generated 8-queens starting states...
  - 14% the time it solves the problem
  - 86% of the time it get stuck at a Local Minimum
  - However...
    - Takes only 4 steps on average when it succeeds
    - And 3 on average when it gets stuck
- (for a state space with  $8^8 = \sim 17$  million

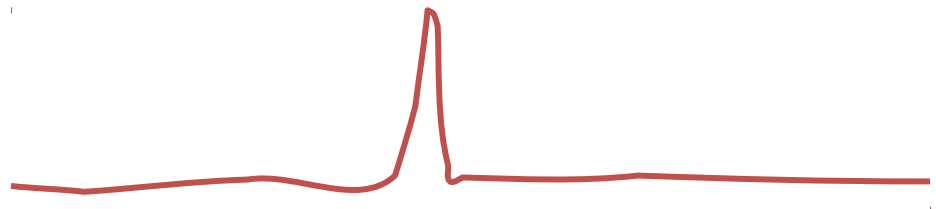


# Hill Climbing Drawbacks

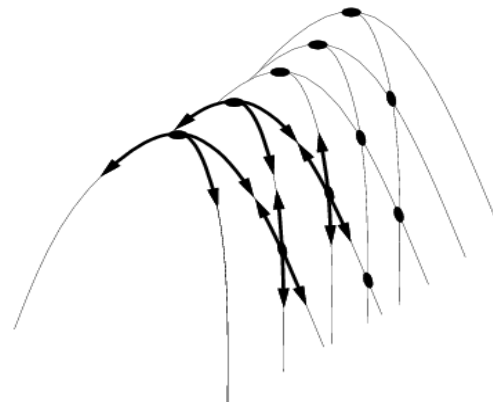
- Local Maxima



- Plateaus



- Diagonal Ridges



# Escaping Shoulders: Sideways Move

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%
  - However....
    - 21 steps for every successful solution
    - 64 for each failure

# Escaping Shoulders/Local Optima

## Enforced Hill Climbing

- Perform Breadth First Search from a Local Optima
  - to find the next state with better  $h$  function
- Typically,
  - prolonged periods of exhaustive search
  - bridged by relatively quick periods of hill-climbing
- Middle ground b/w local and systematic<sup>35</sup>

# Hill-Climbing: Stochastic Variations

- Stochastic Hill-Climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- To avoid getting stuck in Local Minima
  - Random-walk Hill-Climbing
  - Random-restart Hill-Climbing
  - Hill-Climbing with both

# Hill Climbing: Stochastic Variations

→ When the state-space landscape has local minima, any search that moves only in the greedy direction cannot be complete

→ Random walk, on the other hand, is asymptotically complete

Idea: Put random walk into Greedy Hill-Climbing

# Hill-Climbing with Random Restarts

- If at first we don't succeed, try, try again!
- Different variations
  - For each restart: run until termination vs. run for a fixed time
  - Run a fixed number of restarts or run indefinitely
- Analysis
  - Say each search has probability  $p$  of success
    - E.g., for 8-queens,  $p = 0.14$  with no sideways moves
  - Expected number of restarts?
  - Expected number of steps taken?
- If we want to pick one local search algorithm, learn this one!!

# Hill-Climbing with Random Walk

- At each step do one of the two
  - Greedy: With **prob.  $p$**  move to the neighbor with largest value
  - Random: With **prob.  $1-p$**  move to a random neighbor

## Hill-Climbing with both

- At each step do one of the three
  - Greedy: move to the neighbor with largest value
  - Random Walk: move to a random neighbor
  - Random Restart: Resample a new current state

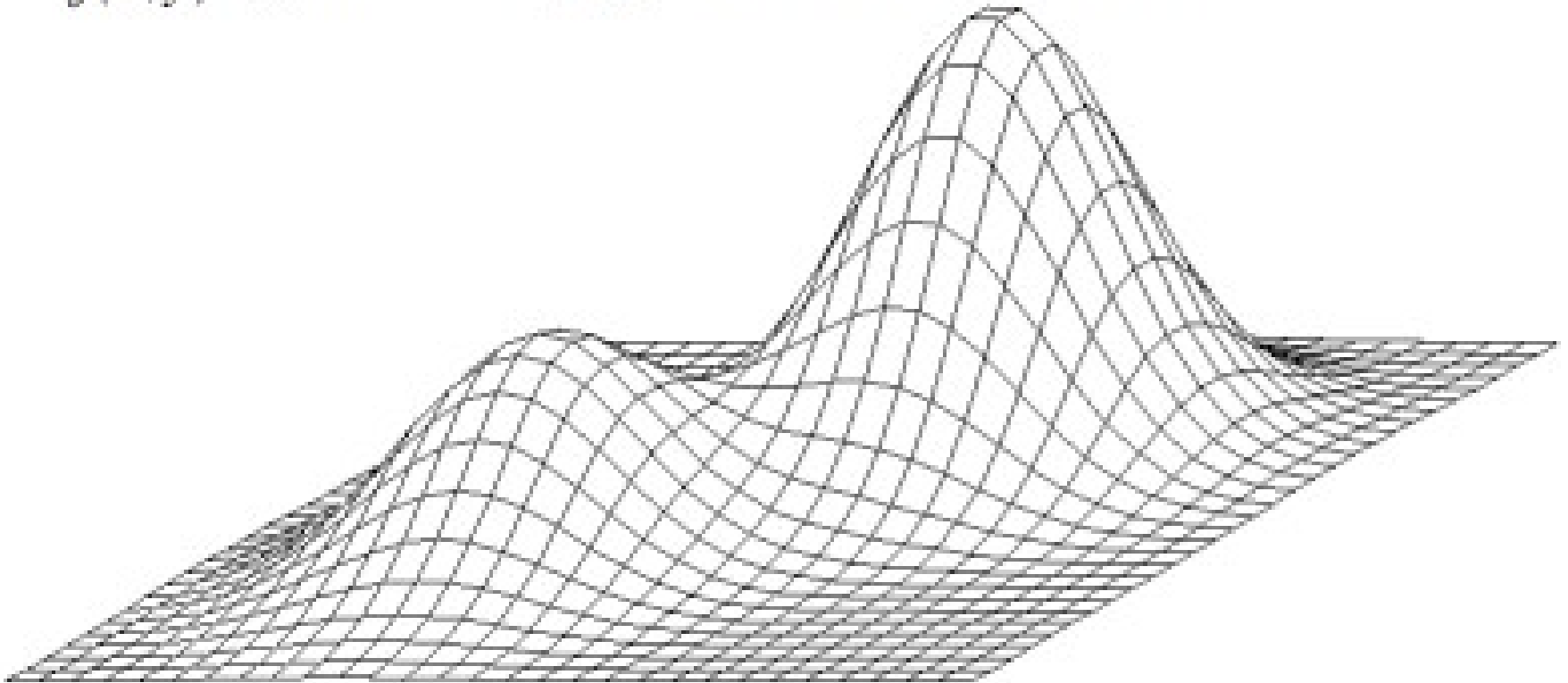
# Optimization of Continuous Functions

- Discretization
  - Use Hill-Climbing
- Gradient Descent
  - Make a move in the direction of the gradient
    - Gradients: Closed Form or Empirical



