.. code:: ipython3

```
from pulp import *
import pandas as pd
import matplotlib.pyplot as plt
import random as rnd
import numpy as np
import datetime
import simpy
import math
import random
import xlsxwriter
import networkx as nx
import glob
import warnings
import glob, os
from ortools.linear_solver import pywraplp
from ortools.sat.python import cp_model
from matplotlib.backends.backend_pdf import PdfPages
warnings.filterwarnings("ignore")
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

.. code:: ipython3

```
def Data_Set(tt,Num_Ops):
    #Range of Operators
    operators = range(0, Num_Ops)
    #Skill Based Jobs
    for i in tt.index:
        tt.loc[i, 'SKILL'] =
random.choice(['ELECTRICAL','STRUCTURE','TEST'])
    #
    td = pd.get_dummies(tt['SKILL'])
    tt = pd.concat([tt,td],axis=1)
    OP = pd.DataFrame({'OPERATOR':operators})
    OP['ELECTRICAL'] = 0
    OP['STRUCTURE'] = 0
    OP['TEST'] = 0


    gen1 = 75
    spec = 50
    gen2 = gen1 #20

    #Find Required Skill Headcounts
    sh = tt.groupby(['SKILL'],as_index=False)['TIME'].sum()
```

```
        sh['HEADS'] = sh['TIME'] / 20.0 / (6.8)
        sh = sh.set_index('SKILL')
        sh = sh.div(sh.sum(axis=0), axis=1)
        sh['HEADS'] = sh['HEADS'].cumsum()
        sh = sh.reset_index()
        OP['LC_UNIT'] = 0
        tt['LC_Per'] = np.where(tt['SKILL'] == 'ELECTRICAL',0.93,0.95)
        tt['LC_Per'] = np.where(tt['SKILL'] ==
'TEST',0.86,tt['LC_Per'])

        for i in OP.index:
            unif_var = rnd.random()
            if unif_var <= sh.loc[0,'HEADS']:
                #ELECTRICAL LEARNING CURVE
                skl = sh.loc[0,'SKILL']
                OP.loc[i, skl] = 1
                OP.loc[i,'LC_UNIT'] = spec
            elif unif_var > sh.loc[1,'HEADS']:
                #TEST LEARNING CURVE
                skl = sh.loc[2,'SKILL']
                OP.loc[i, skl] = 1
                OP.loc[i,'LC_UNIT'] = spec
            else:
                #STRUCTURES LEARNING CURVE
                skl = sh.loc[1,'SKILL']
                OP.loc[i, skl] = 1
                OP.loc[i,'LC_UNIT'] = spec
            unif_var = rnd.random()
            if unif_var <= 0.5:
                skl = sh.loc[0,'SKILL']
                OP.loc[i, skl] = 1
                skl = sh.loc[1,'SKILL']
                OP.loc[i, skl] = 1
                skl = sh.loc[2,'SKILL']
                OP.loc[i, skl] = 1
                OP.loc[i,'LC_UNIT'] = gen2
            elif unif_var <= 0.0:
                skl = sh.loc[0,'SKILL']
                OP.loc[i, skl] = 1
                skl = sh.loc[1,'SKILL']
                OP.loc[i, skl] = 0
                skl = sh.loc[2,'SKILL']
                OP.loc[i, skl] = 1
                OP.loc[i,'LC_UNIT'] = gen1

        #Assign Experience to the Operators
        OP['SPECIALIST'] = np.where(OP['ELECTRICAL'] + OP['STRUCTURE']
+ OP['TEST'] == 1, 1,0)
        OP['ATTRITION'] = 0
        TW = OP.copy()
```

```
        tt['LC_Exp'] = np.log(tt['LC_Per']) / np.log(2)
        for t in tt.index:
            OP[tt['TASK'].iloc[t]] = tt['TIME'].iloc[t] /
(OP['LC_UNIT'] ** (tt['LC_Exp'].iloc[t]))
            TW[tt['TASK'].iloc[t]] = 0

        return(tt,OP,TW)

.. code:: ipython3

    def work_stations(tt,file):
        # Define the variables
        tasks = tt['TASK']
        WST = tt.copy()
        WST = WST[['TASK']]
        for t in tt.index:
            WST.loc[WST['TASK'] == t+1,tt['TASK'].iloc[t]] =
tt['TIME'].iloc[t]
        WST = WST.fillna(0)
        WST = WST.set_index('TASK')

