```python
# (C) Andrew Mason 2023 - ENGSCI 760 Heuristics Assignment
# This code, and any code derived from this, may NOT be posted in any publicly accessible location
# Specifically, this code, and any derived versions of this code (including your assignment answers)
# must NOT be posted publically on Github, Gitlab or similar.

import numpy as np
import matplotlib.pyplot as plt
from enum import IntEnum
import time
import random

class Element(IntEnum):
    """The elements that we measure levels of in the Aluminium we produce"""
    Al = 0
    Fe = 1
    Si = 2

class LocalSearch():
    def __init__(self) -> None:
        self.load_default_problem()

    def load_default_problem(self) -> None:
        """Initialise the configuration parameters with default values"""
        self.no_crucibles=17
        self.no_pots=51
        self.pots_per_crucible=3
        # Initialise the percentage of Al (aluminium), Fe (iron) and Silicon (Si)
        self.pot_quality = np.array(
                           [ [99.136, 0.051, 0.497],
                            [99.733, 0.064, 0.138],
                            [99.755, 0.083, 0.149],
                            [99.198, 0.318, 0.206],
                            [99.297, 0.284, 0.33],
                            [99.23, 0.327, 0.393],
                            [99.485, 0.197, 0.156],
                            [99.709, 0.011, 0.056],
                            [99.729, 0.007, 0.012],
                            [99.118, 0.434, 0.377],
                            [99.372, 0.01, 0.349],
                            [99.505, 0.028, 0.433],
                            [99.187, 0.296, 0.335],
                            [99.043, 0.224, 0.531],
                            [99.206, 0.166, 0.146],
                            [99.395, 0.188, 0.328],
                            [99.436, 0.199, 0.303],
                            [99.796, 0.009, 0.144],
                            [99.186, 0.397, 0.065],
                            [99.455, 0.079, 0.278],
                            [99.553, 0.084, 0.353],
                            [99.539, 0.017, 0.201],
                            [99.38, 0.082, 0.239],
                            [99.504, 0.009, 0.273],
                            [99.391, 0.261, 0.297],
                            [99.374, 0.015, 0.578],
                            [99.462, 0.179, 0.109],
                            [99.03, 0.213, 0.459],
                            [99.328, 0.131, 0.371],
                            [99.674, 0.055, 0.249],
                            [99.413, 0.137, 0.1],
                            [99.538, 0.046, 0.151],
                            [99.41, 0.109, 0.08],
                            [99.163, 0.324, 0.343],
                            [99.502, 0.036, 0.412],
                            [99.66, 0.083, 0.069],
                            [99.629, 0.156, 0.069],
                            [99.592, 0.171, 0.008],
                            [99.684, 0.011, 0.106],
                            [99.358, 0.227, 0.137],
                            [99.145, 0.161, 0.403],
                            [99.729, 0.028, 0.123],
                            [99.335, 0.181, 0.351],
                            [99.725, 0.094, 0.14],
                            [99.124, 0.325, 0.015],
                            [99.652, 0.068, 0.029],
                            [99.091, 0.268, 0.565],
                            [99.426, 0.146, 0.256],
                            [99.383, 0.266, 0.039],
                            [99.481, 0.147, 0.327],
                            [99.163, 0.121, 0.71] ] )
        # Initialise the impurity limits & dolar values associated with the different quality grades of Al (aluminium)
        # We require at least a minimum % Al, and no more than max Fe (iron) and Si (Silicon) %'s
        self.no_grades = 11
        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]
        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]
        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]
        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]

    def load_small_problem(self) -> None:
        """Intialise the configuration parameters with default values, and then modify the sizing to give a smaller problem with 10 crucibles"""
        self.load_default_problem()
        self.no_crucibles=10
        self.no_pots=self.no_crucibles * self.pots_per_crucible

    def calc_crucible_value(self, crucible_quality) -> float:
        """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
           Returns 0 if the aluminium does not satisfy any of the quality grades."""
        tol = 0.00001 # We allow for small errors in 5th decimal point
        for q in reversed(range(self.no_grades)):
            if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
               crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
               crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
                return self.grade_value[q]
        return 0.0

    def calc_crucible_value_with_spread(self, crucible_quality, spread: int, max_spread: int) -> float:
        """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
           Returns 0 if the aluminium does not satisfy any of the quality grades."""
