📖 **exercises**

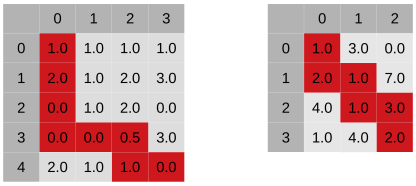
Find the optimal path via dynamic programming

Setup: the problem

We consider this problem:

- given a matrix w (of size $n \times m$) whose elements are non-negative ($w_{ij} \in \mathbb{R}$, $w_{ij} \geq 0$)
- find the path of minimal cost that connects the top-left corner with the bottom-right corner. The path can only move down or to the right, not diagonally.

See the examples in the figure below:



	0	1	2	3
0	1.0	1.0	1.0	1.0
1	2.0	1.0	2.0	3.0
2	0.0	1.0	2.0	0.0
3	0.0	0.0	0.5	3.0
4	2.0	1.0	1.0	0.0

`optc = 4.5`

	0	1	2
0	1.0	3.0	0.0
1	2.0	1.0	7.0
2	4.0	1.0	3.0
3	1.0	4.0	2.0

`optc = 10.0`

The dynamic programming scheme

This problem can be solved efficiently with dynamic programming. In the directory called `"exercises"` you'll find a file called `"findpath.py"` that contains a partial implementation.

The crucial idea is to have an auxiliary matrix, call it c, of size $n \times m$, whose element i, j represents the solution of the problem for the sub-matrix `w[0:(i+1), 0:(j+1)]`.

We have the following recursive relations:

1. $c_{i0} = \sum_{k=0}^{i} w_{k0}$ for $0 \leq i < n$
2. $c_{0j} = \sum_{k=0}^{j} w_{0k}$ for $0 \leq j < m$
3. $c_{ij} = w_{ij} + \min\left(c_{(i-1),j}, c_{i,(j-1)}\right)$ if $i, j > 0$

The first two cases are the base cases of the recursion.

As usual in dynamic programming, we also need another auxiliary matrix `whence` too keep track of the choices made at each step. This matrix will have size $n \times m$ and be made of integers. We will represent with `-1` the "left" choice, and with `1` the "up" choice. The top-left corner has no predecessor and thus will contain a sentinel value.

The code

Large parts of this code are already implemented in the function `findpath`, but the implementation is incomplete. After that, there are a few definitions of test functions (that you can ignore), and at the very end of the file those test functions are called, but the calls are commented out. If you uncomment them, they will raise errors, but solving the tasks will fix them.

Your tasks are described below. In the code, you can find them by looking for the string `"TASK"` in the comments.

1. In the argument checks at the beginning, make sure that the matrix is not empty (has more than zero rows/columns) and that there are no negative entries. For the full score, do this without for loops. If you do this correctly, and uncomment the call to `test1` at the end of the file, it should run without errors.

2. Implement the two base cases for the c matrix, i.e. fill the first row and column. For the full score, don't use

any for loops. If you do this correctly, and uncomment the call to `test2` at the end of the file, it should run without errors.

Note that this task is crucial for the rest.

3. The computation of the costs in the main dynamic programming loop is (clearly) incorrect. Fix it. Basically you need to implement the third case of the recursion. You can check what you did by uncommenting the call to `test3` at the end of the file.

4. The path reconstruction code is very much incomplete. Fix it. After you have done this, the variable `path` should be a list of size $n + m - 2$, containing only the values `-1` and `1`, where `-1` means "horizontal movement" and `1` means "vertical movement", c.f. the main dynamic programming loop. (Therefore there should be $n - 1$ ones, and the remaining $m - 1$ entries should be minus ones.) The order should be such that, starting from the $0, 0$ corner, one should be able to read one element at a time from `path` and reach the bottom-right corner. Check your code by uncommenting `test4` at the end of the file.

5. Also reconstruct the costs encountered along the path. Produce another list, called `pcosts`, and fill it with the costs encountered along the path, starting from the top-left and ending at the bottom-right corner. Also, insert a check (using an assertion) to ensure that the list computed in this way is consistent with the optimal cost obtained previously. Check your code by uncommenting the `test5` call at the end of the file.

6. The current code favors the "left" case over the "top" case, in case of equal costs. Revert this precedence rule by favoring "top" instead. Check your code by uncommenting the `test6` call at the end of the file.

A continuous optimization problem

In the directory called `"exercises"` you'll find the file `"optim.py"`. We will use it to solve a continuous optimization problem. In this problem, we are again given an $n \times m$ matrix called w. This time however, we will optimize over a real vector x of length $n + m$. The vector is made of two parts, one of n nodes and one of m nodes, concatenated together. Let's call them u and v:

1. $u_i = x_i$ for $0 \leq i < n$
2. $v_i = x_{i+n}$ for $0 \leq i < m$

The function that we want to optimize is:

$$f(x) = \sum_{i=0}^{m+n-1} x_i^4 + \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_{ij}\, u_i\, v_j$$

1. Your first task is to write that function. For the full score, don't use any for loops (this will require some broadcasting "tricks" of the sort we have seen during the lectures). Notice that in the code `f` has two inputs, `x` and `w`.

2. In the script, a matrix w 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 `0.0` (look at the `"fun:"` line in the output, and the `"x:"` line too). The optimal cost should not be `0.0`. 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 not give the absolute best value of `f`, unless you are 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.

Theory questions

Answer on paper to the questions below.

1. Consider the quadratic function $f : \mathbb{R}^n \to \mathbb{R}$ to be minimized defined by:

$$f(x) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j + \sum_{i=1}^{n} b_i x_i = \frac{1}{2} x^T A x + b^T x$$

where A is a positive definite matrix and b is a vector.

- Express the gradient of f in a generic point x.
- Starting from some point x, express the new position x' after on one iteration of gradient descent with step size $\alpha = 0.5$ in terms of x.

2. The PageRank algorithm can be understood as a Markov chain on a directed graph of N web pages with transition matrix $Q_{ij} = A_{ij}/k_j$, where A is an adjacency matrix:

$$A_{ij} = \begin{cases} 1 & \text{if a link from webpage } j \text{ to webpage } i \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

and $k_j = \sum_{i=1}^{N} A_{ij}$ is the out-degree of vertex j.

In the very peculiar case in which the matrix A_{ij} is symmetric, and with the further assumption that the graph is connected, it is easy to guess the stationary distribution. With these assumptions, solve the following exercises:

- Find the stationary distribution ρ and prove its stationarity.
- Introduce a parameter $\beta \geq 0$ and consider a generalized page rank algorithm with transition matrix $Q^{(\beta)}$ derived from the previous one as follows:

$$Q_{ij}^{(\beta)} = Q_{ij} \, \min\left(1, \frac{k_i^\beta}{k_j^\beta}\frac{Q_{ji}}{Q_{ij}}\right) \qquad \forall i, j \text{ with i} \neq j.$$

In the case $Q_{ij} = Q_{ji} = 0$, the equation above has to be intended as $Q_{ij}^{(\beta)} = 0$. Also, $Q_{jj}^{(\beta)}$ is fixed by $Q_{jj}^{(\beta)} = 1 - \sum_{i \neq j} Q_{ij}$.

In this case, what would you expect the stationary distribution $\rho^{(\beta)}$ to be? What happens to the stationary distribution in the limits $\beta \to 0$ and $\beta \to +\infty$?