Tabu Search

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

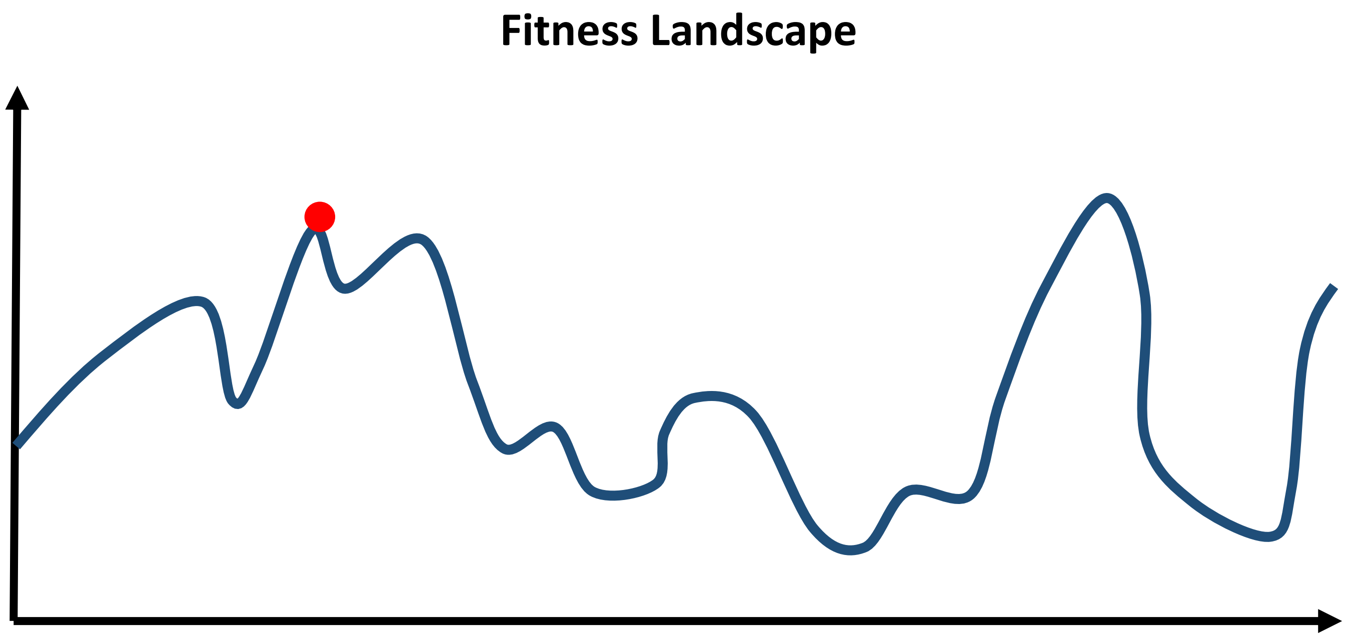
Engineering Systems and Design

Tabu Search

- Tabu search behaves like a hill climbing algorithm, but **it accepts non-improving solutions** to escape from local optima **when all neighbours** are non-improving solutions.
- Usually, the whole neighbourhood is explored (**best descent**), whereas in Simulated Annealing a random neighbour is selected (first descent).
- As in local search, when a better neighbour is found, it replaces the current solution. When a local optima is reached, the search carries on by selecting a candidate worse than the current solution.
- The best solution in the neighbourhood is selected as the new current solution even if it is not improving the current solution.

Tabu-Search

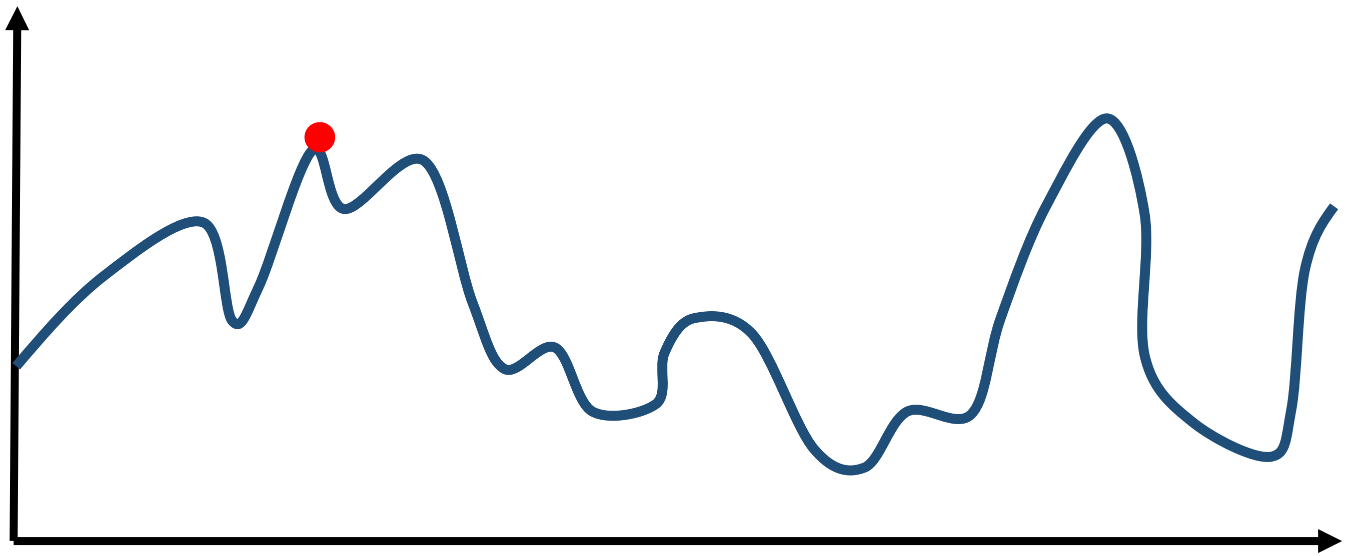
- **Problem:** This approach can easily lead to **cycles** (if the current solution is a local optimum, the search will go to a worse solution and then immediately back to the previous one, the local optimum).
- **Solution:** Introduce a **tabu list** which forbids certain solutions to be visited, to avoid re-visiting already seen solutions.



Tabu-Search

- **Problem:** This approach can easily lead to **cycles** (if the current solution is a local optimum, the search will go to a worse solution and then immediately back to the previous one, the local optimum).
- **Solution:** Introduce a **tabu list** which forbids certain solutions to be visited, to avoid re-visiting already seen solutions.

Fitness Landscape



Short Term Memory

- The role of the short-term memory is to store the recent history of the search to prevent cycling.
- **Explicitly Memory:** the approach of storing complete solutions generally consumes a massive amount of space. Moreover, checking the presence of all neighbour solutions in the tabu list will be prohibitive – useful method if evaluating a given solution is time consuming (e.g. running a detailed simulation)
- **Recency-based Attributive Memory:** Usually we do not store complete solutions, but only attributes of solutions. These features often depend on the search moves (**tabu moves**).
- For instance:
  - If we apply an insertion move to a given city in the Traveling Salesman Problem, we may simply forbid the inserted city from being inserted again.
  - If we apply a single-bit-flip move in the p-median problem, we simply may forbit the same variable from being flipped again.
  - More generally: If we reach a new solution p_{new} via search move, we may **forbit any move touching the same decision variables.**

Example – Knapsack Problem

- **Initial Solution**

Objects	Profit	Weight	Ratio	Value	Profit	Weight
1	10	7	1.43	1	10	7
2	14	12	1.17	0	0	0
3	9	8	1.13	0	0	0
4	8	9	0.89	1	8	9
5	7	8	0.88	1	7	8
6	5	6	0.83	1	5	6
7	9	11	0.82	0	0	0
8	3	5	0.60	1	3	5
				total	33	35

