# Solution Encoding and Search Operators

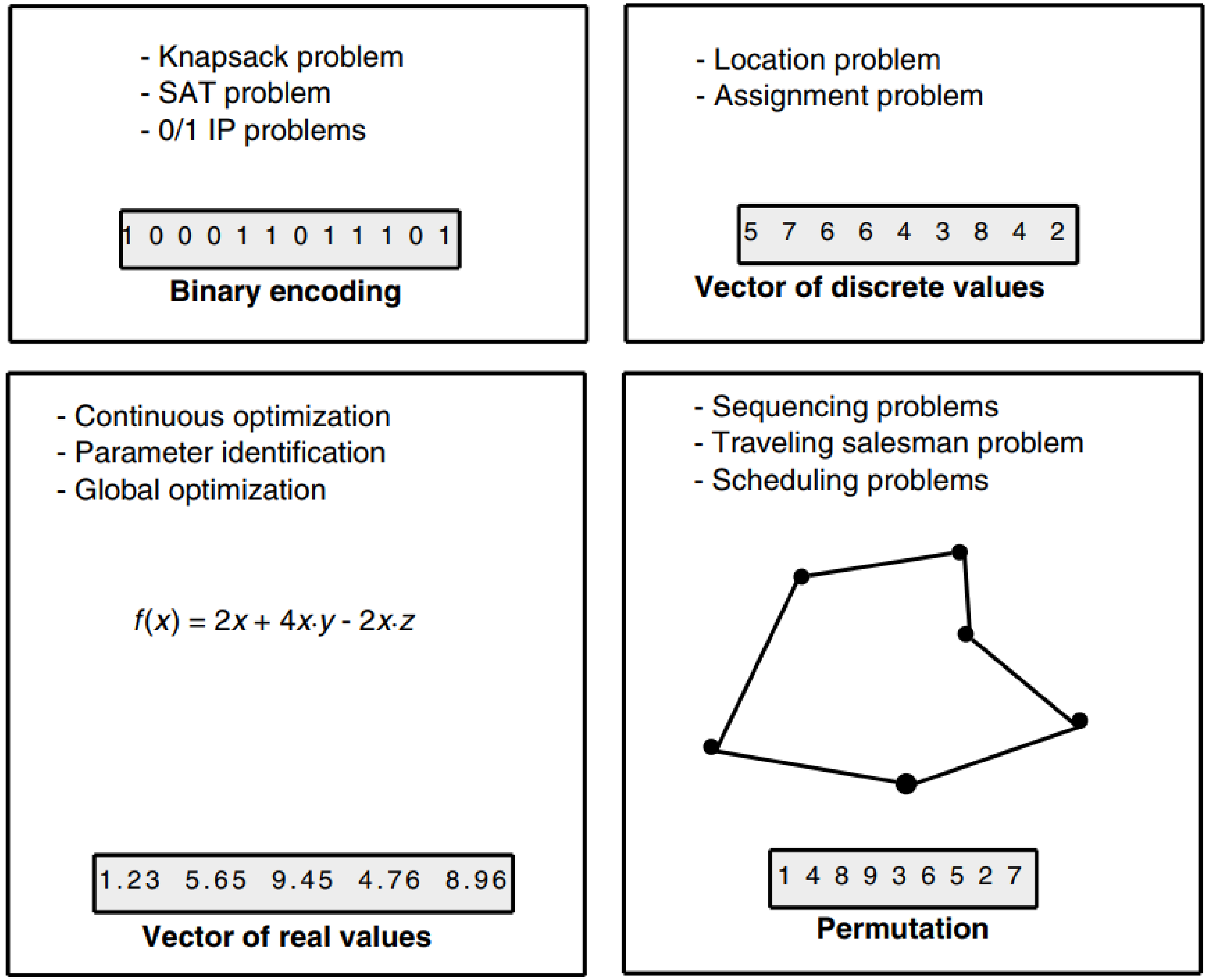

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN | Engineering Systems and Design

# Solution Encoding

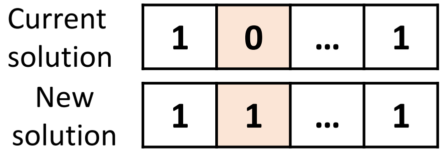
- Designing any iterative metaheuristic needs an **encoding** of a solution
- The encoding plays a major role in the efficiency and effectiveness of a metaheuristic procedure and constitutes an essential step in designing **any** metaheuristic.
- Many straightforward encodings may be applied for some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



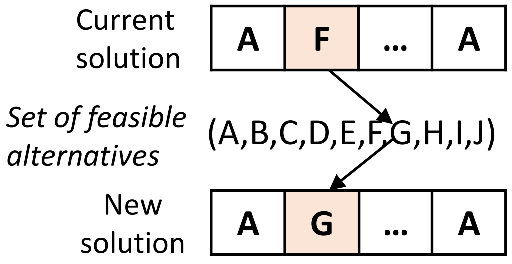
# Search Operator

- The efficiency of a solution encoding is also related to the **search operator**.
- When defining a solution encoding, one has to bear in mind how the solution will be **perturbed**.
- **Default search operators** are often considered – however more sophisticated search operators may be considered, especially when solving problems with large solution spaces and a significant number of constraints.

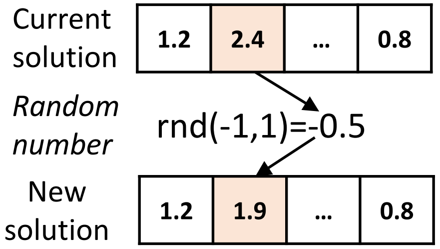
**Binary encoding –** flip n bits of the solution (typically 1 or 2 bits)



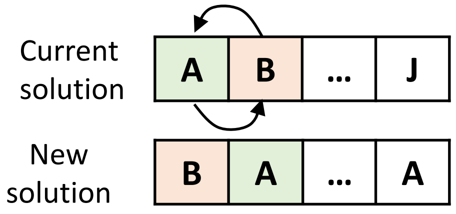

**Discrete encoding –** update n bits of the solution by randomly generating a new value (typically 1 or 2 bits)




**Real encoding –** update n bits of the solution by randomly generating a new value within a certain range (typically 2 elements)




**Permutation encoding –** swap the location of n elements (typically 2 elements)

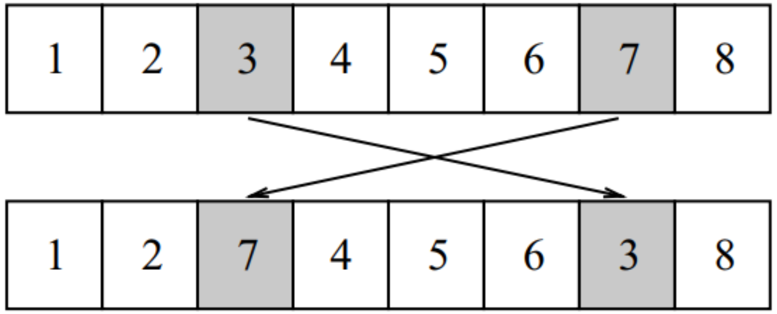


# Search Operators in Permutation Problems

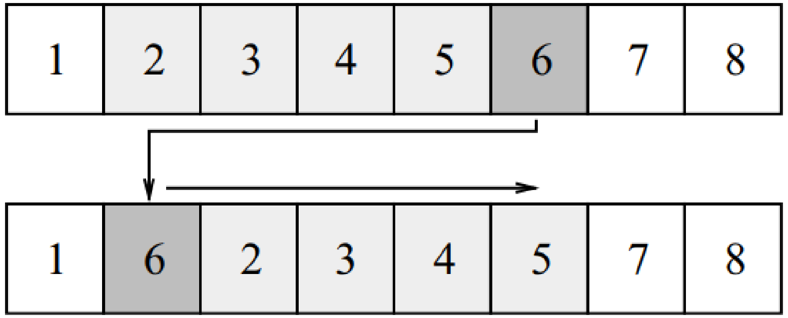
- Many **sequencing, scheduling, planning and routing problems** are considered as permutation problems
- There are two main types of permutation problems:
  - **Priority Problems** (e.g. **scheduling**) - In these problems permutations represent a priority queue and the position in the solution is important
  - **Adjacency Problems** (e.g. **TSP**) - In these problems permutations represent an adjacency list - e.g. city "A" may be the first in the list, the second, or the nth element in the list; the solution is the same provided that all elements of the list are adjacent to the same pair of elements
- **The efficiency of a neighbourhood is related not only to the representation but also the type of problems to solve and corresponding search operators**

# Search Operators in Permutation Problems

- Swap Operator

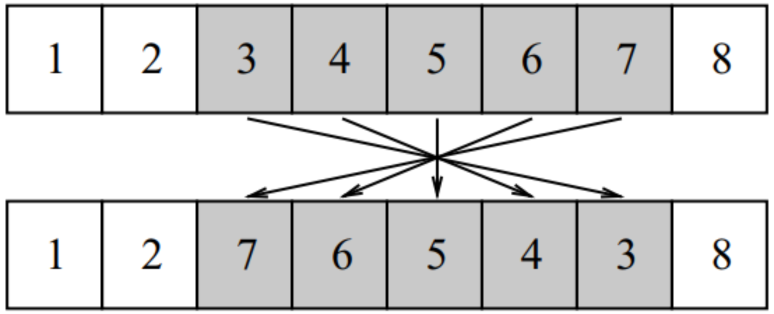



- Insertion Operator



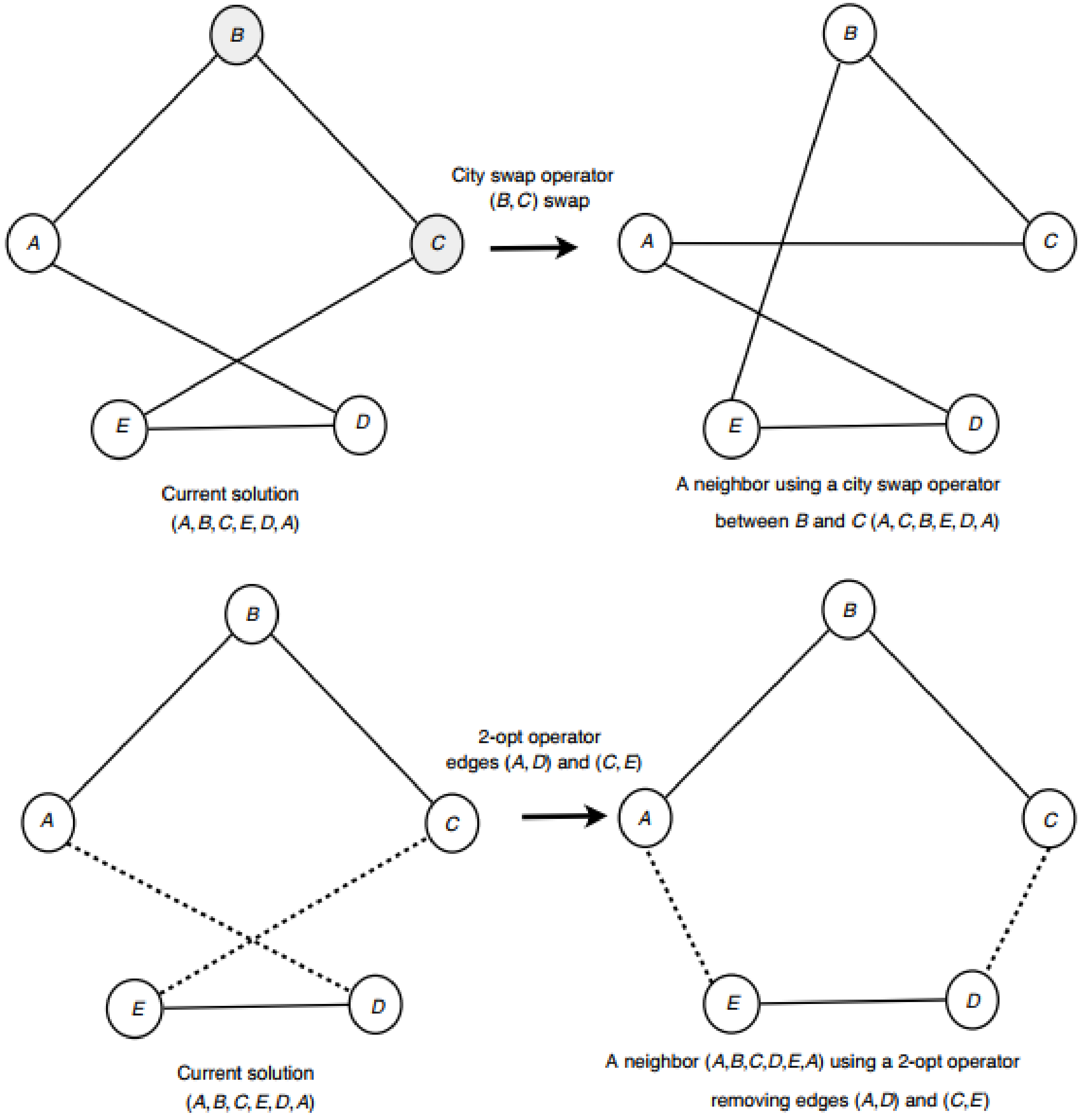

**Used in both: Priority and Adjacency Problems**

- Inversion Operator



# Search Operators in Adjacency Problems

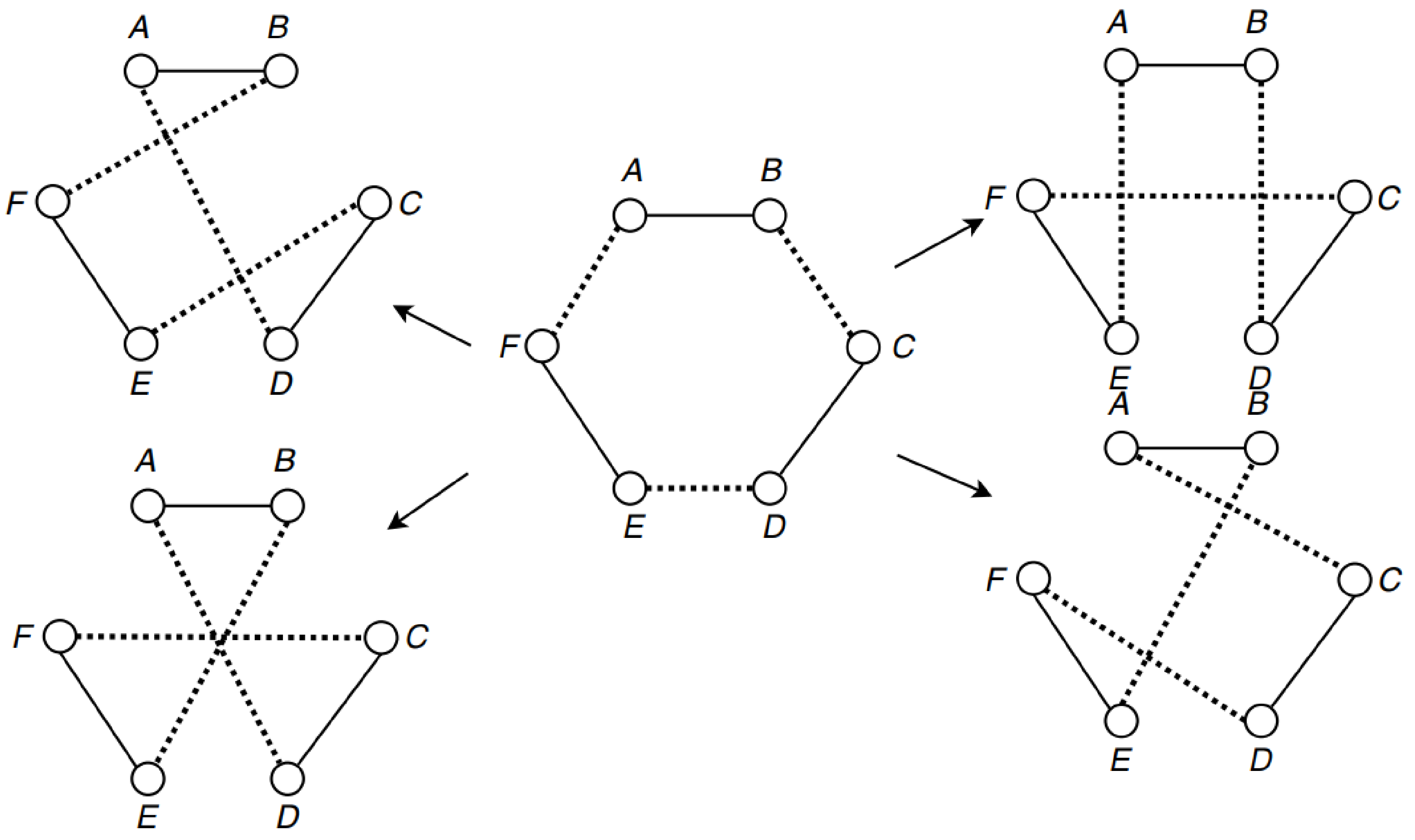
- 2-Opt




**Only used in Adjacency Problems**

# Search Operators in Adjacency Problems

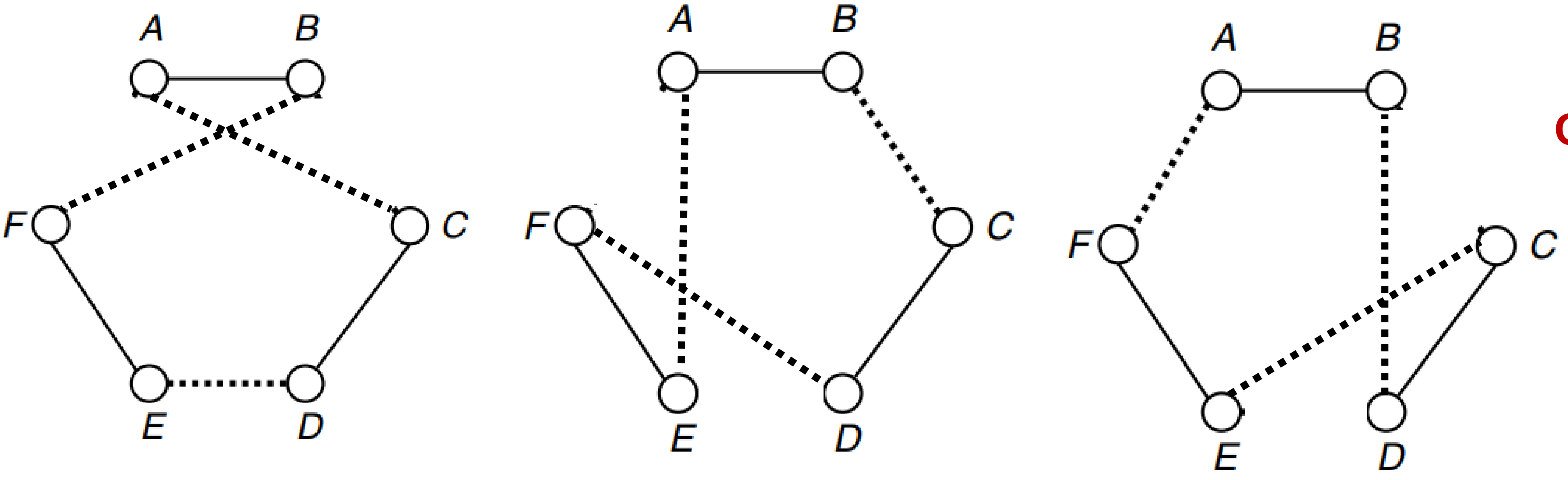
- 3-Opt




**Only used in Adjacency Problems**

# Search Operators in Adjacency Problems

- 3-Opt (2Opt Solutions)




**Only used in Adjacency Problems**

3-Opt Neighbourhood – 7 Solutions

# Search Operators in Adjacency Problems

- 3-Opt (Generalization)
  - Split the tour into 3 random parts (A – B – C)
  - 3-Opt Solutions:
    - A – inv(B) – inv(C)
    - A – C – B
    - A – C – inv(B)
    - A – inv(C) – B
  - 2-Opt Solutions:
    - A – inv(B) – C
    - A – B – inv(C)
    - A – inv(C) – inv(B)

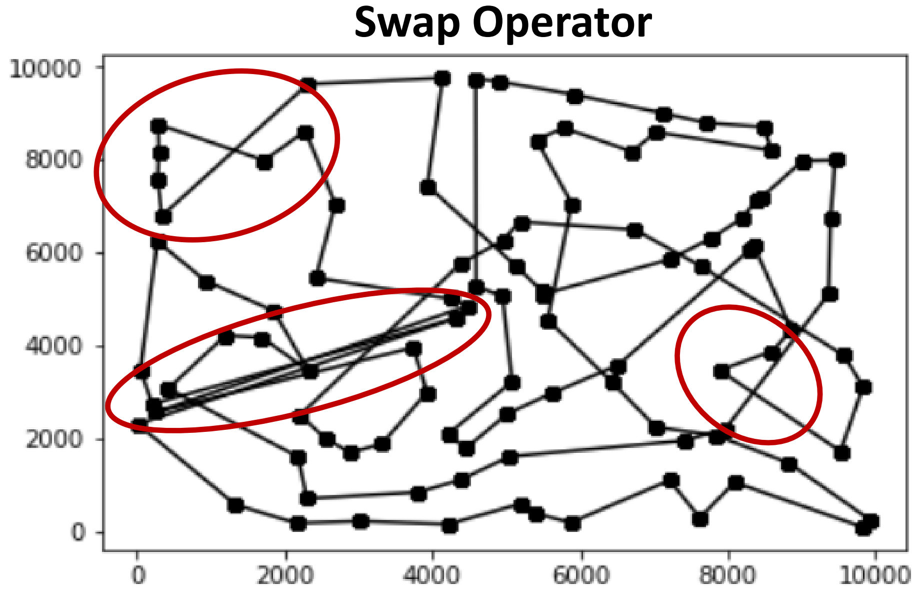
*inv stands for inversion

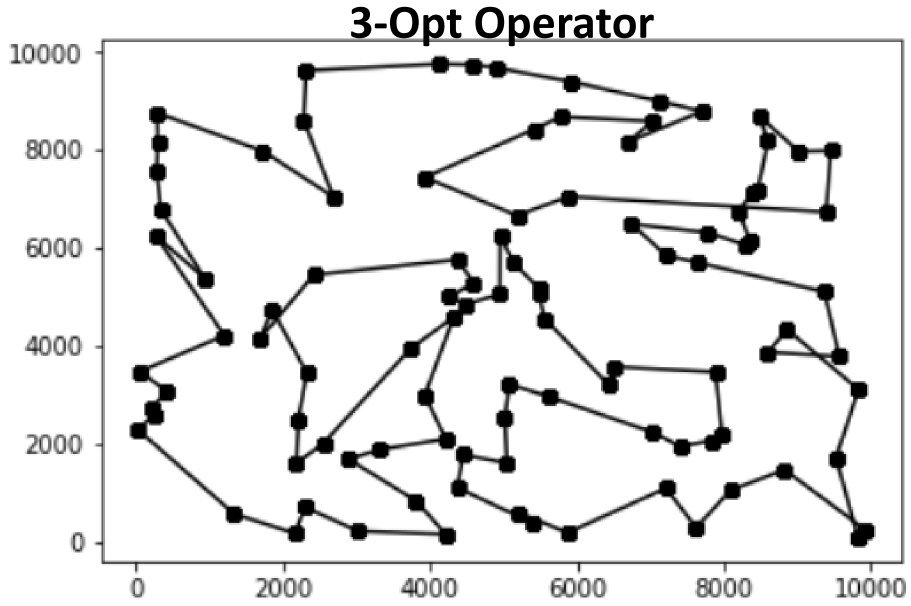
| | | A | | | B | | | C | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

