Week 5
Local Search Strategies

Dr Ammar Masood

Department of Cyber Security,

Air University Islamabad

Table of Contents

Problem

- In the search problems of Chapter 3 we wanted to find paths through the search space, such as a path from Arad to Bucharest. But sometimes we care only about the final state, not the path to get there.
- An example is the the 8-queens problem about finding a valid final configuration of 8 queens (because if you know the configuration, it is trivial to reconstruct the steps that created it).
- This is also true for many important applications such as integrated-circuit design, factory floor layout, job shop scheduling, automatic programming etc
- In such cases, we can use local search algorithms
- Local search= use single current state and move to neighboring states.
- Advantages:
  - Use very little memory
  - Find often reasonable solutions in large or infinite state spaces.
- Also useful for pure optimization problems.
  - Find best state according to some objective function.

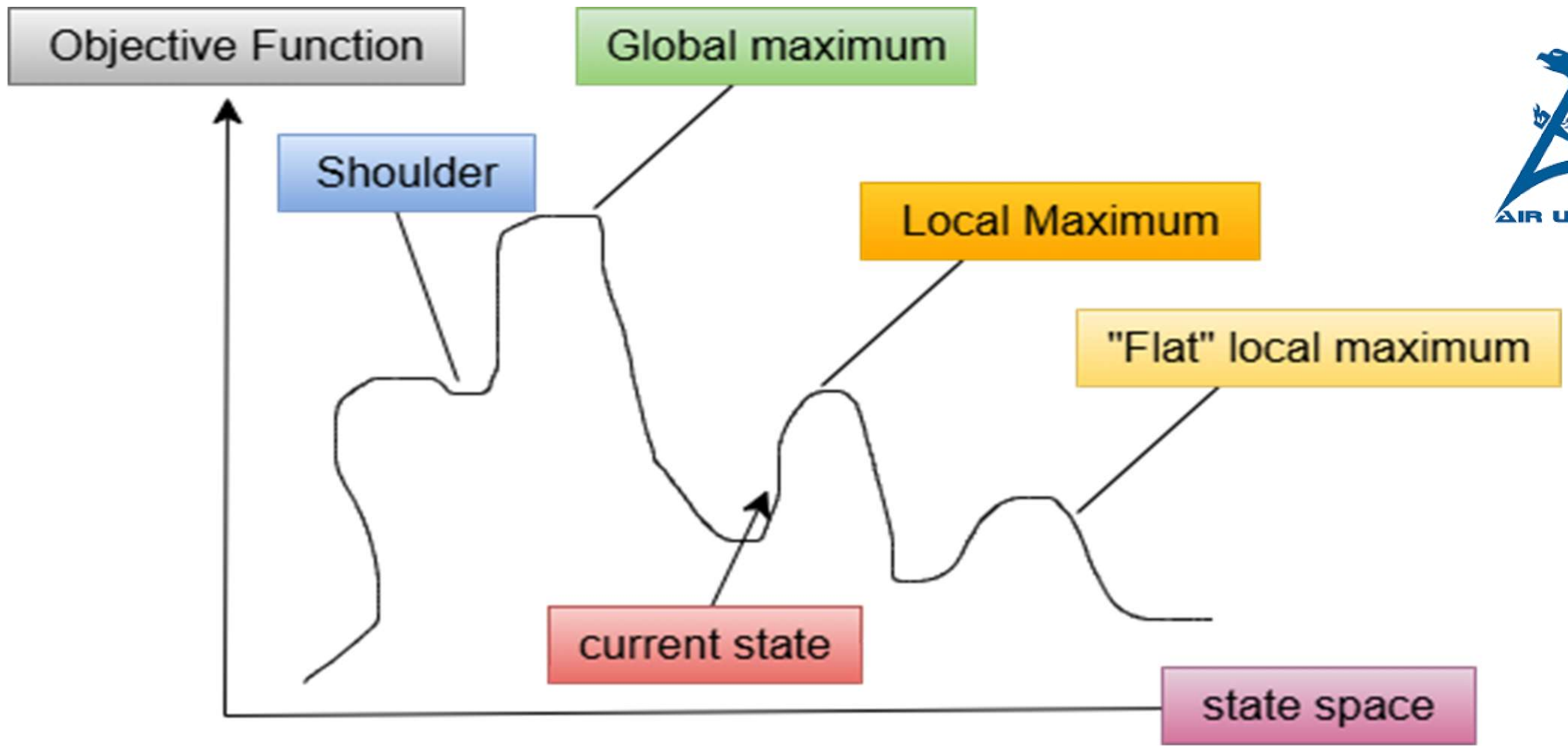
Local Search / Optimization

- Idea is to find the ***best* state**.
- We don't really care how to get to the best state, just that we get there.
- The best state is defined according to an ***objective*** *function*
  - Measures the "fitness" of a state.
- Problem: Find the optimal state
  - The one that maximizes (or minimizes) the objective function.

Local search strategies

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides.

- Local search is used when **state-space is too large** to explore completely.
- It starts from an **initial solution** and iteratively improves it.
- Unlike **uninformed** search, it doesn't maintain a search tree.
- Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function.**



To understand local search, consider the states of a problem laid out in a state-space landscape. Each point (state) in the landscape has an "elevation," defined by the value of the objective function. If elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum—and we call the process hill climbing. If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum—and we call it gradient descent

Advantages

1. They use very little memory.
2. They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable

Hill Climbing

Idea: Always move towards a state with a **higher value** (maximization) or **lower cost** (minimization).

Limitation:

- Gets stuck in **local optima**
- Susceptible to **plateaus and ridges**

Example:

- **Robotics Pathfinding:** A robot moving toward the highest ground to avoid obstacles.
- **Traveling Salesman Problem (TSP):** Finding a shorter route step by step.

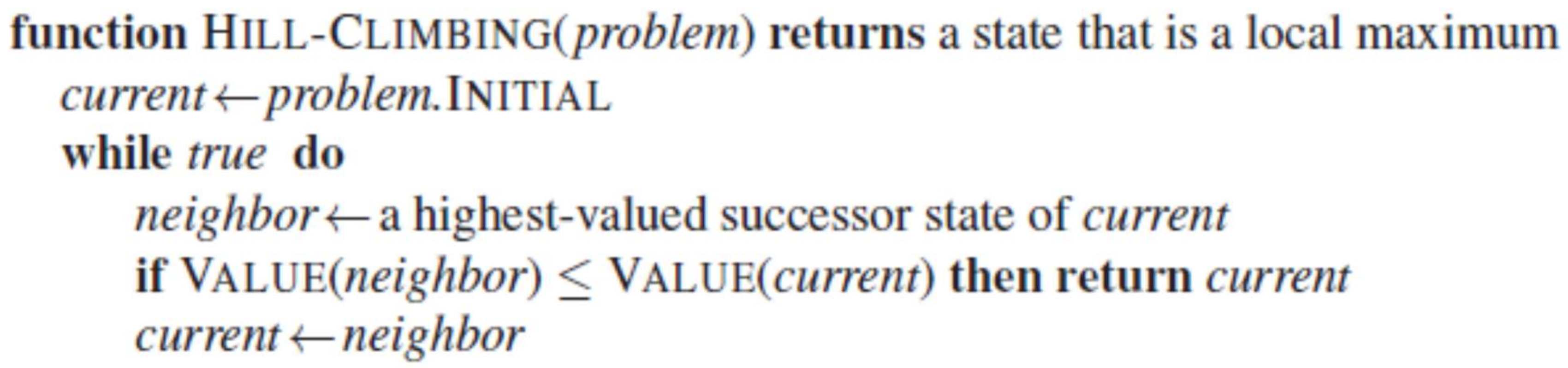
Hill Climbing Search

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Some problem spaces are great for hill climbing and others are terrible.

