Escaping Local Optima

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

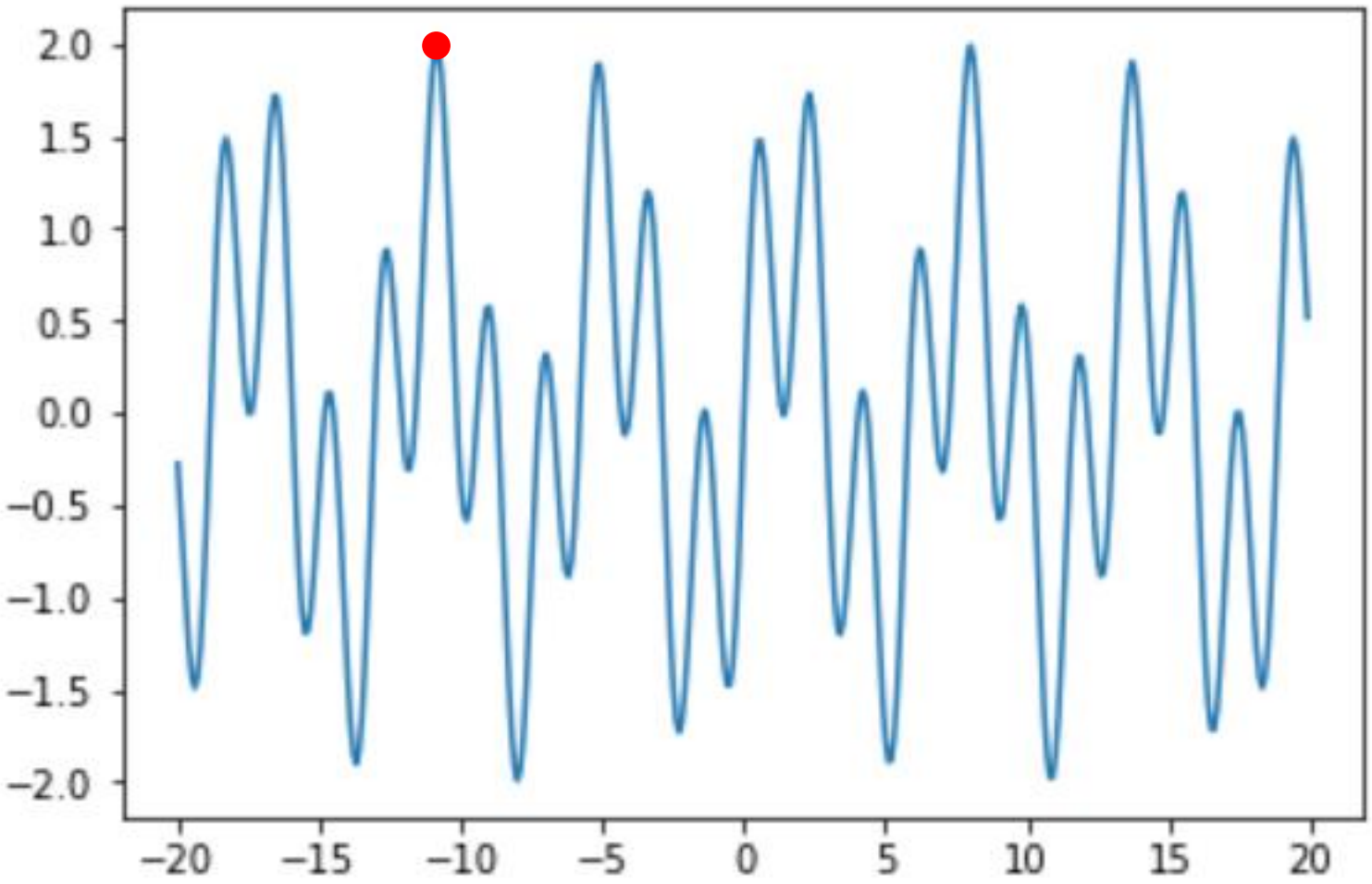
Escaping Local Optima

- In general, local search is a very easy method to design and implement and gives fairly good solutions very quickly. This is why it is a widely used optimization method in practice
- One of the main disadvantages of LS is that it converges toward **local optima**.
- Moreover, the algorithm can be **very sensitive to the initial solution**; that is, a large variability of the quality of solutions may be obtained for some problems.
- Local search works well if there are not too many local optima in the search space or the quality of the different local optima is more or less similar.
- If the objective function is highly multimodal, which is the case for the majority of optimization problems, local search is usually not an effective method to use.

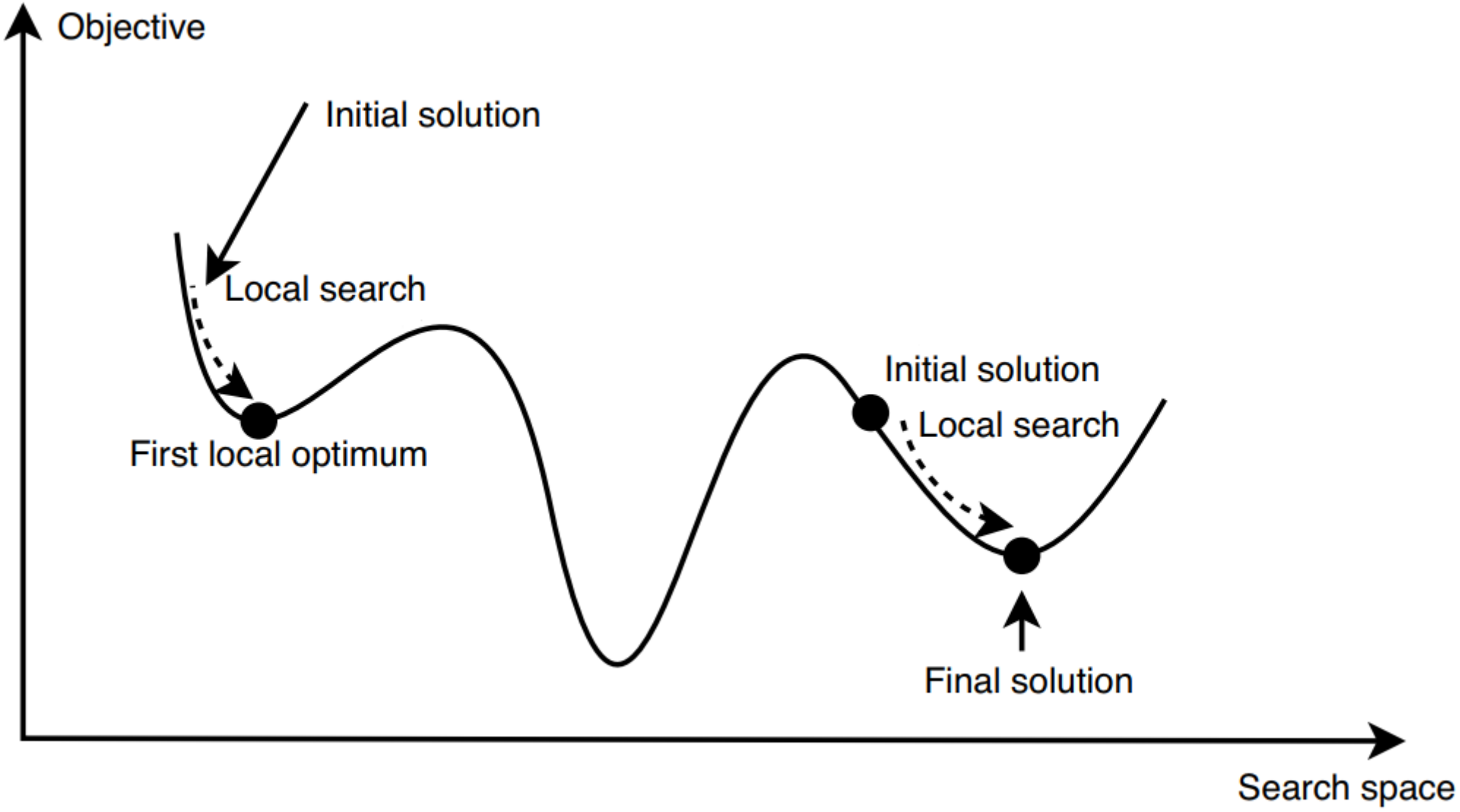
Convergence to Local Optima
Basic Example