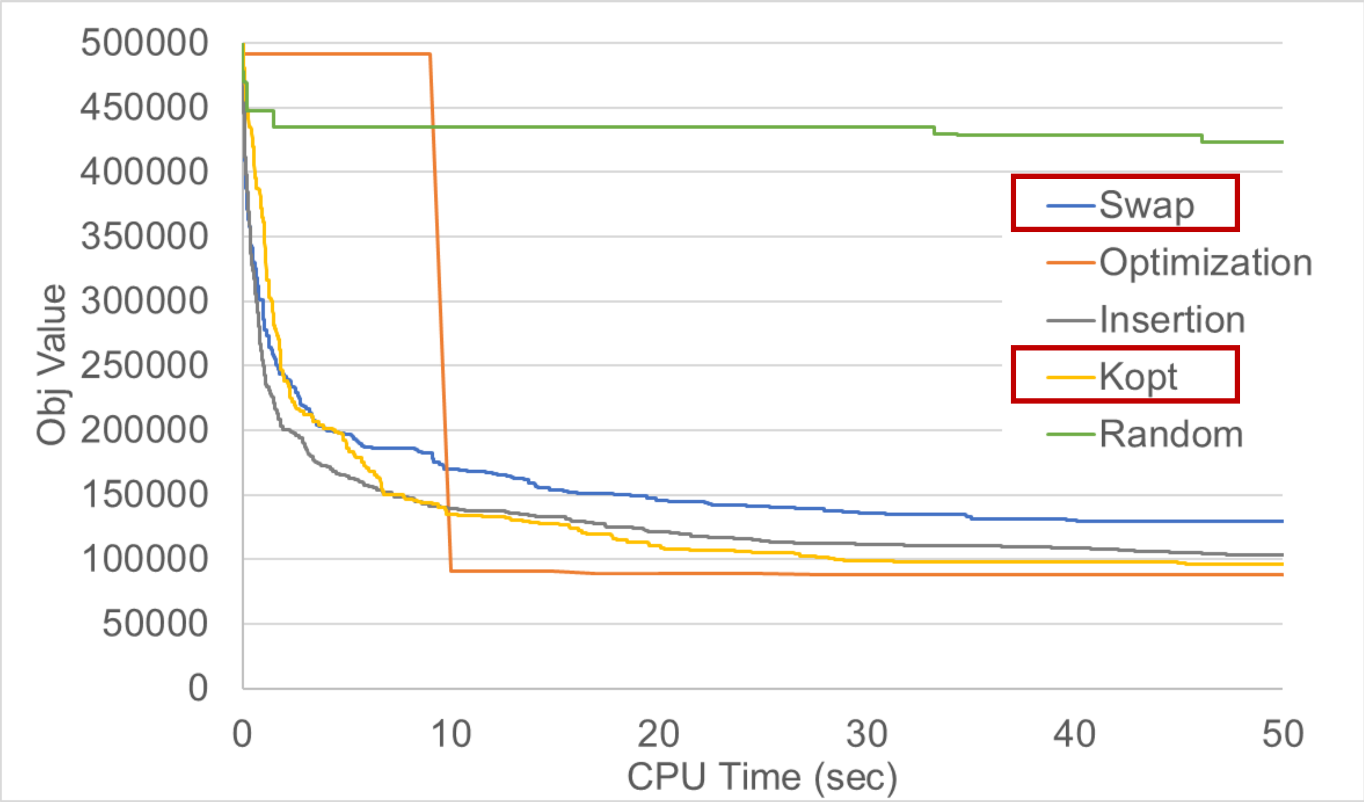
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 6 | 10 | 9 | 8 |
| 1 | 2 | 3 | 4 | 5 | 8 | 9 | 10 | 6 | 7 |
| 1 | 2 | 3 | 4 | 5 | 8 | 9 | 10 | 7 | 6 |
| 1 | 2 | 3 | 4 | 5 | 10 | 9 | 8 | 6 | 7 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 6 | 8 | 9 | 10 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 9 | 8 |
| 1 | 2 | 3 | 4 | 5 | 10 | 9 | 8 | 7 | 6 |

# Swap Operator vs 3-Opt Operator

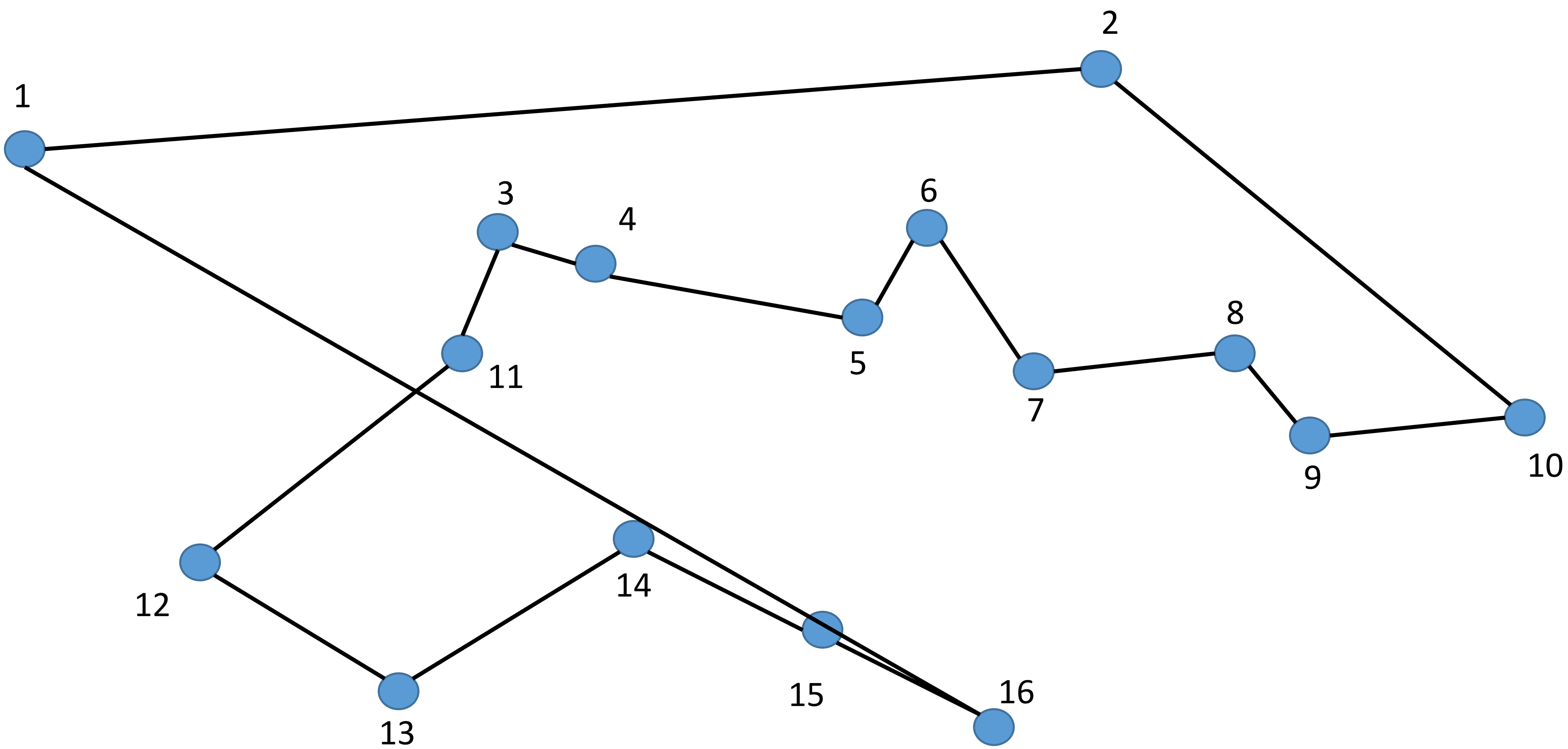


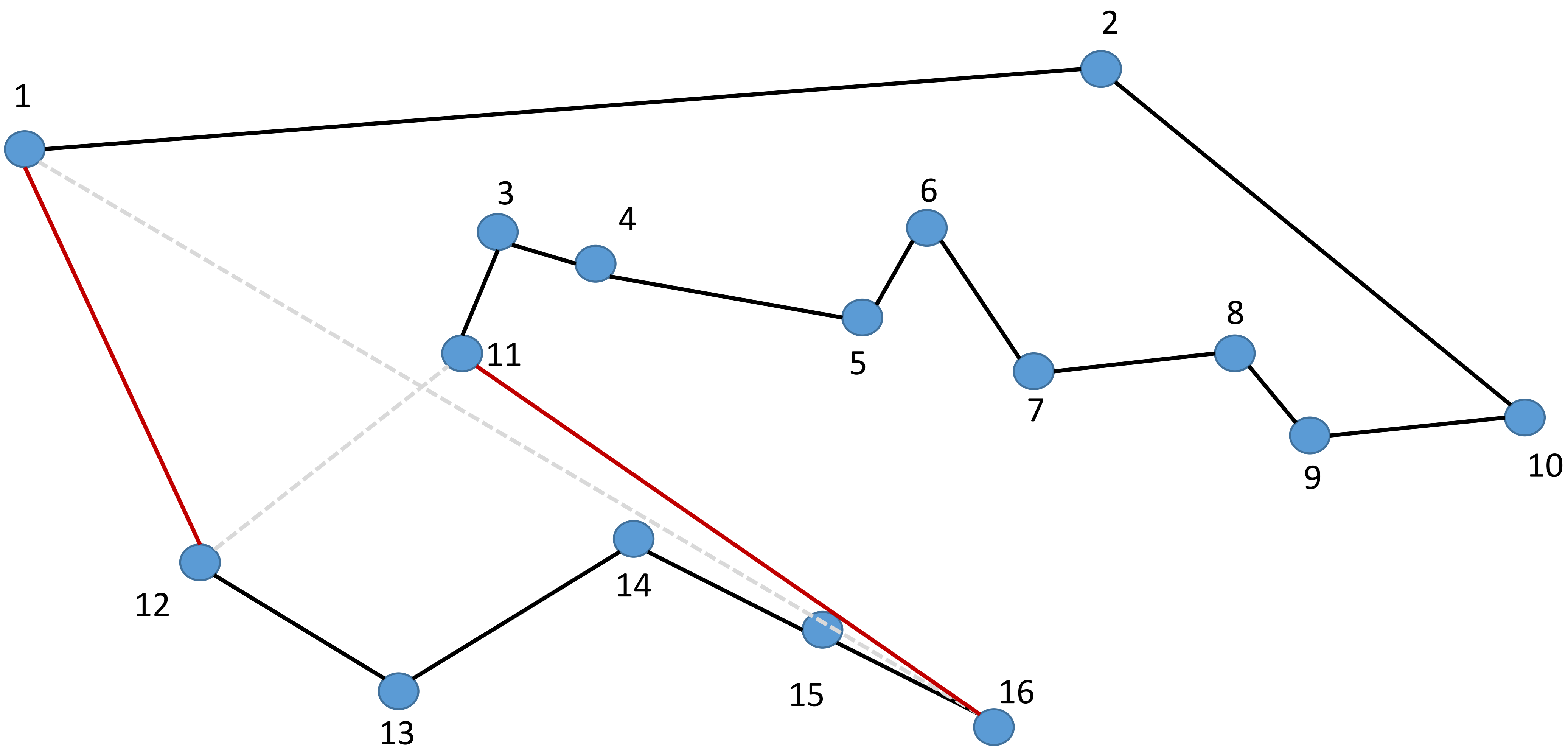
# Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **12** | **13** | **14** | **15** | **16** |



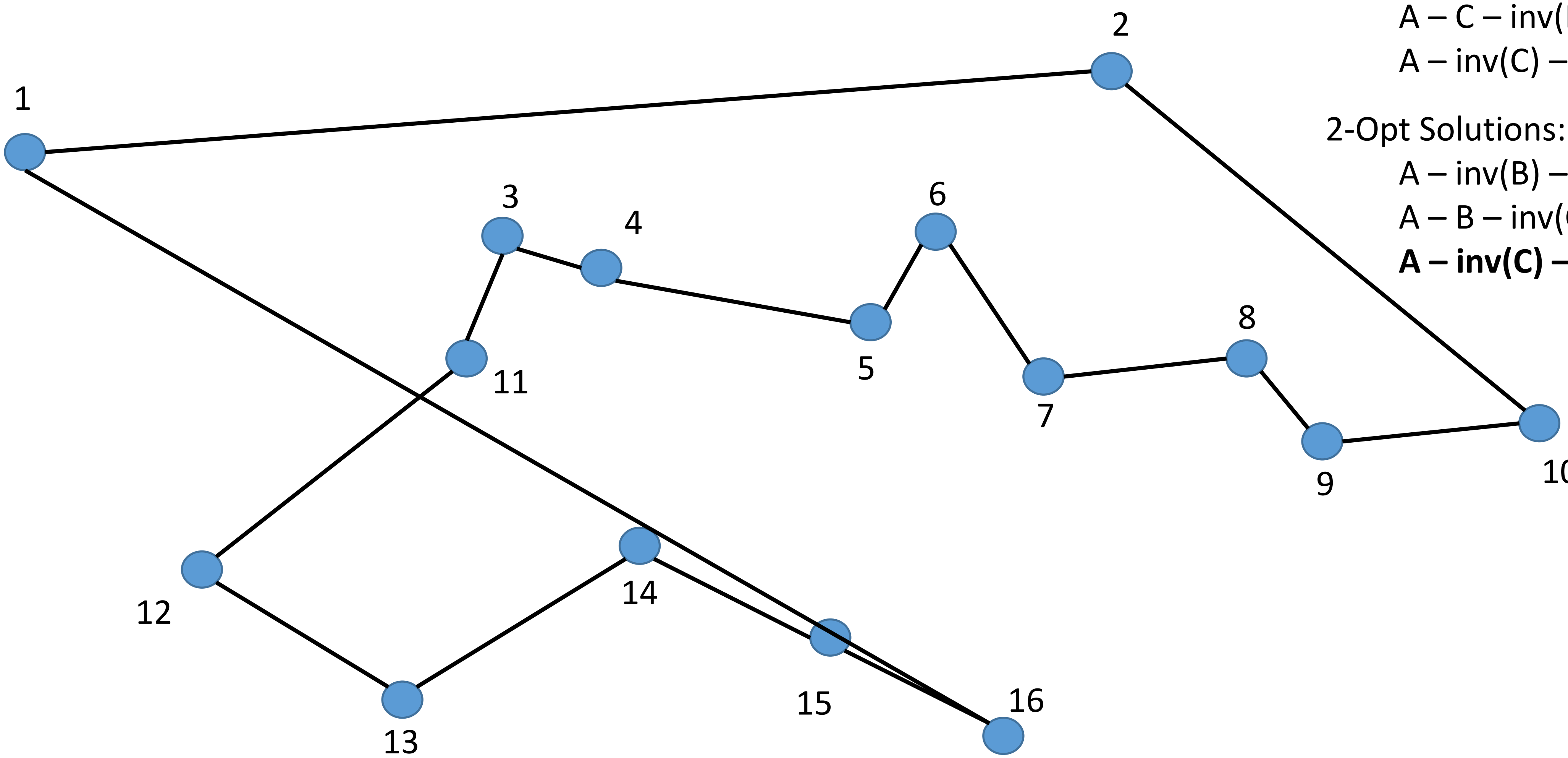
# Optimal Solution

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **12** | **13** | **14** | **15** | **16** |



# 3-Opt Operator

| | | | | A | | | | | | | | B | | C | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **12** | **13** | **14** | **15** | **16** |




3-Opt Solutions:

- A – inv(B) – inv(C)
- A – C – B
- A – C – inv(B)
- A – inv(C) – B

2-Opt Solutions:

- A – inv(B) – C
- A – B – inv(C)
- **A – inv(C) – inv(B)**

# 3-Opt Operator

| A | | | | | | | | | | | B | | | C | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **12** | **13** | **14** | **15** | **16** |

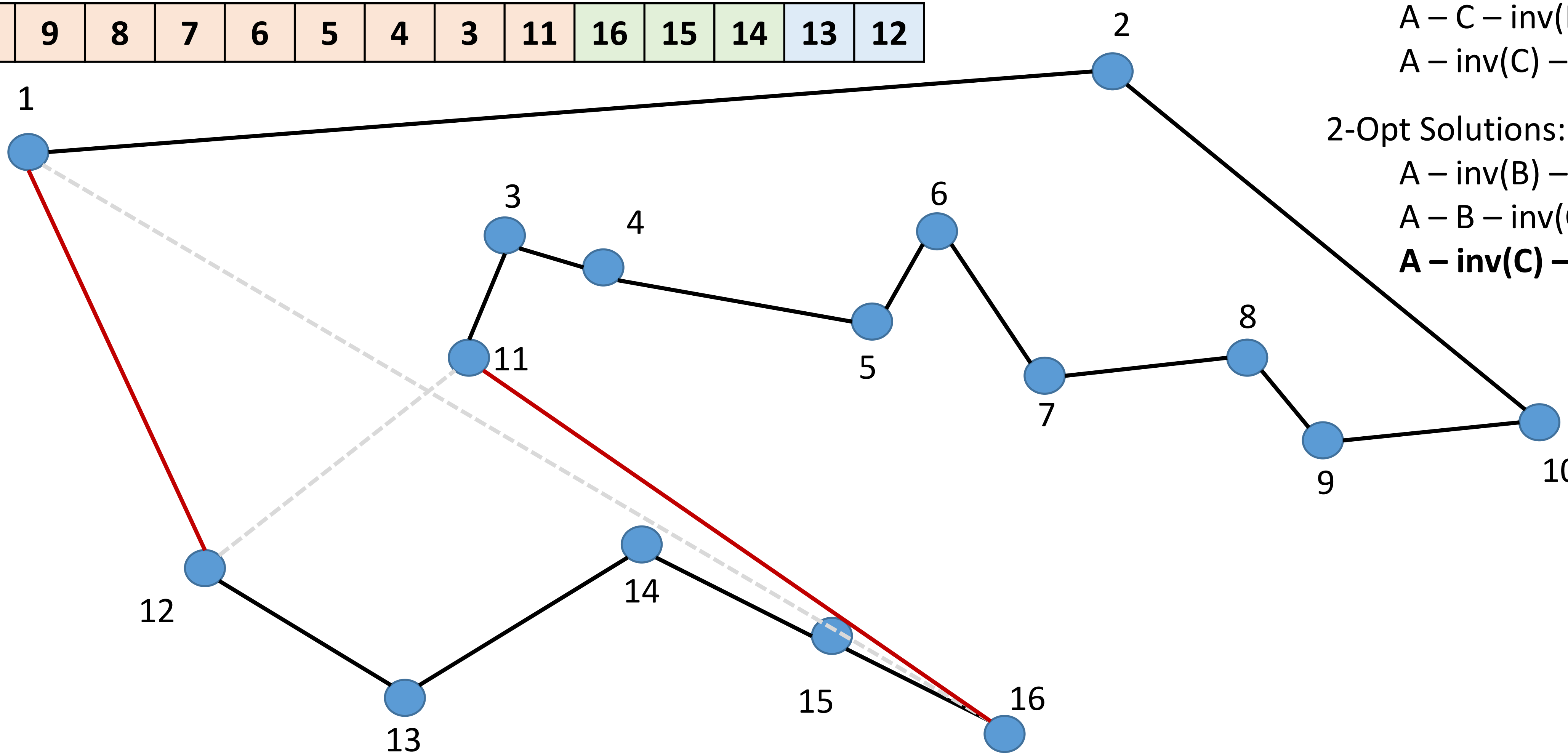
| A | | | | | | | | | | | rev(C) | | | rev(B) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **16** | **15** | **14** | **13** | **12** |

3-Opt Solutions:

- A – inv(B) – inv(C)
- A – C – B
- A – C – inv(B)
- A – inv(C) – B

2-Opt Solutions:

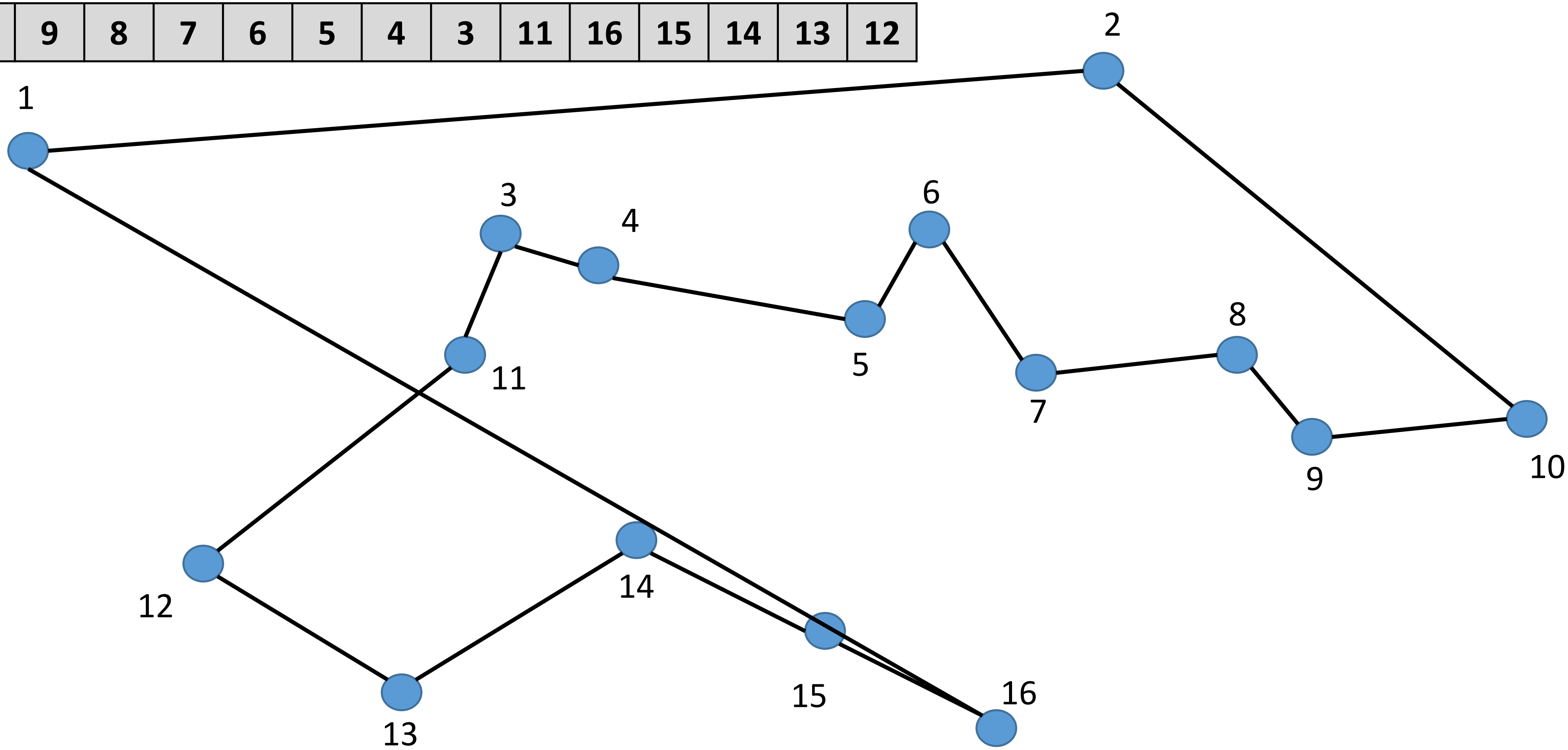
- A – inv(B) – C
- A – B – inv(C)
- **A – inv(C) – inv(B)**



# Swap Operator

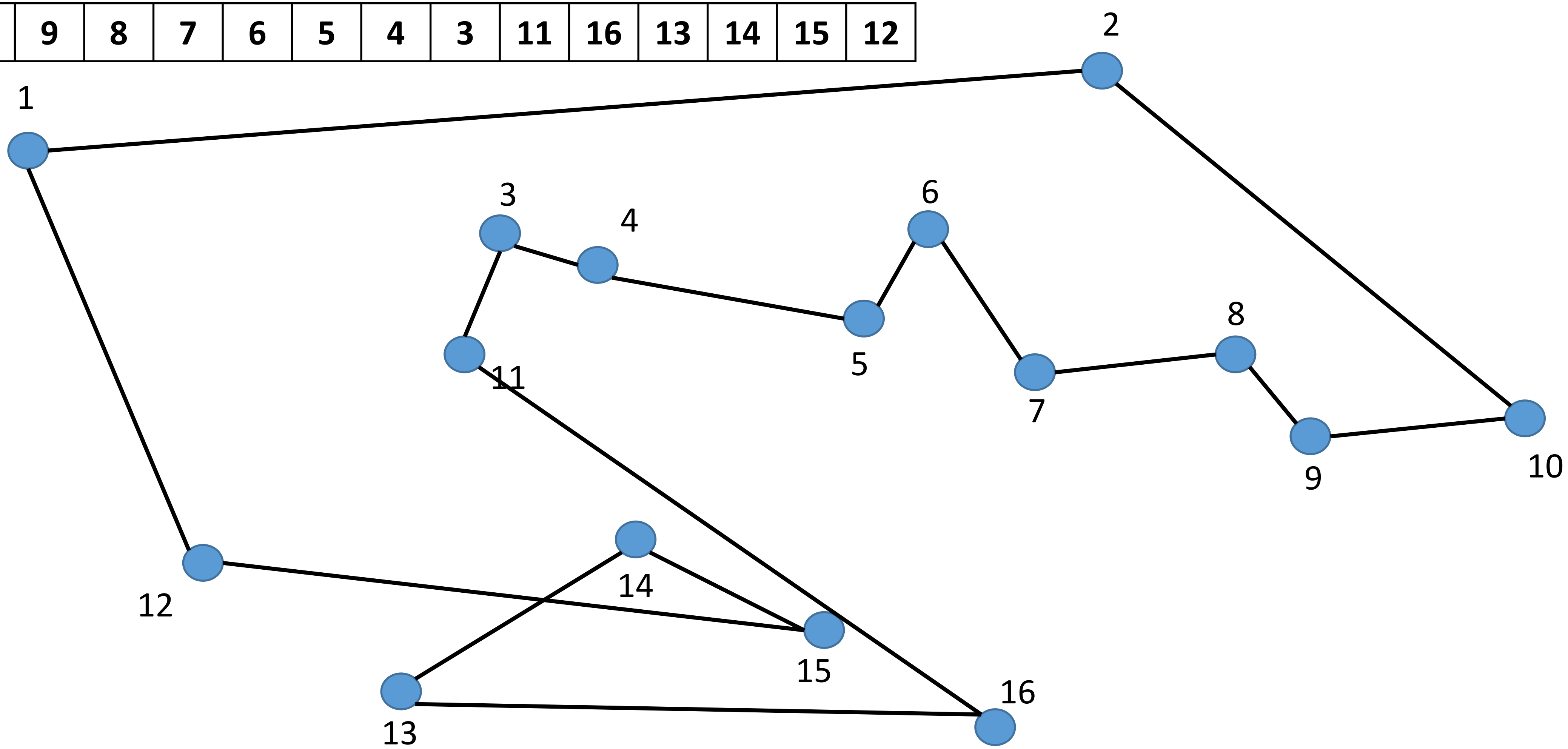
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **12** | **13** | **14** | **15** | **16** |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **16** | **15** | **14** | **13** | **12** |



