

```

1
2 # coding: utf-8
3
4 #
5 # ### ITNPBD8 - Evolutionary and Heuristic Optimisation Assignment
6 #
7
8 # In[1]:
9
10 # Import the libraries used in this Hill climbing algorithm
11 import time
12 import math
13 import time
14 import random
15 import numpy as np
16 import pandas as pd
17 import seaborn as sns
18 from copy import deepcopy
19 from functools import reduce
20 import matplotlib.pyplot as plt
21
22
23 # ##### Function Definitions
24
25 # In[2]:
26
27 def read_colours_file():
28
29     fileName= "100colours.txt" # ammend to 10/100/1000 colours as required
30     with open(fileName, 'r') as kfile:
31         lines = kfile.readlines()
32         no_colours_in_file = int(lines[0]) # read the first row as the number of colours
33         in the file
34         initial_solution = np.arange(no_colours_in_file) # assign a set vector as an
35         initial solution with length equal to the number of colours in the file
36         original_colour_sequence = pd.read_csv(fileName, sep=" ", skiprows=[0],
37         header=None) # read the RGB data into pandas dataframe
38         original_colour_sequence.columns = ['Red', 'Green', 'Blue'] # re-assign the column
39         names to newly constructed dataframe
40         rgb_coordinates = original_colour_sequence.to_string(index=None, header = False)
41
42         return no_colours_in_file, original_colour_sequence
43
44 # In[3]:
45
46 def initialise_a_solution():
47
48     no_colours_in_file, _ = read_colours_file() # identify the number of colours
49     defined in colours file
50     random_permutation = random.sample(range(no_colours_in_file), no_colours_in_file) #
51     create a random set vector solution of length defined above
52
53     return random_permutation
54
55 # In[4]:
56
57 def evaluate(solution):
58
59     no_colours_in_file, original_colour_sequence = read_colours_file() # identify the
60     number of colours defined in colours file and retrieve the colours dataframe
61     euclidean_distance = 0
62     for i in range(0, no_colours_in_file-1):
63         p = original_colour_sequence.iloc[solution[i]]
64         q = original_colour_sequence.iloc[solution[i+1]]
65         euclidean_distance += math.sqrt(math.pow((p['Red'] - q['Red']), 2) +

```

```

61         math.pow((p['Green'] - q['Green']), 2) + math.pow((p['Blue'] - q['Blue']), 2))
62     return euclidean_distance
63
64
65 # In[5]:
66
67 def colour_swop(solution):
68
69     solution_length = len(solution) # identify the length of the solution provided
70     random_colour_1 = random.randrange(0,solution_length) # generate random integer
71     random_colour_2 = random.randrange(0,solution_length)
72     while random_colour_2 == random_colour_1:
73         random_colour_2 = random.randrange(0,solution_length) # ensure the random
74         integers chosen are unique
75     mutated_solution = solution # assign the original solution to a new variable
76     mutated_solution[random_colour_1], mutated_solution[random_colour_2] =
77     mutated_solution[random_colour_2], mutated_solution[random_colour_1] # interchange
78     the two colours chosen
79
80     return mutated_solution
81
82 # In[6]:
83
84 def colour_invert(solution):
85
86     solution_length = len(solution) # identify the length of the solution
87     slice_start = random.randrange(0,solution_length) # create an integer for indexing
88     the start point of the slice
89     slice_end = random.randrange(slice_start,solution_length) # create an integer for
90     indexing the end point of the slice
91     mutated_solution = deepcopy(solution) # constructs a new compound object and
92     inserts copies into it of the objects found in the original
93     mutated_solution[slice_start:slice_end+1] =
94     reversed(mutated_solution[slice_start:slice_end+1])
95     while mutated_solution == solution: # repeat the inversion process if the original
96     solution order matches the mutated solution order
97         slice_start = random.randrange(0,solution_length)
98         slice_end = random.randrange(slice_start,solution_length)
99         mutated_solution = deepcopy(solution)
100         mutated_solution[slice_start:slice_end+1] =
101         reversed(mutated_solution[slice_start:slice_end+1])
102
103     return mutated_solution
104
105 # In[7]:
106
107 def distance_improvement_plot(distance_array):
108
109     no_of_iterations = np.arange(len(distance_array))
110     no_of_iterations[:] = [x + 1 for x in no_of_iterations]
111     sns.set(style='darkgrid', context='notebook')
112     plt.plot(no_of_iterations,distance_array)