- We aim to minimize a continuous function with multiple local optima

$$f(x) = sin(x) + \sin(3.33x)$$



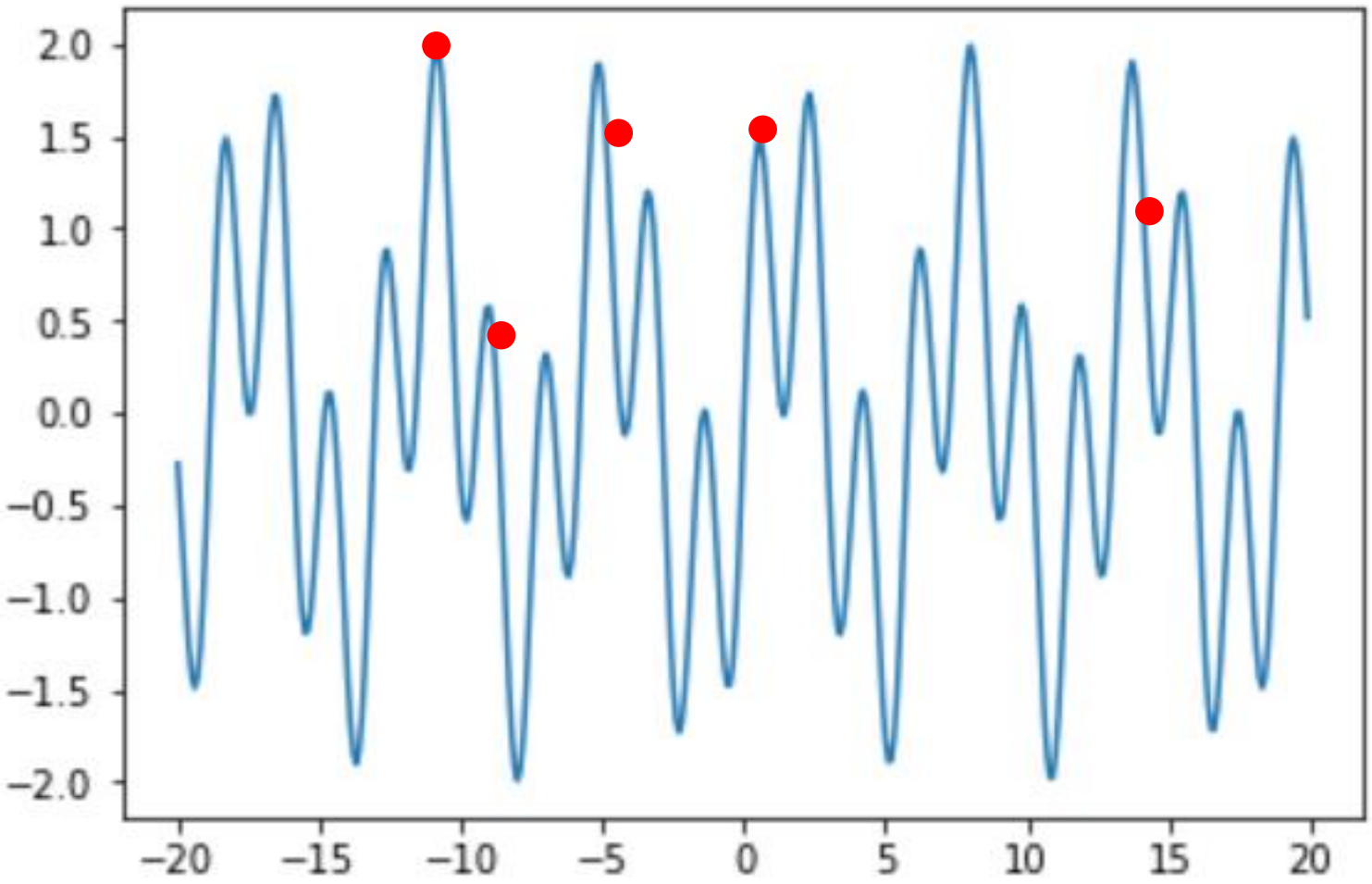
Multistart Local Search



Convergence to Local Optima
Basic Example

- We aim to minimize a continuous function with multiple local optima

$$f(x) = sin(x) + \sin(3.33x)$$



Convergence to Local Optima
Basic Example

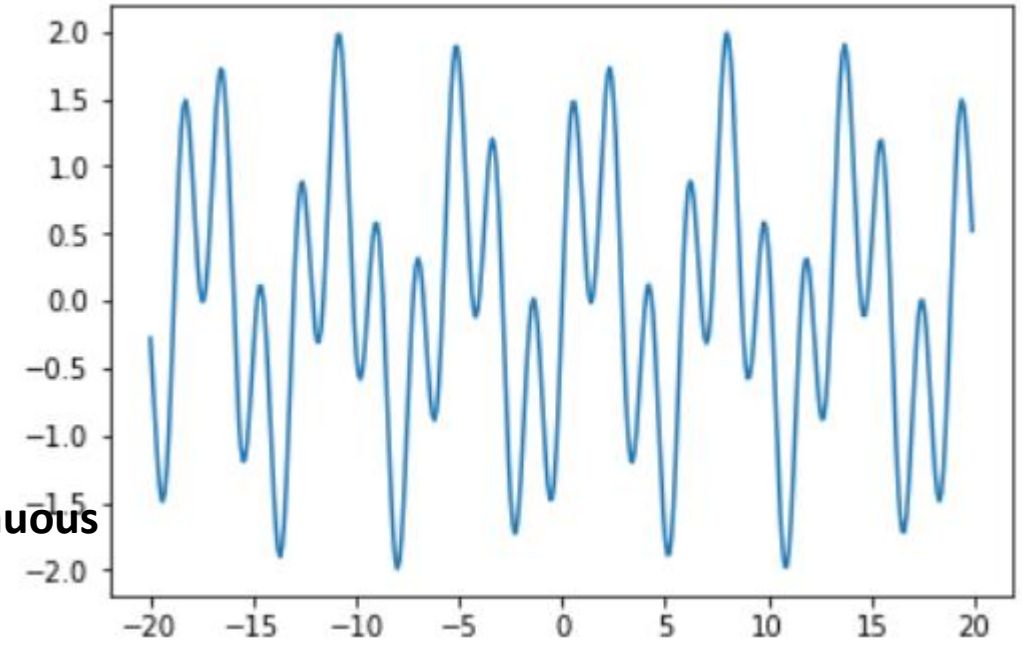
- We aim to minimize a continuous function with multiple local optima

$$f(x) = sin(x) + \sin(3.33x)$$

Function to compute the objective value

```
# objective function
def objective(x):
    return sin(x) + sin((10.0 / 3.0) * x)

# define range for input
r_min, r_max = -20, 20
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = objective(inputs)
# create a line plot of input vs result
pyplot.plot(inputs, results)
```

Plot continuous function



Convergence to Local Optima
Generate Initial Solution

```
# seed the pseudorandom number generator
seed(1)

# define range for input
bounds = asarray([[-20.0, 20.0]])

# generate an initial point
best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])

# evaluate the initial point
best_eval = objective(best)

# current working solution
curr, curr_eval = best, best_eval
```

Generate initial solution within the specified input range

Compute the objective solution of the initial solution

Initial solution is both: the current solution and the best solution found so far

Convergence to Local Optima
Hill Climbing

```
# seed the pseudorandom number generator
seed(1)
# define the total iterations
n_iterations = 100000
# define the maximum step size
step_size = 0.1

scores = list()

# run the algorithm
for i in range(n_iterations):
    # take a step
    candidate = curr + randn(len(bounds)) * step_size
    # evaluate candidate point
    candidate_eval = objective(candidate)
    # check for new best solution
    if candidate_eval < best_eval:
        # store new best point
        best, best_eval = candidate, candidate_eval
        # keep track of scores
    scores.append(best_eval)
```

Neighbourhood step size

Generate random solution within the neighbrood

Compute objective value of the new candidate solution

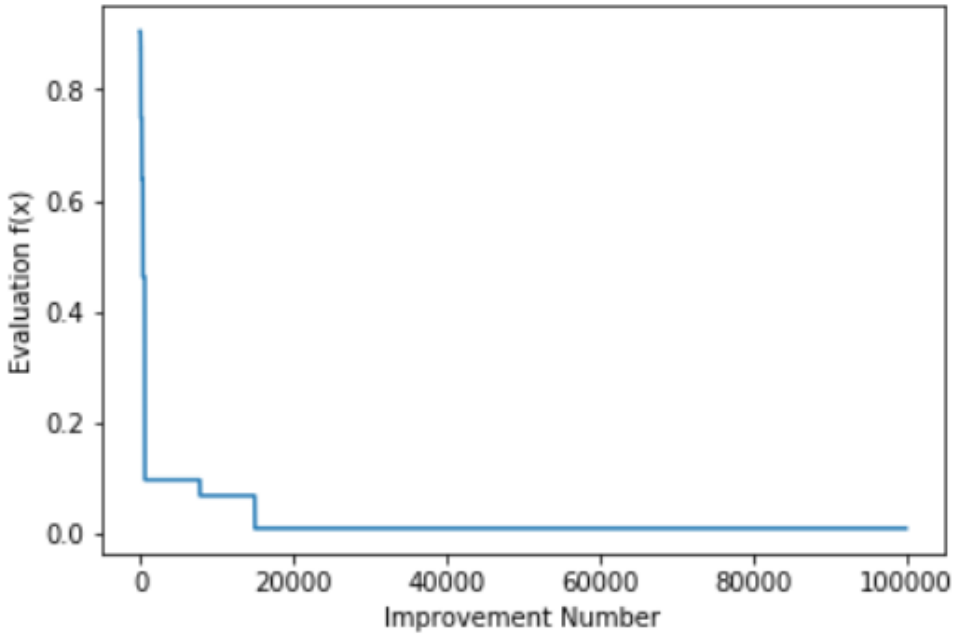
Check if the new candidate solution is better than the best solution found so far; If yes, update the best solution found

Convergence to Local Optima
Local Optima

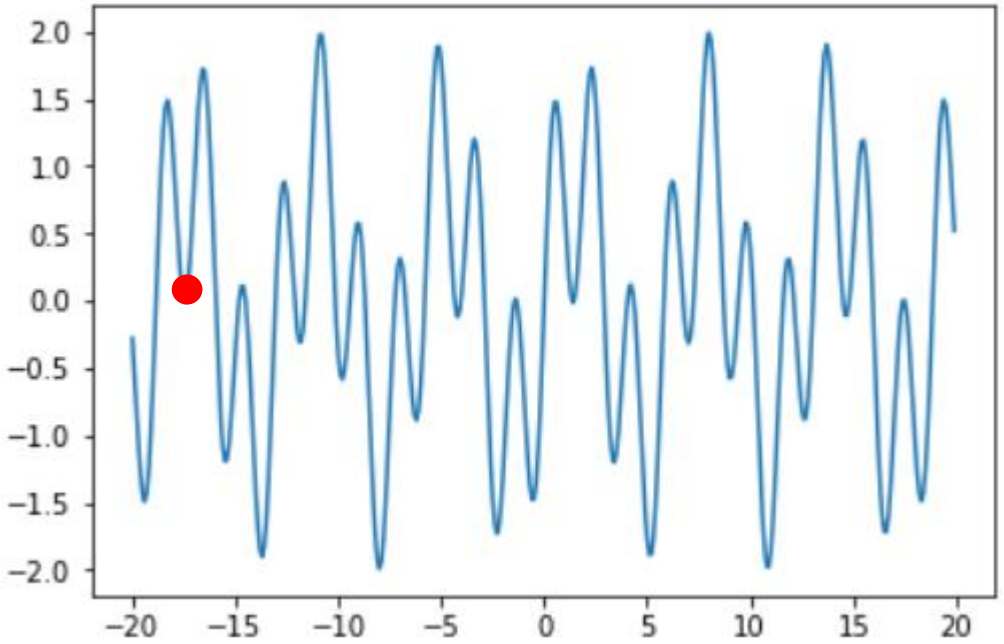
```
min(scores)
```

```
array([0.00999143])
```

```
# line plot of best scores
pyplot.plot(scores, '-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```



Seed = 1

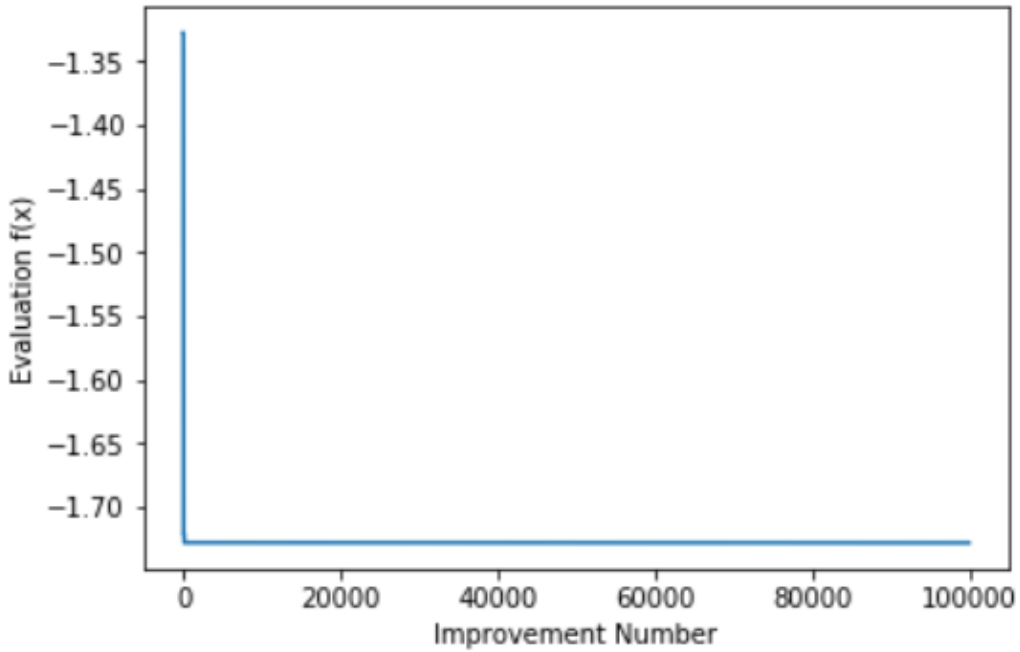


Convergence to Local Optima
Local Optima

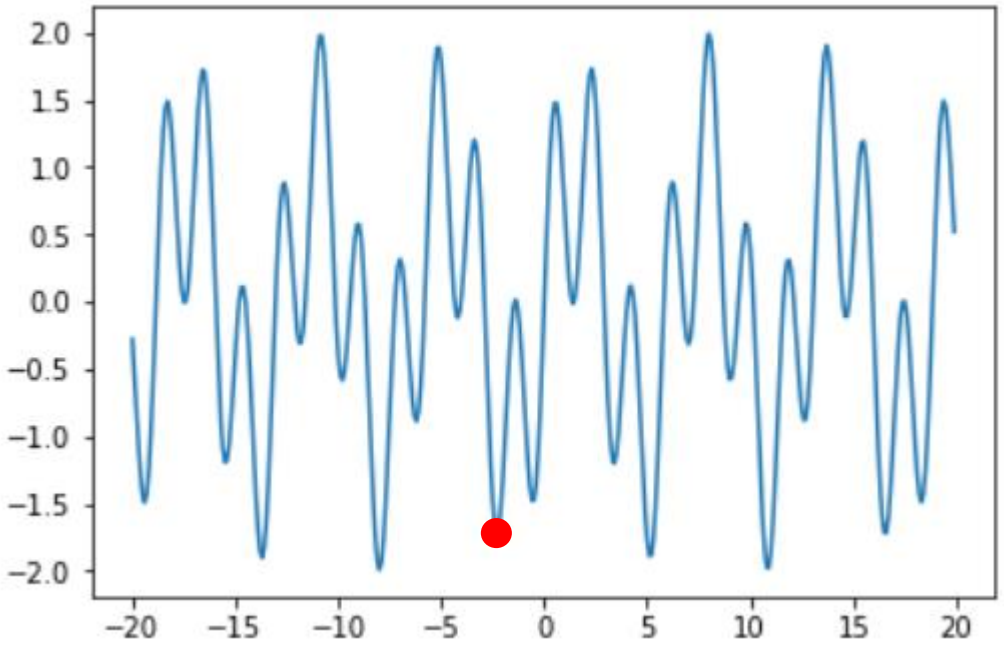
```
min(scores)
```