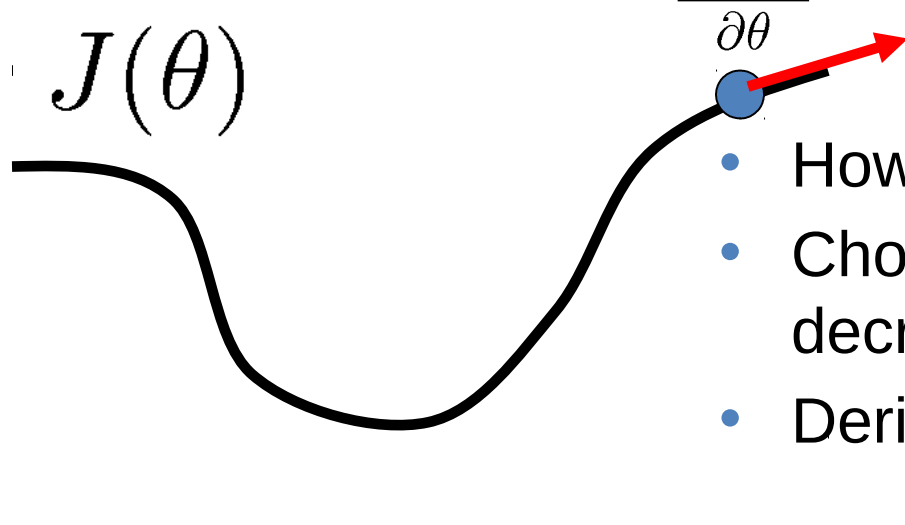
# Gradient Descent

$$f(x,y)=e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$



# Gradient Descent

- Hill-Climbing in Continuous State Spaces
- Denote “State”  $\mu$ ; Cost as  $J(\mu)$



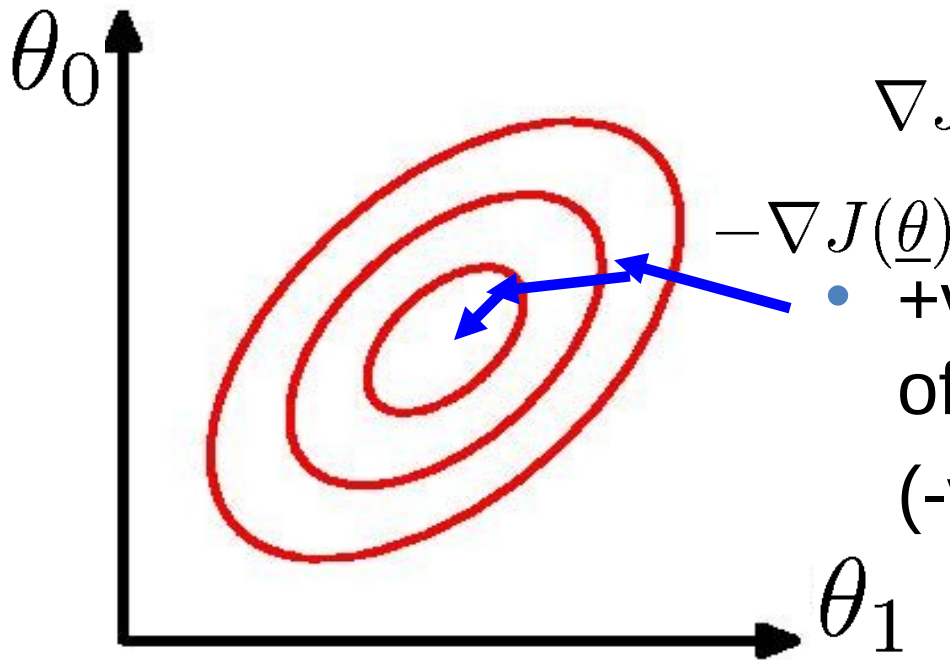
- How to change  $\mu$  to improve  $J(\mu)$ ?
- Choose a direction in which  $J(\mu)$  is decreasing
- Derivative  $\frac{\partial J(\theta)}{\partial \theta}$ 
  - Positive (+ve)  $\Rightarrow$  Increasing
  - Negative (-ve)  $\Rightarrow$  Decreasing

# Gradient Descent

- Hill-Climbing in Continuous Spaces

- Gradient Vector

$$\nabla J(\underline{\theta}) = \begin{bmatrix} \frac{\partial J(\underline{\theta})}{\partial \theta_0} & \frac{\partial J(\underline{\theta})}{\partial \theta_1} & \dots \end{bmatrix}$$

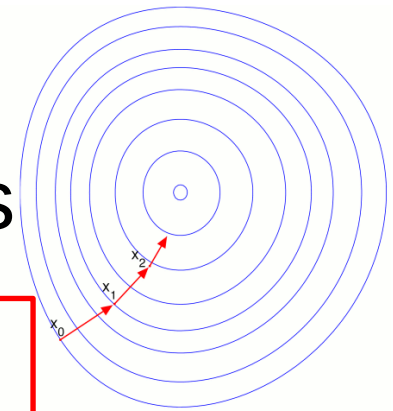


- +ve Indicates the direction of Steepest Ascent  
(-ve = Steepest Descent)

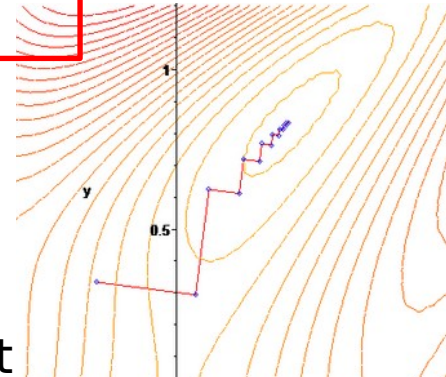
# Gradient Descent

## ■ Hill-Climbing in Continuous Spaces

Gradient = The most direct direction up-hill in the Objective (Cost) Function, so its negative direction minimizes the Cost Function.