113     plt.xlabel('Number of the improved solution found ', fontsize=16)
114     plt.ylabel('Distance', fontsize=14)
115     plt.show()
116
117 # In[8]:
118
119 def algorithm_comparison_plot(hill_climbing_results,local_search_results):
120     #,evolutionary_search_results):
121
122     # hill-climber
123     hill_climber_30_distances = []

```

```

117     for key, value in hill_climbing_results.items():
118         hill_climber_30_distances.append(key)
119     hill_climber_30_distances = [round(float(i), 2) for i in hill_climber_30_distances]
120     no_of_runs_hill_climber = np.arange(len(hill_climber_30_distances))
121     no_of_runs_hill_climber[:] = [x + 1 for x in no_of_runs_hill_climber]
122     mean_hill_climbing_performance = [np.mean(hill_climber_30_distances) for i in
    no_of_runs_hill_climber]
123     #print(len(hill_climber_30_distances),
    len(no_of_runs),len(mean_hill_climbing_performance))
124
125     # local search
126     local_search_30_distances = []
127     for key, value in local_search_results.items() :
128         local_search_30_distances.append(key)
129     local_search_30_distances = [round(float(i), 2) for i in local_search_30_distances]
130     no_of_runs_local_search = np.arange(len(local_search_30_distances))
131     no_of_runs_local_search[:] = [x + 1 for x in no_of_runs_local_search]
132     mean_local_search_performance = [np.mean(local_search_30_distances) for i in
    no_of_runs_local_search]
133     #print(len(local_search_30_distances),
    len(no_of_runs),len(mean_local_search_performance))
134     # evolutionary elgorithm
135     #evolutionary_30_distances = []
136     #for key, value in evolutionary_search_results.items() :
137     #    evolutionary_30_distances.append(key)
138     #evolutionary_30_distances = [round(float(i), 2) for i in evolutionary_30_distances]
139
140
141
142     #sns.set(style='darkgrid', context='notebook')
143     #fig, ax = plt.subplots()
144
145     #Plot hill climbing
146     plt.plot(no_of_runs_hill_climber,hill_climber_30_distances, 'b',label='Hill Climber
    Performance')
147     plt.plot(no_of_runs_hill_climber,mean_hill_climbing_performance, 'b-',label='Hill
    climber mean')
148
149     #Plot local search
150     plt.plot(no_of_runs_local_search,local_search_30_distances,'r', label='Local Search
    Performance')
151     plt.plot(no_of_runs_local_search,mean_local_search_performance,'r-', label='Local
    Search mean')
152
153     #Plot evolutionary algorithm
154     plt.plot(no_of_runs,current_working_population_distance_array,'p'
    label='Evolutionary Algorithm Performance')
155     plt.plot(no_of_runs,mean_evolutionary_distance,'p-' label='Evolutionary mean')
156
157
158     #plt.plot(<X AXIS VALUES HERE>, <Y AXIS VALUES HERE>, 'line type', label='label
    here')
159     #plt.plot(total_lengths, sort_times_heap, 'b-', label="Heap")
160
161
162     plt.title('Algorithm Comparison of Performance for 30 runs ',fontsize=20)
163     plt.xlabel('Number of runs', fontsize=16)
164     plt.ylabel('Final Distance from algorithm', fontsize=14)
165     plt.show()
166
167
168
169 # In[9]:
170
171 def plot_colour_band(solution):
172
173     _,original_colour_sequence = read_colours_file() # load the original vector index

```

```

174 colours = original_colour_sequence.reindex(solution) # re-order the colours in the
175 dataframe with the new vector (solution) index
176 ratio = 10 # ratio of line height/width, e.g. colour lines will have height 10 and
177 width 1
178 img = np.zeros((ratio, len(solution), 3))
179 for i in range(0, len(colours)):
180     img[:, i, :] = colours.iloc[i]
181 fig, axes = plt.subplots(1, figsize=(10,2)) # figsize=(width,height) handles window
182 dimensions
183 axes.imshow(img, interpolation='nearest')
184 axes.axis('off')
185 plt.show()
186
187 # In[10]:
188
189 def hill_climber():
190     random_solution = initialise_a_solution()
191     _,original_colour_sequence = read_colours_file()
192     current_working_solution = random_solution
193     current_working_distance = evaluate(current_working_solution)
194     single_hill_climber_improvements = []
195     single_hill_climber_history = []
196     for i in range(0, 100):
197         neighbourhood_solution = colour_invert(current_working_solution)
198         neighbourhood_solution_distance = evaluate(neighbourhood_solution)
199         single_hill_climber_history.append(neighbourhood_solution_distance)
200         if neighbourhood_solution_distance > current_working_distance:
201             neighbourhood_solution = colour_invert(neighbourhood_solution)
202             neighbourhood_solution_distance = evaluate(neighbourhood_solution)