Maximum Capacity = 40

Example – Knapsack Problem

- **Best Descent**

No.	Move	Neighbor	Profit	Weight	Feasible?
1	X1=0	(4,5,6,8)	23	28	Yes
2	X2=1	(1,2,4,5,6,8)	47	47	No
3	X3=1	(1,3,4,5,6,8)	42	43	No
4	X4=0	(1,5,6,8)	25	26	Yes
5	X5=0	(1,4,6,8)	26	27	Yes
6	X6=0	(1,4,5,8)	28	29	Yes
7	X7=1	(1,4,5,6,7,8)	42	46	No
8	**X8=0**	**(1,4,5,6)**	**30**	**30**	**Yes**

Example – Knapsack Problem

- **Iterations – Tabu Size = 2**

Iteration	Current Solution	Profit	Weight	Tabu Active	Move?
1	(1,4,5,6,8)	33	35		8
2	(1,4,5,6)	30	30	8	3
3	(1,3,4,5,6)	39	38	3 8	6
4	(1,3,4,5)	34	32	6 3	8
5	(1,3,4,5,8)	37	37	8 6	5
6	(1,3,4,8)	30	29	5 8	6
7	(1,3,4,6,8)	35	35	6 5	8
8	(1,3,4,6)	32	30	8 6	5
9	(1,3,4,5,6)	39	38	3 8	6
10	(1,3,4,5)	34	32	6 3	8
11	(1,3,4,5,8)	37	37	8 6	5
12	(1,3,4,8)	30	29	5 8	6
13	(1,3,4,6,8)	35	35	6 5	8
14	(1,3,4,6)	32	30	8 6	5
15	(1,3,4,5,6)	39	38	5 8	6

Example – Knapsack Problem

- **Iterations – Tabu Size = 2**

Iteration	Current Solution	Profit	Weight	Tabu Active	Move?
1	(1,4,5,6,8)	33	35		8
2	(1,4,5,6)	30	30	8	3
3	(1,3,4,5,6)	39	38	3 8	6
4	(1,3,4,5)	34	32	6 3	8
5	(1,3,4,5,8)	37	37	8 6	5
6	(1,3,4,8)	30	29	5 8	6
7	(1,3,4,6,8)	35	35	6 5	8
8	(1,3,4,6)	32	30	8 6	5
9	(1,3,4,5,6)	39	38	3 8	6
10	(1,3,4,5)	34	32	6 3	8
11	(1,3,4,5,8)	37	37	8 6	5
12	(1,3,4,8)	30	29	5 8	6
13	(1,3,4,6,8)	35	35	6 5	8
14	(1,3,4,6)	32	30	8 6	5
15	(1,3,4,5,6)	39	38	5 8	6

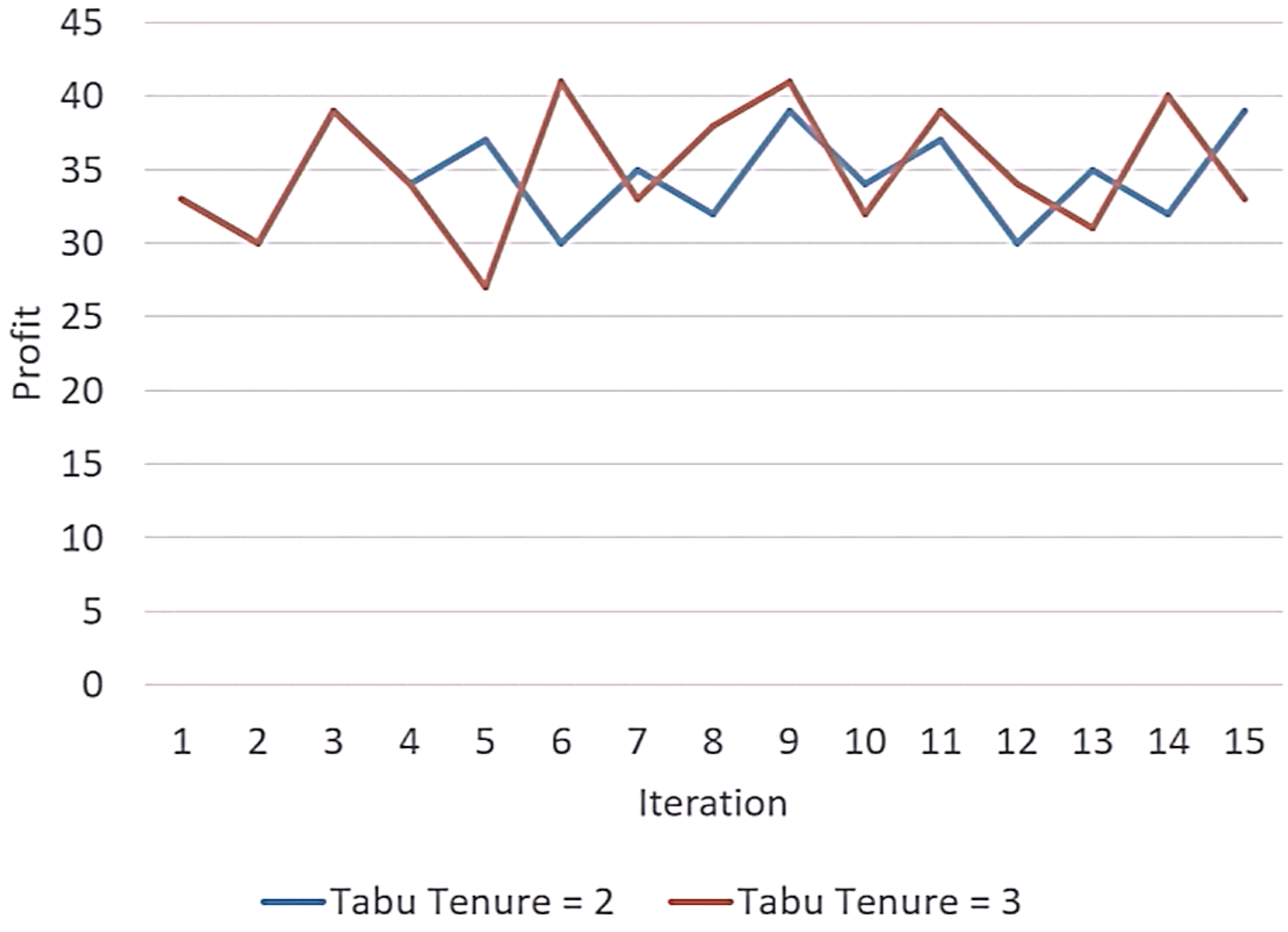
Cycle

Example – Knapsack Problem

- **Iterations – Tabu Size = 3**

Iteration	Current Solution	Profit	Weight	Tabu Active	Move?
1	(1,4,5,6,8)	33	35		8
2	(1,4,5,6)	30	30	8	3
3	(1,3,4,5,6)	39	38	3 8	6
4	(1,3,4,5)	34	32	6 3 8	5
5	(1,3,4)	27	24	5 6 3	2
6	(1,2,3,4)	41	36	2 5 6	4
7	(1,2,3)	33	27	4 2 5	6
8	(1,2,3,6)	38	33	6 4 2	8
9	(1,2,3,6,8)	41	38	8 6 4	3
10	(1,2,6,8)	32	30	3 8 6	5
11	(1,2,5,6,8)	39	38	5 3 8	6
12	(1,2,5,8)	34	32	6 5 3	8
13	(1,2,5)	31	27	8 6 5	3
14	(1,2,3,5)	40	35	3 8 6	5
15	(1,2,3)	33	27	5 3 8	4

Example – Knapsack Problem



Tabu Size

- **Tabu Size (or tenure):** Size of the tabu list – that is **how many iterations a move is tabu.**
- **Critical parameter** - The smaller is the size of the tabu list, the more likely is the probability of cycling. Larger sizes of the tabu list will provide many restrictions and encourage the diversification.
  - **Static:** In general, a constant value is associated with the tabu size. It may depend on the size of the problem instance and particularly the size of the neighborhood
  - **Dynamic:** The size of the tabu list may change during the search without using any information on the search memory (multistage).
  - **Adaptive:** In the adaptive scheme, the size of the tabu list is updated according to the search memory.

