KMedoids problem by Simulated Annealing

In the directory called `"exercises"` there are 3 files related to the Simulated Annealing part of the exam.

1. `"SimAnn.py"` is the generic program for Simulated Annealing that we have used for the lectures and the following exercises. You don't even really need to look at this if you recall how it works.

2. `"KMedoids.py"` contains a class that defines a problem to be optimized via Simulated Annealing. This is the main file that you will need to modify. See below.

3. `"kmedoidsproblrun.py"` is a test script. You can run it in Spyder and it will create a problem and then try to optimize it.

The class definition in `"KMedoids.py"` is intended to represent a discrete optimization problem. The details of the problem that we want to solve are as follows:

1. We have a set of n points in the $[0, 1) \times [0, 1)$ plane, generated randomly (just like for the TSP problem that we've seen in class). Let's call P the set of points, and let's denote with (x_i, y_i) the coordinate of the i-th point, with $i \in 0, \ldots, n-1$.

   We can compute the (Euclidean) distance between any two points. Let's denote with d_{ij} the distance between the i-th and the j-th point.

2. We have a parameter k, an integer with $0 < k \leq n$.

3. We want to choose k of the points of our set P as "centroids" (also called for this specific problem "medoids"). We can identify each centroid by its index in the original list of points P. Say for example that we have $n = 10$ and $k = 3$, and we say that the set of centroids is `[3,0,8]`. This means that we have chosen the points (x_3, y_3), (x_0, y_0) and (x_8, y_8) as our centroids.

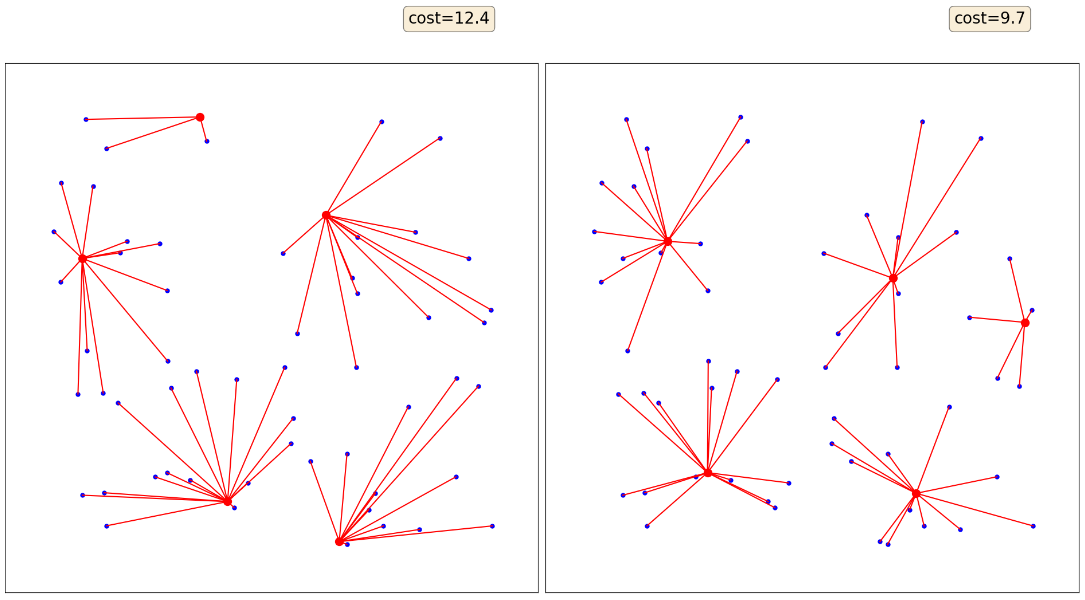
   Let us denote with C this list of indices that identify the centroids; of course its size is fixed to $|C| = k$. This list C determines the configuration of our problem.

4. Given the configuration of centroids C and the distance matrix d, the cost is computed like this:

$$c = \sum_{i=0}^{n-1} \min_{v \in C} (d_{iv})$$

   The meaning of this expression is that for each point i in the set P, we look for the centroid v in C that is closest to it, and add the distance between i and v to the cost.

   The goal is to find the set C of minimum cost. Look at the figure below. In the figure, there are 40 points, and $k = 5$. The centroids are marked as larger red circles, and red lines are drawn from each point to its nearest centroid (what we are trying to optimize then is the sum of the lengths of these segments).