Working

- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the steepest ascent. It terminates when it reaches a “peak” where no neighbor has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- Note that one way to use hill-climbing search is to use the negative of a heuristic cost function as the objective function; that will climb locally to the state with smallest heuristic distance to the goal

Algorithm

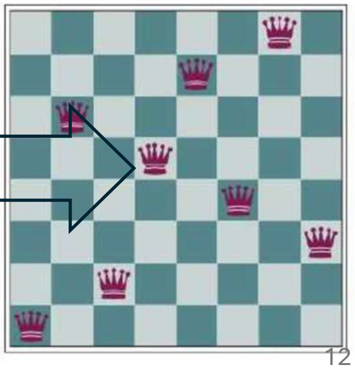


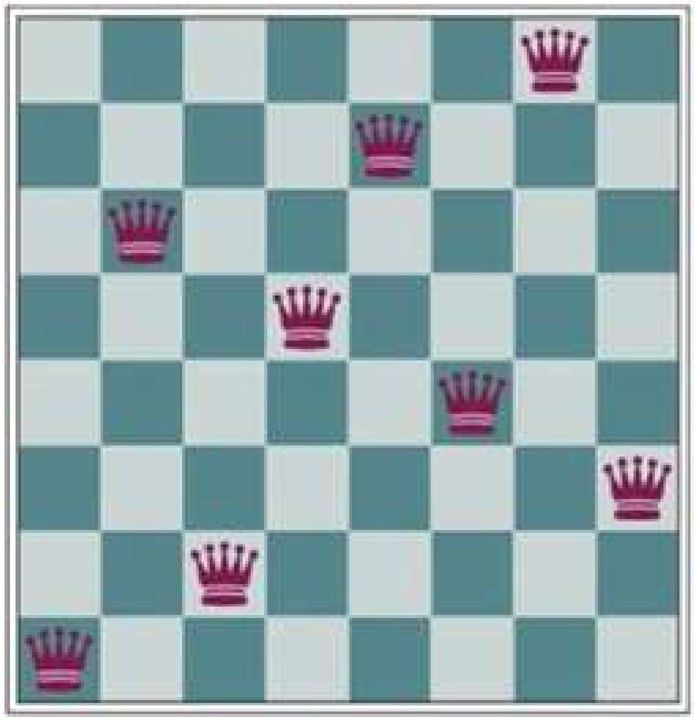
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-valued successor state of current
        if VALUE(neighbor) ≤ VALUE(current) then return current
        current ← neighbor
```

8 queen problem example

- Initial State: 8 queens placed randomly on the board, one per column.
- Successor function: States that obtained by moving one queen to a new location in its column. So, each state has 8×7=56 successors here.
- Heuristic/objective function: The number of pairs of attacking queens. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.)

H(state)=1,
4th and 7th Queen can attack





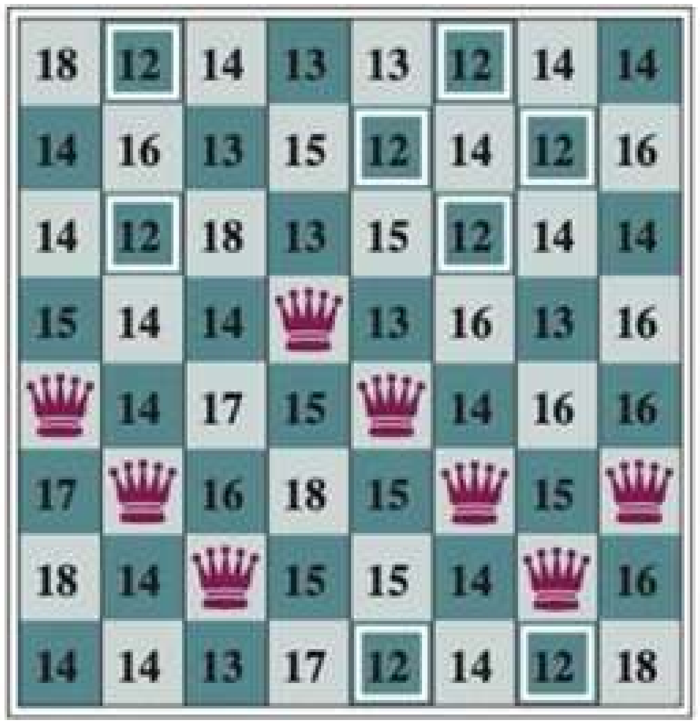


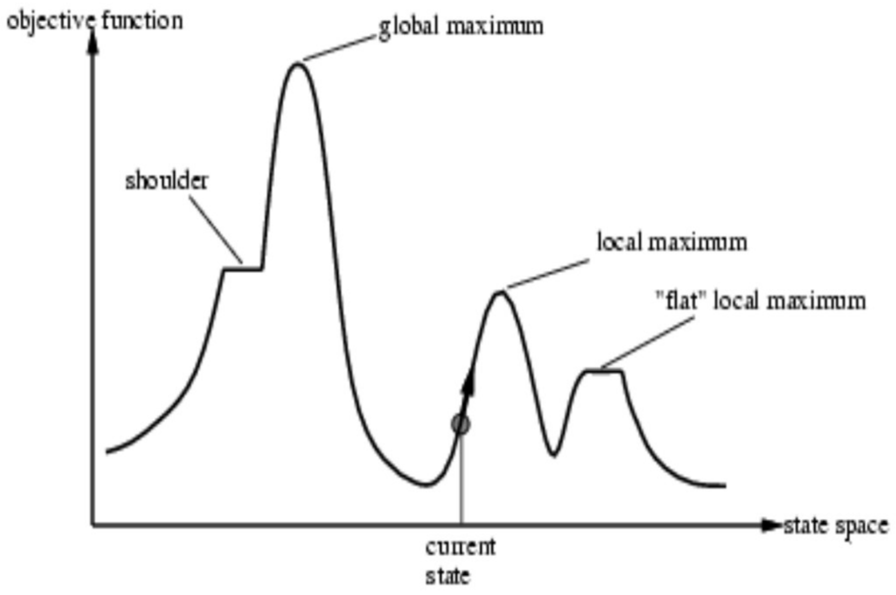
Figure shows a state that has h =17 The figure also shows the h values of all its successors.

The board shows the value of for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with h hill-climbing algorithm will pick one of these. h =12

Drawback of Hill Climbing

- Problems:
  - **Local Maxima:** depending on initial state, can get stuck in local maxima
  - **Plateaus:** the space has a broad flat region that gives the search algorithm no direction (random walk)
  - **Ridges:** flat like a plateau, but with dropoffs to the sides; steps to the North, East, South and West may go down, but a combination of two steps (e.g. N, W) may go up

- Introduce randomness



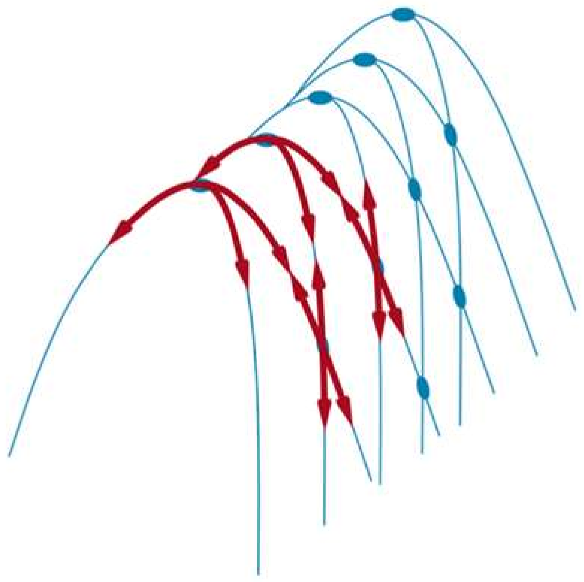
Problem with Hill Climbing

LOCAL MAXIMA: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. More concretely, if the state in queens problem is a local maximum (i.e., a local minimum for the cost); every move of a single queen makes the situation worse.

