Informatyczny kącik olimpijski (68): Drogi stanowe

Tym razem omówimy zadanie o drogach stanowych (*State Roads*) z pierwszej rundy zawodów Yandex.Algorithm 2013. Zadanie to można sformułować w języku teorii grafów, a do tego w ujęciu dynamicznym. Mamy więc graf nieskierowany o zbiorze wierzchołków V, początkowo pusty, w którym w kolejnych momentach pojawiają się i znikają krawędzie. Pomiędzy takimi zdarzeniami otrzymujemy zapytania, z których każde dotyczy pewnego podzbioru wierzchołków grafu $V_i \subseteq V$. W odpowiedzi na takie zapytanie mamy stwierdzić, czy z jakiegokolwiek wierzchołka zbioru V_i wychodzi jakaś krawędź poza ten zbiór; innymi słowy, czy wierzchołki zbioru V_i w rozważanym momencie stanowią sumę pewnej liczby spójnych składowych grafu. W zadaniu występuje też techniczny (ale przyjemny) warunek, że wstawiane krawędzie otrzymują numery będące kolejnymi liczbami naturalnymi i za pomocą tych numerów oznaczane są krawędzie przeznaczone do usunięcia.

Zastanówmy się najpierw nad rozwiązaniem siłowym. Tym razem, celowo, omówimy dość dokładnie nietrywialne szczegóły implementacyjne rozwiązania. Cały graf możemy przechowywać po prostu jako tablicę list sąsiedztwa. Przy wstawianiu krawędzi uv o numerze j dodajemy po jednym nowym elemencie do list sąsiedztwa wierzchołków u i v i w osobnej tablicy indeksowanej numerem krawędzi zapamiętujemy wskaźniki na elementy list, które dodaliśmy dla tej krawędzi. Takie wstawienie działa oczywiście w czasie stałym:

function *wstaw*(u, v, j)
  push_back(*lista*[u], v)
  push_back(*lista*[v], u)
  krawedz_u[j] := *last*(*lista*[u])
  krawedz_v[j] := *last*(*lista*[v])

Dzięki wspomnianemu przyjemnemu warunkowi technicznemu uprzednio wstawioną krawędź możemy usunąć z grafu również w czasie stałym, po prostu usuwając odpowiednie elementy list sąsiedztwa:

function *usun*(u, v, j)
  delete(*lista*[u], *krawedz_u*[j])
  delete(*lista*[v], *krawedz_v*[j])

Bardziej czasochłonną operacją jest, oczywiście, odpowiedź na zapytanie. Wczytując wierzchołki występujące w zapytaniu, możemy równocześnie zaznaczać wszystkie te wierzchołki w pomocniczej tablicy indeksowanej numerami wierzchołków. Żeby nie musieć za każdym razem zerować tej tablicy, wierzchołki z każdego nowego zapytania możemy oznaczać numerem porządkowym tego zapytania. Teraz obsługa zapytania jest już bardzo prosta – przeglądamy kolejne wierzchołki z zapytania, dla każdego z nich przeglądamy wszystkie incydentne krawędzie i sprawdzamy, czy drugie końce tych krawędzi są zaznaczone w tablicy pomocniczej:

function *zapytanie*(V_i)
  foreach $v \in V_i$ **do** *pom*[v] := i
  foreach $v \in V_i$ **do**
    foreach $u \in$ *lista*[v] **do**
      if *pom*[u] $\neq i$ **then return** *false*
  return *true*

Na pierwszy rzut oka widać, że obsługa zapytania jest tutaj *bardzo wolna*. Warto jednak przyjrzeć się temu dokładniej. Przede wszystkim należy ustalić, co jest dla nas rozmiarem danych wejściowych. Oznaczmy przez n liczbę wierzchołków grafu, a przez m – łączną liczbę krawędzi wstawionych kiedykolwiek do grafu. Niech n_i (dla $i = 1, 2, \ldots, z$) oznacza liczbę wierzchołków występujących w i-tym zapytaniu, $n_i = |V_i|$. Całkiem naturalnie jako rozmiar danych wejściowych przyjmiemy $R = n + m + \sum_{i=1}^{z} n_i$.

