Random Sampling

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Optimization Methods

- **Exhaustive search** methods are ineffective when solving very large optimization problems
- **Exact methods of Optimization** solve complex problems without the need to exhaustively search for all possible solutions of a problem. These methos ensure that the solution obtained is the optimal one.
- Several exact methods of optimization in the previous lecture (Simplex ; Branch and Bound; Dynamic Programming ; etc)
- However, as the size of your problem grows, the **computation requirements** to solve optimization increases considerably.
- In many instance, the solution space for solutions is so large that exact methods of optimization cannot even find feasible solutions for a problem.
- In those situations using exact methods of optimization is impractical

Exact Methods - CPU Performance

P-Median				**Travelling Salesman**			**Vehicle Routing**				
n	**Fac.**	**CPU Time**	**Opt. Value**	**n**	**CPU Time**	**Opt. Value**	**n**	**Veh.**	**Cap.**	**CPU Time**	**Opt. Value**
10	1	0.12 s	8061276	10	1.29 s	34993	10	2	180	1.22	32846
20	3	0.16	9814108	20			20	3	230	697	58996
25	4	0.14 s	9359061	25	1.21 s	39224	25	4	250	>6000	64362 **(11.2%)**
50	8	0.17 s	9870230	50	49 s	57546	-	-	-	-	-
75	12	0.25 s	15617651	75	91 s	70395	-	-	-	-	-
100	15	0.32 s	18954163	100	178 s	78357	-	-	-	-	-
200	20	1.25 s	35825600	200	2625 s	105404	-	-	-	-	-
500	30	11.18 s	74794047	500	>6000 s	220704 **(25.9%)**	-	-	-	-	-
1000	30	147.61 s	161388969	-	-	-	-	-	-	-	-
5000	100	>6000	**No Solution Found**	-	-	-	-	-	-	-	-

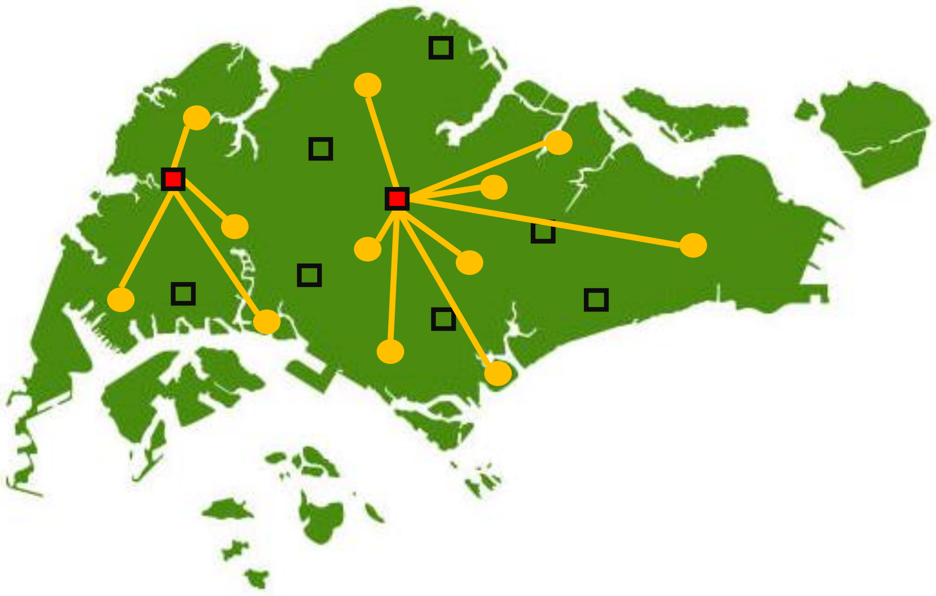
Random Sampling

- The most “naïve” metaheuristic approach consists on randomly sampling solutions from the solution space.

1. **Initialize**: Generate random initial solution, $p_{best} = p_{initial}$
2. **While** (termination criteria is not met, e.g. CPU time)
    3. Create random candidate solution p_{new}
    4. If p_{new} is better than p_{best}, than p_{best}= p_{new}
    5. Go back to 2, until termination criteria is met

P-Median Example

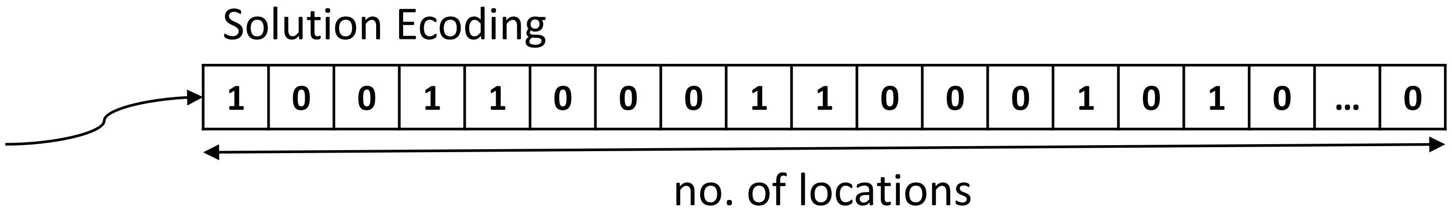
- Location planning involves specifying the physical position of facilities that provide demanded services.



Number of candidate locations
n=100

Number of locations to open
fac=15

Random Sampling: Randomly generate a binary vector at each iteration



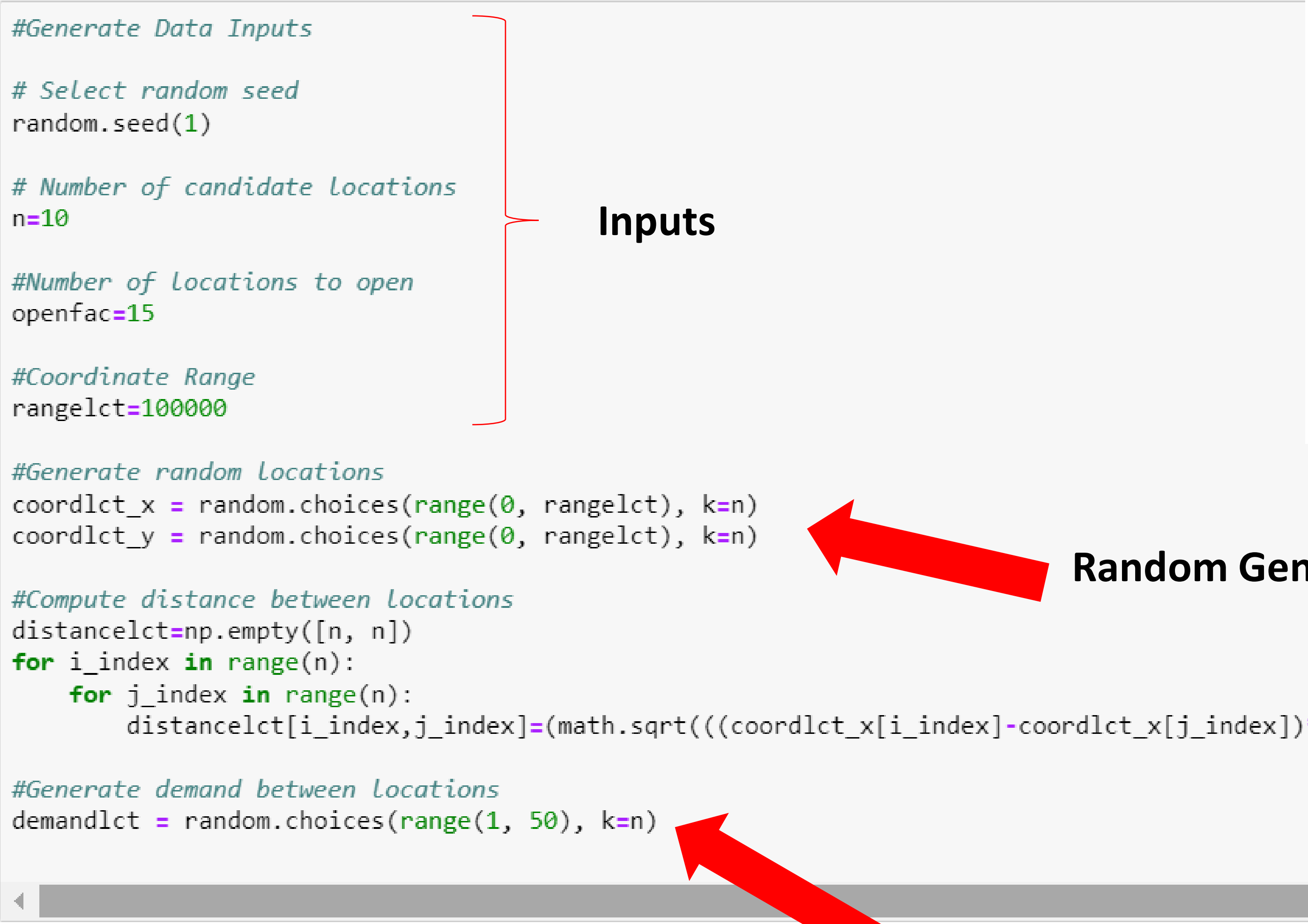
Solution Ecoding

1	0	0	1	1	0	0	0	1	1	0	0	0	1	0	1	0	…	0

no. of locations

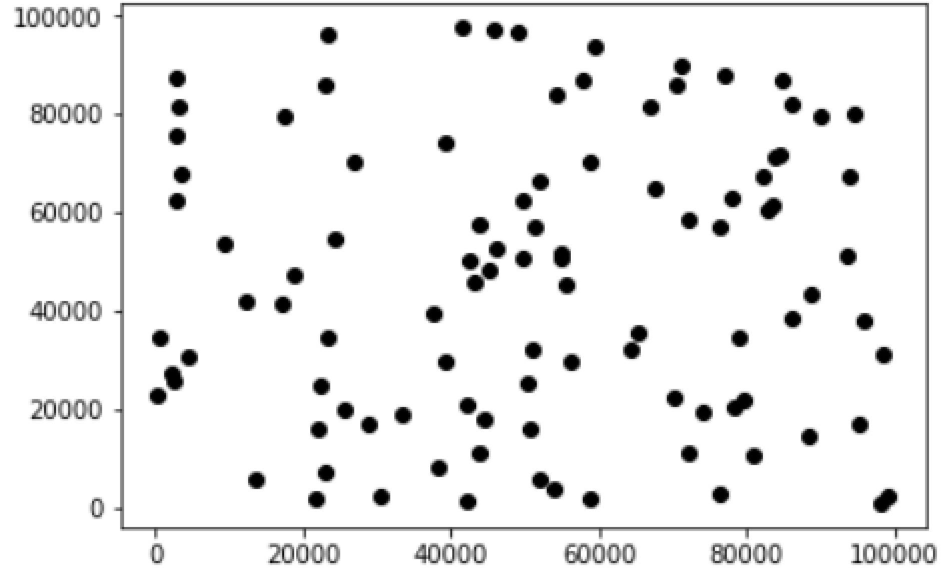
P-Median – Generate Instance

Inputs



```
#Generate Data Inputs

# Select random seed
random.seed(1)

# Number of candidate locations
n=10

#Number of locations to open
openfac=15

#Coordinate Range
rangelct=100000

#Generate random locations
coordlct_x = random.choices(range(0, rangelct), k=n)
coordlct_y = random.choices(range(0, rangelct), k=n)

#Compute distance between locations
distancelct=np.empty([n, n])
for i_index in range(n):
    for j_index in range(n):
        distancelct[i_index,j_index]=(math.sqrt(((coordlct_x[i_index]-coordlct_x[j_index])**2) +((coordlct_y[i_index]-coordlct_y

#Generate demand between locations
demandlct = random.choices(range(1, 50), k=n)
```

Inputs

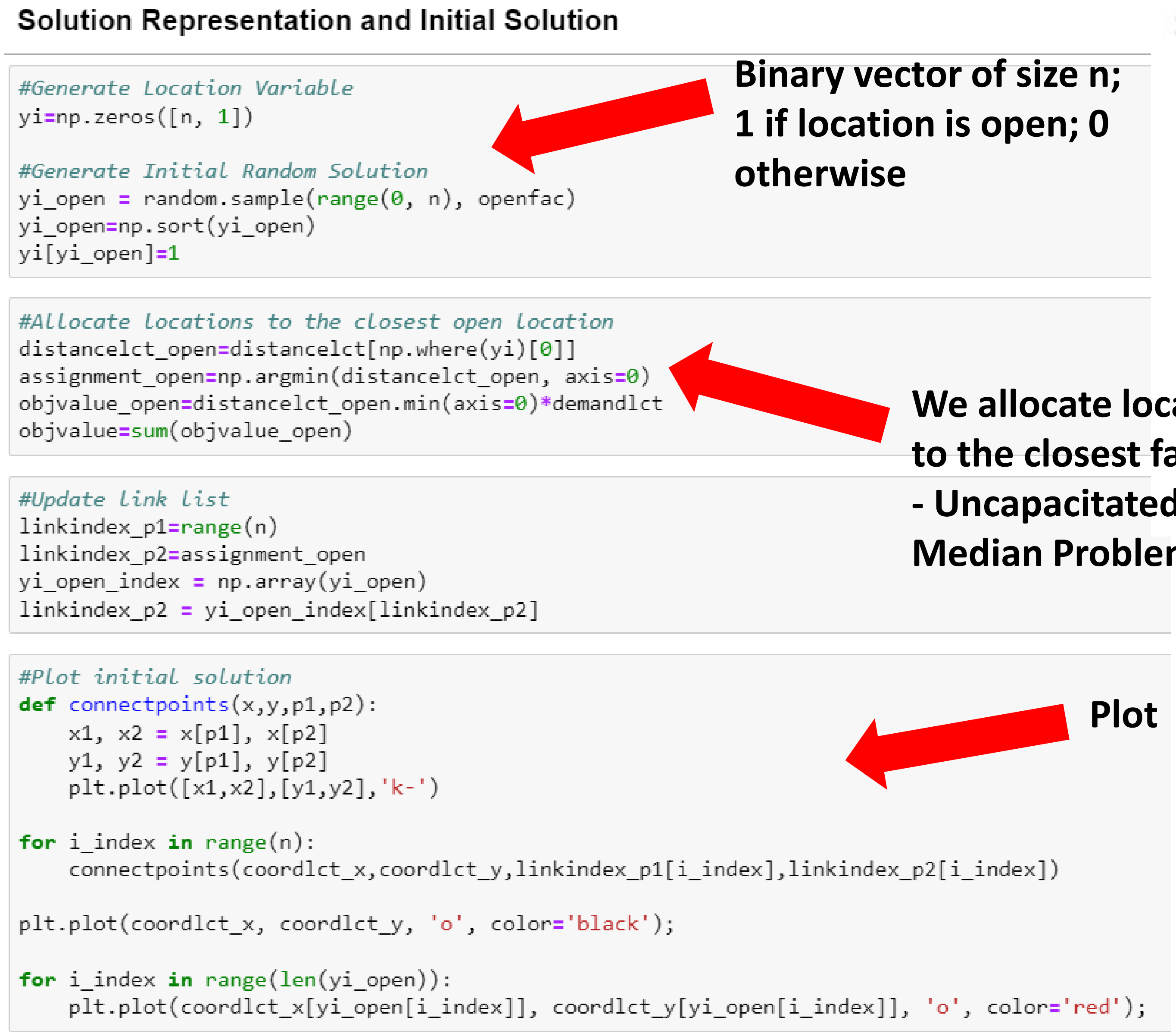


Random Generation of Locations

Array i,j of distances between locations

Demand for each location is generated randomly

P-Median – Initial Solution



Solution Representation and Initial Solution

```
#Generate Location Variable
yi=np.zeros([n, 1])

#Generate Initial Random Solution
yi_open = random.sample(range(0, n), openfac)
yi_open=np.sort(yi_open)
yi[yi_open]=1
```