PLATEAUS: A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search can get lost wandering on the plateau.

Problem cont..

RIDGES: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.



The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

Variants

- **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

Random restart hill climbing

- This variant adopts the adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.
- It is complete with probability 1, because it will eventually generate a goal state as the initial state.
- If p is probability of a search succeeding, then expected number of restarts is 1/p.

Multiple restart hill climbing

Multiple Restarts Hill Climbing is an extension of **Random-Restart Hill Climbing**, where the algorithm runs multiple independent hill climbing searches from different random starting points. The goal is to overcome the issue of getting stuck in **local maxima, plateaus, or ridges**.

Instead of a single climb, the algorithm runs **N times** with different initial states and returns the best solution found across all runs.

Example

Imagine we are climbing a mountain range where some peaks are higher than others.

- If we start climbing randomly, we may reach a local maximum (a peak lower than the highest).
- To improve our chances of finding the **highest peak**, we restart the climb from different random locations **N** times.
- The best peak reached in all runs is our solution.

Comparison

Feature	Hill Climbing	Random Restart Hill Climbing	Multiple Restarts Hill Climbing
Basic Idea	Iteratively moves to the best neighbor	Restarts from a new random point when stuck	Runs multiple independent hill climbs and picks the best
Handling Local Maxima	Gets stuck in local maxima	Escapes local maxima by restarting	Less likely to get stuck due to multiple independent searches
Restart Mechanism	No restarts	Restarts only when stuck	Runs a fixed number **N** of restarts
Efficiency	Fast but can fail on complex landscapes	Better than standard hill climbing	More reliable but computationally expensive
