Bartosz Tarnawski **Karol Farbiś**
Treść zadania, Opracowanie Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień drugi, 13.02.2014*

Rajd

W Bajtogrodzie niedługo odbędzie się doroczny rajd rowerzystów. Bajtogrodzcy rowerzyści są urodzonymi długodystansowcami. Przedstawiciele lokalnej społeczności motorowerzystów, od dawna zwaśnieni z rowerzystami, postanowili sabotować to wydarzenie.

W Bajtogrodzie znajduje się n skrzyżowań, połączonych jednokierunkowymi ulicami. Co ciekawe, w sieci ulic nie występują cykle – jeżeli ze skrzyżowania u można dojechać do v, to na pewno z v nie da się w żaden sposób dostać do u.

Trasa rajdu będzie prowadziła przez bajtogrodzkie ulice. Motorowerzyści postanowili w dniu wyścigu z samego ranka przyjechać na swoich lśniących maszynach na jedno ze skrzyżowań i zupełnie je zablokować. Co prawda wówczas związek kolarski szybko wytyczy alternatywną trasę, ale być może nie będzie ona taka długa i rowerzyści nie będą mogli wykazać się swoimi możliwościami. Na to właśnie liczą motorowerzyści – chcą zablokować takie skrzyżowanie, żeby najdłuższa trasa, która je omija, była możliwie krótka.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($2 \leqslant n \leqslant 500\,000$, $1 \leqslant m \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i ulic w Bajtogrodzie. Skrzyżowania numerujemy liczbami od 1 do n. Kolejne m wierszy zawiera opis sieci drogowej: w i-tym z tych wierszy znajdują się dwie liczby całkowite a_i, b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje jednokierunkowa ulica od skrzyżowania o numerze a_i do skrzyżowania o numerze b_i.

W testach wartych łącznie 33% punktów dla każdej ulicy zachodzi dodatkowy warunek: $a_i < b_i$.

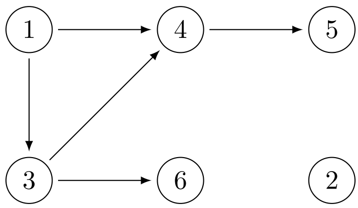
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać dwie liczby całkowite oddzielone pojedynczym odstępem. Pierwsza z tych liczb ma oznaczać numer skrzyżowania, które powinni zablokować motorowerzyści, druga zaś – maksymalną liczbę ulic, którymi mogą przejechać wówczas rowerzyści. W przypadku, gdy istnieje wiele poprawnych rozwiązań, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:
```
6 5
1 3
1 4
3 6
3 4
4 5
```

poprawnym wynikiem jest:
```
1 2
```



Testy „ocen”:

`1ocen`: $n = 10$, $m = 9$, *ścieżka, najlepiej zablokować ją w środku;*

`2ocen`: $n = 100$, $m = 4950$, *istnieją wszystkie drogi ze skrzyżowań o mniejszych numerach do skrzyżowań o większych numerach;*

`3ocen`: $n = 500\,000$, $m = 749\,999$, *ze skrzyżowania* i *wychodzi droga do* $i-1$ *(o ile* $i \geqslant 2$*) oraz* $\frac{i}{2}$ *(o ile* $2 \mid i$*).*

Rozwiązanie

Dany jest graf skierowany bez cykli (ang. *directed acyclic graph*, w skrócie *DAG*). Należy znaleźć wierzchołek, którego usunięcie minimalizuje długość najdłuższej ścieżki.

Nasz graf będziemy oznaczać przez G, jego zbiór wierzchołków przez $V(G)$, zaś zbiór krawędzi – przez $E(G)$. Mamy $|V(G)| = n$, $|E(G)| = m$. Będziemy utożsamiać wierzchołki z ich etykietami, czyli napiszemy $V(G) = \{1, 2, \ldots, n\}$. Przez (a, b) oznaczymy skierowaną krawędź od a do b.

Dla uproszczenia możemy założyć, że graf nie ma izolowanych wierzchołków; wówczas $n = O(m)$.

