Hönerloh, Hagen Alexander

Article

Numerical studies for the scheduling of continuous annealing lines

Junior Management Science (JUMS)

Provided in Cooperation with:
Junior Management Science e. V.

Suggested Citation: Hönerloh, Hagen Alexander (2025) : Numerical studies for the scheduling of continuous annealing lines, Junior Management Science (JUMS), ISSN 2942-1861, Junior Management Science e. V., Planegg, Vol. 10, Iss. 3, pp. 781-809, https://doi.org/10.5282/jums/v10i3pp781-809

This Version is available at:
https://hdl.handle.net/10419/326973

Online-Appendix

„Numerical Studies for the Scheduling of Continuous Annealing Lines“

Hagen Alexander Hönerloh
Leibniz University Hannover

Junior Management Science 10(3) (2025) 781-809

A Appendix

A.1 Code - Model

```
# Import of necessary package
from gurobipy import *

# Definition of model and utilized parameters from the instance
def CAL_model(N, N_coil, N_full, K_range, d, Modes_range, modes, alpha,
        M, c, t, p):
    model = Model()

    # Decision variables.
    y = model.addVars(N_full, N_full, K_range, Modes_range,
        Modes_range, vtype=GRB.BINARY, name="y")
    x = model.addVars(N_full, K_range, Modes_range,
        vtype=GRB.BINARY, name="x")
    delta = model.addVars(N_full, N_full, K_range,
        vtype=GRB.BINARY, name="delta")
    z = model.addVars(N_full, vtype=GRB.BINARY, name="z")
    s = model.addVars(N_full, vtype=GRB.CONTINUOUS, lb=0, name="s")

    # Constraint 1 - 12. Due to the definition of the decision
    # variables, constraint 13 from the mathematical model
    # is not necessary.
    # Time consistency constraints.
    model.addConstrs((s[i] + quicksum(p[i][k][m] * x[i, k, m]
        for k in K_range for m in modes[i][k])
        <= d[i] + z[i] * M for i in N_full), "1")
    model.addConstrs((s[i] + quicksum((p[i][k][m] + t[i][j][k][m][n])
        * y[i, j, k, m, n] for k in K_range
        for m in modes[i][k] for n in modes[j][k])
        <= s[j] + M * (1 - quicksum(delta[i, j, k] for k in K_range))
        for i in N_full for j in N_full), "2")

    # Decision variable consistency constraints.
    model.addConstrs((delta[i, j, k] == quicksum(y[i, j, k, m, n]
        for m in modes[i][k] for n in modes[j][k])
        for i in N_full for j in N_full[:-1] for k in K_range), "3")
    model.addConstrs((quicksum(y[i, j, k, m, n] for j in N_full
        for n in modes[j][k]) <= x[i, k, m]
        for i in N_full for k in K_range for m in modes[i][k]), "4")