        tol = 0.00001 # We allow for small errors in 5th decimal point
        spread_penalty = -10000*(spread - max_spread) if spread > max_spread else 0
        for q in reversed(range(self.no_grades)):
            if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
               crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
               crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
                return self.grade_value[q] + spread_penalty
        return 0.0

    def view_soln(self, x, max_allowed_spread: int=0) -> None:
        """Print solution x with its statistics. Note that our output numbers items from 1, not 0"""
```

```python
        max_spread = 0
        crucible_value_sum = 0
        for c in range (self.no_crucibles):
            spread = max(x[c]) - min(x[c])
            max_spread = max(max_spread, spread)
            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]
            if max_allowed_spread:
                crucible_value = self.calc_crucible_value_with_spread(crucible_quality, spread, max_allowed_spread)
            else:
                crucible_value = self.calc_crucible_value(crucible_quality)

            crucible_value_sum += crucible_value
            print(f'{c+1:>2} [{x[c][0]+1:>2} {x[c][1]+1:>2} {x[c][2]+1:>2} ] '
                  f'{crucible_quality[Element.Al]:>5.3f} %Al, '
                  f'{crucible_quality[Element.Fe]:>5.3f} %Fe, '
                  f'{crucible_quality[Element.Si]:>5.3f} %Si, '
                  f'${crucible_value:>5.2f}, spread = {spread:>2}' )
        print(f'                                         Sum = ${round(crucible_value_sum,2):>6}, MxSprd = {max_spread:>2}')

    def calc_obj(self, x, max_allowed_spread: int=0):
        """Calculate the total profit for a given solution"""
        crucible_value_sum = 0
        for c in range (self.no_crucibles):
            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]
            if max_allowed_spread:
                crucible_value = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_allowed_spread)
            else:
                crucible_value = self.calc_crucible_value ( crucible_quality ) ;
            crucible_value_sum += crucible_value
        return crucible_value_sum

    def trivial_solution(self):
        """Return a solution x=[0,1,2;3,4,5;6,7,8;...;48,49,50] of pots assigned to crucibles"""
        return np.arange(self.no_pots).reshape(self.no_crucibles, self.pots_per_crucible)

    def random_solution(self):
        """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] """
        rng = np.random.default_rng()
        x = np.arange(self.no_pots)
        rng.shuffle(x)
        return x.reshape(self.no_crucibles, self.pots_per_crucible)

    def plot_ascent(self, fx, fy):
        plt.plot(fy,'r', label="f(y)")
        plt.plot(fx,'b', label="f(x)")
        plt.xlabel('Function Evaluation Count')
        plt.ylabel('Objective Function Value')
        plt.legend()
        plt.show()

    ##########
    # TASK 3A #
    ##########
    def next_ascent_to_local_max(self, random_start=True, plotting=False):
        if random_start:
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # intermediate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            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]
            last_crucible_values[c] = self.calc_crucible_value(crucible_quality)

        if plotting:
            fx = []
            fy = []
            fx.append(sum(last_crucible_values))
            fy.append(sum(last_crucible_values))

        # for default case
        last_optimal_indices = (-1, -1, -1, -1)
        while True:
            # loop through neighborhood
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):

                            # exactly one scan since last optimal value found, can return
                            if (k, m, l, n) == last_optimal_indices:
                                if plotting:
                                    self.plot_ascent(fx, fy)
                                return x

                            # calculate crucible values and delta
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            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]
                            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
                            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]
                            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            if plotting:
                                fy.append(sum(last_crucible_values) + delta)


                            # > 0.001 as don't want to accept new solution if floating point error
                            if delta > 0.001:
                                # update intermediate values, solution, and optimal indices
                                last_optimal_indices = (k, m, l, n)
                                last_crucible_values[k] = crucible_k_value
                                last_crucible_values[l] = crucible_l_value
                                x[k][m] = crucible_k[m]
                                x[l][n] = crucible_l[n]

                            if plotting:
                                fx.append(sum(last_crucible_values))

            # case where starting at local max
            if last_optimal_indices == (-1, -1, -1, -1):
                if plotting:
                    self.plot_ascent(fx, fy)
                return x

    ##########
```

```python
    ##########
    # TASK 3B #
    ##########
    def steepest_ascent_to_local_max(self, random_start=True, plotting=False):
        if random_start:
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # intermediate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            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]
            last_crucible_values[c] = self.calc_crucible_value(crucible_quality)

        if plotting:
            fx = []
            fy = []
            fx.append(sum(last_crucible_values))
            fy.append(sum(last_crucible_values))

        while True:
            optimal_swap = (-1, -1, -1, -1)

