

```

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               [99.533, 0.216, 0.353],

```

```

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
81     # Initialise the impurity limits & dollar values associated with the different quality grades of Al (aluminium)
82     # We require at least a minimum % Al, and no more than max Fe (iron) and Si (Silicon) %'s
83     self.no_grades = 11
84     self.grade_min_Al=[95.00,99.10,99.10,99.20,99.25,99.35,99.50,99.65,99.75,99.85,99.90]
85     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]
86     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]
87     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]
88
89     def load_small_problem(self) -> None:
90         """Initialise the configuration parameters with default values, and then modify the sizing to give a smaller problem with 10 crucibles"""
91         self.load_default_problem()
92         self.no_crucibles=10
93         self.no_pots=self.no_crucibles * self.pots_per_crucible
94
95     def calc_crucible_value(self, crucible_quality) -> float:
96         """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
97         Returns 0 if the aluminium does not satisfy any of the quality grades."""
98         tol = 0.00001 # We allow for small errors in 5th decimal point
99         for q in reversed(range(self.no_grades)):
100             if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
101                crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
102                crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
103                 return self.grade_value[q]
104         return 0.0
105
106     # Calculate the crucible value with a maximum allowed spread
107     def calc_crucible_value_with_spread(self, crucible_quality, spread: int, max_spread: int) -> float:
108         """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
109         Returns 0 if the aluminium does not satisfy any of the quality grades."""
110         tol = 0.00001 # We allow for small errors in 5th decimal point
111         # spread penalty calculation
112         spread_penalty = -20*(spread - max_spread) if spread > max_spread else 0
113         for q in reversed(range(self.no_grades)):
114             if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
115                crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
116                crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
117                 return self.grade_value[q] + spread_penalty
118         return 0.0
119
120     def view_soln(self, x, max_allowed_spread: int=0) -> None:
121         """Print solution x with its statistics. Note that our output numbers items from 1 not 0"""

```

```

120 ### Solution x with its statistics. Note that our output numbers items from 1, not 0
121 max_spread = 0
122 crucible_value_sum = 0
123 for c in range (self.no_crucibles):
124     spread = max(x[c]) - min(x[c])
125     max_spread = max(max_spread, spread)
126     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]
127     # max allowed spread functionality added (only calculate with max allowed spread if defined non-zero)
128     if max_allowed_spread:
129         crucible_value = self.calc_crucible_value_with_spread(crucible_quality, spread, max_allowed_spread)
130     else:
131         crucible_value = self.calc_crucible_value(crucible_quality)
132
133     crucible_value_sum += crucible_value
134     print(f'{c+1:>2} {x[c][0]+1:>2} {x[c][1]+1:>2} {x[c][2]+1:>2} ] '
135           f'{crucible_quality[Element.Al]:>5.3f} %Al, '
136           f'{crucible_quality[Element.Fe]:>5.3f} %Fe, '
137           f'{crucible_quality[Element.Si]:>5.3f} %Si, '
138           f'${crucible_value:>5.2f}, spread = {spread:>2}')
139 print(f'                                     Sum = ${round(crucible_value_sum,2):>6}, MxSprd = {max_spread:>2}')
140
141 def calc_obj(self, x, max_allowed_spread: int=0):
142     """Calculate the total profit for a given solution"""
143     crucible_value_sum = 0
144     for c in range (self.no_crucibles):
145         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]
146         # max allowed spread functionality added (only calculate with max allowed spread if defined non-zero)
147         if max_allowed_spread:
148             crucible_value = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_allowed_spread)
149         else:
150             crucible_value = self.calc_crucible_value ( crucible_quality ) ;
151         crucible_value_sum += crucible_value
152     return crucible_value_sum
153
154 def trivial_solution(self):
155     """Return a solution x=[0,1,2;3,4,5;6,7,8;...;48,49,50] of pots assigned to crucibles"""
156     return np.arange(self.no_pots).reshape(self.no_crucibles, self.pots_per_crucible)
157
158 def random_solution(self):
159     """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] """
160     rng = np.random.default_rng()
161     x = np.arange(self.no_pots)
162
163     rng.shuffle(x)
164     return x.reshape(self.no_crucibles, self.pots_per_crucible)
165
166 def plot_ascent(self, fx, fy, save_name: str, title: str):
167     fig = plt.figure()
168     plt.plot(fy, 'r', label="f(y)")
169     plt.plot(fx, 'b', label="f(x)")
170     plt.xlabel('Function Evaluation Count')
171     plt.ylabel('Objective Function Value')
172     plt.legend()
173     plt.title(title)
174     plt.gcf().set_size_inches(11.69, 8.27)
175     plt.savefig(f"./report/assets/{save_name}", orientation="landscape")
176
177 #####
178 # TASK 3A #
179 #####
180 def next_ascent_to_local_max(self, random_start=True, plotting=False):
181     if random start:

```

```

181         x = self.random_solution()
182     else:
183         x = self.trivial_solution()
184
185     # intermediate values
186     last_crucible_values = np.zeros(self.no_crucibles)
187     for c in range(self.no_crucibles):
188         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]
189         last_crucible_values[c] = self.calc_crucible_value(crucible_quality)
190
191     if plotting:
192         fx = []
193         fy = []
194         fx.append(sum(last_crucible_values))
195         fy.append(sum(last_crucible_values))
196
197     # for default case
198     last_optimal_indices = (-1, -1, -1, -1)
199     while True:
200         # loop through neighborhood
201         for k in range(self.no_crucibles-1):
202             for m in range(self.pots_per_crucible):
203                 for l in range(k+1, self.no_crucibles):
204                     for n in range(self.pots_per_crucible):
205
206                         # exactly one scan since last optimal value found, can return
207                         if (k, m, l, n) == last_optimal_indices:
208                             if plotting:
209                                 self.plot_ascent(fx, fy, "next_ascent_chart.pdf", "Task 3C")
210                             return x
211
212                         # calculate crucible values and delta
213                         crucible_k = x[k].copy()
214                         crucible_l = x[l].copy()
215                         crucible_k[m] = x[l][n]
216
217                         crucible_l[n] = x[k][m]
218                         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]
219                         crucible_k_value = self.calc_crucible_value(crucible_k_quality)
220                         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]
221                         crucible_l_value = self.calc_crucible_value(crucible_l_quality)
222                         delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
223
224                         if plotting:
225                             fy.append(sum(last_crucible_values) + delta)
226
227                         # > 0.001 as don't want to accept new solution if floating point error
228                         if delta > 0.001:
229                             # update intermediate values, solution, and optimal indices
230                             last_optimal_indices = (k, m, l, n)
231                             last_crucible_values[k] = crucible_k_value
232                             last_crucible_values[l] = crucible_l_value
233                             x[k][m] = crucible_k[m]
234                             x[l][n] = crucible_l[n]
235
236                             if plotting:
237                                 fx.append(sum(last_crucible_values))
238
239     # case where starting at local max
240     if last_optimal_indices == (-1, -1, -1, -1):

