

```

1 # (C) Andrew Mason 2023 - ENGSCI 760 Heuristics Assignment
2 # This code, and any code derived from this, may NOT be posted in any publicly accessible location
3 # Specifically, this code, and any derived versions of this code (including your assignment answers)
4 # must NOT be posted publically on Github, Gitlab or similar.
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8 from enum import IntEnum
9 import time
10 import random
11
12 class Element(IntEnum):
13     """The elements that we measure levels of in the Aluminium we produce"""
14     Al = 0
15     Fe = 1
16     Si = 2
17
18 class LocalSearch():
19     def __init__(self) -> None:
20         self.load_default_problem()
21
22     def load_default_problem(self) -> None:
23         """Initialise the configuration parameters with default values"""
24         self.no_crucibles=17
25         self.no_pots=51
26         self.pots_per_crucible=3
27         # Initialise the percentage of Al (aluminium), Fe (iron) and Silicon (Si)
28         self.pot_quality = np.array(
29             [ [99.136, 0.051, 0.497],
30               [99.733, 0.064, 0.138],
31               [99.755, 0.083, 0.149],
32               [99.198, 0.318, 0.206],
33               [99.297, 0.284, 0.33],
34               [99.23, 0.327, 0.393],
35               [99.485, 0.197, 0.156],
36               [99.709, 0.011, 0.056],
37               [99.729, 0.007, 0.012],
38               [99.118, 0.434, 0.377],
39               [99.372, 0.01, 0.349],
40               [99.505, 0.028, 0.433],
41               [99.187, 0.296, 0.335],
42               [99.043, 0.224, 0.531],
43               [99.206, 0.166, 0.146],
44               [99.395, 0.188, 0.328],
45               [99.436, 0.199, 0.303],
46               [99.796, 0.009, 0.144],
47               [99.186, 0.397, 0.065],
48               [99.455, 0.079, 0.278],
49               [99.553, 0.084, 0.353],
50               [99.539, 0.017, 0.201],
51               [99.38, 0.082, 0.239],
52               [99.504, 0.009, 0.273],
53               [99.391, 0.261, 0.297],
54               [99.374, 0.015, 0.578],
55               [99.462, 0.179, 0.109],
56               [99.03, 0.213, 0.459],
57               [99.328, 0.131, 0.371],
58               [99.674, 0.055, 0.249],
59               [99.413, 0.137, 0.1],
60               [99.538, 0.046, 0.151],
61               [99.41, 0.109, 0.08],
62               [99.163, 0.324, 0.343],
63               [99.502, 0.036, 0.412],
64               [99.66, 0.083, 0.069],
65               [99.629, 0.156, 0.069],
66               [99.592, 0.171, 0.008],
67               [99.684, 0.011, 0.106],
68               [99.358, 0.227, 0.137],
69               [99.145, 0.161, 0.403],
70               [99.729, 0.028, 0.123],
71               [99.335, 0.181, 0.351],
72               [99.725, 0.094, 0.14],
73               [99.124, 0.325, 0.015],
74               [99.652, 0.068, 0.029],
75               [99.091, 0.268, 0.565],
76               [99.426, 0.146, 0.256],
77               [99.383, 0.266, 0.039],
78               [99.481, 0.147, 0.327],
79               [99.163, 0.121, 0.71] ] )
80         # Initialise the impurity limits & dollar values associated with the different quality grades of Al (aluminium)
81         # We require at least a minimum $ Al, and no more than max Fe (iron) and Si (Silicon) $'s
82         self.no_grades = 11
83         self.grade_min_Al=[95.00,99.10,99.20,99.25,99.35,99.50,99.65,99.75,99.85,99.90]
84         self.grade_max_Fe=[ 5.00, 0.81, 0.81, 0.79, 0.76, 0.72, 0.53, 0.50, 0.46, 0.33, 0.30]
85         self.grade_max_Si=[ 3.00, 0.40, 0.41, 0.43, 0.39, 0.35, 0.28, 0.28, 0.21, 0.15, 0.15]
86         self.grade_value=[ 10.00,21.25,26.95,36.25,41.53,44.53,48.71,52.44,57.35,68.21,72.56]
87
88     def load_small_problem(self) -> None:
89         """Initialise the configuration parameters with default values, and then modify the sizing to give a smaller problem with 10 crucibles"""
90         self.load_default_problem()
91         self.no_crucibles=10
92         self.no_pots=self.no_crucibles * self.pots_per_crucible
93
94     def calc_crucible_value(self, crucible_quality) -> float:
95         """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
96         Returns 0 if the aluminium does not satisfy any of the quality grades."""
97         tol = 0.00001 # We allow for small errors in 5th decimal point
98         for q in reversed(range(self.no_grades)):
99             if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
100                crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
101                crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
102                 return self.grade_value[q]
103         return 0.0
104
105     def calc_crucible_value_with_spread(self, crucible_quality, spread: int, max_spread: int) -> float:
106         """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
107         Returns 0 if the aluminium does not satisfy any of the quality grades."""
108         tol = 0.00001 # We allow for small errors in 5th decimal point
109         spread_penalty = -10000*(spread - max_spread) if spread > max_spread else 0
110         for q in reversed(range(self.no_grades)):
111             if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
112                crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
113                crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
114                 return self.grade_value[q] + spread_penalty
115         return 0.0
116
117     def view_soln(self, x, max_allowed_spread: int=0) -> None:
118         """Print solution x with its statistics. Note that our output numbers items from 1, not 0"""
119

```

```

119 max_spread = 0
120 crucible_value_sum = 0
121 for c in range(self.no_crucibles):
122     spread = max(x[c]) - min(x[c])
123     max_spread = max(max_spread, spread)
124     crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
125     if max_allowed_spread:
126         crucible_value = self.calc_crucible_value_with_spread(crucible_quality, spread, max_allowed_spread)
127     else:
128         crucible_value = self.calc_crucible_value(crucible_quality)
129
130     crucible_value_sum += crucible_value
131     print(f'[c+1:>2] [{x[c][0]+1:>2}] {x[c][1]+1:>2}] {x[c][2]+1:>2}] '
132           f'(crucible_quality[Element.Al]:>5.3f) %Al, '
133           f'(crucible_quality[Element.Fe]:>5.3f) %Fe, '
134           f'(crucible_quality[Element.Si]:>5.3f) %Si, '
135           f'${crucible_value:>5.2f}, spread = {spread:>2}')
136
137 print(f'                               Sum = ${round(crucible_value_sum,2):>6}, MxSprd = {max_spread:>2}')
138
139 def calc_obj(self, x, max_allowed_spread: int=0):
140     """Calculate the total profit for a given solution"""
141     crucible_value_sum = 0
142     for c in range(self.no_crucibles):
143         crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
144         if max_allowed_spread:
145             crucible_value = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_allowed_spread)
146         else:
147             crucible_value = self.calc_crucible_value ( crucible_quality ) ;
148             crucible_value_sum += crucible_value
149     return crucible_value_sum
150
151 def trivial_solution(self):
152     """Return a solution x=[0,1,2,3,4,5,6,7,8;...;48,49,50] of pots assigned to crucibles"""
153     return np.arange(self.no_pots).reshape(self.no_crucibles, self.pots_per_crucible)
154
155 def random_solution(self):
156     """Return a random solution of pots assigned to crucibles by shuffling the values in [0,1,2,3,4,5,6,7,8;...;48,49,50] """
157     rng = np.random.default_rng()
158     x = np.arange(self.no_pots)
159     rng.shuffle(x)
160     return x.reshape(self.no_crucibles, self.pots_per_crucible)
161
162 def plot_ascent(self, fx, fy):
163     plt.plot(fy, 'r', label="f(y)")
164     plt.plot(fx, 'b', label="f(x)")
165     plt.xlabel('Function Evaluation Count')
166     plt.ylabel('Objective Function Value')
167     plt.legend()
168     plt.show()
169
170 #####
171 # TASK 3A #
172 #####
173 def next_ascent_to_local_max(self, random_start=True, plotting=False):
174     if random_start:
175         x = self.random_solution()
176     else:
177         x = self.trivial_solution()
178
179     # intermediate values
180     last_crucible_values = np.zeros(self.no_crucibles)
181     for c in range(self.no_crucibles):
182         crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
183         last_crucible_values[c] = self.calc_crucible_value(crucible_quality)
184
185     if plotting:
186         fx = []
187         fy = []
188         fx.append(sum(last_crucible_values))
189         fy.append(sum(last_crucible_values))
190
191     # for default case
192     last_optimal_indices = (-1, -1, -1, -1)
193     while True:
194         # loop through neighborhood
195         for k in range(self.no_crucibles-1):
196             for m in range(self.pots_per_crucible):
197                 for l in range(k+1, self.no_crucibles):
198                     for n in range(self.pots_per_crucible):
199
200                         # exactly one scan since last optimal value found, can return
