



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

```
1 # Import of necessary package
2 from gurobipy import *
3
4 # Definition of model and utilized parameters from the instance
5 def CAL_model(N, N_coil, N_full, K_range, d, Modes_range, modes, alpha,
6               M, c, t, p):
7     model = Model()
8
9     # Decision variables.
10    y = model.addVars(N_full, N_full, K_range, Modes_range,
11                     Modes_range, vtype=GRB.BINARY, name="y")
12    x = model.addVars(N_full, K_range, Modes_range,
13                     Modes_range, vtype=GRB.BINARY, name="x")
14    delta = model.addVars(N_full, N_full, K_range,
15                          vtype=GRB.BINARY, name="delta")
16    z = model.addVars(N_full, vtype=GRB.BINARY, name="z")
17    s = model.addVars(N_full, vtype=GRB.CONTINUOUS, lb=0, name="s")
18
19    # Constraint 1 – 12. Due to the definition of the decision
20    # variables, constraint 13 from the mathematical model
21    # is not necessary.
22    # Time consistency constraints.
23    model.addConstrs((s[i] + quicksum(p[i][k][m] * x[i, k, m]
24                                     for k in K_range for m in modes[i][k])
25                          <= d[i] + z[i] * M for i in N_full), "1")
26    model.addConstrs((s[i] + quicksum((p[i][k][m] + t[i][j][k][m][n])
27                                     * y[i, j, k, m, n] for k in K_range
28                                     for m in modes[i][k] for n in modes[j][k])
29                          <= s[j] + M * (1 - quicksum(delta[i, j, k] for k in K_range))
30                          for i in N_full for j in N_full), "2")
31
32    # Decision variable consistency constraints.
33    model.addConstrs((delta[i, j, k] == quicksum(y[i, j, k, m, n]
34                                                  for m in modes[i][k] for n in modes[j][k])
35                      for i in N_full for j in N_full[: -1] for k in K_range), "3")
36    model.addConstrs((quicksum(y[i, j, k, m, n] for j in N_full
37                               for n in modes[j][k]) <= x[i, k, m]
38                      for i in N_full for k in K_range for m in modes[i][k]), "4")
39    model.addConstrs((quicksum(y[i, j, k, m, n] for i in N_full
40                               for m in modes[i][k]) <= x[j, k, n]
41                      for j in N_full for k in K_range for n in modes[j][k]), "5")
42
43    # Sequence consistency constraints.
44    model.addConstrs((quicksum(delta[i, j, k] for k in K_range
45                               for j in N_full[1:]) == 1 for i in N_coil), "6")
46    model.addConstrs((quicksum(delta[i, j, k] for k in K_range
47                               for i in N_full[: -1]) == 1 for j in N_coil), "7")
48    model.addConstrs((quicksum(delta[0, j, k] for j in N_full[1:])
49                      == 1 for k in K_range), "8")
50    model.addConstrs((quicksum(delta[i, N_full[-1], k]
51                               for i in N_full[: -1]) == 1 for k in K_range), "9")
```

```

52     model.addConstrs((quicksum(delta[j, i, k]
53         for j in N_full[: -1]) == quicksum(delta[i, j, k]
54         for j in N_full[1:]) for i in N_coil for k in K_range), "10")
55
56     # Additional constraints.
57     model.addConstr((quicksum(z[i] for i in N_coil) <= alpha), "11")
58     model.addConstrs((s[i] >= 0 for i in N_coil), "12")
59
60     # This constraint prevents the solver to set every x to one
61     # and limits x for coil i and all lines and modes to one.
62     model.addConstrs((quicksum(x[i, k, m] for k in K_range
63         for m in Modes_range) == 1 for i in N_coil), "13")
64
65     # Creation of a parameter that includes all stringers
66     # introduced in the schedule.
67     obj = quicksum(c[i][j][k][m][n] * y[i, j, k, m, n]
68         for i in N_coil for j in N_coil for k in K_range
69         for m in modes[i][k] for n in modes[j][k])
70
71     # Setting the objective to minimize the above obj.
72     model.setObjective(obj, GRB.MINIMIZE)
73
74     # Setting up a time limit.
75     model.setParam("TimeLimit", 720)
76
77     # Setting tuning parameters to increase performance.
78     model.Params.Cuts = 1
79     model.Params.PrePasses = 1
80     model.Params.Presolve = 2
81     if 40 >= N >= 30:
82         model.Params.heuristics = 1
83         model.Params.PrePasses = -1
84     elif 60 > N >= 50:
85         model.Params.heuristics = 1
86         model.Params.PrePasses = -1
87         model.Params.MIPFocus = 1
88     elif N == 60:
89         model.Params.heuristics = 1
90         model.Params.PrePasses = -1
91         model.Params.Cuts = 2
92         model.Params.MIPFocus = 1
93     elif N >= 70:
94         model.Params.Presolve = 1
95         model.Params.heuristics = 1
96         model.Params.PrePasses = -1
97         model.Params.Cuts = 2
98         model.Params.MIPFocus = 1
99         model.setParam(GRB.Param.NodefileStart, 0.5)
100         model.setParam("Threads", 8)
101
102     # Returns the solution of the optimization.
103     return model

