n-queens problem by Simulated Annealing

In the directory called `"exercises"` there are 5 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. `"NQueens.py"` and `"NQueensP.py"` contain one class each that defines a problem to be optimized via Simulated Annealing. These are the main files that you will need to modify. See below.
3. `"nqueensproblrun.py"` and `"nqueensproblrunP.py"` are test scripts. You can run them in Spyder and they will create a problem and then try to optimize it.

The class definitions in `"NQueens.py"` and `"NQueensP.py"` are intended to represent a discrete optimization problem known as the "n-queens problem". The details of the problem that we want to solve are as follows (see also the figures below):

1. We have a square $n \times n$ grid (for $n = 8$ you can think of a chessboard), and we have n items (the "queens") that we want to position on the grid. In each position in the grid there can be at most one queen.
2. Our goal is to position the queens such that no two of them can "attack" each other. This means that we want to satisfy the following constraints:

   a. No two queens should be aligned on the same column

   b. No two queens should be aligned on the same row

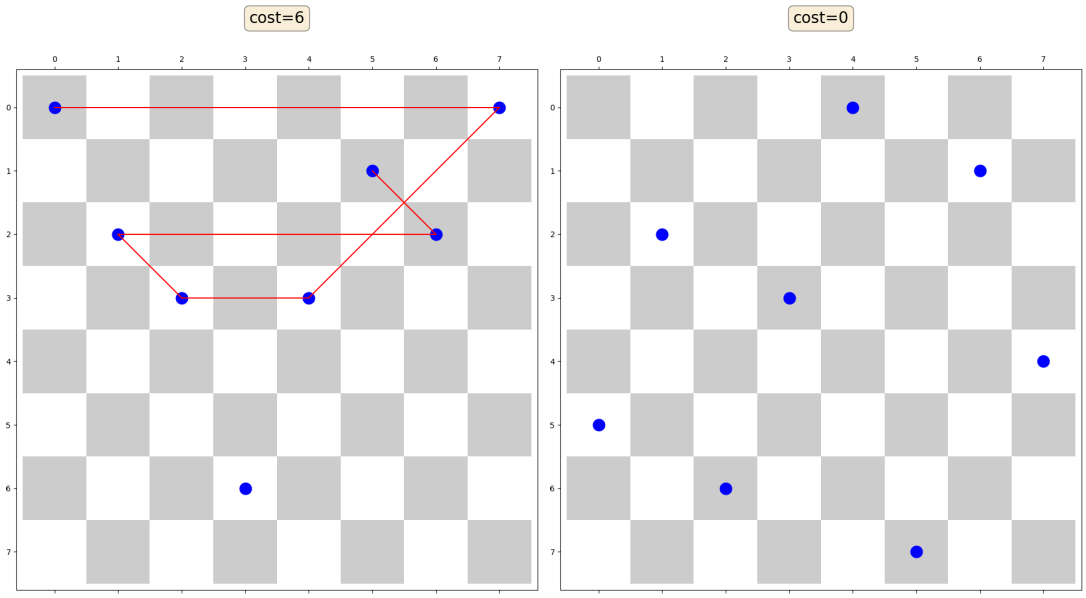
   c. No two queens should be aligned diagonally

   It is known that achieving this is always possible for any $n > 3$, and in fact there are always many configurations that solve the problem. However, it is not easy to do it without resorting to a trial-and-error approach.
3. Since we need to put a queen per column anyway (constraint 2.a above), we will represent the configuration of the pieces with a vector q, such that $q_i \in \{0, 1, \ldots, n-1\}$ represents the row of the queen in the i-th column. For example, $q_3 = 0$ means that the queen in column 3 is positioned in row 0. See the figures below for more examples.
4. Given the configuration q, the cost is defined as the number of violated constraints. Given two queens positions q_i and q_j we introduce a function $v\left(i, q_i, j, q_j\right)$ that returns 1 if the two queens violate one of the constraints (point 2 above), and 0 otherwise. Then the cost is:

   $$c = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} v\left(i, q_i, j, q_j\right)$$

   Notice that the second sum starts from $j = i + 1$: this is to avoid counting each pair (i, j) twice, and also to avoid the case $i = j$.

   Because of the representation that we have chosen, we obviously only need to worry about violations of the constraints 2.b and 2.c, not 2.a which is accounted for implicitly.



Left: example of invalid configuration (red lines show violated constraints); here $n = 8$ and the configuration is $q = [0, 2, 3, 6, 3, 1, 2, 0]$. **Right:** a valid configuration; $q = [5, 2, 6, 3, 0, 7, 1, 4]$.

