

```
'G44', 'G44', 'G44', 'G44', 'G67', 'G67', 'G67', 'G67',  
'G67', 'G47', 'G47', 'G47', 'G47', 'G47', 'G11', 'G11',  
'G11', 'G11', 'G11', 'G65', 'G65', 'G65']
```

```
jobs_eaf2_list_RPP = ['G11', 'G11', 'G11', 'G40', 'G40', 'G40', 'G11', 'G11',  
                     'G11', 'G28', 'G28', 'G28', 'G09', 'G09', 'G09', 'G11',  
                     'G11', 'G11', 'G07', 'G07', 'G07', 'G28', 'G28', 'G28',  
                     'G07', 'G07', 'G78', 'G78', 'G78', 'G08', 'G08', 'G08',  
                     'G47', 'G47', 'G78', 'G78', 'G78', 'G11', 'G11', 'G65',  
                     'G65', 'G65']
```

Grades that appear in the real production plan

```
grades_list_RPP = ['G07', 'G08', 'G09', 'G11', 'G13', 'G28', 'G40', 'G44',  
                  'G46', 'G47', 'G64', 'G65', 'G67', 'G78']
```

▼ Leveled Production plan data (LPP)

```
jobs_eaf1_list_LPP = ['G40', 'G12', 'G08', 'G37', 'G35', 'G31', 'G29', 'G51',  
                     'G59', 'G32', 'G33', 'G24', 'G25', 'G22', 'G46', 'G28',  
                     'G66', 'G60', 'G82', 'G80', 'G42', 'G17', 'G07', 'G73',  
                     'G15', 'G14', 'G26', 'G78', 'G79', 'G77', 'G11', 'G75',  
                     'G72', 'G76', 'G54', 'G68', 'G74', 'G48', 'G55', 'G44',  
                     'G49']
```

```
jobs_eaf2_list_LPP = ['G36', 'G18', 'G52', 'G58', 'G30', 'G10', 'G16', 'G39',  
                     'G34', 'G21', 'G61', 'G69', 'G41', 'G04', 'G27', 'G43',  
                     'G20', 'G09', 'G65', 'G70', 'G81', 'G83', 'G05', 'G56',  
                     'G38', 'G13', 'G64', 'G23', 'G50', 'G03', 'G67', 'G19',  
                     'G06', 'G01', 'G02', 'G45', 'G53', 'G71', 'G57', 'G47',  
                     'G62', 'G63']
```

Complete list of grades

```
grades_list_LPP = ['G01', 'G02', 'G03', 'G04', 'G05', 'G06', 'G07', 'G08',  
                  'G09', 'G10', 'G11', 'G12', 'G13', 'G14', 'G15', 'G16',  
                  'G17', 'G18', 'G19', 'G20', 'G21', 'G22', 'G23', 'G24',  
                  'G25', 'G26', 'G27', 'G28', 'G29', 'G30', 'G31', 'G32',
```

```
'G33', 'G34', 'G35', 'G36', 'G37', 'G38', 'G39', 'G40',  
'G41', 'G42', 'G43', 'G44', 'G45', 'G46', 'G47', 'G48',  
'G49', 'G50', 'G51', 'G52', 'G53', 'G54', 'G55', 'G56',  
'G57', 'G58', 'G59', 'G60', 'G61', 'G62', 'G63', 'G64',  
'G65', 'G66', 'G67', 'G68', 'G69', 'G70', 'G71', 'G72',  
'G73', 'G74', 'G75', 'G76', 'G77', 'G78', 'G79', 'G80',  
'G81', 'G82', 'G83']
```

Start coding or [generate](#) with AI.

