```python
import gurobipy as gp  # import the installed package
```

```
Requirement already satisfied: gurobipy in /usr/local/lib/python3.11/dist-packages (12.0.2)
```

In [43]:
```python
from gurobipy import *
from gurobipy import GRB # explicitly import GRB from gurobipy
```

In [44]:
```python
# Create an environment with your WLS license
params = {
"WLSACCESSID": '********-****-****-****-************',
"WLSSECRET": '********-****-****-****-************',
"LICENSEID": *******,
}
env = gp.Env(params=params)

# Create the model within the Gurobi environment
model = gp.Model(env=env)
```

```
Set parameter WLSAccessID
Set parameter WLSSecret
Set parameter LicenseID to value 2619152
Academic license 2619152 - for non-commercial use only - registered to fi___@isel.pt
```

In [44]:

MODEL IMPLEMENTATION & SIMULATED ANNEALING

In [45]:
```python
k = random.randint(1, 5000)
random.seed(k)

#print(f'Seed in use: {k}')
```

Chosing the neighbour

Neighbour functions considered:

v1 Chooses 1 layer in σ, chooses 2 scraps in that layer and swaps their boxes

v2 Chooses 1 layer in σ and replaces the entire layer permutation for another

v3 Chooses 2 layers in σ, chooses 2 scraps in each and swaps their boxes

v4 Chooses 2 layers in σ and swaps their entire permutation for another

v5 Chooses n layers in σ, chooses 2 scraps in each and swaps their boxes

v6 Chooses n layers in σ and swaps their entire permutation for another **v7** Chooses $n\geq 2$ layers in σ and swaps their entire permutation for another

In [46]:
```
# Function to choose 2 scraps (within same layer) and swap them in sigma

def choose_and_swap(layer):

    # Make copy as a list to allow modification
    new_layer_perm = list(sigma[layer-1])

    # Pick two different scraps in the layer
    scrap1 = random.choice(sigma[layer-1])
    scrap2 = random.choice(sigma[layer-1])
    while scrap2 == scrap1:
        scrap2 = random.choice(sigma[layer-1])
    # Swap them
    new_layer_perm = [scrap2 if x == scrap1 else scrap1 if x == scrap2 else x for x in new_layer_perm] # Use the modifiable

    return new_layer_perm
```

In [47]:
```
# Function to choose new permutation of a given layer

def choose_new_perm(layer):

    # Pick random permutation in layer
    new_perm = random.choice(perms[layer-1])
    while new_perm == sigma[layer-1]:
        new_perm = random.choice(perms[layer-1])
    # Return the new permutation
    return new_perm
```

In [48]:
```
# v1 Chooses 1 layer and swaps 2 scraps in that layer

def get_neighbour_v1(sigma):

    # Make copy
    new_sigma = sigma.copy()
```

```python
    # Pick random layer
    i = random.choice([1,3,4,5,6,7])
    # Pick two different scraps in the chosen layer
    scrap1 = random.choice(sigma[i-1])
    scrap2 = random.choice(sigma[i-1])
    while scrap2 == scrap1:
        scrap2 = random.choice(sigma[i-1])
    # Swap them
    new_sigma[i-1] = [scrap2 if x == scrap1 else scrap1 if x == scrap2 else x for x in sigma[i-1]]

    return new_sigma
```

In [49]:

```python
# v2 Chooses 1 layer and replaces the layer's permutation for another

def get_neighbour_v2(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick random layer
    i = random.choice([1,3,4,5,6,7])
    # Pick random permutation in layer i
    new_layer_perm = random.choice(perms[i-1])
    while new_layer_perm == sigma[i-1]:
        new_layer_perm = random.choice(perms[i-1])
    # Replace old layer i perm by new layer i perm in sigma
    new_sigma[i-1] = new_layer_perm

    return new_sigma
```

