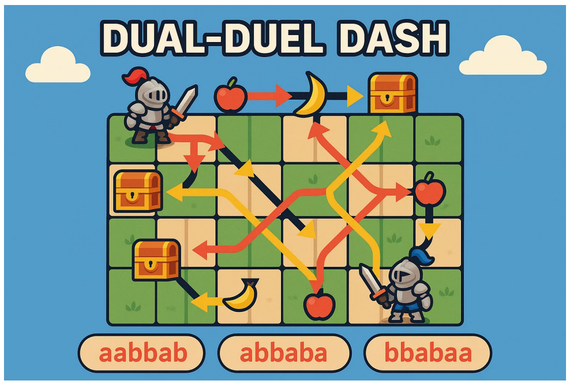
Dual-Duel Dash Master

Peter Tessier, ...

Motivation



You're a competitive player of the newest hit mobile game "Dual-Duel Dash", where you control two knights on a treacherous board. The game works by generating a random board of tiles—some of which have treasure, and others that don't. There are two arrows (an apple arrow and a banana arrow) going from each tile to some other tile. On your turn, you are presented with three different strings of apples and bananas (for instance, 'aabbab', 'abbaba', and 'bbabaa', where 'a' is apple and 'b' is banana). You get to choose one of these strings, at which point the knights begin their course: for each fruit in the sequence, they follow the path which corresponds to it. At the end of the sequence, if both knights land on treasure, they win a grand prize; if one does, it's a modest victory; if neither, they go home empty-handed.

Being a computer science undergraduate, you understand that underneath the fun, the knights' behavior closely resembles a DFA and can hence be predicted. So far, you've been able to scrape the webpage to convert the on-screen paths into a modified DFA with two start states, which you call a DoubleStartDFA, that helps you figure out which path to choose. However, the game recently underwent a big update, making the paths and fruit sequences much larger and more complicated. In order to uphold your reputation as Grand DashMaster, you must "level up" by minimizing the DFA to query strings on it faster.

Problem Statement

A *DoubleStartDFA* $M = (Q, \Sigma, \delta, s_1, s_2, F)$ works like a standard DFA except it has *two* start states. Here $Q = \{0, 1, \ldots, n-1\}$, the input alphabet $\Sigma = \{a, b\}$, and $\delta : Q \times \Sigma \to Q$ is complete (it has a value for every state and input symbol). The start states are $s_1, s_2 \in Q$, and $F \subseteq Q$ is the set of *final* (accepting) states. Upon reading a word $w \in \Sigma^*$, each start state s_i follows transitions to some state q_i. The overall outcome, notated as outcome(M, w), is:

- *Accepting* if both $q_1, q_2 \in F$,
- *Half-Accepting* if exactly one of q_1, q_2 lies in F,
- *Rejecting* if neither lies in F.

Your task is to construct an equivalent minimized *MultiFinalDFA* $M' = (Q', \Sigma, \delta', s', P)$ with a *single* start state s' and a partition $P = \{R, H, A\}$ of its states into *Rejecting*, *Half-Accepting*, and *Accepting* classes. Upon reading a word $w \in \Sigma^*$, its start state s' follows transitions to some state q. The partition that q lies in (*Rejecting*, *Half-Accepting*, or *Accepting*) determines its outcome, notated outcome(M', w), which should always equal outcome(M, w).

Input Format

All inputs use the fixed alphabet $\{a, b\}$. All states are represented as integers. You are given the following lines:

```
n            % number of states (0..n-1) in DoubleStartDFA
p c q        % transitions: from state p on symbol c to state q
...          % (one line per transition)
s1 s2        % the two start states (integers)
f            % number of final states
q            % one final-state per line
```

Transitions cover every $(state, symbol)$ pair exactly once, so there are a total of $2n$ transition lines.

Output Format

Produce the minimized MultiFinalDFA with three final partitions:

```
n'           % number of states in minimized MultiFinalDFA
p c q        % transitions (one per line, as before)
...
start        % the single start state (int)
r            % number of rejecting states
...          % rejecting-states (one per line)
h            % number of half-accepting states
...          % half-accepting-states (one per line)
a            % number of accepting states
...          % accepting-states (one per line)
```

Note

Given any input, the output is not unique, since nodes of the DFA can be labeled differently while still reading the same language. Hence, we don't check that your output matches ours, but rather run a script to determine that the MultiFinalDFA you produced is equivalent (up to relabelling) to ours.

Example

Input

```
8
0 a 1
0 b 2
1 a 3
1 b 4
2 a 5
2 b 6
3 a 1
3 b 4
4 a 4
4 b 5
5 a 4
5 b 5
6 a 7
6 b 7
7 a 6
7 b 6
0 2
2
4
5
```

Output

```
6
0 a 0
0 b 0
1 a 1
1 b 5
2 a 0
2 b 4
3 a 1
3 b 2
4 a 4
4 b 4
5 a 5
5 b 5
3
3
2
3
4
2
0
1
1
5
```