Jak teraz wyrazić względem R koszt obsługi wszystkich zapytań? Zauważmy, że w i-tym zapytaniu będziemy musieli przejrzeć co najwyżej $n_i(n_i - 1)$ krawędzi. Faktycznie, tyle krawędzi (liczonych w obie strony) ma klika o n_i wierzchołkach, więc jeśli w listach sąsiedztwa wierzchołków z zapytania znajdzie się więcej niż tyle krawędzi, to na pewno odpowiedź na to zapytanie będzie negatywna. Asymptotycznie daje to koszt $O(n_i^2)$. Z drugiej strony, w każdym zapytaniu przejrzymy łącznie co najwyżej $2m$ krawędzi. Ostatecznie koszt zapytania to $O(\min(n_i^2, m))$. Skorzystajmy teraz z nierówności między minimum a średnią geometryczną:

$$\min(n_i^2, m) \leqslant \sqrt{n_i^2 m} = n_i\sqrt{m}.$$

Całkowity koszt obsługi wszystkich zapytań możemy oszacować przez:

$$O\Big(\sum_{i=1}^{z} n_i\sqrt{m}\Big) = O(R\sqrt{m}) = O(R\sqrt{R}).$$

Czyli jednak to rozwiązanie nie było znowuż aż takie wolne!

Jednak da się lepiej. I to dużo lepiej, bowiem liniowo względem rozmiaru danych wejściowych. Rozwiązanie opiera się na następującym pomyśle: obliczmy sumę stopni wierzchołków ze zbioru V_i. Jeśli suma ta jest nieparzysta, to na pewno z V_i wychodzi jakaś krawędź na zewnątrz, bowiem w każdym grafie suma stopni wierzchołków jest parzysta. Pomysł ten w ogólności nie działa – może się okazać, że suma stopni wierzchołków w V_i będzie parzysta, a mimo tego odpowiedź na zapytanie będzie negatywna.

Uogólnijmy zatem nasz pomysł i wprowadźmy element randomizacji. Każdej krawędzi wstawianej do grafu przypiszmy losową wagę będącą liczbą całkowitą. Dla każdego wierzchołka $v \in V$ będziemy przechowywać (jako *xor*[v]) bitową alternatywę wykluczającą wag krawędzi incydentnych z tym wierzchołkiem. Przy wstawianiu i usuwaniu krawędzi tablicę *xor* możemy aktualizować w czasie stałym – w obu przypadkach wystarczy do-xor-ować wagę krawędzi do *xor*-ów jej końców. W przypadku wstawiania będzie to:

xor[u] := *xor*[u] $\oplus$ *waga*[j]
xor[v] := *xor*[v] $\oplus$ *waga*[j]

Kluczowa obserwacja jest teraz taka: odpowiedź na zapytanie $V_i = \{u_1, u_2, \ldots, u_k\}$ jest pozytywna wtedy i tylko wtedy, gdy każda krawędź w grafie jest incydentna z dwoma wierzchołkami ze zbioru V_i lub z żadnym z nich. Ponieważ $w \oplus w = 0$, zatem aby odpowiedzieć na zapytanie, obliczamy wartość:

$$xor[u_1] \oplus xor[u_2] \oplus \ldots \oplus xor[u_k].$$

Jeśli powyższa wartość jest różna od zera, na pewno odpowiedź na zapytanie jest negatywna. W przeciwnym razie *z dużym prawdopodobieństwem* odpowiedź jest pozytywna. Prawdopodobieństwo udzielenia błędnej odpowiedzi, czyli trafienia przypadkiem na 0, wynosi $\frac{1}{M}$, gdzie M jest ograniczeniem górnym na wagę krawędzi. Przyjmując, naturalnie, $M = 2^{32}$ lub $M = 2^{64}$, otrzymujemy rzeczywiście bardzo małe prawdopodobieństwo błędu.

Jakub RADOSZEWSKI