Jacek Tomasiewicz
Treść zadania, Opracowanie

Mateusz Baranowski
Program

Dostępna pamięć: 64 MB.

OI, Etap III, dzień próbny, 05.04.2011

Dynamit

Jaskinia Bajtocka składa się z n komór oraz z $n-1$ łączących je korytarzy. Nie wychodząc z jaskini, pomiędzy każdą parą komór można przejść bez zawracania na dokładnie jeden sposób. W niektórych komorach pozostawiono dynamit. Wzdłuż każdego korytarza rozciągnięto lont. W każdej komorze wszystkie lonty z prowadzących do niej korytarzy są połączone, a jeżeli w danej komorze znajduje się dynamit, to jest on połączony z lontem. Lont biegnący korytarzem pomiędzy sąsiednimi komorami spala się w jednej jednostce czasu, a dynamit wybucha dokładnie w chwili, gdy ogień znajdzie się w komorze zawierającej ten dynamit.

Chcielibyśmy równocześnie podpalić lonty w jakichś m komorach (w miejscu połączenia lontów) w taki sposób, aby wszystkie ładunki dynamitu wybuchły w jak najkrótszym czasie, licząc od momentu podpalenia lontów. Napisz program, który wyznaczy najkrótszy możliwy taki czas.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant m \leqslant n \leqslant 300\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio liczbę komór w jaskini oraz liczbę komór, w których możemy podpalić lonty. Komory są ponumerowane od 1 do n. Kolejny wiersz zawiera n liczb całkowitych $d_1, d_2, \ldots, d_n$ ($d_i \in \{0, 1\}$), pooddzielanych pojedynczymi odstępami. Jeśli $d_i = 1$, to w i-tej komorze znajduje się dynamit, a jeśli $d_i = 0$, to nie ma w niej dynamitu. Kolejne $n-1$ wierszy opisuje korytarze w jaskini. W każdym z nich znajdują się dwie liczby całkowite a, b ($1 \leqslant a < b \leqslant n$) oddzielone pojedynczym odstępem i oznaczające, że istnieje korytarz łączący komory a i b. Każdy korytarz pojawia się w opisie dokładnie raz.

Możesz założyć, że w testach wartych łącznie 10% punktów zachodzi dodatkowy warunek $n \leqslant 10$, a w testach wartych łącznie 40% punktów zachodzi $n \leqslant 1\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnemu czasowi od podpalenia lontów, po jakim wybuchną wszystkie ładunki dynamitu.

Przykład

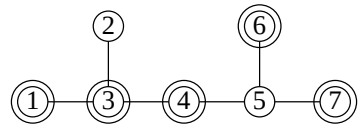
Dla danych wejściowych:

```
7 2
1 0 1 1 0 1 1
1 3
2 3
```

```
3 4
4 5
5 6
5 7
```

poprawnym wynikiem jest:

```
1
```



Wyjaśnienie do przykładu: *Podpalamy lonty w komorach 3 i 5. W chwili zero wybucha dynamit w komorze 3, a po jednostce czasu zapalone zostają dynamity w komorach 1, 4, 6 i 7.*

Rozwiązanie

Wprowadzenie

Niestety wszystkie trywialne rozwiązania tego zadania działają w czasie wykładniczym. Spróbujmy wobec tego na początek włożyć nieco wysiłku w stworzenie jakiegokolwiek rozwiązania wielomianowego. Później przyjdzie czas na jego przyspieszanie.

Skoncentrujmy się na razie na nieco prostszym problemie — sprawdzaniu, czy da się podpalić lonty w co najwyżej m komorach, tak aby wszystkie ładunki wybuchły w pierwszych t sekundach po podpaleniu lontów. Oczywiście, wielomianowe rozwiązanie tego problemu natychmiast daje wielomianowe rozwiązanie całego zadania. Możemy sprawdzać po kolei wszystkie możliwe t.

Problem decyzyjny dla ustalonego t

Dla uproszczenia języka w naszych rozważaniach, komory, w których podpalamy lonty (oraz odpowiadające im wierzchołki), nazwiemy *startowymi*. Układ co najwyżej m komór startowych spełniających opisany warunek nazwiemy zaś *rozwiązaniem*.