201                         if (k, m, l, n) == last_optimal_indices:
202                             if plotting:
203                                 self.plot_ascent(fx, fy)
204                             return x
205
206                         # calculate crucible values and delta
207                         crucible_k = x[k].copy()
208                         crucible_l = x[l].copy()
209                         crucible_k[m] = x[l][n]
210                         crucible_l[n] = x[k][m]
211                         crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
212                         crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
213                         crucible_k_value = self.calc_crucible_value(crucible_k_quality)
214                         delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
215
216                         if plotting:
217                             fy.append(sum(last_crucible_values) + delta)
218
219
220                         # > 0.001 as don't want to accept new solution if floating point error
221                         if delta > 0.001:
222                             # update intermediate values, solution, and optimal indices
223                             last_optimal_indices = (k, m, l, n)
224                             last_crucible_values[k] = crucible_k_value
225                             last_crucible_values[l] = crucible_l_value
226                             x[k][m] = crucible_k[m]
227                             x[l][n] = crucible_l[n]
228
229                         if plotting:
230                             fx.append(sum(last_crucible_values))
231
232     # case where starting at local max
233     if last_optimal_indices == (-1, -1, -1, -1):
234         if plotting:
235             self.plot_ascent(fx, fy)
236         return x
237
238 #####

```

```

239 # TASK 3B #
240 #####
241 def steepest_ascent_to_local_max(self, random_start=True, plotting=False):
242     if random_start:
243         x = self.random_solution()
244     else:
245         x = self.trivial_solution()
246
247     # intermediate values
248     last_crucible_values = np.zeros(self.no_crucibles)
249     for c in range(self.no_crucibles):
250         crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
251         last_crucible_values[c] = self.calc_crucible_value(crucible_quality)
252
253     if plotting:
254         fx = []
255         fy = []
256         fx.append(sum(last_crucible_values))
257         fy.append(sum(last_crucible_values))
258
259     while True:
260         optimal_swap = (-1, -1, -1, -1)
261
262         # min starting delta 0.001 for floating point errors
263         best_delta = 0.001
264         for k in range(self.no_crucibles-1):
265             for m in range(self.pots_per_crucible):
266                 for l in range(k+1, self.no_crucibles):
267                     for n in range(self.pots_per_crucible):
268
269                         # calculate crucible values and delta
270                         crucible_k = x[k].copy()
271                         crucible_l = x[l].copy()
272                         crucible_k[m] = x[l][n]
273                         crucible_l[n] = x[k][m]
274                         crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
275                         crucible_k_value = self.calc_crucible_value(crucible_k_quality)
276                         crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
277                         crucible_l_value = self.calc_crucible_value(crucible_l_quality)
278                         delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
279
280                     if plotting:
281                         fy.append(sum(last_crucible_values) + delta)
282                         fx.append(sum(last_crucible_values))
283
284                     # if new steepest update best delta and save optimal swap location
285                     if delta > best_delta:
286                         best_delta = delta
287                         optimal_swap = (k, m, l, n)
288
289         # if all neighbors scanned and no better solution found, at local max and finish
290         if optimal_swap == (-1, -1, -1, -1):
291             if plotting:
292                 self.plot_ascent(fx, fy)
293             return x
294
295         # Make swap with steepest neighbor and update intermediate values
296         k, m, l, n = optimal_swap
297         crucible_k = x[k].copy()
298         crucible_l = x[l].copy()
299         crucible_k[m] = x[l][n]
300         crucible_l[n] = x[k][m]
301         crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
302         crucible_k_value = self.calc_crucible_value(crucible_k_quality)
303         crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
304         crucible_l_value = self.calc_crucible_value(crucible_l_quality)
305         last_crucible_values[k] = crucible_k_value
306         last_crucible_values[l] = crucible_l_value
