Ascending Paths

It's time to show you how the Codility challenge code-named Magnesium can be solved. You can still give it a try, but no certificate will be granted. The problem is to find, in a weighted graph, the longest path in which the weights are ascending.

Unique weights

First we will solve a slightly easier version of this problem, in which the weights of all the edges are different. For each vertex v in the graph we will calculate the value $w[v]$, which denotes the length of the longest ascending path starting at this vertex.

Let uv be the unique edge of the lowest weight among all the edges in the graph. Suppose that we have removed this edge from the graph, and have calculated the values in array w. How will these values change after we reintroduce edge uv to the graph?

If edge uv belongs to some ascending path, then it must be the first edge on this path, since its weight is the smallest possible. Thus, the edge only affects paths which start in vertex v or vertex u. Such a path starts at one of these vertices, goes through edge uv and then travels along the longest ascending path starting in the other vertex. Therefore, the updated value of $w[v]$ is simply $\max(w[v], w[u] + 1)$ and, by symmetry, the updated value of $w[u]$ is $\max(w[u], w[v] + 1)$.

Repeating this reasoning, we can start with the empty graph and update array w by adding edges to the graph in decreasing order of their weights. In the following code, array *ind* contains the indexes from 0 to $M - 1$, sorted in descending order by the weights of the edges they refer to.

1: Unique weights — $O(N + M \log M)$

```
def solution_unique_weights(N, A, B, C):
    M = len(A)
    w = [0] * N
    ind = range(M)
    ind.sort(cmp=lambda i, j: C[j] - C[i])

    for i in xrange(M):
        u = A[ind[i]]
        v = B[ind[i]]
        w[u], w[v] = max(w[u], w[v] + 1), max(w[v], w[u] + 1)

    return max(w)
```

The important thing is to update both ends of the new edge at the same time (thus in line 10 we update $w[u]$ and $w[v]$ simultaneously).

Initializing array w takes $O(N)$ time. Sorting the list of M edges takes $O(M \log M)$ time. Each update is done in a constant time, so it takes $O(M)$ time to perform all of them. To summarize, the above algorithm runs in a total time complexity of $O(N + M \log M)$. The space complexity is $O(N + M)$.

General solution

The previous solution does not work if there is more than one edge of a given weight. All edges of the same weight must be added to the graph at the same time, otherwise some ascending path may contain more than one edge of the same weight. We do this by accumulating updates to perform in array *updates*, and then performing them only when the next edge to consider is of a smaller weight (or after adding the last edge to the graph). Each entry in array *updates* is a pair comprising some vertex v and the value of $w[v]$ after the update.

2: Golden solution — $O(N + M \log M)$

```
 1  def solution(N, A, B, C):
 2      M = len(A)
 3      w = [0] * N
 4      updates = []
 5      ind = range(M)
 6      ind.sort(cmp=lambda i, j: C[j] - C[i])
 7
 8      for i in xrange(M):
 9          u = A[ind[i]]
10          v = B[ind[i]]
11          updates.append((u, max(w[u], w[v] + 1)))
12          updates.append((v, max(w[v], w[u] + 1)))
13
14          if i == M - 1 or C[ind[i]] > C[ind[i+1]]:
15              for v, val in updates:
16                  w[v] = max(w[v], val)
17              updates = []
18
19      return max(w)
```

Note that several entries in array *updates* may refer to the same vertex v, so we must take into account the value of the largest update (that is why we use the max function in line 16).

The operations on array *updates* do not increase the time and space complexities of the solution.