▼ SPLITS

Split (0) - left / right split of boxes from the Layout 0 (for Layout 0 star)

```
split0star_left_scraps = sorted([26, 18, 13, 4, 1, 23, 22, 3, 15, 14, 31, 24, 19, 25])  
split0star_right_scraps = sorted([10, 6, 8, 12, 28, 5, 2, 11, 7, 9, 0, 30, 17, 29, 16, 21, 20, 27])
```

Split (1) - optimized left/right split of scrap types based on the perfect balance between the total length of both sides of the yard

```
split1_left_scraps = [0, 2, 8, 9, 14, 17, 18, 20, 22, 24, 27, 30, 31]  
split1_right_scraps = [1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 15, 16, 19, 21, 23, 25, 26, 28, 29]
```

Split (2a) - optimized left/right split of scrap types based on:

1. balance between the total length of both sides of the yard;
2. balanced split for scraps of the layers in A .

```
split2a_left_scraps = [0, 3, 4, 5, 6, 7, 15, 16, 18, 19, 21, 22, 25, 26, 27]  
split2a_right_scraps = [1, 2, 8, 9, 10, 11, 12, 13, 14, 17, 20, 23, 24, 28, 29, 30, 31]
```

Split (2b) - optimized left/right split of scrap types based on:

1. balanced split for scraps of the layers in A ;
2. balance between the total length of both sides of the yard.

```
split2b_left_scraps = [0, 2, 4, 5, 7, 9, 11, 13, 16, 17, 18, 20, 21, 24, 26, 29, 30]  
split2b_right_scraps = [1, 3, 6, 8, 10, 12, 14, 15, 19, 22, 23, 25, 27, 28, 31]
```

Split (3) - optimized left/right split of scrap types based on the balance between the total length of both sides of the yard with weight 0.5 and the balance between the split for scraps of layer 1, 4, 5, 6 and 7 with weight 0.5.

```
# alpha = 1.0
```

```
split3_10_left_scraps = [0, 2, 8, 9, 14, 17, 18, 20, 22, 24, 27, 30, 31]  
split3_10_right_scraps = [1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 15, 16, 19, 21, 23, 25, 26, 28, 29]
```

```
# alpha = 0.9
```

```
split3_09_left_scraps = [2, 7, 8, 9, 10, 11, 12, 13, 14, 17, 20, 23, 24, 28, 29, 30, 31]  
split3_09_right_scraps = [0, 1, 3, 4, 5, 6, 15, 16, 18, 19, 21, 22, 25, 26, 27]
```

```
# alpha = 0.8
```

```
split3_08_left_scraps = [4, 5, 7, 11, 13, 16, 17, 18, 19, 21, 22, 23, 25, 26, 27, 29, 30]  
split3_08_right_scraps = [0, 1, 2, 3, 6, 8, 9, 10, 12, 14, 15, 20, 24, 28, 31]
```

```
# alpha = 0.7
```

```
split3_07_left_scraps = [4, 5, 7, 11, 13, 16, 17, 18, 19, 21, 22, 23, 25, 26, 27, 29, 30]  
split3_07_right_scraps = [0, 1, 2, 3, 6, 8, 9, 10, 12, 14, 15, 20, 24, 28, 31]
```

```
# alpha = 0.6
```

```
split3_06_left_scraps = [0, 2, 4, 5, 9, 11, 13, 16, 17, 18, 20, 21, 23, 24, 26, 29, 30]  
split3_06_right_scraps = [1, 3, 6, 7, 8, 10, 12, 14, 15, 19, 22, 25, 27, 28, 31]
```

```

# alpha = 0.5

split3_05_left_scraps = [0, 2, 6, 8, 9, 10, 11, 12, 14, 15, 20, 23, 24, 28, 31]
split3_05_right_scraps = [1, 3, 4, 5, 7, 13, 16, 17, 18, 19, 21, 22, 25, 26, 27, 29, 30]

# alpha = 0.4

split3_04_left_scraps = [6, 7, 8, 10, 11, 12, 14, 15, 19, 22, 23, 25, 27, 28, 31]
split3_04_right_scraps = [0, 1, 2, 3, 4, 5, 9, 13, 16, 17, 18, 20, 21, 24, 26, 29, 30]

# alpha = 0.3

split3_03_left_scraps = [4, 5, 7, 11, 13, 16, 17, 18, 19, 21, 22, 23, 25, 26, 27, 29, 30]
split3_03_right_scraps = [0, 1, 2, 3, 6, 8, 9, 10, 12, 14, 15, 20, 24, 28, 31]

# alpha = 0.2

split3_02_left_scraps = [6, 7, 8, 10, 11, 12, 14, 15, 19, 22, 23, 25, 27, 28, 31]
split3_02_right_scraps = [0, 1, 2, 3, 4, 5, 9, 13, 16, 17, 18, 20, 21, 24, 26, 29, 30]

# alpha = 0.1

split3_01_left_scraps = [0, 1, 2, 6, 8, 9, 10, 12, 14, 15, 20, 23, 24, 28, 31]
split3_01_right_scraps = [3, 4, 5, 7, 11, 13, 16, 17, 18, 19, 21, 22, 25, 26, 27, 29, 30]

# alpha = 0.0

split3_00_left_scraps = [0, 2, 4, 5, 9, 11, 13, 15, 16, 18, 20, 21, 23, 24, 26, 28]
split3_00_right_scraps = [1, 3, 6, 7, 8, 10, 12, 14, 17, 19, 22, 25, 27, 29, 30, 31]

# Function to set left_scraps and right_scraps according to the split

def get_scraps(split):
    left var name = f"{split} left scraps"