```

A.2 Code - Instance

```
1 # Import of necessary packages
2 from gurobipy import *
3 import random
4 import time
5 import CAL_model
6
7 # Creation of function to study several instances at once
8 def com_study():
9
10     # Creation of output file
11     sol = open("Output_file.txt", "a")
12
13     # Input of different instance sizes N to study.
14     for i in []:
15         N = i
16
17     # Input of different service limits alpha to study.
18     for a in []:
19         alpha = a * N
20
21     # Insertion of a paragraph in the output file for better
22     # visualization.
23     sol.write("\n")
24
25     # Input of different seeds to study.
26     # For this study, seeds 12, 452, 535, 123, 363, 25, 50,
27     # 75, 100 and 125 were chosen.
28     for seed in []:
29         random.seed(seed)
30
31     # Information about the instance that is currently
32     # being optimized.
33     print(f"Instance size = {i}, alpha = {a},
34           seed = {seed}")
35
36     # Beginning of time measurement for total time.
37     start = time.time()
38
39     # Creation of two different N: N_full incorporates
40     # virtual coils, N_coil does not.
41     N_coil = range(N + 2)[1:-1]
42     N_full = range(N + 2)
43
44     # Setup of the specific levels for each factor. In this
45     # case, the Basecase scenario is chosen.
46     levels = ["Very low", "Low", "Basecase",
47              "High", "Very high"]
48     heterogeneity = levels[2]
49     urgency = levels[2]
50     process_flexibility = levels[2]
51     str_pro = levels[2]
52
53     # All parameters that are determined by the different
54     # levels of factors.
```

```

55 Par_dict = {
56     "a": [N * 2, N * 1.5, N],
57     "min_w": [70, 60, 60, 60, 60],
58     "max_w": [90, 100, 100, 100, 100],
59     "min_th": [0.15, 0.1, 0.1, 0.1, 0.1],
60     "max_th": [0.35, 0.499, 0.499, 0.499, 0.499],
61     "min_len": [15000, 14000, 14000, 14000, 14000],
62     "max_len": [16000, 17000, 17000, 17000, 17000],
63     "min_temp": [690, 680, 680, 680, 680],
64     "max_temp": [720, 730, 730, 730, 730],
65     "min_speed": [520, 510, 510, 510, 510],
66     "max_speed": [580, 590, 590, 590, 590],
67     "pf_w": [5, 5, 5, 10, 15],
68     "pf_th": [0.05, 0.1, 0.1, 0.1, 0.2],
69     "pf_temp": [0, 0, 10, 10, 20],
70     "pf_speed": [0, 20, 20, 20, 20],
71     "str_time": [0, 1.5, 3, 10, 15]
72 }
73
74 # Setup of index for the chosen level of PF.
75 if process_flexibility == "Very low":
76     pf = 0
77 elif process_flexibility == "Low":
78     pf = 1
79 elif process_flexibility == "Basecase":
80     pf = 2
81 elif process_flexibility == "High":
82     pf = 3
83 else:
84     pf = 4
85
86 # Setup of index for the chosen level of SPT.
87 if str_pro == "Very low":
88     pr = 0
89 elif str_pro == "Low":
90     pr = 1
91 elif str_pro == "Basecase":
92     pr = 2
93 elif str_pro == "High":
94     pr = 3
95 else:
96     pr = 4
97
98 # Setup of index for the chosen level of UC.
99 if urgency == "Very low":
100     u = 0
101 elif urgency == "Low":
102     u = 1
103 else:
104     u = 2
105
106 # Setup of index for the chosen level of HC and
107 # distribution of coil characteristics.
108 if heterogeneity == "Very low":
109     h = 0
110 elif heterogeneity == "Low":