Wciąż nie narzuca nam się żaden trywialny wielomianowy algorytm. W takich przypadkach zwykle pozostaje szukać obserwacji, które pozwolą np. zredukować rozmiar danych czy też nałożyć dodatkowe warunki, które musi spełniać choć jedno rozwiązanie, jeśli w ogóle jakiekolwiek rozwiązanie istnieje.

Zastanówmy się, czy są jakieś wierzchołki, o których możemy bez straty ogólności przyjąć, że są lub nie są startowe. Nietrudno na przykład dostrzec, że jeśli w naszym drzewie jest liść, w którym nie ma dynamitu, nie ma sensu wybierać go jako startowego. Ogień zaprószony w liściu obejmie w ciągu pierwszych t sekund mniej wierzchołków (w sensie inkluzji, czyli zawierania zbiorów) niż zaprószony w jego sąsiedzie. Można zatem przerobić każde rozwiązanie, w którym rozważany liść jest wierzchołkiem startowym, tak aby zamiast niego wierzchołkiem startowym był jedyny sąsiad tego liścia. Podobnie jest wtedy, gdy w liściu jest dynamit, lecz $t > 0$.

Powyższe spostrzeżenie to krok w dobrą stronę. Pierwszą część można nieco wzmocnić — jeśli w liściu nie ma dynamitu, możemy w ogóle usunąć go z drzewa z gwarancją, że odpowiedź na postawione przed nami pytanie pozostanie bez zmiany. Usuwając do skutku takie liście, w końcu dostaniemy drzewo, w którym w każdym liściu znajduje się ładunek dynamitu. Być może będzie to drzewo puste lub mające

tylko pojedynczy wierzchołek. Jeśli tak się stanie, z odpowiedzią poradzimy sobie bez problemu, więc dalej będziemy zakładać przeciwnie.

Z drugą częścią spostrzeżenia jest pewien problem — nie pozwala krok po kroku eliminować wierzchołków jako potencjalnych startowych. Taka sytuacja występuje na przykład dla drzewa składającego się z dwóch wierzchołków zawierających dynamit i połączonych krawędzią. Dla $t > 0$ musimy wybrać któryś z nich, choć obydwa są liśćmi. Aby zapobiec tego rodzaju sytuacjom, ukorzenimy drzewo. Teraz zamiast mówić, że dla każdego liścia istnieje rozwiązanie, w którym ten liść nie będzie startowy, można sformułować silniejszy fakt. Istnieje rozwiązanie, w którym żaden liść, być może z wyjątkiem korzenia, nie jest startowy.

Analizując uzasadnienie wyjściowego spostrzeżenia i idąc dalej tą drogą, możemy otrzymać silniejszą obserwację.

Obserwacja 1. Niech $N_t(v)$ oznacza zbiór wierzchołków, które są odległe od v o co najwyżej t. Niech U będzie zbiorem wierzchołków drzewa o tej własności, że dla każdego $u \in U$ zachodzi $N_t(u) \subseteq N_t(\mathit{parent}(u))$, gdzie $\mathit{parent}(u)$ oznacza ojca w drzewie. Wówczas każde rozwiązanie da się przerobić tak, aby nie zawierało wierzchołków ze zbioru U.

Dowód: Dowodem będzie algorytm odpowiednio przerabiający rozwiązanie. Wystarczy bardzo prosty — dopóki jakiś $u \in U$ jest startowy, zamiast u bierzemy jego ojca. Jeśli ojciec już był wierzchołkiem startowym, po prostu usuwamy u z rozwiązania. Z założeń jest jasne, że po każdym kroku zbiór wierzchołków startowych wciąż jest rozwiązaniem. Ponadto w każdym kroku elementy zbioru startowego wędrują w górę drzewa (lub w ogóle znikają), więc nasza procedura ma własność stopu. ■

Zastanówmy się, co wiemy o zbiorze U. Na pewno nie należy do niego korzeń (nie ma zdefiniowanego ojca), zaś dla $t > 0$ należą do niego liście. Chwila pomyślunku pozwala w pełni scharakteryzować ten zbiór — należą do niego te wierzchołki różne od korzenia, których wszyscy potomkowie są od nich odlegli o co najwyżej $t-1$. Innymi słowy, poddrzewa zaczepione w tych wierzchołkach mają wysokość co najwyżej $t-1$ (uznajemy tu, że poddrzewo puste ma wysokość 0, jednowierzchołkowe 1 itd.).

