Computer Programming - Exam

Ising chain problem by Simulated Annealing

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

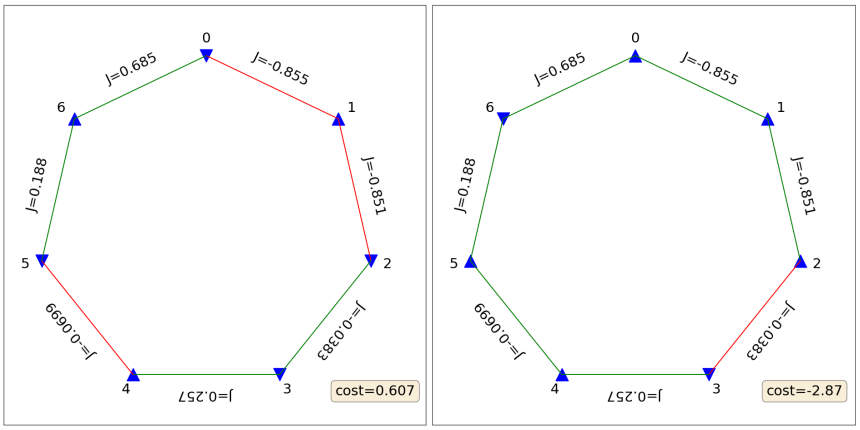
1. `"SimAnn.py"` is the generic program for Simulated Annealing that we have used for the lectures and the following exercises. You don't even really need to look at this if you recall how it works.
2. `"ChainProbl.py"` contains a class that defines a problem to be optimized via Simulated Annealing. This is the main file that you will need to modify. See below.
3. `"chainproblrun.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 `"ChainProbl.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 closed chain (a loop) of `n` nodes, labeled from 0 to $n-1$. Each node can take a value of $+1$ or -1. So we will have a vector x that encodes this information.
2. Each link between two neighboring nodes has a value J, a real number. There are n links. We say that the number J_i is associated with the link between the node i and the node $i+1$. Except that the last link (with value J_{n-1}) is between node $n-1$ and node 0. The values of these J_i are generated at random.
3. Given the values J, the cost of a configuration x is computed like this:

$$c = \sum_{i=0}^{n-1} J_i \, x_i \, x_{i+1}$$

   where for convenience we identified the node x_n with x_0. The goal is to find the x of minimum cost. Look at the figure below. In the figure, there are 7 nodes. Nodes set to $+1$ are represented by upwards triangles, nodes set to -1 by downwards triangles. Each link is colored in green if its contribution to the cost is negative (which is good), or in red if it is positive. Notice that if some link has $J_i > 0$ we want its neighboring nodes to have opposite orientations, and if it has $J_i < 0$ we want them to have the same orientation.



Left: example configuration, unoptimized; **Right:** optimal configuration for these points

The class already has most methods defined. But the implementation is incomplete, and there are bugs. 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.

1. The constructor generates the entries of `J` as uniform random numbers between `-1` and `1`. Instead, make

it such that it generates only numbers at steps of `0.1`, i.e. such that each `J[i]` is one of `-1.0`, `-0.9`, `-0.8`, …, `0.8`, `0.9`, `1.0`. An additional constraint is that you need to use `randint` for this task. For the full score, don't use any for loops.

2. The function `compute_cost` is not implemented. Implement it according to the formula above. For the full score, don't use any for loops.

   Suggestion 1: this last part is not straightforward because of the last link connecting the last node with the first one. The easiest approach is to treat that last link separately from the others.

   Suggestion 2: This task is crucial for the rest. Don't spend too much time on making it efficient. Make sure that it's working properly instead (e.g. run the `"chainproblrun.py"` script and look at the plot; make sure that the cost displayed matches what you see in the plot).

3. If you performed task 2, your code should now work without errors when you run the `"chainproblrun.py"` script, and it should be doing something reasonable. Change the script by increasing the problem size from `7` to `30`. Also set the `numtests` to 10. When you run the code now you should see that it runs `10` times on the same problem and finds slightly different results each time (it prints "UHM…" and a list of costs). Change the parameters `beta0`, `beta1` and `anneal_steps` in the `simann` call, such that it works reliably instead (it should print "LOOKS GOOD" and the final costs should be low, and when you look at the figure displayed it should be mostly green). This should happen even if you change the seed in the `ChainProbl` constructor.

   Important: you are not allowed to change `mcmc_steps` here.

4. The `compute_delta_cost` method is written in a very inefficient way, requiring $O(n)$ operations. Write a new piece of code that computes the cost difference associated with a proposed move in $O(1)$ operations instead. You basically need to write the whole method. It's advisable to have both the new and old versions and compare their output with an assertion while you test it. Once you have an efficient version that works (and *only* at this point) you should comment out the inefficient version and then set the option `debug_delta_cost=False` in the test script.

5. **[EXTRA, NOT PART OF THE ORIGINAL EXAM]** Using Simulated Annealing to solve this particular problem is extremely inefficient: there are much better options. You may take inspiration from the results that you get from Simulated Annealing and notice some characteristic of optimal solutions that should suggest you how this problem can actually be solved in $O(n)$ time. Then you should write an algorithm that does that. *Hint:* maybe start from a simplified situation in which $J_i = 1$ for all i. After that look at the case of uniform random J_i.

From a loop of binary nodes to an open chain of continuous ones

In the directory called `"exercises"` you'll find the file `"optim.py"`. We will use it to solve a very similar problem to the previous one. We will still have n nodes in a chain, but this time the chain is open for simplicity (the last node is not connected to the first one, there are only $n-1$ links instead of n. Also, this time the values in x are not binary, they are real numbers.

The function to optimize this time is:

$$f(x) = \sum_{i=0}^{n-1} x_i^4 + \sum_{i=0}^{n-2} J_i \, x_i \, x_{i+1}$$

Notice the first term which wasn't there previously. Also notice that the second term goes up to $n-2$ instead of $n-1$.

1. You first task is to write that function. For the full score, don't use any for loops. Notice that the two inputs of `f` (i.e. `x` and `J`) have different sizes.

2. In the script, a vector J is generated at random, and there is a call to `minimize`. Even after you have written correctly the function `f`, this call will exit immediately with an optimal value of `0` (look at the `"fun:"` line in

the output, and the `"x:"` line too). The optimal cost should not be `0`. Change the script (in a very simple way) so that it doesn't do that.

3. Even if you now get some better result from `optim`, it will still probably not give the absolute best value of `f`, unless you are very lucky. As a further improvement to the previous task, 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 its corresponding optimal x, and print them.