        # Create a linear programming problem
        model = LpProblem(name="Determine_Workstations",
sense=LpMinimize)

        # Decision variables
        workstations = range(1, len(tasks) + 1)  # Number of
workstations
        x = LpVariable.dicts("Assign", (tasks, workstations),
cat='Binary')
        y = LpVariable.dicts("Assign", (workstations), cat='Binary')

        # Objective function: minimize the number of workstations
        model += lpSum(y[ws] * ws for ws in workstations)

        # Constraints: each task assigned to exactly one workstation
        for task in tasks:
            model += lpSum(x[task][ws] for ws in workstations) == 1

        # Constraints: precedence constraints
        for idx, row in pnet.iterrows():
            task, successor = row['TASK'], row['SUCCESSOR']
            for k in workstations:
                wksts = range(k, len(tasks) + 1)
                model += x[task][k] <= lpSum(x[successor][ws] for ws
in wksts)

        # Constraints: balance the workload across workstations
        for ws in workstations:
            model += lpSum(x[task][ws] * tt.loc[tt['TASK'] == task,
```

```
'TIME'].iloc[0] for task in tasks) <= CT * y[ws]

        # Solve the problem
        slvr=GUROBI(TimeLimit=36000,MIPGap=0.05,msg=1)
        model.solve(slvr)

        #Get the number of Workstations
        wkst = 0
        for ws in workstations:
            wkst = y[ws].value() + wkst
        wkst = int(wkst)
        return wkst

.. code:: ipython3

    def Balance_the_Line(tt,OP,wkst,type,WrmSt):
        # Define the variables
        tasks = tt['TASK']

        workstations = range(1,wkst+1)
        OP_A = OP[OP['ATTRITION'] == 0]

        operators = OP_A['OPERATOR'].unique()

        M = 100000

        #fractional = [0.2 ,0.25, 1/3, 0.4, 0.5, 0.6, 2/3, 0.75, 0.8,
1]
        fractional = [1]

        # Define the problem
        MILP = LpProblem("Assembly_Worker_Assignment_Balancing_MIPL",
LpMinimize)

        # Define the decision variables
        operator_task_assignment =
LpVariable.dicts("OperatorTaskAssign", (operators, tasks, fractional),
lowBound=0.0, upBound=1.0, cat='Continuous')

        # Additional binary decision variables
        task_workstation_assignment =
LpVariable.dicts("TaskWorkstationAssignment", (tasks, workstations),
cat='Binary')
        # Additional binary decision variables
        operator_workstation_assignment =
LpVariable.dicts("OperatorTaskAssignment", (operators, workstations),
cat='Binary')
        #Auxiliary Variable
        y = LpVariable.dicts("AuxiliaryVariable", (operators, tasks),
cat='Binary')
```

```
        #Set Warm Start Variables:
        if len(WrmSt) > 0:
            for oper in operators:
                for task in tasks:
                    for f in fractional:
                        try:
                            operator_task_assignment[oper][task]
[f].varValue  = WrmSt.loc[(WrmSt['Operator'] == oper) & (WrmSt['Task']
== task), 'Value'].iloc[0]
                        except:
                            operator_task_assignment[oper][task]
[f].varValue  = 0
            for task in tasks:
                for ws in workstations:
                    try:
                        if WrmSt.loc[(WrmSt['workstations'] == ws) &
(WrmSt['Task'] == task), 'Value'].iloc[0] > 0:
                            task_workstation_assignment[task]
[ws].varValue  = 1
                        else:
                            task_workstation_assignment[task]
[ws].varValue  = 0
                    except:
                        task_workstation_assignment[task][ws].varValue
= 0
            for oper in operators:
                for ws in workstations:
                    try:
                        if WrmSt.loc[(WrmSt['workstations'] == ws) &
(WrmSt['Operator'] == oper), 'Value'].iloc[0] > 0:
                            operator_workstation_assignment[oper]
[ws].varValue  = 1
                        else:
                            operator_workstation_assignment[oper]
[ws].varValue = 0
                    except:
                        operator_workstation_assignment[oper]
[ws].varValue = 0