Guaranteed Global Maximum?	No	No but better than HC	More likely to find global maximum
Example Use Case	Simple optimization problems	Problems with occasional local maxima	Complex optimization with many peaks
Computational Cost	Low	Moderate	High (due to multiple runs)

Simulated Annealing

Inspired by: Metal cooling process → **gradually reducing randomness**.

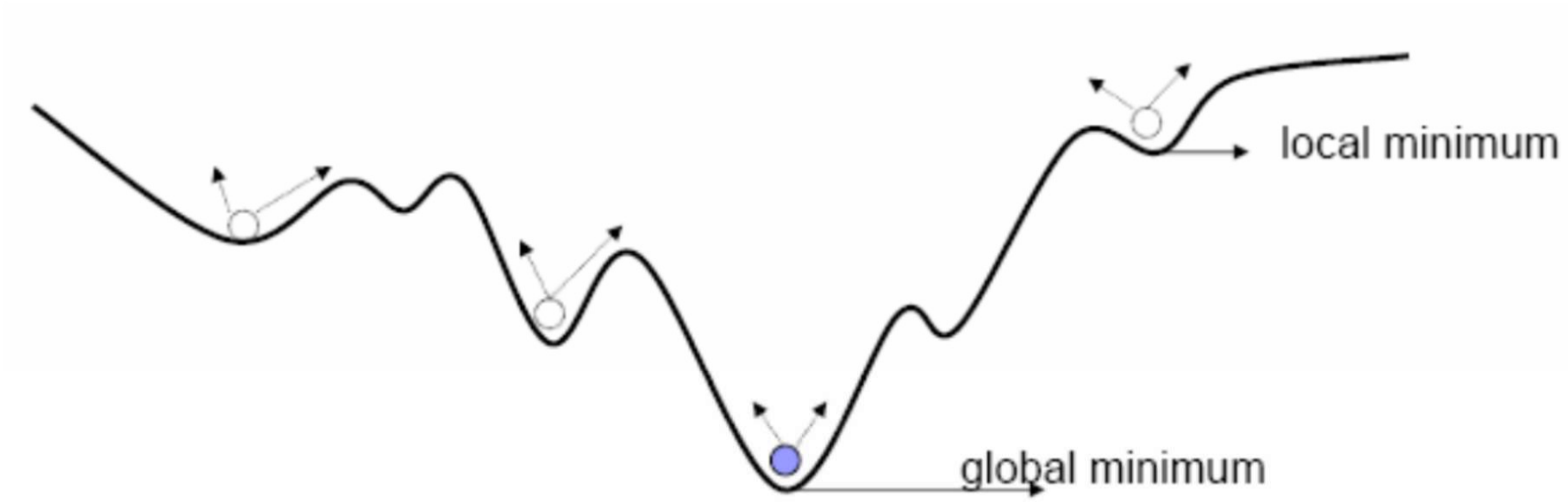
Accepts worse solutions with some probability to avoid local optima.

Uses a cooling schedule (temperature T gradually decreases).

Example:

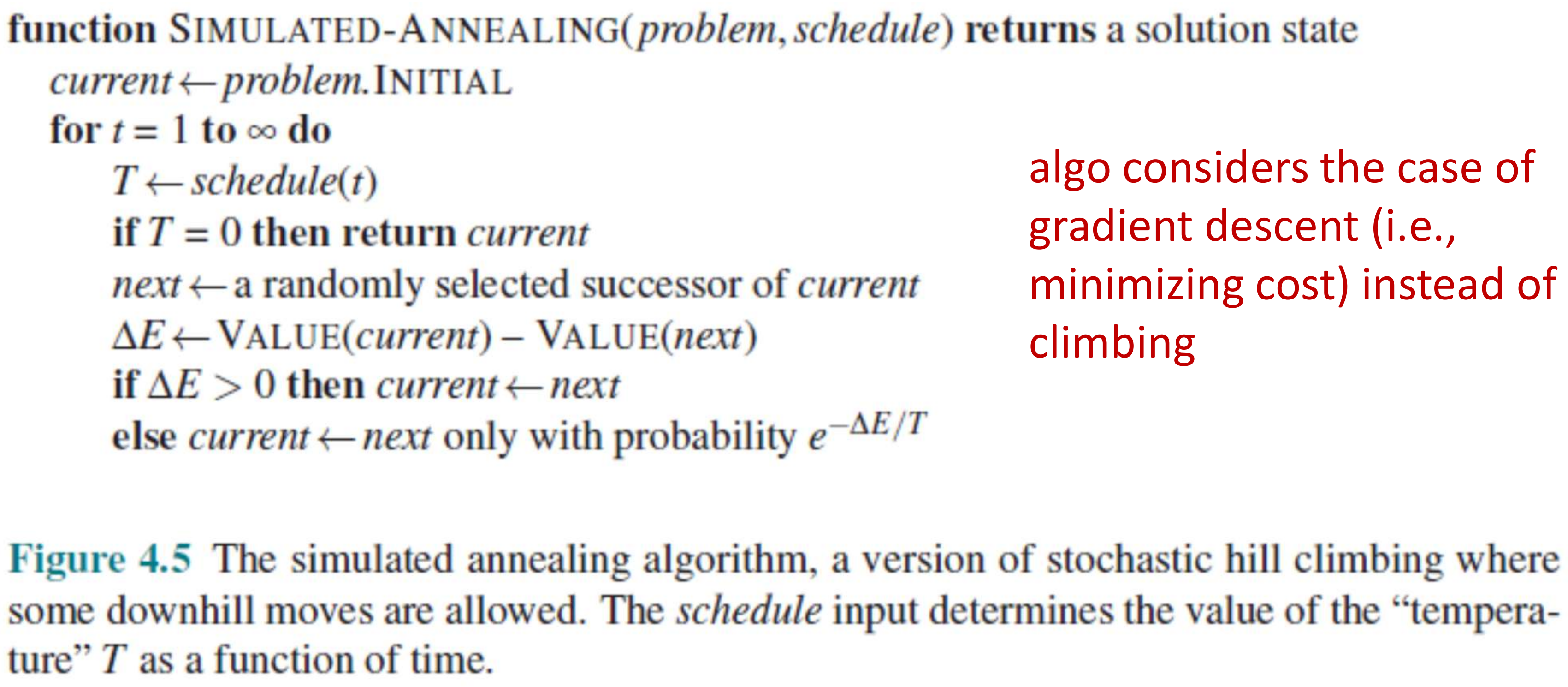
- **Job Scheduling:** Finding the best sequence to reduce delays.
- **Neural Network Training:** Avoids poor convergence to bad local minima.

Considering Minimization Problem



gradually decrease shaking to make sure the ball escape from local minima and fall into the global minimum

Algorithm



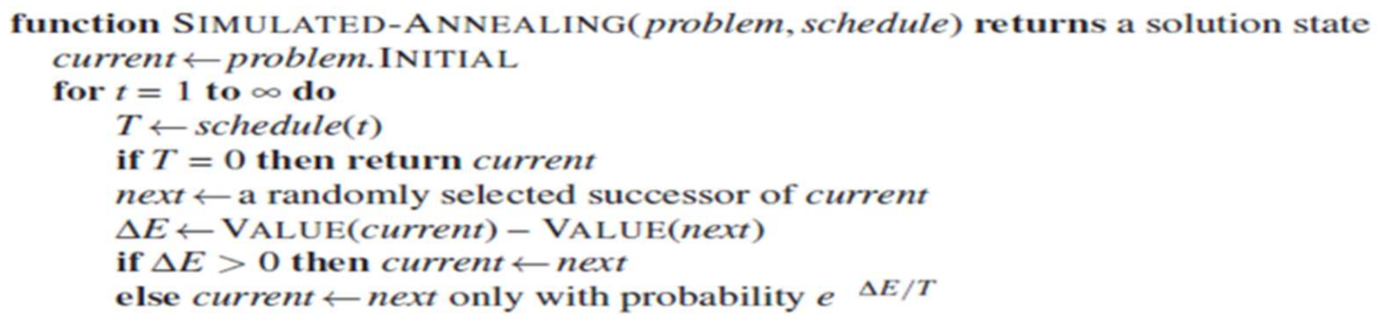
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current ← problem.INITIAL
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE(current) – VALUE(next)
    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

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

algo considers the case of gradient descent (i.e., minimizing cost) instead of hill climbing

Working

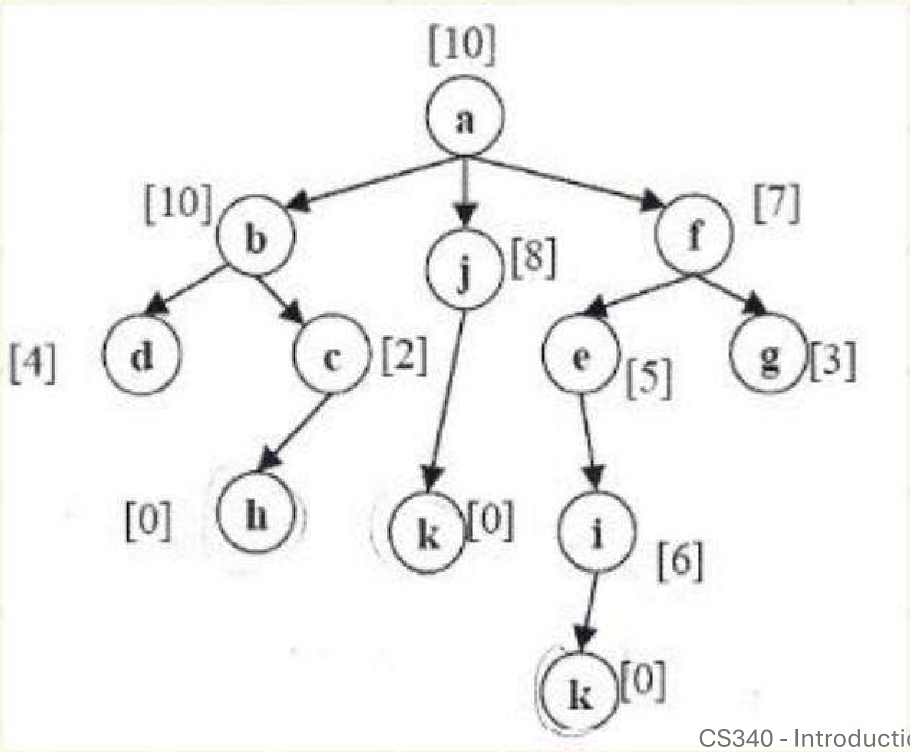
- Idea: Escape local maxima/minima by allowing some “bad” moves
  - But gradually decreasing their frequency
- If the move improves the situation, it is always accepted.
  - Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the “temperature” goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as decreases.



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

Simulated Annealing Example – T=10

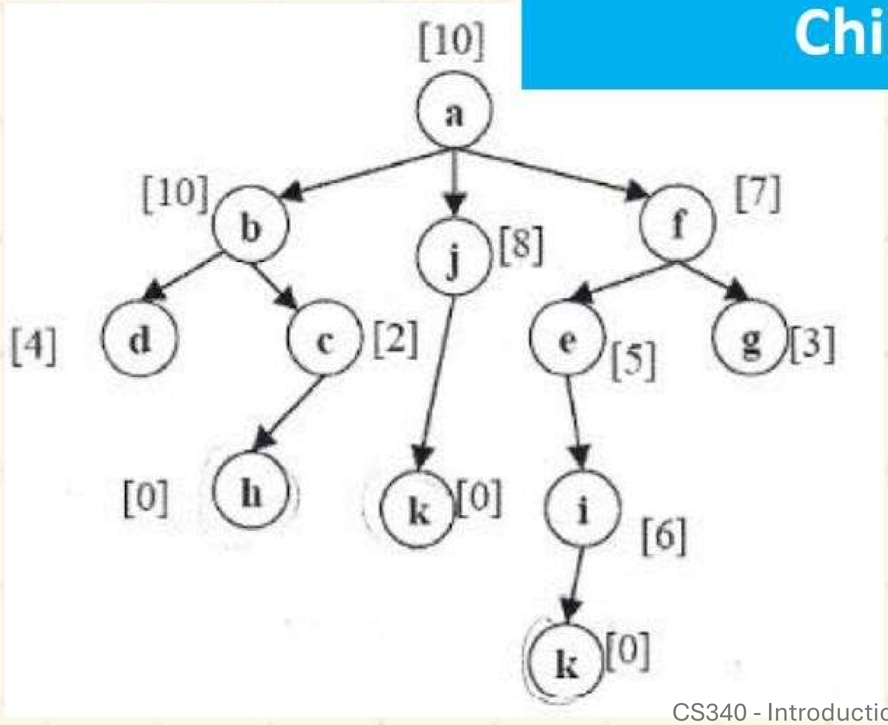
Current	Children
a	



Simulated Annealing

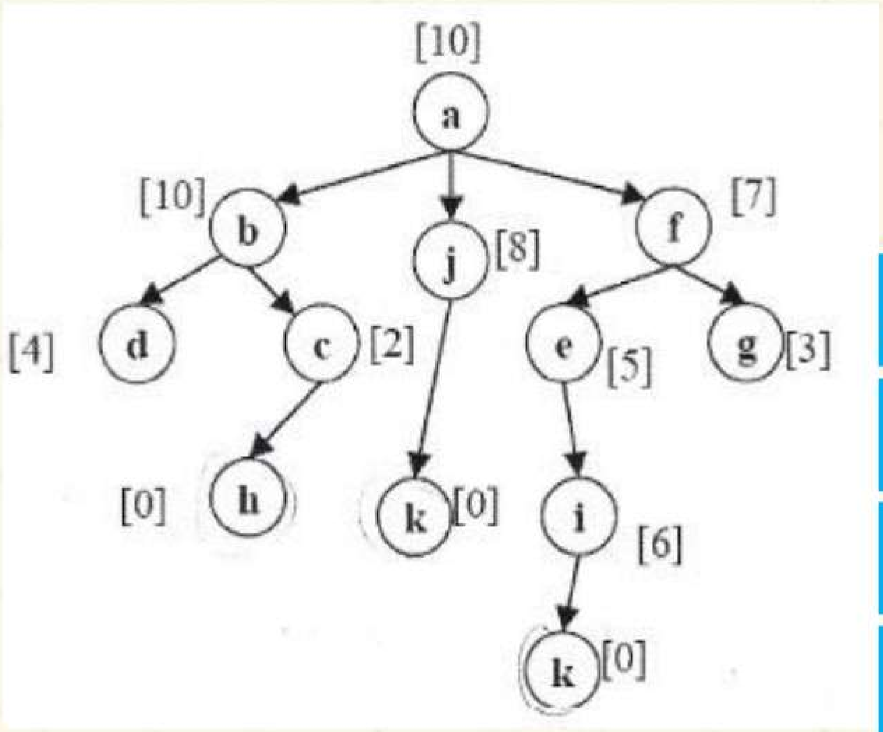
Current	Children
a	---
a	f_7, j_8, b_{10}

Randomly Select a Child



Simulated Annealing

Current	Children
a	---
a	f_7, j_8, b_{10}



Check if next node f_7 is better than current node

$$\Delta E = \text{value}(\text{next}) - \text{value}(\text{current})$$