```

```

111         h = 1
112     elif heterogeneity == "Basecase":
113         h = 2
114         char_coils = []
115         for i in N_full:
116             p = random.random()
117             if p <= 0.5:
118                 char_coils.append(
119                     [random.randrange(
120                         Par_dict["min_w"][h],
121                         Par_dict["max_w"][h]),
122                     round(random.uniform(
123                         Par_dict["min_th"][h],
124                         Par_dict["max_th"][h]), 3),
125                     random.randrange(
126                         Par_dict["min_temp"][h],
127                         Par_dict["max_temp"][h], 10),
128                     random.randrange(
129                         Par_dict["min_speed"][h],
130                         Par_dict["max_speed"][h], 10),
131                     random.randrange(
132                         Par_dict["min_len"][h],
133                         Par_dict["max_len"][h]))
134             else:
135                 char_coils.append([random.triangular(
136                     Par_dict["min_w"][h],
137                     Par_dict["max_w"][h],
138                     (Par_dict["max_w"][h] +
139                     Par_dict["min_w"][h]) / 2),
140                     round(random.triangular(
141                         Par_dict["min_th"][h],
142                         Par_dict["max_th"][h],
143                         (Par_dict["max_th"][h] +
144                         Par_dict["min_th"][h]) / 2), 3),
145                     round(random.triangular(
146                         Par_dict["min_temp"][h] / 10,
147                         Par_dict["max_temp"][h] / 10,
148                         (Par_dict["max_temp"][h] +
149                         Par_dict["min_temp"][h]) / 20))
150                     * 10,
151                     round(random.triangular(
152                         Par_dict["min_speed"][h] / 10,
153                         Par_dict["max_speed"][h] / 10,
154                         (Par_dict["max_speed"][h] +
155                         Par_dict["min_speed"][h]) / 20))
156                     * 10,
157                     round(random.triangular(
158                         Par_dict["min_len"][h],
159                         Par_dict["max_len"][h],
160                         (Par_dict["max_len"][h] +
161                         Par_dict["min_len"][h]) / 2)))
162     elif heterogeneity == "High":
163         h = 3
164     else:
165         h = 4
166

```

```

167 # To save space, HC levels with the same distribution
168 # of coil characteristics are combined.
169 if (heterogeneity == "Very low")
170     or (heterogeneity == "Low"):
171         char_coils = [[random.triangular(
172             Par_dict["min_w"][h],
173             Par_dict["max_w"][h],
174             (Par_dict["max_w"][h] +
175             Par_dict["min_w"][h]) / 2),
176             round(random.triangular(Par_dict["min_th"][h],
177             Par_dict["max_th"][h],
178             (Par_dict["max_th"][h] +
179             Par_dict["min_th"][h]) / 2), 3),
180             round(random.triangular(
181             Par_dict["min_temp"][h] / 10,
182             Par_dict["max_temp"][h] / 10,
183             (Par_dict["max_temp"][h] +
184             Par_dict["min_temp"][h]) / 20)) * 10,
185             round(random.triangular(
186             Par_dict["min_speed"][h] / 10,
187             Par_dict["max_speed"][h] / 10,
188             (Par_dict["max_speed"][h] +
189             Par_dict["min_speed"][h]) / 20)) * 10,
190             round(random.triangular(
191             Par_dict["min_len"][h],
192             Par_dict["max_len"][h],
193             (Par_dict["max_len"][h] +
194             Par_dict["min_len"][h]) / 2))]
195         for i in N_full]
196 elif (heterogeneity == "High")
197     or (heterogeneity == "Very high"):
198         char_coils = [[random.randrange(
199             Par_dict["min_w"][h],
200             Par_dict["max_w"][h]),
201             round(random.uniform(
202             Par_dict["min_th"][h],
203             Par_dict["max_th"][h]), 3),
204             random.randrange(
205             Par_dict["min_temp"][h],
206             Par_dict["max_temp"][h], 10),
207             random.randrange(
208             Par_dict["min_speed"][h],
209             Par_dict["max_speed"][h], 10),
210             random.randrange(
211             Par_dict["min_len"][h],
212             Par_dict["max_len"][h])]
213         for i in N_full]
214
215 # Parameterization of the heterogeneity
216 # of the processing lines k.
217 K = [0, (Par_dict["min_w"][h] +
218         (Par_dict["max_w"][h] -
219         Par_dict["min_w"][h]) / 2 - 5),
220      (Par_dict["min_w"][h] +
221      (Par_dict["max_w"][h] -
222      Par_dict["min_w"][h]) / 2 + 5)]

