Task: Race

Proposed by: Martin Fixman

Given a tree T with N nodes, this task asks for a path P^* of length K with the minimum number of edges. It looks like a usual dynamic programming task. However, when K is large, another approach is required.

The model solution for this task follows a divide-and-conquer approach.

Consider a node u in the graph. There are two possible cases: when node u belongs to the solution path P^*; or when node u does *not*.

In the second case, we can delete node u from the tree and break it into smaller trees. We can then recurse on each of the resulted trees to find the solution.

With this general approach in mind, we have to answer the following questions.

- How to find the best path that contains node u?
- How to choose u to achieve a better running time?

Note that the second question is very important because if we can guarantee that the sizes of all resulting trees are small, we can bound the number of recursion levels.

1 Finding the solution containing u

Consider the case that P^* contains node u. Let's consider a simpler case where we only want to find if there exists a path of length exactly K that contains u.

If u is one of the endpoints in P^*, we can find the path using one application of depth first search (DFS).

However, if u is "inside" P^*, then two of u's adjacent nodes x and y must also be in P^*. Thus, we shall find x and y.

Consider some node w adjacent to u. With one application of DFS, we can find the set L_w of all path lengths for all paths starting at u and containing edge (u, w).

Hence, to find x and y, we need to find two nodes x and y such that there exists a pair $\ell_x \in L_x$ and $\ell_y \in L_y$ for which $\ell_x + \ell_y = K$. This can be done by DFS from u through every edge (u, w) for all adjacent nodes w with careful book keeping using an array $A[0, \ldots, K]$ of size $K + 1$.

The running time for this step is $O(N)$.

2 Finding the right node

Our goal is to find node u such that after deleting u, each resulting trees are all sufficiently "small." In this case, we shall find node u such that each remaining tree has at most $N/2$ nodes. We shall refer to node u as the *central node*.

It is not clear if such a node exists. So let's argue about that first.

Pick an arbitrary node v as a candidate. Denote by $T' = T \setminus \{v\}$ the forest obtained by deleting v from T. For each node w adjacent to v, denote by T_w the tree containing w in T'. If every tree $T_w \in T'$ has at most $N/2$ nodes, we are done and v is the required central node.

Otherwise there exists one tree T_w that contains more than $N/2$ nodes. (Note that there can be only one tree violating our criteria.) In this case, we pick w as our new candidate and repeat the process.

This process will eventually stop at some candidate node and that's the required central node. To see this, note that after leaving v, we shall never go back to pick v again; since there are N nodes, the process can repeat at most N times.

After knowing that the central node exists, there are many ways to find it. We can follow the process directly as in the argument. But this is too slow to be useful.

The following are two procedures that find the central node in $O(N \log N)$ time and $O(N)$ time.

2.1 Bottom-up approach

We can find node u in a bottom-up fashion. We shall keep a priority queue Q of all "processed" subtrees using their sizes as weights.

We maintain, for each node, its state which can either be *new* or *processed*; initially all nodes are new. Every node also has a weight. Initially every node has weight of 1.

We start with all leaf nodes in Q. Note that each node in Q is every node which has all but one of its adjacent nodes processed. For each node $v \in Q$, we denote by $p(v)$ the unique neighbor of v which is new.

While there are nodes in Q, we extract node v with the smallest weight. We update v state to processed and increase the weight of $p(v)$ by v's weight. If all but one neighbor of $p(v)$ are processed, we insert v into Q.

Using this procedure, the last node inserted to Q is the desired central node.

2.2 DFS with bookkeeping

With DFS and a good bookkeeping, we can find the central node in $O(N)$.

We pick an arbitrary node r to start a DFS. With this procedure, we can consider T as rooted at r and the parent-child relationship between adjacent pairs of nodes are clearly defined. While performing DFS, we compute, for each node v, the number of its descendants $D(v)$.

With this information, we can figure out if a candidate u is the central node. For each node w adjacent to u, if w is one of u's children, the size of the resulting tree containing w after deleting

u is $D(w)+1$. If w is u's parent, the size of the resulting tree containing w after deleting u is

$$n-1-\sum_{v\in Ch(u)}\left(D(v)+1\right),$$

where $Ch(u)$ are a set of children of u. If the size of each resulting tree is at most $N/2$, u is the desired central node. The time needed to check u is proportional to u's degree. Therefore, we can check all nodes in time $O(N)$

3 Running time

Let $\mathcal{T}(N)$ be the worst-case running time when the tree has N nodes. We can write the recurrence as

$$\mathcal{T}(N)=A(N)+cN+\sum_i \mathcal{T}(N_i),$$

where $A(N)$ is the time for finding u, N_i is the size of the i-th new trees, and c is some constant.

Since we know that $N_i \le N/2$, there are at most $O(\log N)$ levels of the recursion.

If we use $O(N)$-time to find u, each level would run in time $O(N)$ and the total running time is $O(N \log N)$. If we use a slower $O(N \log N)$-time procedure, the total running time will be $O(N \log^2 N)$.

4 Notes

There are other heuristics for finding u that *do not* always work. Here are some examples.

- In divide-and-conquer solution, the highest degree node is picked.
- In divide-and-conquer solution, the node that minimizes the maximum distance to any node is picked.