Binary vector of size n; 1 if location is open; 0 otherwise

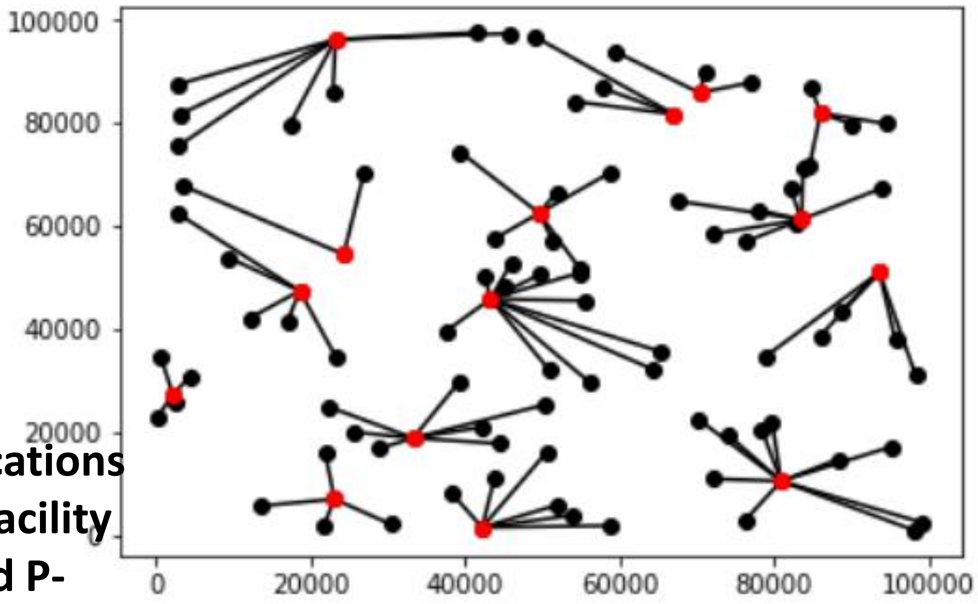
```
#Allocate locations to the closest open location
distancelct_open=distancelct[np.where(yi)[0]]
assignment_open=np.argmin(distancelct_open, axis=0)
objvalue_open=distancelct_open.min(axis=0)*demandlct
objvalue=sum(objvalue_open)
```

We allocate locations to the closest facility - Uncapacitated P-Median Problem

```
#Update link list
linkindex_p1=range(n)
linkindex_p2=assignment_open
yi_open_index = np.array(yi_open)
linkindex_p2 = yi_open_index[linkindex_p2]
```

```
#Plot initial solution
def connectpoints(x,y,p1,p2):
    x1, x2 = x[p1], x[p2]
    y1, y2 = y[p1], y[p2]
    plt.plot([x1,x2],[y1,y2],'k-')

for i_index in range(n):
    connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

plt.plot(coordlct_x, coordlct_y, 'o', color='black');

for i_index in range(len(yi_open)):
    plt.plot(coordlct_x[yi_open[i_index]], coordlct_y[yi_open[i_index]], 'o', color='red');
```

Plot



P-Median – Random Sampling Search Procedure

Random Sampling

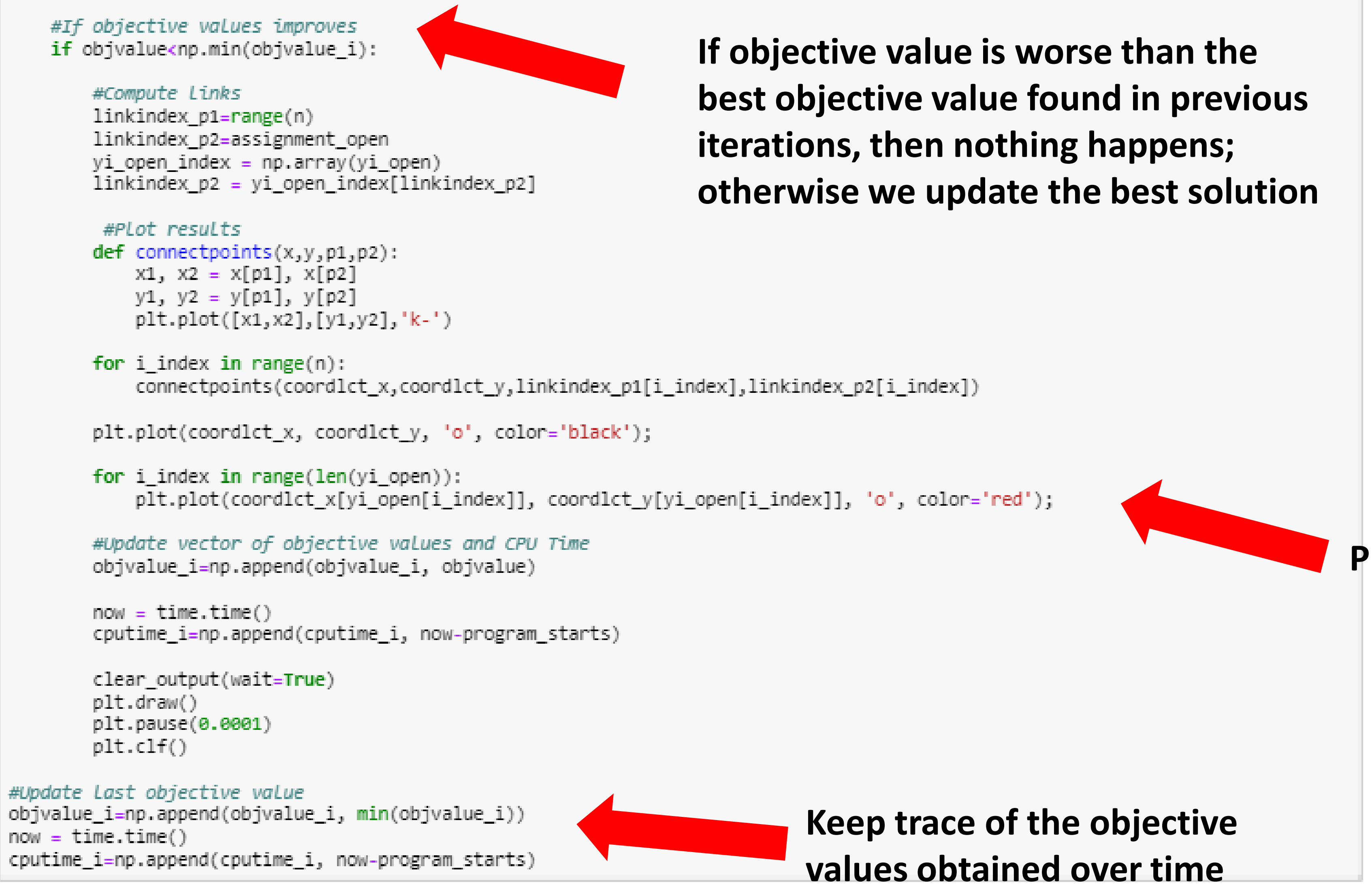
Random Algorithm

```
random.seed(3)
iteration=0
objvalue_i=objvalue
program_starts = time.time()
cputime_i=0

while iteration<100000:
    yi=np.zeros([n, 1])

    # Start random Procedure
    yi_open = random.sample(range(0, n), openfac)
    yi_open=np.sort(yi_open)
    yi[yi_open]=1

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

    iteration=iteration+1
```