W podzadaniu wartym 33% punktów można założyć, że zachodzi następująca własność:

$$\text{dla każdej krawędzi } (a, b) \in E(G) \;\; a < b. \tag{*}$$

Po odpowiednim przenumerowaniu wierzchołków taka własność może zachodzić dla dowolnego DAG-u. Algorytm, który wykonuje takie przenumerowanie, nazywamy sortowaniem topologicznym. Zajmuje on $O(n + m)$ czasu (można go zrealizować za pomocą przeszukiwania w głąb). Jest on opisany w opracowaniu zadania *Licytacja* z XIX Olimpiady [19] oraz w książce [25].

Ponieważ przyda się nam to we wszystkich algorytmach, ustalmy, że zaraz po wczytaniu grafu sortujemy wierzchołki topologicznie. Od tej chwili możemy zakładać, że własność (*) zachodzi dla naszego grafu G.

Zanim przejdziemy do właściwego rozwiązania, zastanówmy się (lub przypomnijmy sobie), jak szybko wyznaczyć najdłuższą ścieżkę w naszym grafie G. Osiągniemy to, obliczając tablicę $longestStart[1..n]$. Wartość na i-tej pozycji ma być równa długości (liczbie krawędzi) najdłuższej ścieżki zaczynającej się w wierzchołku i. Na początku inicjujemy ją zerami, a potem wykonujemy pętlę:

```
for i := n downto 1 do
  for (i, j) ∈ E do
    longestStart[i] := max(longestStart[i], longestStart[j] + 1);
```

Teraz największa wartość w tablicy jest długością najdłuższej ścieżki. To prowadzi nas już do jednego z wolnych rozwiązań.

Rozwiązanie wolne $O(nm)$

Usuwamy po kolei każdy z wierzchołków, a następnie obliczamy długość najdłuższej ścieżki.

To rozwiązanie zostało zaimplementowane w plikach `rajs3.cpp`, `rajs4.pas`. Zdobywało około 30 punktów.

Ścieżki omijające wierzchołek

Powiemy, że ścieżka p *omija* wierzchołek v, gdy v nie należy do p. Żeby zdobyć więcej punktów, trzeba przyjrzeć się temu, jak może wyglądać najdłuższa ścieżka omijająca pewien wierzchołek.

Lemat 1. Każda ścieżka p omijająca wierzchołek v:

1. kończy się w wierzchołku w, który jest wcześniej niż v w porządku topologicznym ($w < v$), albo
2. rozpoczyna się w wierzchołku w, który jest później niż v w porządku topologicznym ($v < w$), albo
3. istnieje krawędź (u, w) na ścieżce p, taka że u jest wcześniej niż v, a w jest później niż v w porządku topologicznym ($u < v < w$).

Dowód: Z własności (*) wiemy, że jeśli p składa się z kolejnych krawędzi $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$, to $v_1 < v_2 < \ldots < v_k$. Stąd już wynika nasz lemat. ■

Wystarczy, że dla każdego wierzchołka rozpatrzymy te trzy przypadki.

Obliczmy teraz, tak jak to zostało opisane wyżej, tablicę $longestStart[1..n]$, a także tablicę jej maksimów sufiksowych $longestAfter[1..n]$, to znaczy

$$longestAfter[i] = \max_{j \geqslant i} longestStart[j].$$

W analogiczny sposób obliczamy jeszcze tablicę $longestEnd[1..n]$ (długość najdłuższej ścieżki kończącej się w danym wierzchołku) oraz tablicę jej maksimów prefiksowych $longestBefore[1..n]$.

Chcielibyśmy jeszcze obliczyć tablicę $longestBypass[1..n]$ taką, aby zachodziło

$$longestBypass[i] = \max_{a<i<b,\ (a,b)\in E(G)} \left(longestEnd[a] + 1 + longestStart[b]\right). \qquad (**)$$

Jeśli to nam się uda, to poznamy długość najdłuższej ścieżki omijającej dowolny wierzchołek v. Na mocy lematu 1 będzie to

$$\max(longestBefore[v-1], longestAfter[v+1], longestBypass[v]).$$

Wtedy możemy już rozwiązać całe zadanie, znajdując minimum z tej wartości dla wszystkich wierzchołków.