```

```

right_var_name = f"{split}_right_scraps"

# Fallback if split is not recognized
if left_var_name not in globals() or right_var_name not in globals():
    left_var_name = split3_05_left_scraps
    right_var_name = split3_05_right_scraps
    print('Error! Using Split 3 0.5 as default')

left_scraps = globals()[left_var_name]
right_scraps = globals()[right_var_name]

return left_scraps, right_scraps

```

Start coding or [generate](#) with AI.

✓ LAYER PERMUTATION

```

layer_1 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 1.0]
layer_2 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 2.0]
layer_3 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 3.0]
layer_4 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 4.0]
layer_5 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 5.0]
layer_6 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 6.0]
layer_7 = [key for key in scrap_layer_dict.keys() if scrap_layer_dict[key] == 7.0]
#print(layer_1)
#print(layer_2)
#print(layer_3)
#print(layer_4)
#print(layer_5)
#print(layer_6)
#print(layer_7)

layer_1_perms = list(permutations(layer_1))
layer_2_perms = list(permutations(layer_2))
layer_3_perms = list(permutations(layer_3))
layer 4 perms = list(permutations(layer 4))

```

```

layer_5_perms = list(permutations(layer_5))
layer_6_perms = list(permutations(layer_6))
layer_7_perms = list(permutations(layer_7))
#layer_1_perms

```

```

perms = [layer_1_perms, layer_2_perms, layer_3_perms, layer_4_perms, layer_5_perms, layer_6_perms, layer_7_perms]

```

Function to get recipe out of a permutation for all layers

```

def get_recipe(grade, sigma):
    # Input: a grade (eg. 'G07') and a sequence of all scraps in each layer
    # Output: the recipe for the grade using the sequences

    ingredients = grade_ingredients_dict[grade]
    recipe = [scrap for scrap in sigma[0] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[1] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[2] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[3] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[4] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[5] if scrap in ingredients]
    recipe += [scrap for scrap in sigma[6] if scrap in ingredients]

    return recipe

```

✓ GET INFO FROM OUTPUT

✓ Layout

```

def get_precedences(scrap):
    precedences = []

    for i in scraps:
        for j in scraps:
            if x[i].il.X == 1:

```

```

        precedences.append(f'[{i},{j}]')

    return precedences

def print_row(scraps, precedences):
    aux = []
    for i in sorted(scraps):
        for j in sorted(scraps):
            if f'[{i},{j}]' in precedences:
                aux.append(i)

    row = []
    for k in range(len(scraps)):
        for i in sorted(scraps):
            if aux.count(i) == k:
                row.append(i)

    return row