Randomly select some locations to open

P-Median – Random Sampling Search Procedure

Random Sampling



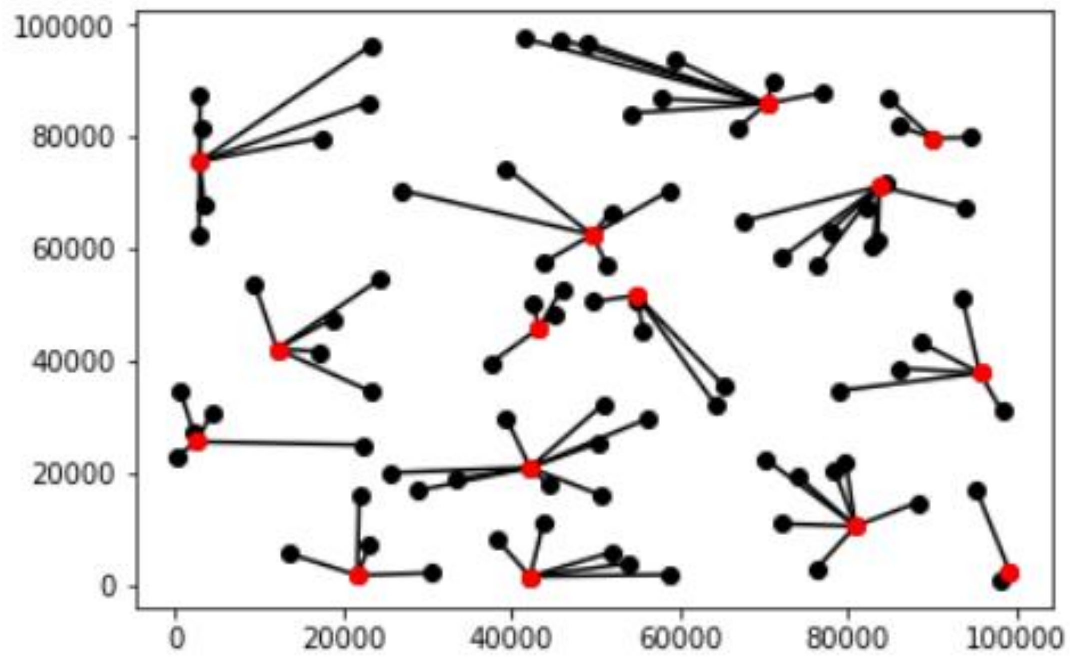
```
    #If objective values improves
    if objvalue<np.min(objvalue_i):

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

         #Plot results
        def connectpoints(x,y,p1,p2):
            x1, x2 = x[p1], x[p2]
            y1, y2 = y[p1], y[p2]
            plt.plot([x1,x2],[y1,y2],'k-')

        for i_index in range(n):
            connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

        plt.plot(coordlct_x, coordlct_y, 'o', color='black');

        for i_index in range(len(yi_open)):
            plt.plot(coordlct_x[yi_open[i_index]], coordlct_y[yi_open[i_index]], 'o', color='red');

        #Update vector of objective values and CPU Time
        objvalue_i=np.append(objvalue_i, objvalue)

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

        clear_output(wait=True)
        plt.draw()
        plt.pause(0.0001)
        plt.clf()

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

If objective value is worse than the best objective value found in previous iterations, then nothing happens; otherwise we update the best solution

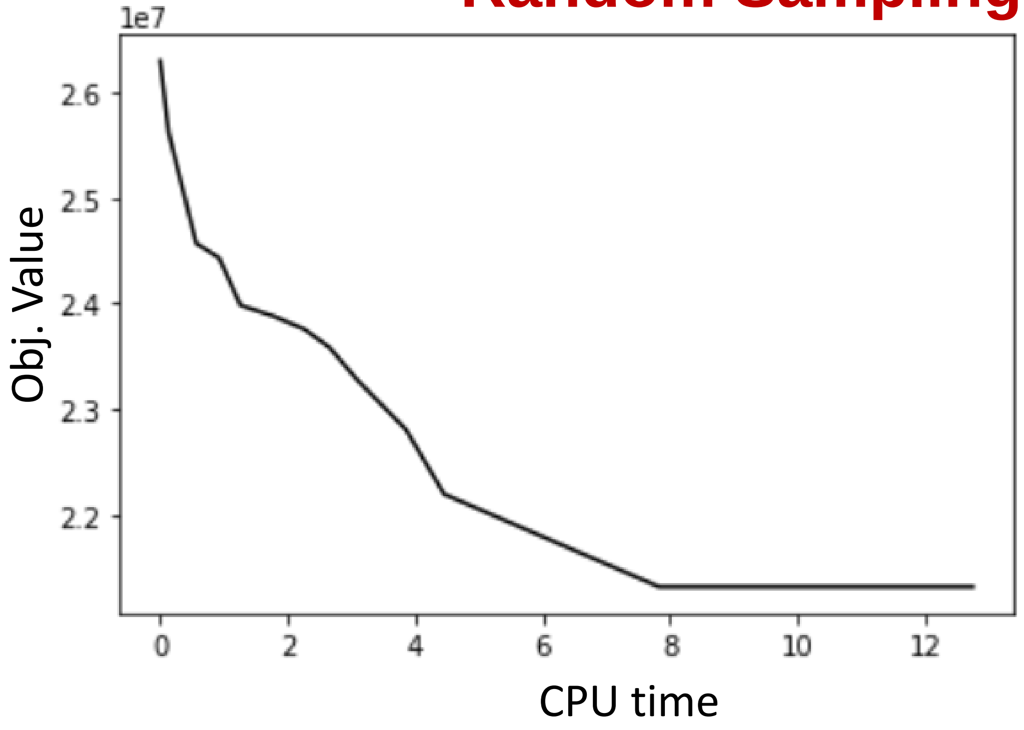
Plot

Keep trace of the objective values obtained over time

P-Median – Random Sampling Solution (n=100 ; fac=15)



Random Sampling



Obj. Value = 21,321,318.07 (Gap = 11.1%)

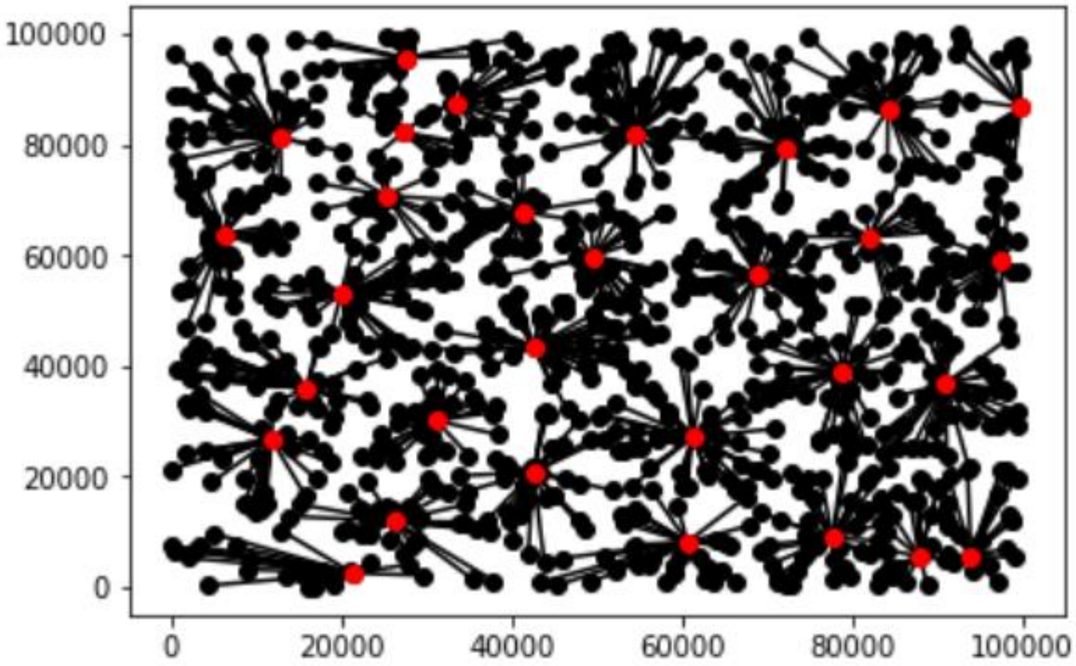
Optimum= 18,954,163.57 (0.7 seconds)

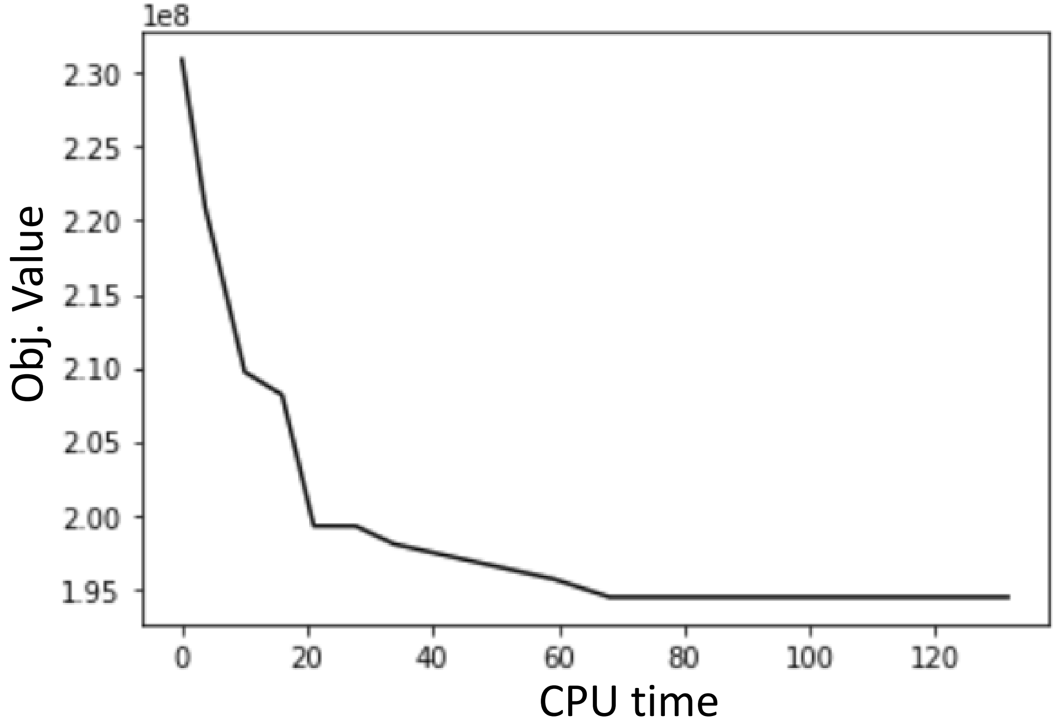
What is the probability of finding the optimal solution through random sampling?

$$\frac{1}{2^{100}} = 1 \text{ in } 1.27 \times 10^{21}$$

P-Median – Random Sampling Solution (n=1000 ; fac=30)

Random Sampling

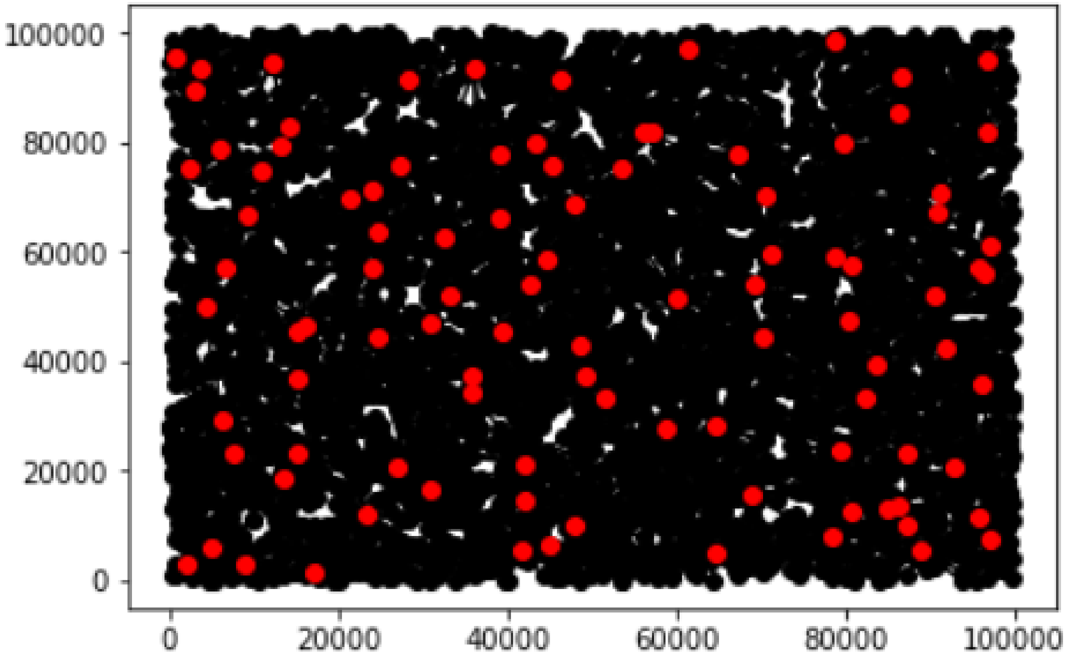


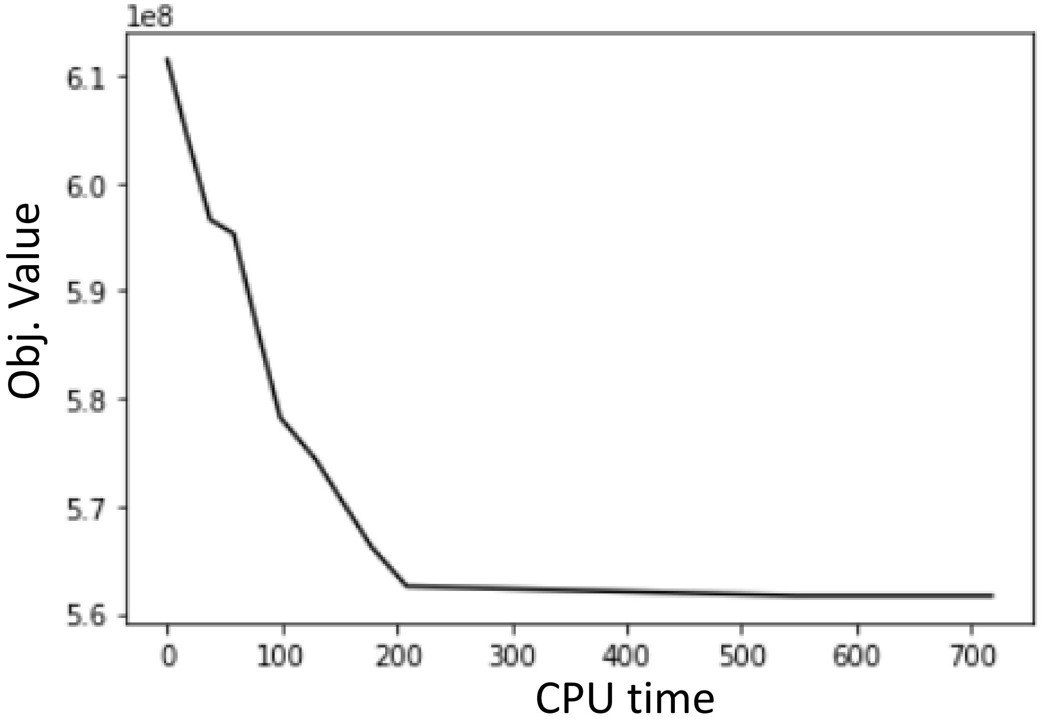


Obj. Value = 194,419,346.65 (Gap = 16.9%)

Optimum= 161,393,599.84 (321 seconds)

P-Median – Random Sampling Solution (n=5000 ; fac=100)





Obj. Value = 561,681,400.18
Optimum=?

Needle in the Haystack

- Random Sampling keeps randomly generating new candidate solutions
- If we imagine a large space, with, say, millions of points, it is clear that it will take very long until we find the optimal solution for the problem
- **It may even take extremely long to find anything remotely good**
- Random Sampling is like exhaustive search – not really used in optimization

Source: https://thriveglobal.com/stories/is-purpose-the-needle-in-the-haystack/

Using random sampling is like finding a needle in the haystack

Introduction to Local Search

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

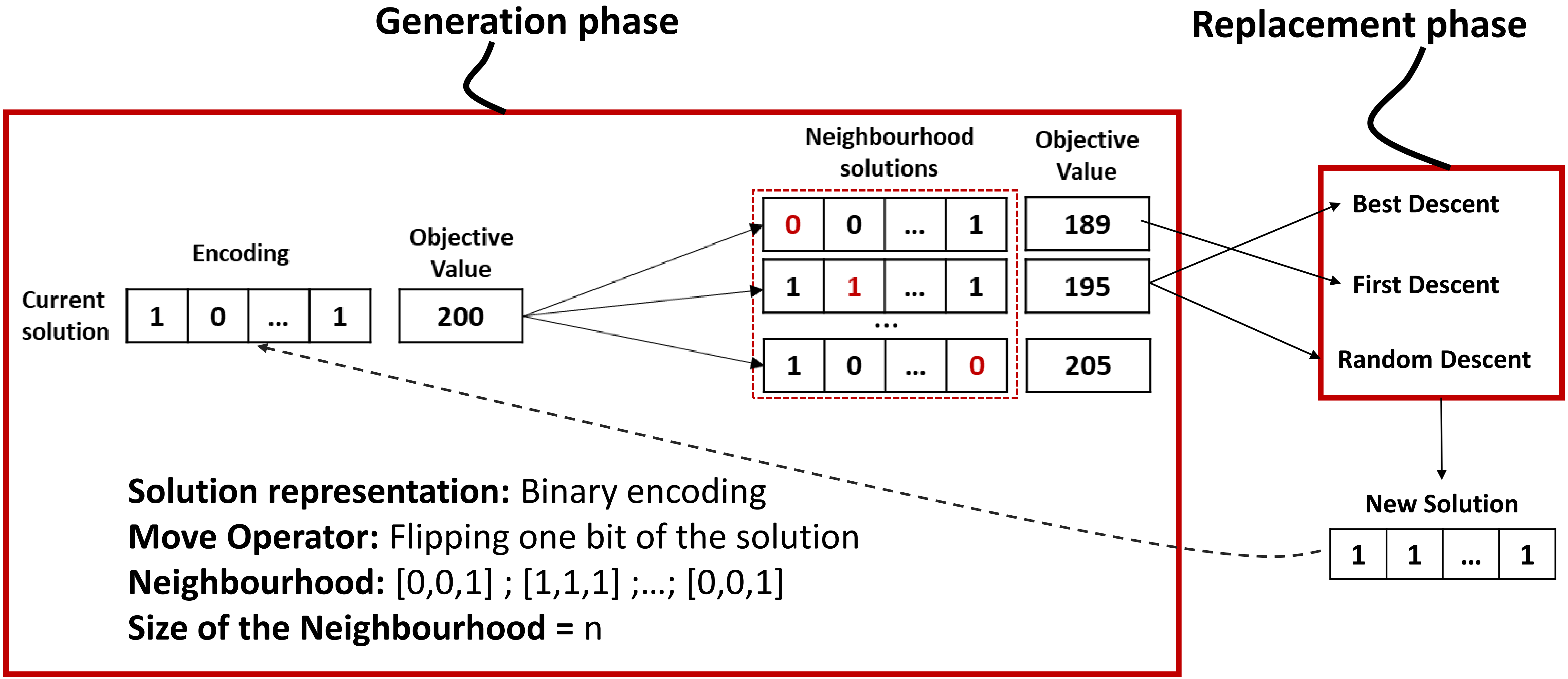
Engineering Systems and Design

Local Search

- **Local Search** is the oldest and simplest metaheuristics method. Also often designated as **hill climbing** ; **steepest descent**; **iterative improvement**, etc.
- At each iteration, the heuristic replaces the current solution by a neighbour that improves the objective function
- A **neighbour** solution can be reached by applying a **move operator**
- The search stops when not better solutions can be found

Source:
https://buildingai.elementsofai.com/Getting-started-with-AI/hill-climbing

Local Search Algorithm



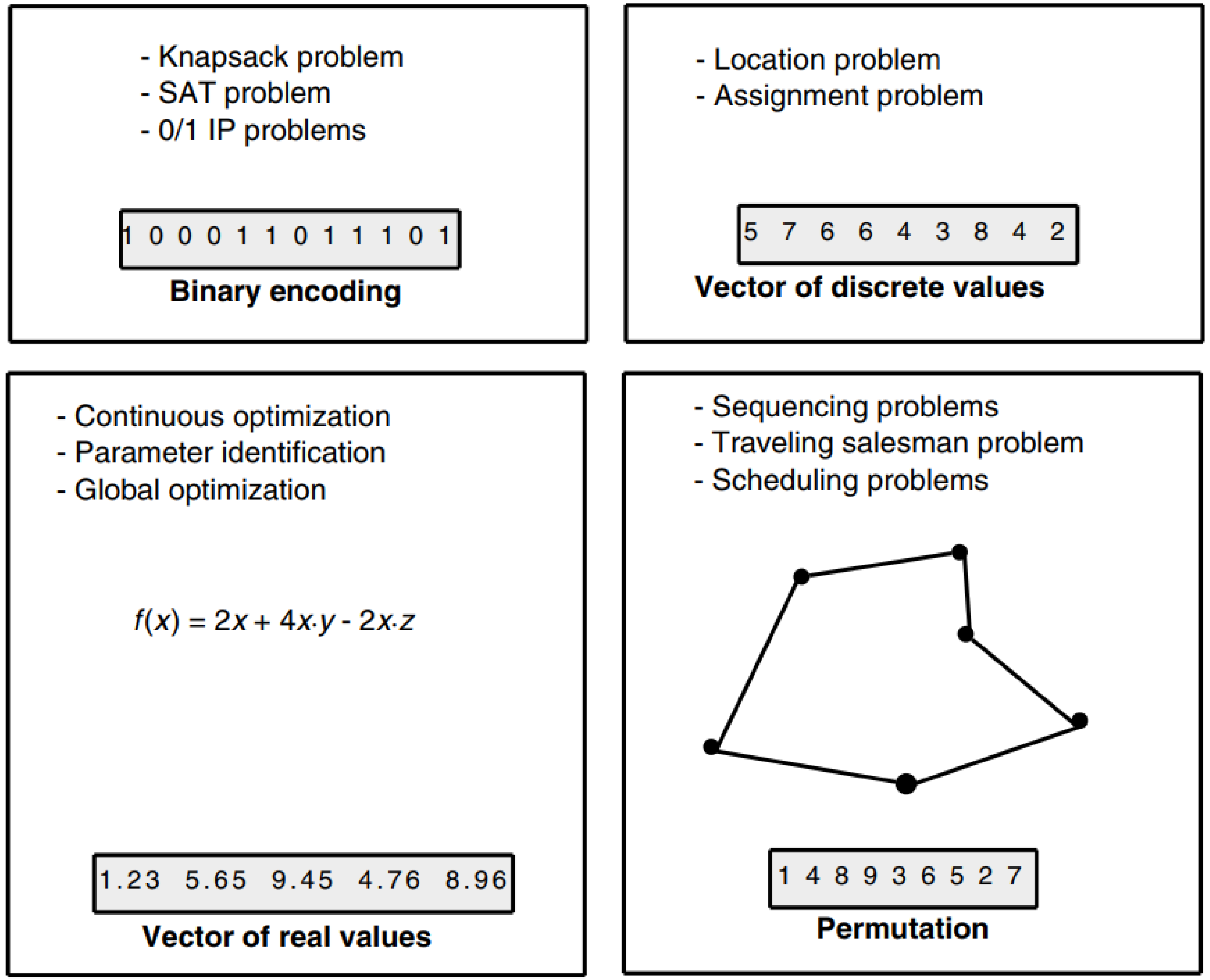
Solution representation: Binary encoding
Move Operator: Flipping one bit of the solution
Neighbourhood: [0,0,1] ; [1,1,1] ;…; [0,0,1]
Size of the Neighbourhood = n

Replacement Phase

- **First Descent:** This strategy consists in choosing the first improving neighbour that is better than the current solution. Then, an improving neighbour is immediately selected to replace the current solution.
- **Best Descent**: In this strategy, the best neighbour (i.e., neighbour that improves the most the cost function) is selected. The neighbourhood is evaluated in a fully deterministic manner. Hence, the exploration of the neighbourhood is exhaustive
- **Random selection**: In this strategy, a random selection is applied to those neighbours improving the current solution

Solution Representation

- Designing any iterative metaheuristic needs an **encoding** of a solution
- The encoding plays a major role in the efficiency and effectiveness of a metaheuristic procedure and constitutes an essential step in designing **any** metaheuristic.
