Love polygon

Tähvend Uustalu

Problem statement

As we all know, TV soap operas with many characters can lead to seriously complicated relationships.

In one TV show, there are N characters. Each character *loves* exactly one character. It is possible for a character to initially love themselves. We say that two characters are *in a relationship* if and only if they love each other. By shooting someone with a love arrow, it is possible to change whom that person loves.

Of course, this allows for a particularly nasty type of complication called a *"cycle of love"*. We say that 2 or more people are in a "cycle of love" if the first person loves the second, the second loves the third and so on, also the last person loves the first.

Recent polling has shown that viewers have grown tired of this drama and would prefer something more romantic. Therefore, it was decided to shoot some characters with love arrows so that everyone is in a relationship. At least how many love arrows are needed to accomplish that?

Input and output

The first line of the input contains one integer — N.

The next N lines all contain two space-separated names s and t, meaning that the character named s initially loves the character named t. Names of the characters are no more than 10 characters long and consist of lowercase Latin letters.

Output one integer — the minimum number of love arrows needed to get everyone into a relationship. If this is not possible, output `-1`.

Subtasks

In all subtasks $2 \leqslant N \leqslant 10^5$.

1. Each person is loved by someone.
2. $N \leqslant 20$.
3. There are no "cycles of love".
4. No additional constraints.

Solution

In essence, we have a directed graph where each vertex has exactly one outgoing arc. We are asked to redirect some of the arcs so that each vertex is in a "pair" and we are asked to do it with the minimum number of redirects. This type of graph, called a "functional graph", inevitably takes the form where each connected component is a directed cycle with directed trees branching off of it.

Note that some sort of solution is always possible if N is even, and always impossible if N is odd. Therefore, if N is odd we can immediately output `-1`. The following explanations deal with the case where N is even.

Subtask 1

In order for everyone to be loved by someone, everyone must be loved by exactly one person. In this subtask, each connected component of the graph takes the form of a cycle. Let's process each component separately, let C be the number of vertices in the component. It is easy to see that if the component has an even number of vertices, then it is optimal to pair each vertex off with one of its neighbours, using $\frac{C}{2}$ arrows, unless the component has 2 vertices, in which case no arrows are needed. On the other hand, if the component has an odd number of vertices, then it is optimal to pair each vertex but one off with one of its neighbours, pairing the last vertex off with a vertex outside of that component, using $\lfloor \frac{C}{2} \rfloor + 1$ arrows. The problem can be solved by $\mathcal{O}(N)$ time by counting the vertices in each component.

Subtask 2

Let G be the set of all people; let S be a subset of G and let $T = G \setminus S$. In order for S to be the set of people shot in any solution, the following conditions must hold:

- If we remove the edges originating from people in S, all connected components in the resulting graph must have at most 2 vertices. This is because the final graph can be constructed by only adding edges to this graph, and in the final graph all connected components have 2 vertices.
- No person who loves themselves can be in T, because then they will love themselves in the final arrangement, which is not permitted.

If these conditions hold, S can be used to construct a solution by pairing off the people in connected components of size 2 and pairing everyone else off randomly. $|S|$ arrows will be used. Therefore, a set S can be the set of people shot if and only if those conditions are met.

To solve the problem, we can iterate over all sets S and check for this property, then pick the smallest fitting set S^* and output $|S^*|$. There are 2^N subsets of G, checking each set can be done in $\mathcal{O}(N)$ time. The complexity is therefore $\mathcal{O}(N2^N)$.

An alternative approach is to use dynamic programming on subsets in $\mathcal{O}(N2^N)$ time.

Subtask 3

Since there are no "cycles of love", that must mean the cycle in each connected component of the graph consists of one character loving themselves. This means each connected component takes the form of a directed tree with all edges directed towards the root.

We call a set of vertices T in the forest *lucky* if and only if:

- For each vertex in T, its parent is not in T;
- For each vertex in T, none of its children are in T;
- For each vertex in T, none of its siblings are in T.

Let S^* be the set of vertices that are not shot with a love arrow in the optimal solution. Then S^* is clearly a lucky set: any character in that set will end up in a relationship with the character they initially loved; that is, they will be paired off with their parent in the tree. Let v be in S^*, then:

- The parent of v must be shot with an arrow to love v. Therefore the parent of v is not in S^*.
- All children of v must be shot with an arrow, otherwise they would end up in a pair with v, but we know v will be paired off with its parent. Therefore no children of v are in S^*.
- All siblings of v must be shot with an arrow, otherwise they would end up in a pair with the parent of v, but we know the parent of v will be paired off with v. Therefore no siblings of v are in S^*.