Tabu Search Algorithm

1. **Initialize**: Generate initial solution, p_{best}
2. **While** (termination criteria (2) is not met)
   3. **Apply non-tabu move to generate a set of new solutions within the same neighbourhood, $\boldsymbol{p_i}$**
   4. Select the best solution found, $p_{new} = best(p_i)$
   5. Update $p_{best} = p_{new}$
   6. **Update tabu list**
   7. Go back to (3), until termination criteria (2) is not met

Aspiration Criteria and Solution Abstraction

- **Aspiration Criteria:** used to override the tabu status of a move. For example, we can still accept a tabu move if it leads to a solution that is better than the best found so far
- **Solution Abstraction:** usually complete solutions are not stored in the tabu list. Instead, the search moves are included. This represents and abstract approximation of the solutions explored. However, this approach may prevent the tabu search from moving into solutions that were not yet explored.
- **Strengthening the abstraction:** store more details of the search move (e.g. move to the exact position in the solution encoding).

Tabu Search in Python

Nuno Antunes Ribeiro

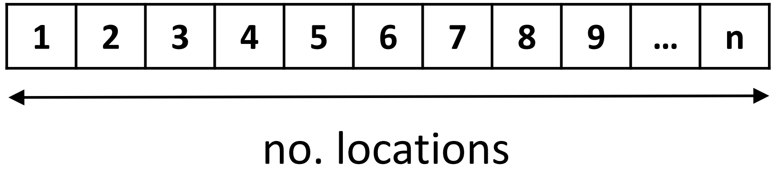
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

TSP Example

- **Solution Representation:** Premutation Encoding
- **Move Operator:** **Insert Operator**
- **Replacement Procedure:** **Best Descent**



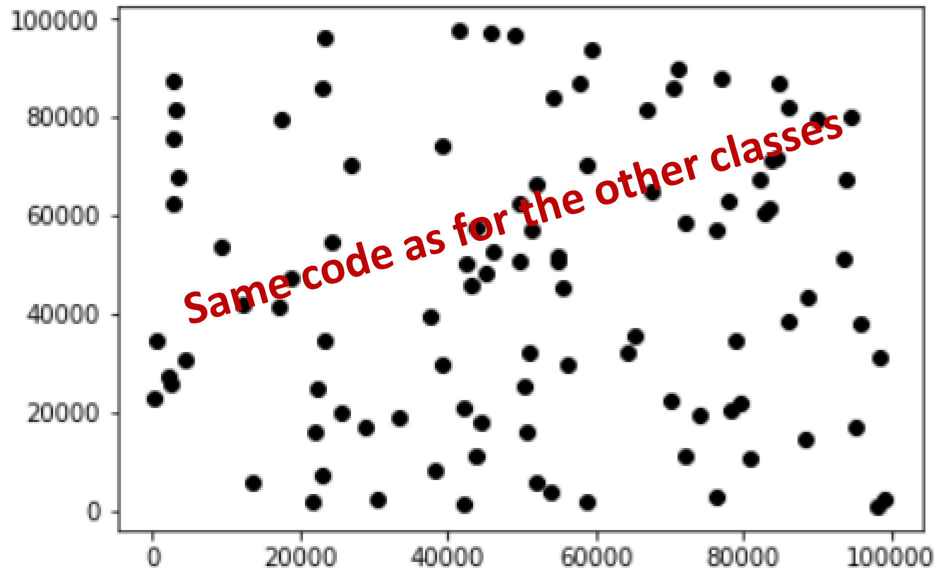
Number of candidate locations
n=100

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]
```

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

Some pre-processing

Same code as for the other classes

```
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');
```

Plot

Tabu Search Algorithm

Tabu Search Algorithm

```
random.seed(3)
iteration=0
Objvalue_fulllist=ObjValue
program_starts = time.time()
cputime_i=[0,0]
TabuList=[]
TabuSize=30

while cputime_i[-1]<120:

    Objvalue_list=9999999999999999 #auxiliary variable, needs to be a large number
    Solution_it=copy.deepcopy(Solution_i)
    idx = range(len(Solution_it))
    for i_index in range(len(TabuList)):
        idx=[i for i in idx if i != TabuList[i_index]]
    i1=random.sample(idx, 1)
    dist=distancelct[i1,]
    dist=dist[0]

    #Select Size of the Neighborooh
    if cputime_i[-1]<3000:
        quartile = int(0.99 * (len(dist)- 1))
    else:
        quartile = int(0.99 * (len(dist)- 1))

    maxdist=dist[np.argpartition(dist, quartile)[quartile]]
    i2_list=np.where(dist<maxdist)[0]
    random.shuffle(i2_list)

    #Best descent - for loop across all possible moves
    for i_index in range(len(i2_list)):
          ...
```

Initially, the tabu list is empty

Size of the tabu list (i.e. number of iterations a move is in the tabu list)

Select all variables that are not in the tabu list

Randomly select one of those variables to apply insert operator

Ignore for now (See slide 26)

Apply best descent

Tabu Search Algorithm

```
#Best descent - for loop across all possible moves
for i_index in range(len(i2_list)):

    #Insert Operator
    remove_index=np.where(np.array(Solution_it)==i1)
    Solution_it.pop(int(remove_index[0]))
    i2=i2_list[i_index]
    Solution_it.insert(i2, i1[0])

    #Solution Processing
    dfSolution_i=pd.DataFrame(Solution_it)
    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])
    Objvalue_list=np.append(Objvalue_list, ObjValue)

    #Select best move from the best descent
    if ObjValue==np.min(Objvalue_list):
        Solution_i=copy.deepcopy(Solution_it)
```

For loop across all insert positions (best descent)

Insert Operator

Just some processing

Compute Objective Value

Update best solution found during the best descent process

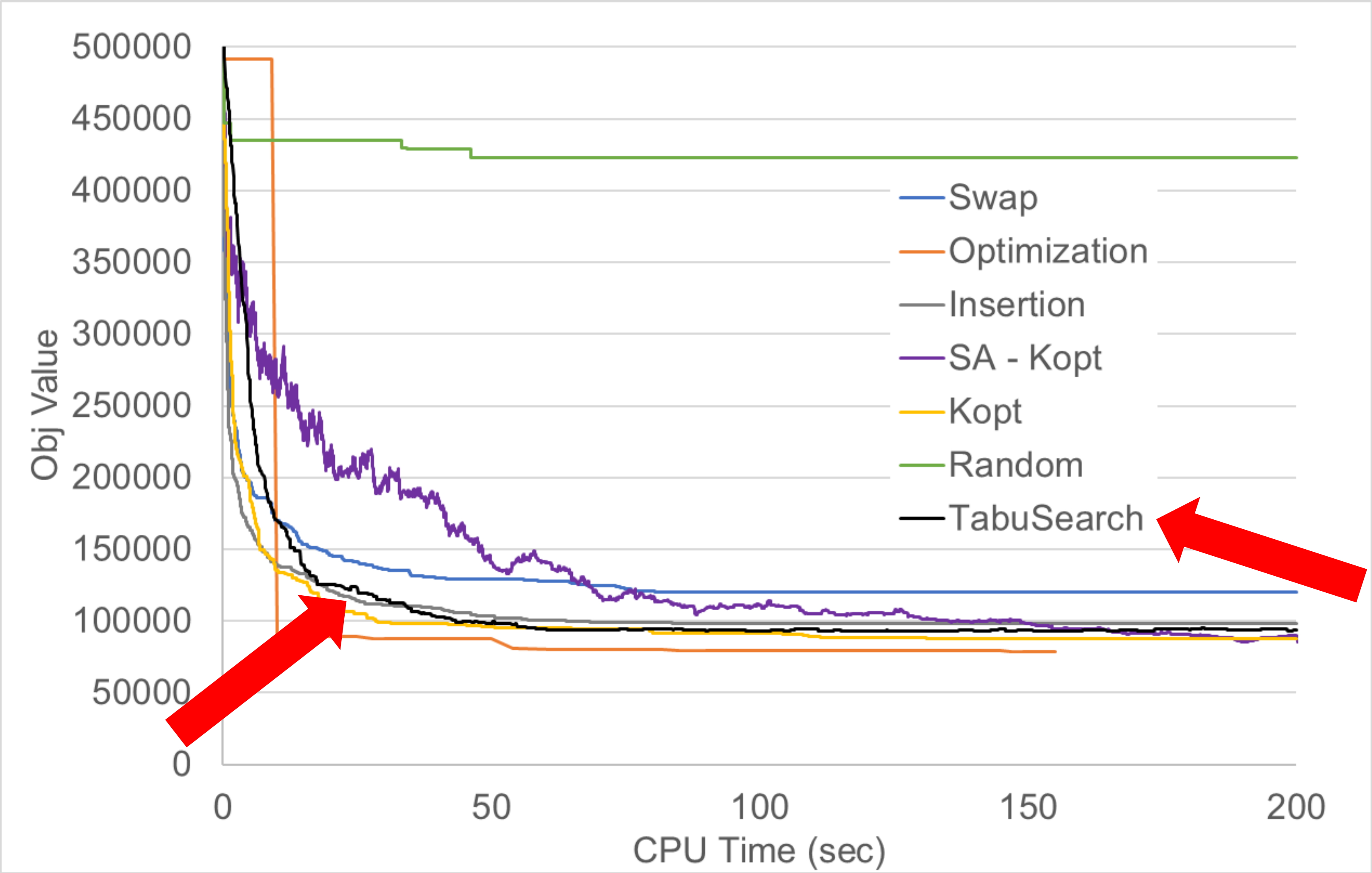
Tabu Search Algorithm