```

✓ Yard total length

```

def compute_side_length(side):
    length = 20*2 + 1.7
    for scrap in side:
        length += box_width_dict[scrap] + 1.7

    return length

def compute_yard_length(L, R):

    return max(compute_side_length(L), compute_side_length(R))

```

✓ Loading time for the production plan

```

def compute_midpoints(L, R):
    midpoints = {}

    L_aux = L.copy()
    L_aux.reverse()
    midpoint = 20 + 1.7 + box_width_dict[L_aux[0]]/2
    midpoints[L_aux[0]] = midpoint
    for i in range(1, len(L_aux)):
        midpoint += box_width_dict[L_aux[i-1]]/2 + 1.7 + box_width_dict[L_aux[i]]/2
        midpoints[L_aux[i]] = midpoint

    R_aux = R.copy()
    R_aux.reverse()
    midpoint = 20 + 1.7 + box_width_dict[R_aux[0]]/2
    midpoints[R_aux[0]] = midpoint
    for i in range(1, len(R_aux)):
        midpoint += box_width_dict[R_aux[i-1]]/2 + 1.7 + box_width_dict[R_aux[i]]/2
        midpoints[R_aux[i]] = midpoint

    return midpoints

def get_grade_loading_time(grade, sigma, L, R):

    recipe = get_recipe(grade, sigma)
    total_grabs = total_grabs_dict[grade]

    midpoint = compute_midpoints(L, R)

    time = 6.2 + 2*1.47 + 0.67*total_grabs + (midpoint[recipe[0]] + midpoint[recipe[-1]])/60
    for i in range(len(recipe)-1):
        time += abs(midpoint[recipe[i]]-midpoint[recipe[i+1]])/40

    return time

def get_eaf_loading_time(prodplan, sigma, L, R):

    time = 0
    for grade in prodplan:

```



```

for grade in prodplan:
    time += get_grade_loading_time(grade, sigma, L, R)

return time

```

```

def get_prod_plan_loading_time(prodplan1, prodplan2, sigma, L, R):

    return max(get_eaf_loading_time(prodplan1, sigma, L, R), get_eaf_loading_time(prodplan2, sigma, L, R))

```

Start coding or [generate](#) with AI.

✓ MODEL FORMULATION

This is the model to be run in each iteration of the simulated annealing. For each grade, there is a single recipe to consider.

✓ Sets

S is the set of scrap types

L is the set of scraps on the left side of the scrap yard

R is the set of scraps on the right side of the scrap yard

G is the set of grades

P is the sequence of jobs in EAF1

Q is the sequence of jobs in EAF2

✓ Parameters

- For each scrap type $s \in S$ we are given the width of the box containing the scrap, w_s .
- For each grade $g \in G$ we are given a recipe, z_g .

- For each grade $g \in G$ we are given the total number of grabs required to load the necessary quantities, n_g .
- (ONLY FOR REAL PROD PLAN) For each grade $g \in G$ we are given the maximum time allowed for the loading of the bucket, r_g (in ms).
- For each job $p \in P$ in EAF1 we are given the corresponding grade g_p .
- For each job $q \in Q$ in EAF2 we are given the corresponding grade h_q .

✓ Variables

For each scrap type $s \in S$:

- m_s is a non-negative continuous variable describing the midpoint of the box containing s , starting at the entry of the scrap yard closer to the EAFs.

For each pair of scrap types $s_1, s_2 \in S$:

- x_{s_1, s_2} is a binary variable that equals 1 iff s_1 and s_2 are both on the left or both on the right side of the scrap yard and scrap type s_1 is closer to the EAFs than s_2 ;
- d_{s_1, s_2} is a non-negative continuous variable describing the distance between m_{s_1} and m_{s_2} (regardless of their being or not in the same side of the scrap yard).

