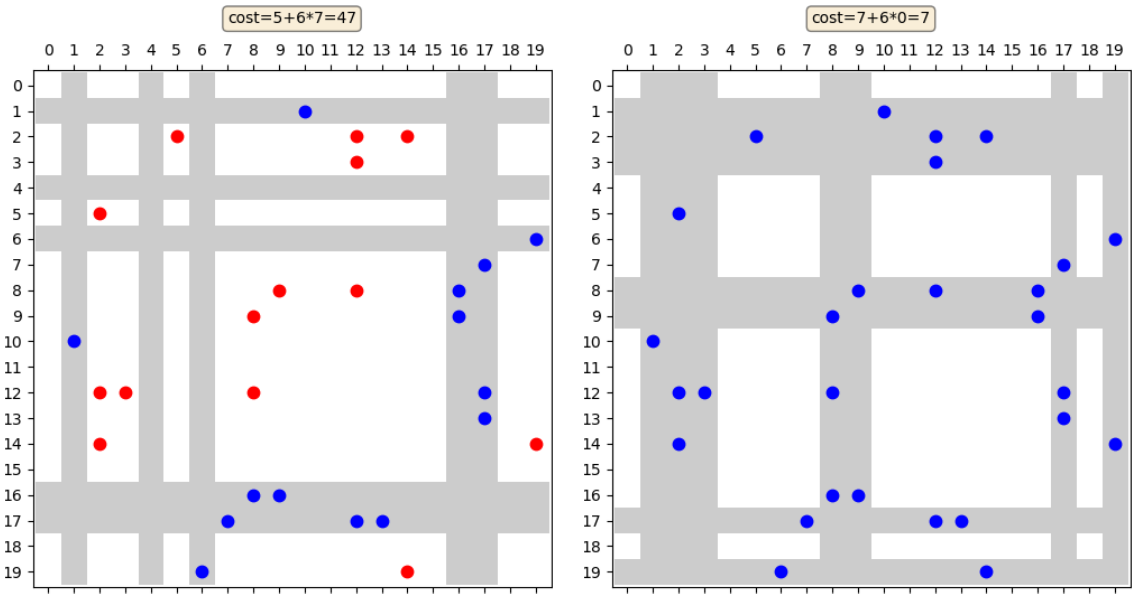
Computer Programming - Exam [31/01/2020]

Minimum Vertex Cover 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. `"MinVertCover.py"` contains one 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. `"minvertcovrun.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 `"MinVertCover.py"` is intended to represent a famous hard discrete optimization problem called "Minimum Vertex Cover". It is described below. While reading, refer to these pictures (the captions will become intelligible after the explanation):



LEFT: example of a bad configuration with $n = 20$ and $k = 6$; the active nodes are represented by the greyed-out rows/columns (here x_i is active for $i \in \{1, 4, 6, 16, 17\}$, and inactive otherwise); the dots represent the edges of the graph (blue=covered, red=non-covered).
RIGHT: an optimal configuration; x_i is active for $i \in \{1, 2, 3, 8, 9, 17, 19\}$.

These are the details:

1. We generate a random unweighted undirected graph with n nodes, where each pair of nodes has some probability p of having an edge. We represent the graph with an adjacency matrix A, where $A_{ij} = 1$ if there is an edge between nodes i and j and 0 otherwise (in Python, we are going to use `True` and `False` instead of 1 and 0). In the pictures above, you can see an example matrix (it's the same in both pictures) in which the edges are represented with dots. The matrix is symmetric since the graph is undirected.
2. The problem consists in choosing some of the nodes. We represent this by using a binary vector x of length n, and we will call a node i "active" if $x_i = 1$ and "inactive" if $x_i = 0$ (again, in Python we'll use `True` and `False`).