```
print(np.min(Objvalue_list))

Objvalue_fulllist=np.append(Objvalue_fulllist, min(Objvalue_list))
iteration=iteration+1
now = time.time()
cputime_i=np.append(cputime_i, now-program_starts)

#Update Tabu List
if len(TabuList)<TabuSize:
    TabuList=np.append(TabuList, i1)
else:
    TabuList=np.delete(TabuList, (0))
    TabuList=np.append(TabuList, i1)
```

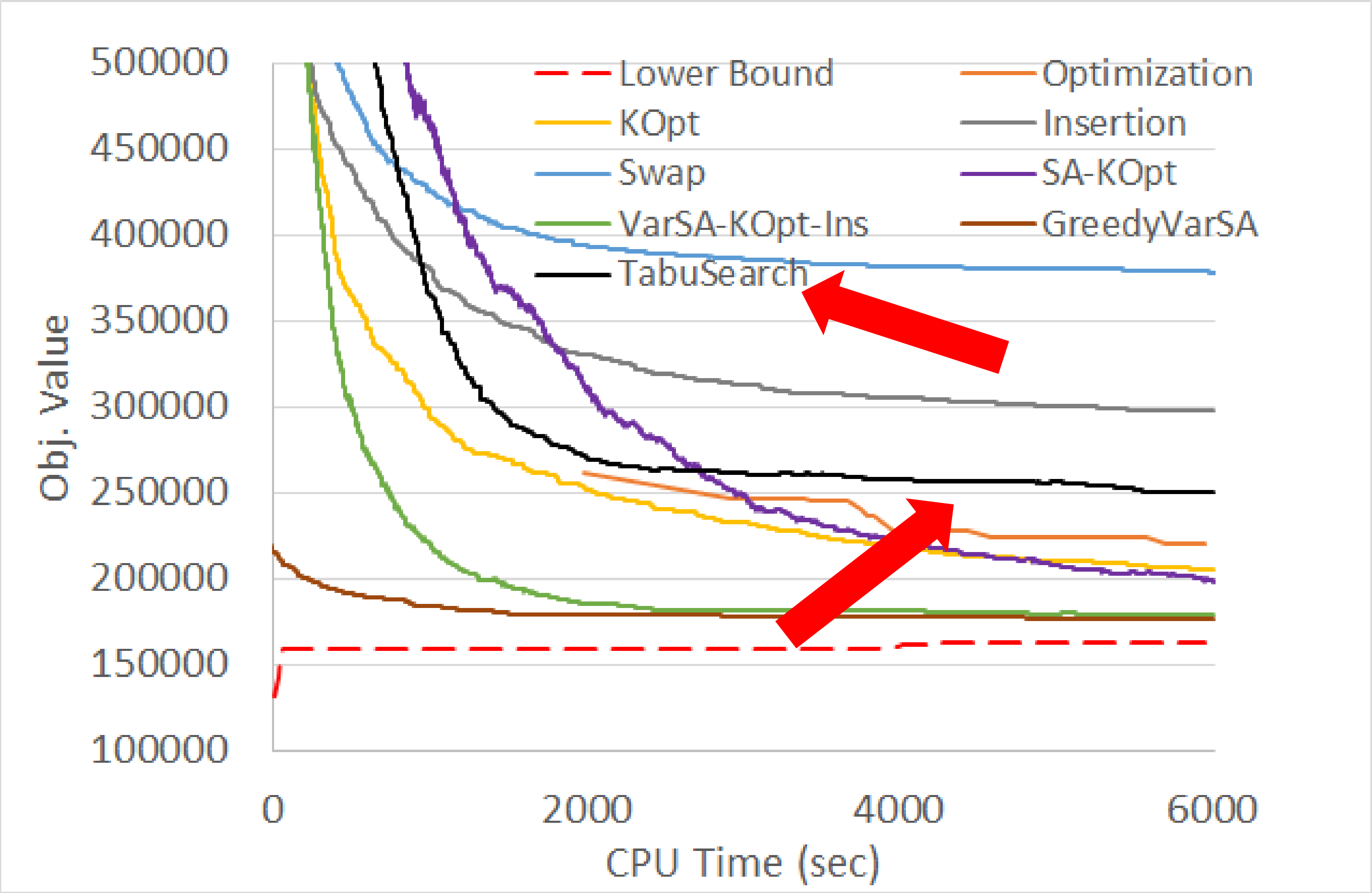
Update current solution and CPU time

Update tabu list

TSP Example (n=100)



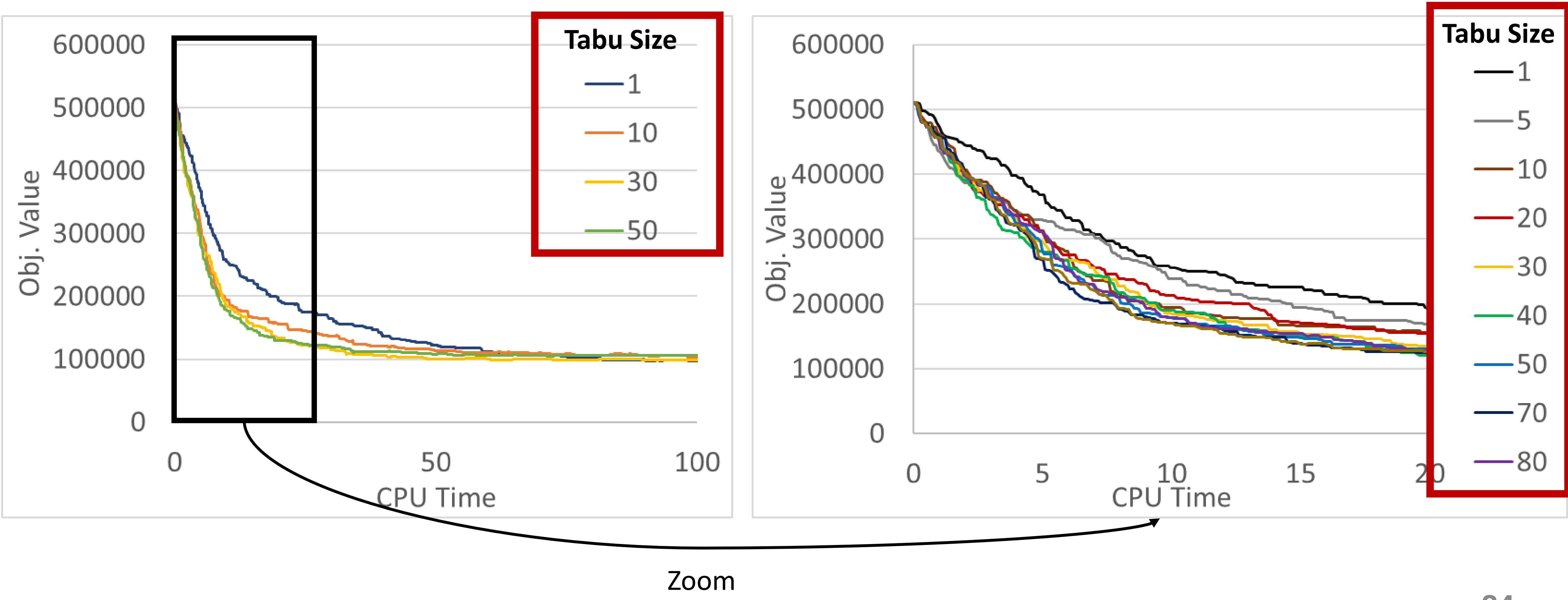
TSP Example (n=500)



TSP Example (n=100)

- Analyzing the effects of the Tabu Size in the results

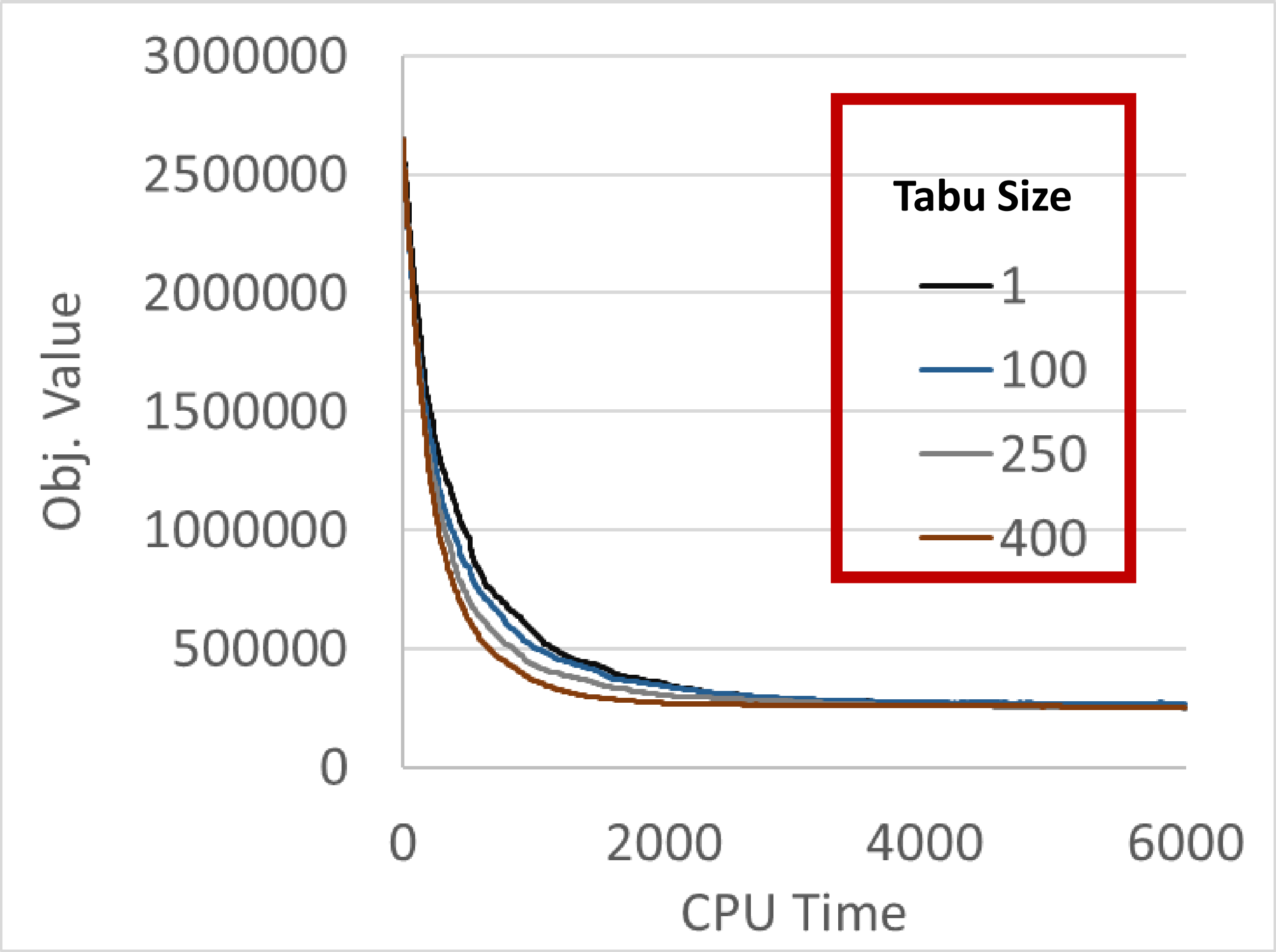
Tabu list is very effective during first iterations
In the long run, solutions will converge to a local optima



TSP Example (n=500)

- Analyzing the effects of the Tabu Size in the results