```

```

223     K_range = range(len(K))
224
225     # Determination of all process modes.
226     Modes = ()
227     for m_t in range(Par_dict["min_temp"][h] - 10,
228                     Par_dict["max_temp"][h]+
229                     11, 10):
230         Modes_ = ((m_t, m_s) for m_s in range(
231                     Par_dict["min_speed"][h] - 10,
232                     Par_dict["max_speed"][h] + 11, 20))
233         Modes += tuple(Modes_)
234     Modes_range = range(len(Modes))
235
236     # Determination of feasible processing modes
237     # for coil i on processing line k.
238     modes = ()
239     for i in N_full:
240         k_modes = ()
241         for index_k, k in enumerate(K):
242             m_comp = ()
243
244             # If coil i is not compatible with line k,
245             # it has no feasible processing modes
246             # on this line.
247             if (index_k == 0) or (index_k == 1 and
248                 k <= char_coils[i][0]) or (
249                 index_k == 2 and k >= char_coils[i][0]):
250                 for index_m, m in enumerate(Modes):
251                     if heterogeneity != "Very high":
252                         if (abs(char_coils[i][2] - m[0])
253                             <= 10) and
254                             (abs(char_coils[i][3] - m[1])
255                             <= 10):
256                             m_comp += (index_m,)
257                     else:
258                         if (0 <= m[0] - char_coils[i][2]
259                             <= 10) and
260                             (0 <= m[1] - char_coils[i][3]
261                             <= 10):
262                             m_comp += (index_m,)
263                 k_modes += (m_comp,)
264             else:
265                 k_modes += ((),)
266         modes += (k_modes,)
267
268     # Calculation of the average processing time
269     # to determine b.
270     avg_proc_time = sum(char_coils[i - 1][4]
271                         / Modes[m][1] for i in N_coil for k in K_range
272                         for m in modes[i - 1][k]) / (N * len(K))
273     b = avg_proc_time * N / len(K) + Par_dict["a"][u]
274
275     # Distribution of due dates d.
276     if urgency == "Very low" or urgency == "low"
277         or urgency == "Basecase":
278         d = [round(random.triangular(

```

```

279         Par_dict["a"][u], b, b),
280         2) for i in N_full]
281     elif urgency == "High":
282         d = []
283         for i in N_full:
284             p = random.random()
285             if p <= 0.5:
286                 d.append(round(random.uniform(
287                     Par_dict["a"][u], b)))
288             else:
289                 d.append(random.triangular(
290                     Par_dict["a"][u], b, b))
291     else:
292         d = [round(random.uniform(
293             Par_dict["a"][u], b)) for i in N_full]
294
295     # Determination of stringer costs c and
296     # stringer processing times t.
297     t = {}
298     c = {}
299     for i in N_full:
300         t[i] = {}
301         c[i] = {}
302         for j in N_full:
303             t[i][j] = {}
304             c[i][j] = {}
305             for k in K_range:
306                 t[i][j][k] = {}
307                 c[i][j][k] = {}
308
309         # If the physical properties of the coils
310         # are not compatible with each other,
311         # the compatibility between modes does
312         # not need to be reviewed.
313         if (abs(char_coils[i][0] -
314             char_coils[j][0])
315             <= Par_dict["pf_w"][pf]) and (
316             abs(char_coils[i][1] -
317             char_coils[j][1])
318             <= Par_dict["pf_th"][pf]):
319             for m in modes[i][k]:
320                 t[i][j][k][m] = {}
321                 c[i][j][k][m] = {}
322                 for n in modes[j][k]:
323                     if (abs(Modes[m][0] -
324                         Modes[n][0])
325                         <= Par_dict[
326                             "pf_temp"][pf]) and
327                         (abs(Modes[m][1] -
328                             Modes[n][1])
329                         <= Par_dict[
330                             "pf_speed"][pf]):
331                         t[i][j][k][m][n] = 0
332                         c[i][j][k][m][n] = 0
333                     else:
334                         t[i][j][k][m][n]