Our optimization method will be Simulated Annealing. We will try two very similar but different strategies, the first one in the class `NQueens` and then an improved one in the class `NQueensP`. The only difference in the second class is the initialization of the configuration and the kind of moves that we propose.

- In `NQueens`, the q is initialized with random entries, and the move proposal consists in choosing a column i and changing the current row q_i to some other random row.
- In `NQueensP`, the q is initialized as a random permutation of the indices $\{0, 1, \ldots, n-1\}$, and the move proposal consists in choosing two columns (i, j) and swapping the column values (q_i, q_j). This is better because the constraint 2.b (clashing rows) is automatically accounted for, and then we only need to worry about constraint 2.c (clashing diagonals).

The classes already have 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're supposed to do the tasks in their numerical order. The tasks are largely independent from one another.

IMPORTANT ADVICE: At the start, the test scripts will run without errors, although they don't actually solve the problem. Whenever you attempt a task, you should still get no error messages from Python. If you do, and you can't fix it, comment out your attempt and restore the code as it was before moving on to another task. Commented out attempts will be evaluated.

More advice: before starting, skim through the code once, and skim through the tasks once to get an overall idea.

1. At the beginning of the `"NQueens.py"` file, there is a function `conflict` that should check whether the entry q_{i_0} at position i_0 is in conflict with the entry q_{i_1} at position i_1, i.e. if those two entries violate one of the constraints 2.b or 2.c (it's the function that was called v in the description above). It is incomplete, and you should fix it. (Note: the version in the other file `"NQueensP.py"` is fine, don't touch it.)
2. The `display` method in `"NQueens.py"` is incomplete: it plots the grid and the queens (as blue circles), but it doesn't plot the lines that show the presence of conflicts. You can see this by running the `"nqueensproblrun.py"` script. You want your initial plots to look like the ones in the figure above (left). Tip: the style for lines is `'-'`. Once you've done this, copy-and-paste your code to the `"NQueensP.py"` file.
3. The `init_config` method in `"NQueens.py"` substitutes `self.q` with a new random array. Change it such that the operation is performed in-place, i.e. that the *contents* of `self.q` are overwritten.
4. The `propose_move` method in `"NQueens.py"` has a subtle bug, it will sometimes produce moves that are always going to be accepted no matter what. Fix it. For the full score, do it without using any loops (advice: don't spend too much time on this last optimization if you can't figure it out).
5. Once everything works, change the test script `"nqueensproblrun.py"` as follows: set the number of queens in the

problem constructor to `20`, change `numtests` to `10`, change `debug_delta_cost` to `False`, and then find values of `beta0`, `beta1` and `mcmc_steps` such that the annealing process works and you get the `"LOOKS GOOD"` message and a solution to the problem (within a reasonably quick computational time). Don't change the `anneal_steps` option.

6. Now move on to `"NQueensP.py"` and `"nqueensproblrunP.py"`. In the constructor of `NQueensP` and in the `init_config` method the configuration is initialized to $(0, 1, \ldots, n-1)$, which is valid but is about the worse choice that one could make. Change it so that `self.q` is initialized to a random permutation of the indices (there are at least two numpy functions that you can use for this…). For the full score, make `init_config` operate in-place, like for task 3. You should also uncomment the tests in `"nqueensproblrunP.py"` at this point.

7. Same as task 4, but for `"NQueensP.py"` (here the move proposes two indices to swap).

8. The `compute_delta_cost` method in `"NQueensP.py"` is written in an inefficient way. Make it efficient, taking inspiration from the corresponding code in `"NQueens.py"` (but it's slightly more complicated here). The code is arranged so that it immediately tests whether your code is working correctly: follow the instructions! Once everything works, you can comment-out the inefficient version and the assertion test, and set `debug_delta_cost=False` in the test script.

9. Same as task 5, but for `"nqueensproblrunP.py"`. You should be able to find solutions in a shorter time with this version.

Continuous differentiable version

In the directory called `"exercises"` you'll find the file `"optim.py"`. We will use it to try to solve the same problem as before, but this time we will make the problem continuous, use `scipy.minimize` to find an optimum, and finally discretize the result (this is not a good idea, it'll only work well for small n…). Our representation will still be a vector q of length n, but this time its elements are going to be real numbers.