            # min starting delta 0.001 for floating point errors
            best_delta = 0.001
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):

                            # calculate crucible values and delta
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            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]
                            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
                            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]
                            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            if plotting:
                                fy.append(sum(last_crucible_values) + delta)
                                fx.append(sum(last_crucible_values))

                            # if new steepest update best delta and save optimal swap location
                            if delta > best_delta:
                                best_delta = delta
                                optimal_swap = (k, m, l, n)

            # if all neighbors scanned and no better solution found, at local max and finish
            if optimal_swap == (-1, -1, -1, -1):
                if plotting:
                    self.plot_ascent(fx, fy)
                return x

            # Make swap with steepest neighbor and update intermediate values
            k, m, l, n = optimal_swap
            crucible_k = x[k].copy()
            crucible_l = x[l].copy()
            crucible_k[m] = x[l][n]
            crucible_l[n] = x[k][m]
            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]
            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
            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]
            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
            last_crucible_values[k] = crucible_k_value
            last_crucible_values[l] = crucible_l_value
            x[k][m] = crucible_k[m]
            x[l][n] = crucible_l[n]


    ##########
    # TASK 3E #
    ##########
    def do_repeated_next_ascents(self, n: int, max_spread: int = 0):
        best_obj_history = []
        obj_history = []
        times = []

        # Iterate through random starts to find history and best solution
        best_obj = 0
        start_time = time.perf_counter()
        for _ in range(n):
            if max_spread:
                x = self.next_ascent_to_local_max_spread(max_spread)
            else:
                x = self.next_ascent_to_local_max()
            obj = self.calc_obj(x)
            if obj > best_obj:
                best_x = x
                best_obj = obj
            best_obj_history.append(best_obj)
            obj_history.append(obj)
            times.append(time.perf_counter() - start_time)

        # Output and plot best solution
        self.view_soln(best_x)
        plt.scatter(times,best_obj_history,c='b',s=1, label="Best objective value")
        plt.scatter(times,obj_history,c='r',s=5, label="Local max")
        plt.xlabel('Time (s)')
        plt.ylabel('Objective Function Value')
        plt.legend()
        plt.gcf().set_size_inches(11.69, 8.27)
        plt.savefig("assets/repeated_next_ascents_chart.pdf", orientation="landscape")

    ##########
    # TASK 3E #
    ##########
    def do_repeated_steepest_ascents(self, n: int):
        best_obj_history = []
        obj_history = []
        times = []

        # Iterate through random starts to find history and best solution
        best_obj = 0
        start_time = time.perf_counter()
        for _ in range(n):
            x = self.steepest_ascent_to_local_max()
```

```python
            obj = self.calc_obj(x)
            if obj > best_obj:
                best_x = x
                best_obj = obj
            best_obj_history.append(best_obj)
            obj_history.append(obj)
            times.append(time.perf_counter() - start_time)

        # Output and plot best solution
        self.view_soln(best_x)
        plt.scatter(times,best_obj_history,c='b',s=1, label="Best objective value")
        plt.scatter(times,obj_history,c='r',s=5, label="Local max")
        plt.xlabel('Time (s)')
        plt.ylabel('Objective Function Value')
        plt.legend()
        plt.show()

    #########
    # TASK 6 #
    #########
    def next_ascent_to_local_max_spread(self, max_spread: int, random_start=True):
        if random_start:
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # init intermeidate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            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]
            last_crucible_values[c] = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_spread)

        # Loop through neighbors
        last_optimal_indices = (-1, -1, -1, -1)
        while True:
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):
                            # looped through all neighbors once and no better solution found
                            if (k, m, l, n) == last_optimal_indices:
                                return x

                            # calculate delta and other relevant params
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            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]
                            crucible_k_value = self.calc_crucible_value_with_spread(crucible_k_quality, np.ptp(crucible_k), max_spread)
                            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]
                            crucible_l_value = self.calc_crucible_value_with_spread(crucible_l_quality, np.ptp(crucible_l), max_spread)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            # better solution so update intermediate values and solution
                            if delta > 0.01:
                                last_optimal_indices = (k, m, l, n)
                                last_crucible_values[k] = crucible_k_value
                                last_crucible_values[l] = crucible_l_value
                                x[k][m] = crucible_k[m]
                                x[l][n] = crucible_l[n]

            # case where already at local max
            if last_optimal_indices == (-1, -1, -1, -1):
                return x
```