For each grade $g \in G$:

- t_g is a non-negative continuous variable describing the time it takes to have the bucket loaded with z_g ready at the EAF, including the unloading of the previous mixture.

And also:

- U is a non-negative continuous variable describing the total time taken to load all grades bound to EAF1;
- V is a non-negative continuous variable describing the total time taken to load all grades bound to EAF2;
- W is a non-negative continuous variable describing the total time taken to load the whole production plan.

✓ Constraints

- Each midpoint m_s is given by (in meters):

- Each midpoint m_s is given by (in meters),

$$m_s = 20 + 1.7 + \left(\sum_{s^*=0}^{31} (w_{s^*} + 1.7)x_{s^*,s} \right) + \frac{w_s}{2}, \quad \forall s \in S$$

- Given any two scrap types on the same side of the scrap yard, exactly one sits closer to the EAFs:

$$\begin{aligned} x_{s_1,s_2} + x_{s_2,s_1} &= 1, \quad \forall s_1, s_2 \in L \\ x_{s_1,s_2} + x_{s_2,s_1} &= 1, \quad \forall s_1, s_2 \in R \end{aligned}$$

- No scrap type precedes itself:

$$x_{s,s} = 0, \quad \forall s \in S$$

- Given any two scrap types on different sides of the scrap yard, no one precedes the other:

$$\begin{aligned} x_{s_1,s_2} &= 0, \quad \forall s_1 \in L, s_2 \in R \\ x_{s_1,s_2} &= 0, \quad \forall s_1 \in R, s_2 \in L \end{aligned}$$

- Variables x are transitive:

$$\begin{aligned} x_{s_1,s_2} = x_{s_2,s_3} = 1 &\Rightarrow x_{s_1,s_3} = 1, \quad \forall s_1, s_2, s_3 \in L \\ x_{s_1,s_2} = x_{s_2,s_3} = 1 &\Rightarrow x_{s_1,s_3} = 1, \quad \forall s_1, s_2, s_3 \in R \end{aligned}$$

- The distance between any two scrap types:

$$d_{s_1,s_2} = |m_{s_1} - m_{s_2}|, \quad \forall s_1, s_2 \in S$$

- The time necessary to load a grade $g \in G$ into the bucket (including the unloading of the previous grade) is given by (in minutes):

$$t_g = 6.2 + 1.47 \times 2 + 0.67n_g + \frac{m_{z_g[0]} + m_{z_g[-1]}}{60} + \sum_{i=0}^{\text{len}(z_g)-2} \frac{d_{z_g[i],z_g[i+1]}}{40}, \quad \forall g \in G$$

- (ONLY FOR REAL PROD PLAN) The time necessary to load a grade $g \in G$ into the bucket cannot exceed the maximum time allowed:

$$t_g \leq \frac{r_g}{1000}, \quad \forall g \in G$$

- The total time it takes to load all grades in the production plan:

$$\begin{aligned} U &= \sum_{p \in P} t_{g_p} \\ V &= \sum_{q \in Q} t_{h_q} \end{aligned}$$

- The variable to minimize:

$$W \geq U$$

Start coding or [generate](#) with AI.

✓ MODEL IMPLEMENTATION & SIMULATED ANNEALING

```
%%skip
```

```
k = random.randint(1, 5000)
random.seed(k)
```

```
k = 4251
random.seed(k)
```

✓ Choosing 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

```
# 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]

return new_layer_perm

```

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

```

v1 Chooses 1 layer and swaps 2 scraps in that layer

```

def get_neighbour_v1(sigma):

    # Make copy
    new_sigma = sigma.copy()

    # 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
```

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
```

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])
    # 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
```

```
# 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
```

```
# 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
```

```
# 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
```

```
# v7 Chooses n»2 layers to be modified and swaps their permutation for another
```

```
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 & running the model

```

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))
    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][i],recipes[g][i+1]]
    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
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

```

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_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

```

currentWs = []
newWs = []

```

```

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, right_scraps,
                                                                                   prod_plan, neighbour_fn, beta, sigma)

```

```
grades_list, jobs_eat1_list, jobs_eat2_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}')
```

```
# Neighbour 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)
```

[illegible]

```
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,
```

```

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

```

```

current_sigma,
current_output,
current_left_row,
current_right_row,
new_t,
new_sigma,
new_output,
new_left_row,
new_right_row,
beta,
i)

```

Start coding or [generate](#) with AI.