Dla dość dużego zbioru wierzchołków U ustaliliśmy, że nie będziemy wybierać ich jako startowe. Wcześniej przez ucinanie liści doprowadziliśmy nasze drzewo do sytuacji, w której w każdym liściu jest dynamit. Okazuje się, że w takiej sytuacji zawsze jesteśmy w stanie wskazać co najmniej jeden wierzchołek, który musimy wybrać jako startowy. Jeśli wysokość drzewa (oznaczmy ją przez h) nie przekracza t, jedynym wierzchołkiem poza U jest korzeń, więc sprawa jest jasna. W przeciwnym przypadku rozpatrzmy najgłębszy liść w drzewie, czyli dowolny liść na głębokości $h-1$. Wiemy, że jest tam ładunek, który musi znaleźć się w zasięgu pewnego wierzchołka startowego. Wszystkie wierzchołki o głębokości $h-t$ lub większej są jednak w zbiorze U. Stąd jedynym wierzchołkiem spoza U, w którego zasięgu jest nasz liść, jest jego przodek o głębokości $h-t-1$. Ten właśnie wierzchołek musi być jednym ze startowych.

Istnienie wierzchołka, który musi być startowy, to dla nas znakomita wiadomość. Możemy bowiem zaznaczyć, że zużyliśmy jeden taki wierzchołek, usunąć ładunki znajdujące się w jego zasięgu i kontynuować obliczenia. Usuwamy w szczególności ładunek w liściu, więc poprzednia redukcja pozwoli zmniejszyć drzewo. Wobec tego nigdy nie

utkniemy w martwym punkcie. Otrzymaliśmy tym samym pierwszy algorytm rozwiązujący nasz problem. Oto jego schematyczny zapis:

```
1:  procedure Sprawdź(t)
2:  begin
3:    while |T| > 0 do begin
4:      if pewien liść nie zawiera ładunku then
5:        usuń ten liść
6:      else if m = 0 then
7:        return NIE
8:      else begin
9:        h := wysokość drzewa T;
10:       if h ⩽ t then return TAK;
11:       v := wierzchołek o t poziomów ponad najgłębszym liściem;
12:       m := m − 1;
13:       usuń ładunki wybuchowe w zbiorze N_t(v);
14:     end
15:   end
16:   return TAK;
17: end
```

Zbadajmy, jak szybko działa otrzymany algorytm. Najwięcej zajmie usuwanie ładunków wybuchowych — odwiedzanie wszystkich wierzchołków z $N_t(v)$ może za każdym razem zajmować czas liniowy. Wszystkie inne operacje w pojedynczym obrocie pętli też oczywiście łatwo wykonamy liniowo, a obrotów jest liniowo wiele (w każdym usuwamy liść, ładunek z liścia lub wychodzimy z pętli). Stąd łączny czas działania możemy oszacować przez $O(n^2)$.

Przypomnijmy, że skonstruowany właśnie algorytm musimy do rozwiązania całego zadania wykonać $O(n)$, a dokładniej $O(wynik)$ razy. Stąd łączna złożoność to $O(n^2 \cdot wynik)$.

Przyspieszamy algorytm

Ograniczamy liczbę sprawdzeń

Jak na razie nie przykładaliśmy wagi do dobrej złożoności, tylko do otrzymania jakiegokolwiek wielomianowego algorytmu. W szczególności, bardzo nieoszczędnie zredukowaliśmy zadanie do sprawdzania, czy istnieje rozwiązanie dla konkretnych t. Oczywiście, ta własność jest monotoniczna — jeśli jest prawdziwa dla t_0, to także dla każdego $t \geqslant t_0$.

Taka sytuacja to typowe okoliczności przemawiające za zastosowaniem wyszukiwania binarnego. Wystarczy zatem wywołać funkcję sprawdzającą $O(\log n)$ razy. Zredukowaliśmy tym samym złożoność do $O(n^2 \log n)$. Programy implementujące taki algorytm zdobywały ok. 40 punktów. Przykładowe rozwiązania można znaleźć w plikach `dyns3.c`, `dyns4.cpp` oraz `dyns5.pas`.