In [50]:

```python
# v3 Chooses 2 layers and swaps 2 scraps in each

def get_neighbour_v3(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick two different random layers
    i = random.choice([1,3,4,5,6,7])
    j = random.choice([1,3,4,5,6,7])
    while j == i:
        j = random.choice([1,3,4,5,6,7])
```

```python
    # Swap scraps in chosen layers
    new_sigma[i-1] = tuple(choose_and_swap(i))
    new_sigma[j-1] = tuple(choose_and_swap(j))

    return new_sigma
```

In [51]:

```python
# v4 Chooses 2 layers and swaps its permutation for another

def get_neighbour_v4(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick two different random layers
    i = random.choice([1,3,4,5,6,7])
    j = random.choice([1,3,4,5,6,7])
    while j == i:
        j = random.choice([1,3,4,5,6,7])
    # Swap scraps in chosen layers
    new_sigma[i-1] = tuple(choose_new_perm(i))
    new_sigma[j-1] = tuple(choose_new_perm(j))

    return new_sigma
```

In [52]:
```
# v5 Chooses n layers to be modified and swaps 2 scraps in each

def get_neighbour_v5(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick the number of layers which will be changed
    n = random.randint(1, 6)
    print(n)

    # Pick n different random layers
    layers = random.sample([1,3,4,5,6,7], n)
    print(layers)
    # Swap scraps in chosen layers
    for i in layers:
        new_sigma[i-1] = tuple(choose_and_swap(i))

    return new_sigma
```

In [53]:
```
# v6 Chooses n layers to be modified and swaps their permutation for another

def get_neighbour_v6(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick the number of layers which will be changed
    n = random.randint(1, 6)
    print(n)

    # Pick n different random layers
    layers = random.sample([1,3,4,5,6,7], n)
    print(layers)
    # Swap scraps in chosen layers
    for i in layers:
        new_sigma[i-1] = tuple(choose_new_perm(i))

    return new_sigma
```

In [54]:
```
# v7 Chooses n»2 layers to be modified and swaps their permutation for another
```

```python
def get_neighbour_v7(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # Pick the number (at least 2) of layers which will be changed
    n = random.randint(2, 6)
    print(n)

    # Pick n different random layers
    layers = random.sample([1,3,4,5,6,7], n)
    print(layers)
    # Swap scraps in chosen layers
    for i in layers:
        new_sigma[i-1] = tuple(choose_new_perm(i))

    return new_sigma
```