$$\Delta E = \text{value}(f_7) - \text{value}(a_{10})$$

$$\Delta E = -7 - (-10) = +3$$

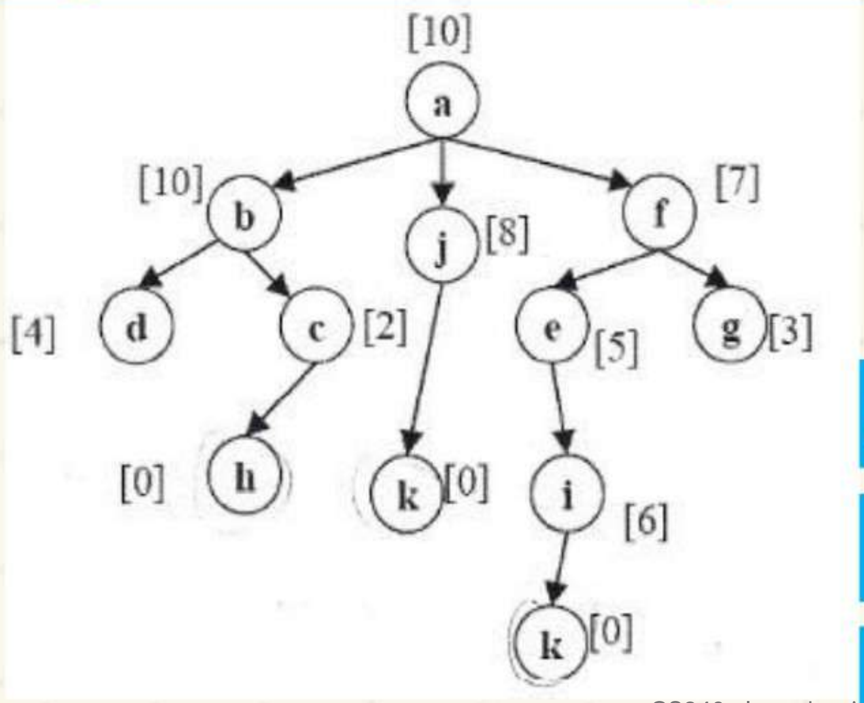
$$\because \Delta E > 0$$

$\therefore f_7$ will be selected with probability 1

Simulated Annealing

$\because \Delta E > 0$

$\therefore e_5$ will be selected with probability 1



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3

Check if next node e_5 is better than current node

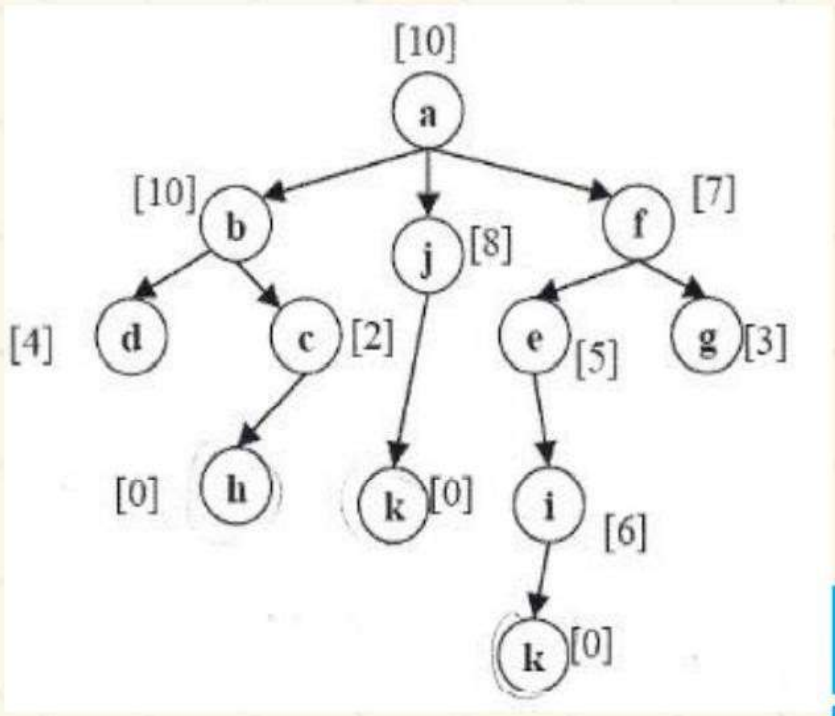
$$\Delta E = \text{value}(\text{next}) - \text{value}(\text{current})$$

$$\Delta E = \text{value}(e_5) - \text{value}(f_7)$$

$$\Delta E = -5 - (-7) = +2$$

Simulated Annealing

$$\Delta E = -6 - (-5) = -1$$



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6

Check if next node i_6 is better than current node

$$\Delta E > 0$$

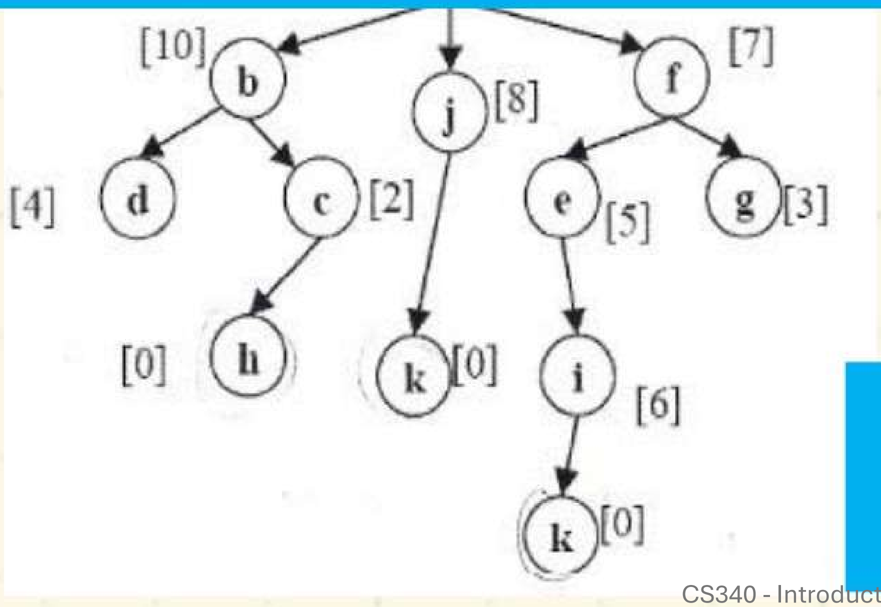
$$\Delta E = \text{value}(\text{next}) - \text{value}(\text{current})$$

$$\Delta E = \text{value}(i_6) - \text{value}(e_5)$$

Simulated Annealing

$\because \Delta E < 0$

$\therefore i_6$ can be selected with probability $p = e^{\frac{\Delta E}{T}}$



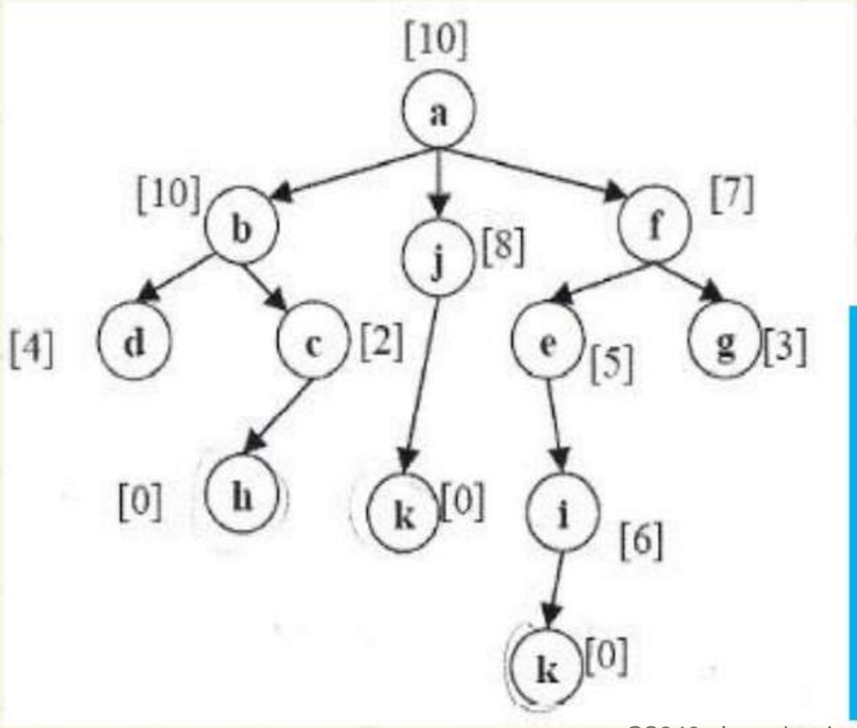
Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6

Check if next node e_5 is better than current node

$$p = e^{\frac{-1}{10}} = e^{\frac{-1}{10}} = .905$$

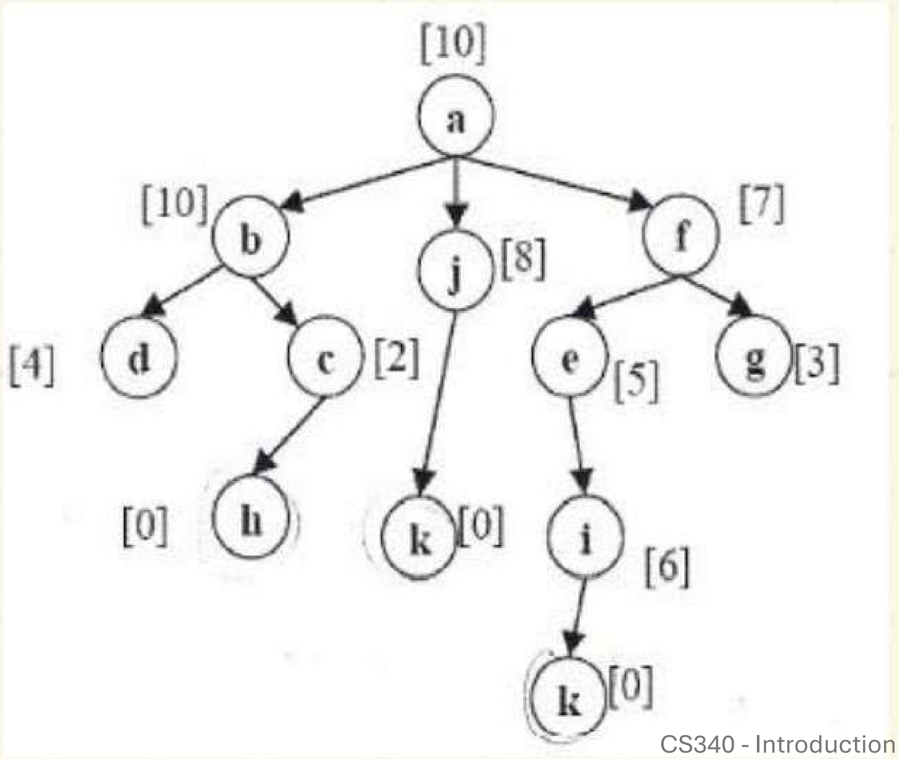
Simulated Annealing

Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6



Because the only child of e_5 is i_6 then it will be selected even if its probability is not 1.

Simulated Annealing

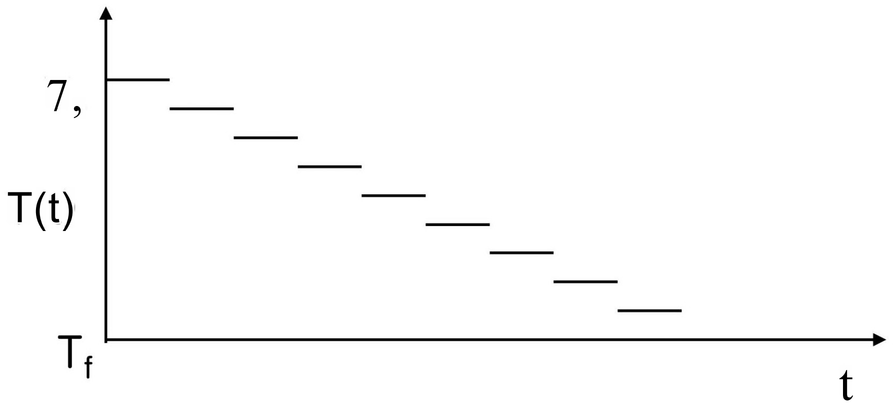


Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6
i	k_0
k	

Simulated Annealing parameters

- Temperature T
  - Used to determine the probability
  - High T : large changes
  - Low T : small changes
- Cooling Schedule
  - Determines rate at which the temperature T is lowered
  - Lowers T slowly enough, the algorithm will find a global optimum
- In the beginning, aggressive for searching alternatives, become conservative when time goes by

Simulated Annealing Cooling Schedule