    model.addConstrs((quicksum(y[i, j, k, m, n] for i in N_full
        for m in modes[i][k]) <= x[j, k, n]
        for j in N_full for k in K_range for n in modes[j][k]), "5")

    # Sequence consistency constraints.
    model.addConstrs((quicksum(delta[i, j, k] for k in K_range
        for j in N_full[1:]) == 1 for i in N_coil), "6")
    model.addConstrs((quicksum(delta[i, j, k] for k in K_range
        for i in N_full[:-1]) == 1 for j in N_coil), "7")
    model.addConstrs((quicksum(delta[0, j, k] for j in N_full[1:])
        == 1 for k in K_range), "8")
    model.addConstrs((quicksum(delta[i, N_full[-1], k]
        for i in N_full[:-1]) == 1 for k in K_range), "9")
```

```
    model.addConstrs((quicksum(delta[j, i, k]
        for j in N_full[:-1]) == quicksum(delta[i, j, k]
        for j in N_full[1:]) for i in N_coil for k in K_range), "10")

    # Additional constraints.
    model.addConstr((quicksum(z[i] for i in N_coil) <= alpha), "11")
    model.addConstrs((s[i] >= 0 for i in N_coil), "12")

    # This constraint prevents the solver to set every x to one
    # and limits x for coil i and all lines and modes to one.
    model.addConstrs((quicksum(x[i, k, m] for k in K_range
        for m in Modes_range) == 1 for i in N_coil), "13")

    # Creation of a parameter that includes all stringers
    # introduced in the schedule.
    obj = quicksum(c[i][j][k][m][n] * y[i, j, k, m, n]
        for i in N_coil for j in N_coil for k in K_range
        for m in modes[i][k] for n in modes[j][k])

    # Setting the objective to minimize the above obj.
    model.setObjective(obj, GRB.MINIMIZE)

    # Setting up a time limit.
    model.setParam("TimeLimit", 720)

    # Setting tuning parameters to increase performance.
    model.Params.Cuts = 1
    model.Params.PrePasses = 1
    model.Params.Presolve = 2
    if 40 >= N >= 30:
        model.Params.heuristics = 1
        model.Params.PrePasses = -1
    elif 60 > N >= 50:
        model.Params.heuristics = 1
        model.Params.PrePasses = -1
        model.Params.MIPFocus = 1
    elif N == 60:
        model.Params.heuristics = 1
        model.Params.PrePasses = -1
        model.Params.Cuts = 2
        model.Params.MIPFocus = 1
    elif N >= 70:
        model.Params.Presolve = 1
        model.Params.heuristics = 1
        model.Params.PrePasses = -1
        model.Params.Cuts = 2
        model.Params.MIPFocus = 1
        model.setParam(GRB.Param.NodefileStart, 0.5)
        model.setParam("Threads", 8)

    # Returns the solution of the optimization.
    return model
```