        # Define the objective function
        MILP += lpSum(OP_A.loc[oper][task] *
(operator_task_assignment[oper][task][f] * f) + M *
operator_workstation_assignment[oper][ws] for oper in operators for
task in tasks for ws in workstations for f in fractional)


        # Constraints
```

```
        for task in tasks:
            MILP += lpSum(task_workstation_assignment[task][ws] for ws
in workstations) == 1

        for ws in workstations:
            MILP += lpSum(task_workstation_assignment[task][ws] for
task in tasks) >= 1

        for oper in operators:
            for f in fractional:
                MILP += lpSum(operator_task_assignment[oper][task][f]
for task in tasks) <= tot_tasks

        for task in tasks:
            MILP += lpSum(operator_task_assignment[oper][task][f] * f
for oper in operators for f in fractional) == 1



        for oper in operators:
            for task in tasks:
                for ws in workstations:
                    for f in fractional:
                        MILP += operator_task_assignment[oper][task]
[f] + task_workstation_assignment[task][ws] <= 1 + y[oper][task]
                        MILP += y[oper][task] <=
operator_workstation_assignment[oper][ws]

        # Constraints: precedence constraints
        for idx, row in pnet.iterrows():
            task, successor = row['TASK'], row['SUCCESSOR']
            for k in workstations:
                wksts = range(k, len(workstations) + 1)
                MILP += task_workstation_assignment[task][k] <=
lpSum(task_workstation_assignment[successor][ws] for ws in wksts)


        # Must be assigned to work they are skilled to perform
        for oper in operators:
            for task in tasks:
                MILP += lpSum(operator_task_assignment[oper][task][f]
* tt.loc[tt['TASK'] == task, 'ELECTRICAL'].iloc[0] for f in
fractional) <= OP_A.loc[OP_A['OPERATOR'] == oper,
'ELECTRICAL'].iloc[0]
                MILP += lpSum(operator_task_assignment[oper][task][f]
* tt.loc[tt['TASK'] == task, 'STRUCTURE'].iloc[0] for f in fractional)
<= OP_A.loc[OP_A['OPERATOR'] == oper, 'STRUCTURE'].iloc[0]
                MILP += lpSum(operator_task_assignment[oper][task][f]
* tt.loc[tt['TASK'] == task, 'TEST'].iloc[0] for f in fractional) <=
OP_A.loc[OP_A['OPERATOR'] == oper, 'TEST'].iloc[0]
```

```
        # Set a minimum time for each operator
        min_operator_time = tt['TIME'].sum() / (len(operators))
        max_operator_time = (20) * (10)
        for oper in operators:
            MILP += lpSum(tt.loc[tt['TASK'] == task,'TIME'] *
(operator_task_assignment[oper][task][f] * f) for task in tasks for f
in fractional) >= min_operator_time #OP.loc[oper][task]



        solver=GUROBI(MIPGap=0.01,msg=1)
        MILP.solve(solver)

        MILP.writeLP('MILP', writeSOS=1)
        if value(MILP.objective) is None:
            print("Infeasible")
            results_df = pd.DataFrame()
        else:
            results_df = pd.DataFrame(columns=['Operator', 'Task',
'Workstation','Value','Hours'])
            i = 0
            # Populate the DataFrame with the assigned values
            for oper in operators:
                for task in tasks:
                    for ws in workstations:
                        for f in fractional:
                            if operator_task_assignment[oper][task]
[f].value() > 0 and task_workstation_assignment[task][ws].value() ==
1:
                                vl = operator_task_assignment[oper]
[task][f].value() * f
                                hrs = operator_task_assignment[oper]
[task][f].value() * OP_A.loc[oper][task] * f
                                d = {"Operator": [oper], "Task":
[task], "Workstation":[ws],"Value":[vl],"Hours":[hrs]}
                                res_df = pd.DataFrame(data=d)
                                if i == 0:
                                    results_df = res_df.copy()
                                    i = i + 1
                                else:
                                    results_df =
pd.concat([results_df,res_df],ignore_index=False)