Żeby efektywnie wyliczyć tablicę $longestBypass[1..n]$, możemy użyć drzewa przedziałowego. Doprowadzi nas to do pierwszego rozwiązania wzorcowego.

Rozwiązanie wzorcowe $O(m \log n)$

Rozwiązanie to nie jest najszybszym znanym, nie prowadzi też do najkrótszego kodu, ale za to jest intuicyjne. Przez to cieszyło się dużą popularnością wśród zawodników.

Skorzystamy z drzewa przedziałowego, którego liście będą odpowiadały kolejnym wierzchołkom $1, 2, \ldots, n$.

Nasze drzewo każdemu liściowi v będzie przypisywało pewną wartość – $value(v)$. Początkowo $value(v) = 0$ dla każdego v. Chcemy, aby po wykonaniu naszego algorytmu $value(v)$ równało się docelowej wartości $longestBypass[v]$ zdefiniowanej jak w (**).

Drzewo powinno udostępniać operacje ustawienia maksimum na przedziale oraz odczytania wartości w punkcie, czyli

1. $set(l, r, max_v)$ – dla każdego i takiego, że $l \leqslant i \leqslant r$, ustaw $value(i) = \max(value(i), max_v)$,
2. $query(i)$ – odczytaj wartość $value(i)$.

Zawodnicy często spotykają drzewa przedziałowe tego rodzaju. O takich i podobnych drzewach przedziałowych można dowiedzieć się dużo ciekawych rzeczy, oglądając *Wykłady z Algorytmiki Stosowanej* (`http://was.zaa.mimuw.edu.pl`).

Niech $(a, b) \in E(G)$ będzie dowolną krawędzią. Oznaczmy długość najdłuższej ścieżki zawierającej tę krawędź przez $longestWith(a, b)$. Wartość tę możemy obliczyć w czasie stałym, ponieważ

$$longestWith(a, b) = longestEnd[a] + 1 + longestStart[b].$$

Teraz dla każdej krawędzi $(a, b) \in E(G)$ i $l := longestWith(a, b)$ wykonamy operację $set(a+1, b-1, l)$. Możemy o tym myśleć jak o „informowaniu” wierzchołków między a i b, że istnieje omijająca je ścieżka o długości l.

Następnie odpytujemy wszystkie wierzchołki za pomocą operacji *query*(), uzyskując tablicę *longestBypass*[1..n]. Ponieważ każda operacja na drzewie działa w czasie $O(\log n)$, całość zajmie nam $O(m \log n)$ czasu.

Takie rozwiązanie zostało zaimplementowane w plikach `raj.cpp`, `raj1.pas`. Zdobywało 100 punktów.

Autor zadania nie miał żadnego pomysłu na szybsze rozwiązanie. Taki pomysł miał za to jeden z zawodników.

Rozwiązanie $O(m \log^* n)$

To rozwiązanie, choć asymptotycznie szybsze od wzorcowego, w praktyce działa podobnie szybko. Opiera się ono na lemacie 1, ale korzysta z niego w inny sposób. Dla wygody wprowadźmy kolejny lemat, który jest prostym wnioskiem z poprzedniego.

Lemat 2. Ustalmy dowolny wierzchołek $v \in V(G)$. Ścieżka o długości l omijająca v istnieje wtedy i tylko wtedy, gdy spełniony jest choć jeden z następujących warunków.

1. Dla pewnego $w < v$ zachodzi $longestEnd[w] \geqslant l$.
2. Dla pewnego $w > v$ zachodzi $longestStart[w] \geqslant l$.
3. Dla pewnej krawędzi (u, w), $u < v < w$, zachodzi $longestWith(u, w) \geqslant l$.