A.2 Code - Instance

```
# Import of necessary packages
from gurobipy import *
import random
import time
import CAL_model

# Creation of function to study several instances at once
def com_study():

    # Creation of output file
    sol = open("Output_file.txt", "a")

    # Input of different instance sizes N to study.
    for i in []:
        N = i

        # Input of different service limits alpha to study.
        for a in []:
            alpha = a * N

            # Insertion of a paragraph in the output file for better
            # visualization.
            sol.write("\n")

            # Input of different seeds to study.
            # For this study, seeds 12, 452, 535, 123, 363, 25, 50,
            # 75, 100 and 125 were chosen.
            for seed in []:
                random.seed(seed)

                # Information about the instance that is currently
                # being optimized.
                print(f"Instance size = {i}, alpha = {a},
                    seed = {seed}")

                # Beginning of time measurement for total time.
                start = time.time()

                # Creation of two different N: N_full incorporates
                # virtual coils, N_coil does not.
                N_coil = range(N + 2)[1:-1]
                N_full = range(N + 2)

                # Setup of the specific levels for each factor. In this
                # case, the Basecase scenario is chosen.
                levels = ["Very low", "Low", "Basecase",
                    "High", "Very high"]
                heterogeneity = levels[2]
                urgency = levels[2]
                process_flexibility = levels[2]
                str_pro = levels[2]

                # All parameters that are determined by the different
                # levels of factors.
```

```
                Par_dict = {
                    "a": [N * 2, N * 1.5, N],
                    "min_w": [70, 60, 60, 60, 60],
                    "max_w": [90, 100, 100, 100, 100],
                    "min_th": [0.15, 0.1, 0.1, 0.1, 0.1],
                    "max_th": [0.35, 0.499, 0.499, 0.499, 0.499],
                    "min_len": [15000, 14000, 14000, 14000, 14000],
                    "max_len": [16000, 17000, 17000, 17000, 17000],
                    "min_temp": [690, 680, 680, 680, 680],
                    "max_temp": [720, 730, 730, 730, 730],
                    "min_speed": [520, 510, 510, 510, 510],
                    "max_speed": [580, 590, 590, 590, 590],
                    "pf_w": [5, 5, 5, 10, 15],
                    "pf_th": [0.05, 0.1, 0.1, 0.1, 0.2],
                    "pf_temp": [0, 0, 10, 10, 20],
                    "pf_speed": [0, 20, 20, 20, 20],
                    "str_time": [0, 1.5, 3, 10, 15]
                }

                # Setup of index for the chosen level of PF.
                if process_flexibility == "Very low":
                    pf = 0
                elif process_flexibility == "Low":
                    pf = 1
                elif process_flexibility == "Basecase":
                    pf = 2
                elif process_flexibility == "High":
                    pf = 3
                else:
                    pf = 4

                # Setup of index for the chosen level of SPT.
                if str_pro == "Very low":
                    pr = 0
                elif str_pro == "Low":
                    pr = 1
                elif str_pro == "Basecase":
                    pr = 2
                elif str_pro == "High":
                    pr = 3
                else:
                    pr = 4

                # Setup of index for the chosen level of UC.
                if urgency == "Very low":
                    u = 0
                elif urgency == "Low":
                    u = 1
                else:
                    u = 2

                # Setup of index for the chosen level of HC and
                # distribution of coil characteristics.
                if heterogeneity == "Very low":
                    h = 0
                elif heterogeneity == "Low":
```

```
            h = 1
        elif heterogeneity == "Basecase":
            h = 2
            char_coils = []
            for i in N_full:
                p = random.random()
                if p <= 0.5:
                    char_coils.append(
                        [random.randrange(
                            Par_dict["min_w"][h],
                            Par_dict["max_w"][h]),
                        round(random.uniform(
                            Par_dict["min_th"][h],
                            Par_dict["max_th"][h]), 3),
                        random.randrange(
                            Par_dict["min_temp"][h],
                            Par_dict["max_temp"][h], 10),
                        random.randrange(
                            Par_dict["min_speed"][h],
                            Par_dict["max_speed"][h], 10),
                        random.randrange(
                            Par_dict["min_len"][h],
                            Par_dict["max_len"][h])])
                else:
                    char_coils.append([random.triangular(
                            Par_dict["min_w"][h],
                            Par_dict["max_w"][h],
                            (Par_dict["max_w"][h] +
                            Par_dict["min_w"][h]) / 2),
                        round(random.triangular(
                            Par_dict["min_th"][h],
                            Par_dict["max_th"][h],
                            (Par_dict["max_th"][h] +
                            Par_dict["min_th"][h]) / 2), 3),
                        round(random.triangular(
                            Par_dict["min_temp"][h] / 10,
                            Par_dict["max_temp"][h] / 10,
                            (Par_dict["max_temp"][h] +
                            Par_dict["min_temp"][h]) / 20))
                            * 10,
                        round(random.triangular(
                            Par_dict["min_speed"][h] / 10,
                            Par_dict["max_speed"][h] / 10,
                            (Par_dict["max_speed"][h] +
                            Par_dict["min_speed"][h]) / 20))
                            * 10,
                        round(random.triangular(
                            Par_dict["min_len"][h],
                            Par_dict["max_len"][h],
                            (Par_dict["max_len"][h] +
                            Par_dict["min_len"][h]) / 2))])
        elif heterogeneity == "High":
            h = 3
        else:
            h = 4