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



1. Compute the *gradient* :  $\frac{\partial}{\partial x_i} J(x_1, \dots, x_n) \quad \forall i$
2. Take a small step downhill in the direction of the gradient

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

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

4. If true then accept move, if not “reject” (decrease step size)

5. Repeat.

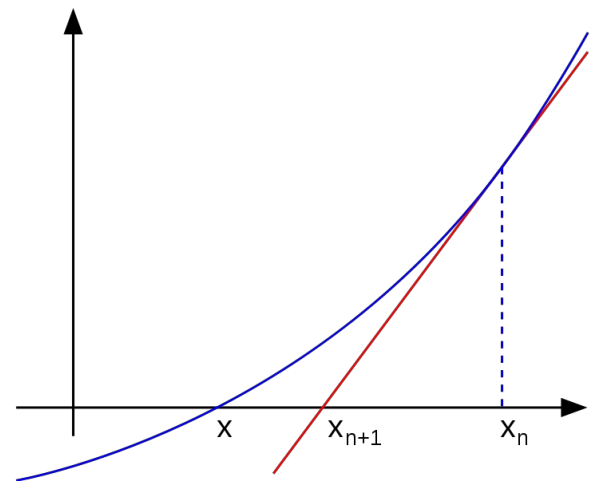
### • How to select $\lambda$

- Line search: successively double
  - until  $f$  starts to increase again

# Gradient Descent

- Hill-Climbing in Continuous Spaces
- How do we 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-Raphson Method for Function Minimization



- Want to find the roots of a polynomial  $f(x)$ 
  - “Root”: value of  $x$  for which  $f(x)=0$
- Initialize to *some* point  $x$
- To minimize a function  $f(x)$ , we need to find the roots of  $f'(x)$   
 $=\nabla f(x) = 0$

- Compute the next point  $x'$  such that the tangent line crosses x-axis
- $$\nabla f(x) = \frac{0 - f(x)}{x' - x} \Rightarrow x' = x - \frac{f(x)}{\nabla f(x)}$$

- Optimize using the Hessian
- $$\nabla \nabla f(x) = \frac{0 - \nabla f(x)}{x' - x} \Rightarrow x' = x - \frac{\nabla f(x)}{\nabla \nabla f(x)}$$

$H_f(x)$

Hessian is costly to compute  
( $n^2$  double derivative entries for an  $n$ -dimensional vector)  
□ approximations

(“Step size” =  $\frac{1}{\nabla \nabla f(x)}$ , inverse curvature)

- Does not always converge; sometimes unstable
- If converges, usually very fast
- Works well for smooth, non-pathological functions, linearization
- Equivalent to fitting a quadratic function for  $f(x)$  accurate
- Works poorly for wiggly, ill-behaved functions in the local neighborhood of  $x$ .

(Multivariate:

$\nabla f(x)$  = gradient vector

$\nabla \nabla f(x)$  = matrix of 2<sup>nd</sup> derivatives

$a/b = a b^{-1}$ , matrix inverse)

# Simulated Annealing (SA)

- Since the first development of Simulated Annealing (SA) by Kirkpatrick et al., SA has been applied in almost every area of optimization. The metaphor of SA came from the annealing characteristics in metal/metallurgy processing; however, SA has, in essence, strong similarity to the classic **Metropolis Algorithm** developed by Metropolis et al.

# Metropolis Algorithm

- Simulate behaviour of a physical system according to principles of statistical mechanics.
- Globally biased toward “downhill” steps, but occasionally makes “uphill” steps to break out of local minima.
- Gibbs-Boltzmann function. The probability of finding a physical system in a state with energy  $E$  is proportional to  $e^{-E / (kT)}$ , where  $T > 0$  is temperature and  $k$  is a Boltzmann constant.
- For any temperature  $T > 0$ , function is monotone decreasing function of energy  $E$ . System is more likely to be in a lower energy state than higher one.

-  $T$  large: high and low energy states have same probability



# Metropolis Algorithm

Metropolis algorithm.

- Given a fixed temperature  $T$ , maintain current state  $S$ .
- Randomly perturb current state  $S$  to new state  $S' \in N(S)$ .
- If  $E(S') \leq E(S)$ , update current state to  $S'$ .  
Otherwise, update current state to  $S'$  with probability  $e^{-\Delta E / (kT)}$ , where  $\Delta E = E(S') - E(S) > 0$ .

**Theorem.** Let  $f_S(t)$  be fraction of first  $t$  steps in which simulation is in state  $S$ . Then, assuming some technical conditions, with probability 1:

$$\lim_{t \rightarrow \infty} f_S(t) = \frac{1}{Z} e^{-E(S) / (kT)},$$

$$\text{where } Z = \sum_{S \in N(S)} e^{-E(S) / (kT)}.$$

**Intuition.** Simulation spends roughly the right amount of time in each state, according to Gibbs-Boltzmann equation.

# Simulated Annealing (SA)

- SA is based on a Metallurgical Metaphor (Physics inspired twist on Random Walk)
  - Start with a temperature set very high and slowly reduce it.
  - Run Hill-Climbing with the twist that we can occasionally replace the Current State with a Worse State based on the current temperature and how much worse the new state is.
- Basic Ideas:
  - Like Hill-Climbing, SA identifies the quality of the local improvements
  - Pick one randomly instead of picking the best move
  - Let  $\delta$  be the change in the objective function
  - If  $\delta$  is positive, then move to that state
  - Otherwise:
    - Move to this state with probability proportional to  $\delta$
    - Thus: Worse moves (very large negative  $\delta$ ) are executed less often
  - However, there is always a chance of escaping from local maxima
  - Over time, make it less likely to accept locally bad moves
  - (Can also make the size of the move random as well

# Physical Interpretation of Simulated Annealing

- A Physical Analogy:
  - Imagine letting a ball roll downhill on the function surface
    - This is like Hill-Climbing (for Minimization)
  - Now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking
    - This is like Simulated Annealing
- **Annealing** = Harden metals by heating them to a high temperature and then gradually cooling them. It is the Physical Process of Cooling a Metal until Particles Achieve a Certain Frozen Crystal State.
  - Simulated Annealing (SA):
    - Free Variables are like Particles
    - Seek “Low Energy” (High Quality) Configuration
    - Slowly Reducing Temp.  $T$  with Particles Moving around Randomly

# Simulated Annealing (SA)

- More formally...
  - Generate a random new neighbor from current state.
  - If it's better take it.
  - If it is worse then take it with some probability proportional to the temperature  $T$  and  $\Delta E$  between the new and old states.
- Probability of a move decreases with the amount  $\Delta E$  by which the evaluation is worsened
- A second parameter (Temperature  $T$ ) is also used to determine the probability: high  $T$  allows more worse moves,  $T$  close to zero results in few or no bad moves
- *Temperature Schedule* input determines the value of  $T$  as a function of the completed cycles

# Simulated Annealing Search

- Idea: Escape local maxima by allowing some "bad" moves but **gradually decrease** their

**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**  $\infty$  **do**

*T* ← *schedule*[*t*]