203         elif neighbourhood_solution_distance < current_working_distance:
204             current_working_distance = neighbourhood_solution_distance
205             current_working_solution = neighbourhood_solution
206             single_hill_climber_improvements.append(current_working_distance)
207         if len(original_colour_sequence) == 10 and current_working_distance < 5.62:
208             break
209         elif len(original_colour_sequence) == 100 and current_working_distance < 43.12:
210             break
211         elif len(original_colour_sequence) == 1000 and current_working_distance <
212             682.32:
213             break
214     return current_working_solution
215
216 # In[11]:
217
218 def hill_climbing_data():
219     start = time.time()
220     hill_climber_solution_array = []
221     hill_climber_distance_array = []
222     print("Hill climbing commencing:\n")
223     hill_climbing_results = {}
224     for x in range(0,30) : # run algorithm 30 times to evaluate a general algorithm
225         performance
226         best_hill_climber_solution = hill_climber()
227         best_hill_climber_distance = evaluate(best_hill_climber_solution)
228         hill_climber_solution_array.append(best_hill_climber_solution)
229         hill_climber_distance_array.append(best_hill_climber_distance)
230         print("run %s done" %(x+1))
231     hill_climbing_results = dict(zip(hill_climber_distance_array,
232         hill_climber_solution_array))
233     hill_climbing_best_solution_distance = min(hill_climbing_results, key=float)
234     hill_climbing_best_solution =
235     hill_climbing_results[hill_climbing_best_solution_distance]

```

```

234     print ("The Hill-climbing algorithm yielded an array of %s distances, with a mean
of %.2f, standard deviation of %.2f and minimum distance of %.2f "
%(x+1,np.mean(hill_climber_distance_array),
np.std(hill_climber_distance_array),hill_climbing_best_solution_distance))
235 end = time.time()
236 print("Time elapsed:",end - start)
237
238     return hill_climbing_results, hill_climbing_best_solution
239
240
241 # In[12]:
242
243 def colour_inversion_perubation(solution):
244
245     solution_swopped_once = colour_invert(solution)
246     solution_swopped_twice = colour_invert(solution_swopped_once) # additional
perubation operator to add momentum and escape local minima
247
248     return solution_swopped_twice
249
250
251 # In[13]:
252
253 def hill_climber_iterated_local(solution):
254
255     _,original_colour_sequence = read_colours_file()
256     current_working_solution = solution
257     current_working_distance = evaluate(current_working_solution)
258     for i in range(0, 10):
259         neighbourhood_solution = colour_invert(current_working_solution)
260         neighbourhood_solution_distance = evaluate(neighbourhood_solution)
261         if neighbourhood_solution_distance >= current_working_distance:
262             neighbourhood_solution = colour_invert(neighbourhood_solution)
263             neighbourhood_solution_distance = evaluate(neighbourhood_solution)
264         elif neighbourhood_solution_distance < current_working_distance:
265             current_working_distance = neighbourhood_solution_distance
266             current_working_solution = neighbourhood_solution
267         if len(original_colour_sequence) == 10 and current_working_distance < 5.62:
268             break
269         elif len(original_colour_sequence) == 100 and current_working_distance < 43.12:
270             break
271         elif len(original_colour_sequence) == 1000 and current_working_distance <
682.32:
272             break
273
274     return current_working_solution
275
276
277 # In[14]:
278
279 def iterated_local_search():
280
281     _,original_colour_sequence = read_colours_file()
282     random_solution = initialise_a_solution()
283     hill_climbing_best_solution = hill_climber_iterated_local(random_solution)
284     hill_climbing_distance = evaluate(hill_climbing_best_solution)
285     current_working_solution = hill_climbing_best_solution
286     current_working_distance = hill_climbing_distance
287     local_search_distance_array = []
288     for i in range(0, 20):
289         perturbed_solution = colour_inversion_perubation(current_working_solution)
290         hill_climbing_on_perturbed_solution =
hill_climber_iterated_local(perturbed_solution)
291         hill_climbing_perturbed_distance = evaluate(hill_climbing_on_perturbed_solution)
292         if hill_climbing_perturbed_distance < current_working_distance:
293             current_working_distance = hill_climbing_perturbed_distance
