```python

# coding: utf-8

#
# ### ITNPBD8 - Evolutionary and Heuristic Optimisation Assignment
#

# In[1]:

# Import the libraries used in this Hill climbing algorithm
import time
import math
import time
import random
import numpy as np
import pandas as pd
import seaborn as sns
from copy import deepcopy
from functools import reduce
import matplotlib.pyplot as plt


# ##### Function Definitions

# In[2]:

def read_colours_file():

    fileName= "100colours.txt" # ammend to 10/100/1000 colours as required
    with open(fileName, 'r') as kfile:
        lines = kfile.readlines()
    no_colours_in_file = int(lines[0])  # read the first row as the number of colours in the file
    initial_solution = np.arange(no_colours_in_file) # assign a set vector as an initial solution with length equal to the number of colours in the file
    origional_colour_sequence = pd.read_csv(fileName, sep=" ", skiprows=[0], header=None) # read the RGB data into pandas dataframe
    origional_colour_sequence.columns = ['Red', 'Green', 'Blue'] # re-assign the column names to newly constructed dataframe
    rgb_coordinates = origional_colour_sequence.to_string(index=None, header = False)

    return no_colours_in_file, origional_colour_sequence


# In[3]:

def initialise_a_solution():

    no_colours_in_file, _ =  read_colours_file() # identify the number of colours defined in colours file
    random_permutation = random.sample(range(no_colours_in_file), no_colours_in_file) # create a random set vector solution of length defined above

    return random_permutation


# In[4]:

def evaluate(solution):

    no_colours_in_file, origional_colour_sequence =  read_colours_file() # identify the number of colours defined in colours file and retrieve the colours dataframe
    euclidean_distance = 0
    for i in range(0, no_colours_in_file-1):
        p = origional_colour_sequence.iloc[solution[i]]
        q = origional_colour_sequence.iloc[solution[i+1]]
        euclidean_distance += math.sqrt(math.pow((p['Red'] - q['Red']), 2) +
```

```
        math.pow((p['Green'] - q['Green']), 2) + math.pow((p['Blue'] - q['Blue']), 2))

    return euclidean_distance


# In[5]:

def colour_swop(solution):

    solution_length = len(solution) # identify the length of the solution provided
    random_colour_1 = random.randrange(0,solution_length) # generate random integer
    random_colour_2 = random.randrange(0,solution_length)
    while random_colour_2 == random_colour_1:
        random_colour_2 = random.randrange(0,solution_length) # ensure the random
        integers chosen are unique
    mutated_solution = solution # assign the original solution to a new variable
    mutated_solution[random_colour_1], mutated_solution[random_colour_2] =
    mutated_solution[random_colour_2], mutated_solution[random_colour_1] # interchange
    the two colours chosen

    return mutated_solution


# In[6]:

def colour_invert(solution):

    solution_length = len(solution) # identify the length of the solution
    slice_start = random.randrange(0,solution_length) # create an integer for indexing
    the start point of the slice
    slice_end = random.randrange(slice_start,solution_length) # create an integer for
    indexing the end point of the slice
    mutated_solution = deepcopy(solution) # constructs a new compound object and
    inserts copies into it of the objects found in the original
    mutated_solution[slice_start:slice_end+1] =
    reversed(mutated_solution[slice_start:slice_end+1])
    while mutated_solution == solution: # repeat the inversion process if the original
    solution order matches the mutated solution order
        slice_start = random.randrange(0,solution_length)
        slice_end = random.randrange(slice_start,solution_length)
        mutated_solution = deepcopy(solution)
        mutated_solution[slice_start:slice_end+1] =
        reversed(mutated_solution[slice_start:slice_end+1])

    return mutated_solution


# In[7]:

def distance_improvement_plot(distance_array):

    no_of_iterations = np.arange(len(distance_array))
    no_of_iterations[:] = [x + 1 for x in no_of_iterations]
    sns.set(style='darkgrid', context='notebook')
    plt.plot(no_of_iterations,distance_array)
    plt.xlabel('Number of the improved solution found ', fontsize=16)
    plt.ylabel('Distance', fontsize=14)
    plt.show()


# In[8]:

def algorithm_comparison_plot(hill_climbing_results,local_search_results):
#,evolutionary_search_results):

    # hill-climber
    hill_climber_30_distances = []
```

```
    for key, value in hill_climbing_results.items():
        hill_climber_30_distances.append(key)
    hill_climber_30_distances = [round(float(i), 2) for i in hill_climber_30_distances]
    no_of_runs_hill_climber = np.arange(len(hill_climber_30_distances))
    no_of_runs_hill_climber[:] = [x + 1 for x in no_of_runs_hill_climber]
    mean_hill_climbing_performance = [np.mean(hill_climber_30_distances) for i in no_of_runs_hill_climber]
    #print(len(hill_climber_30_distances), len(no_of_runs),len(mean_hill_climbing_performance))

    # local search
    local_search_30_distances = []
    for key, value in local_search_results.items() :
        local_search_30_distances.append(key)