**if** *T* = 0 **then return** *current*

*next* ← a randomly selected successor of *current*

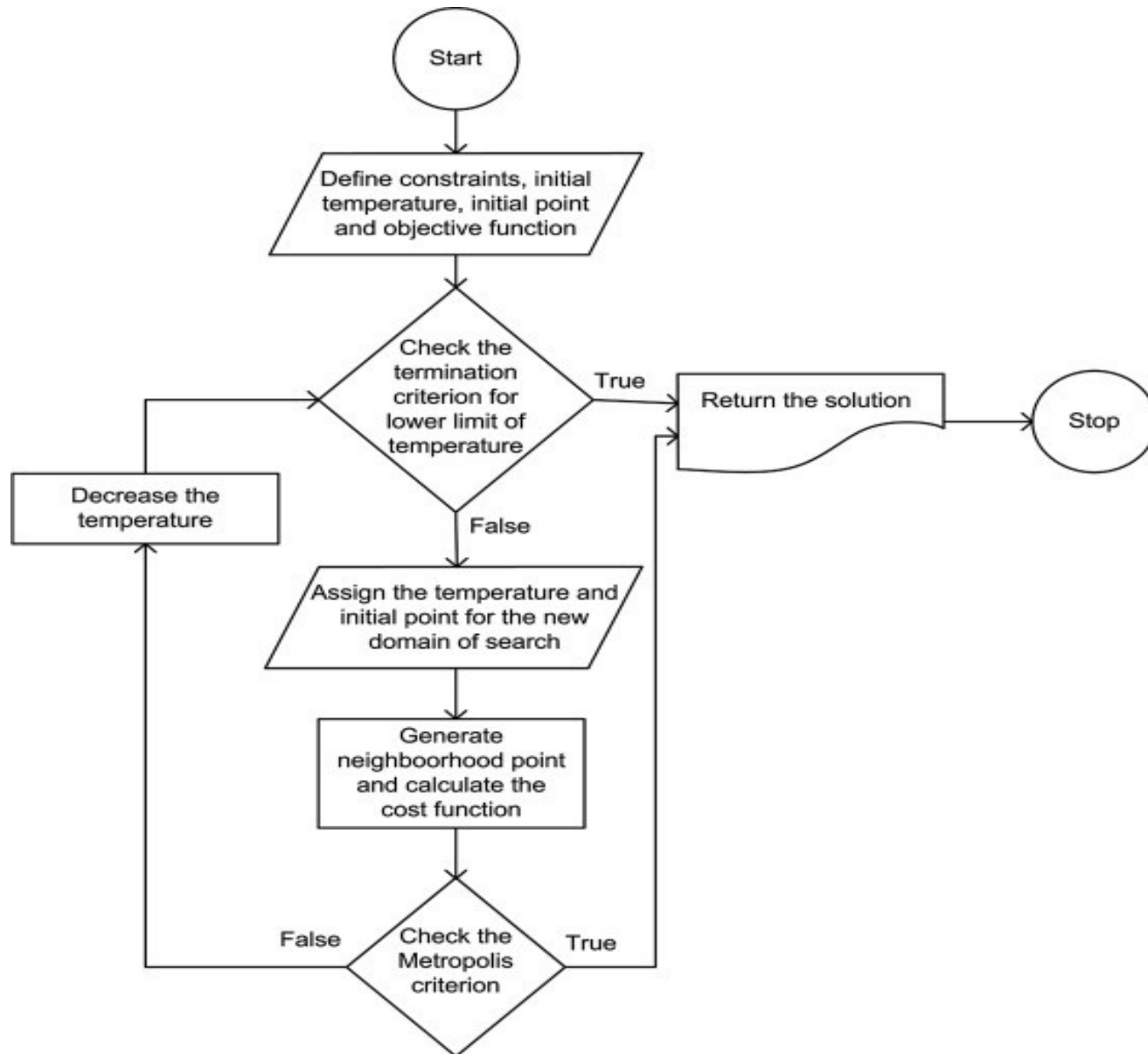
$\Delta E$  ← VALUE[*next*] – VALUE[*current*]

