Shortest path with Simulated Annealing

Setup: the problem

We generate a graph with n nodes. The distance between each pair of nodes is generated uniformly at random and stored in a symmetric matrix called `dist`. We want to find the shortest path from the first node, labeled `0`, to the last one, labeled `n-1`.

We'll do it with Simulated Annealing (this is a terrible idea of course, there are efficient algorithms for this problem that guarantee optimality).

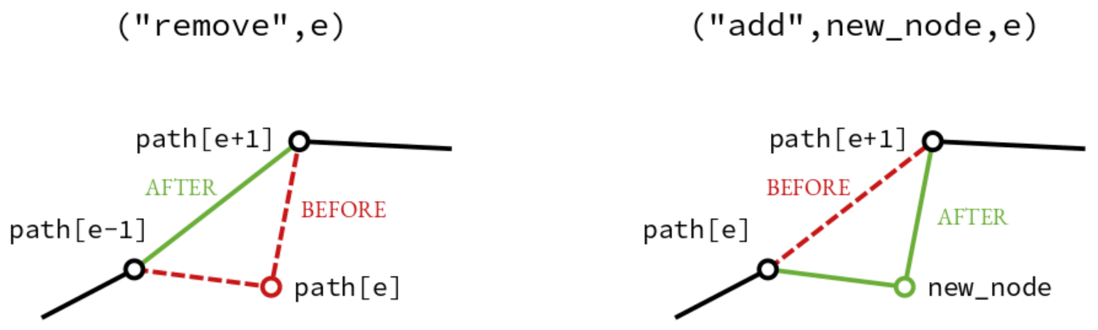
Our configuration will be a path, represented as a list of k node labels. The first node of the list is always `0` and the last one is always `n-1`. For example, say $n = 20$, then a valid path may be `path = [0, 6, 11, 5, 19]`.

We will denote the indices in the path with the letter `e` (for "edge"), to help distinguishing them from the node labels. The edge `e` connects the node `path[e]` to `path[e+1]`. Thus the cost of the path in mathematical notation is $\sum_{e=0}^{k-2}$ `dist[path[e], path[e+1]]`.

The moves for the Markov Chain process will be of two possible kinds: either we remove an intermediate node from the path, or we add one. The moves are illustrated in the figure below.

To remove a node, we choose at random an index `e` in the path, avoiding the first and the last ones. Thus we will end up connecting the nodes `path[e-1]` with `path[e+1]`. The proposal will be encoded in a tuple `("remove", e)`.

To add a node, we pick at random both a node (provided it's not already in the path), call it `new_node`, and an insertion point `e` in the path. The new node will end up between the nodes `path[e]` and `path[e+1]`. The proposal will be encoded in a tuple `("add", new_node, e)`.



We will choose which kind of move to propose, remove or add, randomly with 50% chance (except in extreme cases when one of the operations is impossible, when there are too few or too many nodes in the path).

Since the length of the path will change at each move, and numpy arrays are not good at changing their length dynamically, we will use the following trick: we keep an array called `path` of length `n`, but we will only use its first `k` elements. The rest of the elements will be set to a sentinel value of `-1`. We thus also need to keep around (and update at each move) the value of `k`.

Large parts of the code are already written. Below are the tasks that you need to complete.

Tasks

You'll need to edit the `"ShortestPath.py"` file to do the tasks.

The `"shortestpathrun.py"` can be used for testing the code: there is a test function for each task, which will keep producing errors until you complete the task (and when you do it prints a `"test ok"` message). The tests are called after the definitions, you can comment out the tests for a task in order to skip it. After the individual tasks tests, there is a script that runs Simulated Annealing. It will only work properly if you complete all the tasks.

The `"SimAnn.py"` file is the same that we had at the end of the lectures and you don't need to look at it if you remember how it works.

Advice: before starting, skim through the `"ShortestPath.py"` code once.