```
array([-1.72814962])
```

```
# line plot of best scores
pyplot.plot(scores, '-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```



Seed = 2



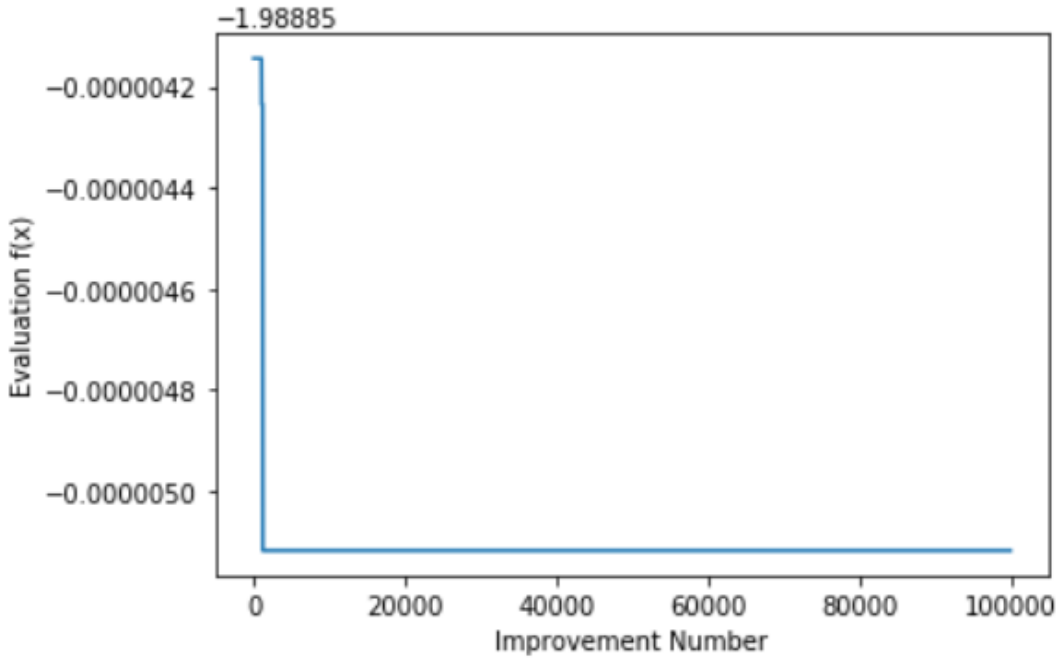
Convergence to Local Optima
Local Optima