▼ **SA parameters**

```

prod_plan = 'RPP'                # RPP / LPP
split = 'split3_05'              # split0 / split1 / split2a / split2b / split3_xx
n_iterations = 250
time_limit = None                # OPTIONAL; in seconds
beta = 0.05
neighbour_fn = 'v5'              # v1 / v2 / v3 / v4 / v5 / v6

```

```

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

```

```

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

```

```
start_time = datetime.now()
```

```
# run SA
```

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

Streaming output truncated to the last 5000 lines.

H	208	148			3344.9095833	3094.22197	7.49%	129	22s
H	234	151			3337.2220833	3094.22197	7.28%	132	22s
H	234	128			3328.3345833	3094.22197	7.03%	132	22s
H	234	124			3326.5795833	3094.22197	6.98%	132	22s
H	234	123			3324.8245833	3094.22197	6.94%	132	22s
H	260	124			3314.2812500	3098.46493	6.51%	141	23s
H	324	147			3310.7712500	3108.89122	6.10%	144	24s
H	325	145			3306.0462500	3108.89122	5.96%	144	24s
	361	159	cutoff	13	3306.04625	3112.17611	5.86%	143	25s
	766	207	cutoff	7	3306.04625	3204.84847	3.06%	144	30s
H	1021	223			3306.0462497	3227.82040	2.37%	141	33s
	1141	225	3303.49702	14	143 3306.04625	3237.91008	2.06%	141	35s
	1635	160	cutoff	39	3306.04625	3273.27958	0.99%	131	40s

Cutting planes:

Gomory: 1

Cover: 7

MIR: 52

Flow cover: 6

Inf proof: 6

Explored 1858 nodes (234697 simplex iterations) in 41.47 seconds (23.89 work units)

Thread count was 2 (of 2 available processors)

Solution count 10: 3306.05 3306.05 3310.77 ... 3352.75

Optimal solution found (tolerance 1.00e-04)

Best objective 3.306046249669e+03, best bound 3.305785499071e+03, gap 0.0079%

Sigma: [(26, 18, 8, 12, 13, 6, 10), (28,), (5, 4), (2, 1, 9, 0, 3, 11, 7, 23, 22), (15, 29, 30, 17), (16, 21, 14, 31), (25,

Time: 3306.0462496692694

Current W: 3288.2562499999945

New W: 3306.0462496692694

Current sigma: [(26, 18, 8, 12, 13, 6, 10), (28,), (5, 4), (2, 1, 9, 0, 3, 11, 7, 23, 22), (17, 29, 15, 30), (16, 21, 14,

New sigma: [(26, 18, 8, 12, 13, 6, 10), (28,), (5, 4), (2, 1, 9, 0, 3, 11, 7, 23, 22), (15, 29, 30, 17), (16, 21, 14, 31),

Current left row: [2, 11, 10, 31, 8, 6, 9, 0, 28, 20, 14, 24, 15, 12, 23]

New left row: [2, 10, 11, 31, 8, 6, 9, 0, 28, 20, 14, 24, 15, 12, 23]

Current right row: [30, 26, 18, 5, 4, 1, 3, 7, 17, 16, 27, 25, 19, 21, 29, 13, 22]

New right row: [30, 26, 18, 5, 4, 1, 3, 7, 17, 16, 27, 25, 19, 29, 21, 13, 22]

No update!