Left: example configuration, unoptimized (random centroids); **Right:** optimal configuration for the same points

5. Our optimization method will be Simulated Annealing. We start from a random configuration C. The moves that we propose will be of this type: we pick one of the centroids of C at random, and replace it with one of its "neighbors". This is how we define the neighbors: we choose an integer parameter $D \geq 2$, and for each point in P we keep a list of the D points closest to it. We will end up with a list (of length n) of lists (each of length D). Since the points don't change, we can compute these lists right at the start, in the constructor.

The class already has most methods defined. But the implementation is incomplete/inefficient. You need to fix it.

Your tasks are described below. In the code, you can find them by looking for the string `"TASK"` in the comments. You should start by having a look at the constructor: read the comments in there, then start doing the tasks in their numerical order. The tasks are largely independent from one another, but to solve task 6 as intended you'll need at least tasks 2 and 4.

1. The `display` method is incomplete: it plots the points in blue and the centroids in red, but it doesn't plot the lines that connect each point to its nearest centroid. You can see this by running the `"kmedoidsproblrun.py"` script. You want your plots to look like the ones in the figure above. Tip: the style for lines is `'-'`. Once you've done this (or if you give up trying), you should uncomment the rest of the code in the test script.

2. The `init_config` method doesn't do anything useful at the moment. You should *overwrite the contents* of the `self.centr` array with a randomized configuration: just create a random permutation of the indices between `0` and `n-1` and take the first `k` elements. One line of code (there's a numpy function for this…).

3. The `cost` method computes the cost explicitly, with two nested loops. By now, you should have seen that we have an auxiliary array called `jopt` in our class. This lets you avoid computing the closest centroid to a point. Use it to write a more efficient and concise expression for the cost (an assertion in the code will help you check if you did it right). For the full score, write everything in one line of code (you'll need a comprehension).

4. In the constructor, the `neighb` list is not computed correctly: for each point `i`, it simply stores an array with `D` other points chosen at random. What we actually want is to store an array with the indices of the `D` closest points to `i`. The `np.argsort` function is what you need here. Keep in mind that the closest point to `i` is `i` itself.

   Bonus half point if at the end you rewrite the whole thing in a single line (without the asserts of course).

5. If you look at the `propose_move` method, you'll see that a move is encoded as two integers: the position `j` in the `centr` list where we want to make a change, and the new value for the centroid that we want to put there. In the `accept_move` method, a `new_centr` list is created that implements the change. Then, the `jopt` array is recomputed from scratch. This is inefficient, we don't need to recompute all of it, since only a few points will be affected by the change.

   Follow the instructions in the comments and update `jopt` only as needed. You can take inspiration from a somewhat similar code that you'll find in `compute_delta_cost`.

6. Once everything works, change the test script `"kmedoidsproblrun.py"` as follows: set the number of centroids in the

problem constructor to `8`, change `numtests` to `10`, change `debug_delta_cost` to `False`, and then find values of `beta0`, `beta1` and `anneal_steps` such that the annealing process works and you get the `"LOOKS GOOD"` message (with a reasonable result and within a reasonably quick computational time). The parameters should work even if you change `D=5` to `D=6` in the constructor. Don't change the `mcmc_steps` option.

Continuous differentiable version (a sort of k-means)

In the directory called `"exercises"` you'll find the file `"optim.py"`. We will use it to solve a problem related to the previous one. We will still have a set P of n points in the plane, but this time the centroids are not members of the set P. Instead, we have a separate list of k coordinates, $C = \{(\hat{x}_0, \hat{y}_0), (\hat{x}_1, \hat{y}_1), \dots, (\hat{x}_{k-1}, \hat{y}_{k-1})\}$.

Our goal is to find an optimal configuration for C, i.e. we want to find the optimal coordinates for the k centroids.

The function that we want to optimize is written below. It takes an extra parameter b (which we will just set to 100 in our code):

$$f(\hat{x}, \hat{y}) = -\frac{1}{b} \sum_{i=0}^{n-1} \log \left[\sum_{j=0}^{k-1} \exp\left(-b \, \left((\hat{x}_j - x_i)^2 + (\hat{y}_j - y_i)^2 \right) \right) \right]$$

Notice that the term in the argument of the exponential is the square of the distance between the i-th point and the j-th centroid. (You may also want to have a look at the simplified, 1-dimensional version of this same formula in the theory part.) Also notice that we have a slightly unusual situation: the x and y are considered <u>fixed</u> here, it's the $\hat{x}$ and $\hat{y}$ that can change. Also, $\log$ is the natural logarithm here.