294             current_working_solution = hill_climbing_on_perturbed_solution

```

```

295         local_search_distance_array.append(current_working_distance)
296     if len(original_colour_sequence) == 10 and current_working_distance < 5.62:
297         break
298     elif len(original_colour_sequence) == 100 and current_working_distance < 43.12:
299         break
300     elif len(original_colour_sequence) == 1000 and current_working_distance <
301         682.32:
302         break
303     return current_working_solution
304
305 # In[15]:
306
307 def iterated_local_search_data():
308
309     start = time.time()
310     local_search_distance_array = []
311     local_search_solution_array = []
312     print("\nIterated local search commencing:\n")
313     for x in range(0,30):
314         best_local_search_solution = iterated_local_search()
315         best_local_search_distance = evaluate(best_local_search_solution)
316         local_search_solution_array.append(best_local_search_solution)
317         local_search_distance_array.append(best_local_search_distance)
318         print("run %s done" %(x+1))
319     local_search_results = dict(zip(local_search_distance_array,
320     local_search_solution_array))
321     best_local_search_solution_distance = min(local_search_results, key=float)
322     local_search_best_solution =
323     local_search_results[best_local_search_solution_distance]
324     print ("The Local-search algorithm yielded an array of %s distances, with a mean of
325     %.2f, standard deviation of %.2f and minimum distance of %.2f "
326     %(x+1,np.mean(local_search_distance_array),
327     np.std(local_search_distance_array),best_local_search_solution_distance ))
328     end = time.time()
329     print("Time elapsed:",end - start)
330
331     return local_search_results, local_search_best_solution
332
333 # In[16]:
334
335 def generate_and_evaluate_population():
336
337     population_size = 20
338     population = []
339     population_distance= []
340     for i in range(0,population_size):
341         random_solution = initialise_a_solution()
342         population.append(random_solution)
343         distance = evaluate(random_solution)
344         population_distance.append(distance)
345
346     return population, population_distance
347
348 # In[17]:
349
350 def tournamentSelection(population):
351
352     selection_one = population[random.randint(0,len(population)-1)]
353     selection_two = population[random.randint(0,len(population)-1)]
354     while selection_one == selection_two:
355         selection_two = population[random.randint(0,len(population)-1)]
356     distance_one = evaluate(selection_one)
357     distance_two = evaluate(selection_two)

```

```

356
357     if distance_one > distance_two:
358         return selection_two
359     else:
360         return selection_one
361
362
363 # In[18]:
364
365 def one_point_recombination(solution_one, solution_two):
366
367     size = min(len(solution_one), len(solution_two))
368     a, b = random.sample(range(size), 2)
369     if a > b:
370         a, b = b, a
371     placeholder_one, placeholder_two = [True]*size, [True]*size
372     for i in range(size):
373         if i < a or i > b:
374             placeholder_one[solution_two[i]] = False
375             placeholder_two[solution_one[i]] = False
376     temp_holder_one, temp_holder_two = solution_one, solution_two
377     k1, k2 = b + 1, b + 1
378     for i in range(size):
379         if not placeholder_one[temp_holder_one[(i + b + 1) % size]]:
380             solution_one[k1 % size] = temp_holder_one[(i + b + 1) % size]
381             k1 += 1
382         if not placeholder_two[temp_holder_two[(i + b + 1) % size]]:
383             solution_two[k2 % size] = temp_holder_two[(i + b + 1) % size]
384             k2 += 1
385     for i in range(a, b + 1):
386         solution_one[i], solution_two[i] = solution_two[i], solution_one[i]
387
388     return solution_one, solution_two
389
390
391 # In[19]:
392
393 def worst(population, population_distances, mutated_child_one, mutated_child_two):
394
395     # Replace the worst individual from population
396     worst_distance = max(population_distances)
397     worst_solution = population_distances.index(worst_distance)
398     population[worst_solution] = mutated_child_one
399     distance_new_child_one = evaluate(mutated_child_one)
400     population_distances[worst_solution] = distance_new_child_one
401     # Replace the second-worst individual from population
402     second_worst_distance = max(population_distances)