**if**  $\Delta E > 0$  **then** *current* ← *next*

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

Improvement: Track the  
Best Result Found So far

# Simulated Annealing Search



# Typical Annealing Schedule

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



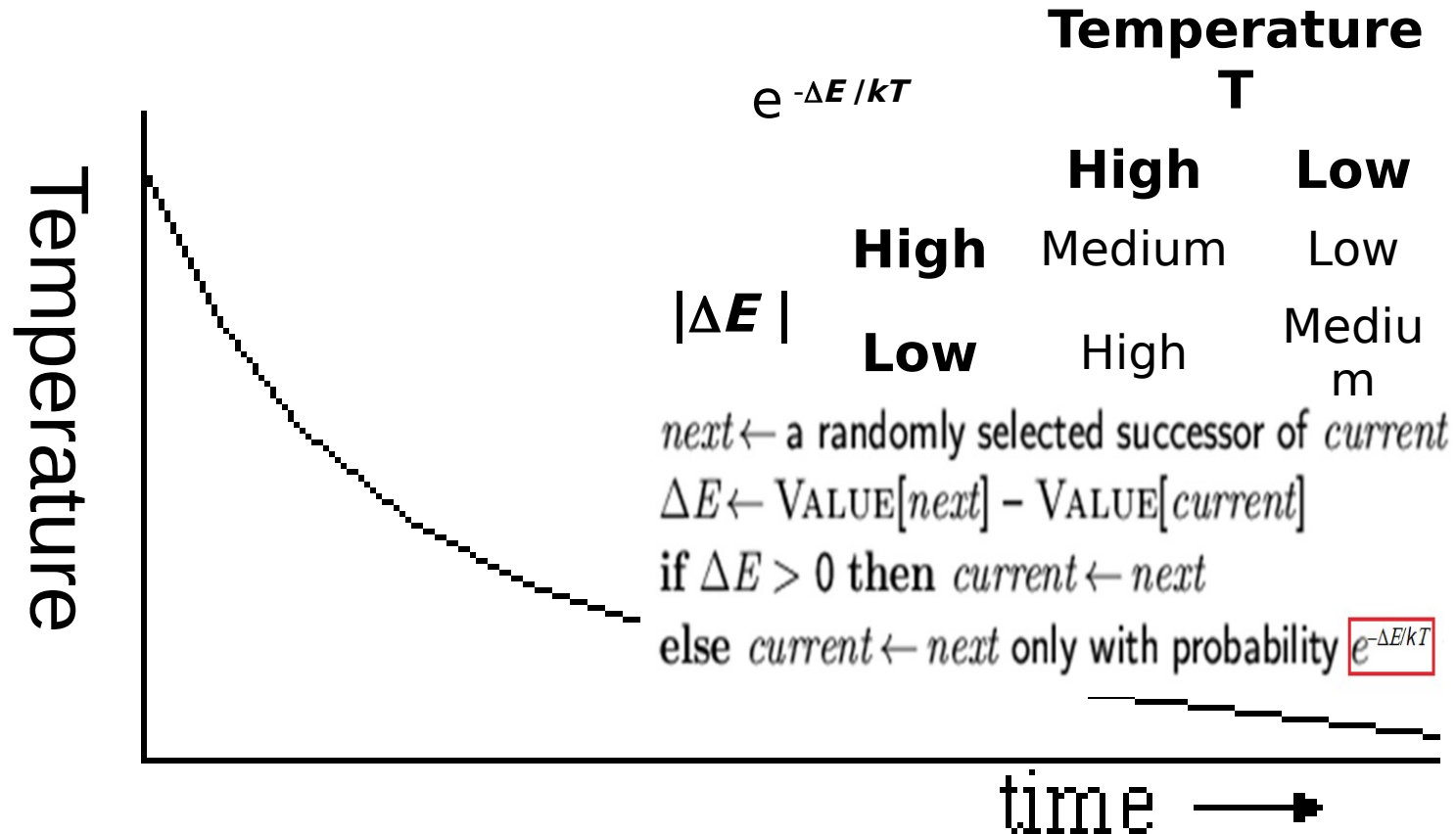
# Temperature T

- High T: probability of “locally bad” move is higher
- Low T: probability of “locally bad” move is lower
- Typically, T is decreased as the algorithm runs longer i.e., there is a “temperature schedule”
- Update the current state S to the new state S' with the probability  $p = e^{-\Delta E / kT}$
- Where, k is the Boltzmann's constant, and for simplicity, we can set k=1. Then T is the temperature for controlling the annealing process