We will first introduce two auxiliary functions (here x is a real number):

$$z(x) = \frac{1}{1 + (1.5x)^{10}}$$
$$r(x) = \begin{cases} 0 & \text{if } x \geq 0 \\ x^2 & \text{if } x < 0 \end{cases}$$

These are already implemented, although you'll be asked to modify r. With these, we define three components of the cost of a given configuration q:

$$c_1(q) = \sum_{i=0}^{n-1} \sin\left(\pi q_i\right)^2$$
$$c_2(q) = \sum_{i=0}^{n-1} \left(r(q_i) + r\left((n-1) - q_i\right)\right)$$
$$c_3(q) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} z\left(q_i - q_j\right) + z\left(|q_j - q_i| - |j - i|\right)$$

The function that we want to optimize is written below. It takes two extra parameters b and k (which we will just set to $\frac{n}{2}$ and 100, respectively, in our code):

$$f(q) = c_1(q) + k\, c_2(q) + b\, c_3(q)$$

[Brief explanation: The first term favors configurations whose components are nearly integer; the second term favors configurations with components inside the range $[0, n-1]$; the third term adds a penalty for violating the constraints 2.b and 2.c of the previous exercise.]

1. The function `r(x)`, as currently written, only works when its argument is a scalar (a single number). Your task is to rewrite it such that it works both for scalars and for numpy vectors. When given a vector argument, the function should return a vector as a result, for example we should get $r((1, -2, -1.5, 0)) = (0, 4, 2.25, 0)$. You shouldn't use any `if` statements or any loops, the vector version should exploit broadcasting rules. For example, notice that an expression like `x > 0` works both for scalars and for vectors.

   If you can't figure this out, keep the previous version and skip to the next task. You won't be able to compute $c_2(q)$ in a

vectorized way, but you can still do it for c_1 and c_3.

2. Write the function $f(q)$. Compute c_1, c_2 and c_3 as three separate quantities inside the function, then sum them up at the end with the appropriate coefficients k and b. You can implement the expression using loops, but for the maximum score you should have a version without loops that exploits broadcasting.

   The broadcasting version of c_2 is only possible if you have done task 1. Otherwise, if you know how to do it in principle just use the loop version and write the broadcasting one in a comment.

   The broadcasting version of c_3 is more complicated to do, because of the double summation, and because the summation only involves non-diagonal elements. Here are two tips on how to do it:

   a. Preliminarily compute a $q_j - q_i$ matrix using the `reshape` trick that we have seen for example when computing distance matrices. Same for a $j - i$ matrix.

   b. It is possible to rewrite c_3 using a summation on the whole matrix, $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \dots$, but you need to subtract the diagonal and then divide by 2. The contribution from the diagonal terms that you need to subtract is very easy to compute, fortunately.

3. Choose a reasonable initial value for `q0`, the starting point to be passed to `minimize`. The starting value should be of the correct type, with the appropriate bounds. Don't restrict yourself to integers using `randint` or similar, and don't risk getting out of bounds with `randn`, etc. (If you want to start from integers you should at least add a little noise to each). Then, uncomment the call to `minimize`, and make sure that the resulting configuration is contained in a variable called `bestq` and its corresponding `f` value is in `bestval`. Finally, uncomment the rest of the plotting code until the end.

4. At the end of the script, the points are plotted. If things work correctly, the points should end up being close to the centers of the cells of the grid, but not exactly, they will have a slight vertical displacement. Modify the solution (before printing it, where indicated in the code) so that they end up in their nearest integer and you obtain a configuration of the same form as in the Simulated Annealing part.

5. Unless you are lucky, the solution that you get will not be a valid one for the n-queens puzzle. Make the code run for 20 times or more, trying to *sample* different "optimal" values. You should record what is the best result that you got, and make sure you display that at the end. You should have enough tests such that you get a solution at the end.

Theory questions

Answer on paper to the questions below.

1. Using a recursive approach, solve the following problem:

   You have a list of coins denominations, supposed to be integers, e.g. for Euro cents you have `l = [1,2,5,10,20,50,100,200].` The coins are in unlimited supply. Given an integer `x`, what is the minimum amount of coins you need in order to return `x` as a sum of those coins?

   Write down the corresponding pseudo-code or Python code, assuming your function takes `l` and `x` as inputs.

2. Consider the function $f(x) : \mathbb{R} \to \mathbb{R}$ defined by

   $$f(x) = (x^2 - a)^2$$

   where $a > 0$ is some positive real number .

   - What are the stationary points of f?
   - Write down the configuration reached after one step of the Newton method with stepsize α starting from some point x_0.