403     second_worst_solution = population_distances.index(second_worst_distance)
404     population[second_worst_solution] = mutated_child_two
405     distance_new_child_two = evaluate(mutated_child_two)
406     population_distances[second_worst_solution] = distance_new_child_two
407     #average_distance = sum(population_distances)/len(population)
408
409     return population, population_distances
410
411
412 # In[20]:
413
414 def evolutionary_algorithm():
415
416     _,original_colour_sequence = read_colours_file()
417     # Generate initial population and distance
418     current_working_population, current_working_population_distance_array =
419     generate_and_evaluate_population()
420     for x in range(0,30):
421         mom = tournamentSelection(current_working_population)
422         dad = tournamentSelection(current_working_population)

```

```

422     child_one , child_two = one_point_recombination(mom, dad)
423     mutated_child_one,mutated_child_two = colour_invert(mom),colour_invert(dad)
424     current_working_population, current_working_population_distances =
        worst(current_working_population, current_working_population_distance_array,
        mutated_child_one, mutated_child_two)
425     if len(original_colour_sequence) == 10 and
        np.mean(current_working_population_distances) < 5.62:
426         break
427     elif len(original_colour_sequence) == 100 and
        max(current_working_population_distances) < 43.12:
428         break
429     elif len(original_colour_sequence) == 1000 and
        max(current_working_population_distances) < 682.32:
430         break
431     evolutionary_results = dict(zip(current_working_population_distances,
        current_working_population))
432     best_evolutionary_solution_distance = min(evolutionary_results, key=float)
433     best_evolutionary_solution =
        evolutionary_results[best_evolutionary_solution_distance]
434
435     return best_evolutionary_solution_distance,best_evolutionary_solution
436
437
438 # In[21]:
439
440 def evolutionary_algorithm_data():
441
442     start = time.time()
443     # Run evolutionary_algorithm() for 30 runs
444     evolutionary_algorithm_solution_array = []
445     evolutionary_algorithm_distance_array = []
446     print("\nEvolutionary steady-state algorithm executing below:\n")
447     for x in range(0,30):
448         best_evolutionary_solution_distance,best_evolutionary_solution =
            evolutionary_algorithm()
449         evolutionary_algorithm_distance_array.append(best_evolutionary_solution_distance)
450         evolutionary_algorithm_solution_array.append(best_evolutionary_solution)
451         print("run %s done" %(x+1))
452     evolutionary_search_results = dict(zip(evolutionary_algorithm_distance_array,
        evolutionary_algorithm_solution_array))
453     best_evolutionary_solution_key = min(evolutionary_search_results, key=float)
454     best_evolutionary_solution =
        evolutionary_search_results[best_evolutionary_solution_key]
455     print ("The steady-state Evolutionary algorithms yielded an array of %s distances,
        with a mean of %.2f, standard deviation of %.2f and best solution distance of %.2f"
        %(x+1,np.mean(evolutionary_algorithm_distance_array),
        np.std(evolutionary_algorithm_distance_array),best_evolutionary_solution_key ))
456     end = time.time()
457     print("Time elapsed:",end - start)
458
459     return evolutionary_search_results, best_evolutionary_solution
460
461
462 # In[22]:
463
464 ## *****      MAIN      ***** #
465
466 # call individual or all algorithms to be run
467 #hill_climber_results, hill_climbing_best_solution = hill_climbing_data()
468 #local_search_results, local_search_solution = iterated_local_search_data()
469 evolutionary_algorithm_results, best_evolutionary_solution =
    evolutionary_algorithm_data()
470
471 # plot the history of all neighbourhood solutions found or best neighbourhood solutions
    found
472 #distance_improvement_plot(hill_climber_distance_array)
473 #distance_improvement_plot(iterated_local_search_array)

```



```
474 #distance_improvement_plot(evolutionary_algorithm_array)
475
476 # plot the solution as a colour band
477 #random_solution = initialise_a_solution()
478 #plot_colour_band(random_solution)
479 #plot_colour_band(hill_climbing_best_solution)
480 #plot_colour_band(local_search_solution)
481 plot_colour_band(best_evolutionary_solution)
482
483 # plot the comparison of algorithms
484 #algorithm_comparison_plot(hill_climber_results,local_search_results) #
,evolutionary_algorithm_results)
485
486
487 # In[ ]:
488
489
490
491
```