Beyond Classical Search

Learning Goals

- Local search algorithms
  - Hill-climbing search
  - Simulated annealing (SA) search
  - Local beam search
  - Genetic algorithms (GA)
- Searching with nondeterministic actions
- Searching with partial observations
- Online search agents and unknown environments

Local search algorithm and optimization problem

Local search algorithm and optimization problem

- **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move **only to neighbors** of that node.
- Typically, the **paths** followed by the search are **not retained**.
- Not systematic
- Key advantages:
  - they use very little memory—usually a constant amount;
  - they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Local search algorithms

- In many **optimization problems**, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying **constraints**,

  e.g.,

  (1) find optimal configuration (e.g., TSP), or,

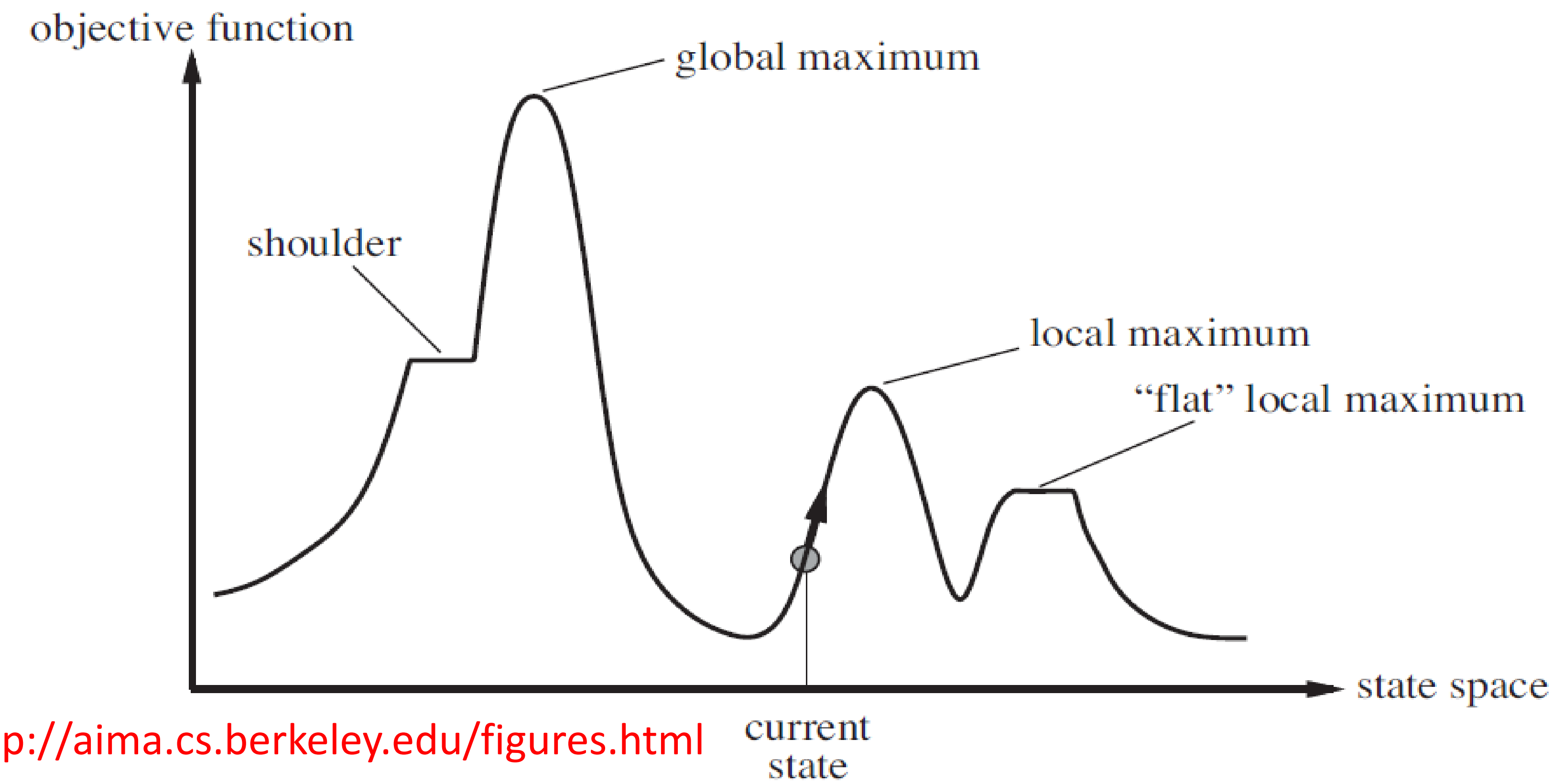
  (2) find configuration satisfying constraints (n-queens)

- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Hill-climbing search

Hill-climbing search

- **state-space landscape**
  - both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function)
  - **Global minimum/maximum**



http://aima.cs.berkeley.edu/figures.html

Hill-climbing search

- **steepest-ascent** version/does not maintain a search tree
- It is simply a loop that continually moves in the direction of increasing value—that is, uphill.
- “Like climbing Everest in thick fog with amnesia (健忘症)“
- Problem: depending on initial state, can get stuck in local maxima