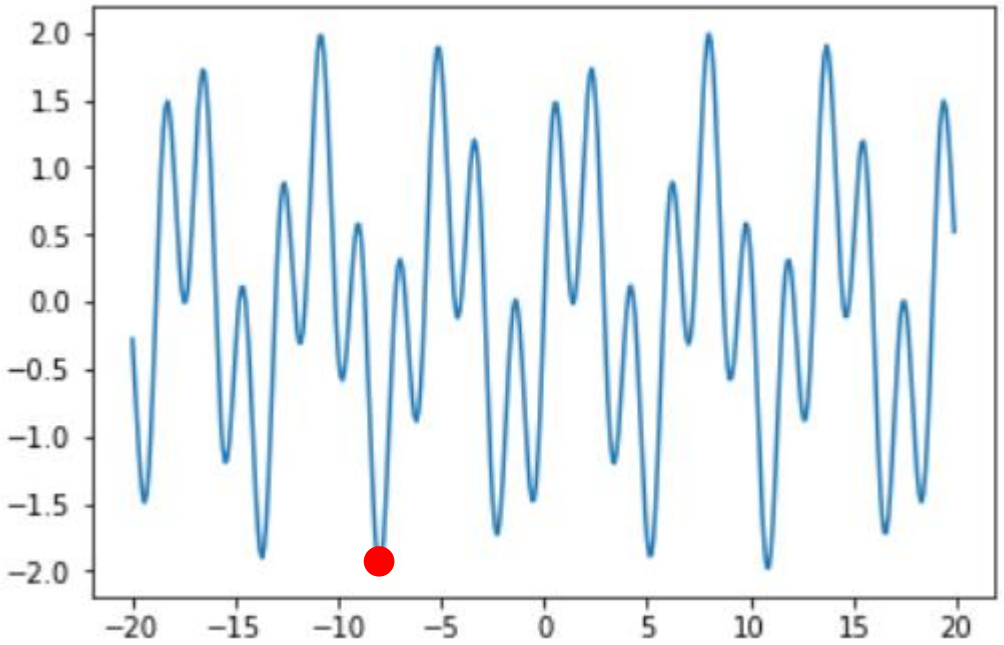
```
min(scores)
```

```
array([-1.98885512])
```

```
# line plot of best scores
pyplot.plot(scores, '-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Seed = 10



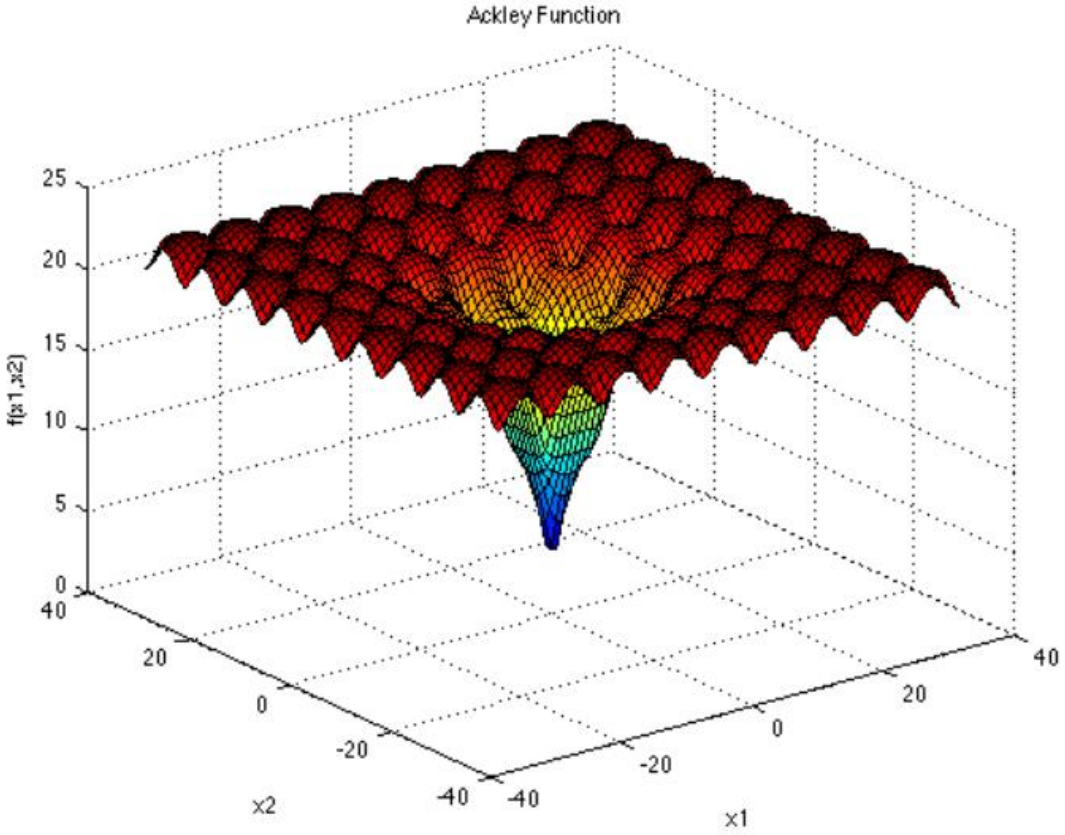


Multistart Local Search

- To escape local optima, local search algorithms can be initialize several times
- As we can generate local optima with high variability, eventually, after many initializations, the algorithm will find the local optima that corresponds to the global optima

1. **While** (termination criteria (1) is not met)
2. **Initialize**: Generate random initial solution, p_{best}
    3. **While** (termination criteria (2) is not met)
        4. Generate a new solution (or a set of new solutions) p_{new} by applying a small perturbation (search operator) to p_{best}
        5. If p_{new} is better than p_{best}, than p_{best}= p_{new}
        6. Go back to 3, until termination criteria (2) is not met
7. **Go back to 2, until termination criteria (1) is not met**

Multistart Local Search – Random Sampling



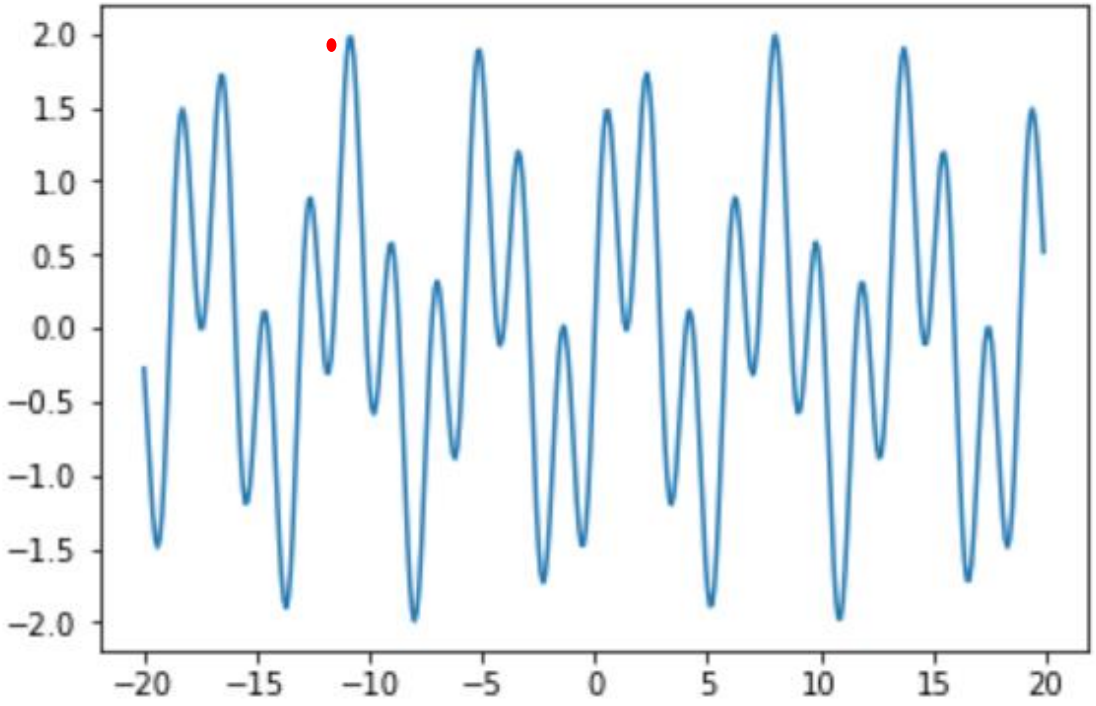
The Ackley function is widely used for testing optimization algorithms. In its two-dimensional form, as shown in the plot, it is characterized by a nearly flat outer region, and a large hole at the centre. The function poses a risk for optimization algorithms, particularly hillclimbing algorithms, to be trapped in one of its many local minima.

$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Source: http://www.sfu.ca/~ssurjano/ackley.html

Random Walks

- Random walks are like hill climbers, with the exception that they do not use the objective function to guide the search direction.
- Start at a (random) location and take random steps



Simulated Annealing

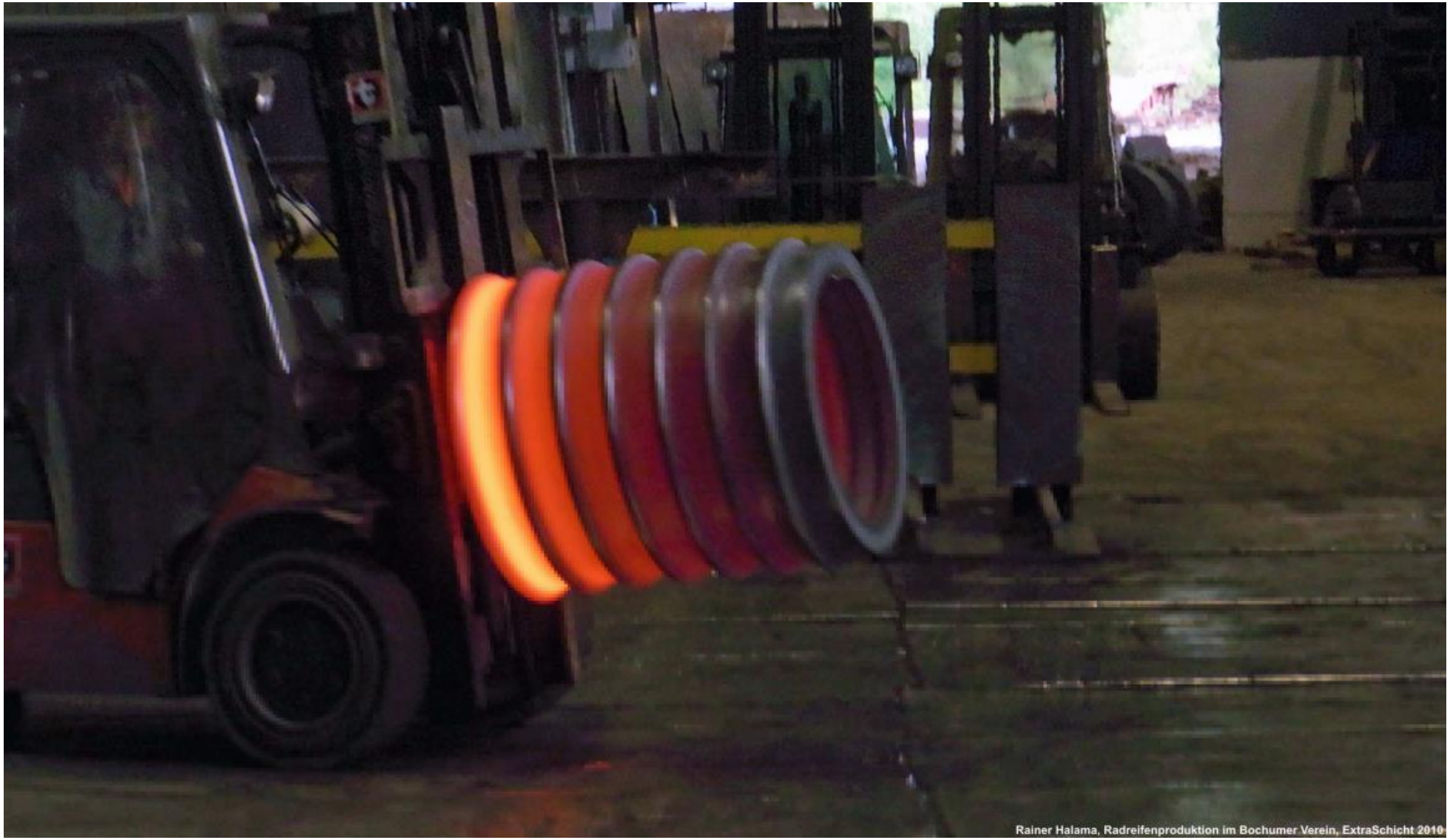
Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

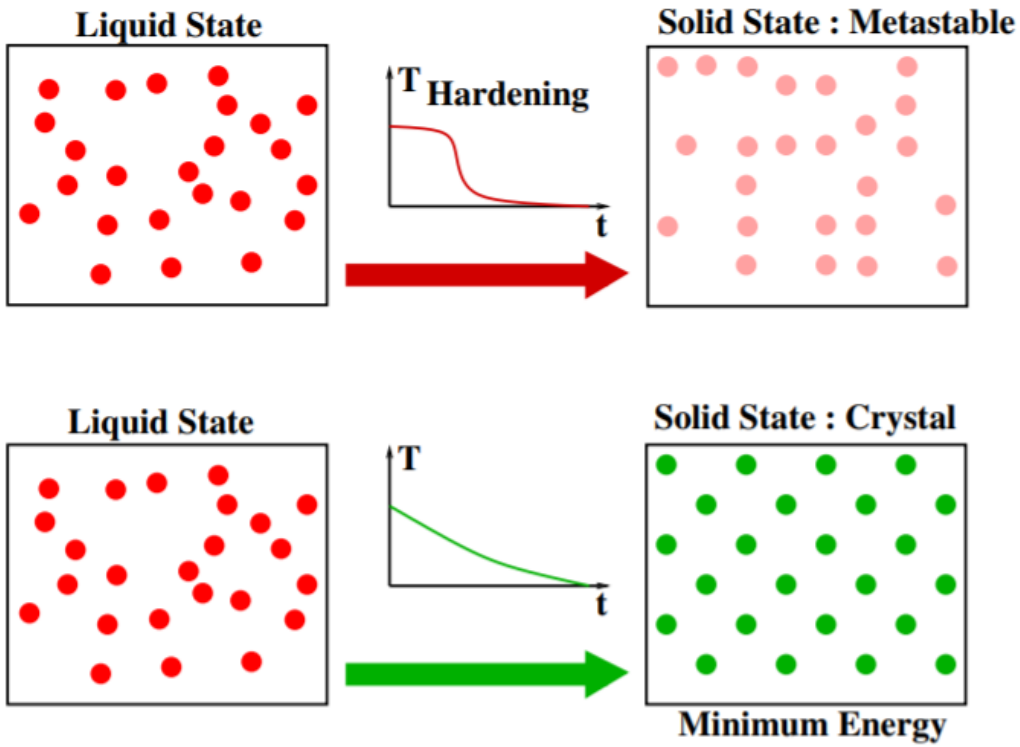
Annealing Process



Source: http://iao.hfuu.edu.cn/teaching/lectures/metaheuristic-optimization

Annealing Process

- The annealing process requires heating and then slowly cooling a substance to obtain a strong crystalline structure. The strength of the structure depends on the rate of cooling metals.
- **If the initial temperature is not sufficiently high or a fast cooling is applied, imperfections (metastable states) are obtained.**
- In this case, the cooling solid will not attain thermal equilibrium at each temperature. **Strong crystals are grown from careful and slow cooling.**
- The Simulated Annealing algorithm simulates the energy changes in a system subjected to a cooling process until it converges to an equilibrium state (steady frozen state). This scheme was developed in 1953 by Metropolis



Metropolis Simulation

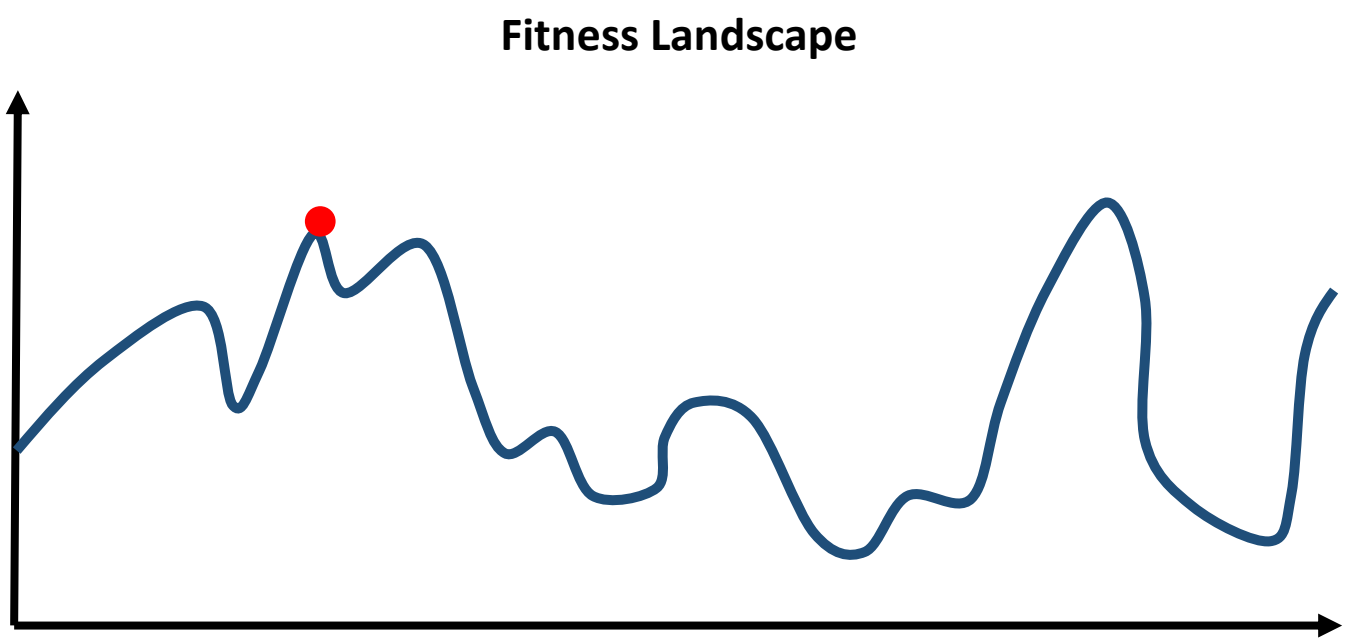
- In 1953, three American researchers (Metropolis, Rosenbluth, and Teller) developed an algorithm to **simulate the physical annealing**.
- Their aim was to reproduce faithfully the evolution of the physical structure of a material undergoing annealing.
- Starting from an initial state i of energy E_i, a new state j of energy E_j is generated by modifying the position of one particle.
- If the energy difference, $E_i - E_j$, is positive, the state j becomes the new current state. If the energy difference is less than or equal to zero, then the probability that the state j becomes the current state is given by:

$$e^{\left(\frac{E_i - E_j}{k_b . T}\right)}$$

- Where T represents the temperature of the solid and k_B is the Boltzmann constant (1.38×10−23 joule/Kelvin)

Simulated Annealing in Optimization

- SA replicates the annealing process by enabling under some conditions the **degradation of a solution**. The goal is to escape from local optima.



$$P(\Delta E) = \begin{cases} e^{-\frac{\Delta E}{T}} & if\ \Delta E > 0 \\ 1 & otherwise \end{cases}$$

- It uses a control parameter, called **temperature**, to determine the **probability of accepting non-improving solutions**.
- The temperature is gradually decreased according to a **cooling schedule** such that few non-improving solutions are accepted at the end of the search.

Metaheuristic - Simulated Annealing

- The **objective function** of the problem is analogous to the **energy** state of the system.
- A **solution** of the optimization problem corresponds to a **system state**.
- The **decision variables** associated with a solution of the problem are analogous to the **molecular positions**.
- The global optimum corresponds to the ground state of the system.
- Finding a local minimum implies that a metastable state has been reached

Physical System	Optimization Problem
System state	Solution
Molecular positions	Decision variables
Energy	Objective function
Ground state	Global optimal solution
Metastable state	Local optimum
Rapid quenching	Local search
Temperature	Control parameter T
Careful annealing	Simulated annealing

Simulated Annealing

- Escaping local optima: The higher the temperature the higher the probability of accepting a worst move.

- Better move is always accepted $\boldsymbol{P = 1}$

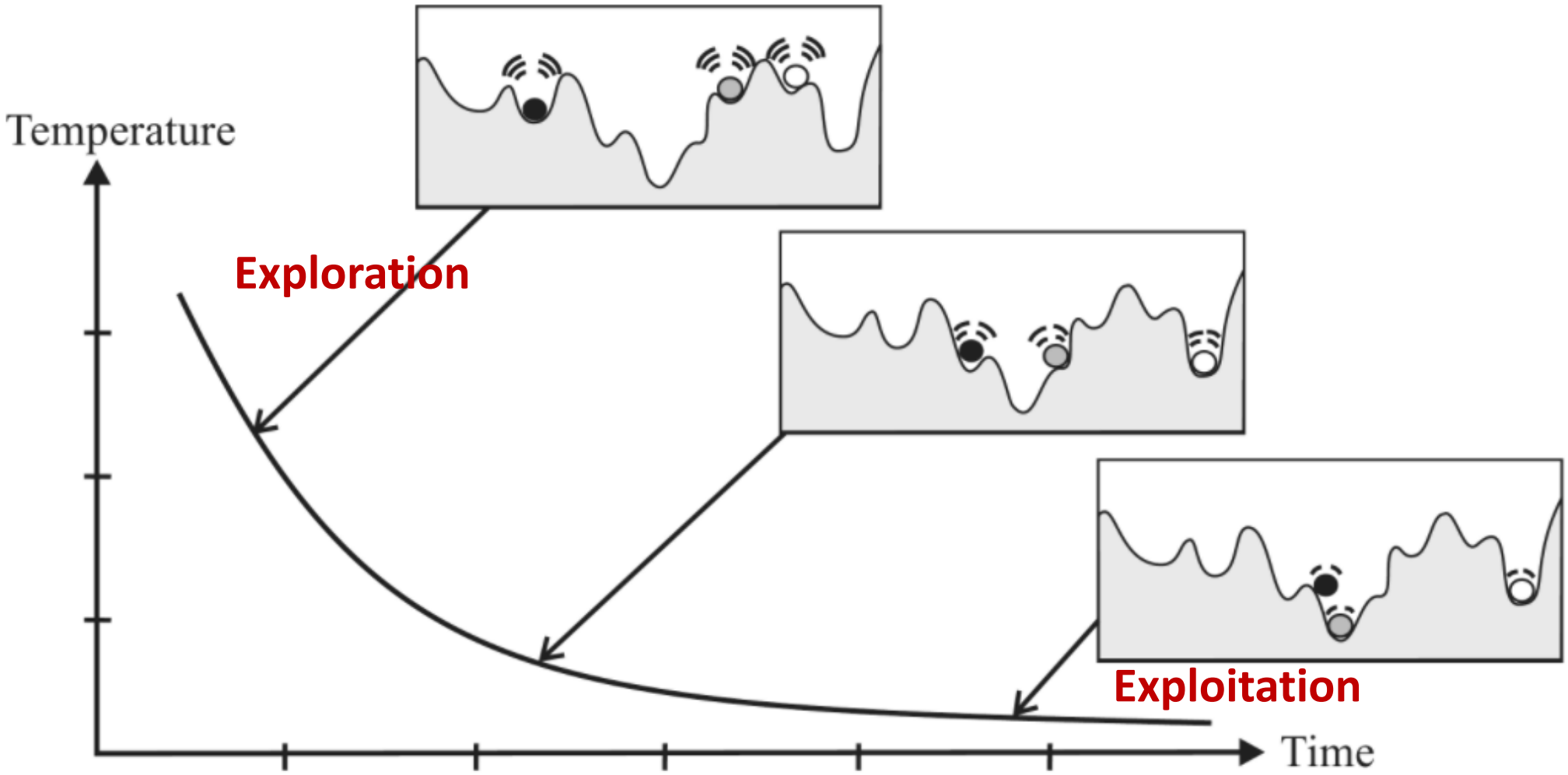
$$P(\Delta E) = \begin{cases} e^{-\frac{\Delta E}{\boldsymbol{T}}} & if\ \Delta E > 0 \\ 1 & otherwise \end{cases}$$

- Δ**E** is the objective value difference between the new $\boldsymbol{f(x')}$ and old candidate solution $\boldsymbol{f(x)}$

$$\Delta E = f(x') - f(x)$$

- Temperature T reduced according to a specific schedule over the iterations

Cooling Schedule



Cooling Schedule

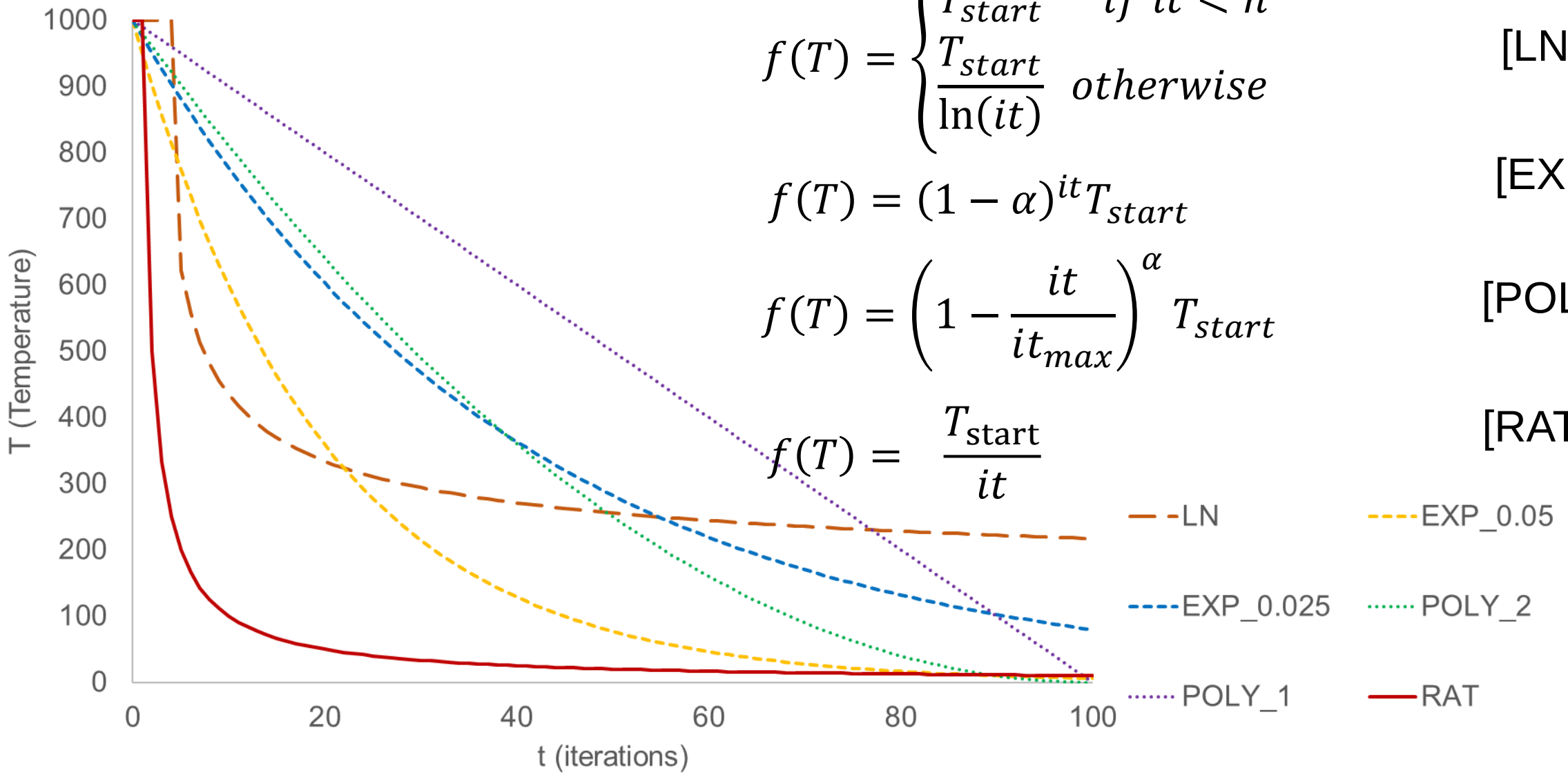
T_{start} - initial temperature
it – current iteration
it_{max}-maximum number of iterations
α- calibration parameter

$$f(T) = \begin{cases} T_{start} & if\ it < n \\ \dfrac{T_{start}}{\ln(it)} & otherwise \end{cases}$$ [LN]

$$f(T) = (1-\alpha)^{it} T_{start}$$ [EXP]

$$f(T) = \left(1 - \frac{it}{it_{max}}\right)^{\alpha} T_{start}$$ [POLY]

$$f(T) = \frac{T_{start}}{it}$$ [RAT]



Cooling Schedule

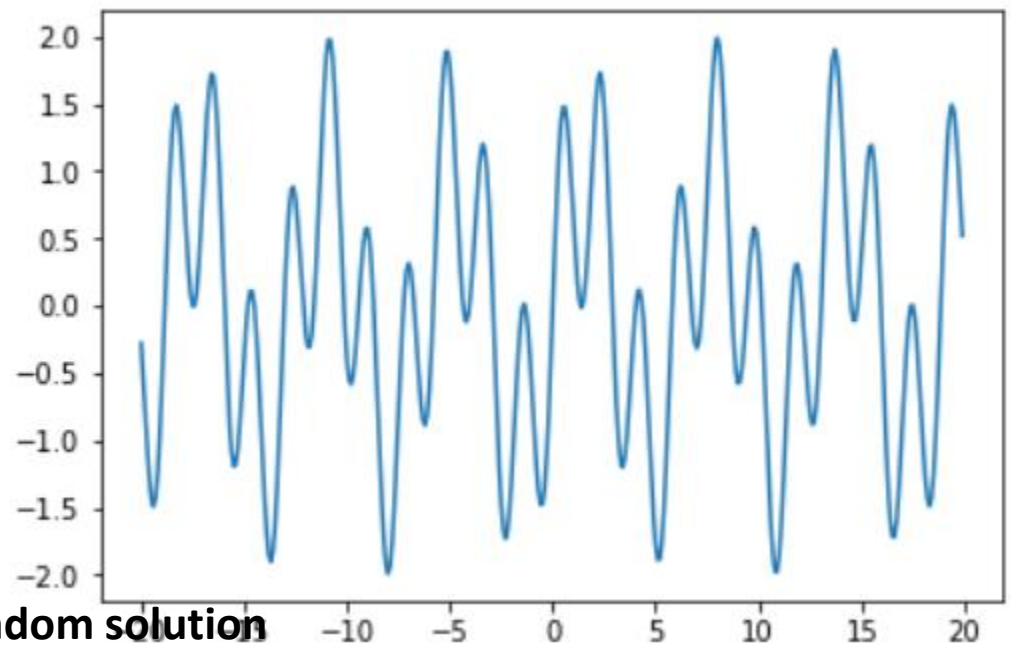
- If **temperature decreases slowly**, convergence to the global optimum has been proven for various optimization problems. However, the number of function evaluations needed to find the **optimum** may be higher than what an exhaustive enumeration would need.
- **Faster cooling schedules** lose guaranteed convergence but progress much faster: Simulated Annealing becomes almost a **local search** algorithm.

Simulated Annealing Algorithm

- Escaping local optima: The higher the temperature the larger the probability of accepting non-improving solutions

1. **Initialize**: Generate initial solution, p_{best}
2. **While** (termination criteria (2) is not met)
   3. Generate a new solution (or a set of new solutions) p_{new} by applying a small perturbation (search operator) to p_{best}
   4. If p_{new} is better than p_{best}, than update p_{best}= p_{new}
   5. **Else update p_{best}= p_{new} with a probability $e^{-\frac{\Delta E}{T}}$**
   6. Update temperature T applying a cooling schedule function
   7. Go back to (4), until termination criteria (3) is not met

Escaping Local Optima
Simulated Annealing