- if T is reduced too fast, poor quality
- Tt = a T(t-1) where a is in between 0.8 and 0.99

Tips for Simulated Annealing

- To avoid of entrapment in local minima
  - Annealing schedule : by trial and error
    - Choice of initial temperature
    - How many iterations are performed at each temperature
    - How much the temperature is decremented at each step as cooling proceeds
- Difficulties
  - Determination of parameters
  - If cooling is too slow -Too much time to get solution
  - If cooling is too rapid ➔Solution may not be the global optimum

Local Beam Search

- Expands only the **k best nodes** at each level, unlike BFS/DFS.
- **Keeps track of the most promising paths.**
- **Tradeoff:** More efficient than exhaustive search, but **may miss the global best solution.**
- The local beam search algorithm keeps track of **k** states rather than just one. It begins with **k** randomly generated states. At each step, all the successors of all **k** states are generated. If any one is a goal, the algorithm halts.

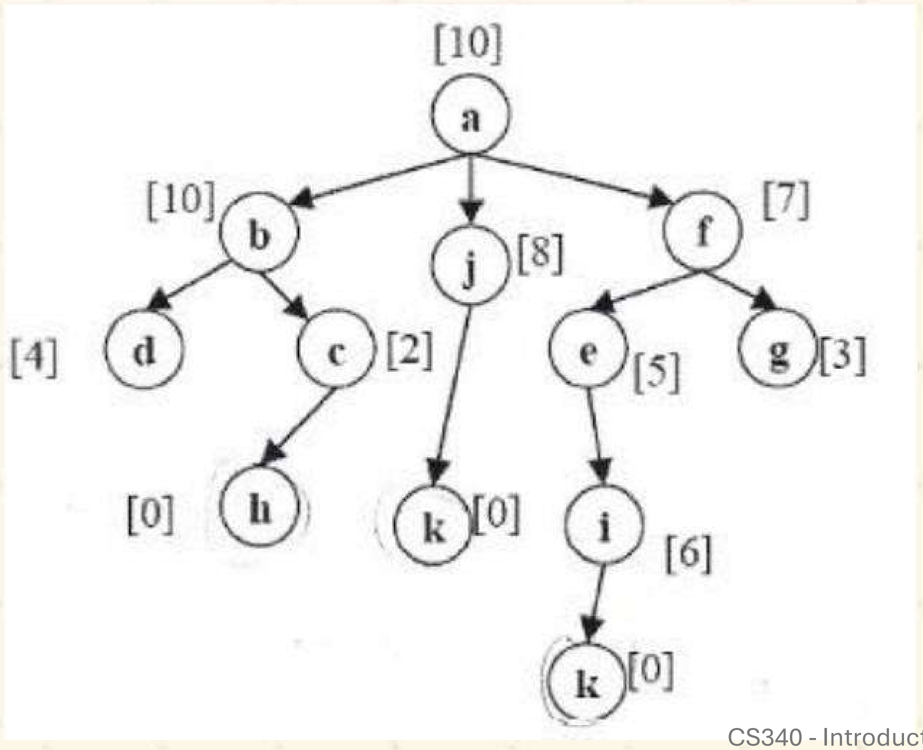
Working cont..

- At first sight, a local beam search with states might seem to be nothing more than running **k** random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different.
- In a random-restart search, each search process runs independently of the others.
- In a local beam search, useful information is passed among the parallel search threads. In effect, the states that generate the best successors say to the others,

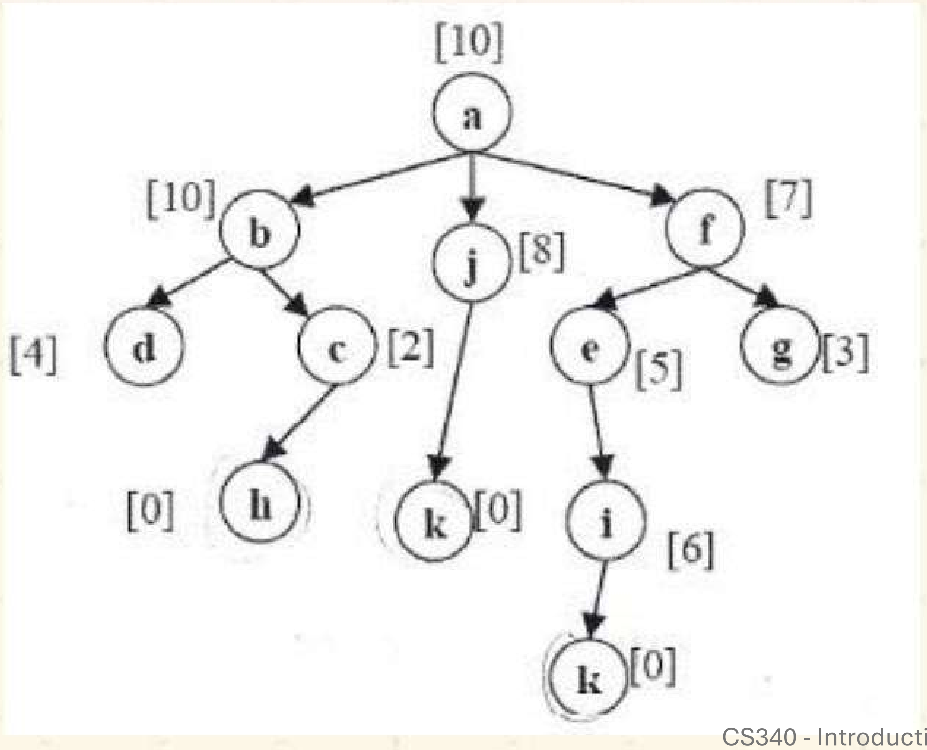
“Come over here, the grass is greener!” The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Beam Search, k=2
Goal – **Node K**

Current	Children
a	---



Beam Search
Goal – **Node K**



Current	Children
a	---
a	f_7, j_8, b_{10}
f j	

Beam Search
Goal – **Node K**

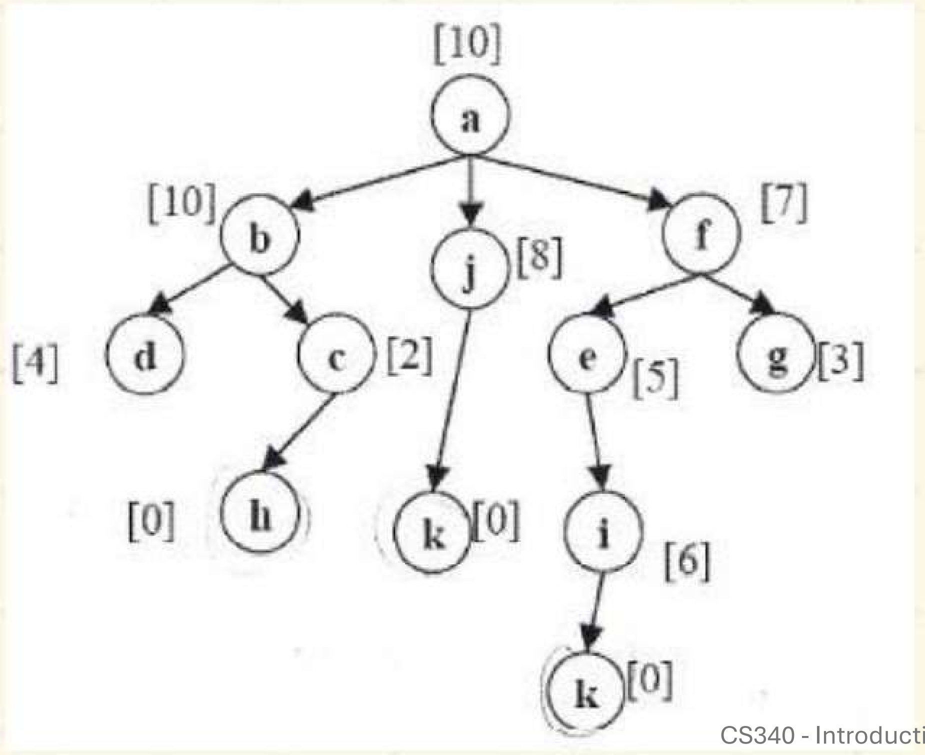
Best k Successors

Current	Children
a	---
a	f_7, j_8, b_{10}

f j

g_3, e_5 k_0

k_0, g_3, e_5



Beam Search
Goal – **Node K**

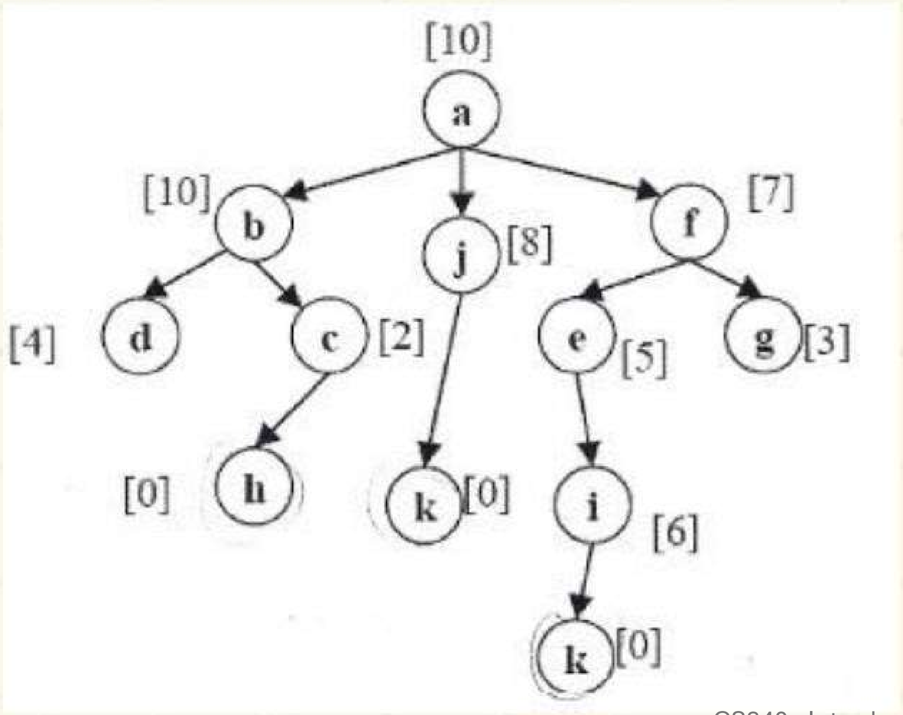
Current	Children
a	---
a	f_7, j_8, b_{10}

f j

g_3, e_5 k_0

k_0, g_3, e_5

k g



Algorithm