307         x[k][m] = crucible_k[m]
308         x[l][n] = crucible_l[n]
309
310 #####
311 # TASK 3E #
312 #####
313 def do_repeated_next_ascents(self, n: int, max_spread: int = 0):
314     best_obj_history = []
315     obj_history = []
316     times = []
317
318     # Iterate through random starts to find history and best solution
319     best_obj = 0
320     start_time = time.perf_counter()
321     for _ in range(n):
322         if max_spread:
323             x = self.next_ascent_to_local_max_spread(max_spread)
324         else:
325             x = self.next_ascent_to_local_max()
326         obj = self.calc_obj(x)
327         if obj > best_obj:
328             best_x = x
329             best_obj = obj
330         best_obj_history.append(best_obj)
331         obj_history.append(obj)
332         times.append(time.perf_counter() - start_time)
333
334     # Output and plot best solution
335     self.view_soln(best_x)
336     plt.scatter(times, best_obj_history, c='b', s=1, label="Best objective value")
337     plt.scatter(times, obj_history, c='r', s=5, label="Local max")
338     plt.xlabel('Time (s)')
339     plt.ylabel('Objective Function Value')
340     plt.legend()
341     plt.gcf().set_size_inches(11.69, 8.27)
342     plt.savefig("assets/repeated_next_ascents_chart.pdf", orientation="landscape")
343
344 #####
345 # TASK 3E #
346 #####
347 def do_repeated_steepest_ascents(self, n: int):
348     best_obj_history = []
349     obj_history = []
350     times = []
351
352     # Iterate through random starts to find history and best solution
353     best_obj = 0
354     start_time = time.perf_counter()
355     for _ in range(n):
356         x = self.steepest_ascent_to_local_max()

```

```

358         obj = self.calc_obj(x)
359         if obj > best_obj:
360             best_x = x
361             best_obj = obj
362         best_obj_history.append(best_obj)
363         obj_history.append(obj)
364         times.append(time.perf_counter() - start_time)
365
366     # Output and plot best solution
367     self.view_soln(best_x)
368     plt.scatter(times,best_obj_history,c='b',s=1, label="Best objective value")
369     plt.scatter(times,obj_history,c='r',s=5, label="Local max")
370     plt.xlabel('Time (s)')
371     plt.ylabel('Objective Function Value')
372     plt.legend()
373     plt.show()
374
375     #####
376     # TASK 6 #
377     #####
378     def next_ascent_to_local_max_spread(self, max_spread: int, random_start=True):
379         if random_start:
380             x = self.random_solution()
381         else:
382             x = self.trivial_solution()
383
384         # init intermeideate values
385         last_crucible_values = np.zeros(self.no_crucibles)
386         for c in range(self.no_crucibles):
387             crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
388             last_crucible_values[c] = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_spread)
389
390         # Loop through neighbors
391         last_optimal_indices = (-1, -1, -1, -1)
392         while True:
393             for k in range(self.no_crucibles-1):
394                 for m in range(self.pots_per_crucible):
395                     for l in range(k+1, self.no_crucibles):
396                         for n in range(self.pots_per_crucible):
397                             # looped through all neighbors once and no better solution found
398                             if (k, m, l, n) == last_optimal_indices:
399                                 return x
400
401                             # calculate delta and other relevant params
402                             crucible_k = x[k].copy()
403                             crucible_l = x[l].copy()
404                             crucible_k[m] = x[l][n]
405                             crucible_l[n] = x[k][m]
406                             crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
407                             crucible_k_value = self.calc_crucible_value_with_spread(crucible_k_quality, np.ptp(crucible_k), max_spread)
408                             crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
409                             crucible_l_value = self.calc_crucible_value_with_spread(crucible_l_quality, np.ptp(crucible_l), max_spread)
410                             delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
411
412                             # better solution so update intermediate values and solution
413                             if delta > 0.01:
414                                 last_optimal_indices = (k, m, l, n)
415                                 last_crucible_values[k] = crucible_k_value
416                                 last_crucible_values[l] = crucible_l_value
417                                 x[k][m] = crucible_k[m]
418                                 x[l][n] = crucible_l[n]
419
420         # case where already at local max
421         if last_optimal_indices == (-1, -1, -1, -1):
422             return x

```