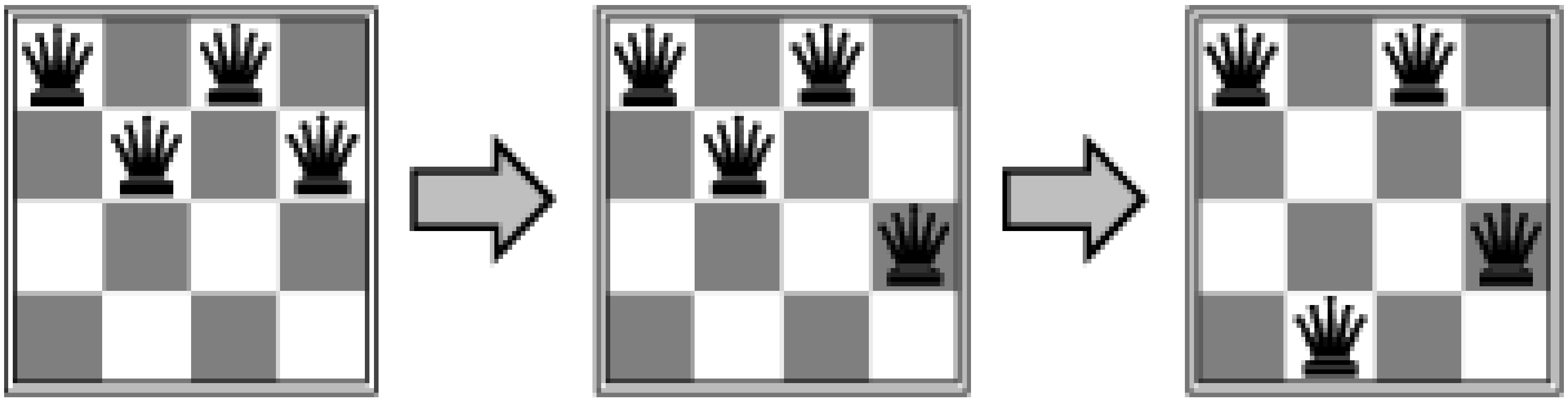
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current $\leftarrow$ MAKE-NODE(*problem*.INITIAL-STATE)
loop do
  neighbor $\leftarrow$ a highest-valued successor of *current*
  if neighbor.VALUE $\leq$ current.VALUE **then return** *current*.STATE
  current $\leftarrow$ *neighbor*

http://aima.cs.berkeley.edu/figures.html

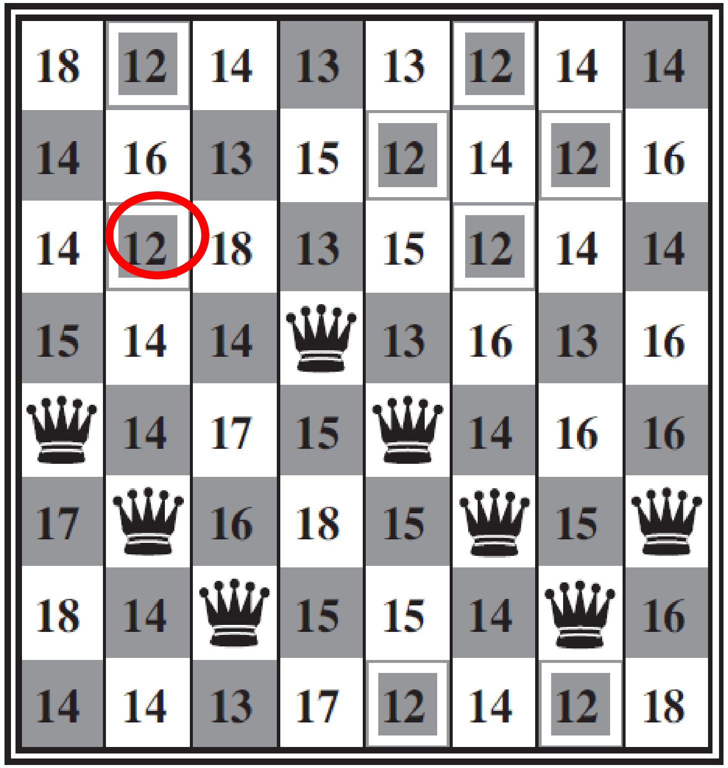
Example: *n*-queens

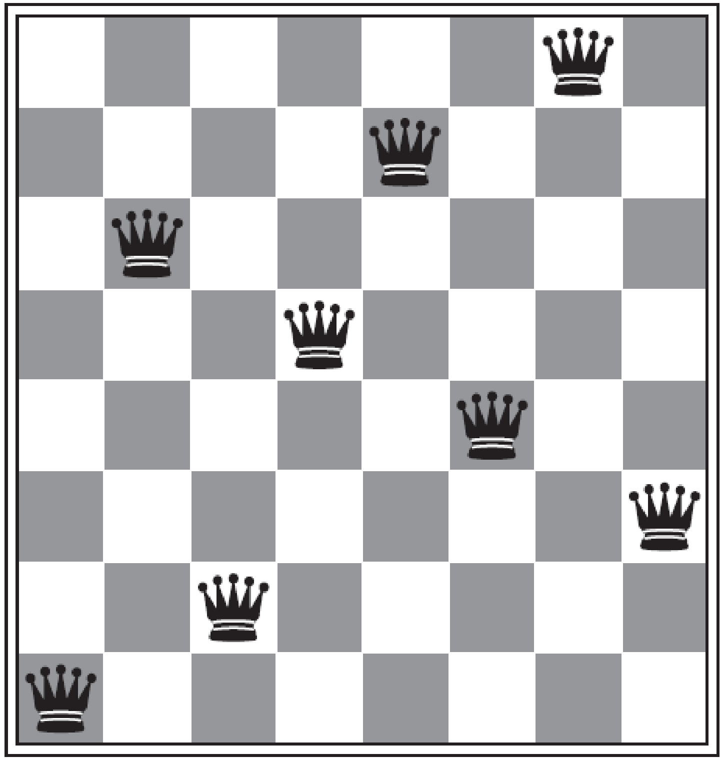
- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.
- Complete-state formulation