```
function LOCAL-BEAM-SEARCH(problem, k) returns a solution state
    start with only one state (initial)
    generate k best successors
        while true do
        generate all successors of the k current states
        select the k best successors based on their evaluation function
        if any of the k states is a goal state then return that state
```

Comparison

Algorithm	**Strengths**	**Weaknesses**
Hill Climbing	Fast, simple	Stuck in local optima
Random Restart	Avoids bad local optima	Computationally expensive
Simulated Annealing	Escapes local minima	Needs a good cooling schedule
Beam Search	Efficient for large problems	May ignore optimal paths

Genetic Algorithms

- Quicker but randomized searching for an optimal parameter vector
- Operations
  - Crossover (2 parents -> 2 children)
  - Mutation (one bit)
- Basic structure
  - Create population
  - Perform crossover & mutation (on fittest)
  - Keep only fittest children

Genetic Algorithms

- Variant of stochastic beam search
- Successor states are generated by combining two parent states
  - Hopefully improves diversity
- Start with **k states**, the *population*
- Each state, or *individual*, represented as a string over a finite alphabet (e.g. DNA)
- Each state is rated by a ***fitness*** *function*
- Select parents for reproduction using the fitness function

Genetic Algorithms

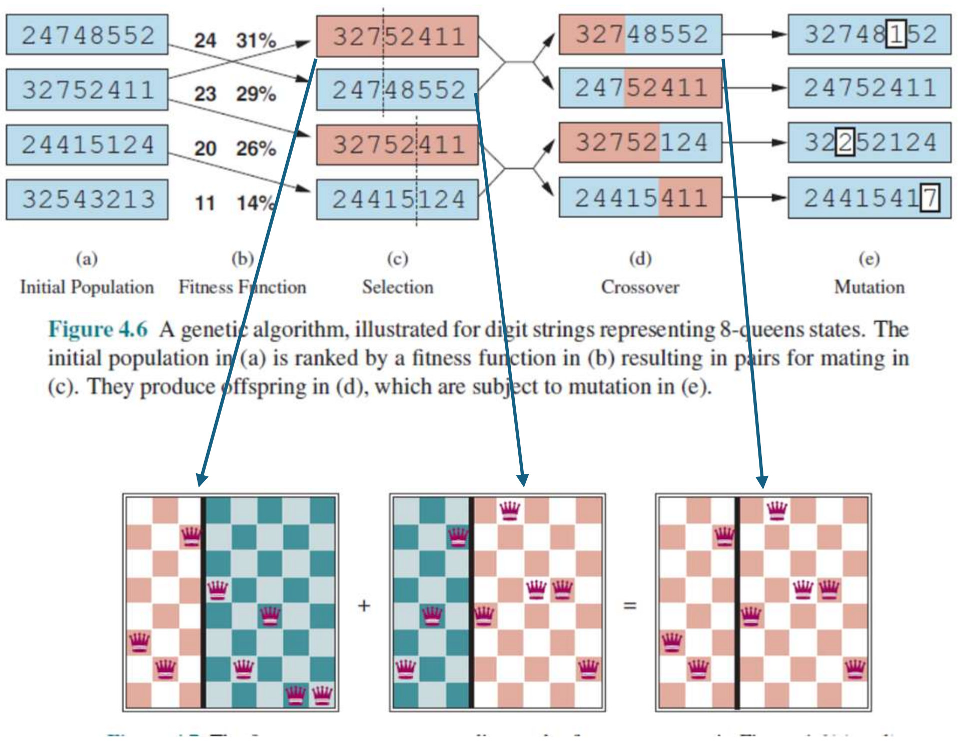
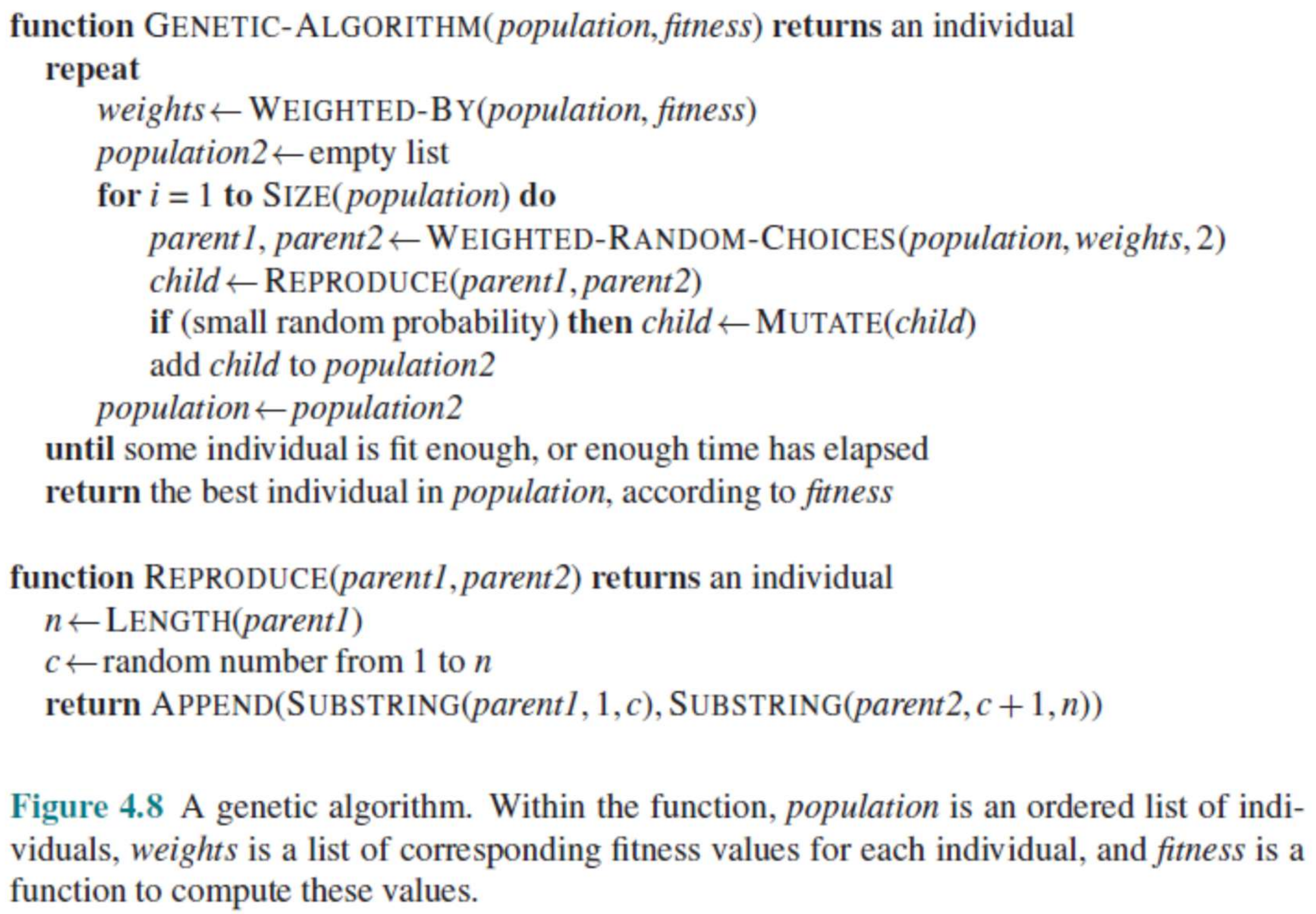


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- Children carry parts of their parents' data
- Only "good" parents can reproduce
  - Children are at least as "good" as parents?
    - No, but "worse" children don't last long



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

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

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Genetic Algorithms

- Representation
  - Children (after crossover) should be similar to parent, not random
  - Binary representation of numbers isn’t good - what happens when you crossover in the middle of a number?
  - Need “reasonable” breakpoints for crossover (e.g. between R, xcenter and ycenter but not within them)
- “Cover”
  - Population should be large enough to “cover” the range of possibilities
  - Information shouldn’t be lost too soon
  - Mutation helps with this issue

Experimenting With GAs

- Be sure you have a reasonable “goodness” criterion
- Choose a good representation (including methods for crossover and mutation)