        return results_df
```

```
.. code:: ipython3

    def Simulate(results_df,OP,TW,wkst,tt,type):
        k = 0
        OP_HOURS['MEAN'] = OP_HOURS['Hours']
        OP_HOURS['SIGMA_SQ'] = 0
        BF_HOURS = OP_HOURS.copy()
        BF_OP = OP.copy()
        BF_TW = TW.copy()
        backfill_df = results_df.copy()
        perf =
pd.DataFrame(columns=['OPERATOR','TASK','WORKSTATION','SKILL','VALUE',
'NONCONF','ITERATION','IT_HRS','HRS','SPECIALIST','GENERALIST','TIME']
)
        bf_perf =
pd.DataFrame(columns=['OPERATOR','TASK','WORKSTATION','SKILL','VALUE',
'NONCONF','ITERATION','IT_HRS','HRS','SPECIALIST','GENERALIST','TIME']
)
        for i in range(0,100):
            OP_HOURS['T_' + str(i)] = 0
            BF_HOURS['T_' + str(i)] = 0
            attr = 0
            for task in results_df['Task'].unique():
                OP_Task = results_df[results_df['Task'] == task]
                BF_Task = backfill_df[backfill_df['Task'] == task]
                for op in OP_Task['Operator'].unique():
                    OP_Inf = OP_Task[OP_Task['Operator'] == op]
                    ws = results_df.loc[results_df['Task'] == task,
'Workstation'].iloc[0]
                    vlu = results_df.loc[(results_df['Task'] == task)
& (results_df['Operator'] == op), 'Value'].iloc[0]
                    #Nonconformance Generator
                    BL_Task = tt.loc[tt['TASK'] == task,'TIME'].item()
                    TASK_NC = BL_Task / TOT_hrs
                    units = TW.loc[TW['OPERATOR'] == op,task].item()
                    lc = tt.loc[tt['TASK'] == task,'LC_Exp'].item()
                    if units == 0:
                        hrs = (OP_Inf['Hours'].iloc[0] * ((units+1) **
lc)) #OP_Inf['LC_Per'].iloc[0])) #/ (i + 1)
                    else:
                        T1 = OP_Inf['Hours'].iloc[0] / ((units) ** lc)
                        hrs = T1 * ((units+1) **  lc)

                    sd = 0.25
                    if OP.loc[OP['OPERATOR'] ==
op,'SPECIALIST'].iloc[0] == 1:
                        TASK_NC = TASK_NC + 0.05
                        sd = 0.5
                    try:
                        sph = vlu * SPEC.loc[SPEC['TASK'] == task,
```

```
'SPECIALIST_' + str(i)].iloc[0]
                    except:
                        sph = vlu * SPEC.loc[SPEC['TASK'] == task,
'TIME'].iloc[0]
                    try:
                        gnh = vlu * GEN.loc[GEN['TASK'] == task,
'GENERALIST_' + str(i)].iloc[0]
                        tm = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                    except:
                        gnh = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                        tm = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                    hrs = max(hrs,tm)
                    nc_rv = random.uniform(0,1)
                    if nc_rv < TASK_NC:
                        NC = 1
                        OP_HOURS['NC'] = np.where((OP_HOURS['Task'] ==
task) & (OP_HOURS['Operator'] == op), OP_HOURS['NC'] + 1,
OP_HOURS['NC'])
                    else:
                        NC = 0
                    task_hrs = np.random.normal(loc=hrs, scale=sd) +
NC * hrs * TASK_NC

                    results_df['Hours'] = np.where((results_df['Task']
== task) & (results_df['Operator'] == op),hrs,results_df['Hours'])
#task_hrs

                    OP_HOURS['T_' + str(i)] =
np.where((OP_HOURS['Task'] == task) & (OP_HOURS['Operator'] == op),
                                                        hrs,
OP_HOURS['T_' + str(i)]) #task_hrs

                    skl = GEN.loc[GEN['TASK'] == task,
'SKILL'].iloc[0]
                    #Updaate how many times the operator has performed
the task
                    TW[task] = np.where(TW['OPERATOR'] == op,TW[task]
+1,TW[task])
                    #Update the expected hours for the operator to
perform the task
                    OP[task] = np.where(OP['OPERATOR'] == op,task_hrs
* (1 / vlu),OP[task])
                    #New DataFrame
                    perf.loc[-1] =
[op,task,ws,skl,vlu,NC,i,task_hrs,hrs,sph,gnh,tm]
                    perf.index = perf.index + 1
```