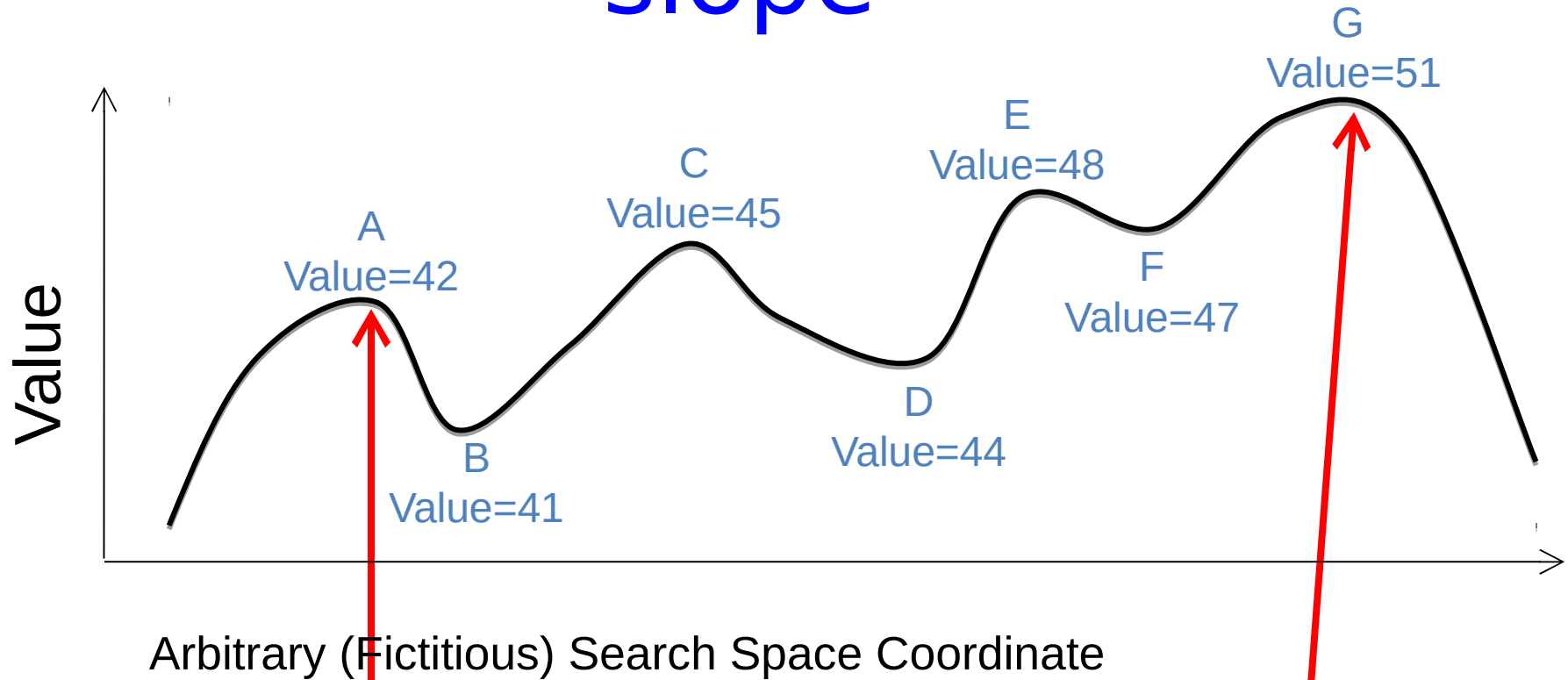


# Pr( Accept Worse Successor )

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



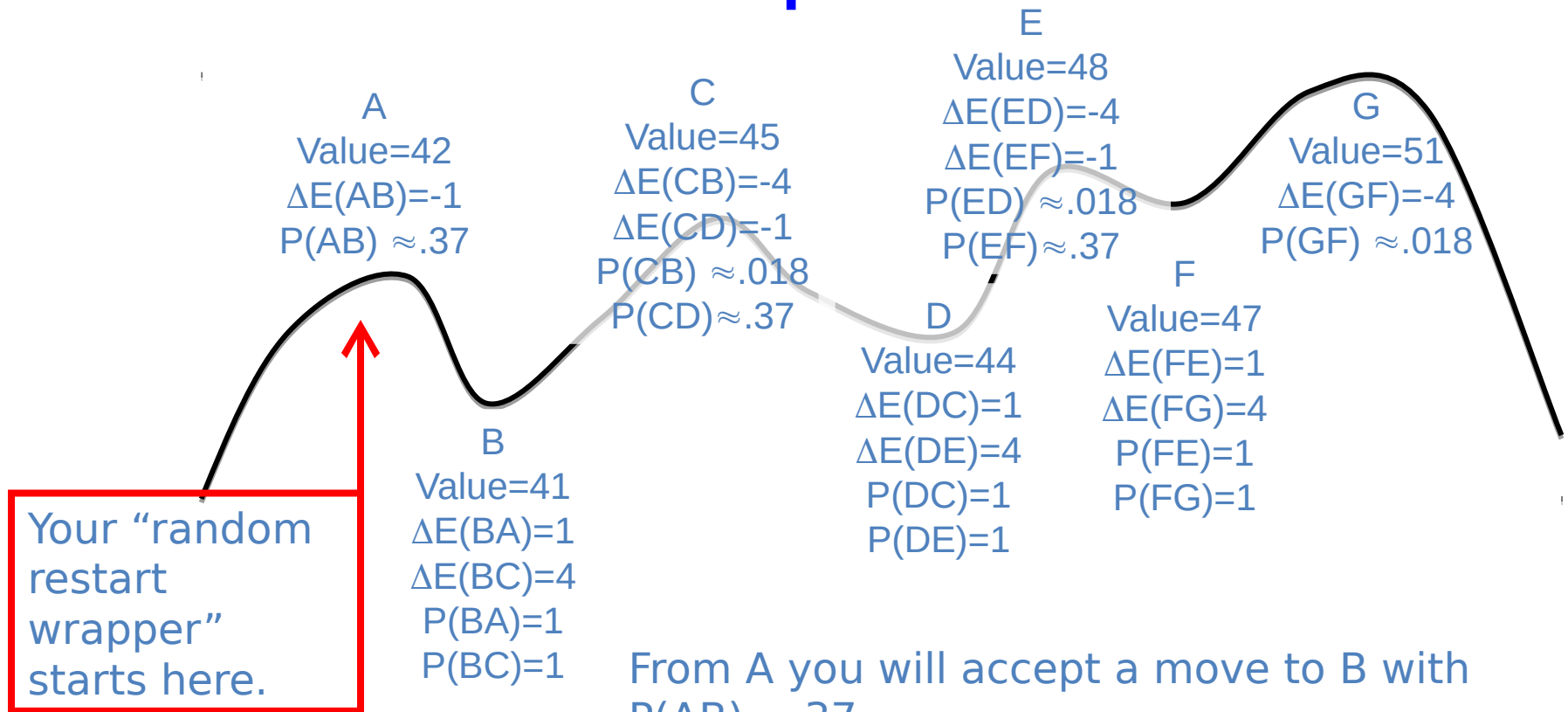
# Goal: "ratchet up" a jagged slope



Your "random restart wrapper" starts here.

You want to get here. HOW??

# Goal: "ratchet up" a jagged slope



$x$	-1	-4
$e_x$	$\approx .37$	$\approx .018$

From A you will accept a move to B with  $P(AB) \approx .37$ .

From B you are equally likely to go to A or to C.

From C you are  $\approx 20X$  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  $\approx 20X$  more likely to go to F

# 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

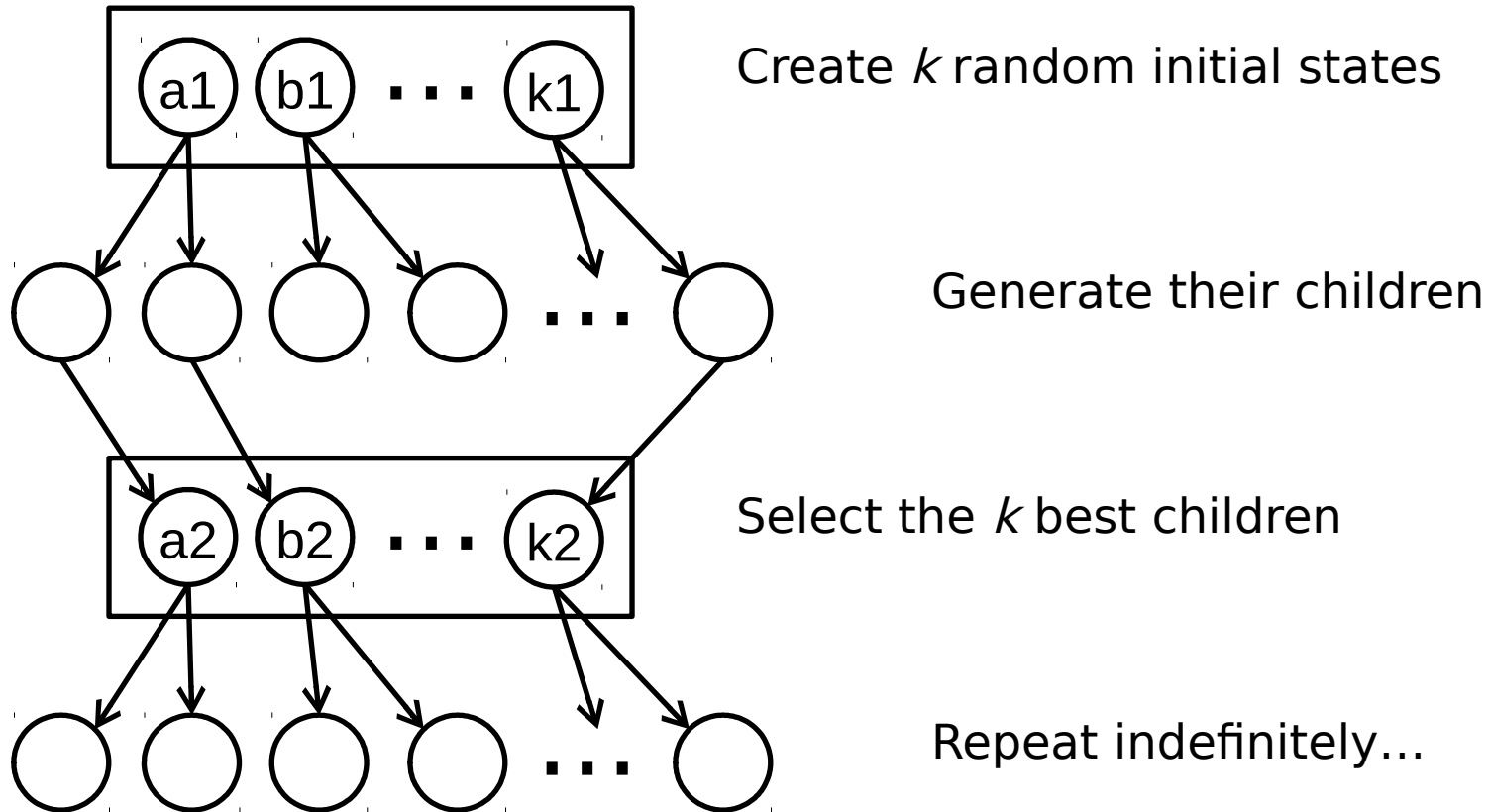
# Simulated Annealing in Practice

- Method proposed in 1983 by IBM researchers for solving VLSI Layout Problems (Kirkpatrick et al., *Science*, 220:671-680, 1983).
  - Theoretically will always find the global optimum
- Other Applications: Traveling Salesperson Problem, Graph Partitioning, Graph Coloring, Scheduling, Facility Layout, Image Processing, ...
- Useful for some problems, but can be very slow

# 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 in Hill-Climbing
- 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 (Contd...)



Is it better than simply running  $k$  searches?  
Maybe...??