```

```
# To save space, HC levels with the same distribution
# of coil characteristics are combined.
if (heterogeneity == "Very low")
    or (heterogeneity == "Low"):
    char_coils = [[random.triangular(
            Par_dict["min_w"][h],
            Par_dict["max_w"][h],
            (Par_dict["max_w"][h] +
            Par_dict["min_w"][h]) / 2),
        round(random.triangular(Par_dict["min_th"][h],
            Par_dict["max_th"][h],
            (Par_dict["max_th"][h] +
            Par_dict["min_th"][h]) / 2), 3),
        round(random.triangular(
            Par_dict["min_temp"][h] / 10,
            Par_dict["max_temp"][h] / 10,
            (Par_dict["max_temp"][h] +
            Par_dict["min_temp"][h]) / 20)) * 10,
        round(random.triangular(
            Par_dict["min_speed"][h] / 10,
            Par_dict["max_speed"][h] / 10,
            (Par_dict["max_speed"][h] +
            Par_dict["min_speed"][h]) / 20)) * 10,
        round(random.triangular(
        Par_dict["min_len"][h],
            Par_dict["max_len"][h],
            (Par_dict["max_len"][h] +
            Par_dict["min_len"][h]) / 2))]
        for i in N_full]
elif (heterogeneity == "High")
    or (heterogeneity == "Very high"):
    char_coils = [[random.randrange(
            Par_dict["min_w"][h],
            Par_dict["max_w"][h]),
        round(random.uniform(
            Par_dict["min_th"][h],
            Par_dict["max_th"][h]), 3),
        random.randrange(
            Par_dict["min_temp"][h],
            Par_dict["max_temp"][h], 10),
        random.randrange(
            Par_dict["min_speed"][h],
            Par_dict["max_speed"][h], 10),
        random.randrange(
            Par_dict["min_len"][h],
            Par_dict["max_len"][h])]
        for i in N_full]

# Parameterization of the heterogeneity
# of the processing lines k.
K = [0, (Par_dict["min_w"][h] +
        (Par_dict["max_w"][h] -
        Par_dict["min_w"][h]) / 2 - 5),
    (Par_dict["min_w"][h] +
        (Par_dict["max_w"][h] -
        Par_dict["min_w"][h]) / 2 + 5)]
```

```
K_range = range(len(K))

# Determination of all process modes.
Modes = ()
for m_t in range(Par_dict["min_temp"][h] - 10,
    Par_dict["max_temp"][h]+
    11, 10):
    Modes_ = ((m_t, m_s) for m_s in range(
        Par_dict["min_speed"][h] - 10,
        Par_dict["max_speed"][h] + 11, 20))
    Modes += tuple(Modes_)
Modes_range = range(len(Modes))

# Determination of feasible processing modes
# for coil i on processing line k.
modes = ()
for i in N_full:
    k_modes = ()
    for index_k, k in enumerate(K):
        m_comp = ()

        # If coil i is not compatible with line k,
        # it has no feasible processing modes
        # on this line.
        if (index_k == 0) or (index_k == 1 and
            k <= char_coils[i][0]) or (
                index_k == 2 and k >= char_coils[i][0]):
            for index_m, m in enumerate(Modes):
                if heterogeneity != "Very high":
                    if (abs(char_coils[i][2] - m[0])
                        <= 10) and
                        (abs(char_coils[i][3] - m[1])
                        <= 10):
                        m_comp += (index_m,)
                else:
                    if (0 <= m[0] - char_coils[i][2]
                        <= 10) and
                        (0 <= m[1] - char_coils[i][3]
                        <= 10):
                        m_comp += (index_m,)
            k_modes += (m_comp,)
        else:
            k_modes += ((),)
    modes += (k_modes,)

# Calculation of the average processing time
# to determine b.
avg_proc_time = sum(char_coils[i - 1][4]
    / Modes[m][1] for i in N_coil for k in K_range
    for m in modes[i - 1][k]) / (N * len(K))
b = avg_proc_time * N / len(K) + Par_dict["a"][u]

# Distribution of due dates d.
if urgency == "Very low" or urgency == "low"
    or urgency == "Basecase":
    d = [round(random.triangular(
```

```
                    Par_dict["a"][u], b, b),
                    2) for i in N_full]
            elif urgency == "High":
                d = []
                for i in N_full:
                    p = random.random()
                    if p <= 0.5:
                        d.append(round(random.uniform(
                            Par_dict["a"][u], b)))
                    else:
                        d.append(random.triangular(
                            Par_dict["a"][u], b, b))
            else:
                d = [round(random.uniform(
                    Par_dict["a"][u], b)) for i in N_full]

            # Determination of stringer costs c and
            # stringer processing times t.
            t = {}
            c = {}
            for i in N_full:
                t[i] = {}
                c[i] = {}
                for j in N_full:
                    t[i][j] = {}
                    c[i][j] = {}
                    for k in K_range:
                        t[i][j][k] = {}
                        c[i][j][k] = {}

                        # If the physical properties of the coils
                        # are not compatible with each other,
                        # the compatibility between modes does
                        # not need to be reviewed.
                        if (abs(char_coils[i][0] -
                            char_coils[j][0])
                            <= Par_dict["pf_w"][pf]) and (
                            abs(char_coils[i][1] -
                            char_coils[j][1])
                            <= Par_dict["pf_th"][pf]):
                            for m in modes[i][k]:
                                t[i][j][k][m] = {}
                                c[i][j][k][m] = {}
                                for n in modes[j][k]:
                                    if (abs(Modes[m][0] -
                                        Modes[n][0])
                                        <= Par_dict[
                                        "pf_temp"][pf]) and
                                        (abs(Modes[m][1] -
                                        Modes[n][1])
                                        <= Par_dict[
                                        "pf_speed"][pf]):
                                        t[i][j][k][m][n] = 0
                                        c[i][j][k][m][n] = 0
                                    else:
                                        t[i][j][k][m][n]
```

```
                                = Par_dict[
                                "str_time"][pr]
                            c[i][j][k][m][n] = 1
            else:
                for m in modes[i][k]:
                    t[i][j][k][m] = {}
                    c[i][j][k][m] = {}
                    for n in modes[j][k]:
                        t[i][j][k][m][n]
                            = Par_dict["str_time"][pr]
                        c[i][j][k][m][n] = 1

# Determination of processing times p for
# coil i in mode m.
p = {}
for i in N_full:
    p[i] = {}
    for k in K_range:
        p[i][k] = {}
        for m in modes[i][k]:
            p[i][k][m] = char_coils[i][4] / Modes[m][1]

# Determination of maximum duration of the schedule M.
M = 0
for k in K_range:
    M_ = 0
    for i in N_coil:
        M__ = 0
        for j in N_coil:
            for m in modes[i][k]:
                for n in modes[j][k]:
                    M___ = p[i][k][m] +
                        t[i][j][k][m][n]
                    M__ = max(M__, M___)
        M_ += M__
    M = max(M, M_)

# Definition of a parameter to optimize the instance
# and obtain characteristics of solution.
com_stud = CAL_model.CAL_model(N, N_coil, N_full,
    K_range, d, Modes_range, modes, alpha, M, c, t, p)

# Definition of output lines in certain scenarios.
try:
    com_stud.optimize()
    if com_stud.status == 9 and com_stud.SolCount == 0:
        sol.write(f"No solution found for {N} coils,
            seed = {seed}, alpha = {a}\n")
    elif com_stud.status == 3:
        sol.write(f"Model is infeasible for {N} coils,
            seed = {seed}, alpha = {a}\n")
    else:
        end = time.time()
        run = end - start
        tard = {}
        for i in N_coil:
```

```
                            z_var = com_stud.getVarByName(f"z[{i}]")
                            tard[i] = z_var.x
                        tardi = sum(tard.values())
                        sol.write(f"Coils = {N}, solution count
                                = {com_stud.SolCount}, "
                            f"solution = {com_stud.ObjVal},
                                MIPGap = {com_stud.getAttr(
                                GRB.Attr.MIPGap)}, "
                            f"Solving time = {com_stud.getAttr(
                                GRB.Attr.Runtime)},
                                Total time = {run},"
                            f"Number of linear constraints
                                = {com_stud.getAttr(
                                GRB.Attr.NumConstrs)},"
                            f"Number of Variables
                                = {com_stud.getAttr(
                                GRB.Attr.NumVars)},"
                            f"Number of binary/integer variables
                                = {com_stud.getAttr(
                                GRB.Attr.NumBinVars)},"
                            f"Tardiness = {tardi}, seed = {seed},
                            alpha = {a}\n")
                except MemoryError:
                    sol.write(f"Out of memory error for {N} coils,
                        seed = {seed}, alpha = {a}\n")

                # Deletion of all model data to free up RAM.
                del com_stud
                disposeDefaultEnv()

# Start of the study.
com_study()
```