```
# seed the pseudorandom number generator
seed(randseed)
# define the total iterations
n_iterations = 100000
# define the maximum st
step_size = 0.1
# initial temperature
temp =1000
scores = li
it_T=0
tp=temp
# run the algorithm
for i in range(n_iterations):
    # take a step
    candidate = curr + randn(len(bounds)) * step_size
    # evaluate candidate point
    candidate_eval = objective(candidate)
    # check for new best solution
    if candidate_eval < best_eval:
        # store new best point
        best, best_eval = candidate, candidate_
    # difference between candidate and current point evaluation
    diff = candidate_eval - curr_eval
    # calculate temperature for current iteration
    if it_T>100:
        tp = temp / float(i + 1)
        it_T=0
    it_T=it_T+1
    # calculate metropolis acceptance criterion
    metropolis = exp(-diff / tp)
    # check if we should keep the new point
    if diff < 0 or rand() < metropolis:
        # store the new current point
        curr, curr_eval = candidate, candidate_eval
    # keep track of scores
    scores.append(curr_eval)
```

Neighbourhood step size

Initial Temperature

Generate random solution within the neighbood

Compute objective value of the new candidate solution

Check if the new candidate solution is better than the best solution found so far;
If yes, update the best solution found
If not, compute the probability of accepting the new candidate solution.

Accept the new candidate solution with with a probability $e^{-\frac{\Delta E}{T}}$

Escaping Local Optima
Simulated Annealing

Random Seed	Hill Climbing	Simulated Annealing	Random Walks
1	0.009	-1.729	-1.989
2	-1.728	-1.989	-1.989
3	0.106	-1.899	-1.989
4	-1.487	-1.729	-1.989
5	0.148	-1.988	-1.989
6	-0.859	-1.729	-1.989
7	0.014	-1.729	-1.989
8	-0.119	-1.729	-1.989
9	-1.49	-1.727	-1.989
10	-1.989	-1.989	-1.989

Not an interesting problem

Simulated Annealing in Python

Nuno Antunes Ribeiro

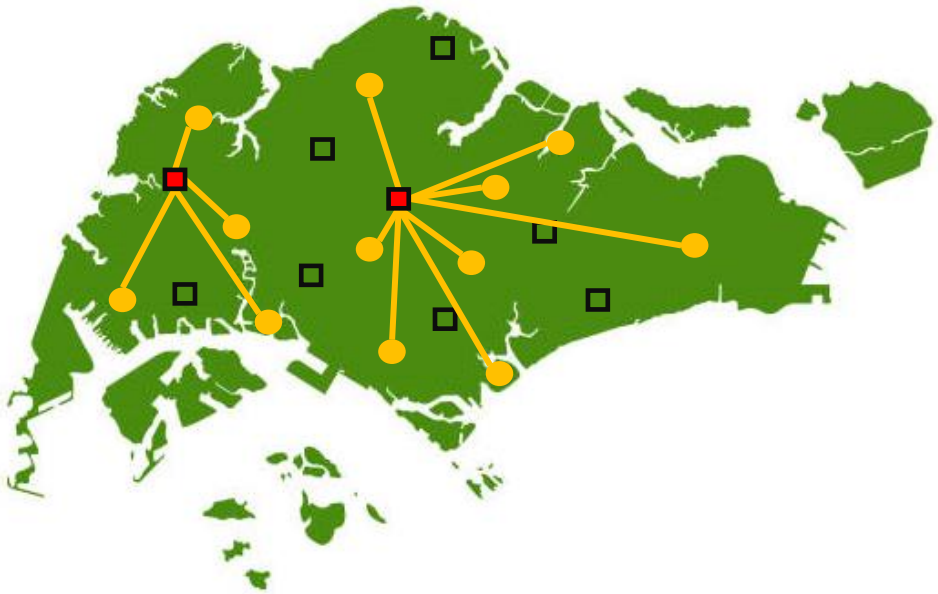
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

P-Median Example

- **Solution Representation:** Binary Encoding
- **Move Operator:** Open a random 1 location and close a random location
- **Replacement Procedure:** First Descent



Number of candidate locations
n=100

Number of locations to open
fac=15

P-Median Example

