Task: Crocodile

Proposed by: Mihai Pătraşcu

1 Subtask 1: Trees

In this subtask, we can infer that the underlying graph is a tree and the exit chambers are precisely the leaves. This is simpler than the general case but will be instructive for the next subtasks. To begin, let $L(u, v)$ denote the time to travel along the corridor connecting node u to node v. For simplicity, we will root the tree at the starting node (chamber 0).

The crucial observation then is that in any successful escape plan, the instruction at each node, if reachable from the root, will always tell Somying to move further away from the root (otherwise, she can be forced to cycle around the underground city forever). This leads to a simple dynamic programming (DP) solution: Let $T[u]$ denote the best time to reach an exit chamber. First, $T[u] = 0$ if u is a leaf. Otherwise, if u has children $v_1, \dots, v_k$, ordered such that $T[v_1] + L(u, v_1) \leq T[v_2] + L(u, v_2) \leq \dots T[v_k] + L(u, v_k)$, then $T[u] = T[v_2] + L(u, v_2)$. The problem statement guarantees that $k \geq 2$.

It is easy to show inductively that $T[u]$ is indeed the best time starting at u. Specifically, we prove that first, Benjamas can reach an exit chamber from u in $T[u]$ time regardless of what the gatekeeper does, and second, the gatekeeper can force Benjamas to spend $T[u]$ time in the underground city. Clearly, if u is a leaf node and hence an exit chamber, $T[u]$ is 0. Assuming inductively that $T[v_i]$ is as claimed for each child v_i of u, we have that in time $T[v_i] + L(u, v_i)$, Benjamas can reach an exit from u, through (u, v_i), if the corridor is not blocked. Since the evil crocodile can only block one corridor at a time, he can force Benjamas to spend $T[v_2] + L(u, v_2)$, by blocking (u, v_1)—blocking any other corridor only helps Benjamas reach an exit faster, in $T[v_1] + L(u, v_1)$ time.

To compute this DP, we traverse the tree in postorder (i.e., the leaves are visited first and each parent is visited after all its children); the final answer is stored in $T[0]$. The computation at each node involves finding the second smallest value, which can be done in $O(d_u)$ time. Here, d_u denotes the (out-)degree of u. Therefore, the total running time is $C \cdot \sum_u d_u = O(N)$, for some positive constant C, because the degrees of the tree nodes sum to $2(N - 1)$.

2 Subtasks 2 and 3: General Graphs

The challenge in generalizing our current algorithm to the general setting is the lack of a clear sense of direction; in our current algorithm, we know Benjamas must always move further from the root as necessitated by the tree structure.

A moment's thought reveals striking similarity between Dijkstra's single-source shortest path algorithm and our algorithm for trees. Indeed, the algorithm iteratively grows the frontier set, where at any point in time, a node u is in the set if $T[u]$ has been determined. From this view, our algorithm for trees can be seen as running Dijkstra's algorithm starting from the exit chambers. The algorithm is standard except for how the cost at a node is defined.

Consider the following algorithm: For all nodes u, set $T[u]$ to ∞ except when u is an exit chamber, set $T[u] = 0$. Initially, the frontier set S contains exactly the exit chambers. During the execution of the algorithm, we maintain that for $w \notin S$, the cost of w can be *conceptually* computed by producing the list[1] $\{v \in N(w) : T[v] + L(w, v)\}$, sorting this list, and returning the second value (i.e., the second smallest value in this list). When a node u enters the frontier (it has the lowest cost among non-frontier nodes), $T[u]$ is set to the cost of u at that moment.

Claim 1. *For each node u, Benjamas can reach an exit from u in $T[u]$ time regardless of what the gatekeeper does. Furthermore, the crocodile gatekeeper can force Benjamas to spend $T[u]$ time in the underground city.*

This claim can be shown by a similar inductive argument as in the tree case and by observing that as in Dijkstra's algorithm, once a node enters the frontier its cost cannot decrease because the edges have positive cost.

Implementation Details. For each node, we can maintain its cost by keeping two numbers—the smallest value and the second smallest value—which can be updated in $O(1)$ when the neighboring values change. Using this, Dijkstra's algorithm takes $O((M + N) \log N)$ using a heap or $O(N^2)$ without using one.

[1] $N(w)$ denotes the set of neighbors of w.