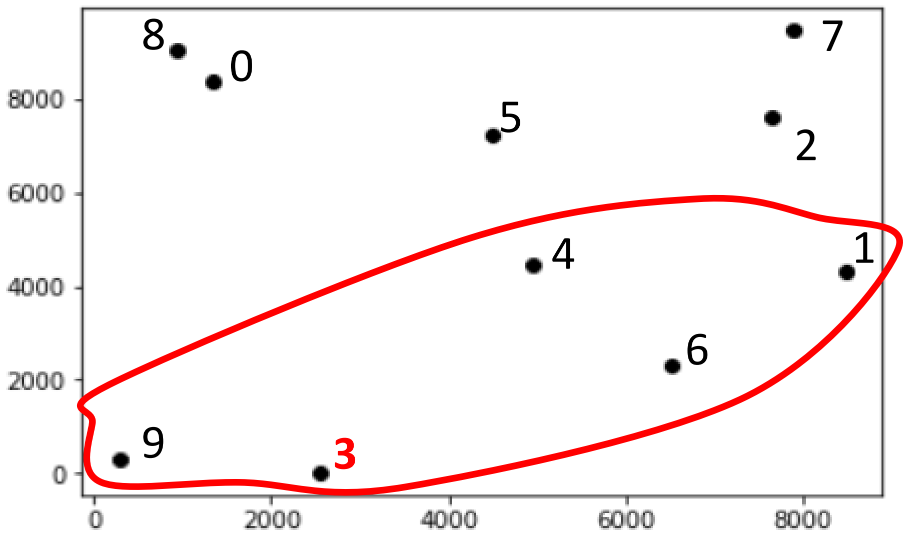
Tabu list is very effective during first iterations
In the long run, solutions will converge to a local optima



Reducing the size of the neighbourhood

- **Exploring the entire neighbourhood** of an insert move operator in the TSP can be **time consuming**
- We may want to reduce the size of the neighbourhood and focus on the most promising insertion moves.
- Example: Select only insertion moves that do not create paths with length larger than a certain amount.

N=10



- **New Insertion Operator**
  - **Randomly select a city (*i1*) that is not in the tabu list**
  - **Select the top 50% cities (*list2*) that are at a minimum distance from the city i1**
  - **Insert city i1 near the cities included in list2**

Reducing the size of the neighbourhood