Stworzymy jeszcze jedną tablicę, tym razem indeksowaną od 0 do n. Jej elementami będą listy par liczb całkowitych. Nazwiemy ją *valIntervals*[0..n]. Konstruujemy ją w taki sposób, aby para (a, b) $(1 \leqslant a \leqslant b \leqslant n)$ należała do listy $valIntervals[l]$, jeżeli

1. $b = n$ oraz $longestEnd[a-1] = l$, lub
2. $a = 1$ oraz $longestStart[b+1] = l$, lub
3. $(a-1, b+1) \in E(G)$ oraz $longestWith(a-1, b+1) = l$.

Zauważmy, że mając już *longestEnd*[1..n] oraz *longestStart*[1..n], prosto obliczamy *valIntervals*[0..n] w czasie $O(n+m)$.

Dzięki temu, jak sformułowaliśmy lemat 2, otrzymujemy od razu kolejną równoważność.

Lemat 3. Ustalmy dowolny wierzchołek $v \in V(G)$. Ścieżka o długości l omijająca v istnieje wtedy i tylko wtedy, gdy istnieją takie liczby całkowite a, b, $1 \leqslant a \leqslant v \leqslant b \leqslant n$, że para (a, b) należy do pewnej listy $valIntervals[l']$, przy czym $l' \geqslant l$.

Zdefiniujmy jeszcze zbiory A_l, B_l dla $l = 0, 1, \ldots, n$. Będą to podzbiory zbioru $V(G) = \{1, \ldots, n\}$. Niech

$$A_l = \{v : a \leqslant v \leqslant b \text{ dla którejś pary } (a, b) \in valIntervals[l]\}$$

$$B_l = A_l \cup A_{l+1} \cup \ldots \cup A_n.$$

Wtedy na mocy lematu 3 B_l jest zbiorem tych wierzchołków, które omija choć jedna ścieżka długości l.

Z definicji wiemy, że zbiór B_{l+1} zawiera się w zbiorze B_l dla każdego $l = 0, \ldots, n-1$.

Potrzebujemy jeszcze ostatniej obserwacji.

Lemat 4. Istnieje takie $0 \leqslant l_0 \leqslant n-1$, że $B_{l_0} = V(G)$ oraz $B_{l_0+1} \neq V(G)$.

Dowód: Łatwo zobaczyć, że $B_n = \emptyset$ (w grafie nie ma ścieżki długości n), zaś $B_0 = \{1, \ldots, n\} = V(G)$. ■

Jesteśmy już w stanie przedstawić nasz algorytm w ogólnym zarysie. Będziemy obliczać zbiory B_l kolejno dla $l = n, n-1, n-2, \ldots$ Zatrzymamy się w momencie, gdy $l = l_0$ jak w lemacie 4.

Wówczas optymalnym rozwiązaniem będzie usunięcie dowolnego wierzchołka z $V(G) \setminus B_{l_0+1}$, a najdłuższa ścieżka w powstałym grafie będzie miała długość l_0.

Obliczanie zbiorów B_l

Pozostaje jeszcze pytanie – w jaki sposób można efektywnie obliczać kolejne zbiory B_l (dla malejących l) oraz sprawdzać, czy są równe $V(G)$.

Aby wprowadzić powiew świeżości, możemy zapomnieć o tym, że nasze zbiory mają coś wspólnego z wierzchołkami, i sformułować ten podproblem na nowo.

> Mamy n szklanek ustawionych w rzędzie od lewej do prawej i ponumerowanych liczbami od 1 do n. Początkowo wszystkie szklanki są puste. Bajtek m razy wybiera dwie szklanki a, b ($1 \leqslant a \leqslant b \leqslant n$), a następnie napełnia wodą szklanki $a, a+1, \ldots, b$. Raz napełniona szklanka pozostaje pełna, ponowne napełnianie już nic nie zmienia. Pytamy się, w którym momencie po raz pierwszy wszystkie szklanki są pełne.

Siłowe rozwiązanie zajmuje $O(nm)$ czasu, ponieważ przy każdym napełnianiu być może trzeba sprawdzić $O(m)$ szklanek. My jednak nie chcemy tracić czasu na przeglądanie już napełnionych szklanek.