- Generate a sufficiently random, large enough population
- Run the algorithm “long enough”
- Find the “winners” among the population
- Variations: multiple populations, keeping vs. not keeping parents, “immigration / emigration”, mutation rate, etc.

Gradient descent

Gradient Descent is an optimization algorithm used to minimize the cost function of a model by iteratively adjusting the model parameters to reduce the difference between predicted and actual values, thereby improving the model's performance.

Working

Initialization: Start with an initial set of parameters (weights), often assigned randomly.

Compute Gradient: Calculate the gradient (partial derivatives) of the cost function with respect to each parameter. This gradient indicates the direction and rate of the steepest increase in the cost function.

Update Parameters: Adjust the parameters in the opposite direction of the gradient by a factor proportional to the learning rate (α):

$$\theta = \theta - \alpha \cdot \nabla J(\theta)$$

Where:
θ represents the parameters
α is the learning rate
$\nabla J(\theta)$ is the gradient of the cost function.

Iterate: Repeat steps 2 and 3 until convergence, i.e., when the change in the cost function is below a predefined threshold or after a set number of iterations.

Functions Involved

Cost Function (J): Quantifies the error of the model.

Gradient (∇J): Provides the direction and rate of the steepest increase in the cost function.

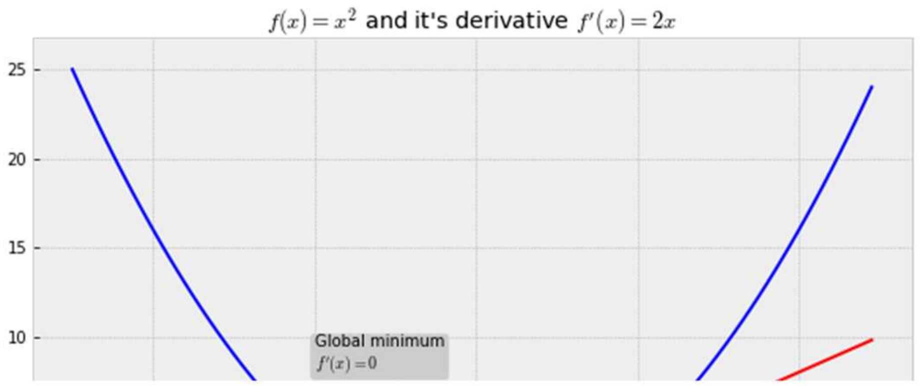
Update Rule: Adjusts the parameters in the opposite direction of the gradient scaled by the learning rate.

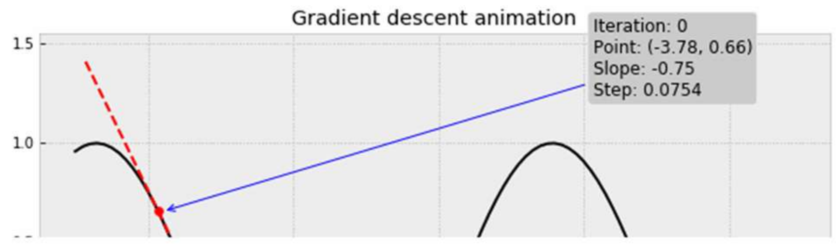
Algorithm

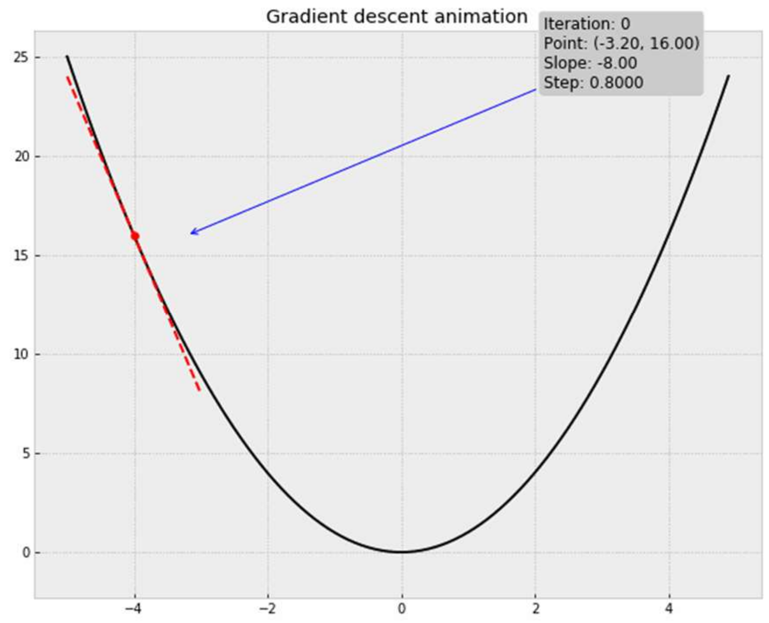
Initialize parameters θ randomly or set them to zero.

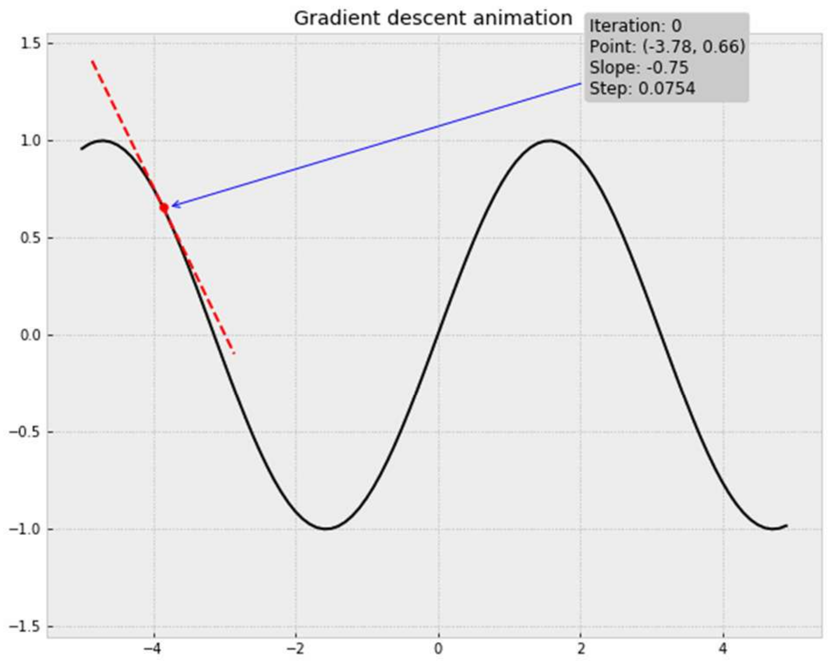
Repeat until convergence (or for a fixed number of iterations):

a. Compute the gradient of the cost function: $\nabla J(\theta)=dJ(\theta)/d\theta$
b. Update parameters using the gradient and learning rate: $\theta=\theta-\alpha\cdot\nabla J(\theta)$
c. Check stopping criteria (e.g., small change in cost function or reaching max iterations).


$f(x) = x^2$ and it's derivative $f'(x) = 2x$
Global minimum
$f'(x) = 0$


Gradient descent animation
Iteration: 0
Point: (-3.78, 0.66)
Slope: -0.75
Step: 0.0754


Gradient descent animation
Iteration: 0
Point: (-3.20, 16.00)
Slope: -8.00
Step: 0.8000


Gradient descent animation
Iteration: 0
Point: (-3.78, 0.66)
Slope: -0.75
Step: 0.0754