```
idx = range(len(Solution_it))
for i_index in range(len(TabuList)):
    idx=[i for i in idx if i != TabuList[i_index]]
i1=random.sample(idx, 1)
dist=distancelct[i1,]
dist=dist[0]

quartile = int(0.5 * (len(dist)- 1))
```

```
maxdist=dist[np.argpartition(dist, quartile)[quartile]]
i2_list=np.where(dist<maxdist)[0]
```

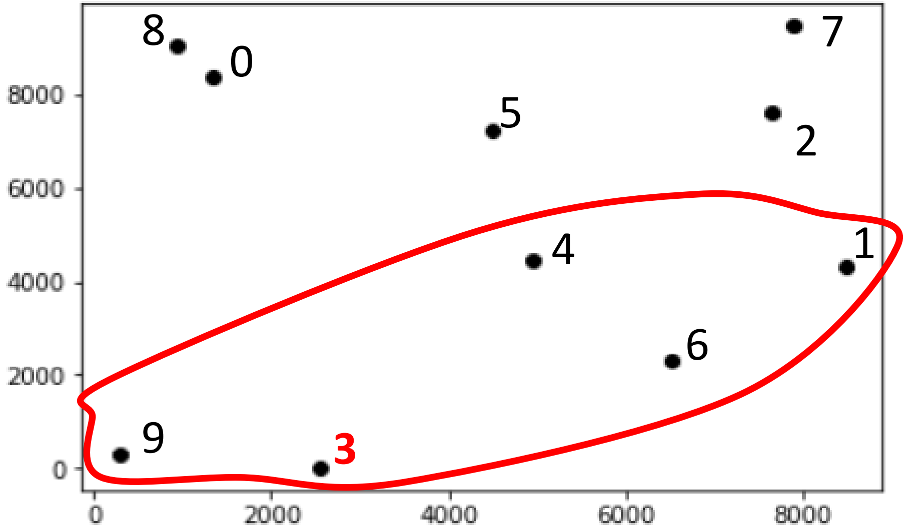
Subset of cities not included in the tabu list

Select a city from the list of cities not included in the tabu list

Compute the distance matrix from the city selected to all the other cities

Identify the top 50% cities that are located at a minimum distance from the city c

N=10




```
for i_index in range(len(i2_list)):
    remove_index=np.where(np.array(Solution_it)==i1)
    Solution_it.pop(int(remove_index[0]))
    i2=i2_list[i_index]
    Solution_it.insert(i2, i1[0])