# Swap Operator

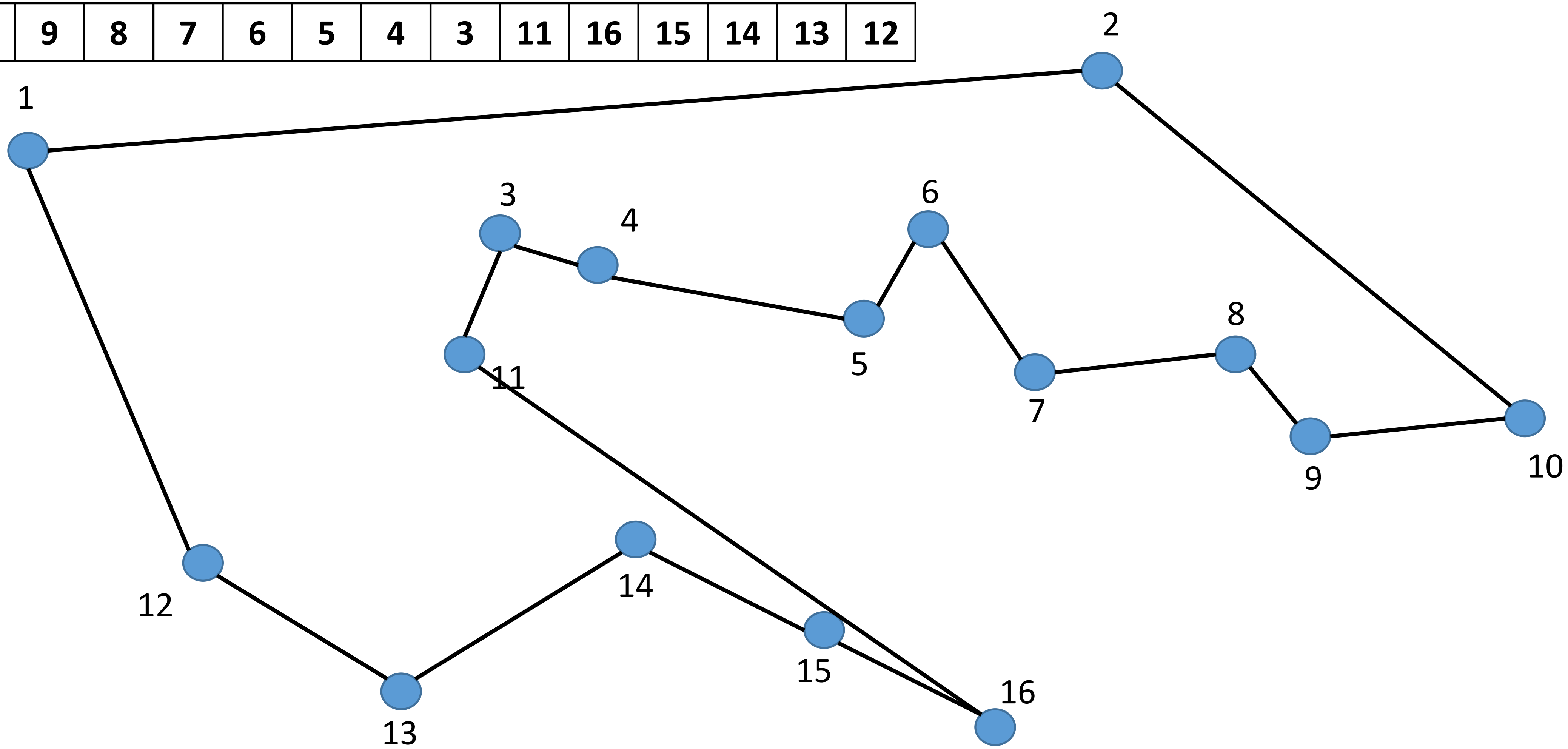
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 16 | 13 | 14 | 15 | 12 |



# Swap Operator

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **16** | **13** | **14** | **15** | **12** |

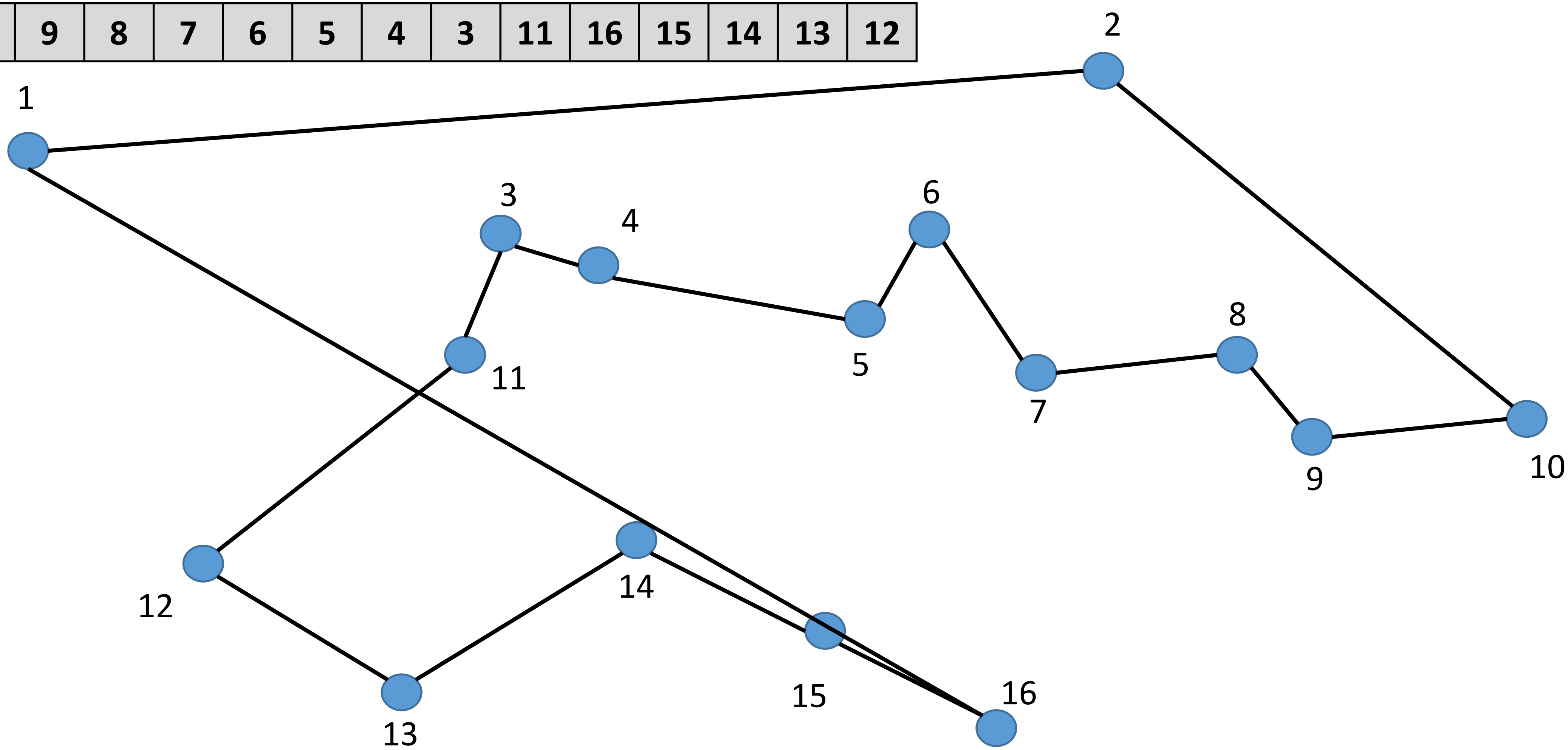
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **16** | **15** | **14** | **13** | **12** |



# Insertion Operator

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 12 | 13 | 14 | 15 | 16 |

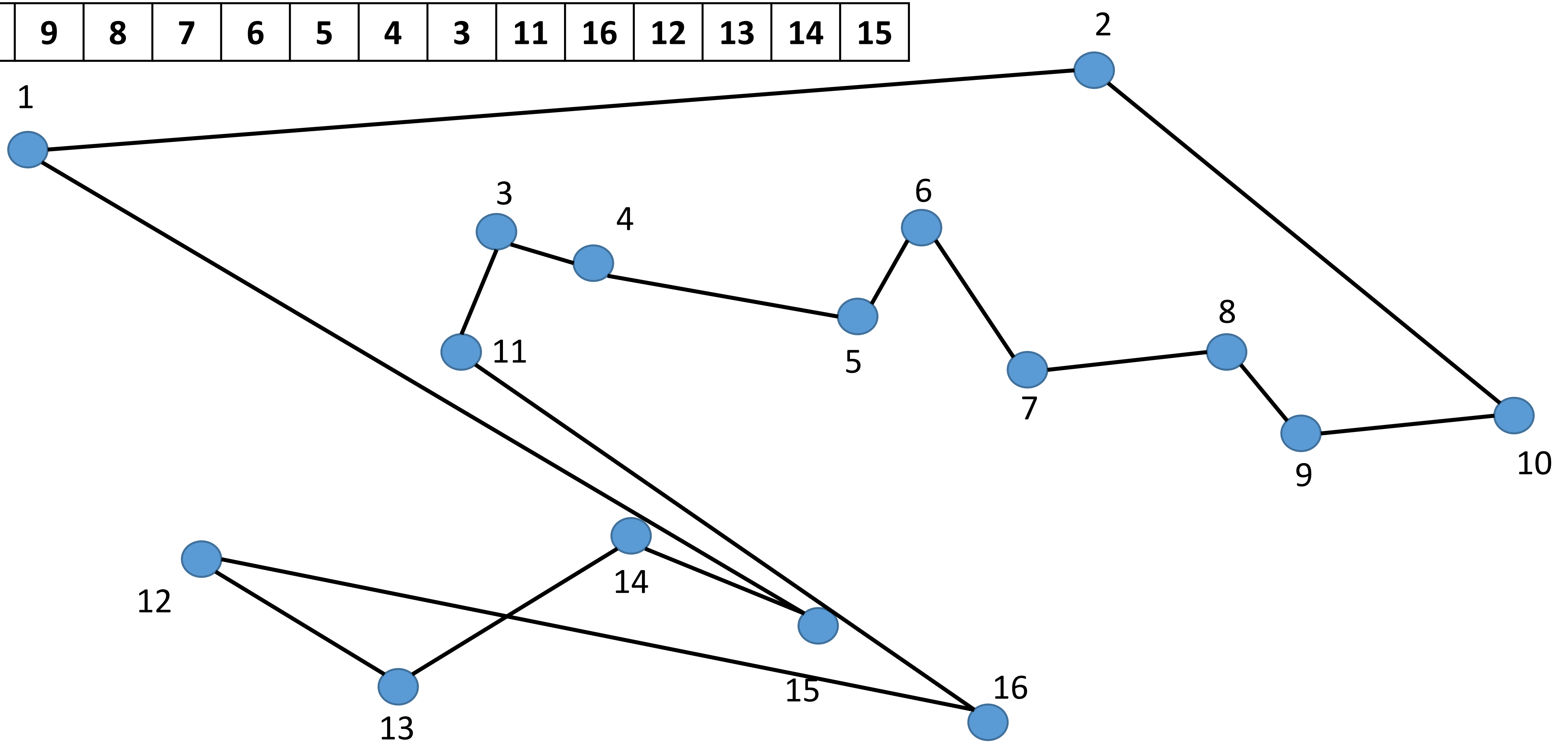
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 16 | 15 | 14 | 13 | 12 |



# Insertion Operator

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 12 | 13 | 14 | 15 | 16 |

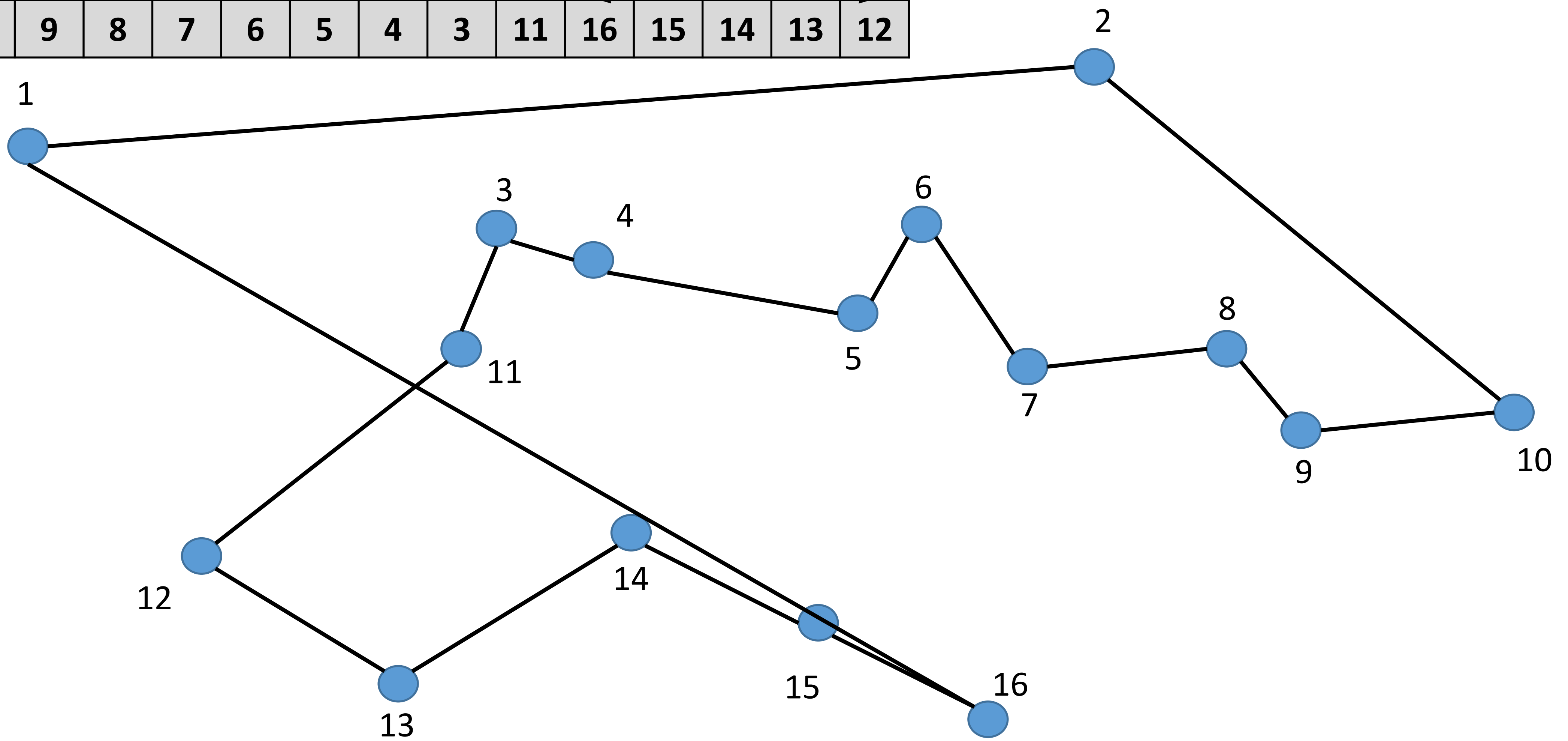
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 16 | 12 | 13 | 14 | 15 |



# Inversion Operator

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 12 | 13 | 14 | 15 | 16 |

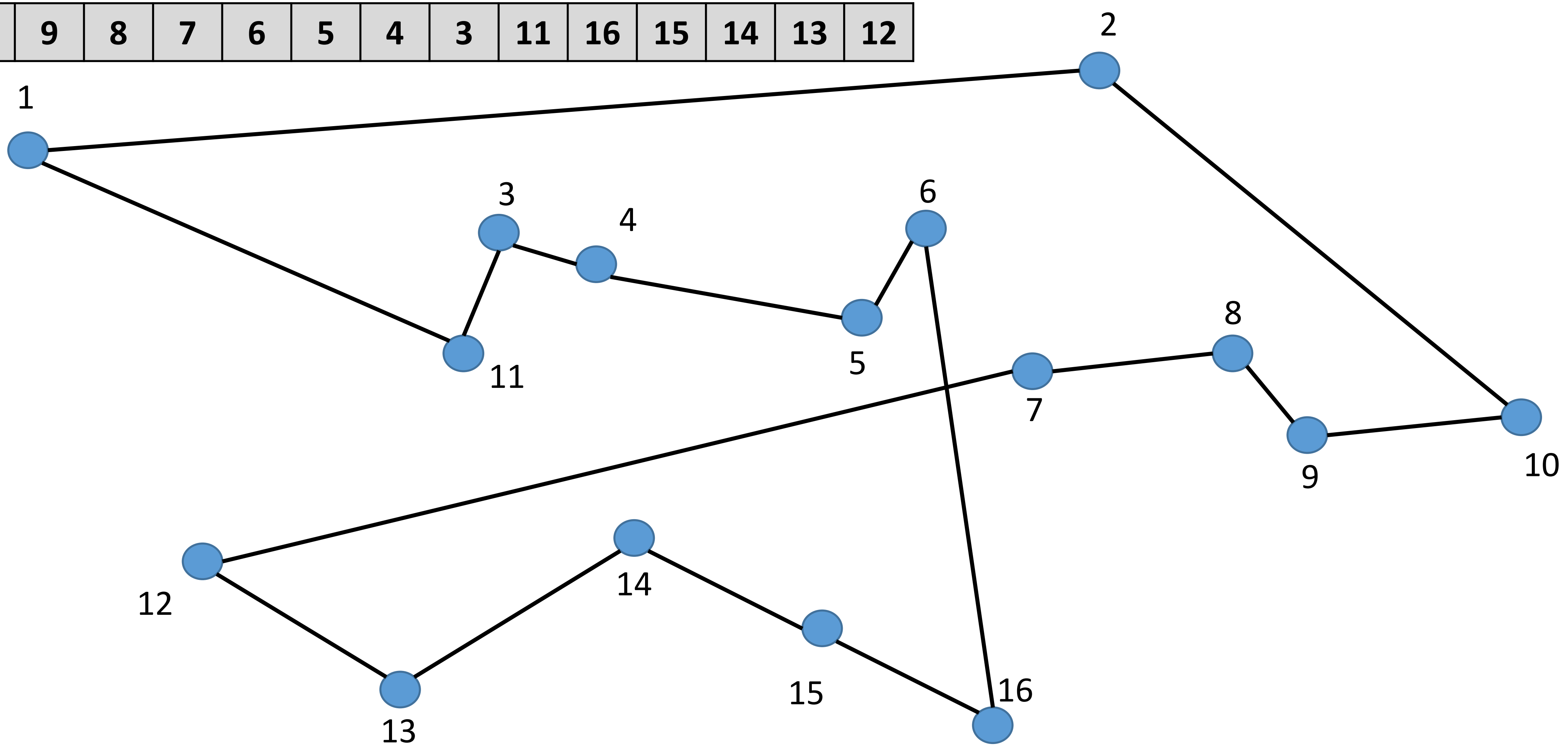
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 16 | 15 | 14 | 13 | 12 |



# Inversion vs 3-Opt Operator

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 12 | 13 | 14 | 15 | 16 | 6 | 5 | 4 | 3 | 11 |

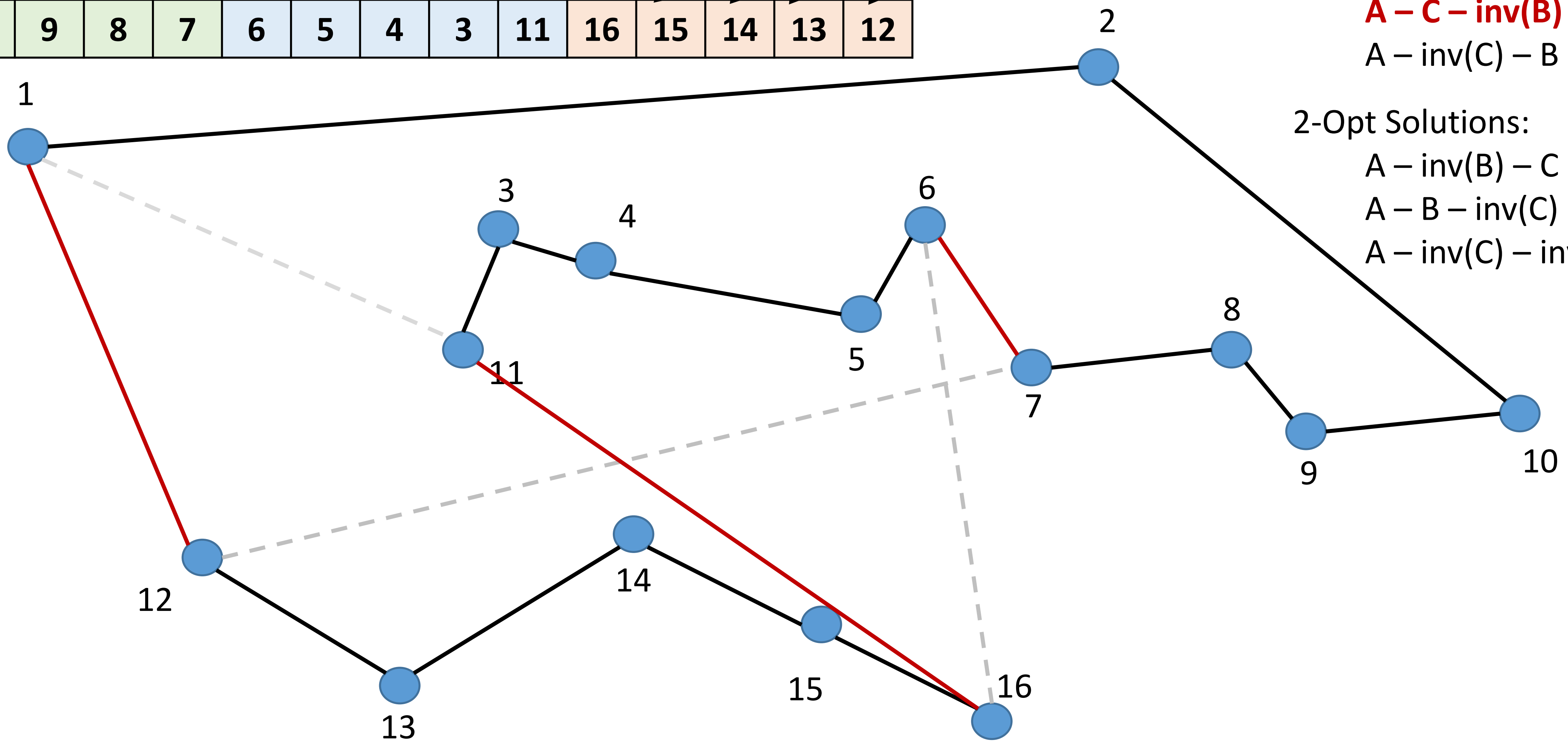
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 16 | 15 | 14 | 13 | 12 |