Heuristic function h for 8-queen problem

***h* = number of pairs of queens that are attacking each other**



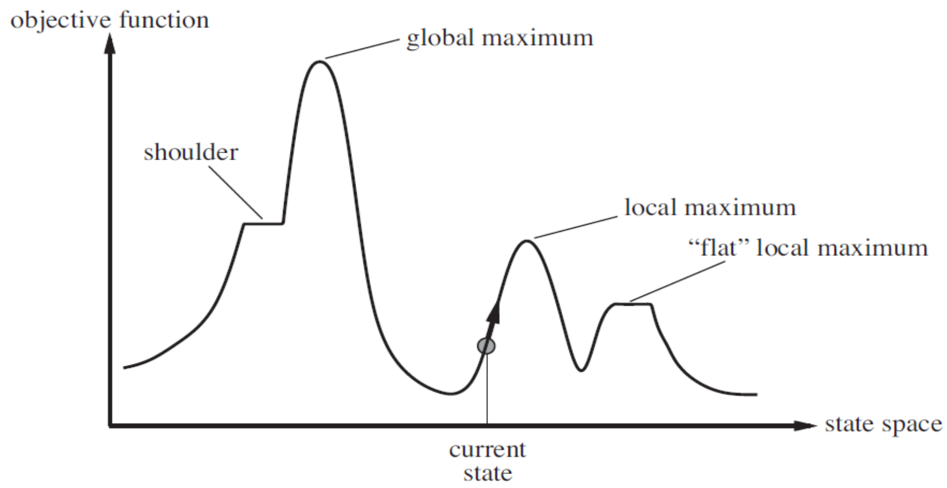


http://aima.cs.berkeley.edu/figures.html

H =1

Hill Climbing Often Gets Stuck

- Hill climbing is sometimes called **greedy local search** because it grabs a **good** neighbor state without thinking ahead about where to go next.
- Hill climbing often gets stuck
  - **Local maxima :** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  - **Ridges (山脊)**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
  - **Plateaux (高原)**: 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.
- 86% get stuck

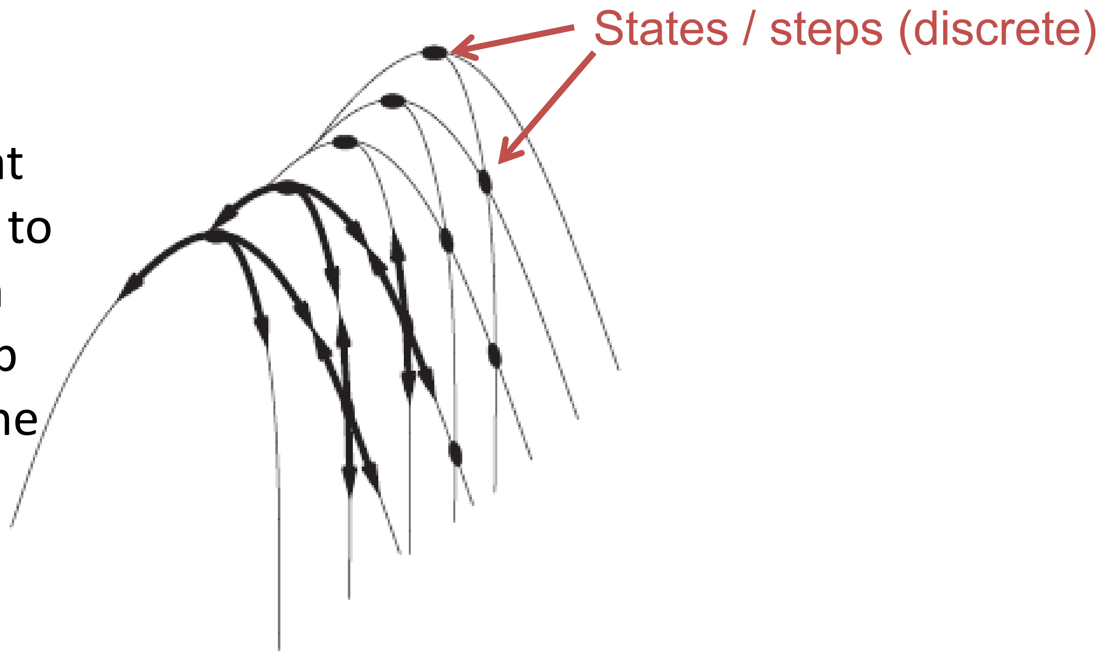


Hill-climbing difficulties

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.

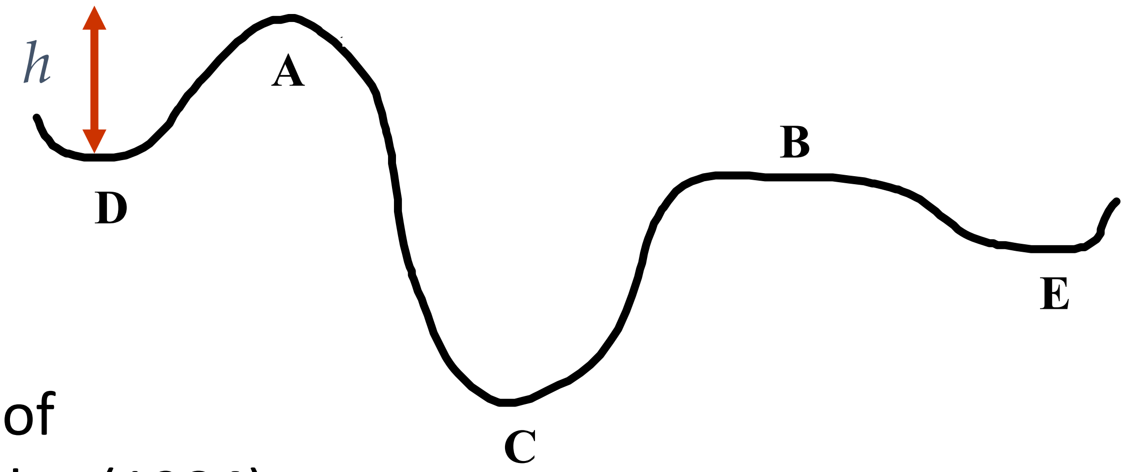


Modify Hill-Climbing

- Sideway move for shoulder/may cause infinite loop=>94% success
- **Stochastic hill climbing**
  - chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- **First-choice hill climbing**
  - by generating successors randomly until one is generated that is better than the current state.
- **Random-restart hill climbing**
  - “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.

Simulated annealing (SA)

Boltzmann machines



- The Boltzmann Machine of
Hinton, Sejnowski, and Ackley (1984)
uses simulated annealing to escape local minima.
- To motivate their solution, consider how one might get a **ball-bearing traveling** along the curve to "probably end up" in the deepest minimum.
- The idea is to shake the box "about h hard" — then the ball is more likely to go from D to C than from C to D. So, on average, the ball should end up in C's valley.

Simulated annealing

- From current state, pick a **random** successor state;
- If it has better value than current state, then "accept the transition," that is, use successor state as current state;
- Otherwise, do not give up, but instead flip a coin and <u>accept the transition with a given probability (that is lower as the successor is worse</u>).
- So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.