Implementing & runing the model

In [55]:
```python
def run_model(sigma, time_limit, left_scraps, right_scraps, grades_list, jobs_eaf1_list, jobs_eaf2_list):

    global x

    model.reset()
    # OPTIONAL: Limit each iteration to time_limit seconds
    #model.setParam("TimeLimit", time_limit)

    # Sets
    L = left_scraps # Use the passed left_scraps argument
    R = right_scraps # Use the passed right_scraps argument
    S = sorted(left_scraps + right_scraps)
    G = grades_list
    P = list(range(len(jobs_eaf1_list)))
    Q = list(range(len(jobs_eaf2_list)))

    # Parameters
    box_width = {s:box_width_dict[s] for s in S}
    recipes = {g:get_recipe(g, sigma) for g in G}
    total_grabs = {g:total_grabs_dict[g] for g in G}
    # Comment for LPP:
    max_time = {g:grades_max_time_dict[g]/1000 for g in G}
    jobs_eaf1 = dict(zip(list(range(len(jobs_eaf1_list))), jobs_eaf1_list))
```

```python
jobs_eaf2 = dict(zip(list(range(len(jobs_eaf2_list))), jobs_eaf2_list))

# Variables
midpoint = model.addVars(S, name="midpoint")
x = model.addVars(S, S, vtype=GRB.BINARY, name="x")
aux = model.addVars(S, S, lb=-gp.GRB.INFINITY, name="aux")
dist = model.addVars(S, S, name="dist")
timetoready = model.addVars(G, name="timetoready")
U = model.addVar(name="U")
V = model.addVar(name="V")
W = model.addVar(name="W")

# Constrains
model.addConstrs((x[i,j] + x[j,i] == 1 for i in L for j in L if j != i), name="sides1a")
model.addConstrs((x[i,j] + x[j,i] == 1 for i in R for j in R if j != i), name="sides1b")
model.addConstr((sum(x[i,i] for i in S) == 0), name="sides2")
model.addConstrs((x[i,j] == 0 for i in L for j in R), name="sides3a")
model.addConstrs((x[i,j] == 0 for i in R for j in L), name="sides3b")
model.addConstrs((x[i,k] >= 1 - 10*(1-x[i,j]) - 10*(1-x[j,k]) for i in L for j in L for k in L), name="sides4a")
model.addConstrs((x[i,k] >= 1 - 10*(1-x[i,j]) - 10*(1-x[j,k]) for i in R for j in R for k in R), name="sides4b")
model.addConstrs((midpoint[s] == 20 + 1.7 + sum( (box_width[y] + 1.7)*x[y,s] for y in S) + box_width[s]/2
    for s in S), name="midpoints")
for i in S:
    for j in S:
        model.addConstr(aux[i, j] == midpoint[i] - midpoint[j], name=f'distanceaux_{i}_{j}')
        model.addConstr(dist[i,j] == abs_(aux[i,j]), name=f'distance_{i}_{j}')
# Comment for LPP:
model.addConstrs((timetoready[g] <= max_time[g] for g in G), name="maxtime")
model.addConstrs((timetoready[g] ==
    (6.2 + 2*1.47 + 0.67*total_grabs[g] + (midpoint[recipes[g][0]] + midpoint[recipes[g][-1]])/60 + sum( dist[recipes[g
    for i in range(len(recipes[g])-1))/40) for g in G), name="time")
model.addConstr((U == sum( timetoready[jobs_eaf1[p]] for p in P)), name="totaltime1")
model.addConstr((V == sum( timetoready[jobs_eaf2[q]] for q in Q)), name="totaltime2")
model.addConstr((U <= W), name = "totaltimeW1")
model.addConstr((V <= W), name = "totaltimeW2");

# Objective function
model.setObjective(W, GRB.MINIMIZE)

# Run model
model.optimize()

# do IIS if the model is infeasible
```

```python
    if model.Status == GRB.INFEASIBLE:
        model.computeIIS()

    # Output results
    new_t = W.X
    new_output = model.getVars()

    print(f'Sigma: {sigma}')
    print(f'Time: {new_t}')

    left_precedences = get_precedences(left_scraps)
    new_left_row = print_row(left_scraps, left_precedences)
    #print(f'Left row: {new_left_row}')
    right_precedences = get_precedences(right_scraps)
    new_right_row = print_row(right_scraps, right_precedences)
    #print(f'Right row: {new_right_row}')

    return new_t, new_output, new_left_row, new_right_row
```

Making the decision

In [56]:
```python
def get_decision(current_t, current_sigma, current_output, current_left_row, current_right_row,
                 new_t, new_sigma, new_output, new_left_row, new_right_row, beta, step):

    if new_t < current_t:
        current_t = new_t
        current_sigma = new_sigma
        current_output = new_output
        current_left_row = new_left_row
        current_right_row = new_right_row
        print('New best solution found!')
        print(f'New left row: {current_left_row}')
        print(f'New right row: {current_right_row}')
    else:
        p = random.random()
        if p < math.exp((current_t - new_t)/5*(1+beta*step)):
            current_t = new_t
            current_sigma = new_sigma
            current_output = new_output
            current_left_row = new_left_row
            current_right_row = new_right_row
            print('p update!')
            print(f'New left row: {current_left_row}')
```

```
            print(f'New right row: {current_right_row}')
        else:
            print('No update!')
            #pass

    return current_t, current_sigma, current_output, current_left_row, current_right_row
```