# Inversion vs 3-Opt Operator

| A | | | | | | B | | | | | C | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **12** | **13** | **14** | **15** | **16** | **6** | **5** | **4** | **3** | **11** |

| A | | | | | | C | | | | | rev(B) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **11** | **16** | **15** | **14** | **13** | **12** |




3-Opt Solutions:

- A – inv(B) – inv(C)
- A – C – B
- **A – C – inv(B)**
- A – inv(C) – B

2-Opt Solutions:

- A – inv(B) – C
- A – B – inv(C)
- A – inv(C) – inv(B)

# K-Opt Operator in Python

Nuno Antunes Ribeiro

Assistant Professor

Engineering Systems and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

# TSP Example

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

***Number of candidate locations n=100***

# TSP – Hill Climbing Generation Phase

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

**Select two random locations**
**Swap location in the permutation**

```python
#Insert Operator
def insert_random(seq):
    idx = range(len(seq))
    i1=random.sample(idx, 1)
    remove_index=np.where(np.array(seq)==i1)
    seq.pop(int(remove_index[0]))
    seq.insert(random.sample(idx, 1)[0], i1[0])
```

**Select location to insert (i1)**
**Remove from permutation list**
**Insert in a new position in the permutation list**

```python
#3-Opt
def k_opt(seq):
    global Solution_i
    idx = range(len(seq))
    i1=np.sort(random.sample(idx, 2))

    #Split in 3
    Opt1=seq[0:i1[0]]
    Opt2=seq[(i1[0]):i1[1]]
    Opt3=seq[(i1[1]):len(seq)]
    Opt2rev=Opt2[::-1]
    Opt3rev=Opt3[::-1]

    #3-Opt Solutions
    Sol1=Opt1+Opt2rev+Opt3rev
    Sol2=Opt1+Opt3+Opt2
    Sol3=Opt1+Opt3+Opt2rev
    Sol4=Opt1+Opt3rev+Opt2

    #2 Opt Solution
    Sol5=Opt1+Opt2rev+Opt3
    Sol6=Opt1+Opt2+Opt3rev
    Sol7=Opt1+Opt3rev+Opt2rev

    OptNeigh=[Sol1,Sol2,Sol3,Sol4,Sol5,Sol6,Sol7]

    ObjValue_Neigh=list();
    #Compute Obj Value of All Solutions
    for i_index in range(len(OptNeigh)):
        Solution_Neigh=OptNeigh[i_index]

        dfSolution_i=pd.DataFrame(Solution_Neigh)
        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_Neigh=np.append(ObjValue_Neigh,ObjValue)
    OptNeigh[np.argmin(ObjValue_Neigh)]
    Solution_i=OptNeigh[np.argmin(ObjValue_Neigh)]
```

**Follow the procedure explained in slide 43**

**Explore the whole neighbourhood (Best Descent) and select the best solution**

# 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])
```

**Apply search operator**

**Generation Phase**

**(Search Operator Code)**

**Compute Objective Value**

# TSP – Hill Climbing Replacement Phase

```python
    #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**

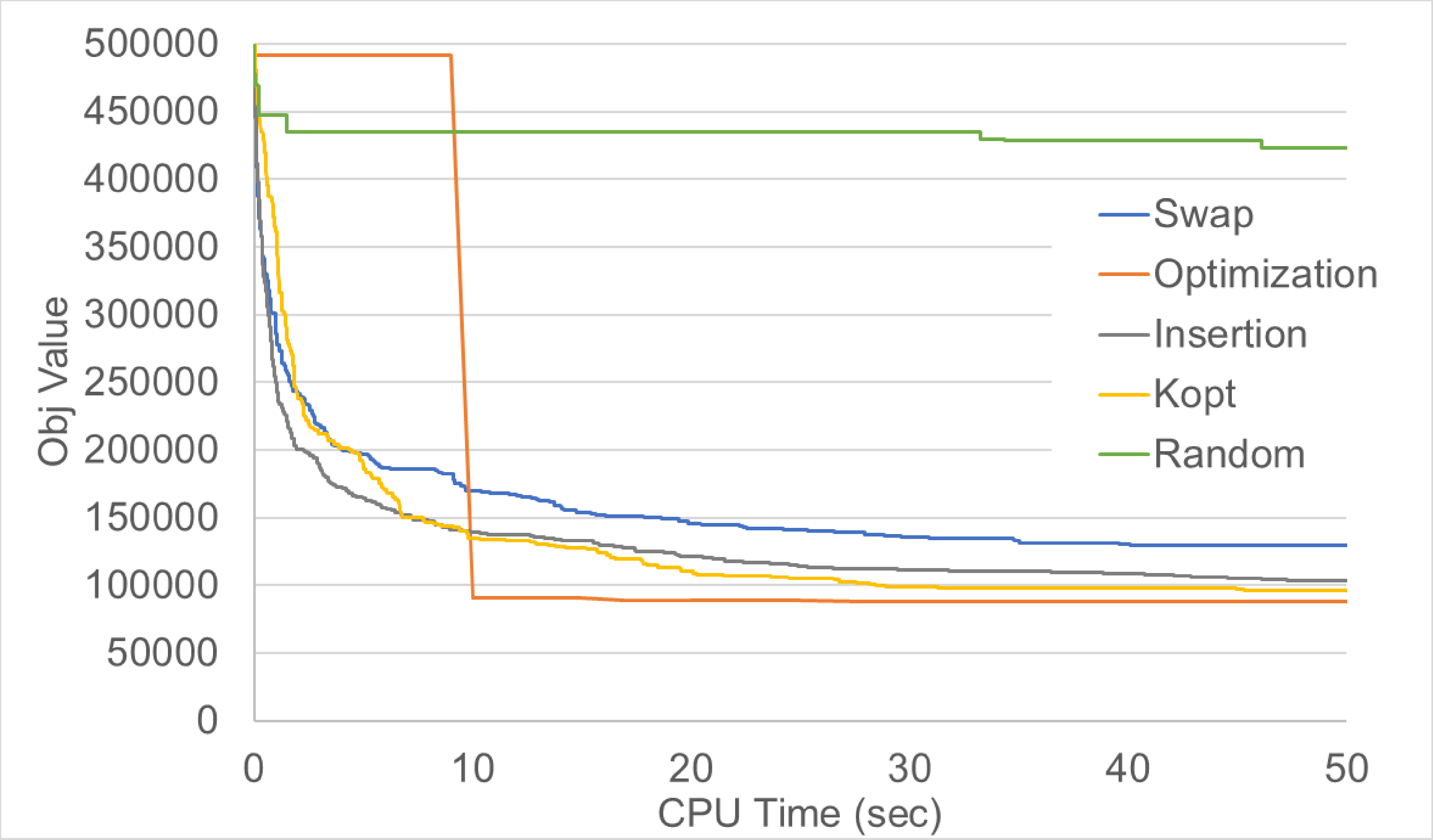
**Plot**

**Replacement Phase**

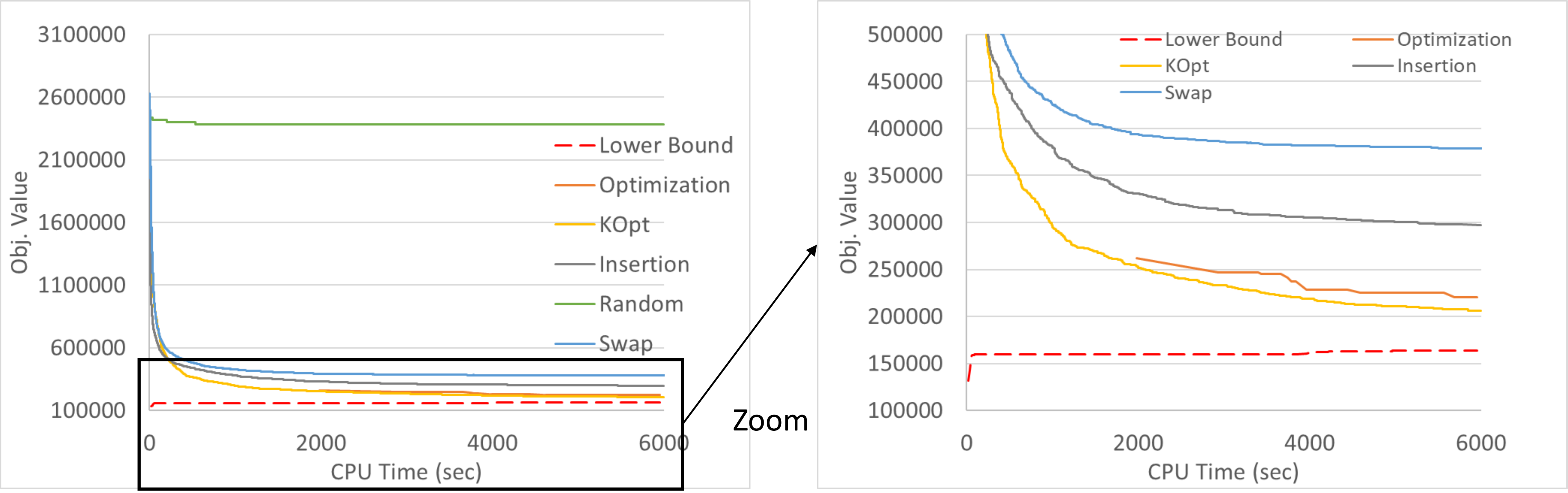
**(First Descent)**

**Keep trace of the objective values obtained over time**

# TSP Solution (n=100)



# TSP Solution (n=500)



# Constraint Handling


Nuno Antunes Ribeiro

Assistant Professor

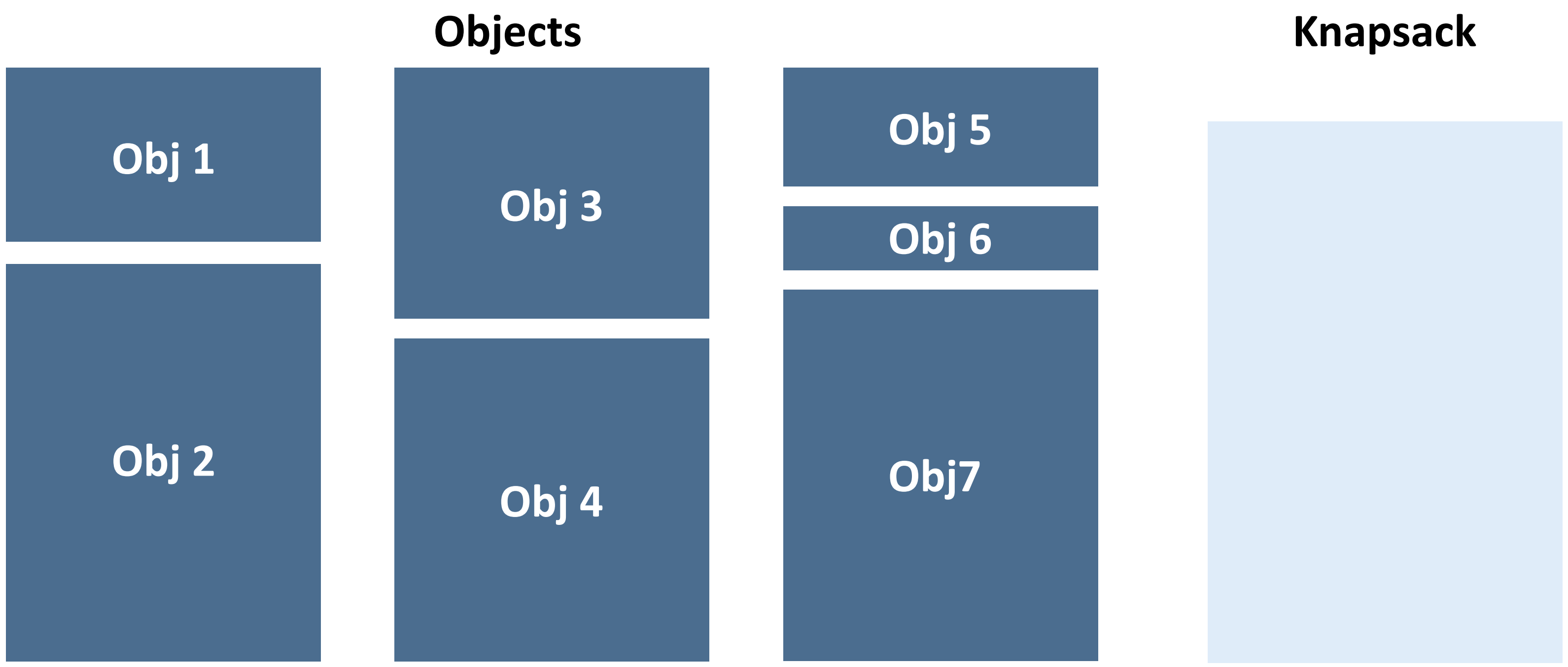
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

# Constraint Handling

- Dealing with constraints in optimization problems is an important topic for the efficient design of metaheuristics.
- Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints.

**Example Knapsack Problem:**




**Solution Encoding**

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

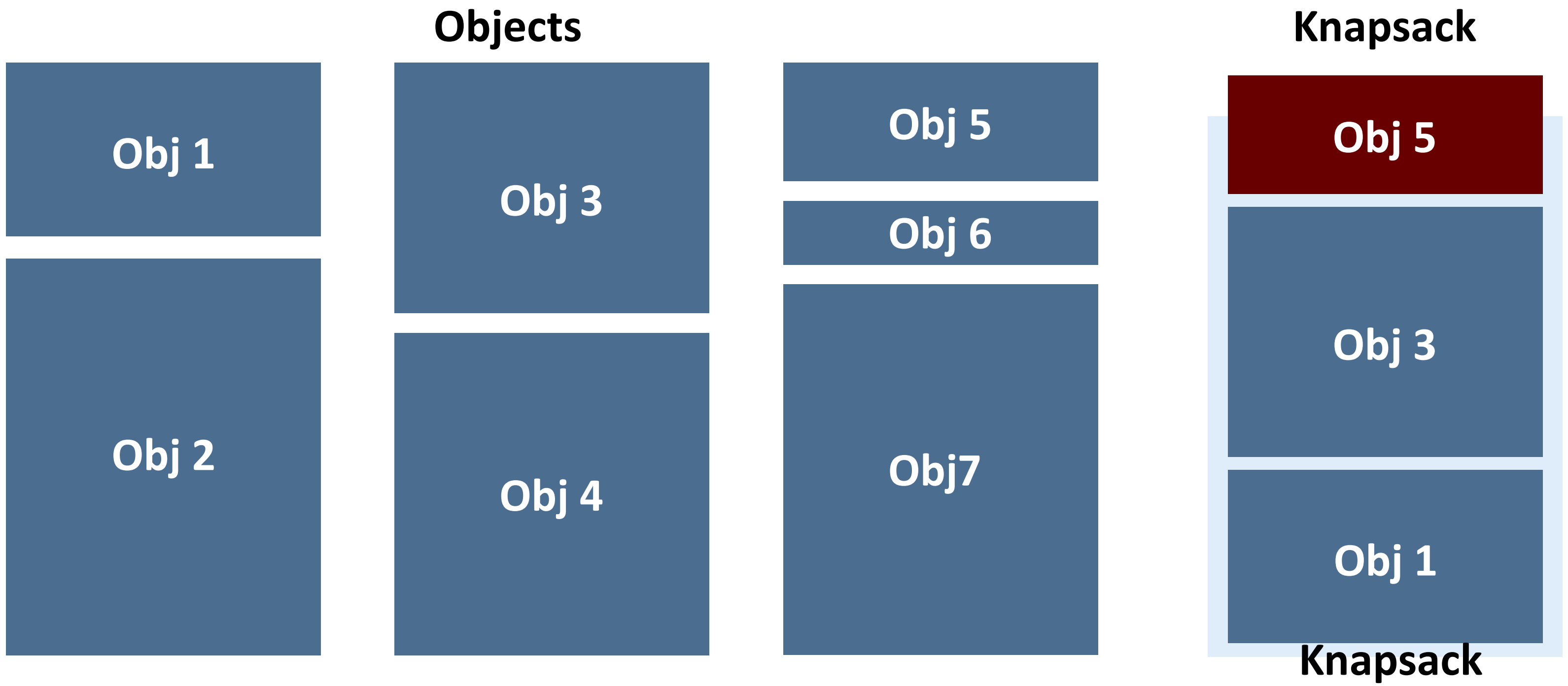
**Search Operator**

Flip n bits of the solution

# Constraint Handling

- Dealing with constraints in optimization problems is an important topic for the efficient design of metaheuristics.
- Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints.

**Example Knapsack Problem:**




**Solution Encoding**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |

**Search Operator**

Flip n bits of the solution

**Infeasible Solution**

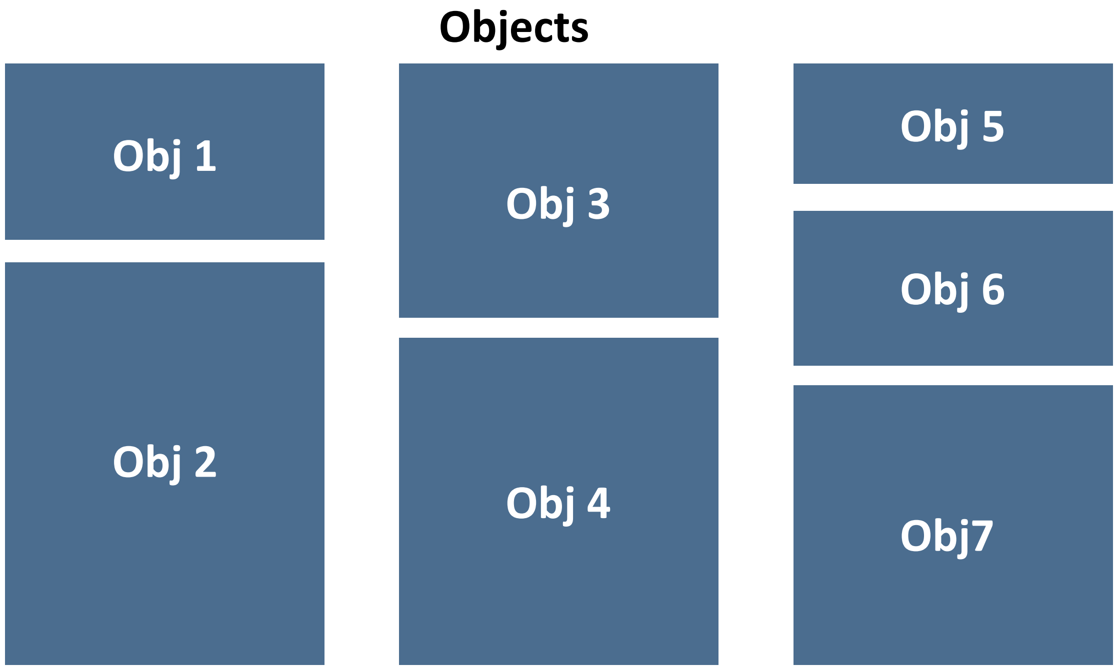
# Constraint Handling Techniques

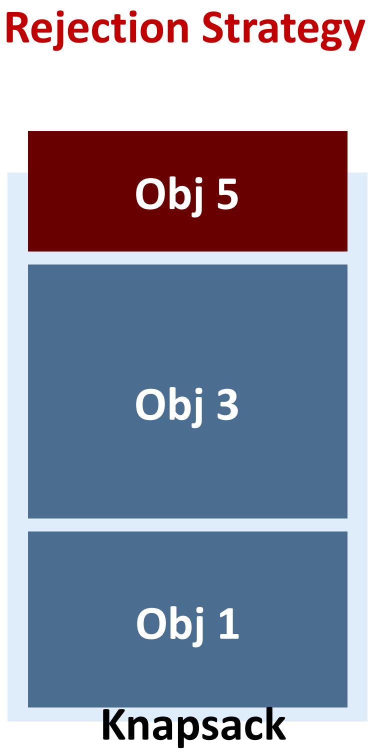
- **Reject Strategies:** represent a simple approach, where **only feasible solutions are kept during the search** and then infeasible solutions are automatically discarded. This kind of strategies are conceivable if the portion of infeasible solutions of the search space is very small.
- **Repairing strategies:** A **repairing procedure** is applied to infeasible solutions to generate feasible ones (e.g. extracting from the knapsack some elements to satisfy the capacity constraint in the knapsack problem)
- **Penalizing Strategies:** reject strategies do not exploit infeasible solutions. Indeed, it would be interesting to use some information on infeasible solutions to guide the search. In penalizing strategies, infeasible solutions are considered during the search process. The unconstrained objective function is extended by a **penalty function that will penalize infeasible solutions**
- **Preserving Strategies:** In preserving strategies for constraint handling, a **specific representation and operators will ensure the generation of feasible solutions.** They incorporate problem-specific knowledge into the representation and search operators to generate only feasible solutions

# Reject Strategies

- Dealing with constraints in optimization problems is an important topic for the efficient design of metaheuristics.
- Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints.

**Example Knapsack Problem:**




**Solution Encoding**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |

**Search Operator**

Flip n bits of the solution

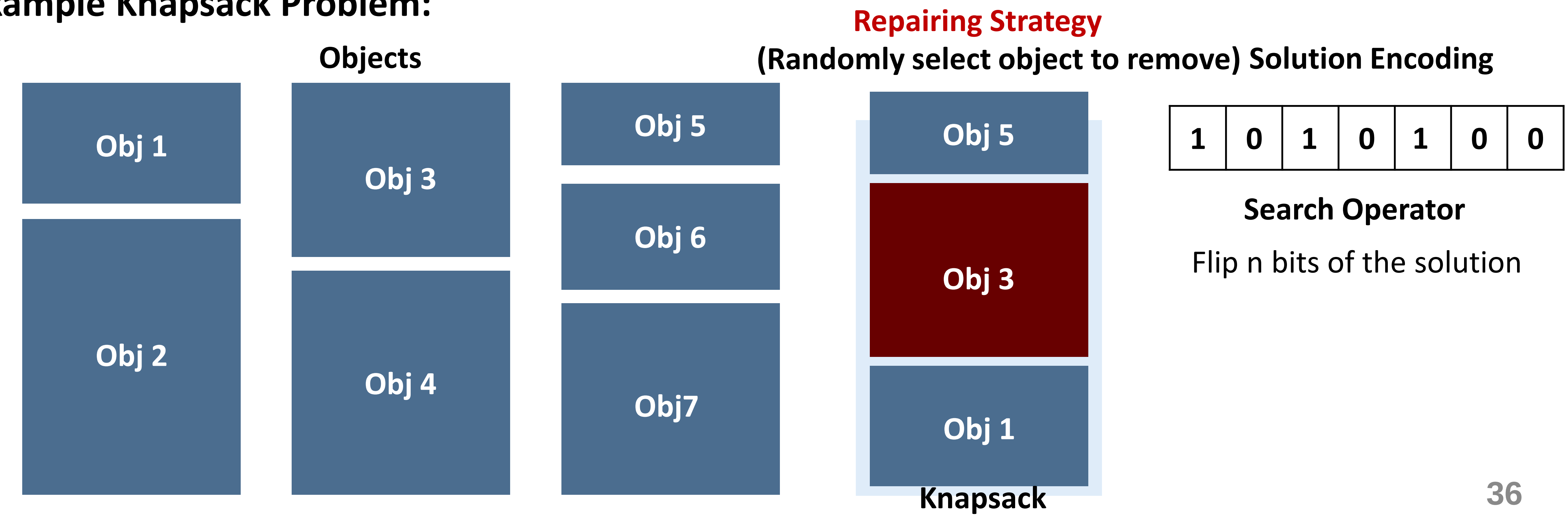
# Constraint Handling Techniques

- **Reject Strategies:** represent a simple approach, where **only feasible solutions are kept during the search** and then infeasible solutions are automatically discarded. This kind of strategies are conceivable if the portion of infeasible solutions of the search space is very small.
- **Repairing strategies:** A **repairing procedure** is applied to infeasible solutions to generate feasible ones (e.g. extracting from the knapsack some elements to satisfy the capacity constraint in the knapsack problem)
- **Penalizing Strategies:** reject strategies do not exploit infeasible solutions. Indeed, it would be interesting to use some information on infeasible solutions to guide the search. In penalizing strategies, infeasible solutions are considered during the search process. The unconstrained objective function is extended by a **penalty function that will penalize infeasible solutions**
- **Preserving Strategies:** In preserving strategies for constraint handling, a **specific representation and operators will ensure the generation of feasible solutions.** They incorporate problem-specific knowledge into the representation and search operators to generate only feasible solutions

# Repairing strategies

- Dealing with constraints in optimization problems is an important topic for the efficient design of metaheuristics.
- Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints.