1. You first task is to write that function. Looking at the code, you'll notice that the function `f` takes 3 arguments. The last two arguments are called `xp` and `yp` and they're supposed to be two arrays of length `n`, representing the coordinates of the points x_i and y_i. The first argument `z` should be an array of length `2*k` that contains the centroids coordinates $\hat{x}_j$ and $\hat{y}_j$, interleaved, i.e. $z = (\hat{x}_0, \hat{y}_0, \hat{x}_1, \hat{y}_1, \dots, \hat{x}_{k-1}, \hat{y}_{k-1})$. So the first thing that you should probably do is to extract two arrays `xc` and `yc` from `z` (use numpy's slicing rules to do it quickly and efficently).

   You can write the function `f` using two nested loops, one for each summation, but for a better score avoid at least the innermost loop and use numpy's broadcasting rules instead, which is relatively easy. For the full score, do everything without for loops and use only broadcasting: for this, you will need a very similar "trick" to the one used for computing the matrix of distances in the `KMedoids` constructor (but with two different sets of points, and without square root). You will also need to compute the row-wise sum of a matrix with the `axis` argument of `np.sum`.

2. In the script, the points `xp` and `yp` are generated at random, and there is a commented call to `minimize`. Set the value of `z0` to a reasonable starting value and uncomment the call to `minimize`.

3. At the end of the script, the points are plotted. Plot the centroids that you obtain from the solutions too, as red circles. The plot line should be similar to the corresponding one in the `KMedoids` class, but notice that now the solution that you get in `sol` will have the same format as the `z` argument to `f`, so you have to extract the `xc` and `yc` values, like in the first task. (Don't bother adding the lines that connect the centroids to the points here.)

4. Even if you now get some result from `optim`, it will still not give the absolute best value of `f`, unless you are lucky. As a further improvement to task 2, make the code run for 20 times, trying to *sample* different "optimal" values. You should record what is the best result that you got in those 20 runs, and make sure you plot that at the end.

Theory questions

Answer on paper to the questions below.

1. Consider the continuous problem discussed above. We now simplify it and assume to be in one dimension instead of two, therefore our n data-points are described by a single coordinate each: x_i with $i = 0, \dots, n-1$. Let's group together the positions of the k centroids, $\hat{x}_j$, in the vector $\hat{x}$. Our task is to compute the centroid positions by minimizing the objective function

$$f(\hat{x}) = -\frac{1}{b} \sum_{i=0}^{n-1} \log \left[\sum_{j=0}^{k-1} \exp\left(-b \, (\hat{x}_j - x_i)^2 \right) \right]$$

where $b > 0$ is some number, $\exp$ is the exponential function and $\log$ the natural logarithm. Keep in mind that, just like in the programming exercise, the datapoints x_i should be considered as fixed here; the optimization is performed over the $\hat{x}_j$ variables.

- Given a generic point $\hat{x}$, compute the gradient of f at $\hat{x}$, i.e. for each index $j \in \{0, 1, \dots, k-1\}$ express its component in the form

  $$(\nabla f(\hat{x}))_j = \dots$$

  Notice that, while in the mathematical convention vector components' indices start from 1, here we are assuming indices to range from 0 to $n-1$ to match Python's 0-indexing convention.

- What is the necessary condition on the gradient for a configuration $\hat{x}^*$ to be a local minimum of f?

2. (***St. Petersburg paradox***) A casino offers a game for a single player in which a fair coin is tossed at each stage. The initial stake starts at 2 euros and is doubled every time heads appears. The first time tails appears, the game ends and the player wins whatever is in the pot. Thus the player wins 2 euros if tails appears on the first toss, 4 euros if heads appears on the first toss and tails on the second, 8 euros if heads appears on the first two tosses and tails on the third, and so on.

   The casino is willing to pay only up to some limited amount of money. Therefore, after a certain number of tosses T, either the player has already "lost" (a tail already appeared), or the game ends and the player wins whatever is in the pot (i.e. 2^T euros).

   Can you devise an approximate method to compute, for a given T, the average win for the player, i.e. the average amount of money they would win if they were to play the game many many times?

   Write down the python code or the pseudocode for the corresponding algorithm.