- Many straightforward encodings may be applied for some traditional families of optimization problems. Those representations may be combined or underlying new representations.



- Knapsack problem
- SAT problem
- 0/1 IP problems

1 0 0 0 1 1 0 1 1 1 0 1

Binary encoding

- Location problem
- Assignment problem

5 7 6 6 4 3 8 4 2

Vector of discrete values

- Continuous optimization
- Parameter identification
- Global optimization

$f(x) = 2x + 4x.y - 2x.z$

1.23 5.65 9.45 4.76 8.96

Vector of real values

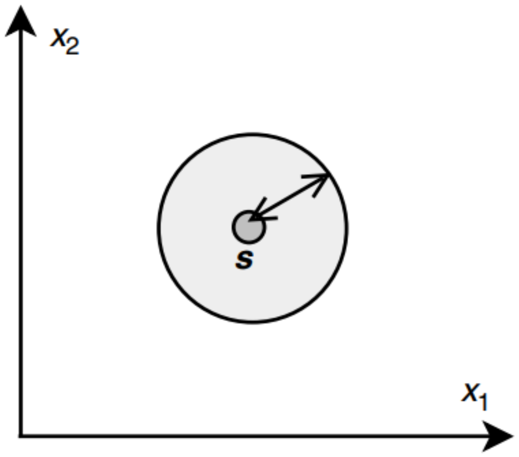
- Sequencing problems
- Traveling salesman problem
- Scheduling problems

1 4 8 9 3 6 5 2 7

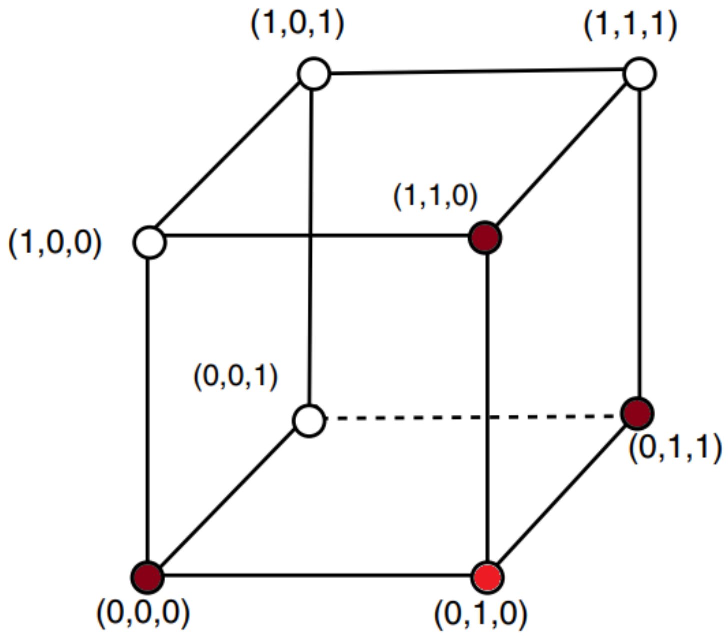
Permutation

Neighbourhood and Move Operator

- The definition of the neighbourhood is a required common step for the design of any Local Search metaheuristic. It plays a crucial role in the performance
- **A neighbour solution is obtained by the application of a search operator** that performs a small perturbation to the solutions



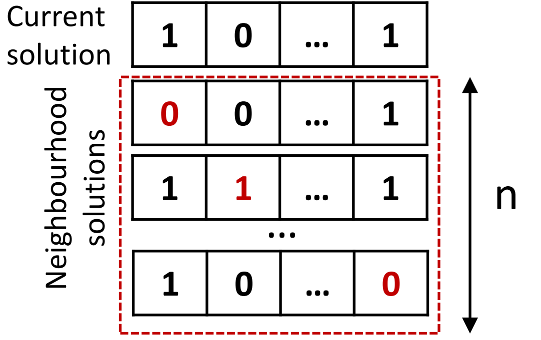
The circle represents the neighborhood of *s* in a continous problem with two dimensions.



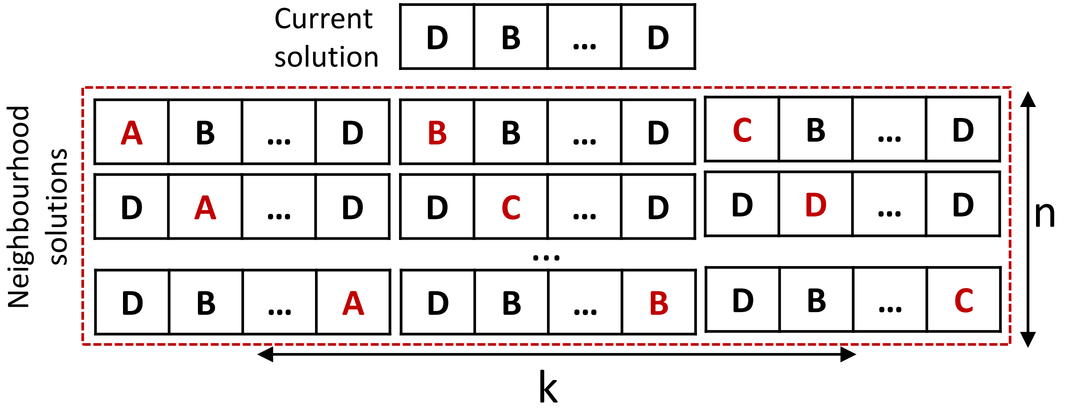
- Nodes of the hypercube represent solutions of the problem.
- The neighbors of a solution (e.g., (0,1,0)) are the adjacent nodes in the graph.

Size of the neighbourhood

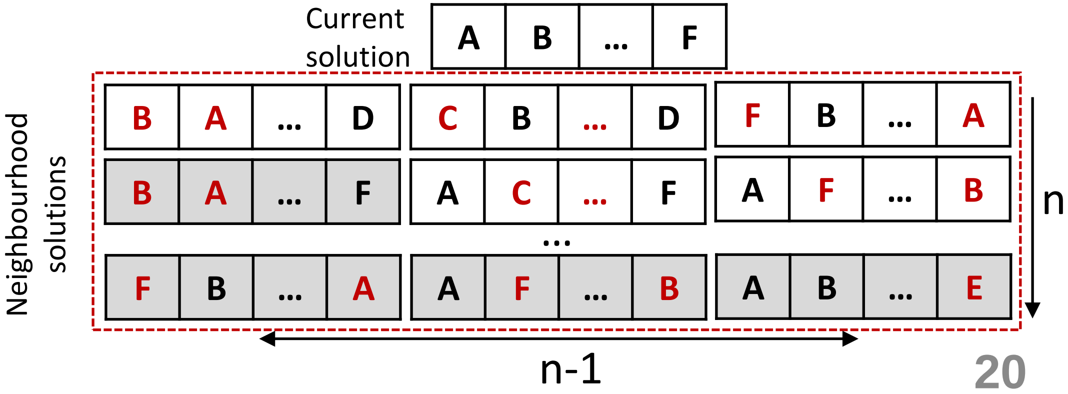
- **Binary encoding:** the neighbourhood of a binary solution consists in flipping one bit of the solution. For a binary vector of size n, the **size of the neighbourhood will be n.**



- **Discrete encoding:** The neighbourhood for binary encodings may be extended to any discrete vector representation using a given alphabet (1,2,…k ; a,b,…k). For a discrete vector of size n, and alphabet with k characters, the **size of the neighbourhood will be (k-1)n.**



- **Permutation encoding**: A usual neighbourhood is based on the "swap" operator that consists in exchanging (or swapping) the location of two elements. For a permutation of size n, the **size of this neighbourhood is n(n − 1)/2**

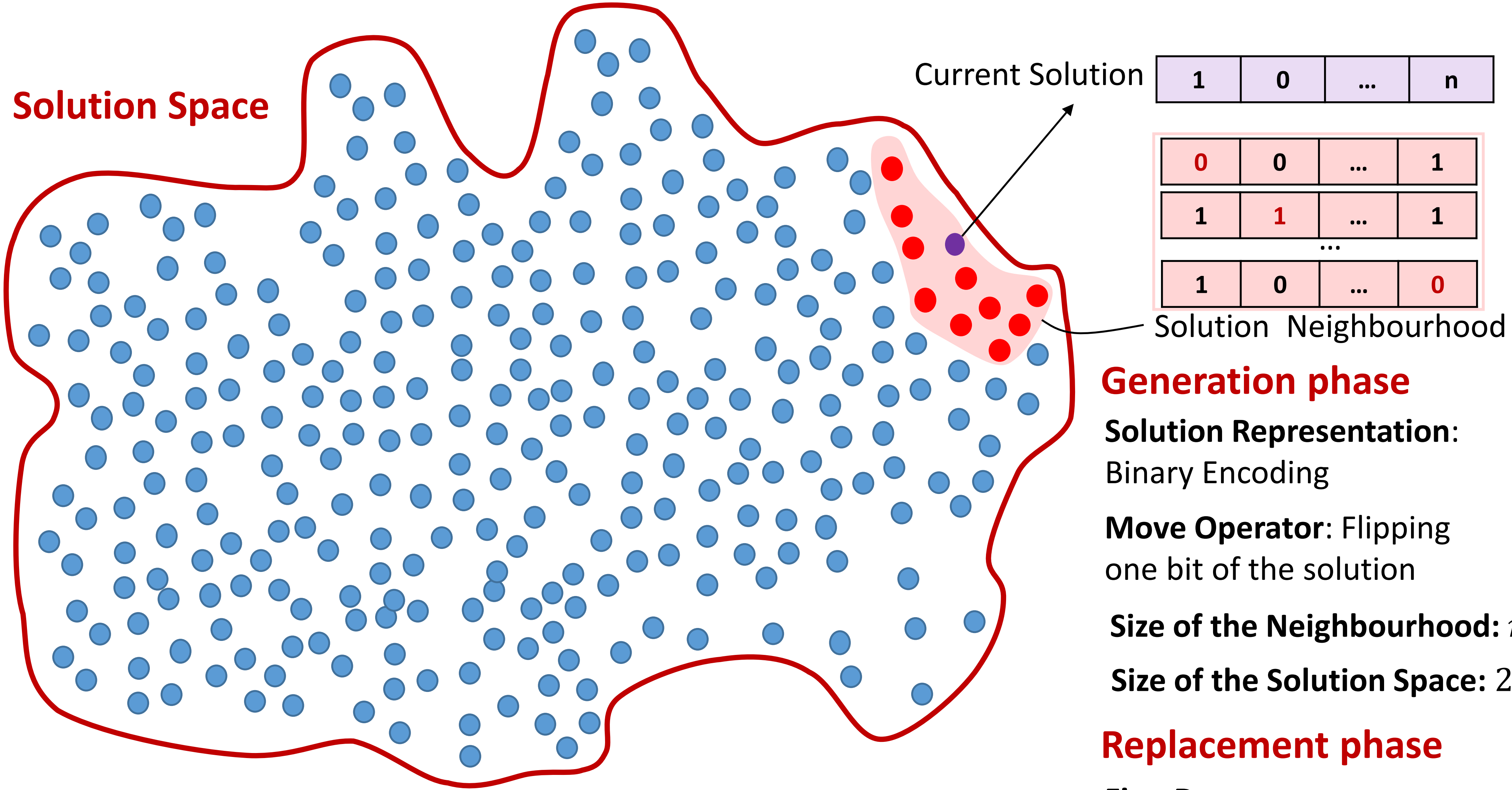


Summary – Local Search

- Local Search iteratively applies a **generation** and a **replacement** procedure;
- In the **generation phase**, a set of candidate solutions is generated from the current solution by applying a move operator;
- In the **replacement phase**, the best solution from the set of candidate solutions is selected and compared with the current solution. If the solution obtained is better than the current solution, then the current solution is replaced.

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

Summary – Local Search



Current Solution

1	0	…	n

Solution Neighbourhood

0	0	…	1
1	1	…	1
…			
1	0	…	0

Generation phase

Solution Representation: Binary Encoding

Move Operator: Flipping one bit of the solution

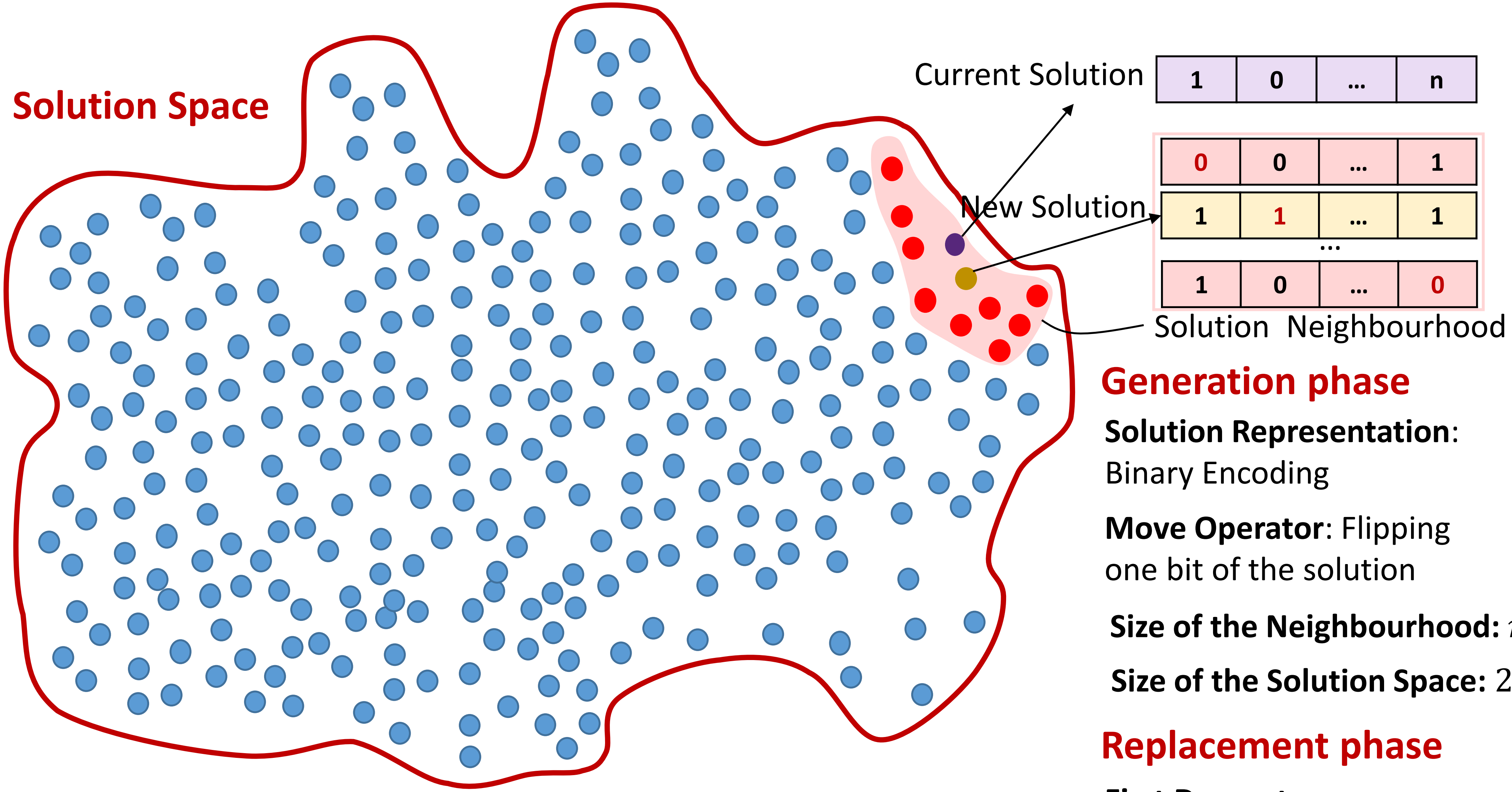
Size of the Neighbourhood: n

Size of the Solution Space: 2^n

Replacement phase

First Descent

Summary – Local Search



Generation phase

Solution Representation: Binary Encoding

Move Operator: Flipping one bit of the solution

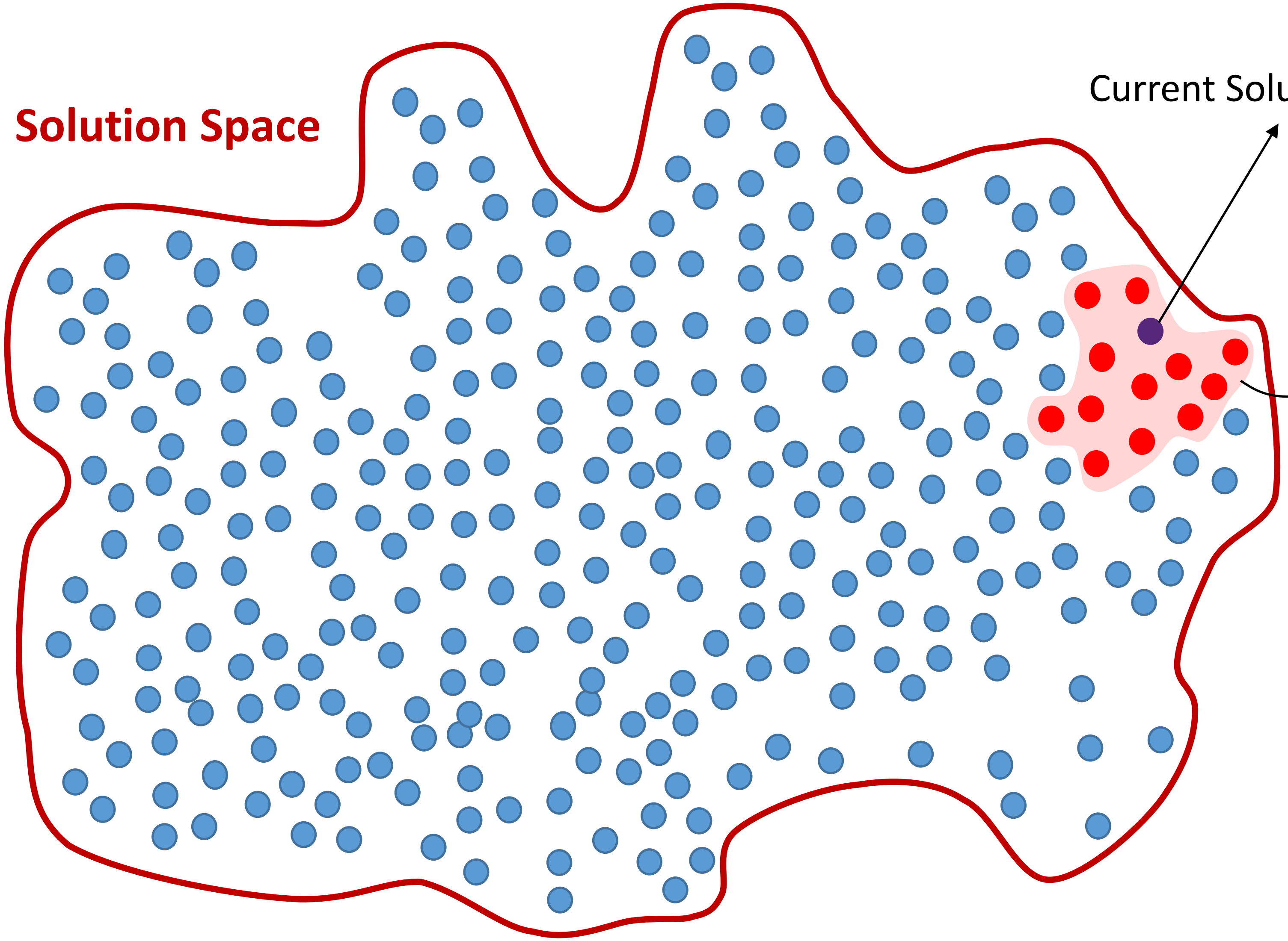
Size of the Neighbourhood: n

Size of the Solution Space: 2^n

Replacement phase

First Descent

Summary – Local Search



Current Solution

1	1	…	1

0	1	…	1
1	0	…	1
…			
1	0	…	0

Solution Neighbourhood

Generation phase

Solution Representation: Binary Encoding

Move Operator: Flipping one bit of the solution

Size of the Neighbourhood: n

Size of the Solution Space: 2^n

Replacement phase

First Descent

Local Search in Python

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

P-Median Example

- **Solution Representation:** Binary Encoding (*we could have used discrete encoding)
- **Move Operator:** Open 1 random location and close 1 random location
- **Replacement Procedure:** First Descent

Binary Encoding

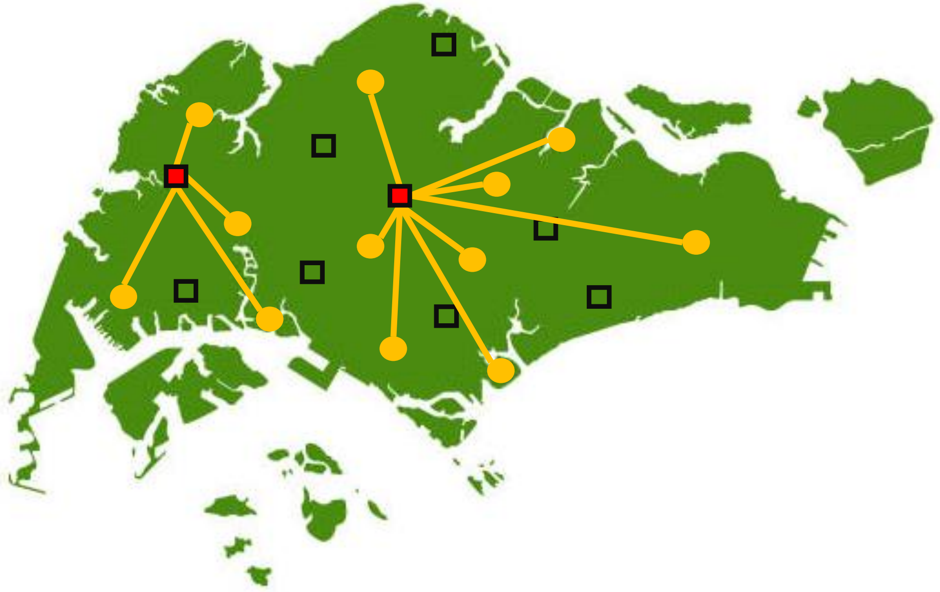
1	**0**	**…**	**n**

no. of locations

Alternative: Discrete Encoding

4	**5**	**…**	**14**

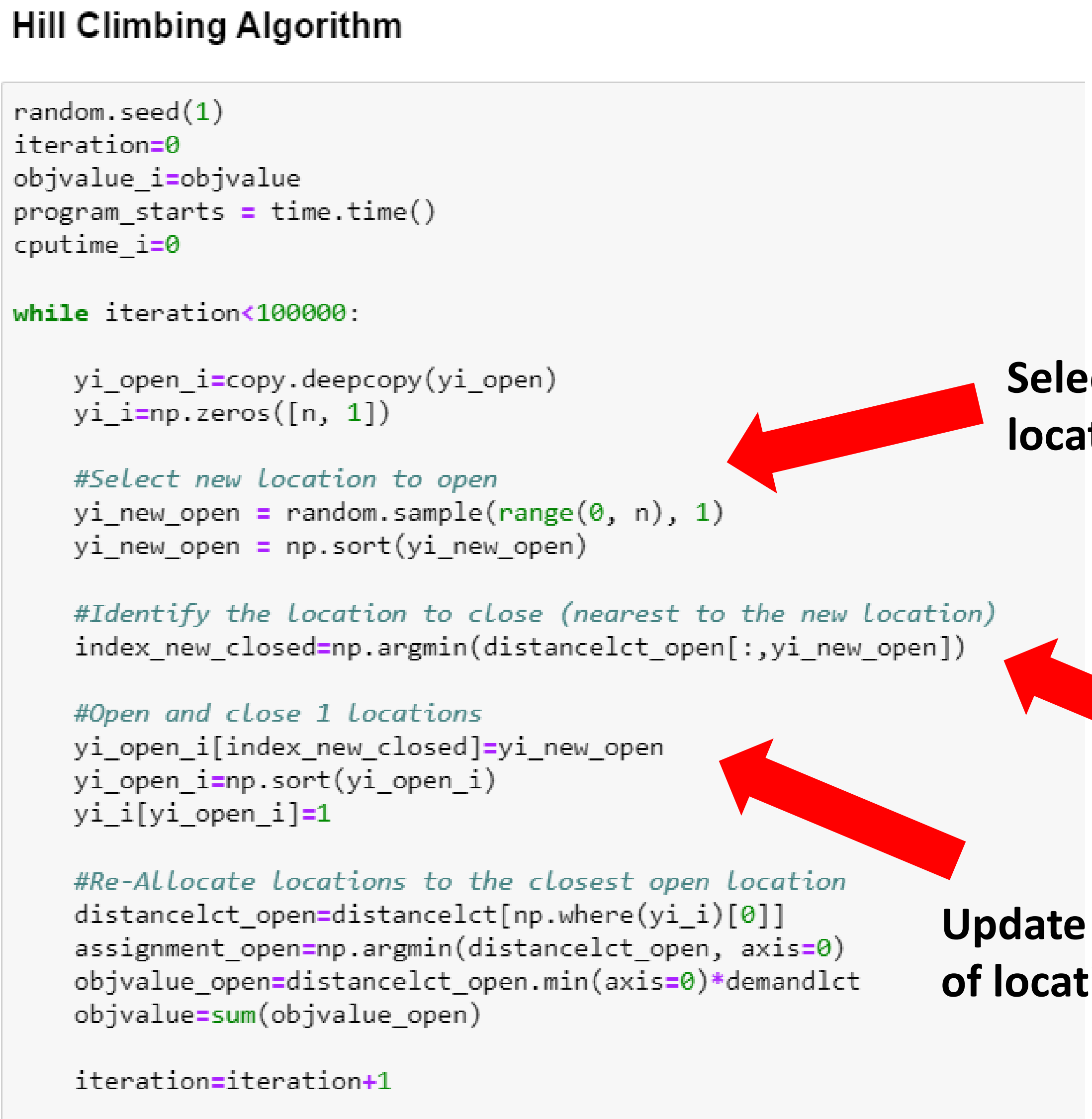
no. facilities



Number of candidate locations
n=100

Number of locations to open
fac=15

P-Median – Hill Climbing Generation Phase



Hill Climbing Algorithm