Simulated Annealing algorithm

In [57]:
```
currentWs = []
newWs = []
```

In [58]:
```
def simulated_annealing(prod_plan, split, n_iterations, time_limit, neighbour_fn, beta, sigma):

    # Prod plan
    if prod_plan == 'RPP':
        grades_list = grades_list_RPP
        jobs_eaf1_list = jobs_eaf1_list_RPP
        jobs_eaf2_list = jobs_eaf2_list_RPP
    else:
        grades_list = grades_list_LPP
        jobs_eaf1_list = jobs_eaf1_list_LPP
        jobs_eaf2_list = jobs_eaf2_list_LPP

    # Split
    left_scraps, right_scraps = get_scraps(split)

    # Initial sigma
    current_sigma = sigma

    # Intial solution
    current_t, current_output, current_left_row, current_right_row = run_model(current_sigma, time_limit, left_scraps, righ
                                                                               grades_list, jobs_eaf1_list, jobs_eaf2_list)
    currentWs.append(current_t)
    newWs.append(current_t)
    print(f'Left row: {current_left_row}')
    print(f'Right row: {current_right_row}')

    for i in range(n_iterations):

        print(f'\nIteration {i}')
```

```
        # Neighbout fn
        if neighbour_fn == 'v1':
          new_sigma = get_neighbour_v1(current_sigma)
        elif neighbour_fn == 'v2':
          new_sigma = get_neighbour_v2(current_sigma)
        elif neighbour_fn == 'v3':
          new_sigma = get_neighbour_v3(current_sigma)
        elif neighbour_fn == 'v4':
          new_sigma = get_neighbour_v4(current_sigma)
        elif neighbour_fn == 'v5':
          new_sigma = get_neighbour_v5(current_sigma)
        elif neighbour_fn == 'v6':
          new_sigma = get_neighbour_v6(current_sigma)
        else:
          new_sigma = get_neighbour_v7(current_sigma)

        new_t, new_output, new_left_row, new_right_row = run_model(new_sigma, time_limit, left_scraps, right_scraps,
                                                                   grades_list, jobs_eaf1_list, jobs_eaf2_list)

        print(f'Current W: {current_t}')
        print(f'New W: {new_t}')
        print(f'Current sigma: {current_sigma}')
        print(f'New sigma: {new_sigma}')
        print(f'Current left row: {current_left_row}')
        print(f'New left row: {new_left_row}')
        print(f'Current right row: {current_right_row}')
        print(f'New right row: {new_right_row}')

        current_t, current_sigma, current_output, current_left_row, current_right_row = get_decision(current_t, current_sigm
                                                                                                      current_left_row, curre
                                                                                                      new_t, new_sigma, new_o
                                                                                                      new_left_row, new_righ

        currentWs.append(current_t)
        newWs.append(new_t)
        print(f'Current W: {current_t}')
        print(f'Current sigma: {current_sigma}')
        print(f'Current left row: {current_left_row}')
        print(f'Current right row: {current_right_row}')

    return current_t, current_sigma, current_output, current_left_row, current_right_row
```

In [58]:

SA parameters

```
In [59]: prod_plan = 'RPP'                                                      # RPP / LPP
         split = 'split3_05'                                                    # split0 / split1 / split2a / split2b / spl
         n_iterations = 0                                                       # nonzero for SA
         time_limit = None                                                      # OPTIONAL; in seconds
         beta = 0.05
         neighbour_fn = 'v1'                                                    # v1 / v2 / v3 / v4 / v5 / v6 / v7


         # Layer permutations in increasing order
         sigma0 = [perms[i][0] for i in range(7)]

         # Random inital sigma
         sigma = [random.choice(perms[i]) for i in range(7)]
```

```
In [60]: start_time = datetime.now()
```

```
In [61]: # run SA

         final_t, final_sigma, final_output, final_left_row, final_right_row = simulated_annealing(prod_plan, split, n_iterations, t
                                                                                  neighbour_fn, beta, sigma0)
```

```
Discarded solution information
Gurobi Optimizer version 12.0.2 build v12.0.2rc0 (linux64 - "Ubuntu 22.04.4 LTS")

CPU model: Intel(R) Xeon(R) CPU @ 2.20GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 1 physical cores, 2 logical processors, using up to 2 threads

Academic license 2619152 - for non-commercial use only - registered to fi___@isel.pt
Optimize a model with 10369 rows, 3121 columns and 29585 nonzeros
Model fingerprint: 0x55cc992d
Model has 1024 simple general constraints
  1024 ABS
Variable types: 2097 continuous, 1024 integer (1024 binary)
Coefficient statistics:
  Matrix range     [2e-02, 7e+01]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 5e+03]
Presolve removed 7853 rows and 2634 columns
Presolve time: 0.12s
Presolved: 2516 rows, 487 columns, 7892 nonzeros
Variable types: 165 continuous, 322 integer (296 binary)
Found heuristic solution: objective 4281.2670833
Found heuristic solution: objective 4158.6937500
Extra simplex iterations after uncrush: 27

Root relaxation: objective 3.056452e+03, 637 iterations, 0.10 seconds (0.02 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 3056.45174    0  151 4158.69375 3056.45174  26.5%     -    0s
H    0     0                    4070.3820833 3056.45174  24.9%     -    0s
H    0     0                    4023.6495833 3056.45174  24.0%     -    0s
     0     0 3056.55160    0  166 4023.64958 3056.55160  24.0%     -    0s
     0     0 3056.57295    0  153 4023.64958 3056.57295  24.0%     -    0s
H    0     0                    3983.1837500 3056.57295  23.3%     -    0s
     0     0 3056.57295    0  146 3983.18375 3056.57295  23.3%     -    0s
     0     0 3056.57295    0  149 3983.18375 3056.57295  23.3%     -    0s
H    0     0                    3968.6745833 3056.57295  23.0%     -    0s
H    0     0                    3959.8995833 3056.57295  22.8%     -    0s
     0     0 3056.57295    0  149 3959.89958 3056.57295  22.8%     -    0s
     0     0 3056.60816    0  150 3959.89958 3056.60816  22.8%     -    1s
H    0     0                    3868.1712500 3056.60816  21.0%     -    1s
```

```
     0     0 3056.60816    0  149 3868.17125 3056.60816  21.0%     -    1s
     0     0 3056.60816    0  147 3868.17125 3056.60816  21.0%     -    1s
H    0     0                    3839.6095833 3056.60816  20.4%     -    1s
H    0     0                    3811.5595833 3056.60816  19.8%     -    1s
     0     0 3056.60816    0  147 3811.55958 3056.60816  19.8%     -    1s
     0     0 3056.60816    0  150 3811.55958 3056.60816  19.8%     -    1s
H    0     0                    3802.5595833 3056.60816  19.6%     -    1s
     0     0 3056.60816    0  147 3802.55958 3056.60816  19.6%     -    1s
     0     0 3056.60816    0  147 3802.55958 3056.60816  19.6%     -    1s
H    0     0                    3764.8504167 3056.60816  18.8%     -    1s
H    0     0                    3648.2479167 3056.60816  16.2%     -    1s
H    0     0                    3638.4412500 3056.60816  16.0%     -    1s
     0     0 3056.60816    0  147 3638.44125 3056.60816  16.0%     -    1s
     0     0 3056.60816    0  147 3638.44125 3056.60816  16.0%     -    1s
     0     0 3056.60816    0  147 3638.44125 3056.60816  16.0%     -    2s
     0     2 3056.60816    0  147 3638.44125 3056.60816  16.0%     -    2s
H   26    26                    3620.6937500 3073.09693  15.1%   334    3s
H   27    27                    3612.8612500 3073.09693  14.9%   325    3s
H   52    52                    3612.7712500 3073.09693  14.9%   228    3s
H   54    54                    3496.4787500 3073.09693  12.1%   221    3s
H   78    76                    3452.2604167 3073.09693  11.0%   190    4s
H   78    76                    3448.7679167 3073.09693  10.9%   190    4s
H   81    77                    3422.9904167 3073.09693  10.2%   186    4s
H  104    90                    3362.6679167 3073.09693  8.61%   157    4s
   120   104 3352.62488   13  165 3362.66792 3073.09693  8.61%   168    5s
H  130   100                    3352.3329167 3083.07553  8.03%   170    5s
H  156   124                    3348.6279167 3083.07553  7.93%   167    5s
H  156   122                    3346.5629167 3083.07553  7.87%   167    5s
H  186   128                    3344.8079167 3109.48842  7.04%   179    6s
H  243   141                    3343.4970833 3113.60055  6.88%   162    7s
H  276   150                    3342.1870833 3137.36629  6.13%   174    7s
   390   190 3208.42088    5  174 3342.18708 3151.28476  5.71%   169   10s
H  397   184                    3340.1245833 3151.28476  5.65%   168   10s
   680   216 3312.83241   17   90 3340.12458 3209.61629  3.91%   175   15s
  1002   225     cutoff   16      3340.12458 3251.35342  2.66%   178   20s
  1288   278 3320.96085   22  147 3340.12458 3273.61939  1.99%   171   25s
  1427   296 3326.85978   22  162 3340.12458 3273.61939  1.99%   178   30s
  1721   280 3327.88350   30  113 3340.12458 3282.08820  1.74%   172   35s
  2161   194 3336.51542   36   35 3340.12458 3298.10496  1.26%   157   40s
  2613   156 3339.58241   32   75 3340.12458 3313.33517  0.80%   149   45s

Cutting planes:
  Implied bound: 2
```

```
  MIR: 6
  Flow cover: 1
  Inf proof: 1

Explored 3174 nodes (425591 simplex iterations) in 49.27 seconds (30.95 work units)
Thread count was 2 (of 2 available processors)

Solution count 10: 3340.12 3342.19 3342.19 ... 3362.67

Optimal solution found (tolerance 1.00e-04)
Best objective 3.340124583333e+03, best bound 3.340096098626e+03, gap 0.0009%
Sigma: [(6, 8, 10, 12, 13, 18, 26), (28,), (4, 5), (0, 1, 2, 3, 7, 9, 11, 22, 23), (15, 17, 29, 30), (14, 16, 21, 31), (19,
20, 24, 25, 27)]
Time: 3340.124583333332
Left row: [2, 10, 11, 31, 8, 6, 0, 28, 9, 20, 14, 15, 24, 23, 12]
Right row: [26, 30, 18, 4, 5, 1, 3, 7, 27, 16, 17, 19, 29, 21, 25, 13, 22]
```