Niech $\mathit{full}[1..n]$ będzie tablicą reprezentującą to, co sugeruje nazwa – $\mathit{full}[i] = \mathbf{true}$, jeżeli w danym momencie i-ta szklanka jest pełna. Dobrze by było, gdybyśmy w każdym momencie oraz dla każdego i mogli szybko obliczyć poniższą funkcję

$$\mathit{firstEmpty}(i) = \min\{j \geqslant i : \mathit{full}[j] = \mathbf{false}\}.$$

Załóżmy na chwilę, że wywołanie tej funkcji zajmuje nam $f(n)$ czasu. Wówczas będziemy w stanie rozwiązać nasz problem w czasie $O((m+n)f(n))$. Kiedy napełniając kubki pomiędzy a i b, natrafimy na pełny kubek, to będziemy mogli przeskoczyć do najbliższego pustego na prawo w czasie $f(n)$. Każdemu takiemu przeskoczeniu (być może poza ostatnim) możemy przyporządkować napełnienie pustego kubka. Pusty kubek będziemy napełniać co najwyżej n razy, do tego jeszcze trzeba doliczyć co najwyżej m „ostatnich przeskoczeń”, po których nic nie napełniliśmy.

Pozostaje więc kwestia obliczania funkcji *firstEmpty*(). Z pomocą przychodzi nam struktura reprezentująca zbiory rozłączne (tzw. *Find-Union*; patrz [25], [35]). Pełne kubki poprzydzielamy do *bloków*, tak aby zachodził niezmiennik:

Kubki a oraz b ($a \leqslant b$) należą do tego samego bloku wtedy i tylko wtedy, gdy wszystkie kubki $a, a+1, \ldots, b$ są pełne.

Przynależność do bloków reprezentujemy właśnie za pomocą struktury Find-Union. Dodatkowo dla każdego bloku pamiętamy numer jego najbardziej prawego kubka. Dzięki temu możemy obliczyć *firstEmpty*() za pomocą pojedynczej operacji *find*(), czyli w zamortyzowanym czasie $O(\log^* n)$. Szczegóły związane z uaktualnianiem bloków podczas napełniania kubków pozostawiamy Czytelnikowi do samodzielnego przemyślenia.

Powyższe rozwiązanie zostało zaimplementowane w pliku `raj8.cpp`. Dostawało oczywiście 100 punktów.

Testy

Przygotowano 24 testy rozdzielone pomiędzy 9 grup. Większość małych testów to testy poprawnościowe. Testy duże to w większości testy losowe. Wykorzystano następujące rodzaje grafów:

- ścieżka,
- turniej (czyli DAG-klika),
- graf z losowymi krawędziami wygenerowanymi w taki sposób, aby wierzchołki o niskich numerach miały dużo krawędzi wychodzących, a te o wysokich numerach – dużo krawędzi wchodzących,
- graf z wyróżnionymi dwoma wierzchołkami s i t, zawierający dla każdego innego wierzchołka v krawędzie (s, v) i (v, t),
- ścieżka z usuniętymi niektórymi krawędziami i dodanymi krawędziami w losowych miejscach,
- graf z jednoznacznym rozwiązaniem (wyrzucenie jednego z wierzchołków znacznie zmniejsza długość najdłuższej ścieżki, a każdego z pozostałych – tylko o 1),
- oraz kilka małych grafów wygenerowanych na kartce.

Wykorzystano także grafy powstałe z powyższych na drodze następujących operacji:

- rozłączna suma dwóch grafów A i B (stawiamy grafy A, B obok siebie, nic więcej nie robimy),
- suma krawędzi grafów A i B (A i B muszą mieć tyle samo wierzchołków, bierzemy najpierw graf A, a potem jeszcze dodajemy do niego wszystkie krawędzie z grafu B),
- „połączenie” grafów A i B (stawiamy grafy A, B obok siebie, następnie dla każdej pary $a \in V(A), b \in V(B)$ dodajemy krawędź (a, b)).

Zawody III stopnia

opracowania zadań