Hence, S^* is a lucky set. Note that the number of love arrows required is $N - |S^*|$. Furthermore, given any lucky set S that doesn't contain roots, we can construct a solution using $N - |S|$ arrows by pairing the vertices in S off with their parents and pairing everyone else off randomly. Therefore, if we define R as the set of lucky sets that don't contain roots, the solution to the problem is

$$\min_{S \in R} (N - |S|) = N - \max_{S \in R} |S|. \tag{1}$$

Our task is therefore to calculate the size of the largest lucky set that doesn't contain any roots. This can be done using dynamic programming.

Let L_v denote the set of initial lovers (children) of vertex v, excluding vertex v itself if v is a root. Define:

- $\mathrm{mls}(v)$ to be the size of the maximum lucky set within the subtree of v whose member v itself is not.
- $\overline{\mathrm{mls}}(v)$ to be the size of the maximum lucky set within the subtree of v whose member v itself is.

The size of the largest lucky set is then $\sum \mathrm{mls}(r)$ over all roots r. If vertex v is a leaf, then clearly $\mathrm{mls}(v) = 0$ and $\overline{\mathrm{mls}}(v) = 1$. Let v be a nonleaf vertex. Then the equation

$$\overline{\mathrm{mls}}(v) = 1 + \sum_{u \in L_v} \mathrm{mls}(u) \tag{2}$$

clearly holds. Lucky sets within the subtree of v that don't contain the vertex v can either:

- Not contain any of the children of v. The largest of them has size $\sum_{u \in L_v} \mathrm{mls}(u)$.
- Contain exactly one of the children of v. The largest lucky set containing w, a child of v, has size $\left(\sum_{u \in L_v} \mathrm{mls}(u)\right) + \overline{\mathrm{mls}}(w) - \mathrm{mls}(w)$.

All those kinds of lucky sets exist, the largest of them has therefore size

$$\begin{aligned}
&\max\left\{\sum_{u\in L_v}\mathrm{mls}(u),\quad \max_{w\in L_v}\left(\left(\sum_{u\in L_v}\mathrm{mls}(u)\right)+\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}=\\
&=\max\left\{\sum_{u\in L_v}\mathrm{mls}(u),\quad \sum_{u\in L_v}\mathrm{mls}(u)+\max_{w\in L_v}\left(\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}=\\
&=\max\left\{\overline{\mathrm{mls}}(v)-1,\quad \overline{\mathrm{mls}}(v)-1+\max_{w\in L_v}\left(\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}=\\
&=\max\left\{0,\quad \max_{w\in L_v}\left(\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}+\overline{\mathrm{mls}}(v)-1.
\end{aligned}$$

Therefore,

$$\mathrm{mls}(v)=\max\left\{0,\quad \max_{w\in L_v}\left(\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}+\overline{\mathrm{mls}}(v)-1. \tag{3}$$

To sum it up, for any vertex v:

$$\mathrm{mls}(v)=\begin{cases}0, & \text{if } v \text{ is a leaf;}\\ \max\left\{0,\quad \max_{w\in L_v}\left(\overline{\mathrm{mls}}(w)-\mathrm{mls}(w)\right)\right\}+\overline{\mathrm{mls}}(v)-1 & \text{otherwise}\end{cases} \tag{4}$$

and

$$\overline{\mathrm{mls}}(v)=\begin{cases}1, & \text{if } v \text{ is a leaf;}\\ 1+\sum_{u\in L_v}\mathrm{mls}(u) & \text{otherwise}\end{cases} \tag{5}$$

hold. Using those recurrences, we can iterate over all connected components of the graph, do a depth-first search on them and calculate the values of $\overline{\mathrm{mls}}(v)$ and $\mathrm{mls}(v)$ for all vertices v. Finally, we can output $N-(\sum\mathrm{mls}(r))$ over all roots r. Each vertex needs to be traversed only once, which gives a runtime of $\mathcal{O}(N)$.

Subtask 4

The subtask is solved similarly to subtask 3. We will process each connected component separately. If the cycle of the current component consists of just one character, we will process it the same way as in subtask 3. If the cycle is longer, we pick one arbitrary character. In the optimal solution, that character either is in a relationship with the person they love, or isn't. In both cases we can eliminate some arcs from the component so that the component becomes a forest of directed trees. Using the solution from subtask 4, we can calculate the number of love arrows needed in both situations and pick the better one. This solves the subtask in $\mathcal{O}(N)$.

Additional subtask idea

Subtask 5

- People on a "cycle of love" are loved by at most 2 people; everyone else is loved by at most 1 person.

The statement of this subtask is quite clumsy and this is the main reason it is currently left out. It would be really cool to come up with some condition that can be concisely stated in terms of the problem that "secretly" implies this subtask.

In each connected component the graph will take the form of a cycle with path graphs branching off of the cycle. This can be solved with simple dynamic programming "around the cycle".