Simulated annealing algorithm

- Idea: Escape local extrema by allowing "bad moves," but gradually decrease their size and frequency.

function SIMULATED-ANNEALING($problem$, $schedule$) **returns** a solution state
  inputs: $problem$, a problem
    $schedule$, a mapping from time to "temperature"

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

Note: goal here is to maximize E.

Cooling achedule

http://aima.cs.berkeley.edu/figures.html

simulated annealing: limit cases

- Boltzmann distribution: accept "bad move" with $\Delta E<0$ (goal is to maximize E) with probability $P(\Delta E) = \exp(\Delta E/T)$
- If T is large: $\Delta E < 0$

  $\Delta E/T < 0$ and very small

  $\exp(\Delta E/T)$ close to 1

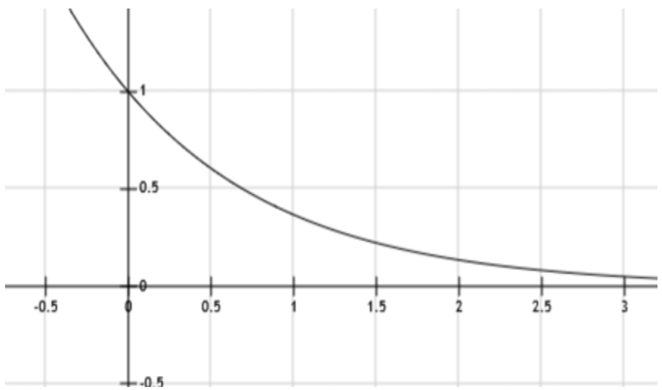
  accept bad move with high probability

- If T is near 0: $\Delta E < 0$

  $\Delta E/T < 0$ and very large

  $\exp(\Delta E/T)$ close to 0

  accept bad move with low probability



F(x)=exp(-x)

Code example of SA

- Simulated annealing algorithm
- Find the global minimum of the function
- $x \mapsto x^2$ on $[-10,10]$
- https://perso.crans.org/besson/publis/notebooks/Simulated_annealing_in_Python.html
- Please reference code example SA.py

Local Beam Search (局部剪枝搜尋)

- 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.
- *In a local beam search, useful information is passed among the* ***parallel search threads****.*
- The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.
- Drawback: the k states tend to regroup very quickly in the same region → lack of diversity.

Stochastic Beam Search

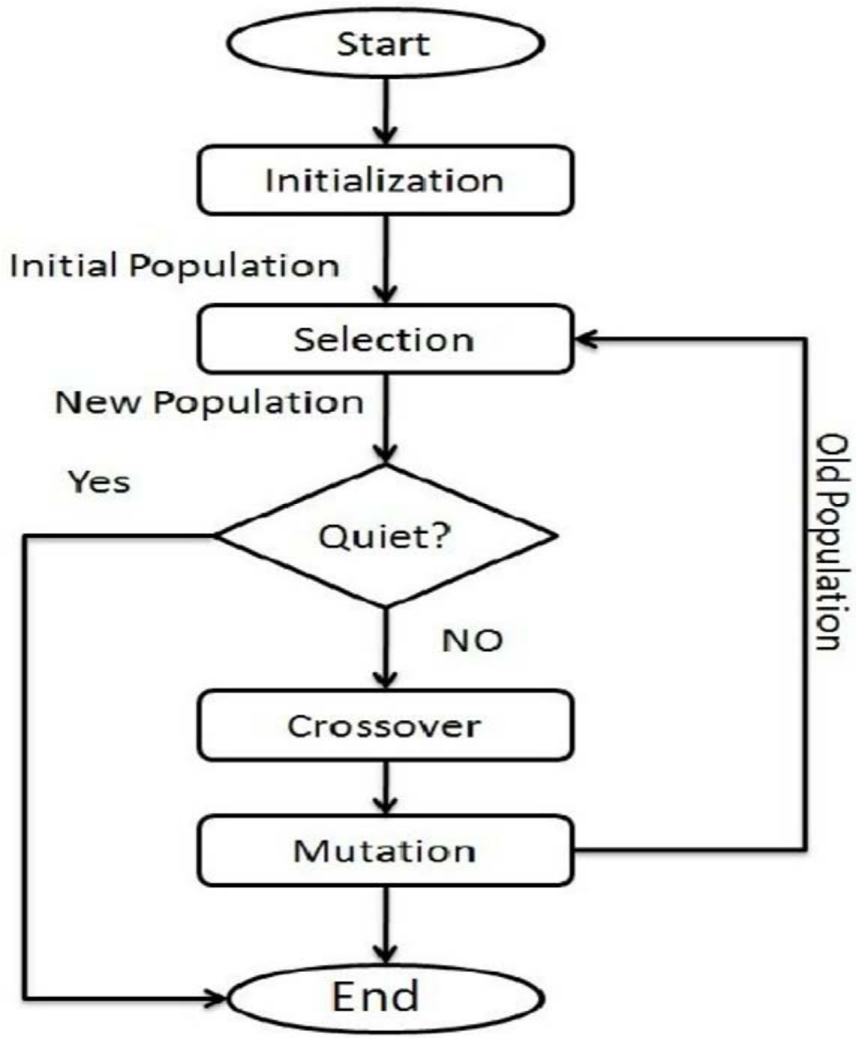
- Chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.
- Bears some resemblance to the process of **natural selection**, whereby the "successors" (**offspring**) of a "state" (**organism**) populate the next generation according to its "value" (**fitness**) -> **genetic algorithm (GA).**

Genetic Algorithm (GA)

Genetic Algorithm (GA)

- Successor states are generated by combining *two* parent states rather than by modifying a single state.
- State = a string over a finite alphabet (an individual) /Encoding
  - 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