    local_search_30_distances = [round(float(i), 2) for i in local_search_30_distances]
    no_of_runs_local_search = np.arange(len(local_search_30_distances))
    no_of_runs_local_search[:] = [x + 1 for x in no_of_runs_local_search]
    mean_local_search_performance = [np.mean(local_search_30_distances) for i in no_of_runs_local_search]
    #print(len(local_search_30_distances), len(no_of_runs),len(mean_local_search_performance))
    # evolutionary elgorithm
    #evolutionary_30_distances = []
    #for key, value in evolutionary_search_results.items() :
    #    evolutionary_30_distances.append(key)
    #evolutionary_30_distances = [round(float(i), 2) for i in evolutionary_30_distances]



    #sns.set(style='darkgrid', context='notebook')
    #fig, ax = plt.subplots()

    #Plot hill climbing
    plt.plot(no_of_runs_hill_climber,hill_climber_30_distances, 'b',label='Hill Climber Performance')
    plt.plot(no_of_runs_hill_climber,mean_hill_climbing_performance, 'b-',label='Hill climber mean')

    #Plot local search
    plt.plot(no_of_runs_local_search,local_search_30_distances,'r', label='Local Search Performance')
    plt.plot(no_of_runs_local_search,mean_local_search_performance,'r-', label='Local Search mean')

    #Plot evolutionary algorithm
    #plt.plot(no_of_runs,current_working_population_distance_array,'p' label='Evolutionary Algorithm Performance')
    #plt.plot(no_of_runs,mean_evolutionary_distance,'p-' label='Evolutionary mean')


    #plt.plot(<X AXIS VALUES HERE>, <Y AXIS VALUES HERE>, 'line type', label='label here')
    #plt.plot(total_lengths, sort_times_heap, 'b-', label="Heap")


    plt.title('Algorithm Comparison of Performance for 30 runs ',fontsize=20)
    plt.xlabel('Number of runs', fontsize=16)
    plt.ylabel('Final Distance from algorithm', fontsize=14)
    plt.show()



# In[9]:

def plot_colour_band(solution):

    _,origional_colour_sequence = read_colours_file() # load the origional vector index
```

```python
    colours = origional_colour_sequence.reindex(solution) # re-order the colours in the dataframe with the new vector (solution) index
    ratio = 10 # ratio of line height/width, e.g. colour lines will have height 10 and width 1
    img = np.zeros((ratio, len(solution), 3))
    for i in range(0, len(colours)):
        img[:, i, :] = colours.iloc[i]
    fig, axes = plt.subplots(1, figsize=(10,2)) # figsize=(width,height) handles window dimensions
    axes.imshow(img, interpolation='nearest')
    axes.axis('off')
    plt.show()


# In[10]:

def hill_climber():

    random_solution = initialise_a_solution()
    _,origional_colour_sequence = read_colours_file()
    current_working_solution = random_solution
    current_working_distance = evaluate(current_working_solution)
    single_hill_climber_improvements = []
    single_hill_climber_history = []
    for i in range(0, 100):
        neighbourhood_solution = colour_invert(current_working_solution)
        neighbourhood_solution_distance = evaluate(neighbourhood_solution)
        single_hill_climber_history.append(neighbourhood_solution_distance)
        if neighbourhood_solution_distance > current_working_distance:
            neighbourhood_solution = colour_invert(neighbourhood_solution)
            neighbourhood_solution_distance = evaluate(neighbourhood_solution)
        elif neighbourhood_solution_distance < current_working_distance:
            current_working_distance = neighbourhood_solution_distance
            current_working_solution = neighbourhood_solution
            single_hill_climber_improvements.append(current_working_distance)
        if len(origional_colour_sequence) == 10 and current_working_distance < 5.62:
            break
        elif len(origional_colour_sequence) == 100 and current_working_distance < 43.12:
            break
        elif len(origional_colour_sequence) == 1000 and current_working_distance < 682.32:
            break

    return current_working_solution


# In[11]:

def hill_climbing_data():

    start = time.time()
    hill_climber_solution_array = []
    hill_climber_distance_array = []
    print("Hill climbing commencing:\n")
    hill_climbing_results = {}
    for x in range(0,30) : # run algorithm 30 times to evaluate a general algorithm performance
        best_hill_climber_solution = hill_climber()
        best_hill_climber_distance = evaluate(best_hill_climber_solution)
        hill_climber_solution_array.append(best_hill_climber_solution)
        hill_climber_distance_array.append(best_hill_climber_distance)
        print("run %s done" %(x+1))
    hill_climbing_results = dict(zip(hill_climber_distance_array, hill_climber_solution_array))
    hill_climbing_best_solution_distance = min(hill_climbing_results, key=float)
    hill_climbing_best_solution = hill_climbing_results[hill_climbing_best_solution_distance]
```

```python
    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))
    end = time.time()
    print("Time elapsed:",end - start)

    return hill_climbing_results, hill_climbing_best_solution


# In[12]:

def colour_inversion_perubation(solution):

    solution_swopped_once = colour_invert(solution)
    solution_swopped_twice = colour_invert(solution_swopped_once) # additional perubation operator to add momentum and escape local minima

