Greedy solver exercises

For each exercise, read it once to the bottom before you start coding.

Important: for any different problem that you are trying to optimize, the parameters such as the number of iterations and of repetitions required will change, and there will be a different trade-off between simulation time and performance improvement. This is even true for the same problem at different sizes, or the same problem with a different proposal of the moves etc.

The exercises from 4 to 9 are in sequence: each of them depends on the previous one. They get progressively harder.

Exercise 10 is unrelated, and not too hard. However, it is intentionally more vague in its description: you are free to make your own decisions about how to approach it.

Exercise 11 is sort of similar to 4-8, but not quite. It's as vague as ex. 10, but harder.

These exercises are also perfectly valid for Simulated Annealing.

1. One more time

Try to recreate all we did in class, not necessarily in the same order but **from scratch**, by your own. It may seem pointless but it's useful. Also, feel free to explore variations and to experiment by your own.

2. The wrong move

In the TSP code, we wanted to extract two indices `e1` and `e2`, each between `0` and `n-1`, such that `e1 < e2`. We did it with this code:

```
while True:
    e1 = np.random.randint(n)
    e2 = np.random.randint(n)
    if e1 > e2:
        e1, e2 = e2, e1
```

You may think that you could avoid the `while` loop by extracting the indices with the right property, like this:

```
e1 = np.random.randint(n-1)
e2 = np.random.randint(e1+1, n)
```

However, the above code produces pairs of integers with a different probability distribution than the previous one. In the previous one, each of the $\frac{n(n-1)}{2}$ possible pairs of indices is produced with the same probability, but in this one there is a bias.

(Note: we will at some point evolve the greedy algorithm to Simulated Annealing, and for that purpose the second code is not just different but definitely wrong: the resulting `i` and `j` will not have the right probability distribution. In turn, this will disrupt the detailed balance condition that is at the basis of the Metropolis algorithm that is at the basis of simulated annealing.)

Check by yourself that the probability distributions produced by the two methods are indeed different, by writing a small script with two small functions and collecting some data (e.g. use `n==10` and `10**5` samples). You can then do a histogram of the distribution of `i` and `j` in the two cases, and verify that they are different. Use the `plt.hist` function, with the keyword argument `bins=n-1`.

Extra: if you want to convince yourself "visually" that the first version is correct, you can use the `hist2d` function of pyplot, using the option `bins=(n-1,n-1)`. The result should look like a table with an upper triangular part which is nearly uniform in color (all yellow) and a lower triangle which is also uniform (all violet). If you do the same with the wrong version, you will see that the upper triangle color is not uniform, but rather banded, meaning that some particular pair of indices are chosen more often than others.

Tip: as a general rule of thumb, also for the next exercises, you're safer in most situations if you apply this same scheme: "keep extracting moves randomly with a flat measure, and reject the invalid extractions, until you get a good one".

3. A worse TSP

Create a new class, which you can call `TSP_SC`. The `SC` stands for "swap cities".

The idea is to try out a different kind of move, i.e. swapping two cities along the route instead of crossing two links.

Therefore, the only thing you need to change are the methods involving the moves.

You can start by just copy-pasting the TSP class we wrote in its entirety, and modifying the relevant methods. *Extra:* Or, if you're brave, you can just use inheritance: the best way to do that is by creating a "base class" (without any functionality related to optimization like the methods to propose and accept moves and computing delta-costs, but still with a route and a cost) and then inherit from it in two different ways, one for each type of move you want to try.

So now to produce a move you just need to choose two indices at random between `0` (included) and `n` (excluded). (Make sure they're different.) Call the indices `i` and `j`. The move would swap the two cities at positions `i` and `j` in the route.

Then, you need to compute the cost of such move. Start out by doing it in the most trivial way, as in the debug code which is in `greedy`: get the old cost with the `cost` function; copy the problem into a new one, swap the cities on the copy and get the new cost, and return the cost difference. This is inefficient but it's easy to do and it works.

Then compute the cost more efficiently: only a small part of the route is affected. Note that this is more difficult than in the previous cross-links case, mainly because there are more terms involved. You also need to consider different possibilities, depending on the chosen cities. Analyze the problem with small examples (8 cities or so), consider several situations, etc. Make sure that your new code gives you the same result as the trivial one, except at most for floating-point approximations. You can also use the `debug_delta_cost` option in the solver. Note that it is *very easy* to get it wrong. It's likely that you will get it right a few times and then find a case when it's wrong. If so, try to understand why and fix it. Keep going until you have fixed all the cases, then you can abandon the trivial code.

Experiment with this new move scheme. You can try to compare the two, city-swap vs cross-links, and see which is better. Try different problem sizes and see how things change as you increase the size.

4. Latin squares, attempt 1

Open a new file, called `LatinSquare.py`. Inside that file, write a class called `LatinSquare`, like for TSP. As for TSP, you need to use an auxiliary script, call it e.g. `lsqrun.py` or something like that.

The constructor of the class needs to take one positive integer argument `n`, and create a square `n` × `n` table. The table will represent the configuration (the analog of the route for TSP) and needs to be stored as an attribute in the `self` object. Create an `init_config` method that fills that table with random numbers between `0` (included) and `n` (excluded). Use `np.random.randint`, just once, to do this. (Be sure to fill in the *contents* of the table, as in the TSP example, see the comments there.) Be sure to call this function in the constructor, as we did for TSP.

For convenience, immediately write a `__repr__` method that returns the table in string form, so that you can easily see the contents of your object. To make the `greedy` code work, just also write a `display` function that

does nothing. And immediately write a `copy` method too.

The goal here is that we want the internal table to have a particular structure: each row and each column should have all the numbers between `0` and `n-1`. Basically, like a Sudoku, but without the sub-squares. See the Wikipedia entry. (We start from `0` rather than `1` though, contrary to Sudoku and the Wikipedia examples.)

Here is an example of what we want, with `n==4`:

```
0 1 3 2
2 0 1 3
1 3 2 0
3 2 0 1
```

Let us say that the cost function in this case is defined like this: it's the number of rows and columns which violate the constraints. For example, the following table has cost `3` (violations are row `1` and columns `0` and `2`):

```
0 1 2 ✔
1 2 2 ✖
1 0 2 ✔
✖ ✔ ✖
```

With this definition, the minimum possible cost is zero, and if we get that we have achieved our goal.

In order to compute the cost of each column/row quickly, write a small auxiliary function (not a method!) which takes two arguments, a 1d-array and a number `n`. Use the `numpy` function `unique` to determine whether the array contains all the numbers between `0` and `n-1`: if it does, the result of `unique` should have length `n`. Then write a `cost` method in which you apply this function to each row and column. Remember to test this method thoroughly: create a small problem, print it (the `__repr__` method will print its table), inspect it visually and check if the cost is right. Do it a few times. You can even change the table entries yourself by hand to do additional checks. Once you're sure it works, move on.

Next, write `propose_move`. The move proposal could be: pick an entry at random (thus, pick a random row and a random column) and change its value at random into one of the other possible values. Make sure that the new value is different than the old one. For example, if the current value is `3`, and `n==9`, then you would set it to any number between `0` and `2` or between `4` and `8`. To represent such a move, you need to provide 3 items: the row and column indices, and the new value. Store all that in a tuple. After this, it's easy to write the `accept_move` method. Be sure to debug what you did by performing some manual tests in the console, producing moves and accepting them a few times.

Advanced: Can you think of a way to choose a new value with a single random extraction, i.e. with no loops and rejections involved, and which makes sure that the resulting value is chosen with uniform probability among the allowed ones?

Next you need to write `compute_delta_cost`. Start with the naive approach: get the old cost, create a copy, accept the move on the copy and get the new cost, return the cost difference. Then write an efficient version: this one must only involve the row and the column that are affected. Use the auxiliary function described above for this. Debug thoroughly.

Very much advanced: maybe you can come up with a better, improved way to compute the cost and the cost updates, by using some auxiliary structures.

Do a few tests with this scheme. It should work, maybe with some effort, for very small values of `n`, but if you go to, say, `n==15` it should be very unlikely that it succeeds in any reasonable amount of time.

5. Latin squares, attempt 2

Modify the previous exercise. Copy-and-paste the code into a new file, and create a new test script to run experiments. This time we try to reduce the space of configurations, to make the search for a solution much more effective. The idea is to limit ourselves to only a subset of all configurations by (1) starting off in a particular state

that is within that subset, and (2) only using moves that keep us within the subset. If we know that the solutions that we're looking for are in the subset, this can be a very effective technique.

In this particular case the subset is: explore all the tables in which each column is a permutation of the numbers `0` to `n-1`. Clearly any solution to our Latin Square problem is like that, it just has the additional condition that the rows are also permutations. In this way we simply need to fix the rows.

Here are the changes that you need to do:

1. First, change the initialization. Instead of producing a random table, start off with a table in which each column has the numbers between `0` and `n-1`, in order. Now each column is automatically valid, but all the rows violate the constraints (well, unless `n` has a particular value, if you really think about it…). Thus, the initial cost is `n` (one per row). You should be able to build such a matrix in one relatively short line of code, using broadcasting (or other techniques, there are several ways…). If not, just use loops and maybe come back to this point later.
2. Change the move. Instead of changing the value of one entry, choose one column at random and swap two entries at random in that column. Why? Because this ensures that the column still contains all the elements, and thus that the cost of each column is still zero. Thus, you only need to compute the cost change of the two affected rows. (Keep using the auxiliary function written before.)

Of course, the `propose_move` method must be changed: this time you need to extract a random column, and then extract two (different!) random rows. The representation of the move must reflect that: it's still 3 integers, but very different from before. The `accept_move` method must be changed but it's still easy. The `compute_delta_cost` method is also quite short, but it needs care, be sure to use the `debug_delta_cost` option until you're reasonably sure that everything works as intended.

Trying to solve the problem in this new way should be much better than the previous one. It should also be clear to you *why* it is better, at an intuitive level.

6. Latin squares, attempt 3

Modify again the previous exercise. Copy-and-paste the code, and create a new test script. This time, we try to "refine" the cost function. The previous attempts had the following problem: they did not distinguish between a "completely wrong" row, e.g. `[0,0,0,0,0]`, and an "almost correct" one, e.g. `[0,4,2,1,1]`. This means that while the algorithm searches at random, it needs to "stumble" upon one of the few valid configurations; it will not "be aware" whether it is close to fixing a row or not.

So now do this: simply change the auxiliary function that you use to compute the costs of each row. Instead of returning `0` for a correct row and `1` for an incorrect one, return the difference between `n` and the number of unique elements of the input array. The result is still `0` for correct rows and positive for incorrect ones, but it now penalizes more rows that are very incorrect, and less those that are just barely wrong. If you did everything as suggested previously, this will affect very few lines of code (between 1 and 4, depending how you wrote the functions).

Experiment with this version, the performances should improve rather dramatically, especially at large-ish `n`. This demonstrates the importance of choosing a suitable cost function (when you have the freedom to do so like in our case), as well as a suitable representation and set of moves.

7. Sudoku (generator) [slightly more advanced]

Copy and paste the previous code for Latin Squares, the third attempt, in a new file. Change the name of the class to `Sudoku`. Also create a new auxiliary script, as usual. In the constructor of the class, add a check that the input `n` is a perfect square integer, e.g. `1`, `4`, `9`, `16` etc.

Our goal here is to generate a valid Sudoku puzzle, starting from scratch (but not to solve one yet).

The only difference with the previous exercise is that now we want to **also** enforce additional constraints. Call `sn==√n`: then the main table can be divided into `sn` × `sn` sub-squares, each of which has size `sn` × `sn` and thus

contains `n` elements. We want each of these sub-squares to also contain all the numbers between `0` and `n-1`.

Here are two examples with `n==4` and `n==9`, with the sub-squares marked (note that each row and column is also valid).

```
3 1 | 0 2     4 0 2 | 6 3 7 | 8 1 5
2 0 | 1 3     7 8 1 | 5 2 0 | 4 6 3
-----+-----     3 5 6 | 4 1 8 | 0 7 2
1 2 | 3 0     -------+-------+-------
0 3 | 2 1     2 3 8 | 1 4 5 | 7 0 6
              5 4 7 | 3 0 6 | 1 2 8
              6 1 0 | 8 7 2 | 3 5 4
              -------+-------+-------
              0 2 3 | 7 5 4 | 6 8 1
              8 7 4 | 2 6 1 | 5 3 0
              1 6 5 | 0 8 3 | 2 4 7
```

You can basically use everything that you wrote for the case of the Latin Square almost as-is. You need two major new things:

1. We have `sn` × `sn` additional cost terms, one per each sub-square. To compute the cost of one sub-square, you need to be able to get it. Given two sub-square indices `si` and `sj`, both ranging between `0` and `sn-1`, you want to get a **view** of the correct elements in the original array. So you need to use slicing. For example, the first sub-square (sub-indices `0,0`) in a `9` × `9` grid would be indexed as `[0:3,0:3]`; the sub-square to its right (sub-indices `0,1`) would be indexed as `[0:3,4:6]`, etc. So you need to compute correctly the starting and ending indices for each of the two slices depending on `si` and `sj`. You also want to take a sub-square and pass it to the auxiliary function that computes the costs, and which up to now was expecting 1-d arrays. Luckily, it should still work even if you pass it a 2-d array view, because `np.unique` works the same whatever the dimension of the array that you pass to it.
2. You need to be careful that now each move affects two rows (like before), and either one or two sub-squares (so now you have two cases to consider in `compute_delta_cost`). For any given table entry, you need to compute the sub-indices, i.e. the indices of the sub-square in which the entry is located. E.g. in the `n==9` case, the entry `(i,j)==(6,4)` is located in the sub-square `(si,sj)==(2,1)`.

8. Sudoku (solver) [rather advanced]

The goal now is to use the previous code to generate a valid Sudoku scheme, then we want to keep only a fraction of the entries and discard the rest.

This will lave us with a Sudoku puzzle, and by construction one that can surely be solved.

(Here, solving the puzzle means finding any valid assignment. Most published puzzles have only one solution; in our case, there may well be several solutions, different from the one that we started from.)

So you need to change your code in this way: in the constructor, you need to accept either an integer number or another Sudoku object as an argument. If the constructor gets a number, it should behave as before. If instead it receives a Sudoku object, it should choose at random a fraction of the entries to keep fixed, and shuffle the rest.

Here is how you do it. Accept an optional extra argument `r` in the constructor: this will need to be between `0` and `1`, and represent the fraction of entries you keep. Create an auxiliary `n` × `n` table of `bool`s (let's call it a "mask"), and for each entry randomly pick either `True` with probability `r` or `False` with probability `1-r`. (*Extra:* this technique will produce `r*n*n` fixed entries, but only *on average*; try to come up with a way to produce exactly `round(r*n*n)` fixed entries instead, still ensuring a uniform distribution; hint: this involves using random permutations and array views.)

The `True` entries of the mask will correspond to the fixed entries in the puzzle (i.e. the initial numbers that are usually given in the published puzzles).

Still in the constructor, you then need to shuffle each column (permute randomly its elements), but you also need to make sure that you never touch the entries that are fixed, those marked as `True` in the mask. The easiest way is to use a "sufficient" number of swaps: just pick random pairs of entries, making sure that they are not fixed, and swap them, and repeat this (say) a hundred times. However there is a harder, but correct and efficient way to do the "shuffle with fixed elements": it involves generating a random permutation of the non-fixed entries, and applying it.

You also need to modify the code for the proposal of the move: you need to ensure that you will never change the fixed entries (those marked `True` in the mask). (*Tip:* the code for this and the one for the "easy way" of shuffling the columns in the constructor are quite similar.)

What kind of problems would you expect to be able to solve this way (in a reasonable time), i.e. what fractions `r` do you expect to work? Check your expectations (at various system sizes, e.g. `4`, `9`, `16`, …).

9. Sudoku solver, slight improvements.

Instead of "cheating" by passing a solved Sudoku puzzle to the constructor, create a method for the Sudoku class that returns just a table that is a copy of the internal Sudoku table but with only a random fraction `r` of the entries, while the remaining ones are set to a nonsensical value, e.g. to `-1`. Call this method `gen_puzzle`, for example.

Then, change the constructor to either accept an integer number as its argument, or a 2d-array. When it gets the 2d-array, you need to "read" the mask from the table (the non-negative entries), and randomly initialize the rest. The tricky part here is that you need to make sure that the initialization is valid, i.e. that each column contains all elements between `0` and `n-1`. This requires some work. (Obviously, compared to the previous version, you would also remove the argument `r` from the constructor, since that was now moved to `gen_puzzle`.)

This would allow you to generate puzzles as before (by first generating a valid scheme), but also to potentially pass an arbitrary puzzle (e.g. you can create one by hand by copying it from somewhere, or read it from a file…)

10. Weighted MaxCut

Say you are given a symmetric real matrix with non-negative entries, which represent the weights on each edge of a complete graph. (You can easily generate one such matrix yourselves, of course, say with uniform weights in `[0,1)`.)

Your task is to implement the weighted MaxCut optimization problem. Note that you are trying to find a maximum, but our code seeks a minimum of the cost. So, for starters, you need to make sure to define the cost such that it's minimum when the cut is maximum (this is really trivial, if you have computed the weight of the cut).

Make your own choices about how to represent the internal configuration, the moves, how to compute the `delta_cost` efficiently etc. Test everything and debug it thoroughly. Then see what results you get when you try to solve it.

11. Magic squares [hard]

Let's say I want a Magic square of size `n` in which each row, column and the two diagonals sum to some given number `s`. All entries must be integer. However, to start with, do not put any kind of constraint on the entries (they do not have any minimum or maximum value, contrary to the standard definition).

Write a class that allows you to frame it as an optimization problem, basically like we did for the latin square series of exercises.

Comments:

1. You can come up with your own choice for the cost function, it just needs to be `0` when all constraints are satisfied, and positive otherwise. Same for the moves.

2. Note that for some combination of `n` and `s` there will be no solutions at all, particularly when you impose the constraints from points 3 and 4 below. In general, for a given `n`, how would you expect the difficulty to change with `s`? I.e. does it get simpler or harder if you increase `s`? Verify your intuitions as you proceed.

3. Of course, if you are allowed to repeat the entries, the problem may have some trivial solutions, e.g. if `s==n` you could just set all elements to `1`. You can start by just allowing this possibility, but if you can you should try to add the constraint that all entries of the square must be different. (In this context, adding a constraint is done by imposing some kind of additional cost when the constraint is violated.) This is non-trivial to code, and makes the problem much harder to solve as well.

4. The problem is also much easier if you allow negative numbers. A further complication is then to only allow positive numbers (still without upper bounds though).