```

At each iteration, the algorithm explores all possible insertions that are located nearby the city selected

Reducing the size of the neighbourhood

City selected $i_1 = 3$

Distance $dist =$

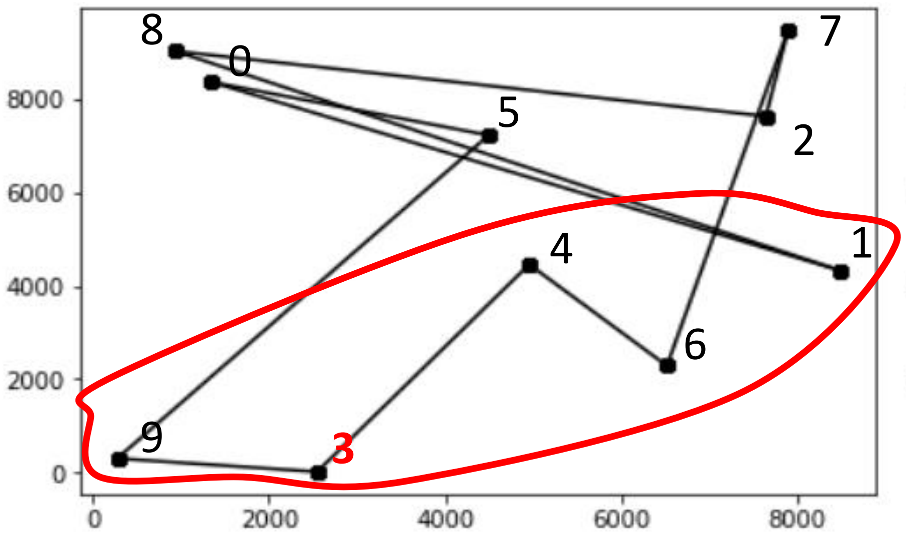
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9
8423	7324	9146	9999	5042	7452	4567	10836	9136	2284

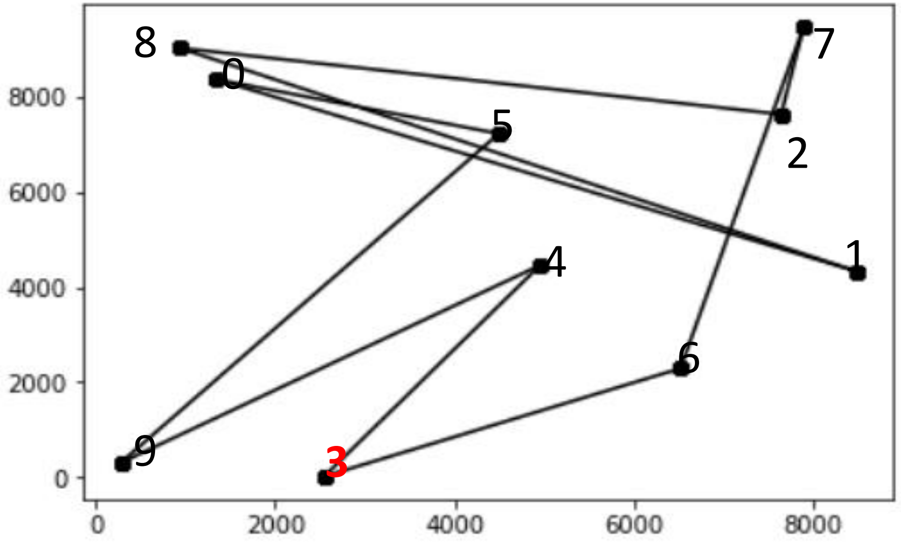
Quartile 50% $= 7452$

List of top 50% cities $= [1,4,6,9]$

	Ob. Value
Current Solution: [0,1,8,2,7,6,4,3,9,5]	54490
Neighbourhood Solutions:	
[0,3,1,8,2,7,6,4,9,5]	60966
[0,1,8,2,3,7,6,4,9,5]	71546
[0,1,8,2,7,6,3,4,9,5]	60349
[0,1,8,2,7,6,4,9,5,3]	65934

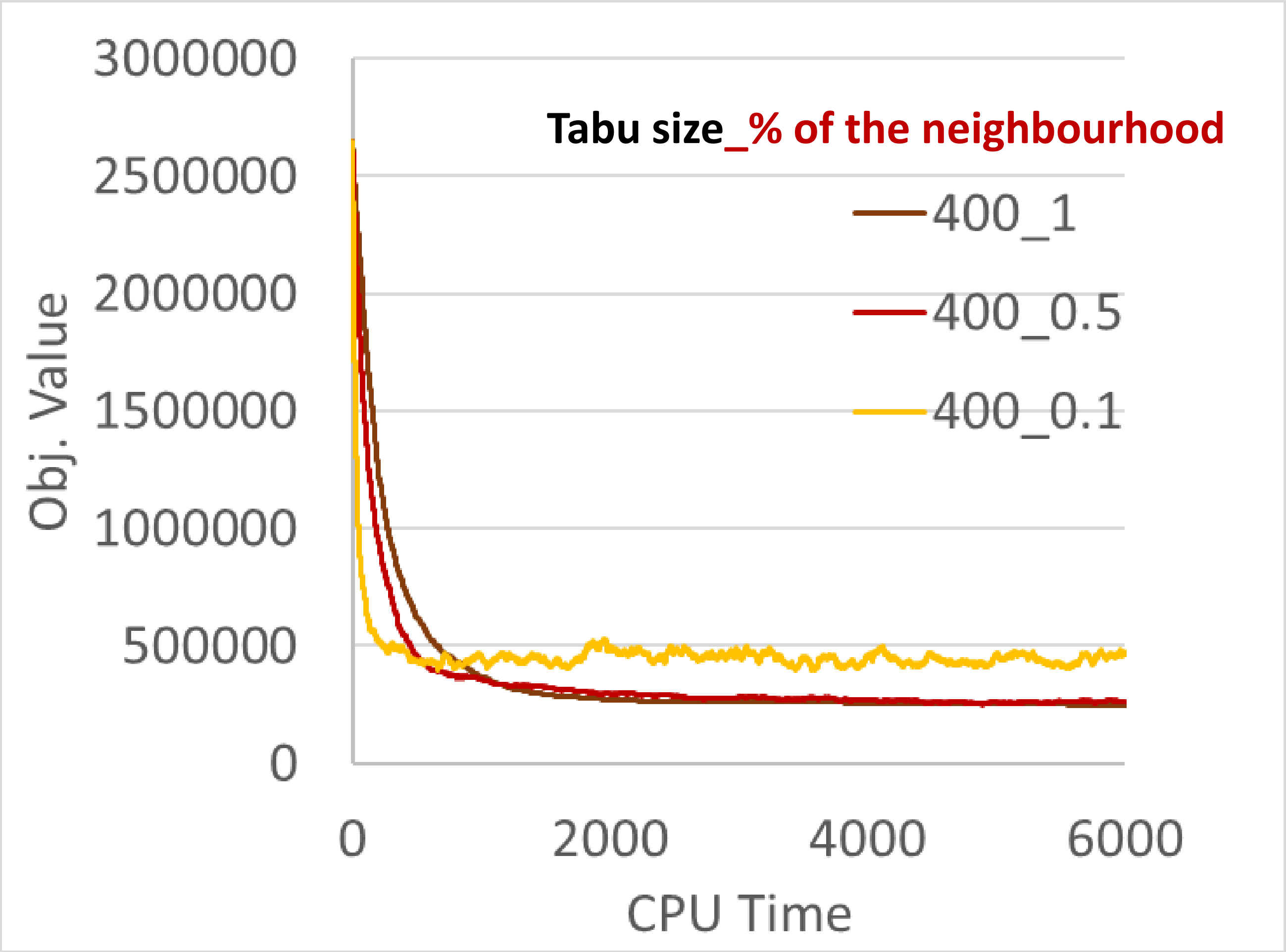
N=10



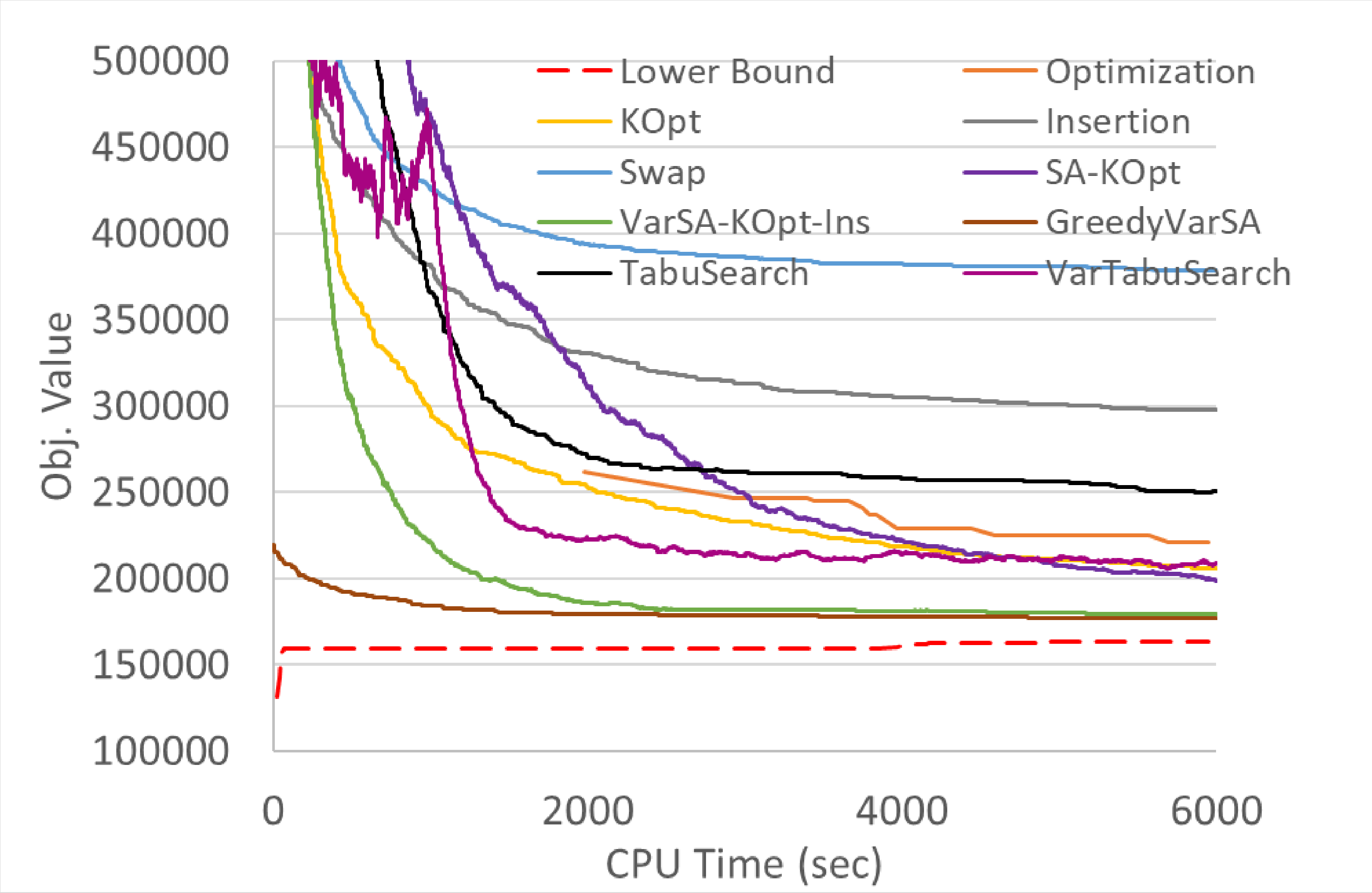


TSP Example (n=500)

- Size of the neighborood



Multistage Tabu Search