Flowchart of GA



```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

http://aima.cs.berkeley.edu/figures.html

Applying GA to train a Machine Learning model

Data	x_1	x_2	x_3	x_4	x_5	x_6	y
	4	-2	7	5	11	1	44.1

44.1= $y = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$

- Goal is to find the set of parameters (w_1:w_6) that maps the following input to its output.

$$y' = 4w_1 - 2w_2 + 7w_3 + 5w_4 + 11w_5 + w_6$$

Solution 1	w_1	w_2	w_3	w_4	w_5	w_6
	2.4	0.7	8	-2	5	1.1

$$y' = 4w_1 - 2w_2 + 7w_3 + 5w_4 + 11w_5 + w_6$$

$$y' = 110.3$$

$$error = |y - y'|$$
$$error = |44.1 - 110.3|$$
$$error = 66.2$$

What are the Genes?

- Gene is anything that is able to enhance the results when changed.
- By exploring the following model, the 6 weights are able to enhance the results. Thus each weight will represent a gene in GA.

$$y = w_1x_1+w_2x_2+w_3x_3+w_4x_4+w_5x_5+w_6x_6$$

Gene 0	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5
w_1	w_2	w_3	w_4	w_5	w_6

Solution 2

w_1	w_2	w_3	w_4	w_5	w_6
-0.4	2.7	5	-1	7	0.1

$$y' = 100.1$$

Absolute Error

$$error = |y - y'|$$
$$error = |44.1 - 100.1|$$
$$error = 56$$

Solution 3

w_1	w_2	w_3	w_4	w_5	w_6
-1	2	2	-3	2	0.9

$$y' = 13.9$$

Absolute Error

$$error = |y - y'|$$
$$error = |44.1 - 13.9|$$
$$error = 30.2$$

Initial Population of Solutions (Generation 0)

2.4	0.7	8	-2	5	1.1
-0.4	2.7	5	-1	7	0.1
-1	2	2	-3	2	0.9
4	7	12	6.1	1.4	-4
3.1	4	0	2.4	4.8	0
-2	3	-7	6	3	3

Population Size = 6

Chromosome

Gene

						y'	$F(C)$
2.4	0.7	8	-2	5	1.1	110.3	0.015
-0.4	2.7	5	-1	7	0.1	100.1	0.018
-1	2	2	-3	2	0.9	13.9	0.033
4	7	12	6.1	1.4	-4	127.9	0.012
3.1	4	0	2.4	4.8	0	69.2	0.0398
-2	3	-7	6	3	3	3	0.024

Survival of the Fittest

↓

Fitness Function

↓

Fitness Value

↓

The Higher the Value, the Better the Solution

$$F(c) = \frac{1}{error} = \frac{1}{|y - y'|}$$

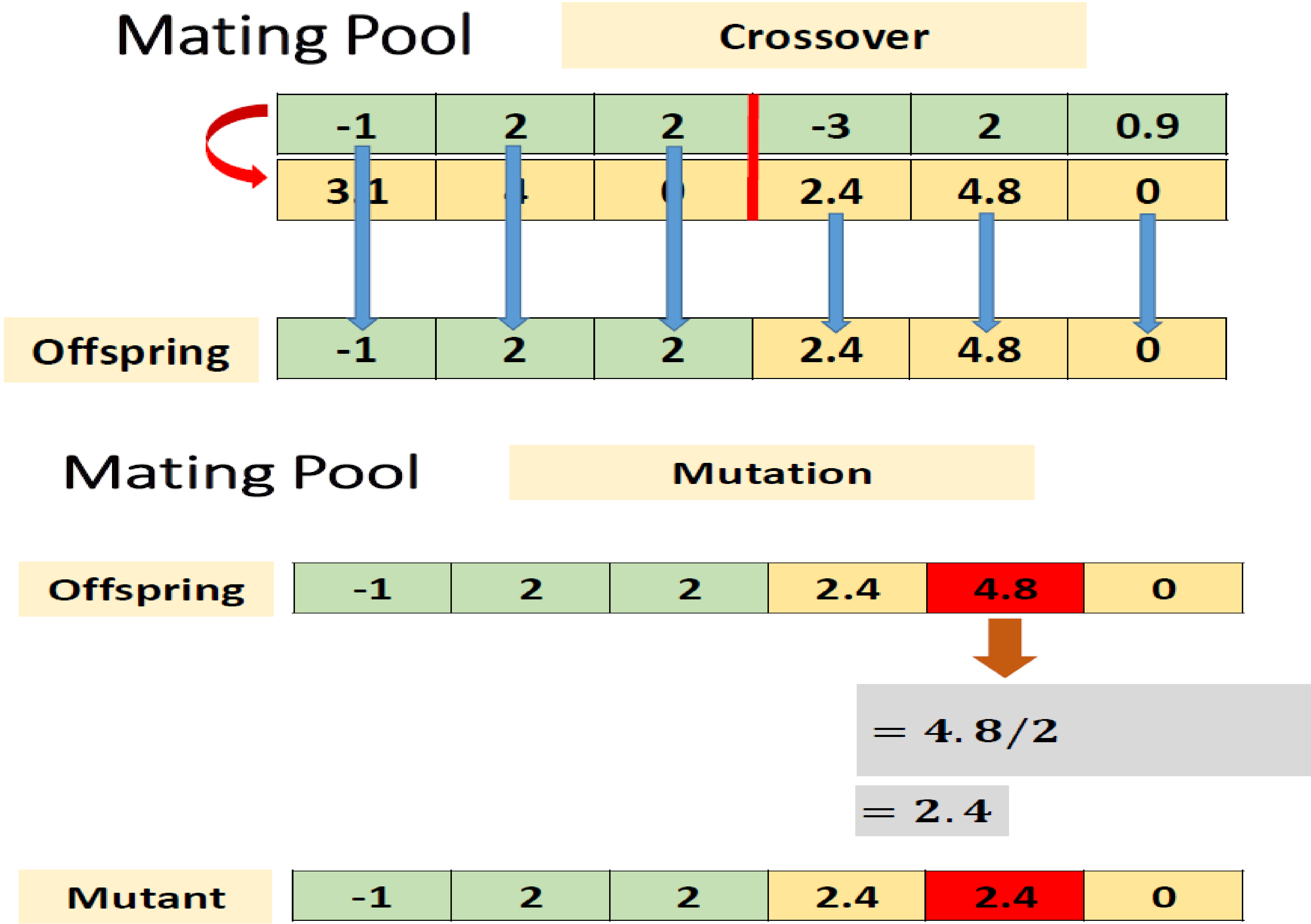
$$y' = 4w_1 - 2w_2 + 7w_3 + 5w_4 + 11w_5 + w_6$$
$$y' = 4 * 2.4 - 2 * 0.7 + 7 * 8 + 5 * -2 + 11 * 5 + 1.1$$
$$y' = 110.3$$

$$F(c) = \frac{1}{error} = \frac{1}{|44.1 - 110.3|} = \frac{1}{66.2} = 0.015$$

Mating Pool

- **Add best 3 individuals to the mating pool for producing the next generation of solutions.**

-1	2	2	-3	2	0.9
3.1	4	0	2.4	4.8	0
-2	3	-7	6	3	3


Mating Pool
Crossover
-1 2 2 -3 2 0.9
3.1 4 0 2.4 4.8 0
Offspring
-1 2 2 2.4 4.8 0
Mating Pool
Mutation
Offspring
-1 2 2 2.4 4.8 0
= 4.8/2
= 2.4
Mutant
-1 2 2 2.4 2.4 0

Selection

- GA is a random-based optimization technique. There is no guarantee that the new individuals will be better than the previous individuals. **Keeping the old individuals at least saves the results from getting worse**.

New Population (Generation 1)

Old Individuals

-1	2	2	-3	2	0.9
3.1	4	0	2.4	4.8	0
-2	3	-7	6	3	3

New Individuals

-1	2	2	2.4	2.4	0
3.1	4	0	6	1.5	3
-2	3	-7	-3	1	0.9

New Population (Generation 1)

						y'	$F(C)$
-1	2	2	-3	2	0.9	13.9	0.033
3.1	4	0	2.4	4.8	0	69.2	0.04
-2	3	-7	6	3	3	3	0.024
-1	2	2	2.4	2.4	0	44.4	3.333
3.1	4	0	6	1.5	3	53.9	0.102
-2	3	-7	-3	1	0.9	-66.1	0.009

New Population (Generation 2)

						y'	$F(C)$
3.1	4	0	2.4	4.8	0	69.2	0.04
-1	2	2	2.4	2.4	0	44.4	3.333
3.1	4	0	6	1.5	3	53.9	0.102
3.1	4	0	2.4	1.2	0	29.6	0.069
-1	2	2	6	0.75	3	47.25	0.318
3.1	4	0	2.4	2.4	0	42.8	0.77

Code Example for GA

- https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6
- The tutorial starts by presenting the equation that we are going to implement. The equation is shown below:

$$\mathbf{Y = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6}$$

- The equation has 6 inputs (x_1 to x_6) and 6 weights (w_1 to w_6) as shown and inputs values are $(x_1,x_2,x_3,x_4,x_5,x_6)=(4,-2,7,5,11,1)$.
- We are looking to find the parameters (weights) that maximize such equation.