# Local Beam Search (Contd...)

- Not the same as *k random-start searches run in parallel!*
- Searches that find good states recruit other searches to join it. Successors can become concentrated in a part of state space
- Problem: Quite often, all *k states end up on same Local Hill*
- Idea: Stochastic Beam Search
  - Choose *k successors randomly, biased towards good ones, with probability of choosing a given successor increasing with value*
- Observe the close analogy to natural selection!
- Natural Selection: Successors (**Offspring**) of a state (**Organism**) populate the next generation



# Genetic Algorithm (GA)

- GA is a variant of Stochastic Beam Search
- Twist on Local Search: Successor is generated by combining two parent states
- State is represented as a string over a finite alphabet (**Individual**)
  - A successor state is generated by combining two parent states
  - 8-queens Problem
    - State = Position of 8 queens each in a column
- Start with  $k$  randomly generated states (**Initial Population**)
- Evaluation Function (**Fitness Function**):
  - Higher values for better states.
  - Opposite to Heuristic Function, e.g., # non-attacking pairs in 8-queens
- Produce the next generation of states by “Simulated Evolution”
- **Selection**: Select individuals for next generation based on fitness
  - $P(\text{indiv. in next gen}) = \text{indiv. fitness} / \text{total population fitness}$
- **Crossover**: Fit parents to yield next generation

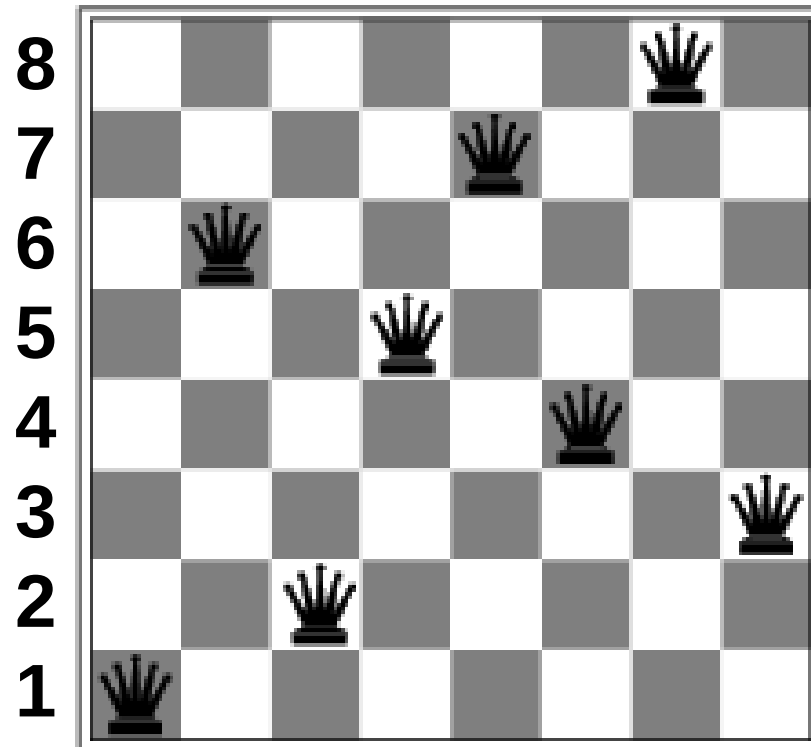
# Genetic Algorithm (GA)

- Representation of Individuals
  - *Classic Approach*: individual is a string over a finite alphabet with each element in the string called a *Gene*
  - Usually binary instead of AGTC as in real DNA
- Selection Strategy
  - Random
  - Selection probability proportional to fitness
  - Selection is done with replacement to make a very fit individual reproduce several times
- Reproduction
  - Random pairing of *Selected Individuals*
  - Random selection of *Crossover* points
  - Each gene can be altered by a random *Mutation*

# Genetic Algorithm (GA)

```
function GA (pop, fitness-fn)
Repeat
  new-pop = {}
  for i from 1 to size(pop):
    x = rand-sel(pop,fitness-fn)
    y = rand-sel(pop,fitness-fn)
    child = reproduce(x,y)
    if (small rand prob): child
    □mutate(child)
    add child to new-pop
  pop = new-pop
Until an indiv is fit enough, or out of
time
Return best indiv in pop, according to
fitness-fn
function reproduce(x,y)
  n = len(x)
  c = random num from 1 to n
  return:

  append(substr(x,1,c),substr(y,c+1,
  n))
```



String Representation  
16257483

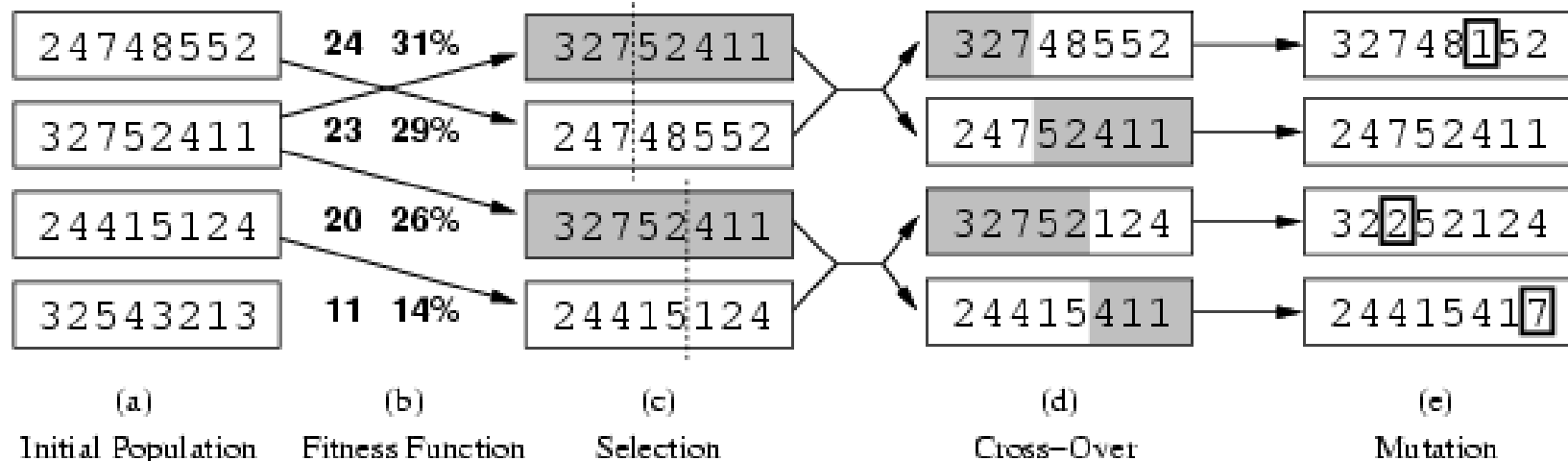
Can we evolve 8-queens through Genetic Algorithm?

# Evolving 8-queens



?