```
random.seed(1)
iteration=0
objvalue_i=objvalue
program_starts = time.time()
cputime_i=0

while iteration<100000:

    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)

    iteration=iteration+1
```

Hill Climbing

Select a random location to open

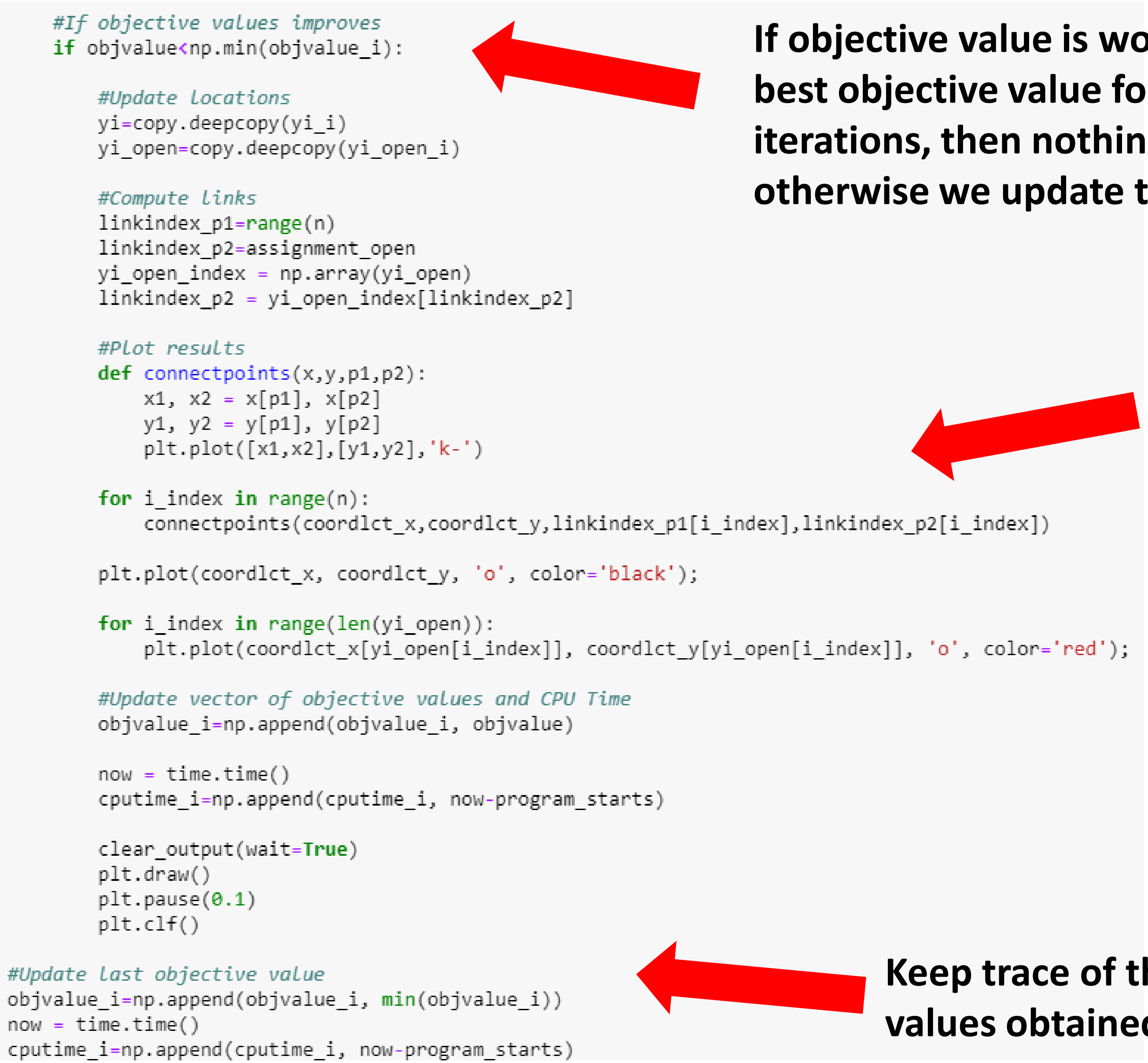
Generation Phase

(Move Operator Code)

Identify the nearest open location to close

Update binary vector of locations

P-Median – Hill Climbing Replacement Phase