```python
import numpy
import ga

"""
The y=target is to maximize this equation ASAP:
    y = w1x1+w2x2+w3x3+w4x4+w5x5+6wx6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)
    What are the best values for the 6 weights w1 to w6?
    We are going to use the genetic algorithm for the best
    possible values after a number of generations.
"""

# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = 6

"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4

# Defining the population size.
pop_size = (sol_per_pop,num_weights)
```

```python
pop_size = (sol_per_pop,num_weights)
# The population will have sol_per_pop chromosome where
# each chromosome has num_weights genes.
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)

num_generations = 5
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = ga.cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = ga.select_mating_pool(new_population, fitness,
                                      num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = ga.crossover(parents,\
        offspring_size=(pop_size[0]-parents.shape[0], num_weights))

    # Adding some variations to the offsrping using mutation.
    offspring_mutation = ga.mutation(offspring_crossover)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

    # The best result in the current iteration.
    print("Best result : ",\
          numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))
```

&B

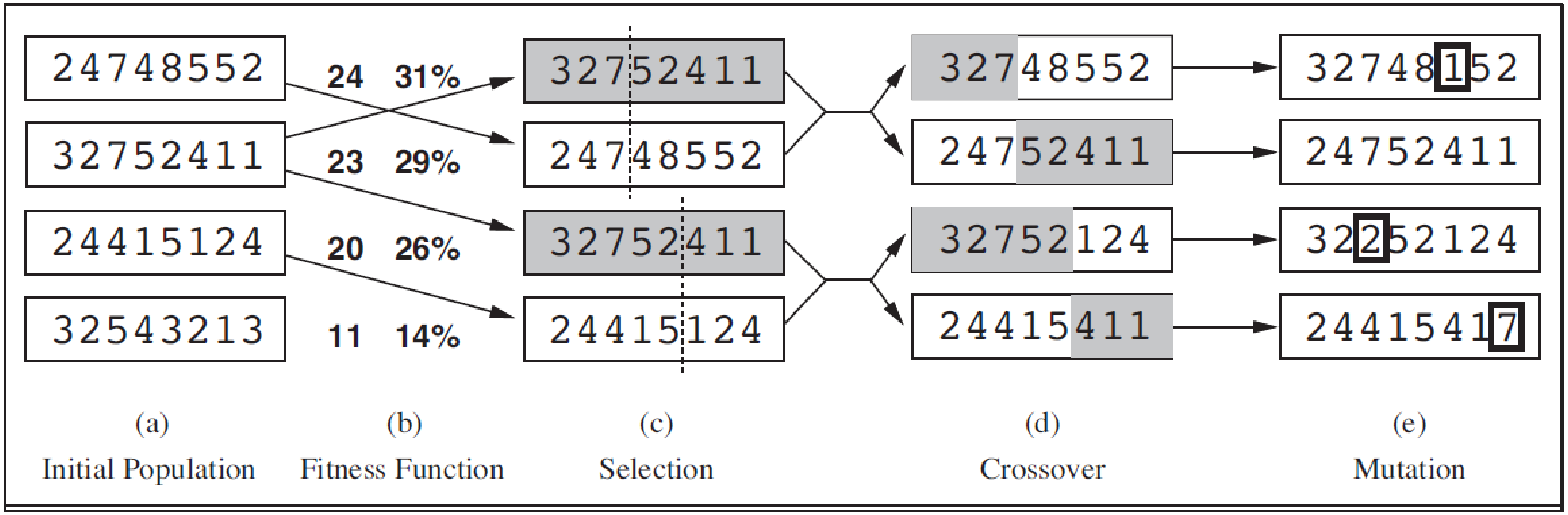
```

# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation
fitness = ga.cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])
```

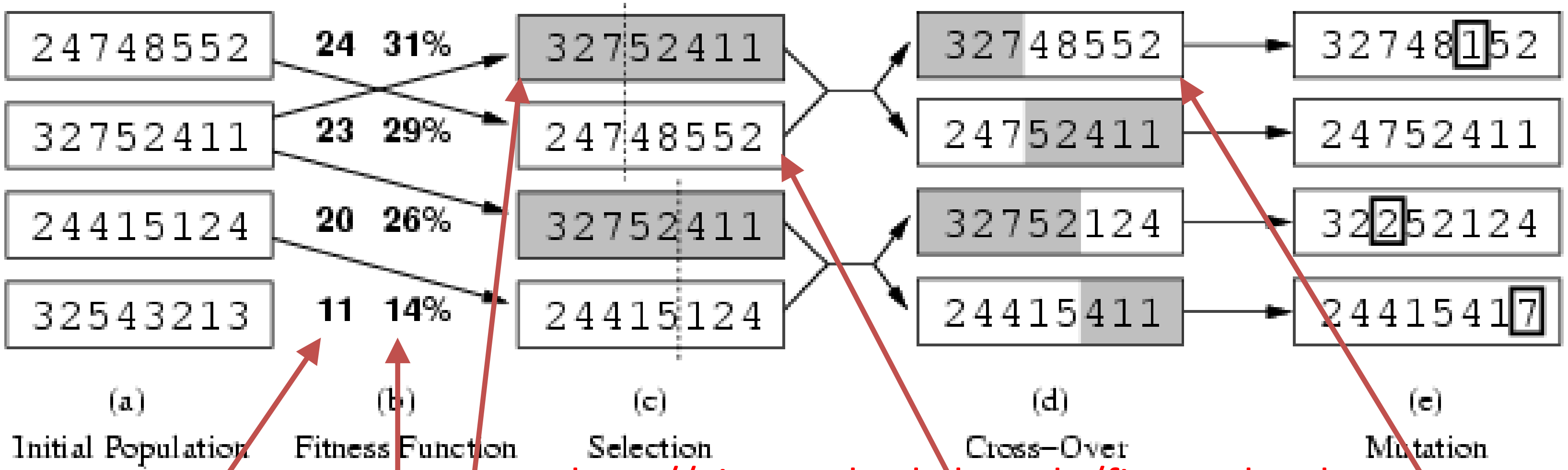
```
[[ 0.44914368 -3.68276722 -3.28688763 -2.51142983 -3.64088132 -1.95845449]
 [-0.71403688  0.55629491 -2.91976592 -1.07344219  0.39909999 -0.66709182]
 [-0.40454659  1.53022263 -0.77733991 -2.56663068  1.98915243 -1.82231662]
 [-1.66459878  2.98133991 -2.09794521  0.52346122 -1.68002539 -2.88114995]
 [-3.32598056 -2.7915747   1.72678859  1.89099232 -2.83302684  3.09768898]
 [ 2.91902592  0.0623401   2.95717192 -1.52771546  3.25655068  2.2282033 ]
 [-1.60688609 -1.84650636 -1.0702571   3.66832867 -1.30358346  3.25418259]
 [ 3.54387828 -3.00561578 -3.79383594  2.34597843 -2.9657875   0.12516974]]
Generation :  0
Best result :  50.67357759562205
Generation :  1
Best result :  54.36619169370305
Generation :  2
Best result :  61.41518272205099
Generation :  3
Best result :  62.22627484711039
Generation :  4
Best result :  71.6404640252752
Best solution :  [[[ 3.54387828 -3.00561578 -3.79383594  2.34597843 -4.87186825
    0.12516974]]]
Best solution fitness :  [71.64046403]
```

Genetic Algorithm for 8-Queue

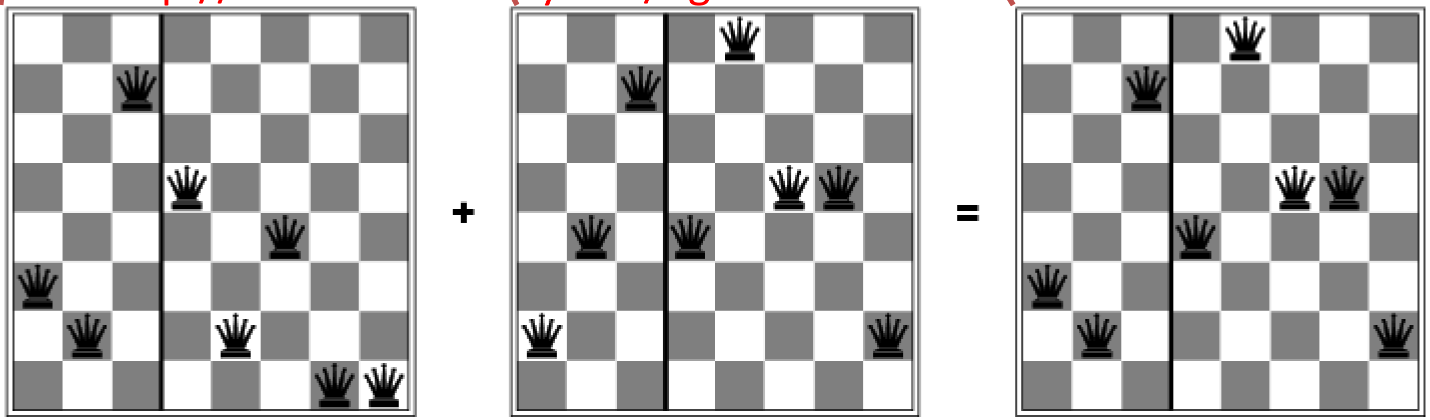


http://aima.cs.berkeley.edu/figures.html

- 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.



http://aima.cs.berkeley.edu/figures.html



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

Local Search in Continuous Spaces

Local Search in Continuous Spaces

- Infinite branching factor
- P-center problem / p=3

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

- Let the objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ be a function on six continuous-valued variables
- **Discretize/12 branching factor**
- The gradient of the objective function $\nabla f = 0$ is a vector that gives the magnitude and the direction of the steepest slope

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