1. Complete the implementation of the `cost` function, according to the formula above. Make sure that you are summing `k-1` terms, one per each edge. You can do this with a `for` loop, no need for fancy numpy tricks.
2. In `propose_move`, the `"add"` move is already written, but the `"remove"` part is not. Complete it according to the description in the text. One (easy) line of code.
3. In `compute_delta_cost`, the `"add"` case is already written, but the `"remove"` case is incomplete. You need to complete it.
4. In `accept_move`, the `"add"` case is already written, but the `"remove"` case is incomplete. You need to complete it. Notice that you must ensure to update the internal value of `k`, and pay attention to the sentinel value.

After you complete the tasks, the test script should run without errors. You can disable the `debug_delta_cost` option at this point, and it should run fairly quickly and find an optimal solution with `k=5` and a cost of about `0.2282`).

Fit a function from some data

In the directory called `"exercises"` you'll find the file `"optim.py"`. In that file, there is a function f whose definition is incomplete. It should be defined as follows:

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$f(x) = \tanh\left(\sum_{i=0}^{n-1} w_i x_i + b\right)$$

Notice that x and w are vectors of length n, while b is a scalar.

1. Your first task is to complete the definition. For the full score, don't use any for loops. Until you do that, running the code will give you an `AssertionError`.

Our next goal is: given some data points $x^{\text{train}}, y^{\text{train}}$ (the y are the scalar outputs) we wish to find the values of the parameters w and b such that f best fits this data.

In the code, we set $n = 20$, then some values w_{true} and b_{true} are generated, and with these some random noisy training data is produced.

Then, a cost function `fit_error` is defined. In the cost function, the argument `z` represents a numpy array of length $n + 1$ that contains the values of w followed by b. The remaining two arguments are the training data. It computes the mean square error.

2. You should use scipy's `optimize` to find the value of `z` that minimizes the error. As a starting point, we pick w randomly and set $b = 0$. You should print the optimal values of w and b at the end.

   For the full score, you should:

   - Use the initial values for `w` and `b`, putting them together in an array `z0`
   - Use a `lambda` expression to transform the function `fit_error` as defined in the code into a function of a single argument, with the remaining ones being fixed, and call `minimize`
   - Extract the optimal parameters called `w_opt` and `b_opt` from the solution (which are then printed, alongside the true values, so that they can be compared - they should be pretty close)

Theory questions

(Answer the questions in the provided file `"theory.txt"`. You can do it in Spyder too.)

1. For each of the following sentences, tell which ones are true and which are false:

   a. A binary heap is usually implemented as an implicit data structure

   b. A binary search tree is always automatically balanced, and thus the depth of the tree is guaranteed to be $O\left(\log n\right)$.

   c. A binary heap tree is always automatically balanced, and thus the depth of the tree is guaranteed to be $O\left(\log n\right)$.

   d. In a given binary search tree, finding both the min and max values is efficient, $O(1)$.

   e. In a given binary heap, finding both the min and max values is efficient, $O(1)$.

2. Our implementation of the Simulated Annealing algorithm for the shortest path (first programming exercise) has a problem: the code in `"SimAnn.py"` assumes that the proposals probabilities are symmetric, but in fact in our `"ShortestPath.py"` code this is not the case. As a result, the stationary distribution of the MCMC process at fixed β is not the Boltzmann distribution that we have seen in class.

   Answer briefly to the following questions (no justifications required):

   a. Suppose that n is large enough (e.g. $n = 20$) and that your current configuration (the path) is $x = [0, n-1]$. Now consider a new configuration $x' = [0, j, n-1]$ for some $j \in \{1, 2, \dots, n-2\}$. What is the probability of proposing the transition $x \to x'$ with our code?

   b. Now consider the same situation in reverse. If your current configuration is x', what is the probability of proposing the transition $x' \to x$?

   c. Suppose that you knew the formula for the probability of the proposals $C\left(x \to x'\right)$ for any two configurations x, x'. At what point of the Simulated Annealing algorithm should you make use of this information, in order to fix the wrong assumption?