```
                #BackFill Simulation
                for op in BF_Task['Operator'].unique():
                    OP_Inf = BF_Task[BF_Task['Operator'] == op]
                    ws = backfill_df.loc[backfill_df['Task'] == task,
'Workstation'].iloc[0]
                    vlu = backfill_df.loc[(backfill_df['Task'] ==
task) & (backfill_df['Operator'] == op), 'Value'].iloc[0]
                    #Nonconformance Generator
                    BL_Task = tt.loc[tt['TASK'] == task,'TIME'].item()
                    TASK_NC = BL_Task / TOT_hrs
                    units = BF_TW.loc[BF_TW['OPERATOR'] ==
op,task].item()
                    lc = tt.loc[tt['TASK'] == task,'LC_Exp'].item()
                    if units == 0:
                        hrs = (OP_Inf['Hours'].iloc[0] * ((units+1) **
lc)) #OP_Inf['LC_Per'].iloc[0])) #/ (i + 1)
                    else:
                        T1 = OP_Inf['Hours'].iloc[0] / ((units) ** lc)
                        hrs = T1 * ((units+1) **  lc)

                    sd = 0.25
                    if BF_OP.loc[BF_OP['OPERATOR'] ==
op,'SPECIALIST'].iloc[0] == 1:
                        TASK_NC = TASK_NC + 0.05
                        sd = 0.75
                    try:
                        sph = vlu * SPEC.loc[SPEC['TASK'] == task,
'SPECIALIST_' + str(i)].iloc[0]
                    except:
                        sph = vlu * SPEC.loc[SPEC['TASK'] == task,
'TIME'].iloc[0]
                    try:
                        gnh = vlu * GEN.loc[GEN['TASK'] == task,
'GENERALIST_' + str(i)].iloc[0]
                        tm = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                    except:
                        gnh = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                        tm = vlu * GEN.loc[GEN['TASK'] == task,
'TIME'].iloc[0]
                    hrs = max(hrs,tm)
                    '''nc_rv = random.uniform(0,1)
                    if nc_rv < TASK_NC:
                        NC = 1
                        BF_HOURS['NC'] = np.where((BF_HOURS['Task'] ==
task) & (BF_HOURS['Operator'] == op), BF_HOURS['NC'] + 1,
BF_HOURS['NC'])
                    else:
                        NC = 0'''
```

```
                    if perf.loc[perf['TASK'] == task, 'NONCONF'].sum()
> 0:
                        NC = 1
                        BF_HOURS['NC'] = np.where((BF_HOURS['Task'] ==
task) & (BF_HOURS['Operator'] == op), BF_HOURS['NC'] + 1,
BF_HOURS['NC'])
                    else:
                        NC = 0

                    task_hrs = np.random.normal(loc=hrs, scale=sd) +
NC * hrs * TASK_NC

                    backfill_df['Hours'] =
np.where((backfill_df['Task'] == task) & (backfill_df['Operator'] ==
op),hrs,backfill_df['Hours']) #task_hrs

                    BF_HOURS['T_' + str(i)] =
np.where((BF_HOURS['Task'] == task) & (BF_HOURS['Operator'] == op),
                                                        task_hrs,
BF_HOURS['T_' + str(i)])

                    skl = GEN.loc[GEN['TASK'] == task,
'SKILL'].iloc[0]
                    #Updaate how many times the operator has performed
the task
                    BF_TW[task] = np.where(BF_TW['OPERATOR'] ==
op,BF_TW[task]+1,BF_TW[task])
                    #Update the expected hours for the operator to
perform the task
                    BF_OP[task] = np.where(BF_OP['OPERATOR'] ==
op,task_hrs * (1 / vlu),BF_OP[task])
                    #New DataFrame
                    bf_perf.loc[-1] =
[op,task,ws,skl,vlu,NC,i,task_hrs,hrs,sph,gnh,tm]
                    bf_perf.index = bf_perf.index + 1


                OP_Task = OP_HOURS[OP_HOURS['Task'] == task]
                OP_Task = OP_Task.groupby(['Task'],as_index=False)
[['Hours','T_'+str(i)]].sum()


            if type == 'No_Attr':
                #Cae 0 Arrtition = 0
                attr = 0
            elif type == 'Poisson3%':
                #Case 2 Attrition - Model the attrition as a RV where
the mean is 3% - poisson distribution
                attr = (len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()) * 0.03)
```