```
    #If objective values improves
    if objvalue<np.min(objvalue_i):

        #Update locations
        yi=copy.deepcopy(yi_i)
        yi_open=copy.deepcopy(yi_open_i)

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

        #Plot results
        def connectpoints(x,y,p1,p2):
            x1, x2 = x[p1], x[p2]
            y1, y2 = y[p1], y[p2]
            plt.plot([x1,x2],[y1,y2],'k-')

        for i_index in range(n):
            connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

        plt.plot(coordlct_x, coordlct_y, 'o', color='black');

        for i_index in range(len(yi_open)):
            plt.plot(coordlct_x[yi_open[i_index]], coordlct_y[yi_open[i_index]], 'o', color='red');

        #Update vector of objective values and CPU Time
        objvalue_i=np.append(objvalue_i, objvalue)

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

        clear_output(wait=True)
        plt.draw()
        plt.pause(0.1)
        plt.clf()

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

If objective value is worse than the best objective value found in previous iterations, then nothing happens; otherwise we update the best solution

Hill Climbing

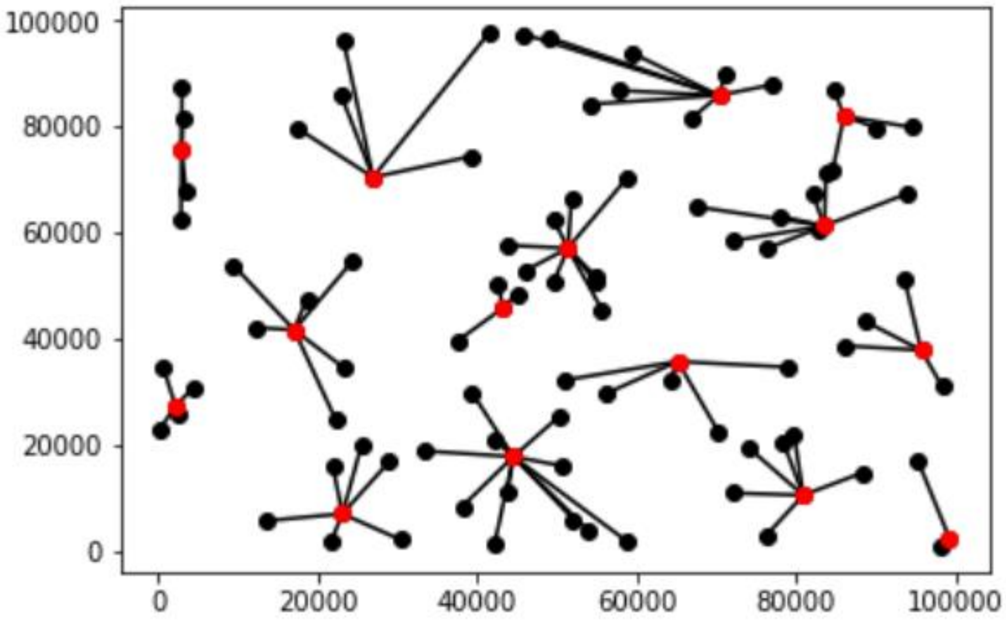
Plot

Replacement Phase

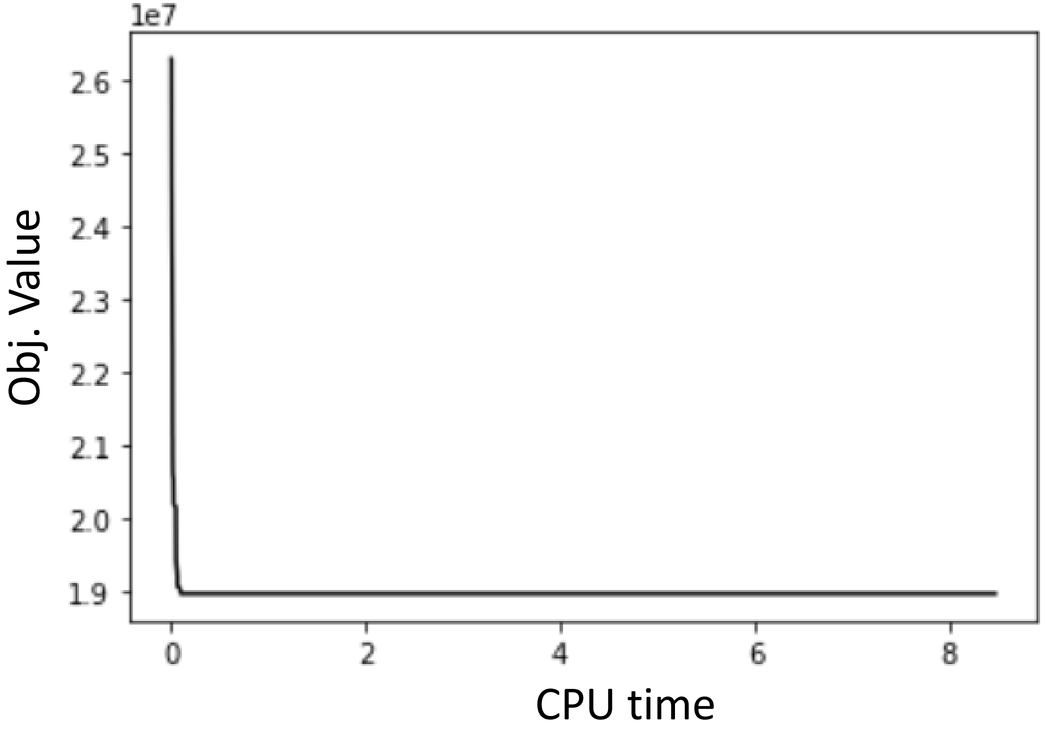
(First Descent)

Keep trace of the objective values obtained over time

P-Median – Hill Climbing Solution (n=100 ; fac=15)



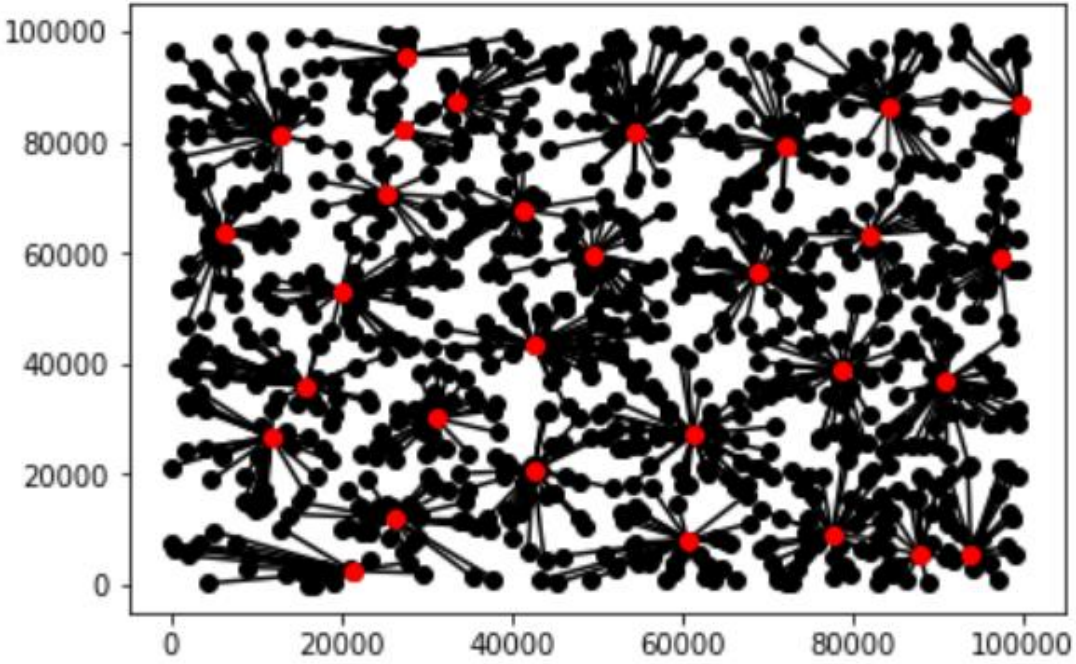
Hill Sampling

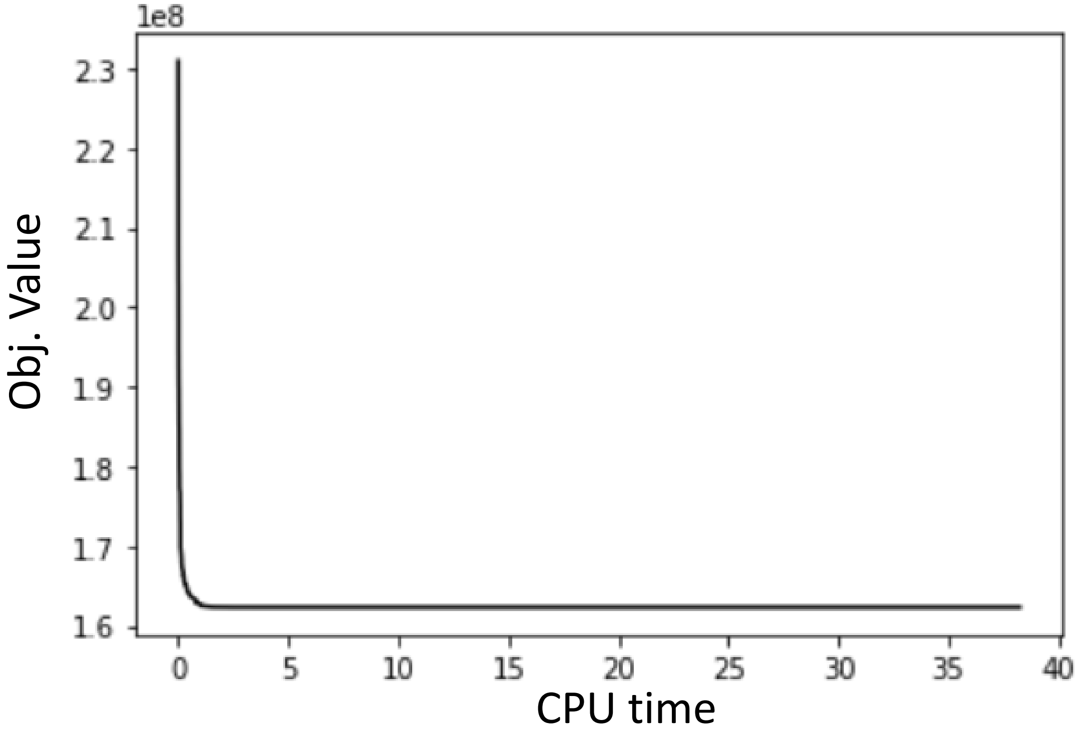


Obj. Value Rand. Sampling = 21,321,318.07 (Gap = 11.1%)
Obj. Value Hill Climbing = 18,983,919.72 (Gap = 0.16%)
Optimum= 18,954,163.57 (0.7 seconds)

P-Median – Hill Climbing Solution (n=1000 ; fac=30)

Hill Climbing

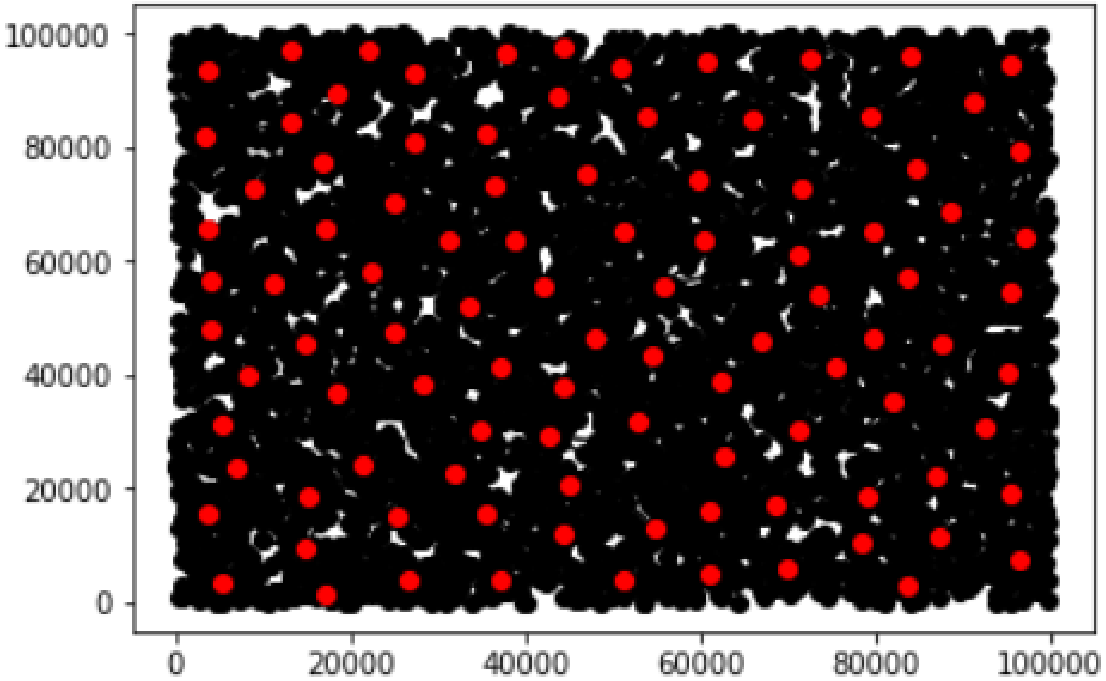


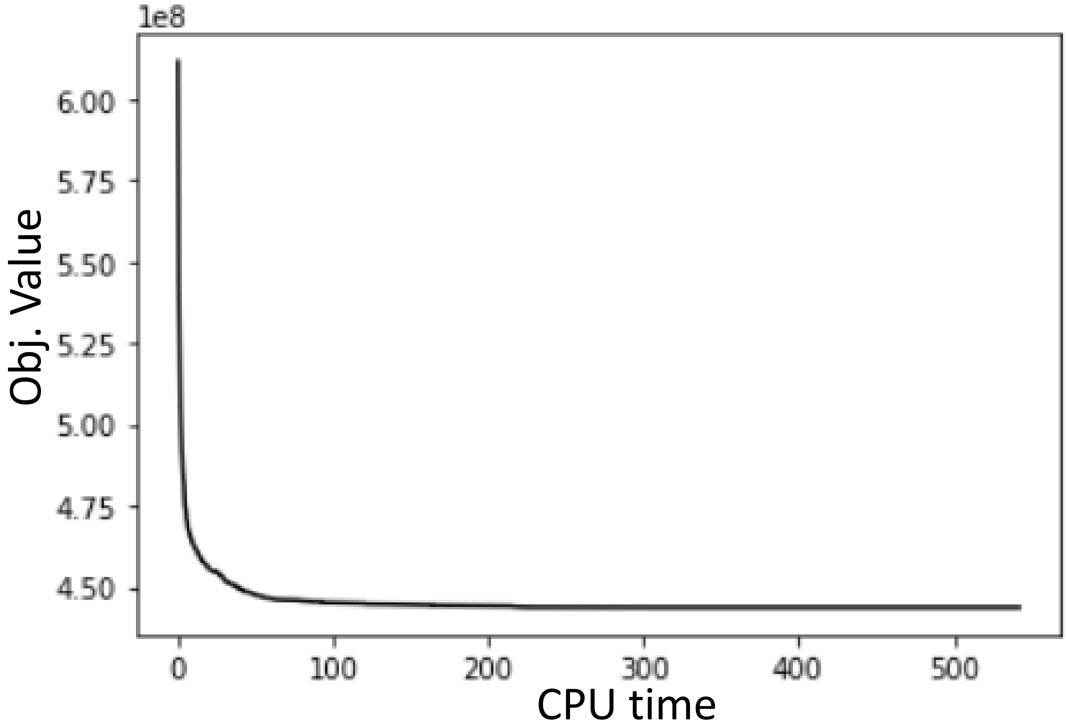


Obj. Value Rand. Sampling = 194,419,346.65 (Gap = 16.9%)
Obj. Value Hill Climbing = 162,325,709.91 (Gap = 0.57%)
Optimum= 161,393,599.84 (321 seconds)

P-Median – Hill Climbing Solution (n=5000 ; fac=100)

Hill Climbing





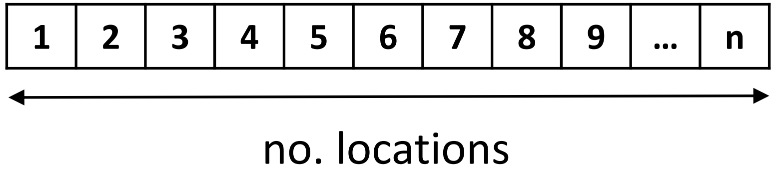
Obj. Value Rand. Sampling = 561,681,400.18 (21%)

Obj. Value Hill Climbing = 443,792,460.62

Optimum=?

