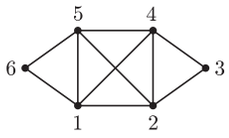
Przez 32

Jakub RADOSZEWSKI

We współczesnej informatyce szeroko rozumiana kwestia efektywności odgrywa niebagatelną rolę. Jest to związane z koniecznością operowania na coraz większych ilościach danych przy jednoczesnym uwzględnieniu wzrastających wymagań użytkowników względem szybkości i jakości usług – bodaj najlepszą, ale na pewno nie jedyną, ilustracją tego zjawiska jest Internet. Dlatego też informatycy na wszelkie sposoby starają się przyspieszać tworzone aplikacje. Stosowane są do tego zasadniczo dwa podejścia: „bardziej teoretyczne", polegające na zmniejszaniu złożoności czasowych i pamięciowych wykorzystywanych algorytmów, a w ostateczności stałego czynnika w złożoności (patrz np. artykuł *Średnio lepiej* w *Delcie* 12/2009), oraz „bardziej praktyczne", w którym projektuje się coraz to wydajniejsze procesory, łączy istniejące procesory w większe zespoły, powiększa pamięci podręczne i dyskowe, zwiększa przepustowości łączy sieciowych etc. (o skali tego zjawiska można przeczytać w artykule o mikroprocesorach w *Delcie* 5/2009). W tym artykule zaprezentujemy metodę optymalizacji programów, która w tej klasyfikacji znajduje się gdzieś pośrodku.



Rys. 1

Nasze rozważania będą opierać się na pewnym dosyć prostym problemie grafowym. Dany jest graf nieskierowany G, mający n wierzchołków oraz m krawędzi. Naszym zadaniem jest zliczenie w tym grafie wszystkich *trójkątów*, czyli takich trójek wierzchołków u, v, w, że istnieją wszystkie krawędzie: z u do v, z v do w i z u do w. Na przykład, graf z rysunku 1 zawiera sześć trójkątów.

Najprostszy algorytm rozwiązujący opisany problem polega na rozważeniu wszystkich trójek wierzchołków; łatwo zauważyć, że można go zaimplementować w złożoności czasowej $O(n^3)$. Powstaje pytanie, czy da się to jakoś poprawić. Intuicja podpowiada, że powinno się dać – źródłem takiej intuicji może być pierwsze zadanko (nie)informatyczne z *Delty* 8/2009, w którym pokazano, jak zliczać w grafie naraz wszystkie trójkąty i antytrójkąty (czyli trójki wierzchołków, z których żadne dwa nie są połączone krawędzią) w czasie $O(n^2)$. Niestety, po chwili namysłu można dojść do wniosku, że podana tam sztuczka nie zadziała w przypadku zliczania samych trójkątów. Innym z podejść jest uzależnienie złożoności czasowej konstruowanego algorytmu od m, a nie – lub w mniejszym stopniu – od n. I tak, znane jest rozwiązanie problemu trójkątów o złożoności czasowej $O(m^{\frac{3}{2}})$ (przy naturalnym założeniu, że $n = O(m)$), które w przypadku grafów rzadkich ($m = O(n)$) jest istotnie lepsze od powyższego, lecz dla grafów gęstych ($m = \Theta(n^2)$) jego przewaga znika. Nas jednak takie manipulacje nie będą tym razem interesować – powiedzmy, że poszukujemy algorytmu, którego złożoność czasową możemy oszacować tylko za pomocą n. W dalszej części tekstu skonstruujemy algorytm, który w pewnym sensie spełnia postawione wymagania.

Rozwiązanie zadania M 1264.
Wykażemy, że w ciągu $x_1, x_2, \ldots$ występuje liczba parzysta.

Przypuśćmy, że wszystkie liczby $x_1, x_2, \ldots$ są nieparzyste. Różnice między nimi są parzyste, a więc każda z tych liczb ma w zapisie dziesiętnym pewną niezerową cyfrę parzystą.