```
                attr = np.random.poisson(lam=attr)
            elif type == 'Poisson10%':
                #Personnel % Turnover
                attr = (len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()) * 0.10)
                attr = np.random.poisson(lam=attr)
            elif type == 'Poisson12%':
                #Personnel % Turnover
                attr = (len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()) * 0.12)
                attr = np.random.poisson(lam=attr)
            elif type == 'Poisson7.5%':
                #Case 3 Attition - Model higher attrition rates 7.5% &
10%
                attr = (len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()) * 0.075)
                attr = np.random.poisson(lam=attr)
            if attr > len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()):
                attr = len(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique()) - 1
            op_att = []
            new_oper = []
            OP_new =
pd.DataFrame(columns=['OPERATOR','ELECTRICAL','STRUCTURE','TEST','LC_U
NIT','SPECIALIST','ATTRITION'])
            if attr == 0:
                pass
            elif attr == 1:
                ro = random.choice(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique())
                op_att.append(ro)
                OP['ATTRITION'] = np.where(OP['OPERATOR'] == ro, 1,
OP['ATTRITION'])
                BF_OP['ATTRITION'] = np.where(BF_OP['OPERATOR'] == ro,
1, BF_OP['ATTRITION'])
                newop = OP['OPERATOR'].max() + 1
                new_oper.append(newop)
                rv = random.uniform(0,1)
                if OP.loc[OP['OPERATOR'] == ro,'SPECIALIST'].item() ==
1:
                    if rv > 1: #0.9:
                        ele = OP.loc[OP['OPERATOR'] ==
ro,'ELECTRICAL']
                        strt = OP.loc[OP['OPERATOR'] ==
ro,'STRUCTURE']
                        tst = OP.loc[OP['OPERATOR'] == ro,'TEST']
                        lcu = 50
                        sp = 1
                        arrtr = 0
```

```
                    else:
                        ele = 1
                        strt = 1
                        tst = 1
                        lcu = 75
                        sp = 0
                        arrtr = 0
                else:
                    ele = 1
                    strt = 1
                    tst = 1
                    lcu = 75
                    sp = 0
                    arrtr = 0
                OP_new.loc[-1] = [newop,ele,strt,tst,lcu,sp,arrtr]
                OP_new.index = OP_new.index + 1
                #OP_new = OP_new.sort_index()
                TW_new = OP_new.copy()
            else:
                for an in range(0,attr):
                    ro = random.choice(OP.loc[OP['ATTRITION'] ==
0,'OPERATOR'].unique())
                    op_att.append(ro)
                    OP['ATTRITION'] = np.where(OP['OPERATOR'] == ro,
1, OP['ATTRITION'])
                    BF_OP['ATTRITION'] = np.where(BF_OP['OPERATOR'] ==
ro, 1, BF_OP['ATTRITION'])
                    newop = OP['OPERATOR'].max() + 1 + an
                    new_oper.append(newop)
                    rv = random.uniform(0,1)
                    if OP.loc[OP['OPERATOR'] ==
ro,'SPECIALIST'].item() == 1:
                        if rv > 1: #0.9:
                            ele = OP.loc[OP['OPERATOR'] ==
ro,'ELECTRICAL']
                            strt = OP.loc[OP['OPERATOR'] ==
ro,'STRUCTURE']
                            tst = OP.loc[OP['OPERATOR'] == ro,'TEST']
                            lcu = 50
                            sp = 1
                            arrtr = 0
                        else:
                            ele = 1
                            strt = 1
                            tst = 1
                            lcu = 75
                            sp = 0
                            arrtr = 0
                    else:
                        ele = 1
```

```
                        strt = 1
                        tst = 1
                        lcu = 75
                        sp = 0
                        arrtr = 0
                    OP_new.loc[-1] = [newop,ele,strt,tst,lcu,sp,arrtr]
                    OP_new.index = OP_new.index + 1
                    #OP_new = OP_new.sort_index()
                    TW_new = OP_new.copy()
            if attr > 0:
                for t in tt.index:
                    OP_new[tt['TASK'].iloc[t]] = tt['TIME'].iloc[t] /
(OP_new['LC_UNIT'] ** (tt['LC_Exp'].iloc[t]))
                    TW_new[tt['TASK'].iloc[t]] = 0