**Example Knapsack Problem:**



# Constraint Handling Techniques

- **Reject Strategies:** represent a simple approach, where **only feasible solutions are kept during the search** and then infeasible solutions are automatically discarded. This kind of strategies are conceivable if the portion of infeasible solutions of the search space is very small.
- **Repairing strategies:** A **repairing procedure** is applied to infeasible solutions to generate feasible ones (e.g. extracting from the knapsack some elements to satisfy the capacity constraint in the knapsack problem)
- **Penalizing Strategies:** The reject strategies do not exploit infeasible solutions. Indeed, it would be interesting to use some information on infeasible solutions to guide the search. In penalizing strategies, infeasible solutions are considered during the search process. The unconstrained objective function is extended by a **penalty function that will penalize infeasible solutions**
- **Preserving Strategies:** In preserving strategies for constraint handling, a **specific representation and operators will ensure the generation of feasible solutions.** They incorporate problem-specific knowledge into the representation and search operators to generate only feasible solutions

# Penalizing Strategies

- The objective function *f* may be penalized in a linear manner, where *c(s)* represents the cost of the constraint violation and *λ* the weights given to infeasibilities.

$$f'(s) = f(s) + \lambda c(s)$$

- Different penalty functions may be use:
  - **Violated constraints**: A straightforward function is to **count the number of violated constraints**. No information is used on how close the solution is to the feasible region of the search space. (e.g. number of bins with capacity violated in the bin-packing problem)
  - **Amount of infeasibility**: Information on **how close a solution is to a feasible region** is taken into account (e.g. how much the capacity of a bin is exceeded in the bin-packing problem).

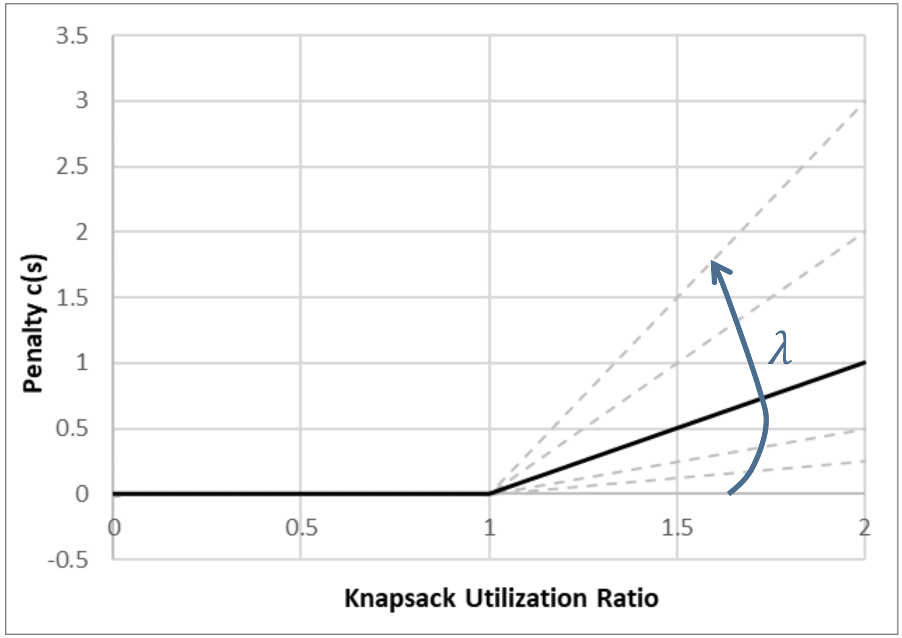
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

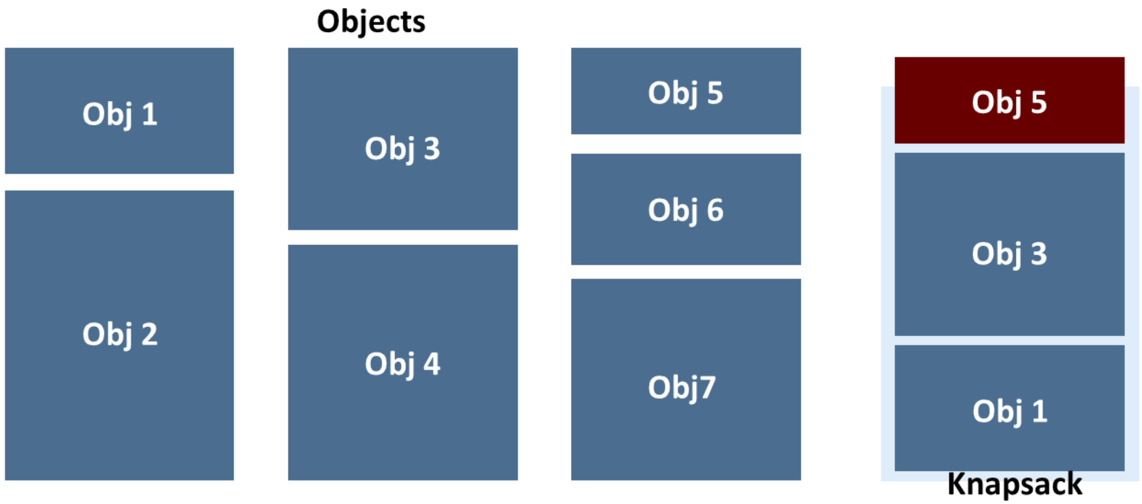
$$f'(s) = f(s) + \lambda c(s)$$

$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




**Example Knapsack Problem:**




5.5 kg

$$\lambda = 1$$

$$\lambda c(s) = -\frac{|(2 + 2.5 + 1 - 4)|}{4} = -0.375$$

$$f(s) = 5 + 3 + 4 = 12$$

$$f'(s) = \mathbf{11.625}$$

**Infeasible Solution**

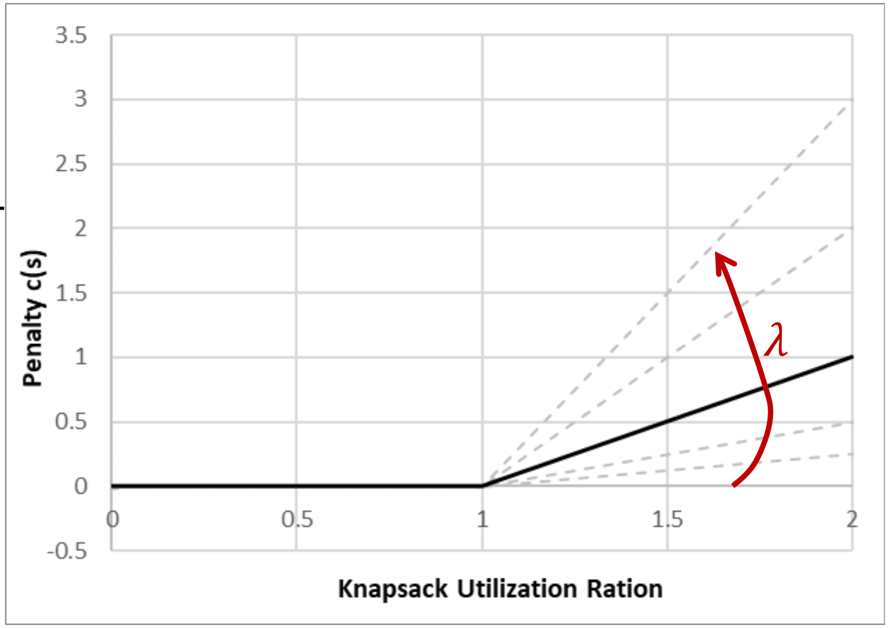
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

$$f'(s) = f(s) + \lambda c(s)$$

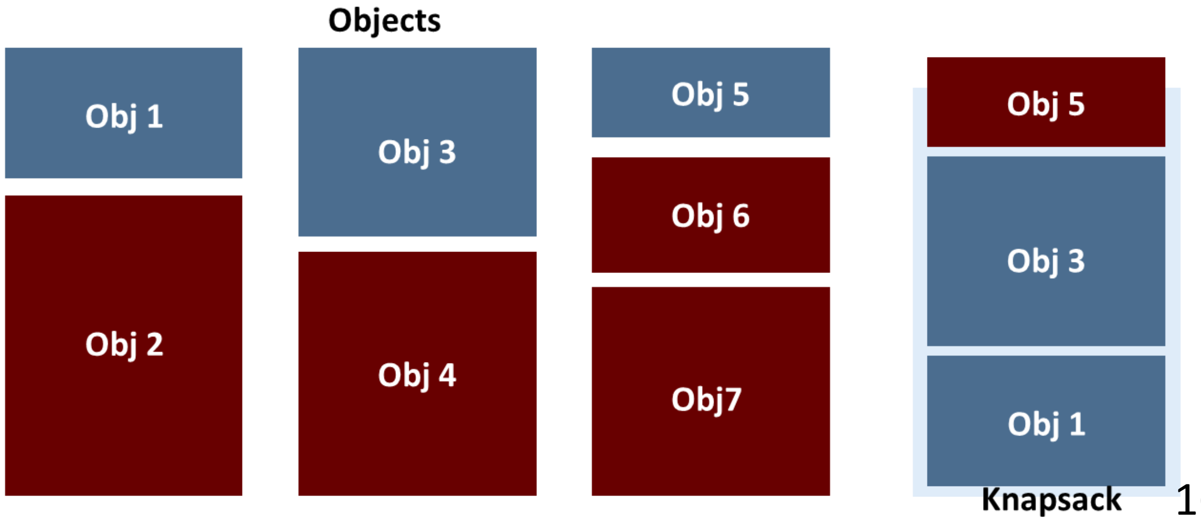
$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




$$\lambda = 1$$

**Example Knapsack Problem:**




16.5 kg

$$\lambda c(s) = -4.125$$

$$f(s) = 39$$

$$f'(s) = 34.875$$

**Infeasible Solution**

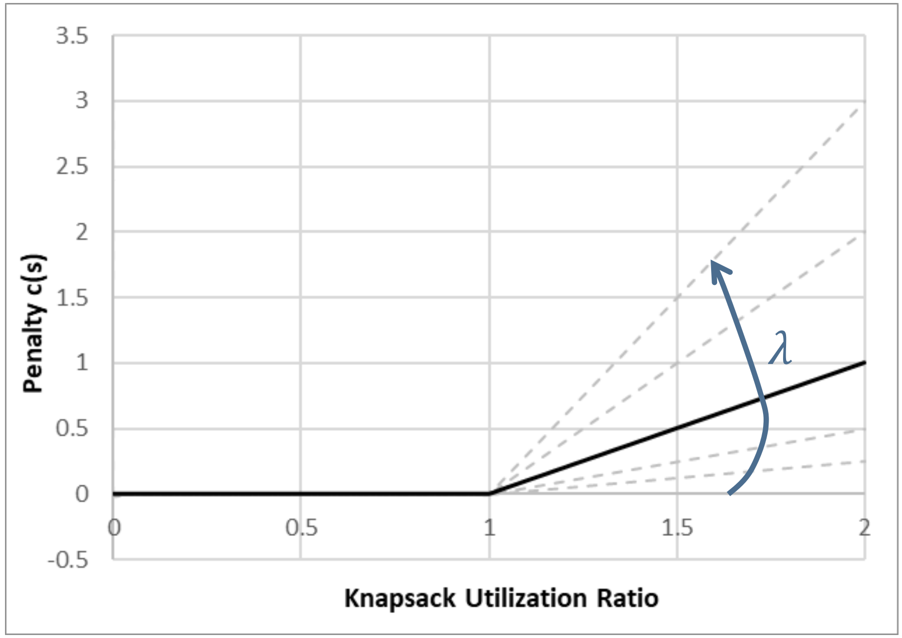
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

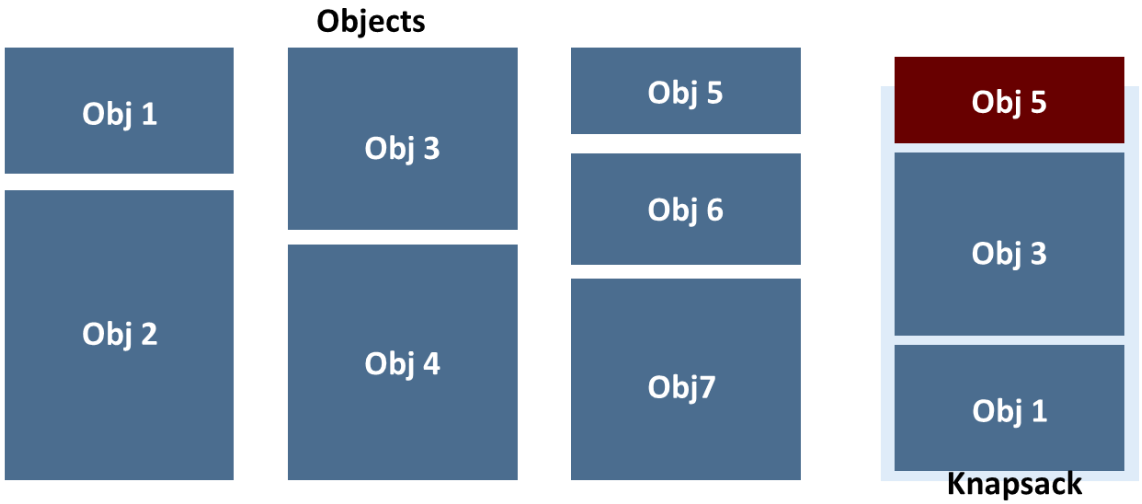
$$f'(s) = f(s) + \lambda c(s)$$

$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




**Example Knapsack Problem:**




5.5 kg

$$\lambda = 10$$

$$\lambda c(s) = -3.75$$

$$f(s) = 12$$

$$f'(s) = \mathbf{8.25}$$

**Infeasible Solution**

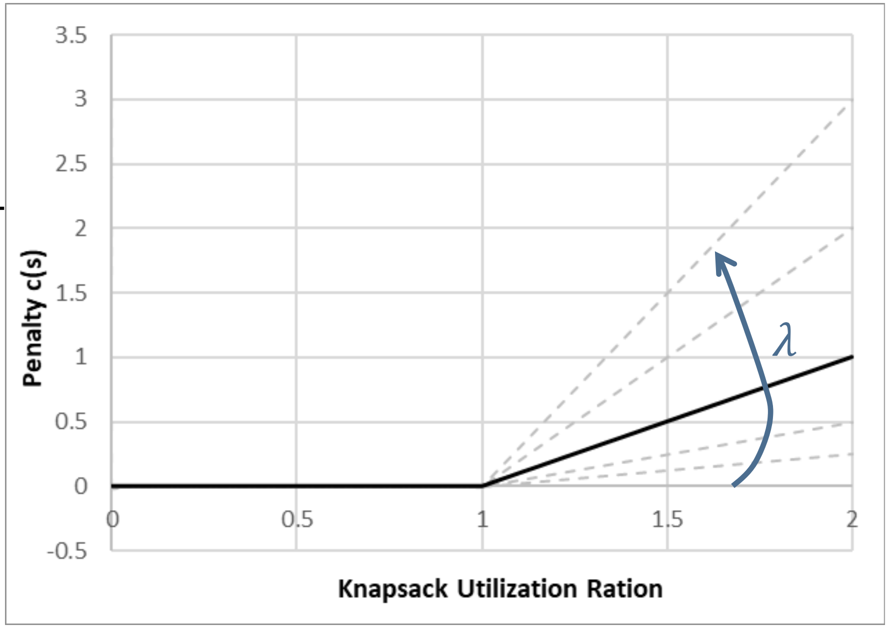
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

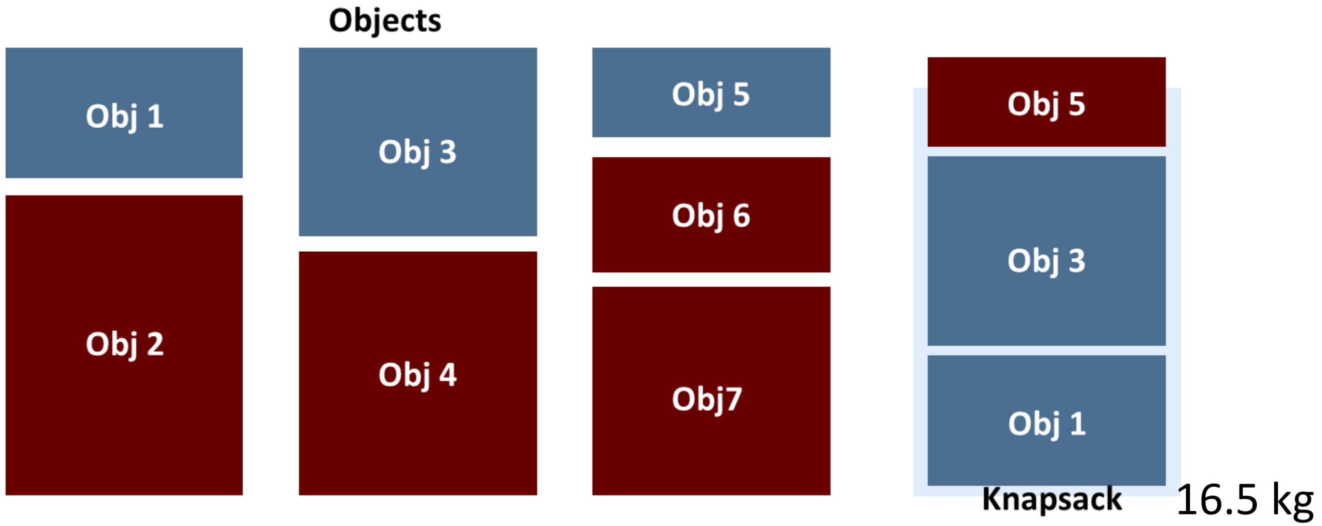
$$f'(s) = f(s) + \lambda c(s)$$

$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




**Example Knapsack Problem:**




$$\lambda = 10$$

$$\lambda c(s) = -41.25$$

$$f(s) = 39$$

$$f'(s) = \mathbf{-2.25}$$

**Infeasible Solution**

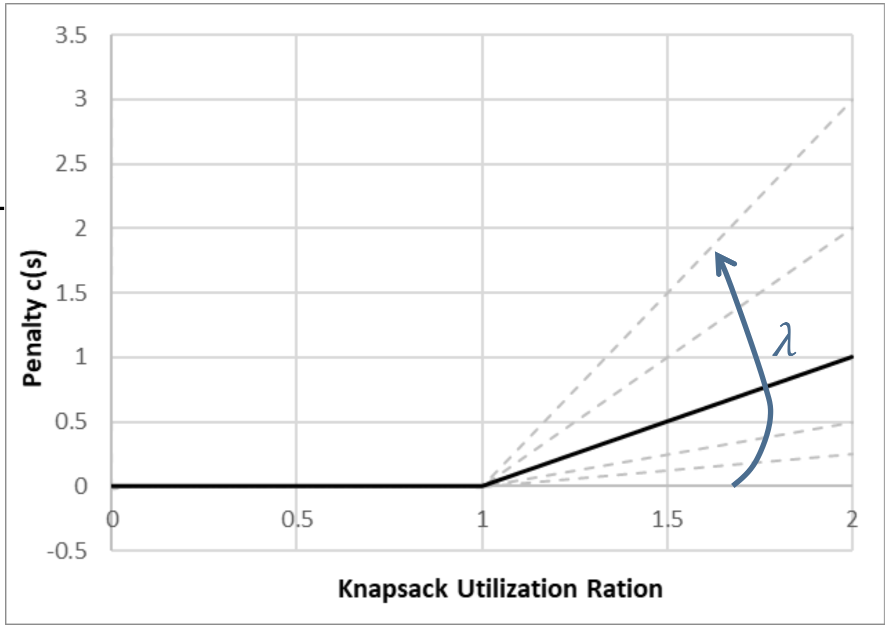
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

$$f'(s) = f(s) + \lambda c(s)$$

$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




$$\lambda = 10$$

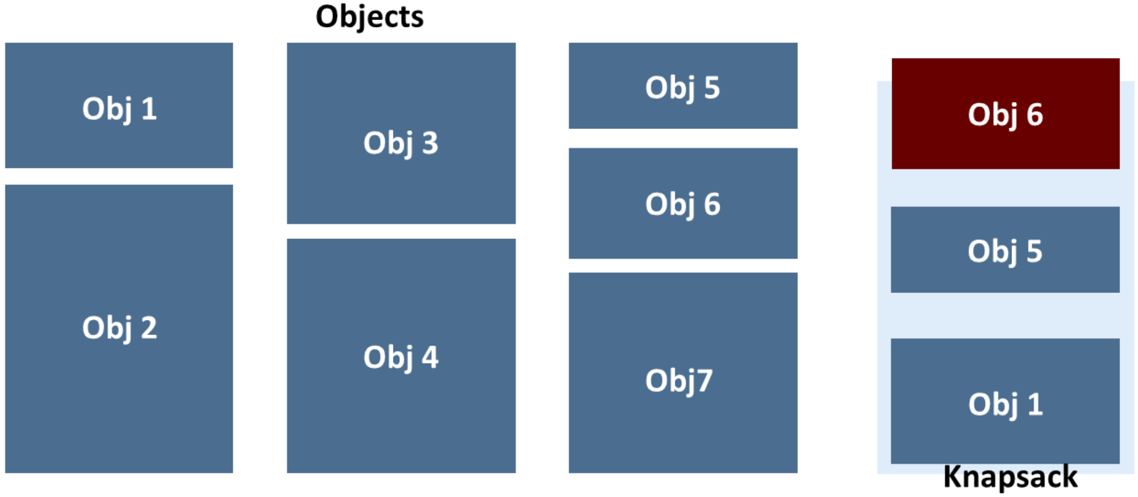
$$\lambda c(s) = -2.813$$

$$f(s) = 17$$

$$f'(s) = \mathbf{14.187}$$

**Infeasible Solution**

**Example Knapsack Problem:**




4.5 kg

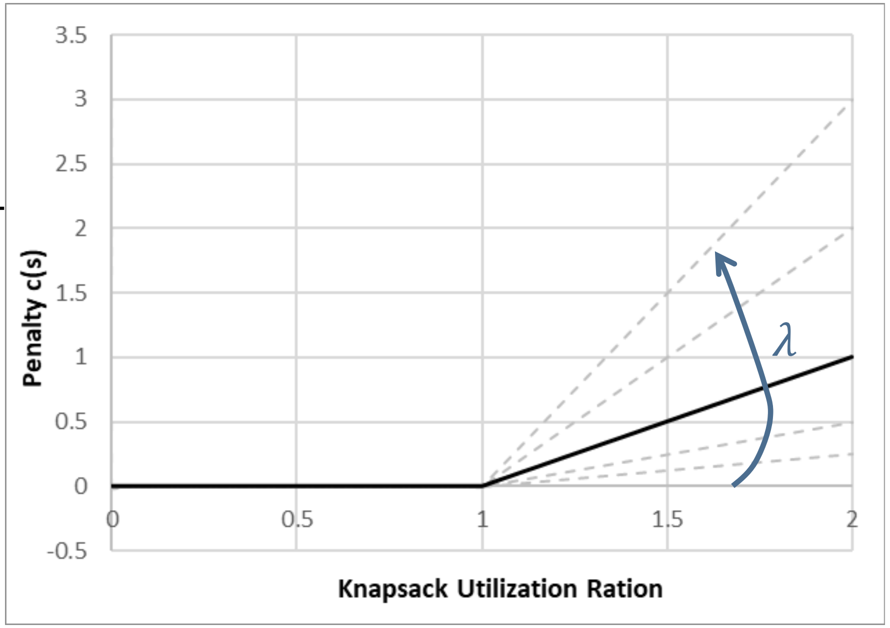
# Penalizing Strategies

| | $w_i$ | $f_i$ |
|---|---|---|
| Obj 1 | 2 | 5 |
| Obj 2 | 3.75 | 7 |
| Obj 3 | 2.5 | 3 |
| Obj 4 | 3 | 5 |
| Obj 5 | 1 | 4 |
| Obj 6 | 1.5 | 8 |
| Obj 7 | 2.75 | 7 |
| cap | 4 | |

$$f'(s) = f(s) + \lambda c(s)$$

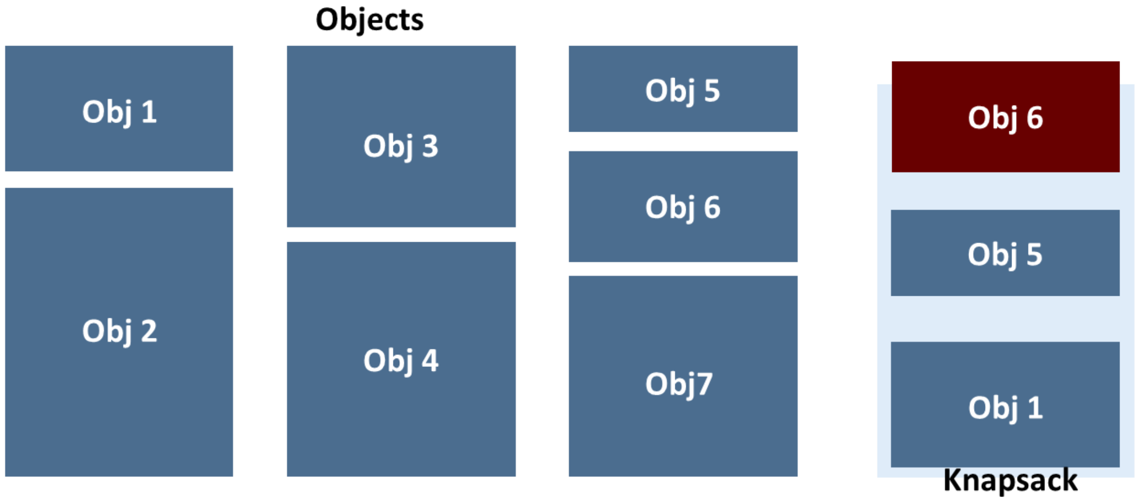
$$c(s) = -\frac{min(0, \sum_i w_i - Cap)}{Cap}$$

$$f(s) = \sum_i f_i$$




$$\lambda = 10$$

**Example Knapsack Problem:**




4.5 kg

$$\lambda c(s) = 0$$

$$f(s) = 5 + 1 + 8 = 13$$

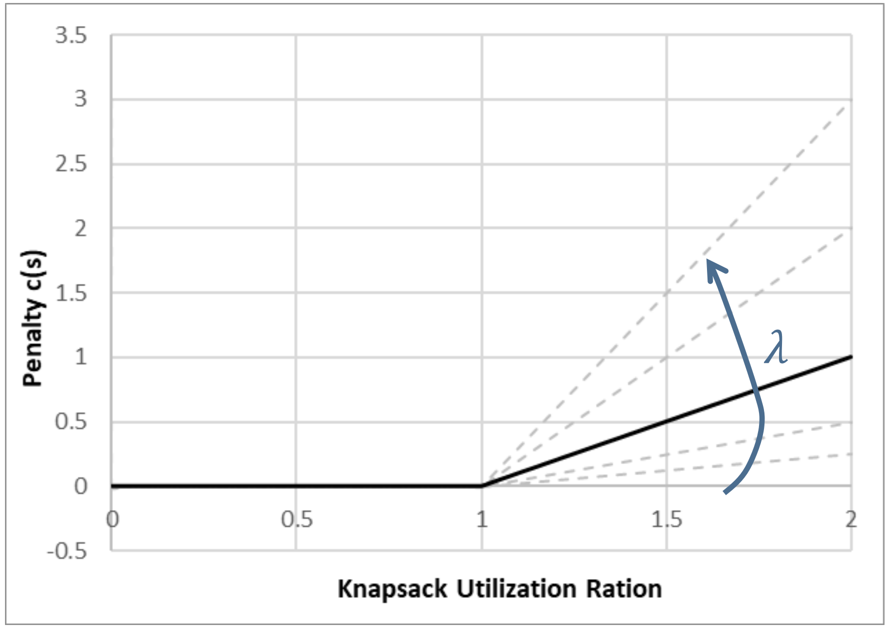
$$f'(s) = 13$$

Feasible solution, but worse objective value

# Adaptative Penalization

- The previously presented penalty functions do not exploit any information of the search process. In adaptive penalty functions, **knowledge on the search process is included to improve the efficiency and the effectiveness of the search.**