    return solution_swopped_twice


# In[13]:

def hill_climber_iterated_local(solution):

    _,origional_colour_sequence = read_colours_file()
    current_working_solution = solution
    current_working_distance = evaluate(current_working_solution)
    for i in range(0, 10):
        neighbourhood_solution = colour_invert(current_working_solution)
        neighbourhood_solution_distance = evaluate(neighbourhood_solution)
        if neighbourhood_solution_distance >= current_working_distance:
            neighbourhood_solution = colour_invert(neighbourhood_solution)
            neighbourhood_solution_distance = evaluate(neighbourhood_solution)
        elif neighbourhood_solution_distance < current_working_distance:
            current_working_distance = neighbourhood_solution_distance
            current_working_solution = neighbourhood_solution
        if len(origional_colour_sequence) == 10 and current_working_distance < 5.62:
            break
        elif len(origional_colour_sequence) == 100 and current_working_distance < 43.12:
            break
        elif len(origional_colour_sequence) == 1000 and current_working_distance < 682.32:
            break

    return current_working_solution


# In[14]:

def iterated_local_search():

    _,origional_colour_sequence = read_colours_file()
    random_solution = initialise_a_solution()
    hill_climbing_best_solution = hill_climber_iterated_local(random_solution)
    hill_climbing_distance = evaluate(hill_climbing_best_solution)
    current_working_solution = hill_climbing_best_solution
    current_working_distance = hill_climbing_distance
    local_search_distance_array =[]
    for i in range(0, 20):
        perturbed_solution = colour_inversion_perubation(current_working_solution)
        hill_climbing_on_perturbed_solution = hill_climber_iterated_local(perturbed_solution)
        hill_climbing_perturbed_distance = evaluate(hill_climbing_on_perturbed_solution)
        if hill_climbing_perturbed_distance < current_working_distance:
            current_working_distance = hill_climbing_perturbed_distance
            current_working_solution = hill_climbing_on_perturbed_solution
```

```
                local_search_distance_array.append(current_working_distance)
            if len(origional_colour_sequence) == 10 and current_working_distance < 5.62:
                break
            elif len(origional_colour_sequence) == 100 and current_working_distance < 43.12:
                break
            elif len(origional_colour_sequence) == 1000 and current_working_distance <
            682.32:
                break

        return current_working_solution


    # In[15]:

    def iterated_local_search_data():

        start = time.time()
        local_search_distance_array = []
        local_search_solution_array = []
        print("\nIterated local search commencing:\n")
        for x in range(0,30):
            best_local_search_solution = iterated_local_search()
            best_local_search_distance = evaluate(best_local_search_solution)
            local_search_solution_array.append(best_local_search_solution)
            local_search_distance_array.append(best_local_search_distance)
            print("run %s done" %(x+1))
        local_search_results = dict(zip(local_search_distance_array,
        local_search_solution_array))
        best_local_search_solution_distance = min(local_search_results, key=float)
        local_search_best_solution =
        local_search_results[best_local_search_solution_distance]
        print ("The Local-search 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(local_search_distance_array),
        np.std(local_search_distance_array),best_local_search_solution_distance ))
        end = time.time()
        print("Time elapsed:",end - start)

        return local_search_results, local_search_best_solution


    # In[16]:

    def generate_and_evaluate_population():

        population_size = 20
        population = []
        population_distance= []
        for i in range(0,population_size):
            random_solution = initialise_a_solution()
            population.append(random_solution)
            distance = evaluate(random_solution)
            population_distance.append(distance)

        return population, population_distance


    # In[17]:

    def tournamentSelection(population):

        selection_one = population[random.randint(0,len(population)-1)]
        selection_two = population[random.randint(0,len(population)-1)]
        while selection_one == selection_two:
            selection_two = population[random.randint(0,len(population)-1)]
        distance_one = evaluate(selection_one)
        distance_two = evaluate(selection_two)
```

```python

    if distance_one > distance_two:
        return selection_two
    else:
        return selection_one


# In[18]:

def one_point_recombination(solution_one, solution_two):

    size = min(len(solution_one), len(solution_two))
    a, b = random.sample(range(size), 2)
    if a > b:
        a, b = b, a
    placeholder_one, placeholder_two = [True]*size, [True]*size
    for i in range(size):
        if i < a or i > b:
            placeholder_one[solution_two[i]] = False
            placeholder_two[solution_one[i]] = False
    temp_holder_one, temp_holder_two = solution_one, solution_two
    k1 , k2 = b + 1, b + 1
    for i in range(size):
        if not placeholder_one[temp_holder_one[(i + b + 1) % size]]:
            solution_one[k1 % size] = temp_holder_one[(i + b + 1) % size]
            k1 += 1
        if not placeholder_two[temp_holder_two[(i + b + 1) % size]]:
            solution_two[k2 % size] = temp_holder_two[(i + b + 1) % size]
            k2 += 1
    for i in range(a, b + 1):
        solution_one[i], solution_two[i] = solution_two[i], solution_one[i]

    return solution_one, solution_two


# In[19]:

def worst(population, population_distances, mutated_child_one, mutated_child_two):

    # Replace the worst individual from population
    worst_distance = max(population_distances)
    worst_solution = population_distances.index(worst_distance)
    population[worst_solution] = mutated_child_one
    distance_new_child_one = evaluate(mutated_child_one)
    population_distances[worst_solution] = distance_new_child_one
    # Replace the second-worst individual from population
    second_worst_distance = max(population_distances)
    second_worst_solution = population_distances.index(second_worst_distance)
    population[second_worst_solution] = mutated_child_two
    distance_new_child_two = evaluate(mutated_child_two)
    population_distances[second_worst_solution] = distance_new_child_two
    #average_distance = sum(population_distances)/len(population)

    return population, population_distances


# In[20]:

def evolutionary_algorithm():

    _,origional_colour_sequence = read_colours_file()
    # Generate initial_population and distance
    current_working_population, current_working_population_distance_array =
    generate_and_evaluate_population()
    for x in range(0,30):
        mom = tournamentSelection(current_working_population)
        dad = tournamentSelection(current_working_population)
```

```
        child_one , child_two = one_point_recombination(mom, dad)
        mutated_child_one,mutated_child_two = colour_invert(mom),colour_invert(dad)
        current_working_population, current_working_population_distances  = worst(current_working_population, current_working_population_distance_array, mutated_child_one, mutated_child_two)
        if len(origional_colour_sequence) == 10 and np.mean(current_working_population_distances) < 5.62:
            break
        elif len(origional_colour_sequence) == 100 and max(current_working_population_distances)  < 43.12:
            break
        elif len(origional_colour_sequence) == 1000 and max(current_working_population_distances)  < 682.32:
            break
    evolutionary_results = dict(zip(current_working_population_distances, current_working_population))
    best_evolutionary_solution_distance = min(evolutionary_results, key=float)
    best_evolutionary_solution = evolutionary_results[best_evolutionary_solution_distance]

    return best_evolutionary_solution_distance,best_evolutionary_solution


# In[21]:

def evolutionary_algorithm_data():

    start = time.time()
    # Run evolutionary_algorithm() for 30 runs
    evolutionary_algorithm_solution_array = []
    evolutionary_algorithm_distance_array = []
    print("\nEvolutionary steady-state algorithm executing below:\n")
    for x in range(0,30):
        best_evolutionary_solution_distance,best_evolutionary_solution = evolutionary_algorithm()
        evolutionary_algorithm_distance_array.append(best_evolutionary_solution_distance)
        evolutionary_algorithm_solution_array.append(best_evolutionary_solution)
        print("run %s done" %(x+1))
    evolutionary_search_results = dict(zip(evolutionary_algorithm_distance_array, evolutionary_algorithm_solution_array))
    best_evolutionary_solution_key = min(evolutionary_search_results, key=float)
    best_evolutionary_solution = evolutionary_search_results[best_evolutionary_solution_key]
    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 ))
    end = time.time()
    print("Time elapsed:",end - start)

    return evolutionary_search_results, best_evolutionary_solution


# In[22]:

## **************    MAIN  ************** #

# call individual or all algorithms to be run
#hill_climber_results, hill_climbing_best_solution = hill_climbing_data()
#local_search_results, local_search_solution = iterated_local_search_data()
evolutionary_algorithm_results, best_evolutionary_solution = evolutionary_algorithm_data()

# plot the history of all neighbourhood solutions found or best neighbourhood solutions found
#distance_improvement_plot(hill_climber_distance_array)
#distance_improvement_plot(iterated_local_search_array)
```

```
#distance_improvement_plot(evolutionary_algorithm_array)

# plot the solution as a colour band
#random_solution = initialise_a_solution()
#plot_colour_band(random_solution)
#plot_colour_band(hill_climbing_best_solution)
#plot_colour_band(local_search_solution)
plot_colour_band(best_evolutionary_solution)

# plot the comparison of algorithms
#algorithm_comparison_plot(hill_climber_results,local_search_results) #
,evolutionary_algorithm_results)


# In[ ]:
```