```
random.seed(1)
iteration=0
objvalue_i=objvalue
objvaluelist=objvalue
program_starts = time.time()
cputime_i=0
temp=821687323 #(InitialSol-LocalSearchSol)/ln(0.9);(26295266-18983919)/ln(0.9)=69393614 ;(230942655-165441358)/ln(0.9)=6216873
#temp=1
#temp=1000000000000000000000000000000000000000
it_t=0
tp=temp
#while (it_t)<600:
while iteration<100000:

    iteration=iteration+1

    yi_open_i=copy.deepcopy(yi_open)
    yi_i=np.zeros([n, 1])

    #Select new location to open
    yi_new_open = random.sample(range(0, n), 1)
    yi_new_open = np.sort(yi_new_open)

    #Identify the location to close (nearest to the new location)
    index_new_closed=np.argmin(distancelct_open[:,yi_new_open])

    #Open and close 1 locations
    yi_open_i[index_new_closed]=yi_new_open
    yi_open_i=np.sort(yi_open_i)
    yi_i[yi_open_i]=1

    #Re-Allocate locations to the closest open location
    distancelct_open=distancelct[np.where(yi_i)[0]]
    assignment_open=np.argmin(distancelct_open, axis=0)
    objvalue_open=distancelct_open.min(axis=0)*demandlct
    objvalue=sum(objvalue_open)
```

$$P = e^{-\frac{\Delta E}{T}} \Leftrightarrow \ln(P) = -\frac{\Delta E}{T}$$

$$\boxed{\Delta E_i = ObjVal_{Rand} - ObjVal_{HillClimbing}}$$

$$P_i = Initial\ Acceptance\ Rate$$

$$\boxed{\ln(P_i) = -\frac{\Delta E_i}{T_i} \Leftrightarrow T_i = -\frac{\Delta E_i}{\ln(P_i)}}$$

P-Median Example

```
random.seed(1)
iteration=0
objvalue_i=objvalue
objvaluelist=objvalue
program_starts = time.time()
cputime_i=0
temp=821687323 #(InitialSol-LocalSearchSol)/ln(0.9);(26295266-18983919)/ln(0.9)=69393614 ;(230942655-165441358)/ln(0.9)=62168732
#temp=1
#temp=1000000000000000000000000000000000000000
it_t=0
tp=temp
#while (it_t)<600:
while iteration<100000:

    iteration=iteration+1

    yi_open_i=copy.deepcopy(yi_open)
    yi_i=np.zeros([n, 1])

    #Select new location to open
    yi_new_open = random.sample(range(0, n), 1)
    yi_new_open = np.sort(yi_new_open)

    #Identify the location to close (nearest to the new location)
    index_new_closed=np.argmin(distancelct_open[:,yi_new_open])

    #Open and close 1 locations
    yi_open_i[index_new_closed]=yi_new_open
    yi_open_i=np.sort(yi_open_i)
    yi_i[yi_open_i]=1

    #Re-Allocate locations to the closest open location
    distancelct_open=distancelct[np.where(yi_i)[0]]
    assignment_open=np.argmin(distancelct_open, axis=0)
    objvalue_open=distancelct_open.min(axis=0)*demandlct
    objvalue=sum(objvalue_open)
```

Select randomly a location to open

Identify the nearest location to the new location to open – location to close

Update binary vector of locations

P-Median Example

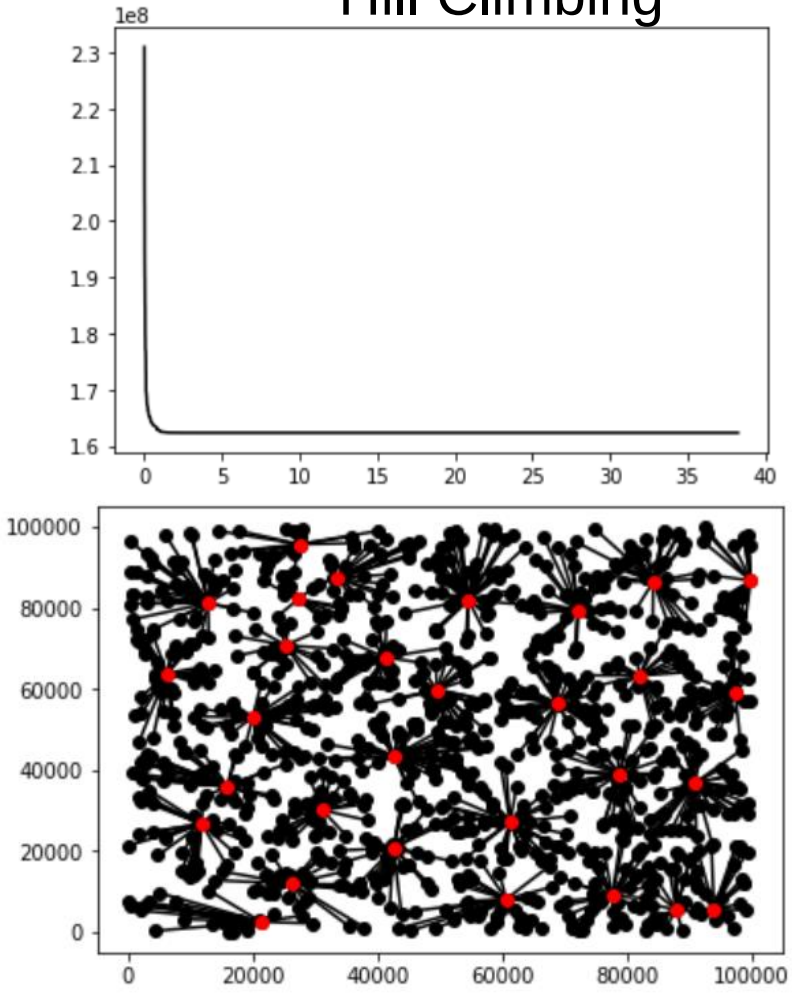
```
    #Compute Acceptance Rate
    diff = objvalue-objvalue_i

    #Calculate temperature for current iteration
    tp = temp / float(iteration + 1)
    #tp = max(0.999*tp,50000)
    #Calculate metropolis acceptance criterion
    metropolis = np.exp(-diff / tp)

    #print(diff)

    #Check if we should keep the new solution
    if diff < 0 or random.random() < metropolis:

        #Update locations and Objective Value
        yi=copy.deepcopy(yi_i)
        yi_open=copy.deepcopy(yi_open_i)
        objvalue_i=copy.deepcopy(objvalue)

        #Compute links
        linkindex_p1=range(n)
        linkindex_p2=assignment_open
        yi_open_index = np.array(yi_open)
        linkindex_p2 = yi_open_index[linkindex_p2]

        #Store new objective value in the objective value list
        objvaluelist=np.append(objvaluelist, objvalue)

        now = time.time()
        cputime_i=np.append(cputime_i, now-program_starts)
        it_t=now-program_starts