- Example:
  - The parameters $\lambda$ is self-adjusting. Initially, the parameter is initialized to 1.
  - The parameters $\lambda$ is reduced (resp. increased) if the last $\mu$ visited solutions are all feasible (resp. all infeasible), where μ is a user-defined parameter. The reduction (resp. increase) may consist in dividing (resp. multiplying) the actual value by 2, for example.

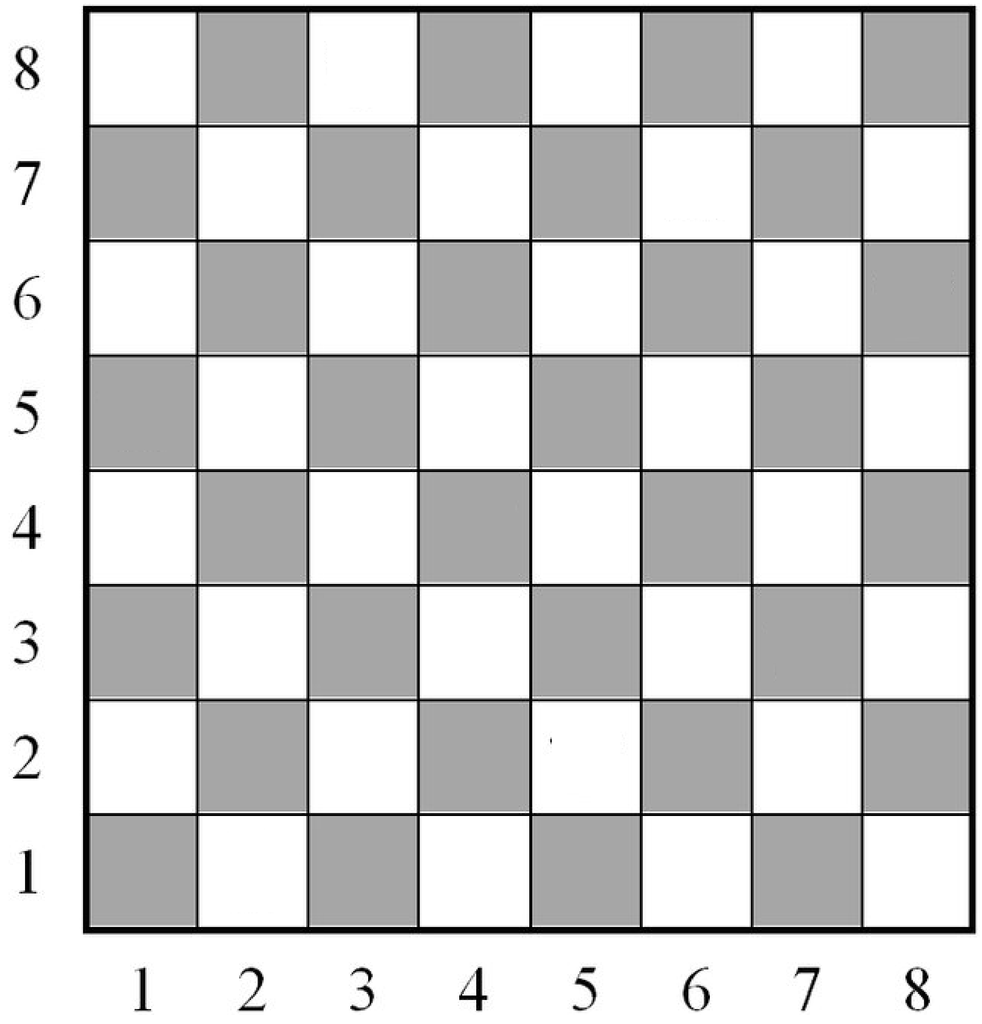


# Constraint Handling Techniques

- **Reject Strategies:** represent a simple approach, where **only feasible solutions are kept during the search** and then infeasible solutions are automatically discarded. This kind of strategies are conceivable if the portion of infeasible solutions of the search space is very small.
- **Repairing strategies:** A **repairing procedure** is applied to infeasible solutions to generate feasible ones (e.g. extracting from the knapsack some elements to satisfy the capacity constraint in the knapsack problem)
- **Penalizing Strategies:** The reject strategies do not exploit infeasible solutions. Indeed, it would be interesting to use some information on infeasible solutions to guide the search. In penalizing strategies, infeasible solutions are considered during the search process. The unconstrained objective function is extended by a **penalty function that will penalize infeasible solutions**
- **Preserving Strategies:** In preserving strategies for constraint handling, a **specific representation and operators will ensure the generation of feasible solutions.** They incorporate problem-specific knowledge into the representation and search operators to generate only feasible solutions

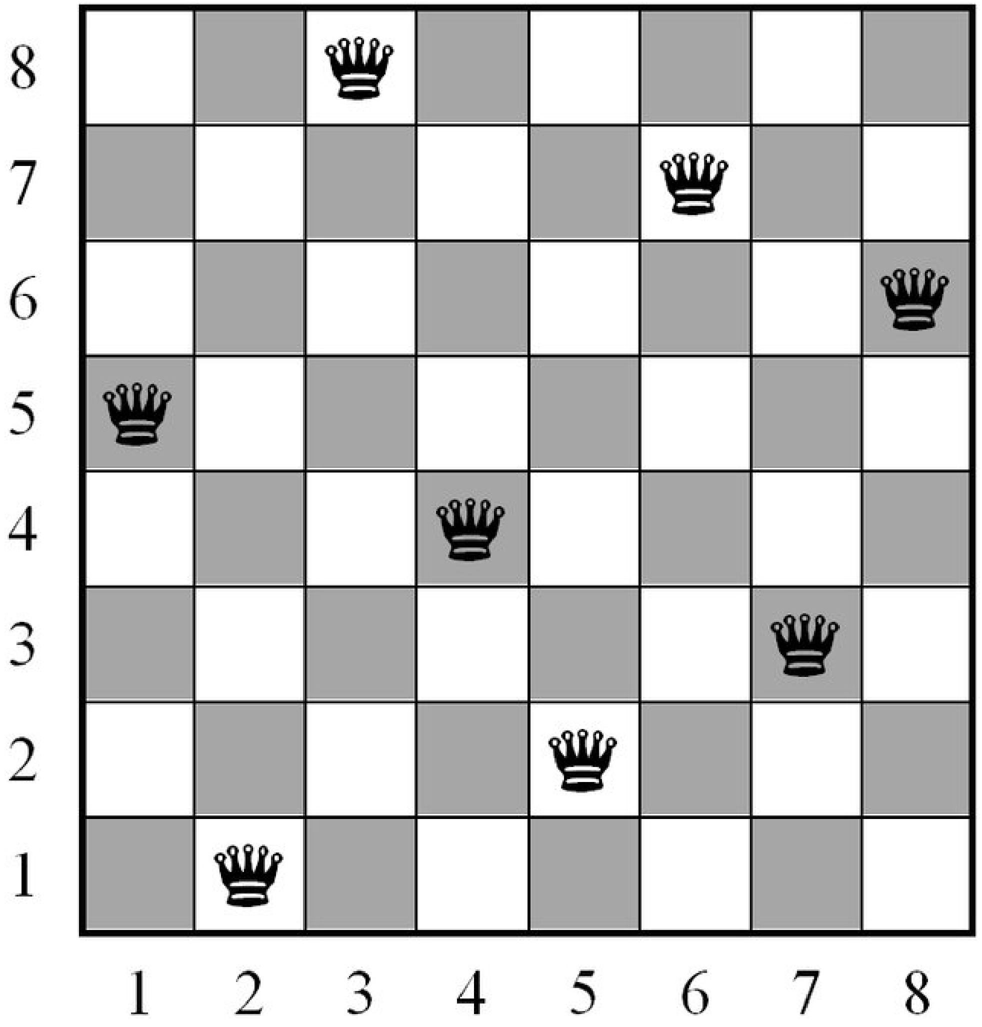
# Preserving Strategies

- Constructing a solution encoding/search operator that always guarantee that feasible solutions are obtained, is often not possible.
- Therefore, preserving strategies are not an alternative in many cases
- Yet, before applying other constraint handling techniques, one must think if it is possible to design a solution encoding/search operator that always provide feasible solutions
- Let’s take a look on a classic example: the N-Queens Puzzle



# Preserving Strategies

- The N-Queens Puzzle problem consists on putting N chess queens on an NxN chessboard such that none of queens is able to capture any other.
- By exhaustive search, the number of possibilities is $64^8$, that is over 4 billion solutions (size of the search space)
- If we prohibit more than one queen per row, then the search space will have $8^8$ solutions, that is over 16 millions.
- If we forbid two queens to be both in the same column or row, the encoding will be reduced to $n!$, that is 40,320



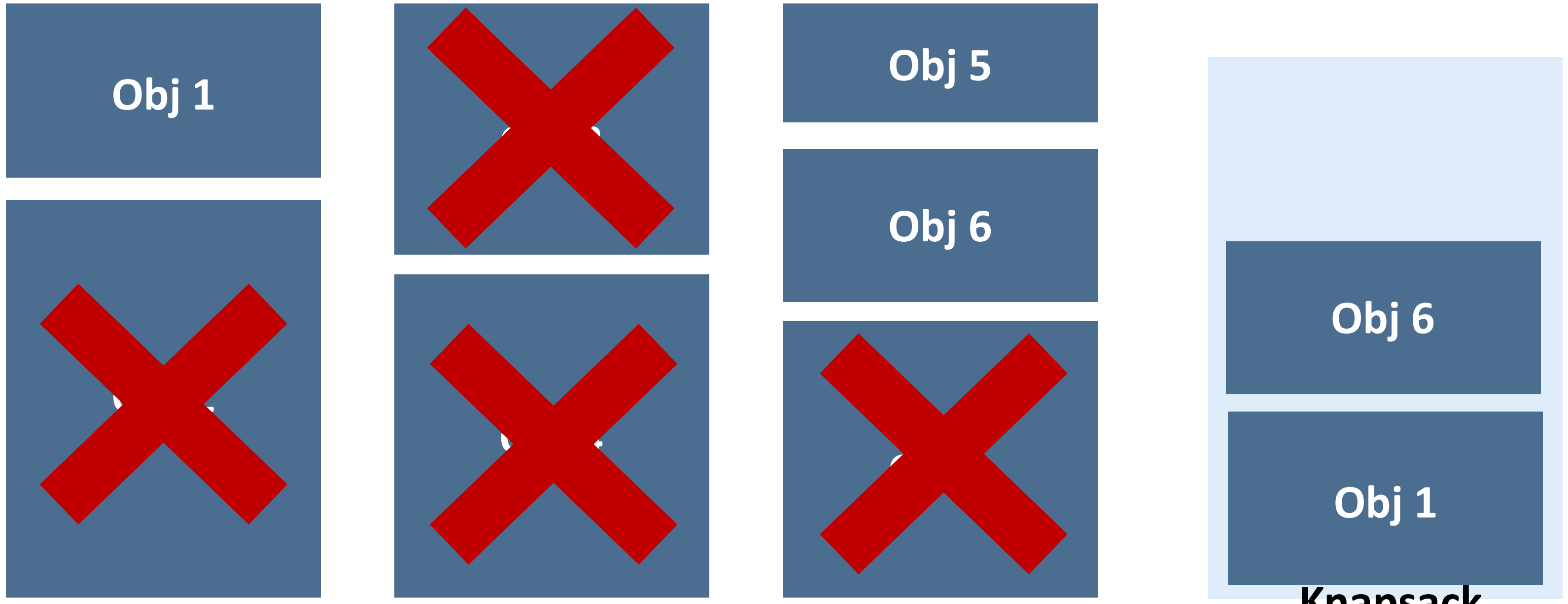
# Preserving Strategies

- Dealing with constraints in optimization problems is an important topic for the efficient design of metaheuristics.
- Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints.

**Example Knapsack Problem:**

**Rejection Strategy**

**Objects**




**Knapsack**

**Solution Encoding**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |

**Search Operator**

Flip n bits of the solution

# Solution Encoding and Search Operator

- A solution encoding must have the following characteristics:
  - **Completeness**: all solutions associated with the problem must be represented.
  - **Connexity**: A search path must exist between any two solutions of the search space. Any solution of the search space, especially the global optimum solution, can be attained.
  - **Efficiency**: The representation must be easy to manipulate by the search operators. The time and space complexities of the operators dealing with the representation must be reduced.

**Solution encoding is key in metaheuristics performance – but also very problem specific!**

# Direct and Indirect Encoding

Nuno Antunes Ribeiro

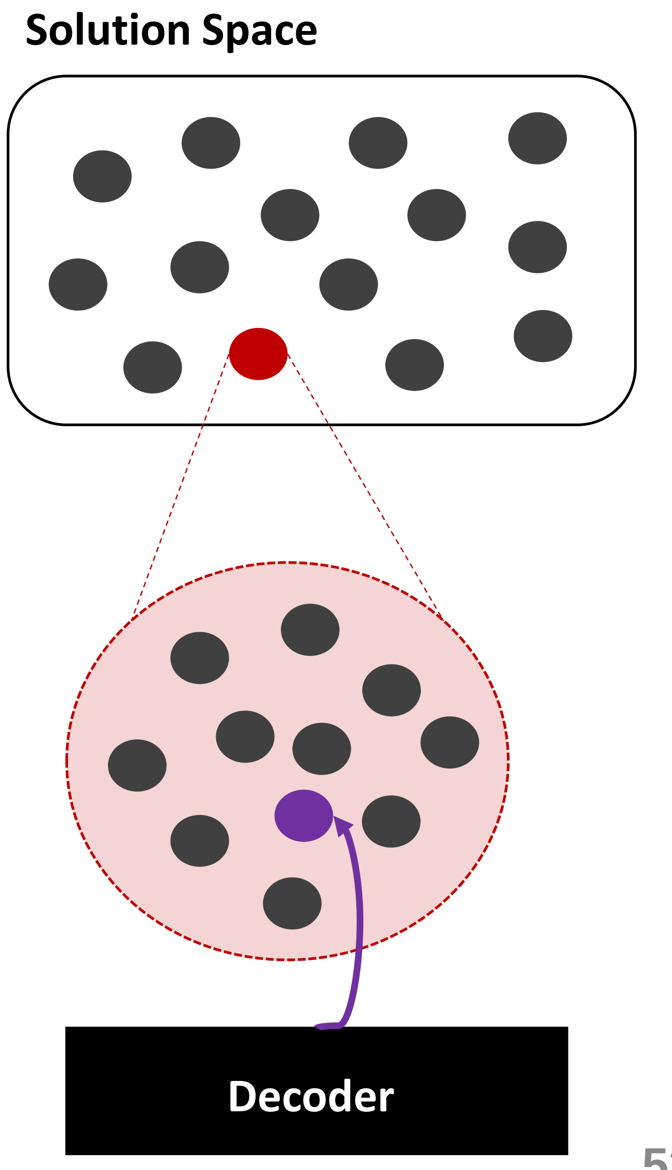
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

# Direct vs Indirect Encoding

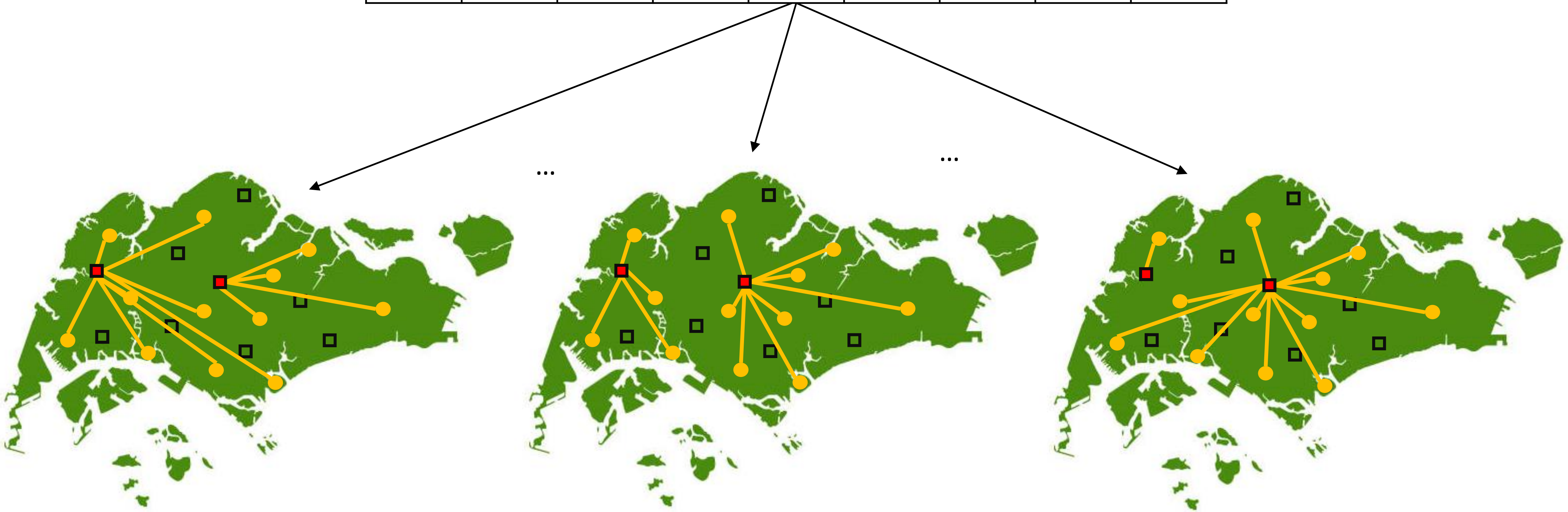
- **Indirect encoding** is characterized by a lack of details in the solution representation - i.e. some information on the solution is not explicitly represented.
- A **decoder** is required to express the solution given by the encoding. According to the information that is present in the indirect encoding, the decoder has more or less work to derive a complete solution
- Indirect encoding aims to **reduce the size of the original search space**. They are particularly popular in optimization problems **dealing with many constraints** such as scheduling problems.