```

```

335                                     = Par_dict[
336                                     "str_time"][pr]
337                                     c[i][j][k][m][n] = 1
338     else:
339         for m in modes[i][k]:
340             t[i][j][k][m] = {}
341             c[i][j][k][m] = {}
342             for n in modes[j][k]:
343                 t[i][j][k][m][n]
344                 = Par_dict["str_time"][pr]
345                 c[i][j][k][m][n] = 1
346
347     # Determination of processing times p for
348     # coil i in mode m.
349     p = {}
350     for i in N_full:
351         p[i] = {}
352         for k in K_range:
353             p[i][k] = {}
354             for m in modes[i][k]:
355                 p[i][k][m] = char_coils[i][4] / Modes[m][1]
356
357     # Determination of maximum duration of the schedule M.
358     M = 0
359     for k in K_range:
360         M_ = 0
361         for i in N_coil:
362             M__ = 0
363             for j in N_coil:
364                 for m in modes[i][k]:
365                     for n in modes[j][k]:
366                         M___ = p[i][k][m] +
367                             t[i][j][k][m][n]
368                         M__ = max(M__, M___)
369             M_ += M__
370     M = max(M, M_)
371
372     # Definition of a parameter to optimize the instance
373     # and obtain characteristics of solution.
374     com_stud = CAL_model.CAL_model(N, N_coil, N_full,
375                                     K_range, d, Modes_range, modes, alpha, M, c, t, p)
376
377     # Definition of output lines in certain scenarios.
378     try:
379         com_stud.optimize()
380         if com_stud.status == 9 and com_stud.SolCount == 0:
381             sol.write(f"No solution found for {N} coils ,
382                     seed = {seed}, alpha = {a}\n")
383         elif com_stud.status == 3:
384             sol.write(f"Model is infeasible for {N} coils ,
385                     seed = {seed}, alpha = {a}\n")
386         else:
387             end = time.time()
388             run = end - start
389             tard = {}
390             for i in N_coil:

```

```

391         z_var = com_stud.getVarByName(f"z[{i}]")
392         tard[i] = z_var.x
393     tardi = sum(tard.values())
394     sol.write(f"Coils = {N}, solution count
395             = {com_stud.SolCount}, "
396             f"solution = {com_stud.ObjVal},
397             MIPGap = {com_stud.getAttr(
398             GRB.Attr.MIPGap)}, "
399             f"Solving time = {com_stud.getAttr(
400             GRB.Attr.Runtime)},
401             Total time = {run}, "
402             f"Number of linear constraints
403             = {com_stud.getAttr(
404             GRB.Attr.NumConstrs)}, "
405             f"Number of Variables
406             = {com_stud.getAttr(
407             GRB.Attr.NumVars)}, "
408             f"Number of binary/integer variables
409             = {com_stud.getAttr(
410             GRB.Attr.NumBinVars)}, "
411             f"Tardiness = {tardi}, seed = {seed},
412             alpha = {a}\n")
413 except MemoryError:
414     sol.write(f"Out of memory error for {N} coils,
415             seed = {seed}, alpha = {a}\n")
416
417 # Deletion of all model data to free up RAM.
418 del com_stud
419 disposeDefaultEnv()
420
421 # Start of the study.
422 com_study()

```