```
quartile = int(0.5 * (len(dist)- 1))

Variable N. Search
if cputime_i[-1]<1000:
    quartile = int(0.5 * (len(dist)- 1))
else:
    quartile = int(0.9 * (len(dist)- 1))
```

We could also propose an adaptive multistage procedure.
The size of the neighbourhood would be updated after n iterations without improvements

Exploration and Exploitation

Nuno Antunes Ribeiro

Assistant Professor

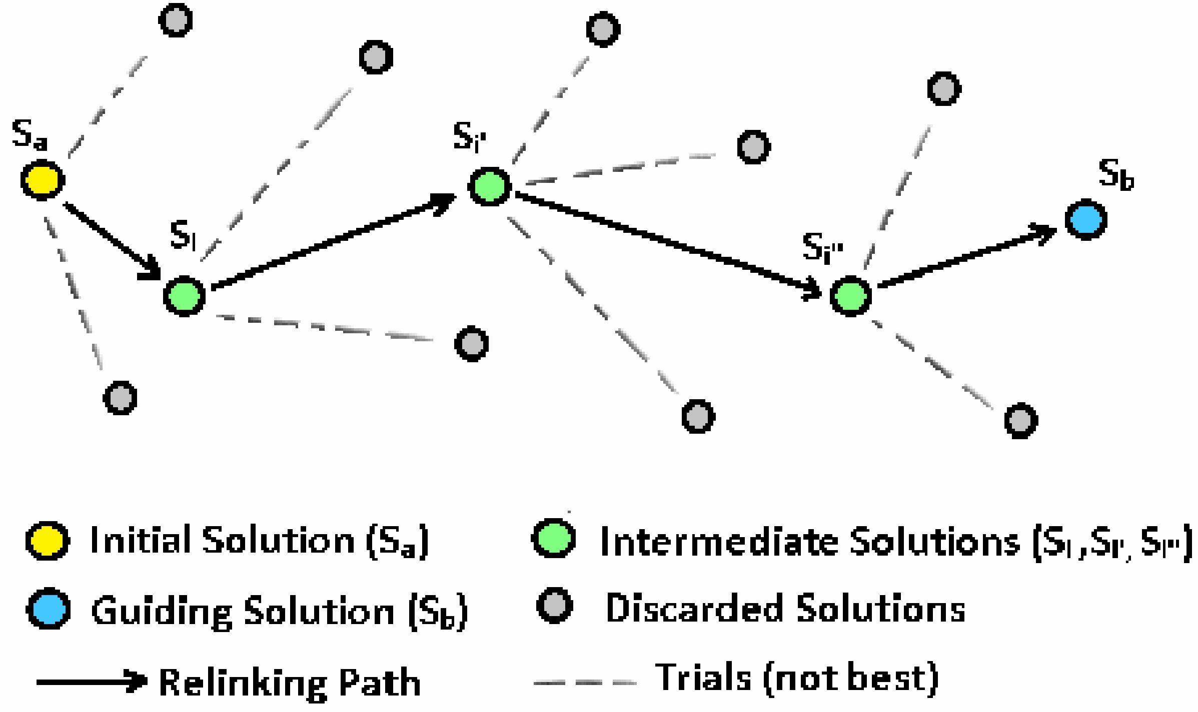
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN | Engineering Systems and Design

Medium Term Memory

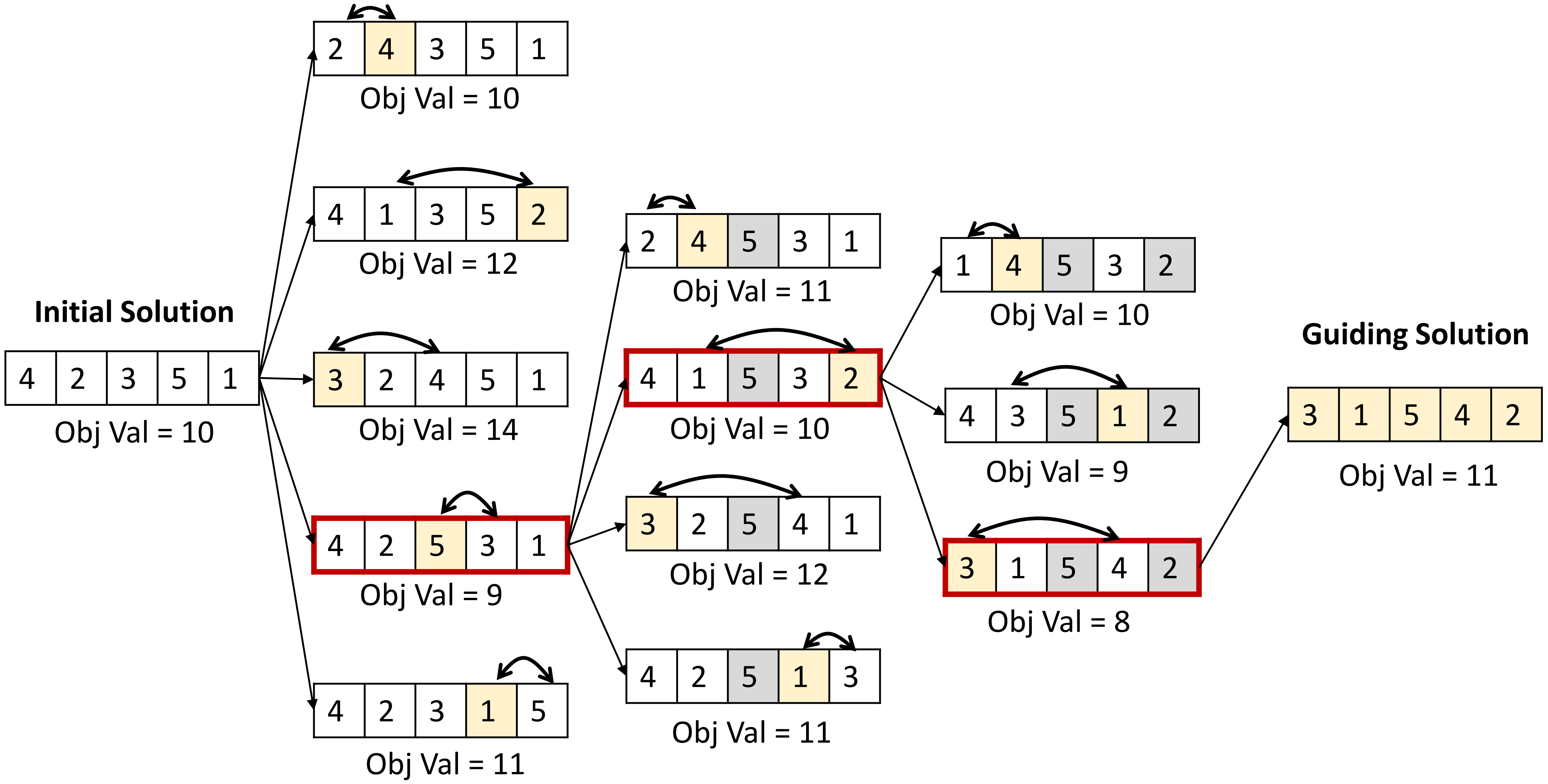
- Medium-term memory has been introduced in tabu search to encourage **exploitation** of the search.
- The role of medium-term memory is to exploit the information of the best-found solutions (**elite solutions**) to guide the search in promising regions of the search space.
- This information is stored in a medium-term memory. The idea consists in extracting the (common) features of the elite solutions and then intensifying the search around solutions sharing those features.
- **Path-relinking** is a common strategy used for exploitation.
- Path-relinking can also be integrated in other metaheuristics, such as the simulated annealing and/or GRASP metaheuristics

Path Relinking

- From the set of elite solutions (best solutions found so far), two solutions are randomly selected – one is designated as **initial solution** and the other as **guiding solution**
- A path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution
- At each step, all moves that incorporate attributes of the guiding solution are evaluated and the best move selected (**intermediate solution**).



Path Relinking



Long Term Memory

- Long-term memory has been introduced in tabu search to encourage the **exploration** of the search.
- The role of the long-term memory is to **force the search in non-explored regions** of the search space.
- The main representation used for the long-term memory is the **frequency memory**.
- Two popular diversification strategies may be applied:
  - **Continuous diversification:** This strategy introduces during a search a bias to encourage diversification
  - **Restart diversification:** This strategy consists in introducing in the current or best solution the least visited components. Then a new search is restarted from this new solution.

Continuous Diversification

- This strategy introduces during a search a bias to encourage diversification
  - v is the actual move value
  - v' is the penalized move value
  - w is a penalty factor
  - q is the frequency ratio

$$v' = \begin{cases} v & \textit{if solution improves} \\ v(1 \pm wq) & \textit{if solution does not improve} \end{cases}$$

Bit Move	Frequency Ratio
1	0.1
2	0.3
3	0.05
4	0.35
5	0.2

Current Solution

1	0	0	1	0

Obj. Value
6

Best Descent					Move Value	Penalized Move Value
0	0	0	1	0	10	10 (1+0.1)=11
1	**1**	0	1	0	8	8 (1+0.3)=10.4
1	0	**1**	1	0	10	10 (1+0.05)=10.5
1	0	0	**0**	0	7	7 (1+0.35)=9.45
1	0	0	1	**1**	8	**8 (1+0.2)=9.2**

Restart Diversification

- This strategy consists in introducing in the current or best solution the least visited components. Then a **new search is restarted** from this new solution.
- A **perturbation** is applied to the current solution considering the frequency memory of the search procedure so far.
- The **frequency memory** storesfor each component of the solution encoding the number of times the component is present in all visited solutions
- Example:
  - How often a variable had assumed a value 1 in a binary problem
  - How often a variable has assumed a certain value in a discrete problem
  - How often an edge have been selected in a permutation problem
  - Etc.

Multistart vs Iterated Local Search

- In multistart local search, the initial solution is always chosen randomly and then is unrelated to the generated local optima.
- **Iterated Local Search** improves the classical multistart local search by perturbing the local optima and reconsidering them as initial solutions – Similar to restart diversification