# P-median Problem

Solution Encoding

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** |

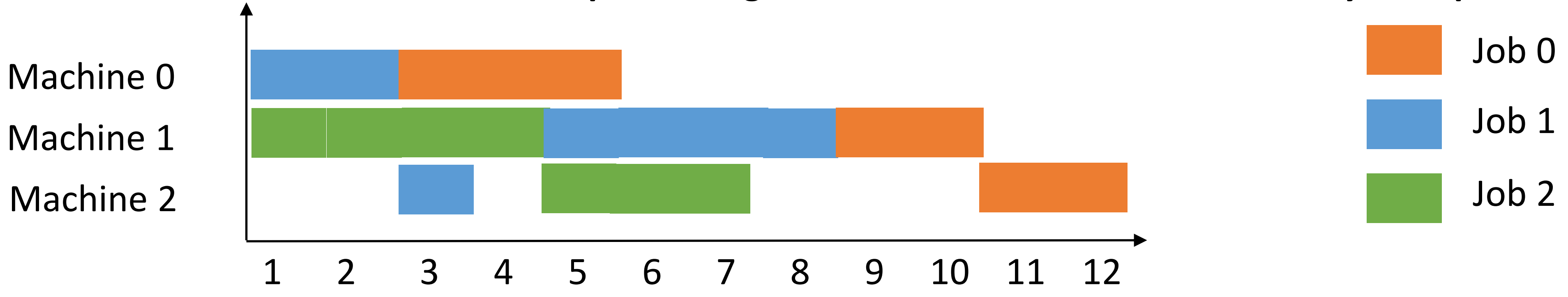


# Job-Shop Scheduling Problem (JSSP)

- In the classical Job-Shop Problem there are *n* jobs that must be processed on *m* machines. Each job consists of a sequence of different tasks. Each task needs to be processed during an uninterrupted period of time on a given machine.
- Here is an example with m=3 machines and n=3 jobs. We count jobs, machines and tasks starting from 0.

| Job | (Machine,Duration) | (Machine,Duration) | (Machine,Duration) |
|---|---|---|---|
| 0 | (0,3) | (1,2) | (2,2) |
| 1 | (0,2) | (2,1) | (1,4) |
| 2 | (1,4) | (2,3) | |

**Gant Chart representing a feasible solution - not necesseraly the optimal**



# JSSP Solution Representation

- **Direct** Encoding:
  - List of starting times:

| Job | (Machine, Duration) | (Machine, Duration) | (Machine, Duration) |
|---|---|---|---|
| **0** | (0,3) | (1,2) | (2,2) |
| **1** | (0,2) | (2,1) | (1,4) |
| **2** | (1,4) | (2,3) | |

| Job | Task | Task | Task |
|---|---|---|---|
| **0** | 1 | 2 | 3 |
| **1** | 4 | 5 | 6 |
| **2** | 7 | 8 | |

| Task 1 | Task 2 | Task 3 | Task 4 | Task 5 | Task 6 | Task 7 | Task 8 | Task 9 |
|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 6 | 2 | 3 | 8 | 3 | 7 | 10 |

Very ineffective encoding ; Mostly infeasible solutions will be generated

# JSSP Solution Representation

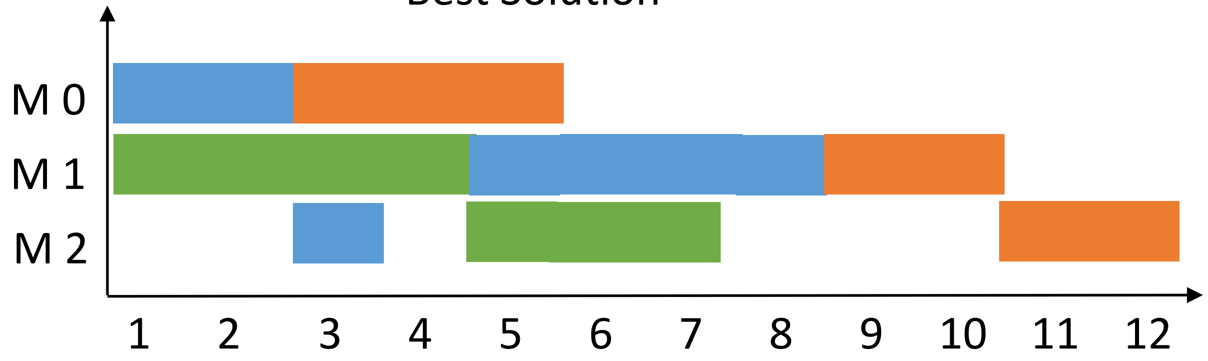
- **Indirect** Encoding:
  - Job Sequence Matrix:

| Machine 0 | | Machine 1 | | | Machine 2 | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 0 | 1 | 2 | 0 |

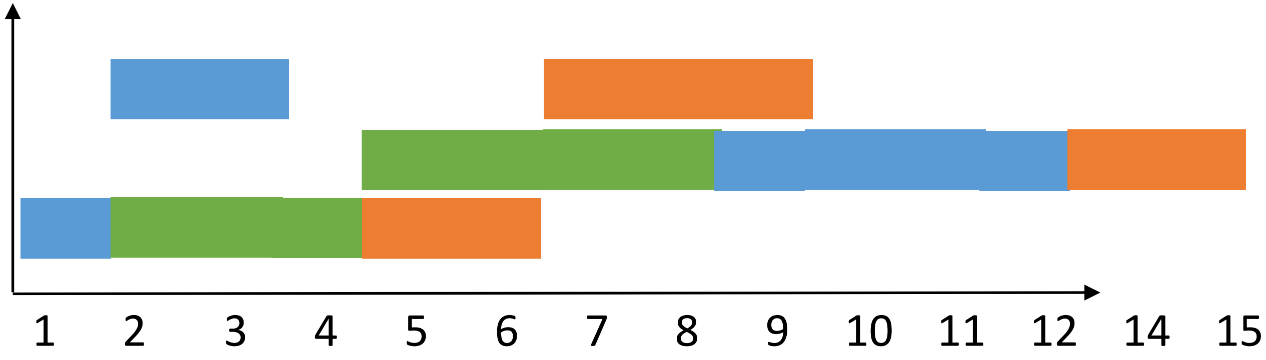
**In indirect encoding several solutions are represented by the same encoding. Some information on the solution is not explicitly represented. This will reduce the size of the original search space.**


Several solutions are represented by this encoding

Best Solution



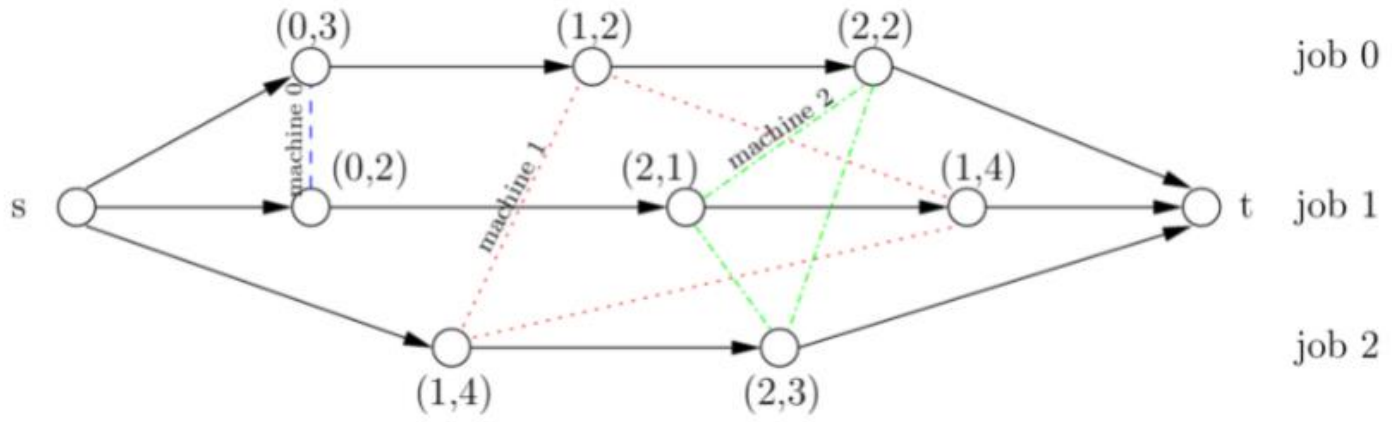

Another solution that can be obtained using the same job sequence



# JSSP Solution Representation

- **How to obtain the best solution given a job sequence matrix?**
  - A Gant chart can be represented as a disjunctive graph G=(V,C,D).
    - V is the set of vertices corresponding to the tasks
    - C is a set of conjunctive arcs between tasks of a job
    - D is a set of disjunctive arcs between tasks to be processed on the same machine.
  - A **topological ordering algorithm** can be used to determine the optimal critical path of a gant chart (represented as a disjunctive graph)

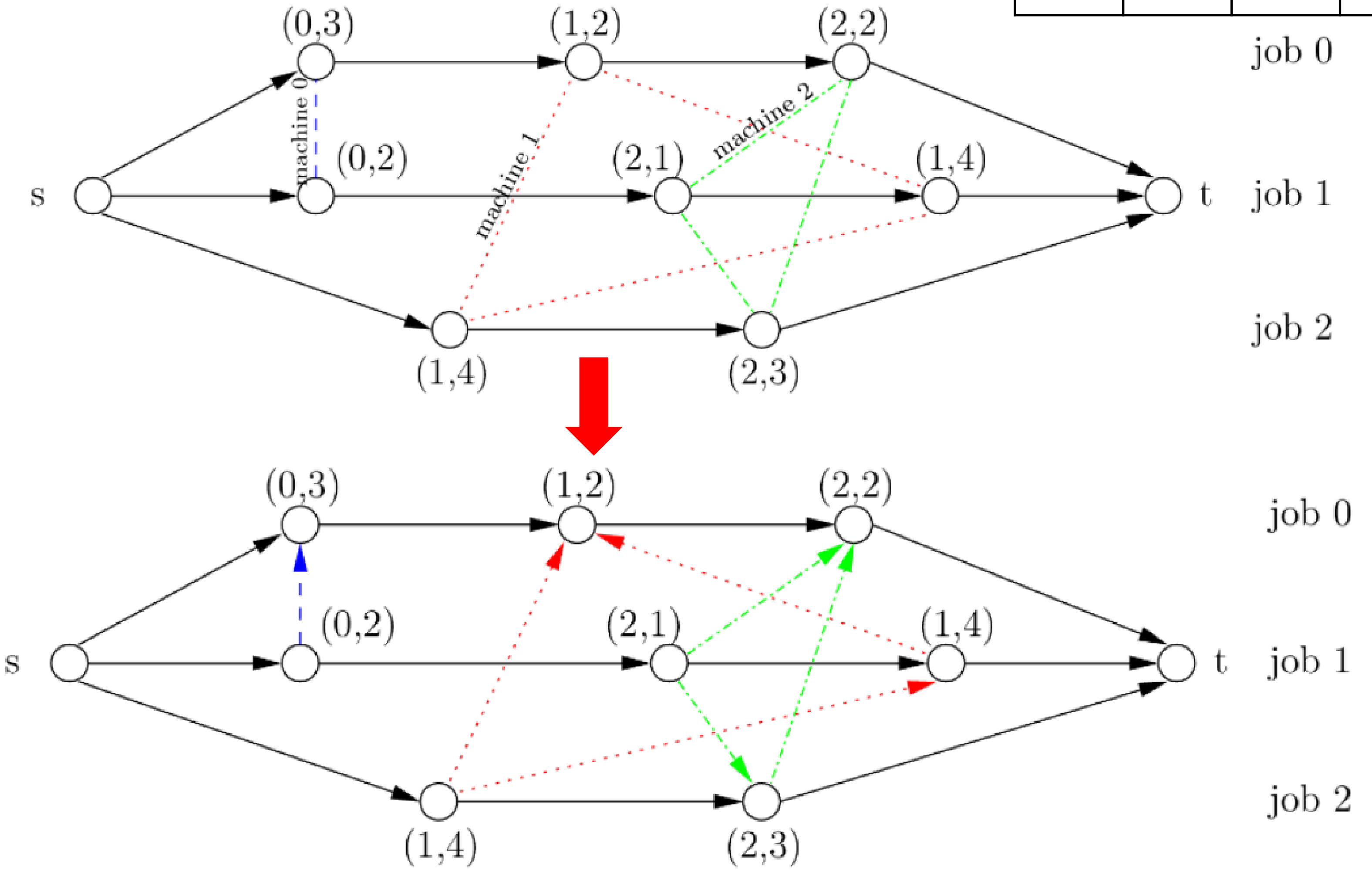
| Job | (Machine, Duration) | (Machine, Duration) | (Machine, Duration) |
|---|---|---|---|
| 0 | (0,3) | (1,2) | (2,2) |
| 1 | (0,2) | (2,1) | (1,4) |
| 2 | (1,4) | (2,3) | |



# JSSP Solution Representation

- Disjunctive graph solution representation

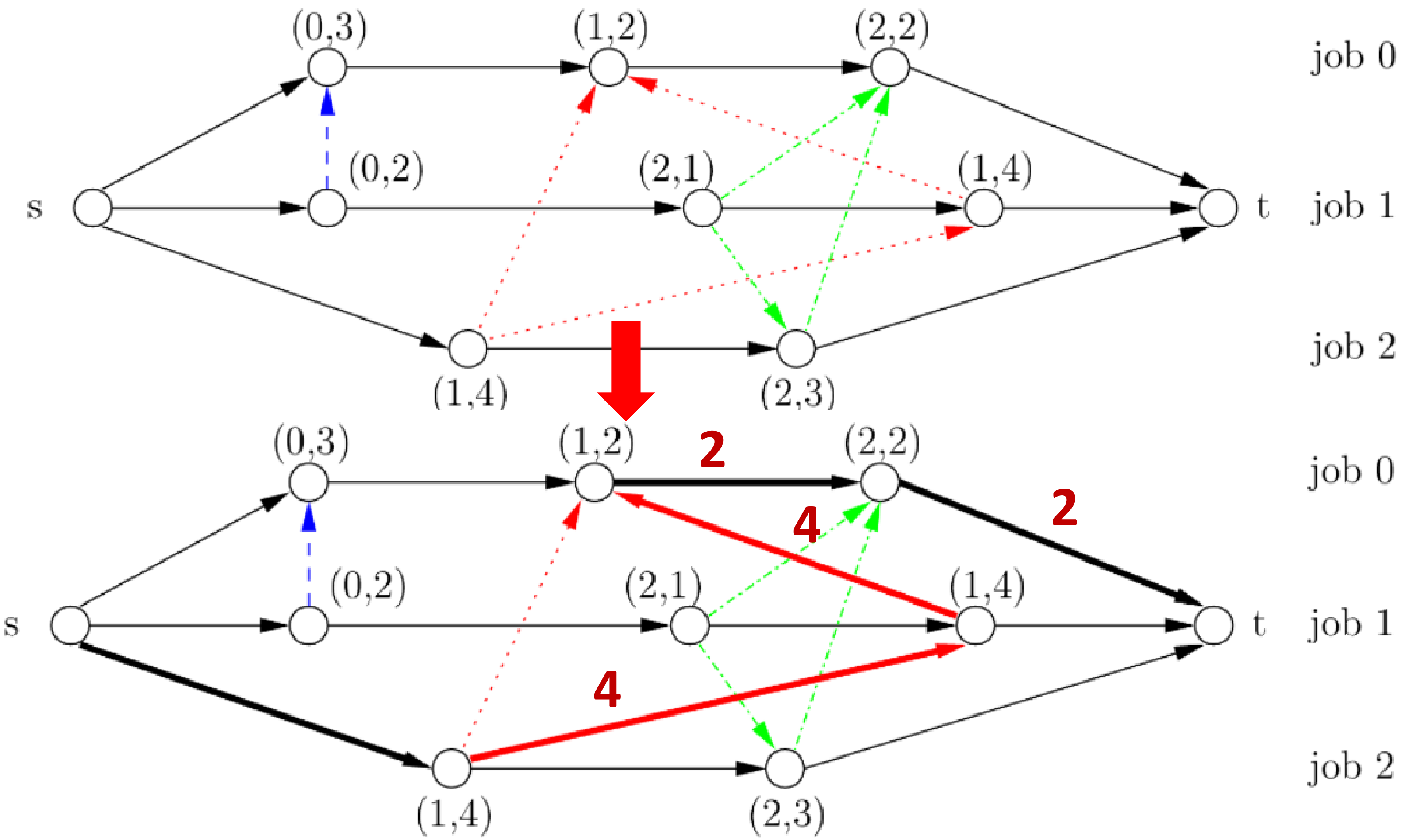
| Machine 0 | | Machine 1 | | | Machine 2 | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 0 | 1 | 2 | 0 |



# JSSP Solution Representation

- Optimal critical path is given by the longest weighted path from *s* to *t*

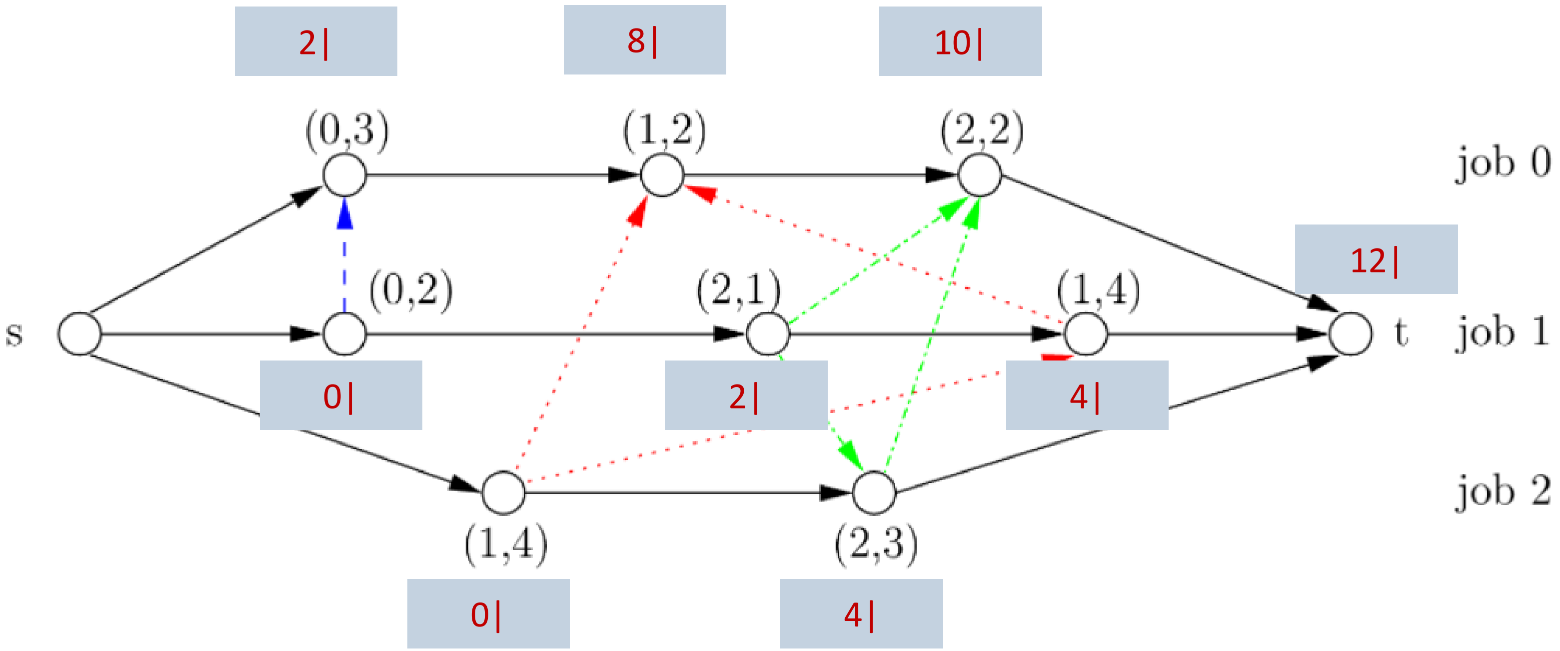
| Machine 0 | | Machine 1 | | | Machine 2 | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 0 | 1 | 2 | 0 |




**Critical Path Length =4+4+2+2=12**

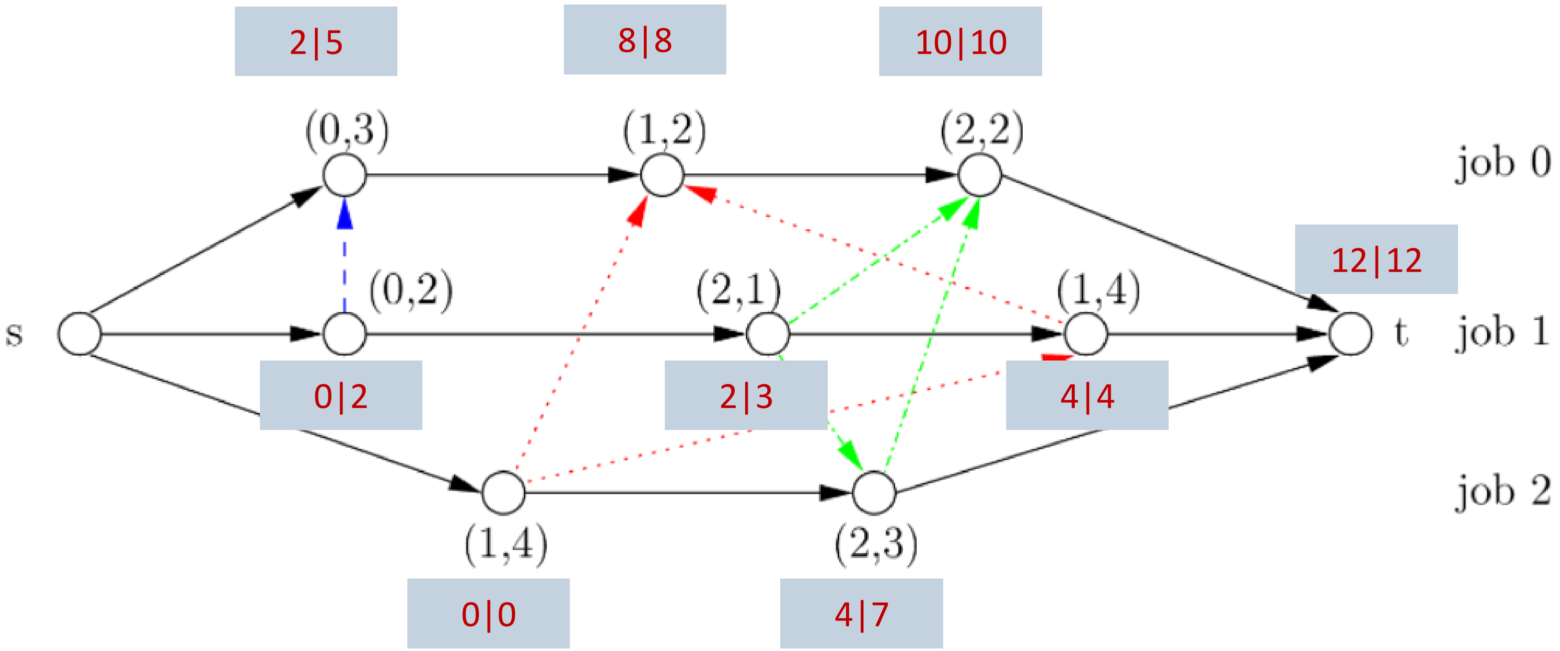
# JSSP Solution Representation

- How to obtain the longest weighted path of a disjunctive graph?
- We can use a topological ordering algorithm



# JSSP Solution Representation

- How to obtain the longest weighted path of a disjunctive graph?
- We can use a topological ordering algorithm

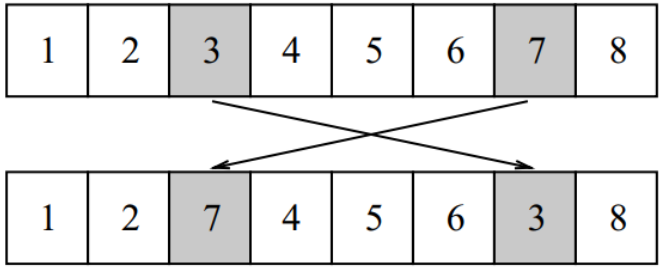


# JSSP Move Operators

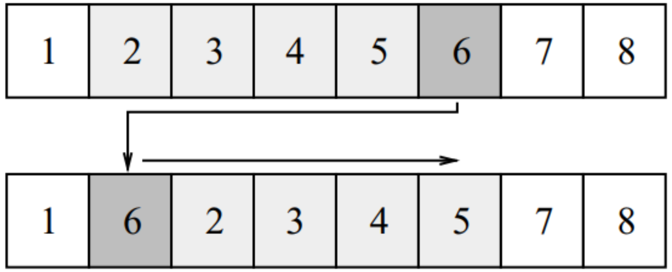
- **Permutation encoding**

| Machine 0 | | Machine 1 | | | Machine 2 | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 0 | 1 | 2 | 0 |

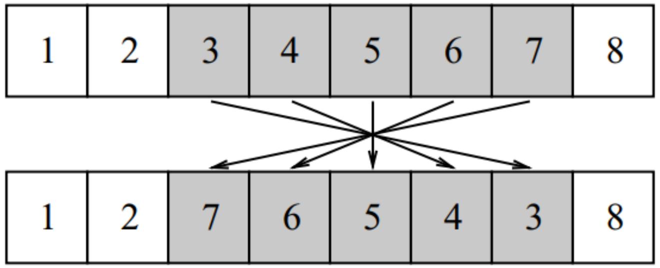
  - Swap/Exchange Operator




  - Insertion Operator

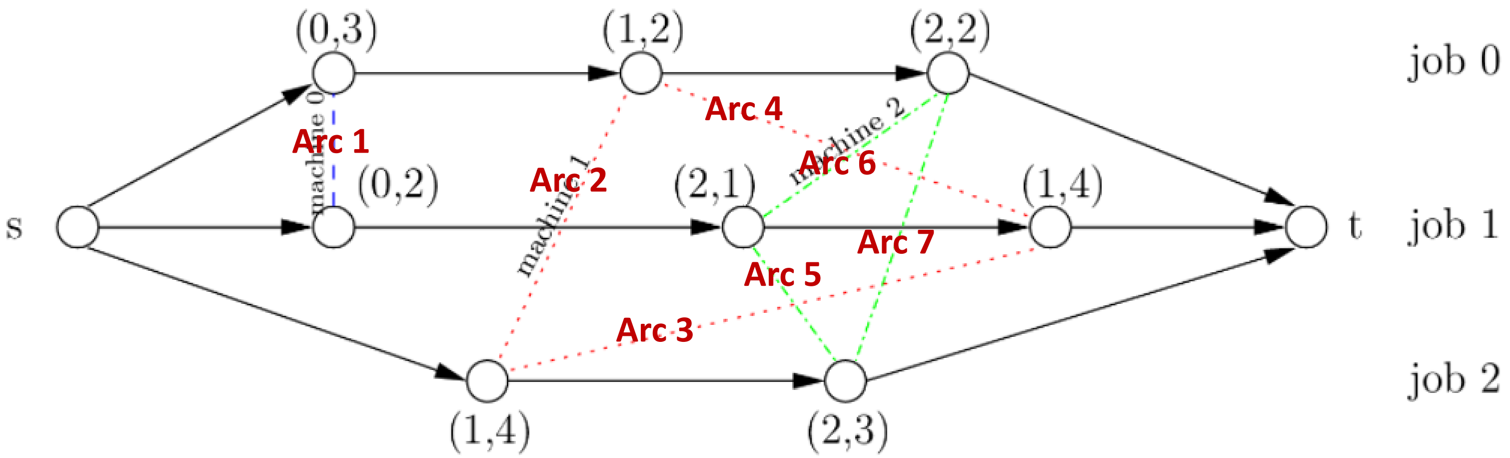



  - Inversion Operator



# JSSP Move Operators

- **Binary encoding** – each bit represents the orientation of a disjunction arc in the disjunction graph




| Arc 1 | Arc 2 | Arc 3 | Arc 4 | Arc 5 | Arc 6 | Arc 7 |
|---|---|---|---|---|---|---|
| 1 | 0 | **0** | 1 | 0 | 1 | 1 |

⬇

| Arc 1 | Arc 2 | Arc 3 | Arc 4 | Arc 5 | Arc 6 | Arc 7 |
|---|---|---|---|---|---|---|
| 1 | 0 | **1** | 1 | 0 | 1 | 1 |

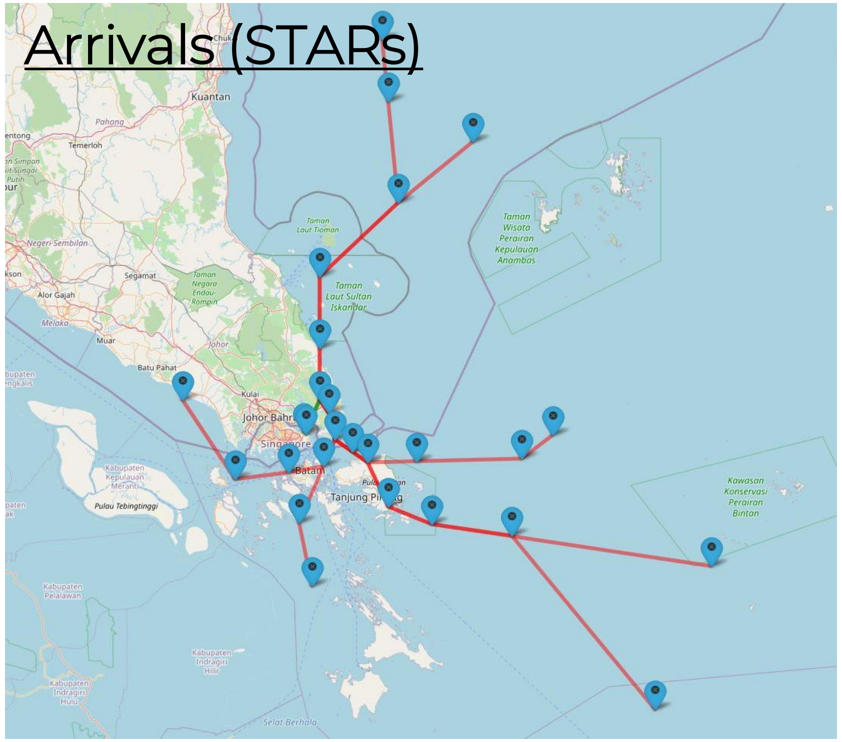
# Optimizing Terminal Maneouvering Airspace Operations

- TMA is designated area of controlled airspace surrounding a major airport where there is a high volume of traffic.
- It is a critical region - where all arriving aircraft from different entry points are merged and sequenced into an orderly stream towards the airport.
- Departing flights also use the TMA region, therefore air traffic controllers must ensure that arrivals and departures do not conflict with each other


**Radarbox website:** https://www.radarbox.com/

# SIDs and STARs

- Aircraft flying in the TMA must follow pre-defined routes, designated as Standard Terminal Arrival Routes (STAR) and Standard Instrument Departure (SID) routes.