TSP Example

- **Solution Representation:** Premutation Encoding
- **Move Operator:** Swap 2 locations
- **Replacement Procedure:** First Descent



1	2	3	4	5	6	7	8	9	…	n

no. locations

Number of candidate locations
n=100

TSP – Generate Instance

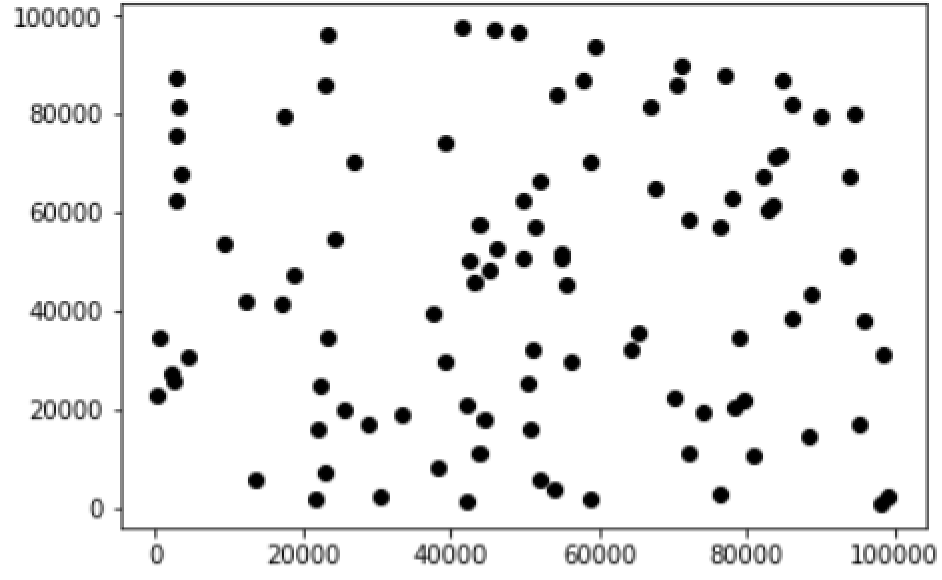
Inputs

```
#Generate Data Inputs

# Select random seed
random.seed(1)

# Number of cities
n=500

#Coordinate Range
rangelct=10000

#Generate random locations
coordlct_x = random.choices(range(0, rangelct), k=n)
coordlct_y = random.choices(range(0, rangelct), k=n)

#Compute distance between locations
distancelct=np.empty([n, n])
for i_index in range(n):
    for j_index in range(n):
        distancelct[i_index,j_index]=(math.sqrt(((coordlct_x[i_index]-coordlct_x[j_index])**2) +((coordlct_y[i_index]-coordlct_y

distancelct[np.diag_indices_from(distancelct)] = 99999

df = pd.DataFrame(distancelct)
df.index += 1
df.columns += 1
cij_model=df.stack().to_dict()
```

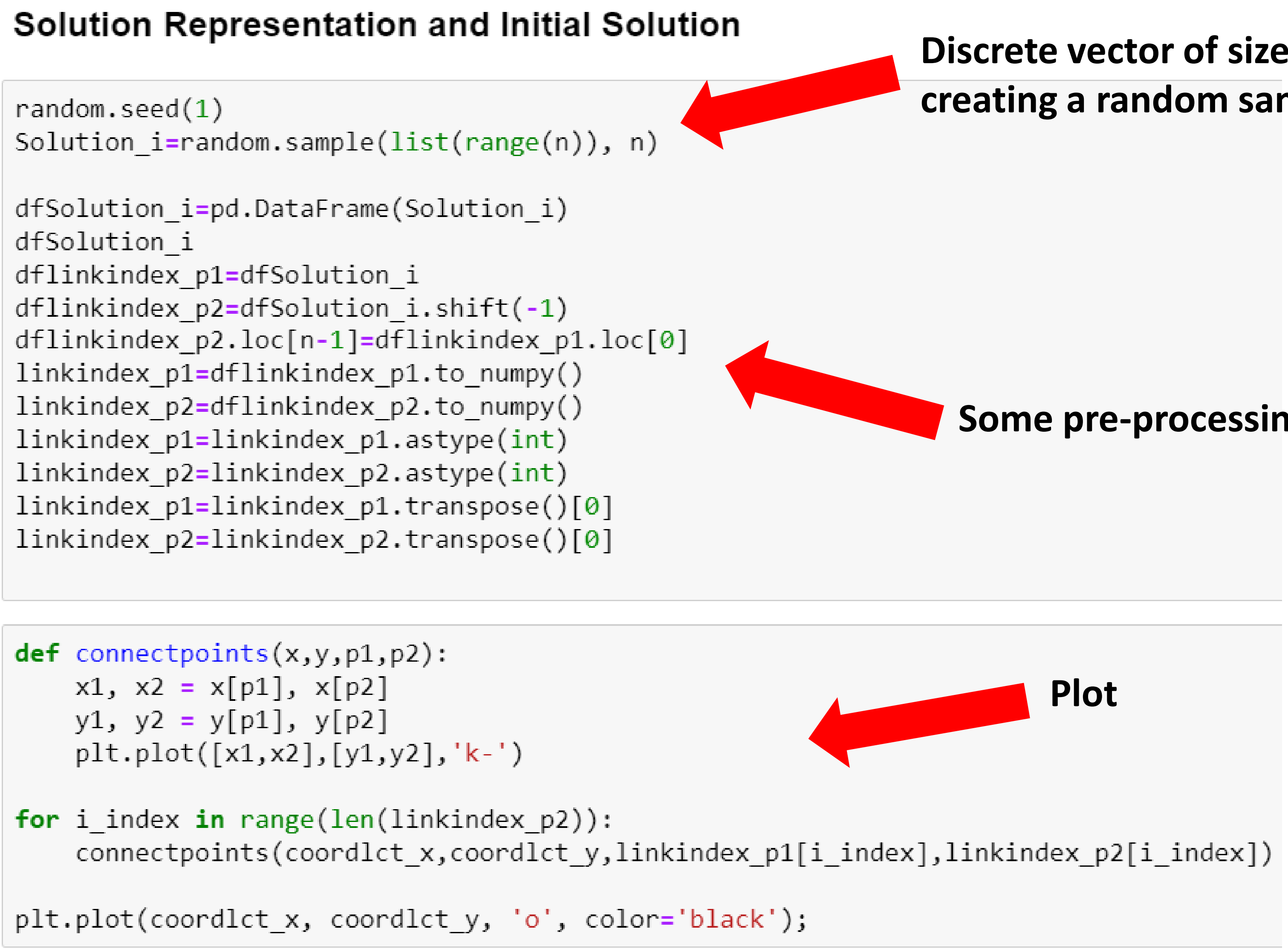
Inputs

Random Generation of Locations

Array i,j of distances between locations



TSP – Initial Solution



Solution Representation and Initial Solution

```
random.seed(1)
Solution_i=random.sample(list(range(n)), n)

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]
```

```
def connectpoints(x,y,p1,p2):
    x1, x2 = x[p1], x[p2]
    y1, y2 = y[p1], y[p2]
    plt.plot([x1,x2],[y1,y2],'k-')

for i_index in range(len(linkindex_p2)):
    connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

plt.plot(coordlct_x, coordlct_y, 'o', color='black');
```

Discrete vector of size n is generated by creating a random sample of size n

Some pre-processing

Plot

TSP – Random Sampling Search Procedure

Random Sampling

Random Algorithm

```
random.seed(3)
iteration=0
ObjValueOpt=ObjValue
Objvalue_list=ObjValue
program_starts = time.time()
cputime_i=[0,0]

while cputime_i[-1]<6000:

    iteration=iteration+1

    #Random permutation
    Solution_i=random.sample(list(range(n)), n)
    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])
```

Generate new random permutation of locations

Compute Objective Value

TSP – Random Sampling Search Procedure

```
    #Update Optimal Solution
    if ObjValue<ObjValueOpt:
        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)
        #def connectpoints(x,y,p1,p2):
        #    x1, x2 = x[p1], x[p2]
        #    y1, y2 = y[p1], y[p2]
        #    plt.plot([x1,x2],[y1,y2],'k-')

        #for i_index in range(len(linkindex_p2)):
        #    connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

        #plt.plot(coordlct_x, coordlct_y, 'o', color='black');

        #clear_output(wait=True)
        #plt.draw()
        #plt.pause(0.1)
        #plt.clf()

#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)
```

Random Sampling

If objective value is worse than the best objective value found in previous iterations, then nothing happens; otherwise we update the best solution

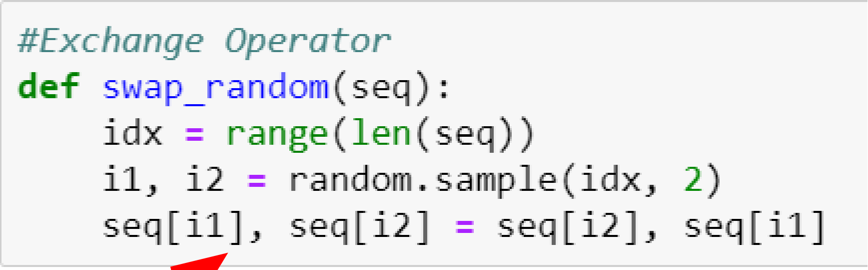
Plot

Keep trace of the objective values obtained over time

TSP – Hill Climbing Generation Phase

```
random.seed(3)
iteration=0
ObjValueOpt=ObjValue
Objvalue_list=ObjValue
program_starts = time.time()
cputime_i=[0,0]
OptSolution=Solution_i

while cputime_i[-1]<6000:

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

    swap_it=0
    while swap_it<no_swap:
        swap_random(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])
```

Hill Climbing



```
#Exchange Operator
def swap_random(seq):
    idx = range(len(seq))
    i1, i2 = random.sample(idx, 2)
    seq[i1], seq[i2] = seq[i2], seq[i1]
```

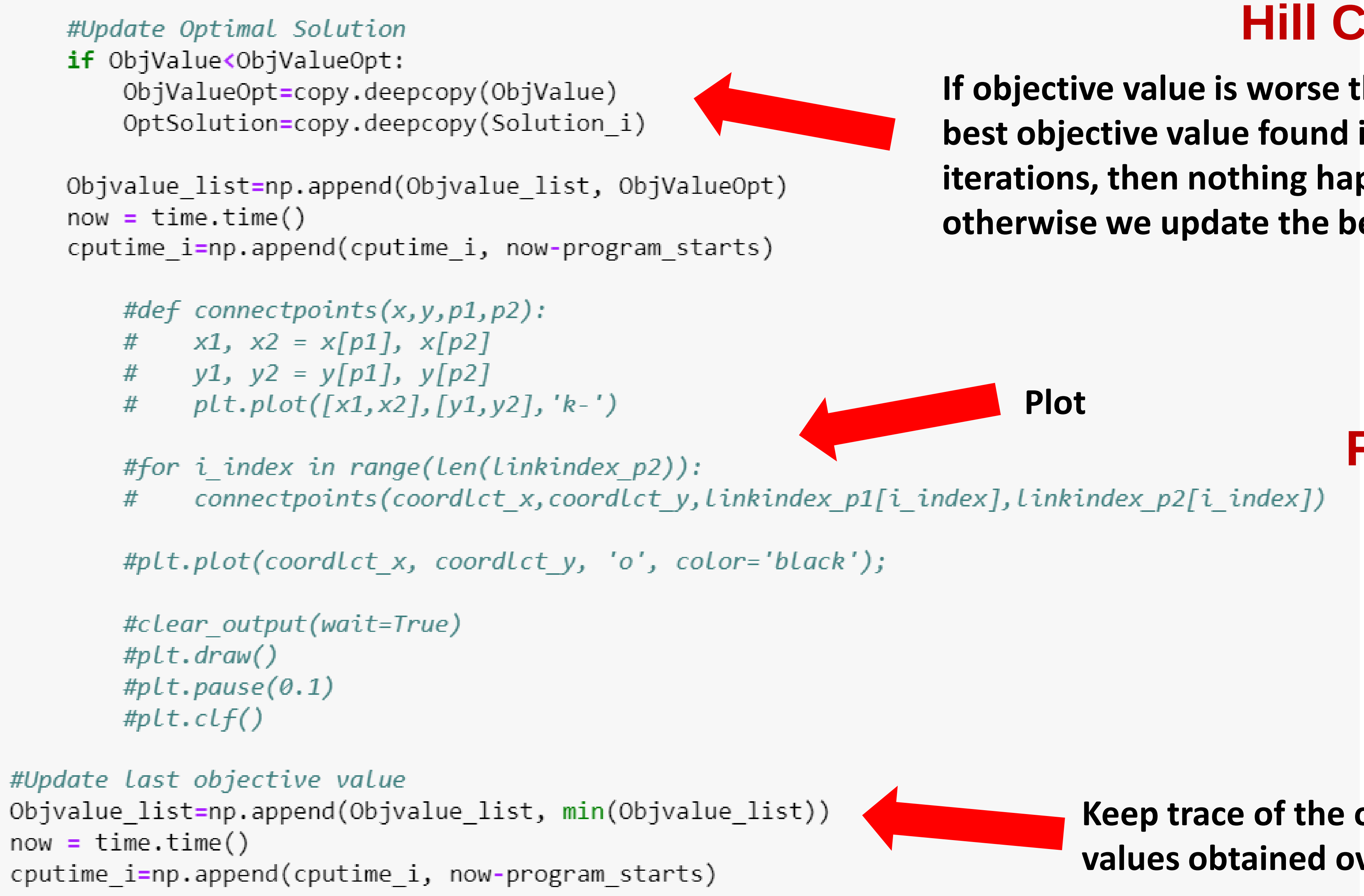
Apply swap operator

Select two random locations
Swap location in the permutation

Generation Phase
(Move Operator Code)

Compute Objective Value

TSP – Hill Climbing Replacement Phase



```
    #Update Optimal Solution
    if ObjValue<ObjValueOpt:
        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)

        #def connectpoints(x,y,p1,p2):
        #    x1, x2 = x[p1], x[p2]
        #    y1, y2 = y[p1], y[p2]
        #    plt.plot([x1,x2],[y1,y2],'k-')

        #for i_index in range(len(linkindex_p2)):
        #    connectpoints(coordlct_x,coordlct_y,linkindex_p1[i_index],linkindex_p2[i_index])

        #plt.plot(coordlct_x, coordlct_y, 'o', color='black');

        #clear_output(wait=True)
        #plt.draw()
        #plt.pause(0.1)
        #plt.clf()

#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)
```