# Genetic Algorithm



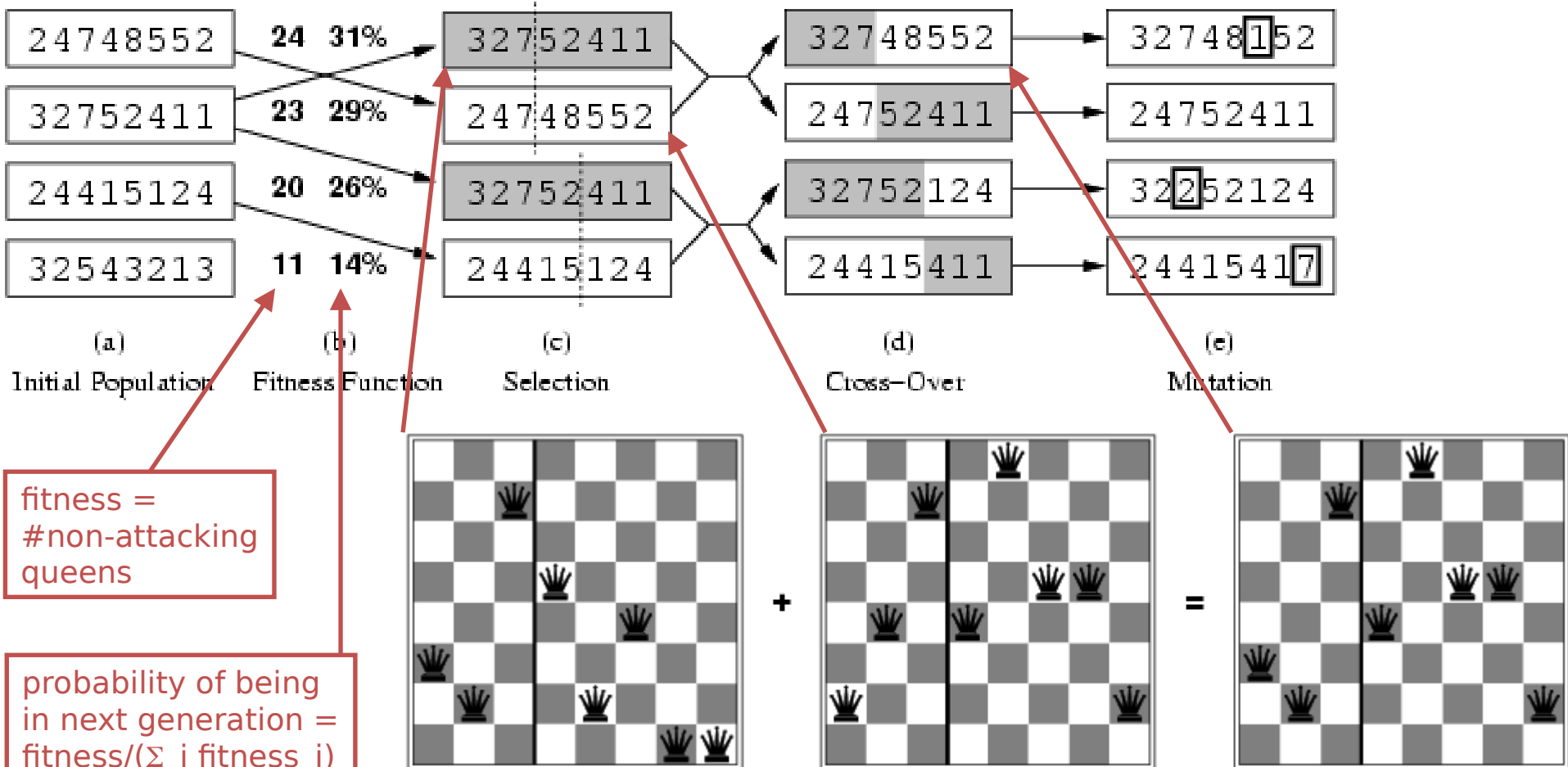
4 states for  
8-queens  
problem

2 pairs of 2 states  
randomly selected based  
on fitness. Random  
crossover points selected

New states  
after crossover

Random  
mutation  
applied

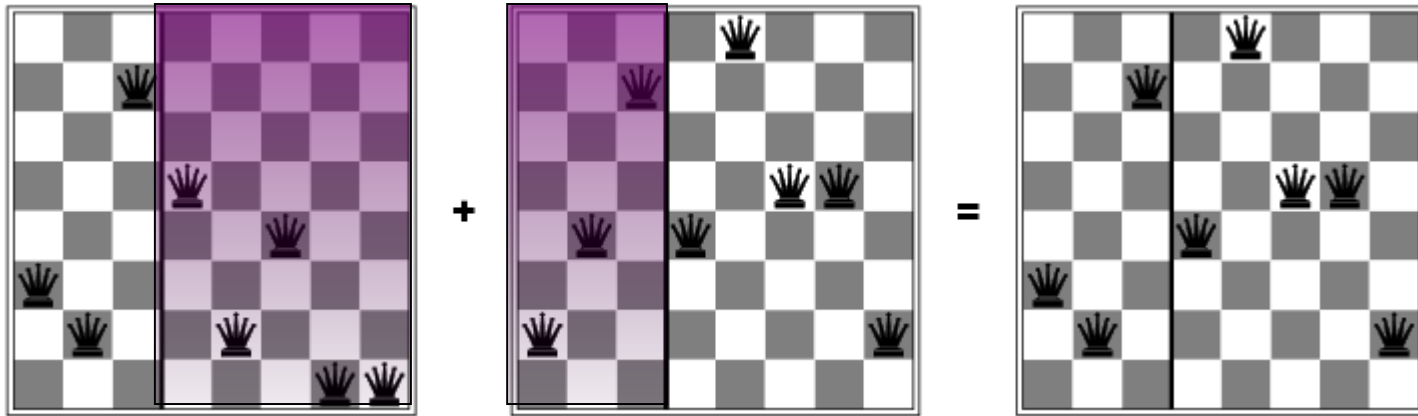
- Fitness Function: Number of non-attacking pairs of queens  
(min = 0, max =  $8 \times 7/2 = 28$ )
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$  etc.



- Fitness function: #non-attacking queen pairs
  - min = 0, max =  $8 \times 7/2 = 28$
- $\sum_i \text{fitness}_i = 24 + 23 + 20 + 11 = 78$
- $P(\text{pick child}_1 \text{ for next gen.}) = \text{fitness}_1 / (\sum_i \text{fitness}_i) = 24/78 = 31\%$
- $P(\text{pick child}_2 \text{ for next gen.}) = \text{fitness}_2 / (\sum_i \text{fitness}_i) =$

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

# Genetic Algorithm



Has the effect of “jumping” to a completely different new part of the search space (quite non-local)



# Comments on Genetic Algorithm

- Genetic algorithm is a variant of “stochastic beam search”
- Genetic Algorithm is easy to apply and the Results can be good on some problems, but bad on other problems
- Positive points
  - Random exploration can find solutions that local search can't
    - (via crossover primarily)
  - Appealing connection to human evolution
    - “neural” networks, and “genetic” algorithms are **metaphors!**
- Negative points
  - Large number of “tunable” parameters
    - Difficult to replicate performance from one problem to another
  - Lack of good empirical studies comparing to simpler

# 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)