#Update last objective value
objvaluelist=np.append(objvaluelist, min(objvaluelist))
now = time.time()
cputime_i=np.append(cputime_i, now-program_starts)
```

Compute probability $e^{-\frac{\Delta E}{T}}$

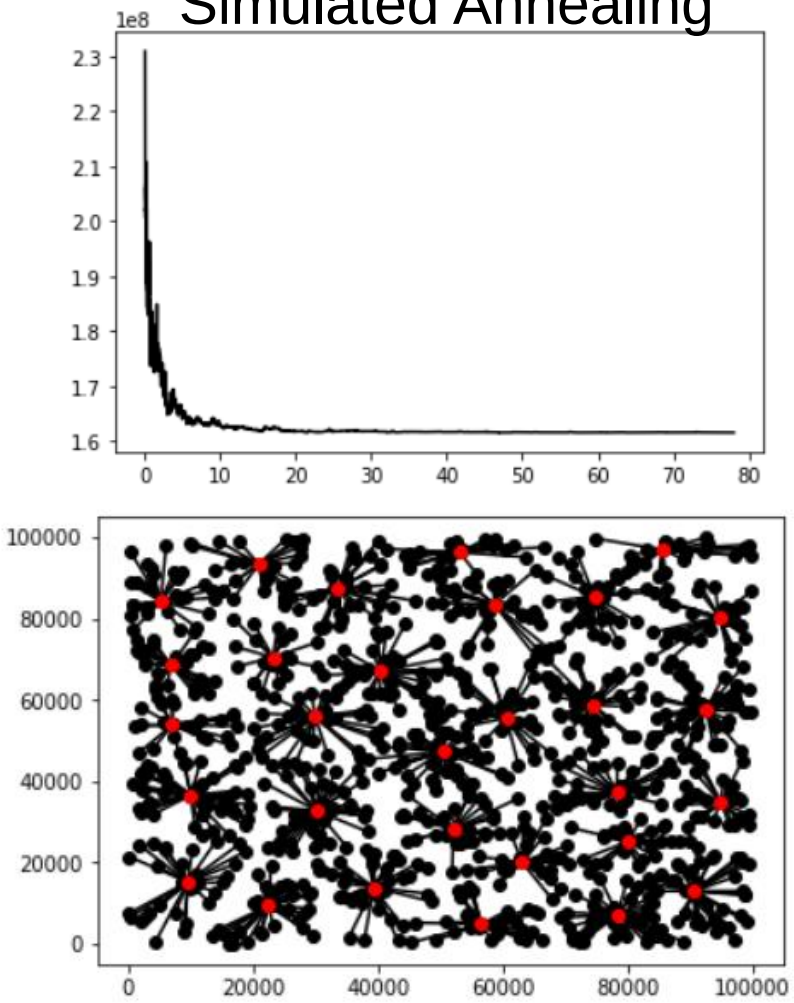
Check if the new candidate solution is better than the best solution found so far;
If yes, update the best solution found
If not, Accept the new candidate solution with a probability $e^{-\frac{\Delta E}{T}}$

P-Median Example

Hill Climbing



Simulated Annealing



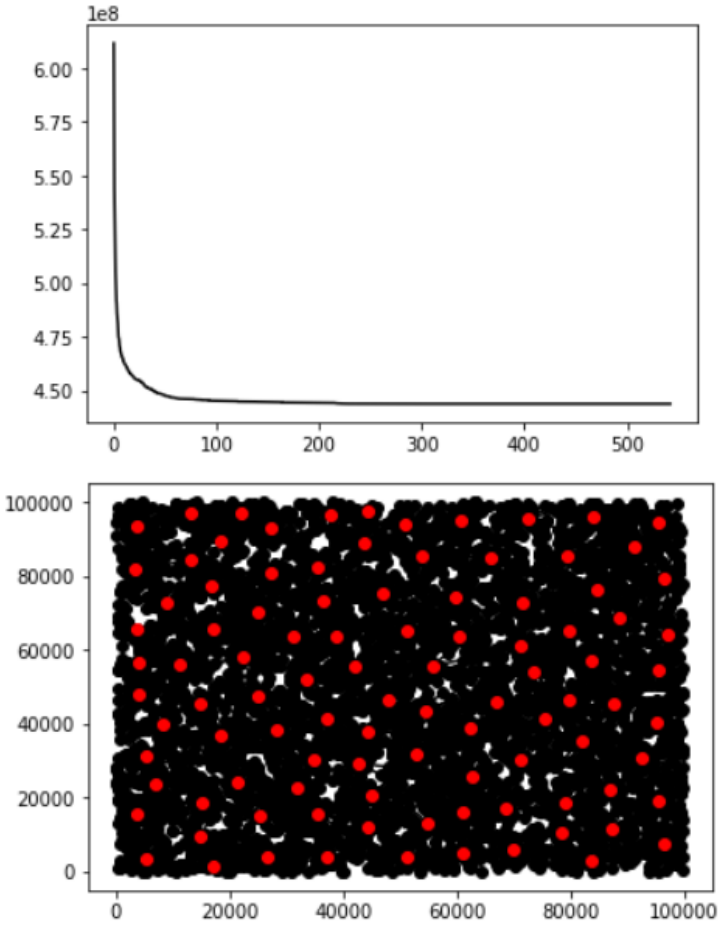
Obj. Value = 162,325,709.91 (Gap = 0.58%)

Obj. Value = 161,530,034.29 (Gap = 0.08%)

Optimum= 161,393,599.84 (321 seconds)

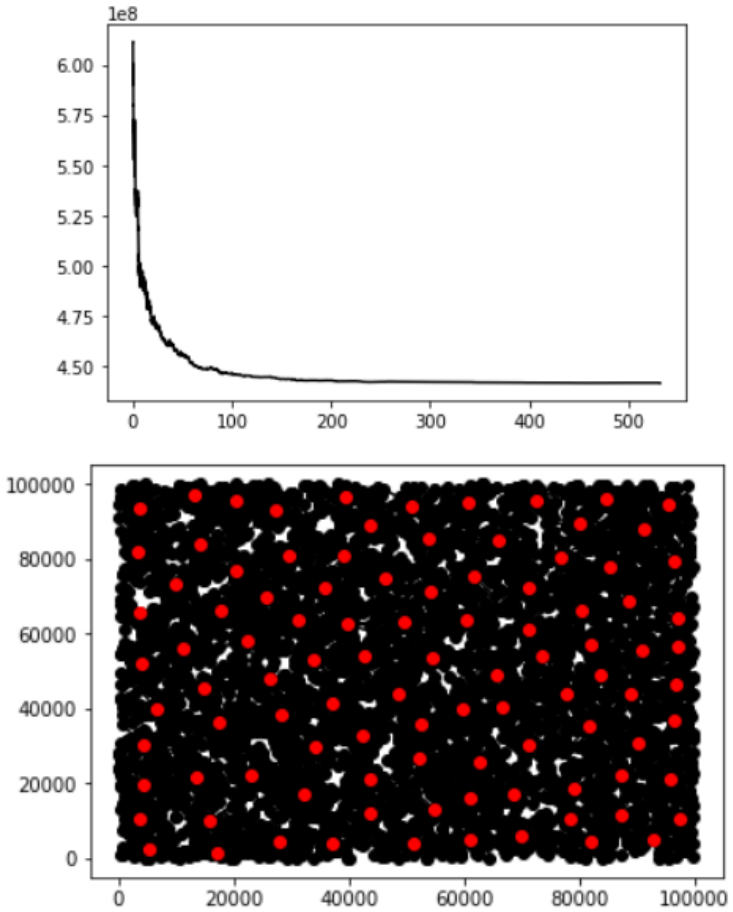
P-Median Example

Hill Climbing



Obj. Value = 443,792,460.62

Simulated Annealing



Obj. Value = 441,252,999.99

Optimum= Memory Error

TSP Example

- **Solution Representation:** Premutation Encoding
- **Move Operators:** Swap 2 locations ; Insertion ; 3-Opt
- **Replacement Procedure:** First Descent ; First Descent ; Best Descent

Number of candidate locations
n=100

TSP Example

```
random.seed(3)
iteration=0
ObjValueOpt=ObjValue
Objvalue_list=ObjValue
program_starts = time.time()
cputime_i=[0,0]
OptSolution=Solution_i
#temp=28601323 #3682176 #18601323
temp=4015897
#temp=0.1
it_T=0
#itmax=20000

tp = temp
while cputime_i[-1]<200:
#while iteration<itmax:

    iteration=iteration+1
    Solution_i=copy.deepcopy(OptSolution)

    swap_it=0

    while swap_it<no_swap:
        k_opt(Solution_i)
        swap_it=swap_it+1

    dfSolution_i=pd.DataFrame(Solution_i)
    dfSolution_i
    dflinkindex_p1=dfSolution_i
    dflinkindex_p2=dfSolution_i.shift(-1)
    dflinkindex_p2.loc[n-1]=dflinkindex_p1.loc[0]
    linkindex_p1=dflinkindex_p1.to_numpy()
    linkindex_p2=dflinkindex_p2.to_numpy()
    linkindex_p1=linkindex_p1.astype(int)
    linkindex_p2=linkindex_p2.astype(int)
    linkindex_p1=linkindex_p1.transpose()[0]
    linkindex_p2=linkindex_p2.transpose()[0]

    #Compute Objective Value
    ObjValue=sum(distancelct[linkindex_p1,linkindex_p2])
```

Initial Temperature

Move Operator

Compute Objective Value for New Solution

TSP Example

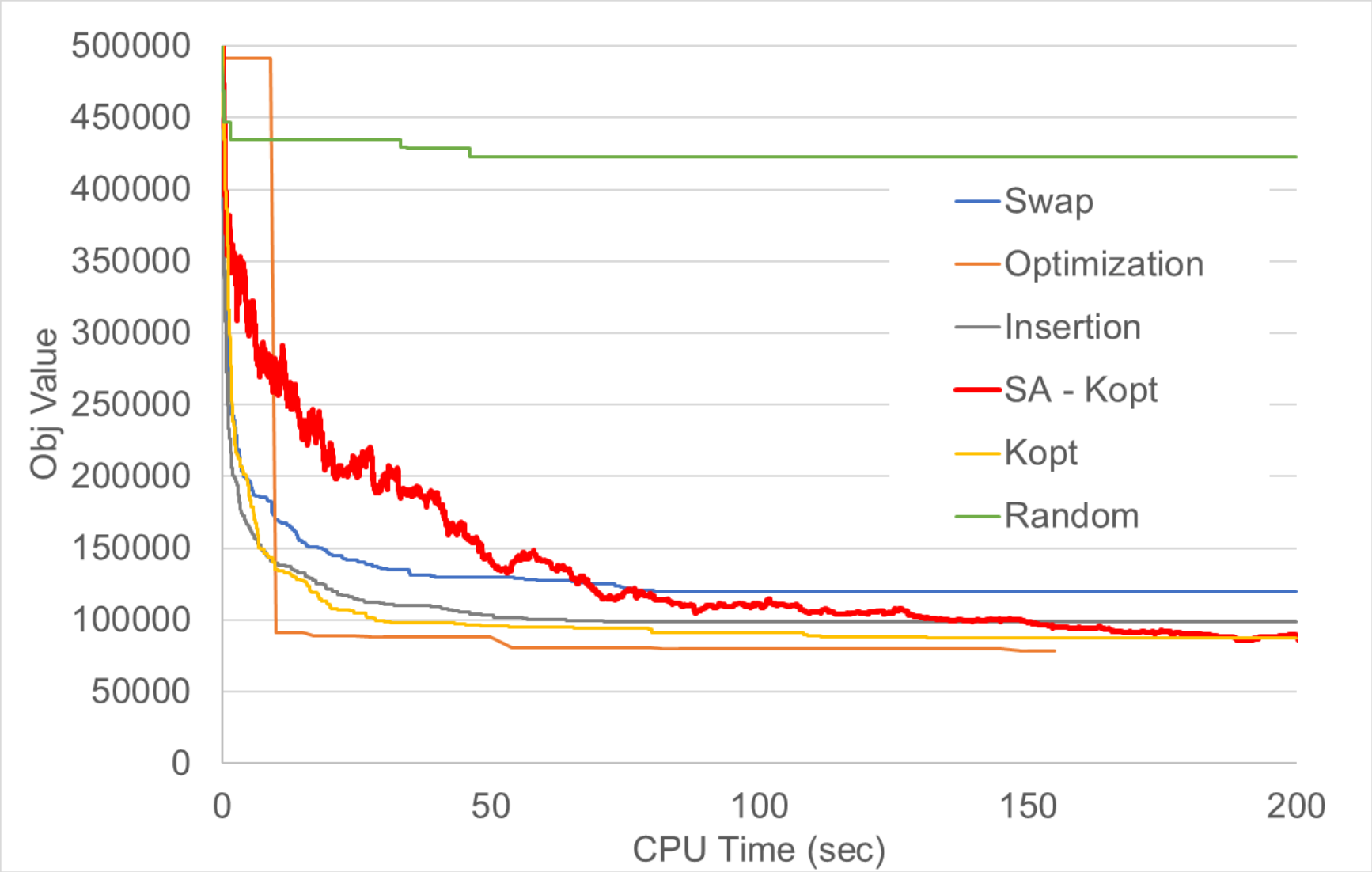
```
    #Compute Acceptance Rate
    diff = ObjValue-ObjValueOpt