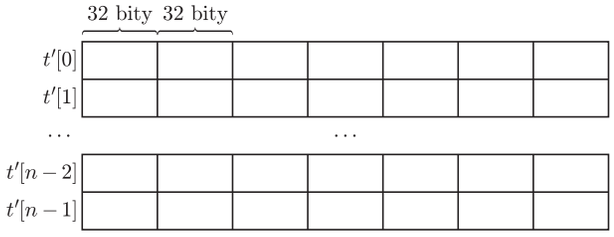
Niech $x_1 = \overline{a_1 a_2 \ldots a_m}$ oraz niech j będzie najmniejszą liczbą, dla której cyfra a_j jest parzysta i różna od 0. Ponieważ każdy następny wyraz ciągu powstaje poprzez dodanie do poprzedniego liczby większej od 0 i mniejszej niż 10, więc dla pewnego k otrzymamy liczbę $x_k = \overline{b_1 b_2 \ldots b_m}$, gdzie $b_i = a_i$ dla $i < j$, $b_j = a_j + 1$, $b_i = 0$ dla $j < i < m$; liczba b_m jest, oczywiście, nieparzysta. To oznacza, że liczba x_k ma w zapisie dziesiętnym jedynie cyfry nieparzyste i zera. Wobec tego liczba x_{k+1} musi być parzysta.

Nasz graf będziemy reprezentować za pomocą macierzy sąsiedztwa, czyli dwuwymiarowej tablicy t, w której $t[i][j]$ jest jedynką, jeśli w G istnieje krawędź łącząca wierzchołki i oraz j, a zerem w przeciwnym przypadku. Zakładając, że wierzchołki G są ponumerowane od 0 do $n-1$, opisany na wstępie podstawowy algorytm zliczania trójkątów można zapisać tak:

```
w := 0;
for i := 0 to n − 1 do
  for j := 0 to n − 1 do
    if (t[i][j]) then
      for k := 0 to n − 1 do
        if (t[i][k] and t[j][k]) then
          w := w + 1;
return w;
```

W powyższym pseudokodzie zastosowaliśmy konwencję traktującą wartość 1 w tablicy jako prawdę, a 0 jako fałsz (jest to popularne w wielu językach programowania). Co prawda, konwencja ta sama w sobie ani trochę nie przyspiesza rozwiązania, jednakże naprowadza nas na istotne spostrzeżenie, że cała tablica t

wypełniona jest tak naprawdę wartościami logicznymi. Jest ono istotne dlatego, że takie jednobitowe wartości w informatyce samodzielnie występują stosunkowo rzadko. Faktycznie, zazwyczaj bity grupuje się co najmniej w bajty, jeżeli nie w czwórki bajtów (najpopularniejsze obecnie liczby całkowite 32-bitowe), ósemki itd., a dopiero na takich większych paczkach w procesorze wykonywane są wszystkie interesujące nas operacje.



Rys. 2

Idąc za tą sugestią, możemy trochę lepiej „upakować” tablicę t, na przykład w ten sposób, że podzielimy każdy jej wiersz $t[i]$ nie na pojedyncze bity, ale na paczki po $B = 32$ bity (rys. 2). Każdy wiersz otrzymanej tablicy t' będzie się składał wówczas z $\lceil \frac{n}{B} \rceil$ takich paczek, przy czym wartość logiczną odpowiadającą $t[i][j]$ w starej tablicy można odnaleźć w nowej jako bit o numerze j **mod** B w paczce bitów $t'[i][j$ **div** $B]$ (tutaj **div** i **mod** oznaczają odpowiednio iloraz i resztę z dzielenia). Ponieważ – co może być dla Czytelnika nieco zaskakujące – w większości popularnych języków programowania wartości logiczne pamiętane są w zmiennych jednobajtowych (a nie jednobitowych!), więc w ten sposób udało nam się choć trochę zmniejszyć zapotrzebowanie algorytmu na pamięć. Dużo bardziej zaskakujące jest jednak to, że przedstawiona modyfikacja tablicy t pozwala istotnie zredukować czas działania rozwiązania.

Aby to zauważyć, przyjrzyjmy się najbardziej wewnętrznej pętli **for**. Otóż przeglądamy w niej równolegle kolejne pola wierszy $t[i]$ oraz $t[j]$ i zwiększamy wynik wtedy, gdy natrafimy na parę pól zawierających jedynki. Przy nowej reprezentacji tablicy możemy to wykonać trochę efektywniej: zamiast obliczać koniunkcje logiczne pojedynczych par bitów, będziemy to robić na całych paczkach B-bitowych naraz. Korzystamy tu z faktu, iż działanie **and** przedłuża się we wszystkich bodaj językach programowania na działanie na liczbach całkowitych „po bitach”, które to działanie z kolei stanowi pojedynczą instrukcję standardowych procesorów. Oto pseudokod takiego rozwiązania:

$w := 0;$
for $i := 0$ **to** $n - 1$ **do**
  for $j := 0$ **to** $n - 1$ **do**
    if $(t'[i][j$ **div** $B]$ **and** $2^{j \bmod B})$ **then**
      for $k := 0$ **to** $\lceil \frac{n}{B} \rceil - 1$ **do**
        $w := w +$ zapalone_bity$(t'[i][k]$ **and** $t'[j][k]);$
return $w;$

W stosunku do poprzedniego pseudokodu zmienił nam się, po pierwsze, warunek w pierwszej instrukcji **if**. Na szczęście jego nowa postać daje się łatwo zrealizować za pomocą pojedynczych operacji arytmetycznych oraz logicznych: koniunkcji i przesunięcia bitowego.

W językach programowania występują dwa typy przesunięć bitowych – w lewo, np. $(101)_2$ **shl** $3 = (101000)_2$, oraz w prawo, np. $(11010)_2$ **shr** $2 = (110)_2$. One także stanowią pojedyncze instrukcje w popularnych procesorach.

A zatem pozostała nam już tylko jedna niewyjaśniona kwestia, a mianowicie implementacja zliczania zapalonych bitów (czyli jedynek) w liczbie całkowitej. Okazuje się, że w niektórych wariantach języków programowania po prostu mamy do dyspozycji odpowiednią funkcję, np. `__builtin_popcount` w kompilatorach z rodziny GCC (języki C/C++). Jeśli jednak nie mamy tyle szczęścia, zawsze możemy wykorzystać metodę zwaną *tablicowaniem*. Polega ona na tym, że jeszcze przed uruchomieniem całego algorytmu obliczamy tablicę *bity*, która dla każdej liczby całkowitej D-bitowej k przechowuje liczbę zapalonych bitów w k. Wartość D dobieramy tak, aby tablica ta była stosunkowo niewielka, żeby dało się ją szybko wypełnić wartościami najprostszą możliwą metodą, czyli przeglądając kolejne bity poszczególnych liczb D-bitowych – dobrym kandydatem jest np. $D = 16$. Mając do dyspozycji tablicę *bity*, możemy dokończyć rozwiązanie na dwa sposoby: najprościej, tzn. zmniejszając w całym rozwiązaniu wartość parametru B do 16 (wówczas obliczenie wartości funkcji zapalone_bity sprowadza się do pojedynczego odwołania do tablicy *bity*), albo utrzymując wartość $B = 32$, ale każdorazowo dzieląc liczby, których zapalone bity chcemy zliczyć, na połówki odpowiadające

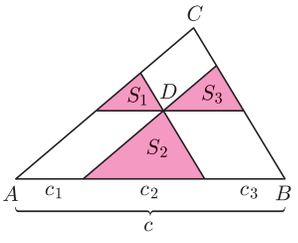
Czytelników, którzy nie czują się zbyt pewnie, używając operacji bitowych, zachęcamy do potrenowania na łamigłówkach bitowych z *Delty* 11/2008.

16 mniej i 16 bardziej znaczącym bitom (tutaj mamy dwa odwołania do tablicy). Można sprawdzić, że w każdym przypadku otrzymujemy rozwiązanie wykonujące mniej operacji od wyjściowego, siłowego $O(n^3)$.

W ten sposób zakończyliśmy opis ulepszonego algorytmu zliczania trójkątów w grafie. Nie sposób jednak oprzeć się wrażeniu, iż owo ulepszenie jest bardzo nieznaczne – na tyle nieznaczne, że nie jest pewne, czy warto było pisać o nim cały ten artykuł. Aby się przekonać o tym, że jednak może ono mieć sens, wystarczy się przyjrzeć temu, jak rozwija się współcześnie praktyczna strona informatyki. Otóż już w standardowych procesorach mamy do dyspozycji zmienne 64-bitowe (są one jednak mniej efektywne niż 32-bitowe – to fakt), ale coraz częściej pojawiają się już tzw. maszyny 64-bitowe i 128-bitowe (te ostatnie są jednak wciąż dosyć rzadkie). To oznacza także wzrost możliwej wartości parametru B, a przecież nasz algorytm (zakładając, dla uproszczenia, możliwość zliczenia zapalonych bitów liczby za pomocą pojedynczej instrukcji procesora) ma złożoność czasową $O(n^3/B)$. Mając takie względy na uwadze, informatycy-teoretycy już od jakiegoś czasu uwzględniają parametr B w analizach złożoności czasowych swoich algorytmów. Ważne jest tu podstawowe założenie, iż operacje arytmetyczne i logiczne na wbudowanych liczbach całkowitych można wykonywać w czasie stałym (niezależnym od B), które to założenie jest podstawą analiz złożoności bodaj wszystkich istniejących algorytmów.