Hill Climbing

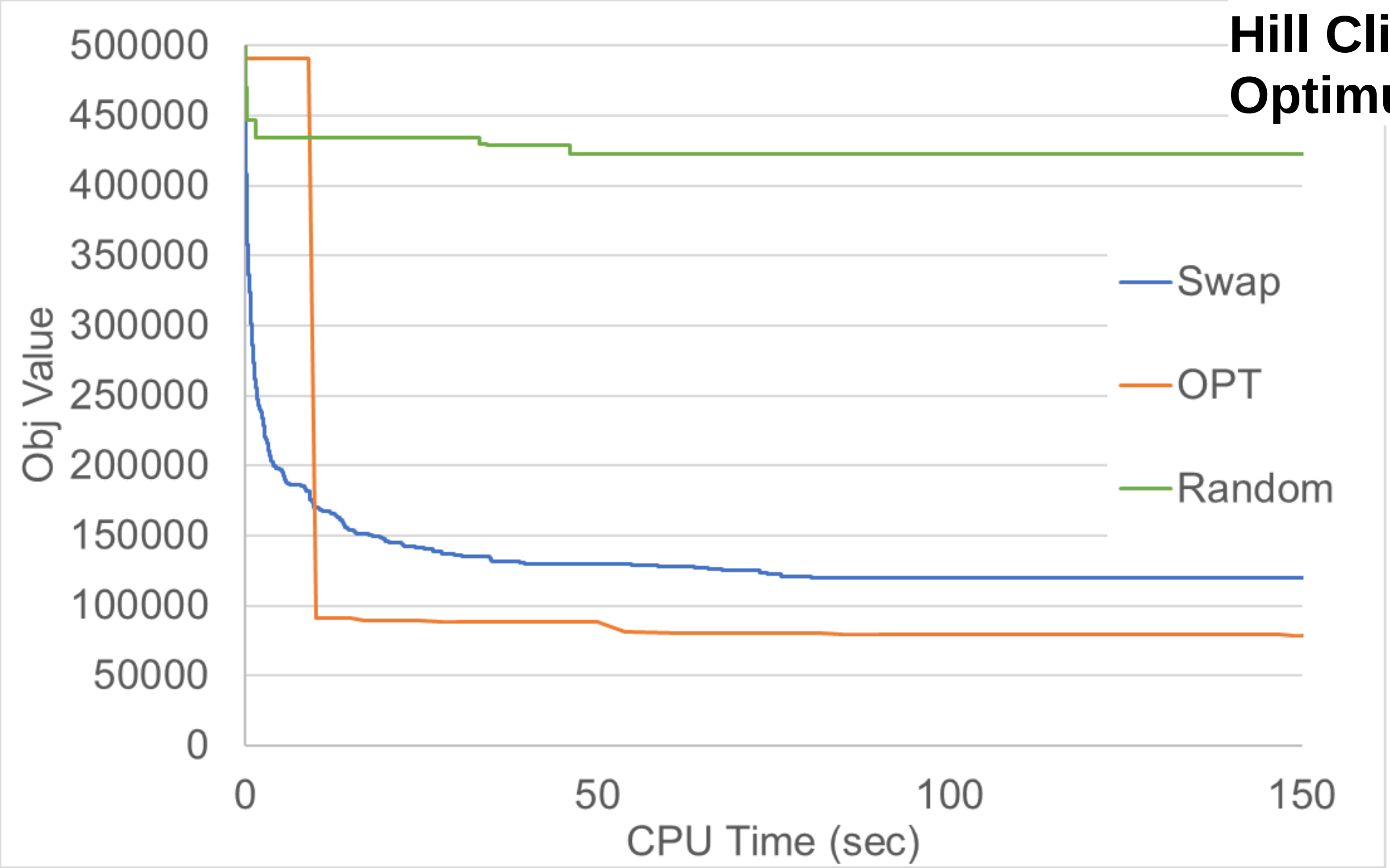
If objective value is worse than the best objective value found in previous iterations, then nothing happens; otherwise we update the best solution

Plot

Replacement Phase
(First Descent)

Keep trace of the objective values obtained over time

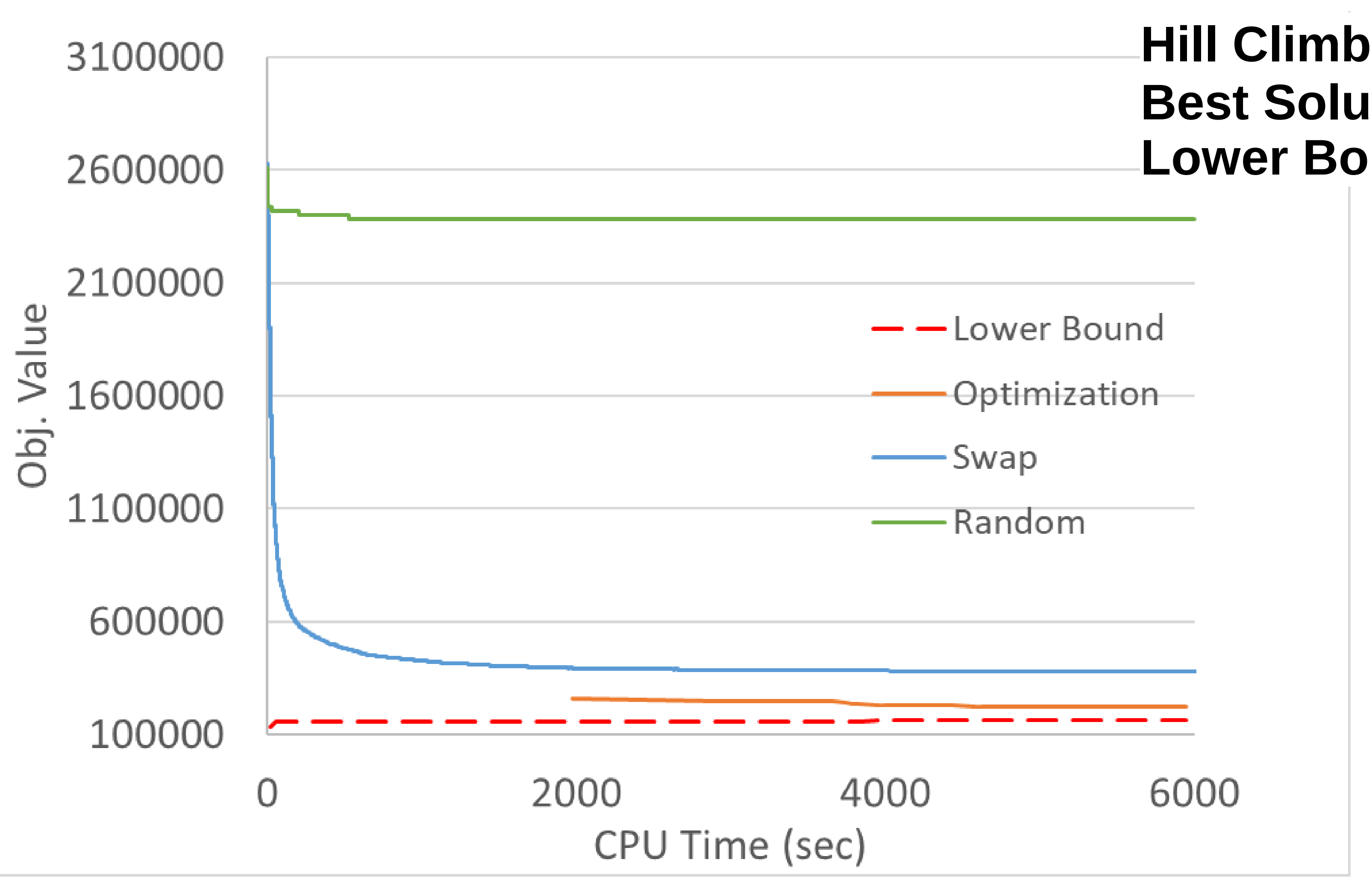
TSP Solution (n=100)



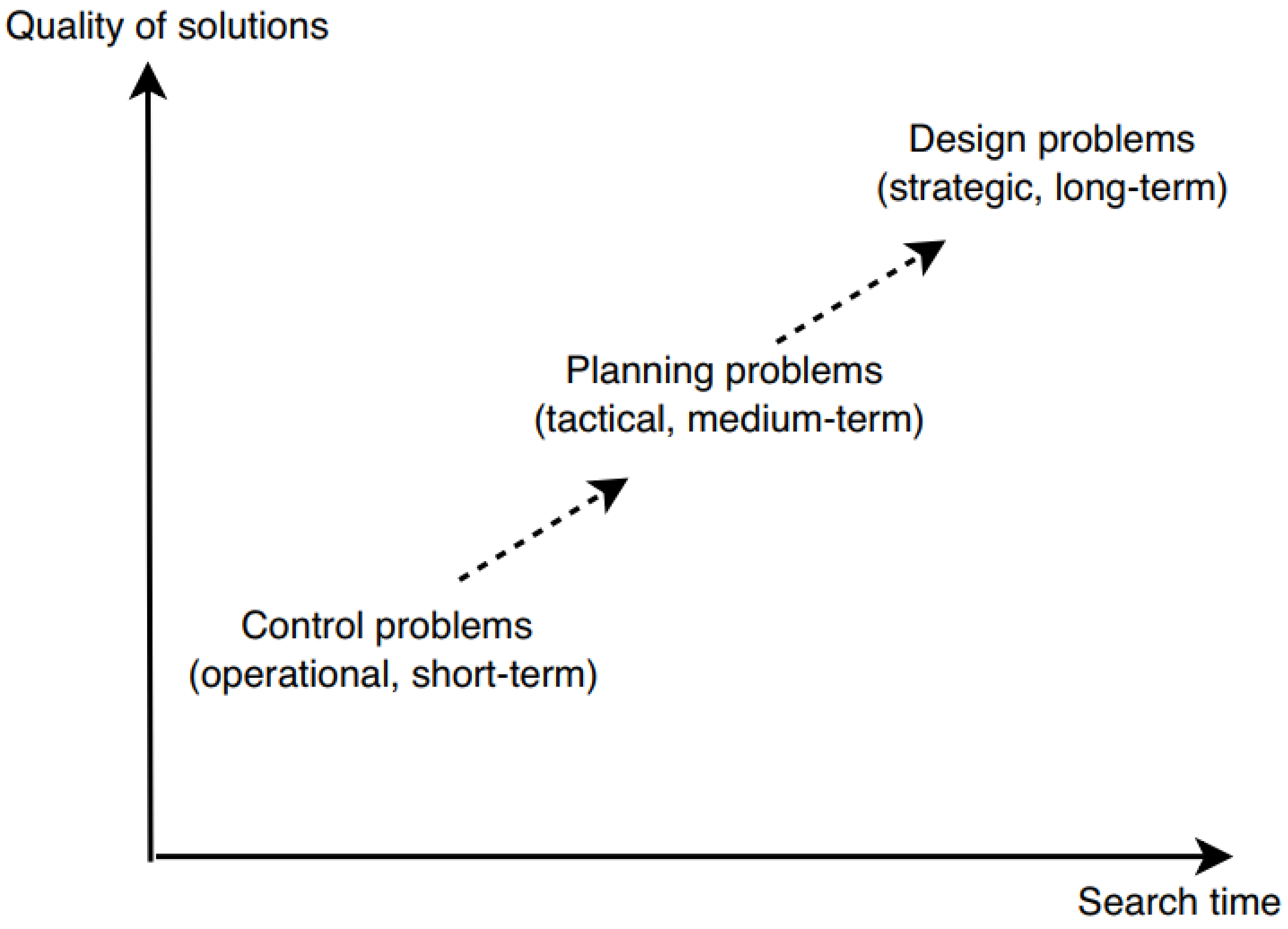
Rand. Sampling = 422,933 (81.5%)
Hill Climbing = 120,133 (34.7%)
Optimum= 78,357

TSP Solution (n=500)

Rand. Sampling = 2,383,336 (93.1%)
Hill Climbing =297,504 (45.0%)
Best Solution= 220,704 (25.9%)
Lower Bound= 163,571



Design versus Control Problems



Design versus Control Problems

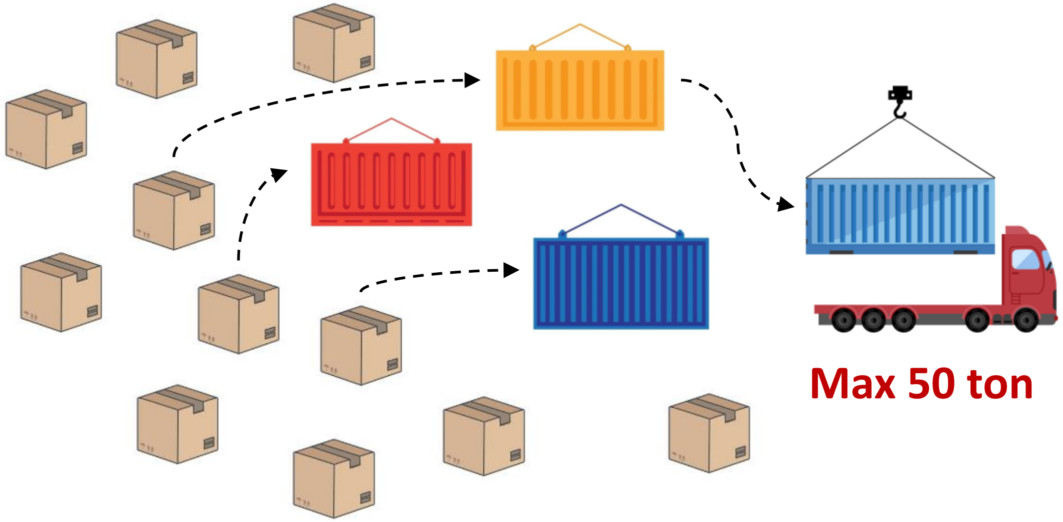
- **Design problems:** Design problems are generally solved once. They need a very good quality of solutions whereas the time available to solve the problem is important. These problems involve an important financial investment; (e.g. telecommunication network design and processor design, etc.)

- **Control problems:** Control problems represent the other extreme where the problem must be solved frequently in real time. These problems require very fast heuristics are needed; the quality of the solutions is less critical (e.g. routing messages in a computer network; traffic management in a city; ride-sharing operations .

- **Planning problems:** Between these extremes, one can find an intermediate class of problems represented by planning problems. In this class of problems, a trade-off between the quality of solution and the search time must be optimized; (e.g. scheduling of operations ; task assignment, etc.)

Activity 1

- Consider the following problem: Given a set of n packages with profit p_j and weight w_j , and a set of m containers with weight capacity c_i, select m disjoint subsets of packages so that the total profit of the selected packages is maximum, while ensuring the containers' capacity is never exceeded
- Exercise 1: Formulate the problem mathematically
- Exercise 2: Solve the problem using pyomo (instances in the next slide)
- **Exercise 3: Propose and apply a random sampling and a local search algorithm for the problem**



Activity 1 - Instances

- Instance 1

```
random.seed(1)
n = 100 #number of objects
b= 5 #number of bins
cap=50

#Generate random locations
value = random.choices(range(10, 100), k=n)
weights = random.choices(range(5, 20), k=n)
```

- Instance 2

```
random.seed(1)
n = 10000 #number of packages
b= 200 #number of bins
cap=50

#Generate random locations
profit = random.choices(range(10, 100), k=n)
weights = random.choices(range(5, 20), k=n)
```

Solution*=2356
CPU time = 0.53 sec

Solution*=117925.0
CPU time = 986 sec