    #Calculate temperature for current iteration
    if it_T>20:
        tp = temp / float(iteration + 1)
        it_T=0
    #tp = tp *0.9999
    #tp= temp / np.log(iteration + 1)
    #print(tp)

    it_T=it_T+1

    #Calculate metropolis acceptance criterion
    metropolis = np.exp(-diff / tp)

    #Update Optimal Solution
    if diff < 0 or random.random() < metropolis:
        ObjValueOpt=copy.deepcopy(ObjValue)
        OptSolution=copy.deepcopy(Solution_i)

        Objvalue_list=np.append(Objvalue_list, ObjValueOpt)
        now = time.time()
        cputime_i=np.append(cputime_i, now-program_starts)

    print(ObjValueOpt)

#Update last objective value
Objvalue_list=np.append(Objvalue_list, min(Objvalue_list))
now = time.time()
cputime_i=np.append(cputime_i, now-program_starts)
```

Update Temperature

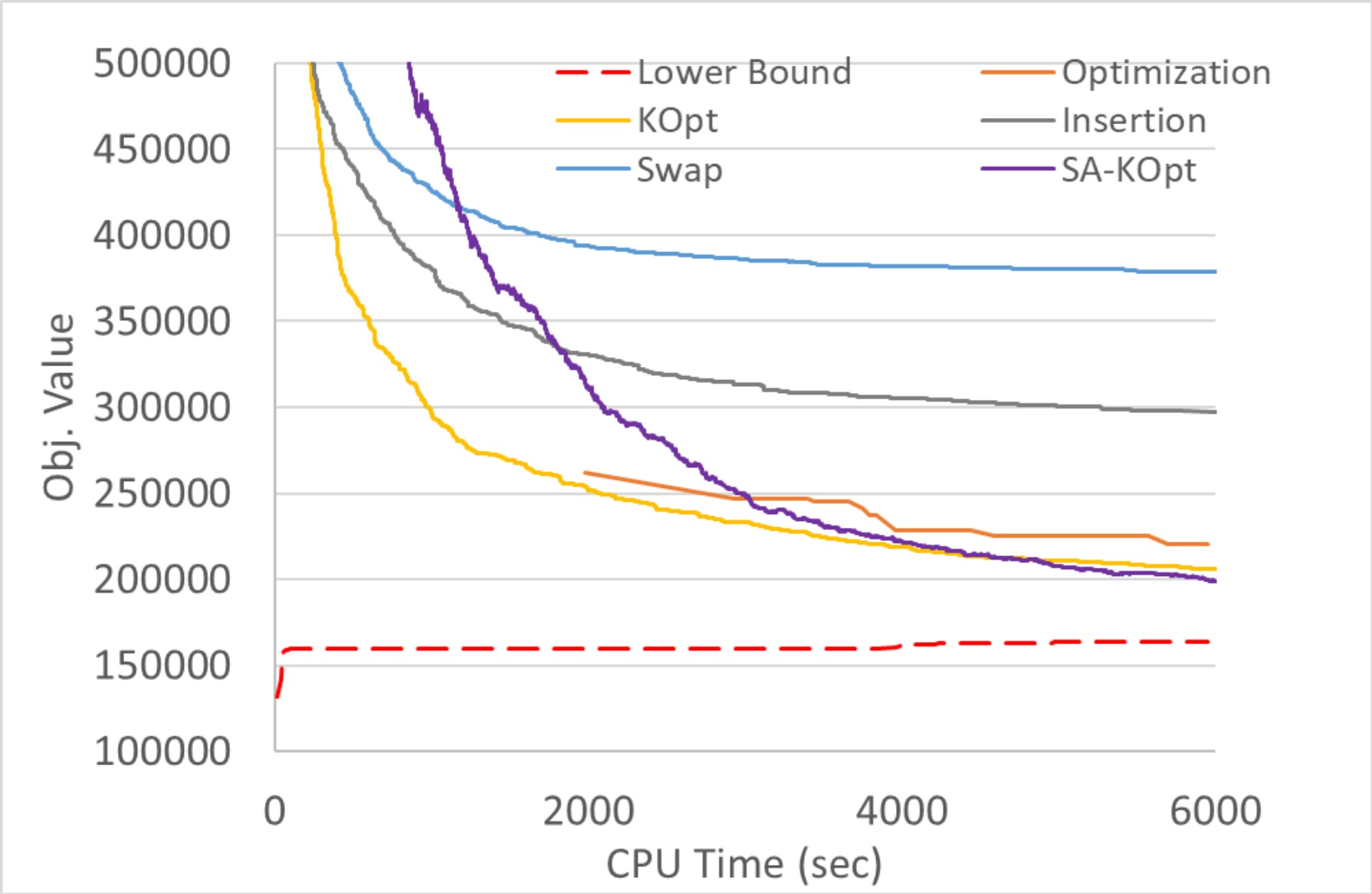
Compute Acceptance Probability

Update Solution If Random number smaller than acceptance probability

TSP Solution (n=100)



TSP Solution (n=500)



Simulated Annealing Variants

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Adaptive Cooling Schedule

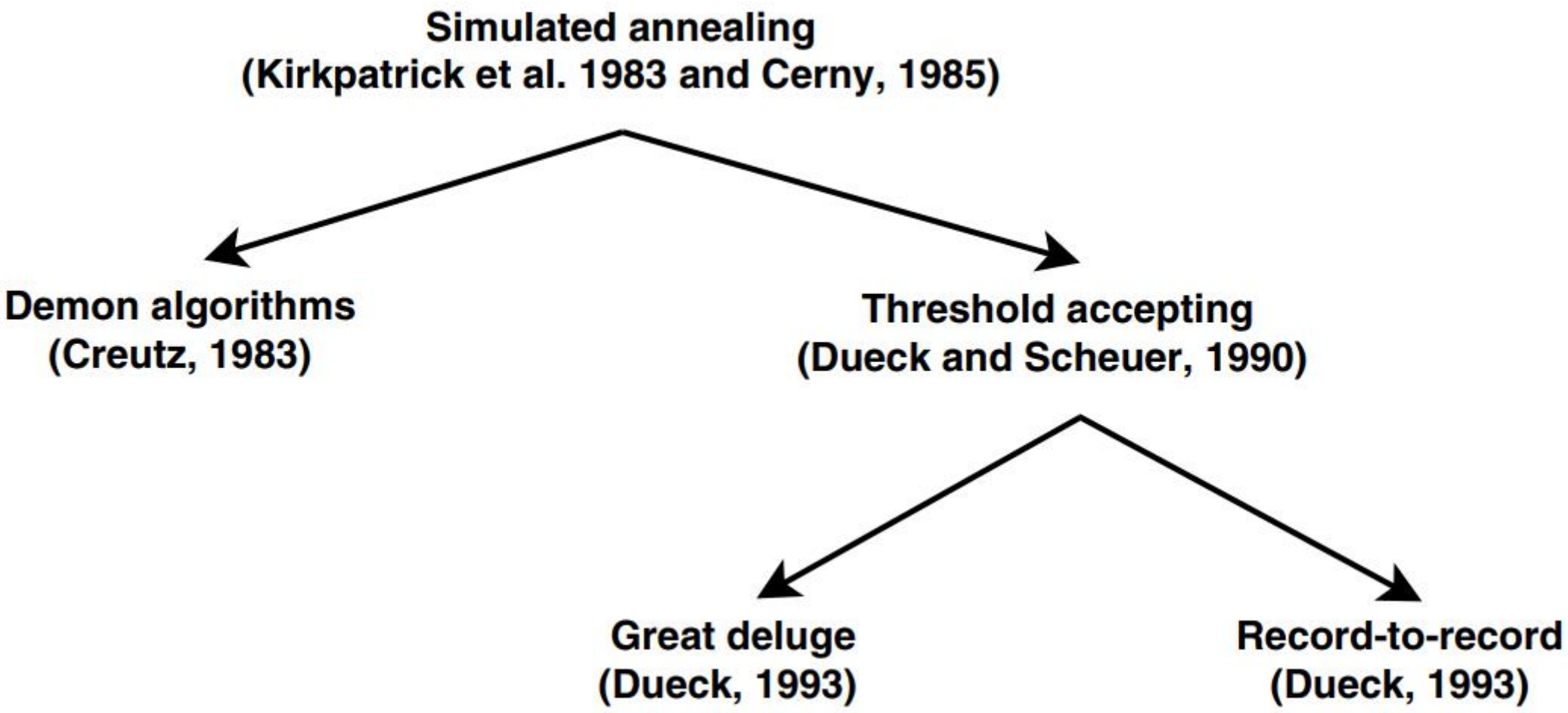
- The cooling schedules presented so far are static in the sense that they are defined completely a priori.
- In an adaptive cooling schedule, the decreasing rate is dynamic and depends on some information obtained during the search.
- Example:

$$P(\Delta E) = \begin{cases} \alpha + e^{-\frac{\Delta E}{T}} & if\ \Delta E > 0 \\ 1 & otherwise \end{cases}$$

$$\text{if } \min_{i \in M}(\Delta E_i) > 0 \text{ then } \alpha = \alpha + b \text{ else } \alpha = 0$$

Where α is a **auxiliary value** added to the metropolis probability every time the objective value is not improved in M iterations (i.e. $\min_{i \in M}(\Delta E_i) > 0$ if we are dealing with a minimization problem).

Simulated Annealing Variants



Simulated Annealing Variants

- **Threshold Accepting:** TA escapes from local optima by accepting solutions that are not worse than the current solution by more than a given **threshold** Q.
- **Record-to-Record Travel:** RRT accepts a non improving neighbor solution with an objective value less than the ***RECORD* minus a deviation δ**. *RECORD* represents the best objective value of the visited solutions during the search.
- **Great Deluge Algorithm** The inspiration of the GDA algorithm comes from the analogy that the direction a hill climber would take in a great deluge to keep his feet dry. As it rains incessantly without end, the **level of the water increases**. The algorithm never makes a move beyond the water level. The initial value of the water level is equal to the initial objective value. During the search, the value of the level is decreased **monotonically**. The decrement of the reduction is a parameter of the algorithm.
- **Demon Algorithms:** The acceptance function is based on the energy value of the demon (**credit**). The demon credit is initialized with a given value. A **nonimproved solution is accepted if the demon has more credits** than the decrease of the objective value. When a DA algorithm accepts a solution of increased objective value, **the change value of the objective is credited to the demon**. In the same manner, when a **DA algorithm accepts an improving solution, the decrease of the objective value is debited from the demon**.

Simulated Annealing Variants

- **Demon Algorithms:** The acceptance function is based on the energy value of the demon (**credit**). The demon credit is initialized with a given value. A **nonimproved solution is accepted if the demon has more credits** than the decrease of the objective value. When a DA algorithm accepts a solution of increased objective value, **the change value of the objective is credited to the demon**. In the same manner, when a **DA algorithm accepts an improving solution, the decrease of the objective value is debited from the demon**.
- **Demon Algorithm Variants**

Algorithm	Specificity
BDA	Initial demon value (upper bound): D_0
	Demon value update: if $D > D_0$, then $D = D_0$
ADA	Demon value update: annealing schedule
RBDA and RADA	Initial demon value: mean D_m
	Acceptance function: $D = D_m +$ Gaussian noise
	Demon value update: $D_m = D_m - \Delta E$