📖 **exercises**

Knapsack problem by Simulated Annealing

You are provided with a directory called `"exercises"` which contains 3 files:

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. `"KnapsackProbl.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. `"knapproblrun.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 `"KnapsackProbl.py"` is intended to represent a knapsack problem. The details of the problem that we want to solve are as follows:

1. The data for the problem is passed via the constructor. It's constituted by two equal-length vectors, called `weights` and `values`, and by one float value, called `limit`. The constructor already implements the argument checks and most of what is needed.
2. Our goal is to solve the "Knapsack problem": we are given a list of items, each of which has a certain weight and a certain value, and we are given a weight limit (the "capacity" of our knapsack), and we want to pick those items that maximize the total value without exceeding the weight limit. (If it helps, you can also think that the weights are costs in euros, the limit is your available budget, and each of the values is how much you like that particular item.)

   In formulas: let's say that the weights vector is called w, that the values vector is called v, that their length is n, and that the weight limit is called m. Our problem is:

   $$\text{maximize}: \quad \sum_{i=0}^{n-1} x_i v_i$$

   $$\text{constrained to}: \quad \sum_{i=0}^{n-1} x_i w_i \leq m \;\; \text{and} \; x_i \in \{0, 1\}$$

   So we need to find the best binary vector x. If $x_i = 1$ it means that we are picking item i, if it is zero it means that we're discarding it.
3. We want to start from a random a candidate vector x, called `x` in the code. Then we want to propose changes to it and run Simulated Annealing. Because Simulated Annealing is a minimizer, we will change the sign of our objective function. Also, instead of enforcing the constraint on the maximum weight, we will just put it inside the objective function in the form of an extra cost that we pay for exceeding the limit, with a (large) scaling factor that we call k. So let us first define an auxiliary function

   $$\Theta(s) = \begin{cases} s & \text{if } s > 0 \\ 0 & \text{if } s \leq 0 \end{cases}$$

   With that definition, we can write the actual formula that we'll be using (notice the minus sign at the beginning):

   $$\text{minimize:} \;\; \text{cost}(x) = -\sum_{i=0}^{n-1} x_i v_i + k \, \Theta\left(\left(\sum_{i=0}^{n-1} x_i w_i\right) - m\right)$$

   The use of the Θ function ensures that if we are below the limit we don't pay any cost, and if we're above we pay a price proportional to the exceeding part.

   The parameter k can be passed as an option via the constructor of the class, but we will generally just use the default value $k = 100$ throughout, so you don't need to touch it.

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

Your tasks are described below (most of these points are independent from each other; however, for the last one to make sense you need at the very least to have done point 3). In the code, you will find the pieces of code that you need to change by looking for comments that start with “TASK”.

1. The constructor initializes the vector `x` with zeros, then calls `init_config`. We want that method to initialize `x` at random, such that each element has a 50% chance of being 1. For the full score, do the initialization without using any for loops (but don’t bother too much about it if you can’t figure out how to not use loops, focus on the other tasks instead).

2. The cost function computes the quantity `totv` as $\sum_{i=0}^{n-1} x_i v_i$ using a `for` loop. Do it using numpy’s functions in a single line instead, without loops. This task is not crucial for the rest.

3. The cost function only implements the “value” part of the function that we want to minimize. You need to complete it with the second part, the one with k and the Θ function defined above. Note that k is already defined as a member of the class. This task is crucial for the optimization to work.

4. The current code proposes just one possible type of move: it picks an item at random and proposes to flip it (make it 1 if it’s 0 and vice-versa). You need to implement a new kind of move, which we call a “swap” move. This move will be produced with 50% probability (and provided some extra conditions are satisfied). The idea is that we swap one item i that we have in the knapsack (for which $x_i = 1$) with an item j that we don’t have (for which $x_j = 0$). Basically we flip both x_i and x_j. This however is only possible provided x is not all-1 or all-0.

   So your code should check whether we’re in the all-1 or all-0 situation, in which case only the “flip” move is allowed and thus should be the chosen move. If we’re not in that situation, then the “swap” move is possible, and with 50% probability you should choose it. If you do,you then need to two items at random provided that their x values are different, and return them as a tuple. In the remaining 50% of the cases you should still do the “flip” move. In the `accept_move` method, you should account for the fact that two types of moves are now possible, by checking whether the `move` argument is a single index (it’s a “flip” move) or a tuple (then it’s a “swap” move), and proceed accordingly.

5. (**Tip:** you may want to save your file and have a backup before attempting this.) Make the code more efficient. The quantities $\sum_{i=0}^{n-1} x_i v_i$ and $\sum_{i=0}^{n-1} x_i w_i$ can be computed once at the beginning, right after the internal state `x` is generated by `init_config`, and stored as attributes inside the problem object. Call them `totv` and `totw`. Then, they can be used for the cost computation (thereby simplifying that method), and updated whenever a move is accepted. You need to figure out how to update them, both for the “flip” and for the “swap” moves, and you need to do it where appropriate and in a computationally efficient way (i.e. not just recomputing them from scratch). Make sure that you update **all** methods that require it.

6. In the `"matchingproblrun.py"` file, there are two lists `w` and `v`, and a `limit` variable, that you can use for preliminary testing. Make sure that when you run the code you get the optimal solution for that case, which has a cost of `-3.6` and corresponds to bagging items `0,1,3,4`.

   Then, create new test instances: produce `w` and `v` as random floating-point (not integer!) vectors of length `40`. Each of them should have entries uniformly distributed in the interval $[0, 10)$. Set the limit to `5.0`.

   After this, do some experiments and change the values of the parameters to the `simann` function, such that it finds a reasonable solution in a reasonable time (a few seconds at most!). The parameters should make sense in the context of the simulated annealing algorithm. Set them such that the initial acceptance ratio (at the first step) is somewhere between 30% and 90%, and such that at the end is less than 2%. You should also make sure that, if you run the solver multiple times, it should work fine most of the times: modify the script such that it calls `simann` a few times (5 to 10 is enough) with a different seed each time, collecting the resulting costs and displaying the minimum and maximum that you find. What you want to see is that those maximum and minimum values are basically the same.