3. We want the set of active nodes to be small. We will thus define a component of the cost that expresses that:

$$c_x = \sum_{i=0}^{n-1} x_i$$

(Remember that the mathematical notation for the summation ranges uses a different convention from Python's ranges…)

4. We also want our active node to "cover" all the edges. This means that for each edge (i, j), for which $A_{ij} = 1$, we want at least one of x_i or x_j to be 1. In the above pictures, the active nodes x are represented by graying out all the rows and the columns for which $x_i = 1$; the covered edges are the blue-colored dots, and the non-covered ones are the red-colored dots.

   We thus need a component of the cost that counts the non-covered edges:

$$c_u = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{ij} \times (1 - x_i) \times (1 - x_j)$$

   (The fraction $1/2$ is used to avoid double-counting since each edge is represented twice in A. Each term in the sum is 1 only if all three terms are 1, which happens when (i, j) is an edge, and it is not covered by node i, and it is not covered by node j.)

5. The overall cost puts together the two terms; we use a factor $k \geq 1$ to express the fact that we want to penalize non-covered edges more than active nodes:

$$c = c_x + k \times c_u$$

   In our code, we won't initially have the parameter k (it's going to be as if it were 1) but it will be your task to introduce it, then we'll use $k = 6$ in our tests.

6. For the Simulated Annealing algorithm, we need to decide what moves to propose and how. We'll pick an index $i = 0, \ldots, n - 1$ uniformly at random, and propose to flip the value of x_i (make the node inactive if it is currently active and vice-versa). The initial configuration x will be set randomly.

The class `MinVertCover` 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're supposed to do the tasks in their numerical order. The tasks are largely independent from one another, but some are crucial.

IMPORTANT ADVICE: At the start, the test script `"minvertcovrun.py"` will run without errors and display a configuration. You'll notice that there are no grey bands: that's because the `x` is initialized to all `False` (you'll need to fix that). After completing a few tasks, you should uncomment the rest of the test script and actually start testing the simulated annealing part. 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. The only truly crucial task, without which nothing will make sense at all, is 4. In order to get some reasonable results task 3 is also extremely important.

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

Here are your tasks:

1. The `init_config` method in `"MinVertCover.py"` currently does nothing, and as a result the vector `x` of the class is initalized to all `False` (look a few lines above, in the constructor). Your task is to fill `x` with random values, such that each entry `x[i]` is `True` with probability `0.3`. For the full score: 1) do it in one line of code without for loops, and 2) make sure to overwrite the contents of `x`, without changing its identity. You should

now see some grey bands when you run `"minvertcovrun.py"`. **NOTE:** unless you do this it will be hard to inspect the result of the other tasks; if you can't do it, at least set `x` to some combination of `True` and `False` values (e.g. alternating them, or pick a few values by hand and set them to `True` or whatever).

2. The `display` method at the end is incomplete: all the edges are plotted in blue. We want to highlight the non-covered edges by plotting them in red instead, so that the initial plots look like the one in the figure above (left).

3. The `split_cost` is supposed to compute both components of the cost `cx` and `cu`, but it's currently only computing `cu` correctly. Implement the computation of `cx`. For the full score, without for loops. (This task is necessary in order to get anything meaningful at all from the optimization.)

4. The `accept_move` method is not implemented, it's supposed to flip one entry in `x` corresponding to index proposed by `propose_move`. For the full score, do this without using `if`. (This task is crucial to run simulated annealing.) After this is done, you should be able to uncomment the code in `"minvertcovrun.py"` and not get errors.

5. The code in the whole `MinVertCover` class is not using the parameter k. Change the constructor so that it takes an argument `k` with a default set to `6`. Then store it inside the object and go through each method of the class, changing what needs to be changed such that the cost is computed according to the formula written above. Note: it's not only the cost-related methods that need changing, although those are the most crucial.

6. The `compute_delta_cost` function carries out its computation in a naive and inefficient way, and returns early. After that, there is a partially-implemented efficient version. Complete the implementation (comment/uncomment parts of the code as described in the instructions in the comments). Note: if you get what the first half of the implementation is doing, the second half is very easy. (If you can't work it out and keep getting errors, restore the early return.)

7. The computation of `cu` in the `split_cost` method is inefficient. Make it efficient by using broadcasting instead of for loops. (Make sure not to mess-up the computation; if that happens and you can't fix it, restore the initial version and leave yours in a comment.)

8. Start changing the parameters in the `simann` call in `"minvertcovrun.py"` so that simulated annealing starts working properly. If everything works, you can set `debug_delta_cost=False` to go a little faster in your tests, then when you have something good change `numtests` to `10` and check if you get the `"LOOKS GOOD"` message and the final result looks like the picture above on the right (in a typical solution, there should be about one third of the `x` active). Then uncomment the line at the beginning that creates a problem with `n=60`, and keep adjusting the parameters until you get the `"LOOKS GOOD"` message (within a reasonably quick computational time).

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 binarize the result. Our representation will still be a vector x of length n, but this time its elements are going to be real numbers.

We will use almost the same cost function as before:

$$c = c_x + k \times c_u + \alpha \times c_b$$

The extra term at the end is intended to favor configurations in which each entry is close to either 0 or 1: α is a parameter (we will use $\alpha = 20$ in our tests), and the expression of c_b is:

$$c_b = \sum_{i=0}^{n-1} x_i^2 \left(1 - x_i\right)^2$$

Some parts of the code in `"optim.py"` are nearly copied-and-pasted from `"MinVertCover.py"`. In particular, the last part of the script is taken from the `display` method: you may want to modify it like for task 2 of the previous exercise so as to spot non-covered edges more easily.

Here are your tasks:

1. Implement the function `f` in the code so that it returns the cost as written above. The only term that actually needs implementing is `c_b` (for the full score, do it without using for loops). The other terms can almost be copied-and-pasted from the simulated annealing exercise (if you have done task 3 there). One thing to pay attention to is the fact that now we are no longer dealing with integers but with floats, so there is a small adjustment that should be made.

2. Implement the `binarize` function, that transforms a float vector into a bool vector by deciding for each entry whether it's closer to 0 or 1, as described in the comments. For the full score, without for loops.

3. Choose a reasonable initial value for `x0`, the starting point to be passed to `minimize`. The starting value should be of the correct type. Don't restrict yourself to integers or bools, in particular don't use the same initialization as for the simulated annealing code: make sure that each entry is (with high probability) non-integer, although not too far from the expected range either. Then, uncomment the call to `minimize`, fix the call so that it actually works, binarize the resulting configuration (store it in a variable called `besty`) and store the cost of the *binarized* configuration in a variable called `bestc`.

4. It is quite possible that after task 3 you still won't obtain a very good configuration, for example there may be considerably more active nodes than in the optimal one found by simulated annealing (depicted in the figure, on the right). Adopt 2 strategies:

   - Make the code run for 10 times or more, trying to *sample* different "optimal" values. You should record what is the best result that you got, where we intend "best" *after* the binarization, not before. Make sure you display that at the end.

   - Try to adjust the initial `x0` so that its values are somehow closer to 0 than to 1, and see if results improve.

   With a few repetitions and some judicious initialization it should be easy to get configurations of the same cost as those found for the same problem by simulated annealing (the actual configuration might be different, several solutions are possible).

Theory questions

Answer on paper to the questions below.

1. You are given a 3×3 transition matrix P, where P_{ij} represents the probability of transitioning from state j to state i for a Markov chain with 3 states. The matrix is the following:

$$P = \begin{pmatrix} \frac{1}{10} & \frac{3}{5} & \frac{3}{10} \\ \frac{3}{10} & \frac{1}{10} & \frac{3}{5} \\ \frac{3}{5} & \frac{3}{10} & \frac{1}{10} \end{pmatrix}$$

   i. Verify that that P is a well-normalized stochastic matrix.
   ii. Check that the uniform distribution $\pi = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ is a stationary distribution for P.
   iii. Does π also satisfy the detailed balance condition?

2. Write down the implementation (python or pseudo-code) of the Floyd-Warshall algorithm in a top-down recursive approach (only the computation of the costs matrix, not the path reconstruction). Describe the time and memory complexity of your implementation.