```

```

241         if plotting:
242             self.plot_ascent(fx, fy, "next_ascent_chart.pdf", "Task 3C")
243         return x
244
245 #####
246 # TASK 3B #
247 #####
248 def steepest_ascent_to_local_max(self, random_start=True, plotting=False):
249     if random_start:
250         x = self.random_solution()
251     else:
252         x = self.trivial_solution()
253
254     # intermediate values
255     last_crucible_values = np.zeros(self.no_crucibles)
256     for c in range(self.no_crucibles):
257         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]
258         last_crucible_values[c] = self.calc_crucible_value(crucible_quality)
259
260     if plotting:
261         fx = []
262         fy = []
263         fx.append(sum(last_crucible_values))
264         fy.append(sum(last_crucible_values))
265
266     while True:
267         optimal_swap = (-1, -1, -1, -1)
268
269         # min starting delta 0.001 for floating point errors
270         best_delta = 0.001
271         for k in range(self.no_crucibles-1):
272             for m in range(self.pots_per_crucible):
273                 for l in range(k+1, self.no_crucibles):
274                     for n in range(self.pots_per_crucible):
275
276                         # calculate crucible values and delta
277                         crucible_k = x[k].copy()
278                         crucible_l = x[l].copy()
279                         crucible_k[m] = x[l][n]
280                         crucible_l[n] = x[k][m]
281                         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]
282                         crucible_k_value = self.calc_crucible_value(crucible_k_quality)
283                         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]
284                         crucible_l_value = self.calc_crucible_value(crucible_l_quality)
285                         delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
286
287                         if plotting:
288                             fy.append(sum(last_crucible_values) + delta)
289                             fx.append(sum(last_crucible_values))
290
291                         # if new steepest update best delta and save optimal swap location
292                         if delta > best_delta:
293                             best_delta = delta
294                             optimal_swap = (k, m, l, n)
295
296         # if all neighbors scanned and no better solution found, at local max and finish
297         if optimal_swap == (-1, -1, -1, -1):
298             if plotting:
299                 self.plot_ascent(fx, fy, "steepest_ascent_chart.pdf", "Task 3D")
300             return x

```

```

301
302     # Make swap with steepest neighbor and update intermediate values
303     k, m, l, n = optimal_swap
304     crucible_k = x[k].copy()
305     crucible_l = x[l].copy()
306     crucible_k[m] = x[l][n]
307     crucible_l[n] = x[k][m]
308     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]
309     crucible_k_value = self.calc_crucible_value(crucible_k_quality)
310     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]
311     crucible_l_value = self.calc_crucible_value(crucible_l_quality)
312     last_crucible_values[k] = crucible_k_value
313     last_crucible_values[l] = crucible_l_value
314     x[k][m] = crucible_k[m]
315     x[l][n] = crucible_l[n]
316
317
318 #####
319 # TASK 3E #
320 #####
321 def do_repeated_next_ascents(self, n: int, max_spread: int = 0, plotting=True):
322     best_obj_history = []
323     obj_history = []
324
325     times = []
326
327     # Iterate through random starts to find history and best solution
328     best_obj = 0
329     start_time = time.perf_counter()
330     for _ in range(n):
331         # If max spread specified then do with max spread (for Task 6)
332         if max_spread:
333             x = self.next_ascent_to_local_max_spread(max_spread)
334         else:
335             x = self.next_ascent_to_local_max()
336         obj = self.calc_obj(x)
337         if obj > best_obj:
338             best_x = x
339             best_obj = obj
340         best_obj_history.append(best_obj)
341         obj_history.append(obj)
342         times.append(time.perf_counter() - start_time)
343
344     # Output and plot best solution
345     print(f"repeated next ascents max_spread={max_spread}")
346     self.view_soln(best_x)
347     if plotting:
348         fig = plt.figure()
349         plt.scatter(times, best_obj_history, c='b', s=1, label="Best objective value")
350         plt.scatter(times, obj_history, c='r', s=5, label="Local max")
351         plt.xlabel('Time (s)')
352         plt.ylabel('Objective Function Value')
353         plt.legend()
354         if max_spread:
355             plt.title(f"Task 6 Repeated Next Ascents (n={n}, max_spread={max_spread})")
356         else:
357             plt.title(f"Task 3E Repeated Next Ascents (n={n})")
358         plt.gcf().set_size_inches(11.69, 8.27)
359         if max_spread:
360             plt.savefig(f"./report/assets/repeated_next_ascents_chart__max_spread_{max_spread}.pdf", orientation="landscape")
361         else:
362             plt.savefig(f"./report/assets/repeated_next_ascents_chart.pdf", orientation="landscape")