In [61]:

```

```

SA results

In [62]:

```
end_time = datetime.now()
runtime = (end_time - start_time).total_seconds()
print('Runtime (sec): {}'.format(runtime))
```

```
Runtime (sec): 50.259908
```

In [63]:

```
print(f'Prod plan: {prod_plan}')
print(f'Split: {split}')
print(f'Initial sigma: sigma0')                                          # sigma0 / random
if n_iterations != 0:
    print(f'Number of iterations: {n_iterations}')
    print(f'Time limit: {time_limit}')
    print(f'beta: {beta}')
    print(f'Neighbour fn: {neighbour_fn}')
    print(f'Seed: {k}')
```

```
Prod plan: RPP
Split: split3_05
Initial sigma: sigma0
```

In [64]:

```
print(f'Final W: {final_t}')
print(f'Final sigma: {final_sigma}')
print(f'Final left row: {final_left_row}')
```

```
print(f'Final right row: {final_right_row}')
```

```
Final W: 3340.124583333332
Final sigma: [(6, 8, 10, 12, 13, 18, 26), (28,), (4, 5), (0, 1, 2, 3, 7, 9, 11, 22, 23), (15, 17, 29, 30), (14, 16, 21, 31),
(19, 20, 24, 25, 27)]
Final left row: [2, 10, 11, 31, 8, 6, 0, 28, 9, 20, 14, 15, 24, 23, 12]
Final right row: [26, 30, 18, 4, 5, 1, 3, 7, 27, 16, 17, 19, 29, 21, 25, 13, 22]
```

In [65]:

```
print(f'currentWs = {currentWs}')
print(f'newWs = {newWs}')
```

```
currentWs = [3340.124583333332]
newWs = [3340.124583333332]
```

In [65]:

In [65]: