codility

WE TEST CODERS

Wire Burnouts

Let us see how the Ψ (Psi) challenge can be solved. You can still give it a try, but no certificate will be granted.

In this task, there is a grid of wires through which an electric current flows between the lower-left and upper-right corners. The wires burn out sequentially in some order. The question is: how many wires must burn out to cause the current to stop flowing?

Checking whether the two corners are connected is a standard graph problem, and there are standard algorithms for checking graph connectivity: for example, DFS and BFS. Each of them requires $O(N^2)$ time, where N is the size of the grid. But there are up to $2N(N-1)$ wires that can burn out, and equally as many moments at which we should check whether the two corners are still connected. So the overall time complexity of such a naive solution is $O(N^4)$. Here is an implementation of such a solution in Python:

1: Repetitive graph search solution — $O(N^4)$

```
def check_connections(horizontal, vertical):
    N = len (horizontal)
    visited = [[False] * N for j in xrange(N)]

    def dfs(i, j):
        if (not visited[i][j]):
            visited[i][j] = True
            if vertical[i][j]:
                dfs(i, j+1)
            if horizontal[i][j]:
                dfs(i+1, j)
            if (i > 0 and horizontal[i-1][j]):
                dfs(i-1, j)
            if (j > 0 and vertical[i][j-1]):
                dfs(i, j-1)

    dfs(0, 0)
    return visited[N-1][N-1]

def wire_burnouts(N, A, B, C):
    vertical   = [[True] * N for j in xrange(N)]
    horizontal = [[True] * N for j in xrange(N)]
    for i in xrange(N):
        vertical[i][N-1]   = False
        horizontal[N-1][i] = False
    for t in xrange(len(A)):
        if C[t] == 0:
```

```
            vertical[A[t]][B[t]] = False
        else:
            horizontal[A[t]][B[t]] = False
        if not check_connections(horizontal, vertical):
            return t+1
    return -1
```

Unfortunately, such a solution fails on larger tests due to stack overflow. Also, there are faster solutions.

Firstly, we don't have to check all possible moments in time: once the current stops flowing it never starts flowing again. Hence, we can use a bisection to find the moment when the current stops flowing. Bisection requires us to check connectivity at $O(\log N)$ different moments in time, and each such check takes $O(N^2)$ time, so the overall time complexity is $O(N^2 \log N)$ time. Secondly, instead of using recursion, we can implement DFS using an explicit stack.

2: Bisection and graph searching — $O(N^2 \log N)$

```
def check_connections(horizontal, vertical):
    N = len (horizontal)
    visited = [[False] * N for j in xrange(N)]
    stack = [(0,0)] * (2 * N * N)
    sp = 1

    while sp > 0:
        sp -= 1
        (i,j) = stack[sp]
        if (not visited[i][j]):
            visited[i][j] = True
            if vertical[i][j]:
                stack[sp] = (i,j+1)
                sp += 1
            if horizontal[i][j]:
                stack[sp] = (i+1,j)
                sp += 1
            if (i > 0 and horizontal[i-1][j]):
                stack[sp] = (i-1,j)
                sp += 1
            if (j > 0 and vertical[i][j-1]):
                stack[sp] = (i,j-1)
                sp += 1

    return visited[N-1][N-1]

def wire_burnouts(N, A, B, C):

    def burn_wires(t):
        horizontal = [[True] * N for j in xrange(N)]
        vertical   = [[True] * N for j in xrange(N)]
        for i in xrange(N):
            vertical[i][N-1]   = False
            horizontal[N-1][i] = False
        for i in xrange(t):
            if C[i] == 0:
                vertical[A[i]][B[i]] = False
            else:
                horizontal[A[i]][B[i]] = False
        return check_connections(horizontal, vertical)
```

```
    def bisection(l, p):
        # Search for the first moment without a connection
        if l == p:
            burn_wires(p)
            if burn_wires(p):
                return -1
            else:
                return l
        else:
            m = (l+p)/2
            if burn_wires(m):
                return bisection(m+1, p)
            else:
                return bisection(l, m)

    return bisection(0, len(A))
```

Even so, this is still not the best possible solution. Instead of burning the wires out, we can reverse time, keep adding missing wires and check when the two corners are connected. For this approach the best data structure to use is a find–union tree. This is suitable for storing information about a set of elements (nodes of the grid) grouped into disjoint subsets (here, connected components). Using find–union trees, we can quickly check whether two elements are connected (find) or add a new wire (union). The amortized time cost of operations on such trees is $O(\log^* N)$. ($\log^*$ is the iterated logarithm, the number of times one has to iterate the $\log_2$ function in order to obtain a number not greater than 1. In practice, its value doesn't exceed 5 and can be treated as constant.) Hence, the overall time complexity of the solution is $O(N^2 \log^* N)$. Here is an implementation of such a solution:

3: Model solution — $O(N^2 \log^* N)$

```
def find((a,b)):
    global vertices, rank

    (c,d) = (a,b)
    while vertices[a][b] != (a,b):
        (a,b) = vertices[a][b]
    while vertices[c][d] != (a,b):
        (e,f) = vertices[c][d]
        vertices[c][d] = (a,b)
        (c,d) = (e,f)
    return (a,b)

def union((a,b), (c,d)):
    global vertices, rank
    (a,b) = find((a,b))
    (c,d) = find((c,d))
    if rank[a][b] < rank[c][d]:
        vertices[a][b] = vertices[c][d]
    else:
        vertices[c][d] = vertices[a][b]
        if rank[a][b] == rank[c][d]:
            rank[c][d] += 1

def wire_burnouts(N, A, B, C):
    global vertices, rank
    M = len(A)

```

```
        # Find-union data-structure
        vertices = [[(x,y) for y in xrange(N)] for x in xrange(N)]
        rank = [[0] * N] * N

        # Edges left at the end
        v_edges = [[True for y in xrange(N-1)] for x in xrange(N)]
        h_edges = [[True for y in xrange(N)] for x in xrange(N-1)]
        for i in xrange(M):
            if C[i] == 0:
                v_edges[A[i]][B[i]] = False
            else:
                h_edges[A[i]][B[i]] = False

        # Merge vertices connected at the end
        for i in xrange(N):
            for j in xrange(N):
                if i < N-1 and h_edges[i][j]:
                    union((i,j), (i+1,j))
                if j < N-1 and v_edges[i][j]:
                    union((i,j), (i,j+1))

        if find((0,0)) == find((N-1,N-1)):
            return -1

        # Simulate wires burning out, in a reversed order
        for i in xrange(M-1, -1, -1):
            if C[i] == 0:
                union((A[i],B[i]), (A[i],B[i]+1))
            else:
                union((A[i],B[i]), (A[i]+1,B[i]))
            if find((0,0)) == find((N-1,N-1)):
                return i+1
```