Current W: 3288.2562499999945

Current sigma: [(26, 18, 8, 12, 13, 6, 10), (28,), (5, 4), (2, 1, 9, 0, 3, 11, 7, 23, 22), (17, 29, 15, 30), (16, 21, 14,

Current left row: [2, 11, 10, 31, 8, 6, 9, 0, 28, 20, 14, 24, 15, 12, 23]

Current right row: [30, 26, 18, 5, 4, 1, 3, 7, 17, 16, 27, 25, 19, 21, 29, 13, 22]

Iteration 207

3

[3, 7, 1]

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 2167121 rows, 652289 columns and 6183265 nonzeros

Model fingerprint: 0xe6ffd3fa

Model has 214016 simple general constraints

Start coding or [generate](#) with AI.

▼ SA results

```
end_time = datetime.now()
```

```
runtime = (end_time - start_time).total_seconds()
```

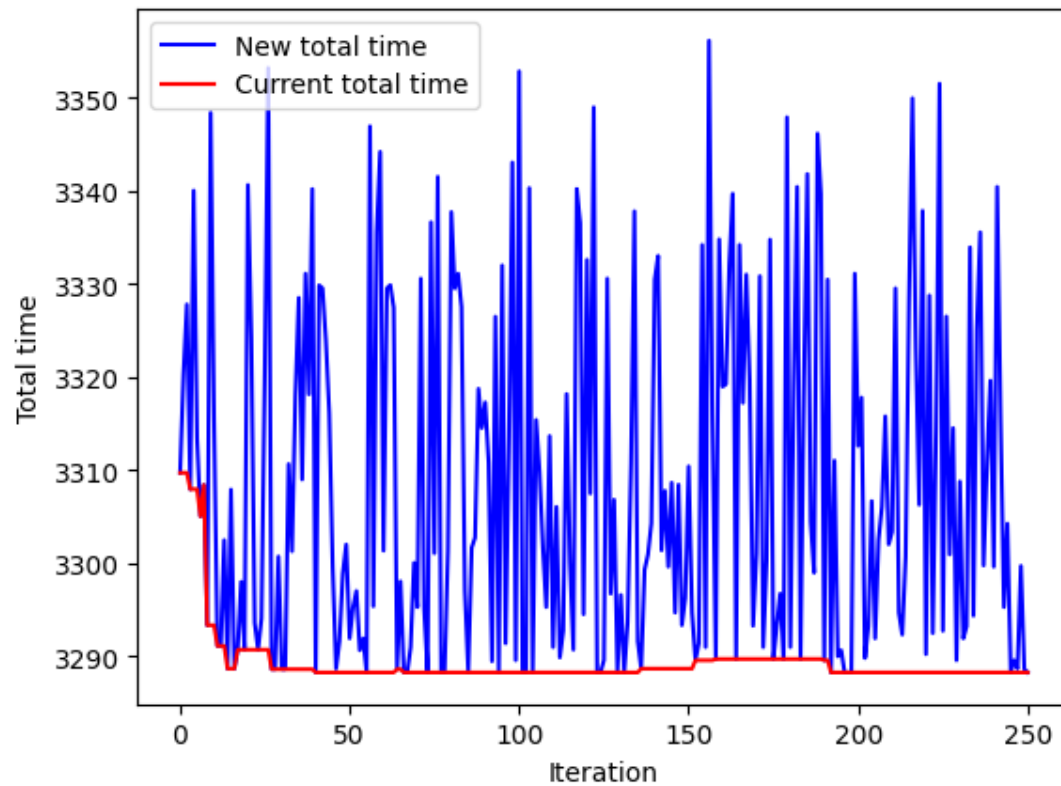


```

xs = [x for x in range(len(currentWs))]

plt.plot(xs, newWs, 'b')
plt.plot(xs, currentWs, 'r')
plt.legend(['New total time', 'Current total time'])
plt.xlabel('Iteration')
plt.ylabel('Total time')
plt.show()
# Make sure to close the plt object once done
plt.close()

```



Start coding or [generate](#) with AI.