A na sam koniec pozostawiamy Czytelnikom ćwiczenie z tej samej serii. Popularną metodą obliczania najkrótszych ścieżek między wszystkimi parami wierzchołków w grafie (skierowanym bądź nie) jest algorytm Floyda–Warshalla. Istnieje także prostsza wersja tego algorytmu, w której dla każdej pary wierzchołków sprawdzamy tylko, czy istnieje jakakolwiek ścieżka łącząca te wierzchołki (tę wersję nazywa się też często algorytmem wyznaczania domknięcia przechodniego grafu) – patrz pseudokod poniżej. W jaki sposób można użyć operacji na bitach do usprawnienia tego algorytmu?

```
for k := 0 to n − 1 do
  for i := 0 to n − 1 do
    for j := 0 to n − 1 do
      t[i][j] := t[i][j] or (t[i][k] and t[k][j]);
```



Jeszcze jeden wzór na pole trójkąta

W związku z *Deltoidem* z numeru 4/2009, w którym autorka zachęca do poszukiwania wzorów na pole trójkąta, chciałbym przedstawić pewien dość nietypowy wzór „fraktalny”. Nietypowy, bo pole trójkąta przedstawia jako pewną zależność od pól swoich „wewnętrznych” trójkątów – i tylko od nich. Fraktalny, ponieważ w nieskończoność można powielać pewien algorytm.

Wygląda to tak. Trójkąt ABC dzielimy trzema prostymi równoległymi do boków, przecinającymi się w dowolnym punkcie wewnętrznym D (rysunek). W wyniku takiego podziału otrzymamy trzy trójkąty podobne do trójkąta ABC i trzy równoległoboki. Oznaczając przez S_1, S_2, S_3 pola wskazanych trójkątów oraz przez S pole trójkąta ABC, otrzymujemy wzór

$$S = (\sqrt{S_1} + \sqrt{S_2} + \sqrt{S_3})^2.$$

Dowód. Z podobieństwa trójkątów mamy

$$\frac{c_1}{c} = \frac{\sqrt{S_1}}{\sqrt{S}}, \qquad \frac{c_2}{c} = \frac{\sqrt{S_2}}{\sqrt{S}}, \qquad \frac{c_3}{c} = \frac{\sqrt{S_3}}{\sqrt{S}},$$

i otrzymujemy

$$c = c_1 + c_2 + c_3 = c\frac{\sqrt{S_1}}{\sqrt{S}} + c\frac{\sqrt{S_2}}{\sqrt{S}} + c\frac{\sqrt{S_3}}{\sqrt{S}}.$$

Stąd

$$1 = \frac{\sqrt{S_1}}{\sqrt{S}} + \frac{\sqrt{S_2}}{\sqrt{S}} + \frac{\sqrt{S_3}}{\sqrt{S}},$$

więc

$$S = (\sqrt{S_1} + \sqrt{S_2} + \sqrt{S_3})^2. \ \square$$

Dalej, analogicznie, można podzielić dowolny z trzech małych trójkątów i kontynuować ten proces, otrzymując trójkąty o polach $T_1, T_2, \ldots, T_n$ oraz wzór

$$S = \Big(\sum_{k=1}^{n} \sqrt{T_k}\Big)^2.$$

Przy $n \to \infty$ uzyskujemy zatem

$$S = \Big(\sum_{k=1}^{\infty} \sqrt{T_k}\Big)^2.$$

Wiadomo więc, że szereg $\sum_{k=1}^{\infty} \sqrt{T_k}$ jest zbieżny. Biorąc konkretne algorytmy dzielenia trójkątów, otrzymamy pewne szeregi liczbowe, których zbieżność w sposób klasyczny może być kłopotliwa do wykazania. Zachęcam Czytelników do poszukania takich przykładów.

Andrzej WRZESIEŃ[*]

[*] Politechnika Łódzka