                OP = pd.concat([OP,OP_new],ignore_index=True)
                BF_OP = pd.concat([BF_OP,OP_new],ignore_index=True)
                TW = pd.concat([TW,TW_new],ignore_index=True)
                BF_TW = pd.concat([BF_TW,TW_new],ignore_index=True)
                WrmSt = 1

                for idx, x in enumerate(op_att):
                    backfill_df['Operator'] =
np.where(backfill_df['Operator'] == x, new_oper[idx],
backfill_df['Operator'])
                    #reset backfill_df hours for these tasks as well
                    bfch = backfill_df[backfill_df['Operator'] ==
new_oper[idx]]
                    bfch = bfch.reset_index(drop=True)
                    for op_as in bfch['Task'].unique():
                        hrs = BF_OP.loc[BF_OP['OPERATOR'] ==
new_oper[idx], op_as].item()
                        backfill_df['Hours'] =
np.where((backfill_df['Operator'] == new_oper[idx]) &
(backfill_df['Task'] == op_as),
                                                            hrs *
backfill_df['Value'], backfill_df['Hours']) #
                WrmSt = backfill_df.copy()
                rebal_df = Balance_the_Line(tt,OP,wkst,type,WrmSt)
                if len(rebal_df) == 0:
                    for idx, x in enumerate(op_att):
                        results_df['Operator'] =
np.where(results_df['Operator'] == x, new_oper[idx],
results_df['Operator'])
                        #reset backfill_df hours for these tasks as
well
                        bfch = results_df[results_df['Operator'] ==
new_oper[idx]]
                        bfch = bfch.reset_index(drop=True)
                        for op_as in bfch['Task'].unique():
```

```
                            hrs = OP.loc[OP['OPERATOR'] ==
new_oper[idx], op_as].item()
                            results_df['Hours'] =
np.where((results_df['Operator'] == new_oper[idx]) &
(results_df['Task'] == op_as),
                                                                hrs *
results_df['Value'], results_df['Hours']) #
                else:
                    results_df = rebal_df.copy()


        return perf,TW,OP,bf_perf,BF_TW


.. code:: ipython3


    os.chdir(r"/Users/Mailman82/Documents/GWU/SALBP-data-sets/small
data set_n=20/")
    dst = "Details of the small data set (n=20).xlsx"

    i = 0
    n = 0
    no = 0
    fls = 1
    excs = 1
    ds_inf = []
    pp = PdfPages('n20_Output_Tasks.pdf')
    FLD = 'n20_Output_TB.xlsx'
    FLD2 = 'n20_Output_OP_TB.xlsx'


    writer = pd.ExcelWriter(FLD)
    writer2 = pd.ExcelWriter(FLD2)
    for file in sorted(glob.glob("*.txt")):
        CT = 0
        OS = 0
        numtask = 0
        CT = 0
        OS = 0
        TA = 0
        tsk = []
        tst = []
        prd = []
        ssr = []
        PR = 0

        if fls == 51:
            excs = excs + 1
            fls = 1
```

```
            writer.close()


        SN = file.replace('.txt','')



        with open(file) as input_data:

            # Skips text before the beginning of the interesting
block:
            for line in input_data:
                if line.strip() == '<number of tasks>':  # Or whatever
test is needed
                    numtask = 1
                elif numtask == 1:
                    numtask = int(line.strip())

                if line.strip() == '<cycle time>':  # Or whatever test
is needed
                    CT = 1
                elif CT == 1:
                    CT = float(line.strip())

                if line.strip() == '<order strength>':  # Or whatever
test is needed
                    OS = 1
                elif OS == 1:
                    try:
                        OS = float(line.strip())
                    except:
                        p = line.strip().split(",")
                        OS = int(p[1])/1000

                if line.strip() == '<task times>':  # Or whatever test
is needed
                    TA = 1
                    ind = 0
                elif TA == 1:
                    if line.strip() == "":
                        TA = 2
                    else:
                        tsk.append(int(line.strip().split(' ',1)[0]))
                        tst.append(int(line.strip().split(' ',1)[1]))

                if line.strip() == '<precedence relations>':  # Or
whatever test is needed
                    PR = 1
                    ind = 0
```