- In many cases, we cannot solve equation $\nabla f = 0$ in closed form (globally), but can compute the gradient locally.
- We can perform **steepest-ascent hill climbing** by updating the current state u via the formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathrm{u})$$

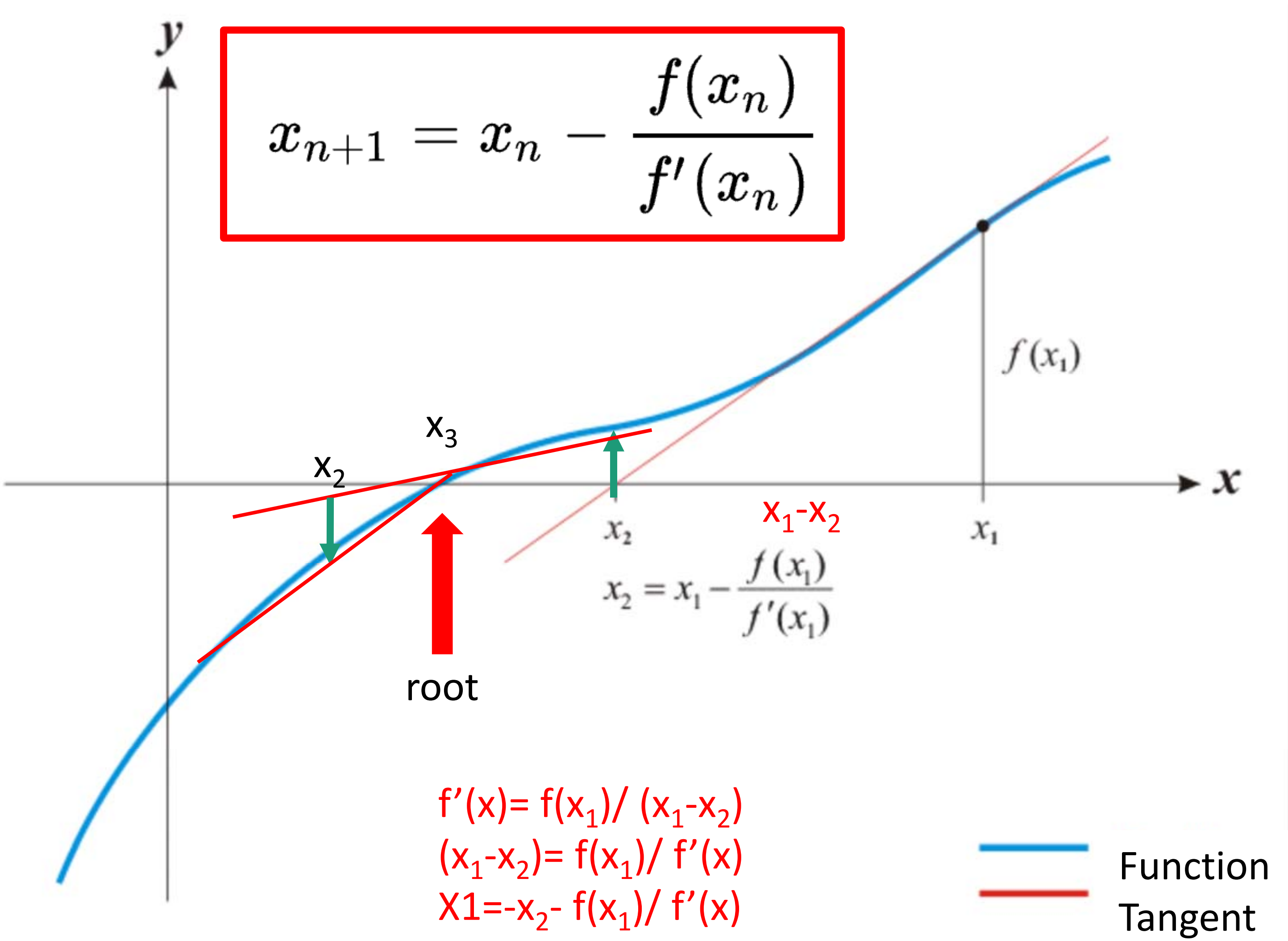
$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c)$$

- Where α is a small constant (Step size)

Empirical gradient 經驗梯度

- If the objective function is not differentiable, the empirical gradient can be determined by evaluating the response to small increments and decrements is each coordinate.
- Adjusting the value of constant α is a central;
  - if α is too small, too many steps are needed;
  - if α is too large, the search could overshoot the maximum.
- Line search repeatedly doubles the value of α until f starts to decrease again.
- Equations of the form g(x)=0 can be solved by using the **Newton–Raphson method**.
- It works by computing a new estimate for the **root x** according to the Newton's formula $x \leftarrow x - \frac{g(x)}{g\prime(x)}$

Newton–Raphson method



https://zh.wikipedia.org/wiki/牛顿法

Newton–Raphson method

- To find a maximum or minimum of f, we need to find $\boldsymbol{x}$ s.t. the gradient is zero; i.e., $\nabla f(\boldsymbol{x}) = 0$
- Setting $g(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$ in Newton's formula and writing it matrix-vector form, we have, $\boldsymbol{x} \leftarrow \boldsymbol{x} - H_f^{-1}(\boldsymbol{x}) \nabla f(\boldsymbol{x})$, where $H_f(\boldsymbol{x})$ is the Hessian matrix of second derivatives, $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$
- The Newton-Raphson becomes expensive in high-dimensional spaces.
- Local search suffers from local maxima, ridges, and plateaus in continuous state spaces just as much as in discrete spaces.

Hessian matrix

$$f(x_1, x_2, \dots, x_n)$$

$$\mathrm{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

$$\mathrm{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

https://zh.wikipedia.org/wiki/牛顿法