```

```

361         plt.savefig("../report/assets/repeated_steepest_ascents_chart.pdf", orientation="landscape",
362
363 #####
364 # TASK 3E #
365 #####
366 def do_repeated_steepest_ascents(self, n: int):
367     best_obj_history = []
368     obj_history = []
369     times = []
370
371     # Iterate through random starts to find history and best solution
372     best_obj = 0
373     start_time = time.perf_counter()
374     for _ in range(n):
375         x = self.steepest_ascent_to_local_max()
376         obj = self.calc_obj(x)
377         if obj > best_obj:
378             best_x = x
379             best_obj = obj
380         best_obj_history.append(best_obj)
381         obj_history.append(obj)
382         times.append(time.perf_counter() - start_time)
383
384     # Output and plot best solution
385     self.view_soln(best_x)
386     fig = plt.figure()
387     plt.scatter(times, best_obj_history, c='b', s=1, label="Best objective value")
388     plt.scatter(times, obj_history, c='r', s=5, label="Local max")
389     plt.xlabel('Time (s)')
390     plt.ylabel('Objective Function Value')
391     plt.title(f"Task 3E Repeated Steepest Ascents (n={n})")
392     plt.legend()
393     plt.gcf().set_size_inches(11.69, 8.27)
394     plt.savefig("../report/assets/repeated_steepest_ascents_chart.pdf", orientation="landscape")
395
396 #####
397 # TASK 6 #
398 #####
399 def next_ascent_to_local_max_spread(self, max_spread: int, random_start=True):
400     if random_start:
401         x = self.random_solution()
402     else:
403         x = self.trivial_solution()
404
405     # init intermeidate values
406     last_crucible_values = np.zeros(self.no_crucibles)
407     for c in range(self.no_crucibles):
408         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]
409         last_crucible_values[c] = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_spread)
410
411     # Loop through neighbors
412     last_optimal_indices = (-1, -1, -1, -1)
413     while True:
414         for k in range(self.no_crucibles-1):
415             for m in range(self.pots_per_crucible):
416                 for l in range(k+1, self.no_crucibles):
417                     for n in range(self.pots_per_crucible):
418                         # looped through all neighbors once and no better solution found
419                         if (k, m, l, n) == last_optimal_indices:
420                             return x
421

```

```

422         # calculate delta and other relevant params
423         crucible_k = x[k].copy()
424         crucible_l = x[l].copy()
425         crucible_k[m] = x[l][n]
426         crucible_l[n] = x[k][m]
427         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]
428         crucible_k_value = self.calc_crucible_value_with_spread(crucible_k_quality, np.ptp(crucible_k), max_spread)
429         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]
430         crucible_l_value = self.calc_crucible_value_with_spread(crucible_l_quality, np.ptp(crucible_l), max_spread)
431         delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]
432
433         # better solution so update intermediate values and solution
434         if delta > 0.01:
435             last_optimal_indices = (k, m, l, n)
436             last_crucible_values[k] = crucible_k_value
437             last_crucible_values[l] = crucible_l_value
438             x[k][m] = crucible_k[m]
439             x[l][n] = crucible_l[n]
440
441         # case where already at local max
442         if last_optimal_indices == (-1, -1, -1, -1):
443             return x
444
445 if __name__ == "__main__":
446     ls = LocalSearch()
447     ls.load_small_problem()
448     ls.next_ascent_to_local_max(random_start=False, plotting=True)
449     ls.steepest_ascent_to_local_max(random_start=False, plotting=True)
450     ls.load_default_problem()
451     ls.do_repeated_next_ascents(200)
452     ls.do_repeated_steepest_ascents(200)
453     ls.do_repeated_next_ascents(200, max_spread=6, plotting=False)
454     ls.do_repeated_next_ascents(200, max_spread=8, plotting=False)
455     ls.do_repeated_next_ascents(200, max_spread=11, plotting=False)

```