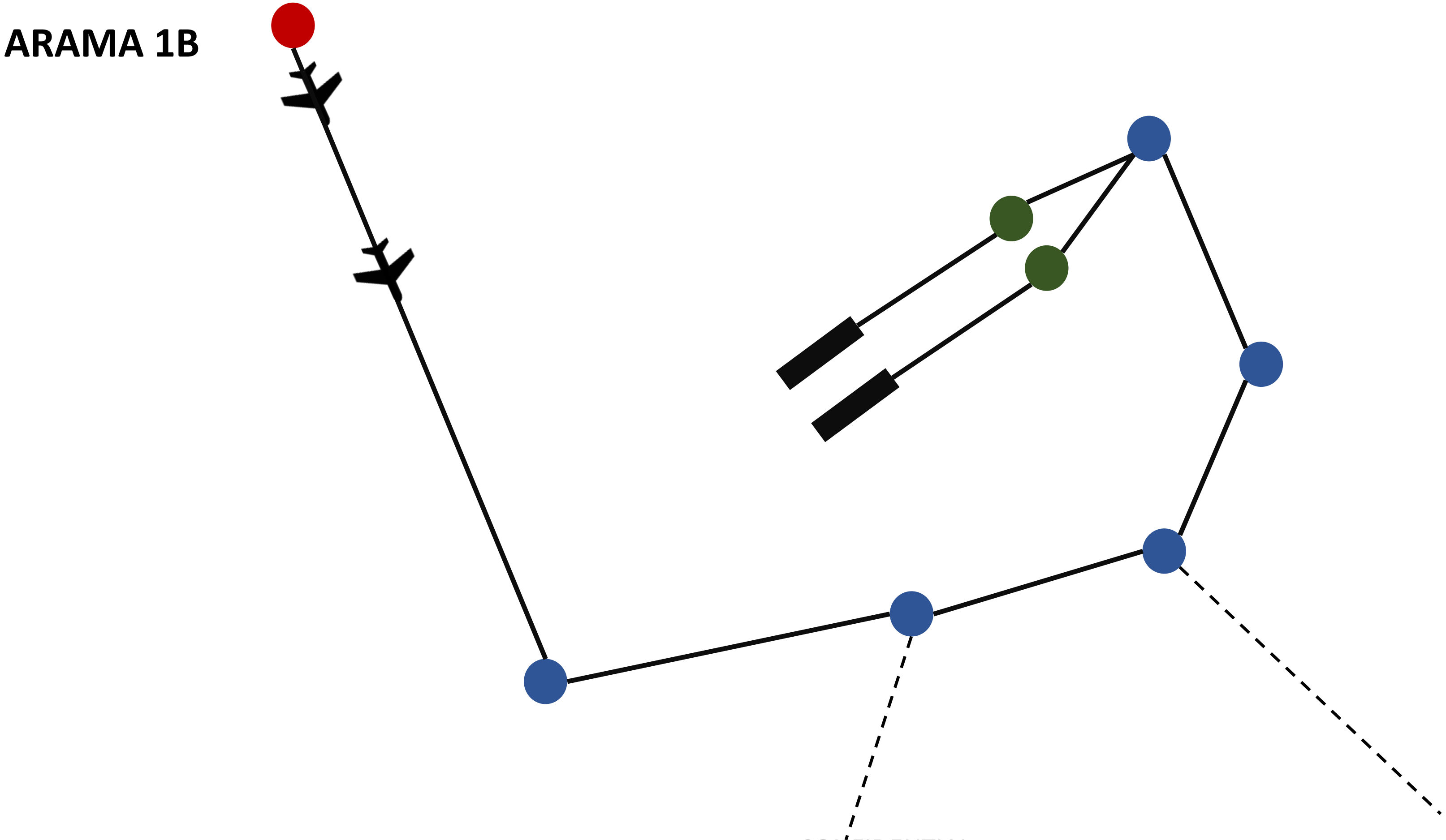




STAR - Standard Terminal Arrival Route
SID – Standard Instrument Departure

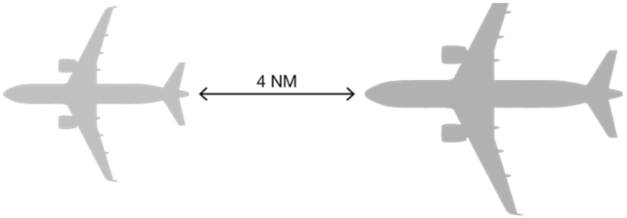
# TMA Operations

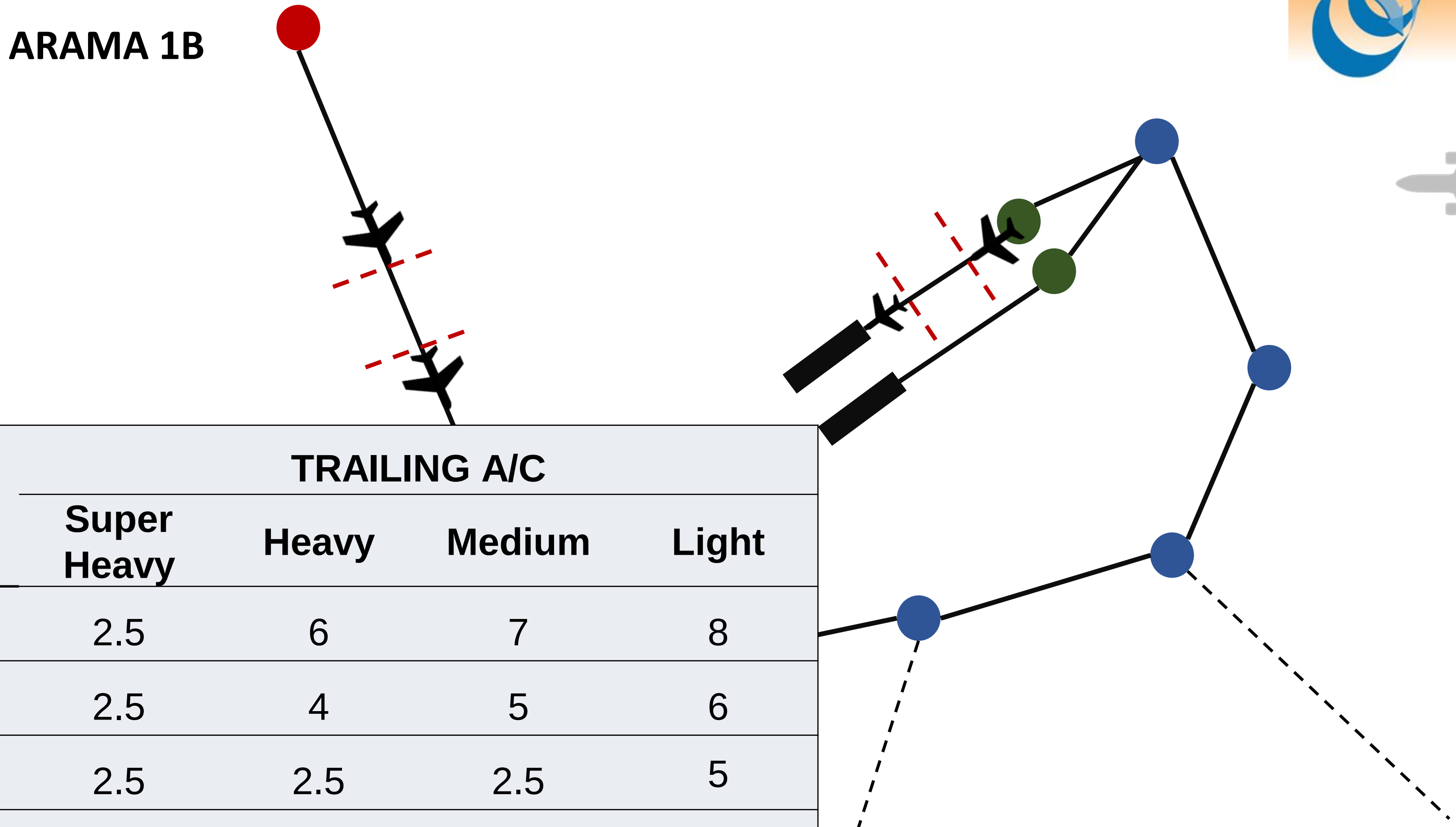
- Separation Requirements



# TMA Operations

- Optimization-based approach

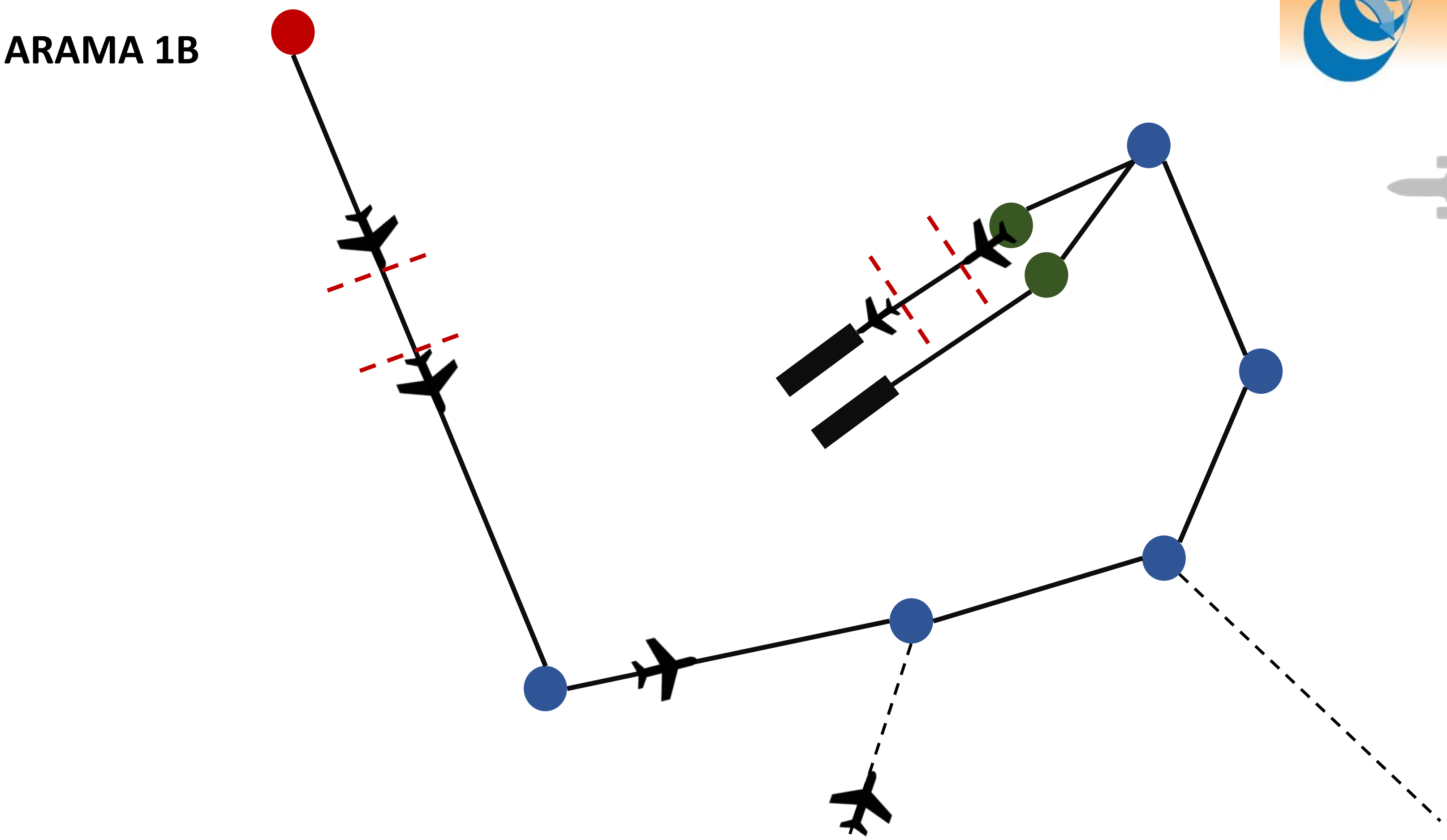


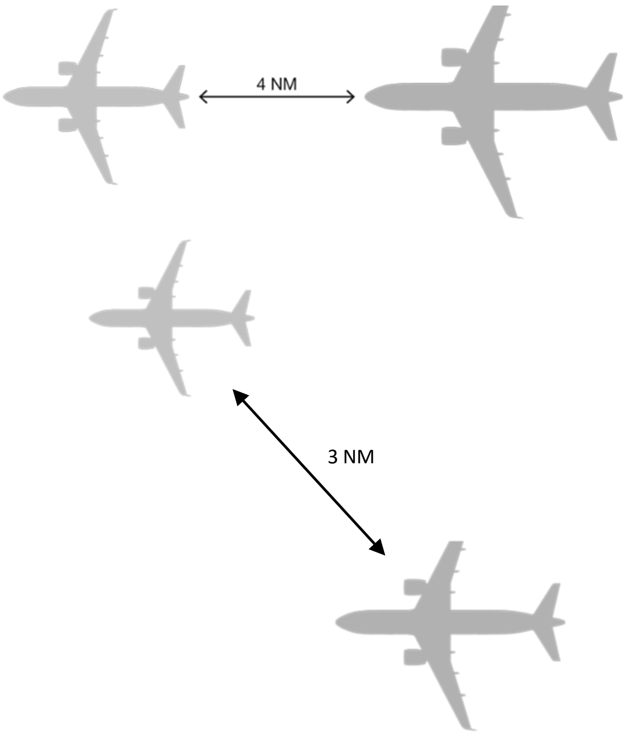




| LEADING A/C | TRAILING A/C | | | |
|---|---|---|---|---|
| | Super Heavy | Heavy | Medium | Light |
| Super Heavy | 2.5 | 6 | 7 | 8 |
| Heavy | 2.5 | 4 | 5 | 6 |
| Medium | 2.5 | 2.5 | 2.5 | 5 |
| Light | 2.5 | 2.5 | 2.5 | 2.5 |

# TMA Operations

- Optimization-based approach

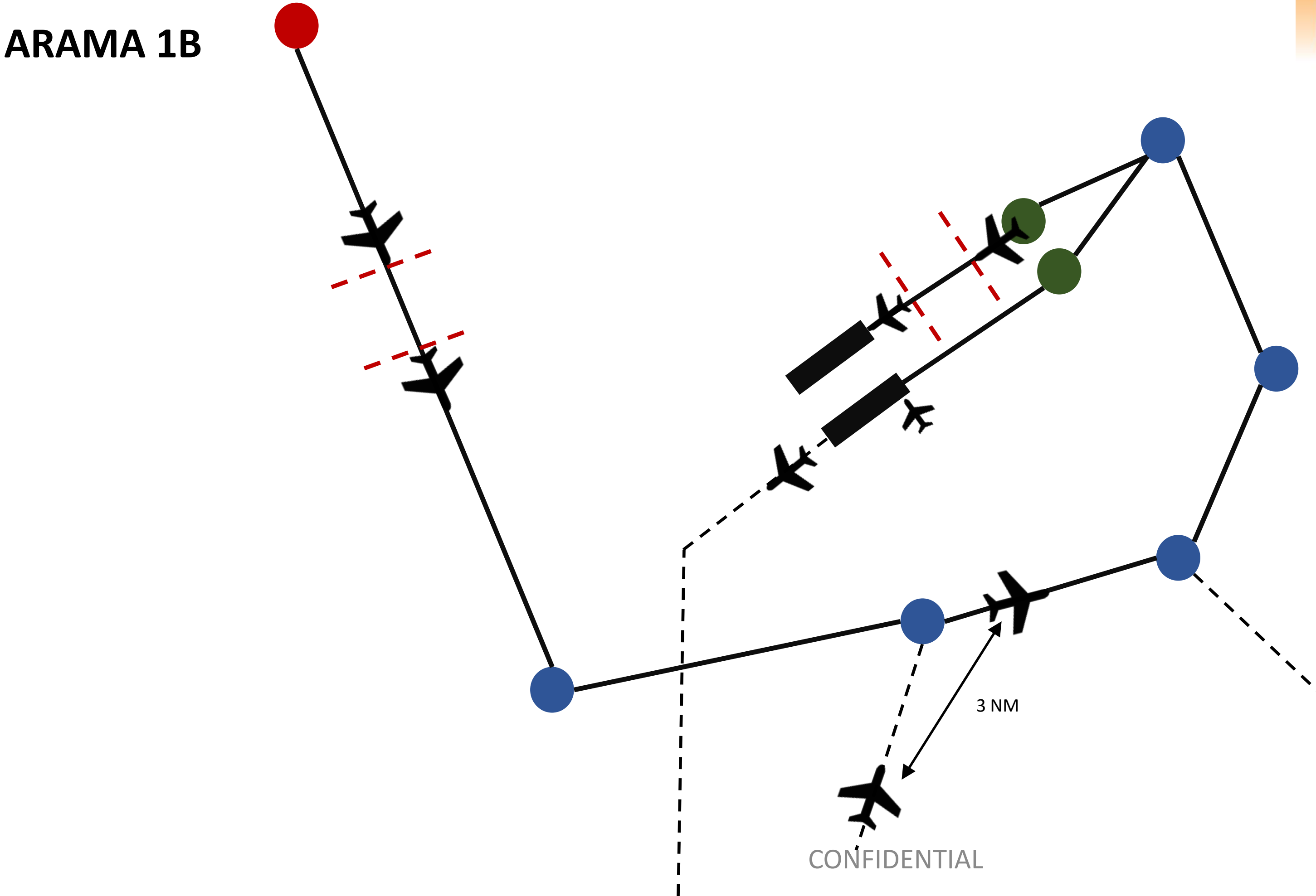


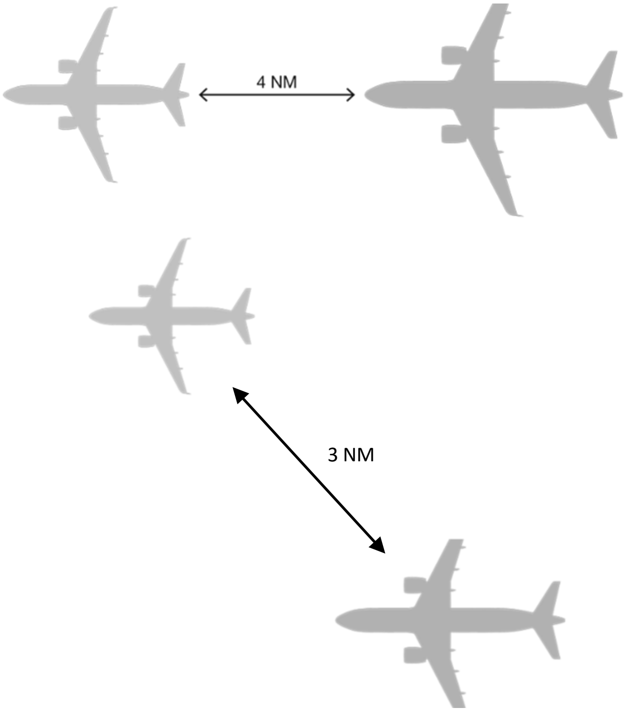




CONFIDENTIAL

# TMA Operations

- Optimization-based approach

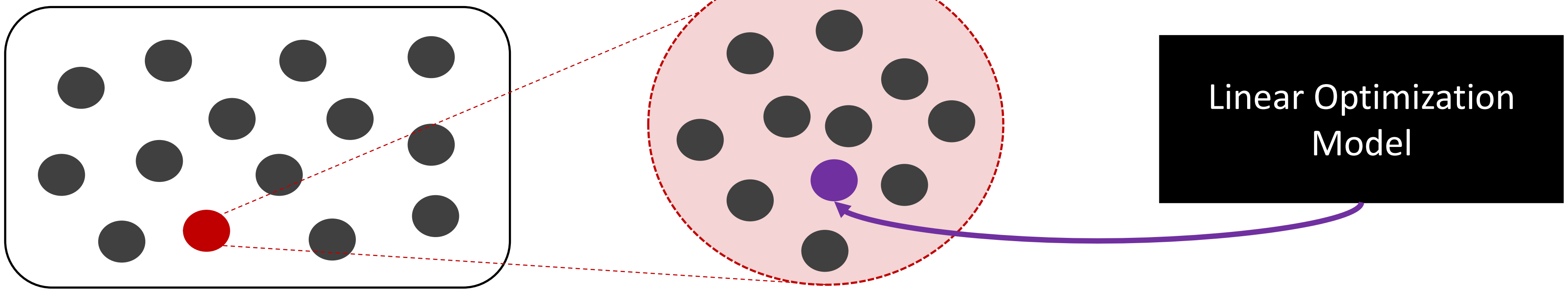



**Time-based Separation between Departures (e.g. 2 minutes)**

# Indirect Encoding

**Solution Space**




Solution representation: permutation encoding

| Aircraft Runway Sequencing | | | | | | | |
|---|---|---|---|---|---|---|---|
| Flights | 2 | 4 | 1 | 3 | … | 15 | 22 |

By fixing aircraft runway sequencing, aircraft speeds are optimized to minimize the amount of delays – the problem can be formulated as a linear optimization problem and can be solved in 30 seconds