```
                elif PR == 1:
                    if line.strip() == "":
                        PR = 2
                    else:
                        prd.append(int(line.strip().split(',',1)[0]))
                        ssr.append(int(line.strip().split(',',1)[1]))
            tt = pd.DataFrame({'TASK':tsk,'TIME':tst})
            pnet = pd.DataFrame({'TASK':prd,'SUCCESSOR':ssr})
            #Total Hours
            TOT_hrs = (tt['TIME'].sum())
            #Total Operators required for 20 day takt time
            Num_Ops = math.ceil(tt['TIME'].sum() / (20) / (6.8))
            #Total Hours per 4 Work Stations
            WS_hrs = (tt['TIME'].sum() / (20 / 2.0))
            tot_tasks = np.ceil(20/Num_Ops)

            if Num_Ops/numtask < 5:
                n = n + 1
            else:
                i = i + 1
            if no < Num_Ops/numtask:
                no = Num_Ops/numtask
            rbl = 1

            tt,OP1,TW1 = Data_Set(tt,Num_Ops)
            wkst = work_stations(tt,file)

            for type in ['No_Attr','Poisson3%','Poisson7.5%']:
                OP = OP1.copy()
                TW = TW1.copy()
                print(type)
                WrmSt = pd.DataFrame()
                res_df = Balance_the_Line(tt,OP,wkst,type,WrmSt)

                res = res_df.groupby(['Operator'],as_index=False)
[['Value','Hours']].sum()
                res2 = res_df.groupby(['Operator'],as_index=False)
[['Workstation']].nunique()
                res3 = res_df.groupby(['Task'],as_index=False)
[['Workstation']].nunique()
                WS_HOURS =
res_df.groupby(['Workstation'],as_index=False)[['Hours']].sum()
                TASK_HOURS = res_df.groupby(['Task'],as_index=False)
[['Hours']].sum()


                OP_SKILL = OP[['OPERATOR','LC_UNIT','SPECIALIST']]
                OP_HOURS =
pd.merge(res_df,OP_SKILL,how='left',left_on='Operator',right_on='OPERA
TOR')
```

```
                OP_HOURS['NC'] = 0
                OP_HOURS.head()
                SNM = SN + "_" + type
                SNM2 = SNM + "_B"
                gen = 75 + 1
                GEN = tt.copy()
                GEN['GENERALIST_0'] = GEN['TIME'] / ((gen)  **
(GEN['LC_Exp']))
                spec = 50 + 1
                SPEC = tt.copy()
                SPEC['SPECIALIST_0'] = SPEC['TIME'] / ((spec)  **
(SPEC['LC_Exp']))
                for i in range(1,gen-1):
                    GEN['GENERALIST_' + str(i)] = GEN['GENERALIST_0']
* ((i+1)  ** (GEN['LC_Exp']))
                    if i <= spec-1:
                        SPEC['SPECIALIST_' + str(i)] =
SPEC['SPECIALIST_0'] * ((i+1) ** (SPEC['LC_Exp']))

                res,TW,OP,bf_res,TWB =
Simulate(res_df,OP,TW,wkst,tt,type)
                res.to_excel(writer,sheet_name=SNM, index=False)
                OP.to_excel(writer2,sheet_name=SNM, index=False)
                res['OPTIMIZATION_HRS'] = res['IT_HRS']
                rest = res.groupby(['ITERATION'],as_index=False)
[['OPTIMIZATION_HRS','SPECIALIST','GENERALIST','TIME']].sum()
                if type == 'No_Attr':
                    pass
                else:
                    bf_res.to_excel(writer,sheet_name=SNM2,
index=False)
                    bf_res['BACKILL_SIM_HRS'] = bf_res['IT_HRS']
                    bfrest =
bf_res.groupby(['ITERATION'],as_index=False)
[['BACKILL_SIM_HRS']].sum()
                    rest =
pd.merge(rest,bfrest,how='left',on='ITERATION')
                rest = rest.set_index('ITERATION')
                rest.plot()
                plt.title(SNM)
                plt.tight_layout()
                pp.savefig()
                plt.close()
                if type == 'Poisson7.5%':
                    wlp = 1

    pp.close()
    writer.close()
    writer2.close()
    plt.close()
```