Dociekliwy Czytelnik może zauważyć, że dla małych wyników tak naprawdę nasz algorytm jest teraz wolniejszy. Okazuje się jednak, że można tak przerobić wyszukiwanie binarne kosztem stałej ukrytej w notacji O, aby działało w $O(\log(wynik + 2))$ krokach. Pozostawiamy tę modyfikację jako ćwiczenie dla Czytelnika. W zadaniu nie była oczywiście wymagana.

Przyspieszamy sprawdzenie

Doszliśmy do momentu, w którym nie jesteśmy w stanie istotnie poprawić złożoności całego rozwiązania, nie przyspieszając funkcji sprawdzającej. Poprawmy wobec tego jej złożoność.

Nasz poprzedni algorytm idzie z grubsza w kierunku od liści do korzenia, zapewniając, że wszystkie ładunki na kolejnych poziomach znajdują się w zasięgu pewnego wierzchołka startowego. Staramy się przy tym umieścić wierzchołki startowe najbliżej korzenia jak tylko się da. Na razie nie idziemy jednak po prostu z dołu do góry, czasami skaczemy t poziomów w górę, by ustawić wierzchołek startowy i usunąć pewne ładunki. Spróbujemy „opóźnić" te operacje i postępować jednoprzebiegowo. Cały czas chcemy umieszczać wierzchołki startowe najwyżej jak się da, czyli tak, jak czyni to poprzedni algorytm. Na takie podejście można również spojrzeć jak na programowanie dynamiczne — zdefiniujemy pewną funkcję i wyznaczymy jej wartość dla wierzchołka na podstawie wartości dla synów.

Zastanówmy się, kiedy wybieramy jakiś wierzchołek jako startowy. Na pewno wówczas, gdy pewien ładunek znajduje się w poddrzewie tego wierzchołka dokładnie t poziomów niżej i wciąż nie zapewniliśmy jego detonacji (w skrócie: nie zdetonowaliśmy go). Również wtedy, gdy taki ładunek jest nieco wyżej, ale dotarliśmy już do korzenia. Ta sytuacja odpowiada umieszczeniu w poprzednim algorytmie ładunku o t poziomów wyżej niż najgłębszy dotychczas nierozpatrzony. Nie dopuścimy do sytuacji, gdy pewien ładunek pozostanie o więcej niż t poziomów głębiej niż aktualnie rozważany wierzchołek, więc przy okazji tej operacji zdetonujemy wszystkie ładunki w naszym poddrzewie. Nie trzeba zatem pamiętać wszystkich niezdetonowanych ładunków — wystarczy najgłębszy z nich, a właściwie jego głębokość względem aktualnie rozpatrywanego wierzchołka.

Wybierając wierzchołek jako startowy, możemy również zdetonować pewne ładunki spoza poddrzewa. Ogień dociera do nich przez ojca wierzchołka, w którym zaczepione jest poddrzewo, więc ich rozpatrzenie zostawimy na później. W tym celu musimy zapamiętać, gdzie umieściliśmy wierzchołek startowy. Wystarczy nam względna głębokość najwyżej położonego wierzchołka startowego w rozważanym poddrzewie. Dzięki temu, gdy będziemy przetwarzać pewien wierzchołek, będziemy mogli sprawdzić, czy najniżej położony niezdetonowany ładunek w poddrzewie pewnego syna jest w zasięgu wierzchołka startowego w poddrzewie innego syna.

Opisane dwie wartości wystarczają zatem do przeprowadzenia obliczeń. Pierwszą, czyli głębokość najniższego niezdetonowanego ładunku (lub $-\infty$, gdy takiego nie ma), zapiszmy w tablicy *dynamit*. Drugą, czyli głębokość najwyższego wierzchołka startowego (lub ∞, gdy takiego nie ma) — w tablicy *ogień*. Zbierzmy dotychczasowe rozważania w postaci pseudokodu, w którym zilustrujemy pojedynczy krok algorytmu. Warto pamiętać o szczególnym traktowaniu korzenia. Ten detal może łatwo umknąć przy implementacji.

```
 1: procedure Obsłuż(v)
 2: begin
 3:   { niech v oznacza bieżący wierzchołek }
 4:   ogień[v] := ∞;
 5:   dynamit[v] := −∞;
 6:   if w v jest dynamit then
 7:     dynamit[v] := 0;
 8:   foreach w : syn v do begin
 9:     dynamit[v] := max(dynamit[v], dynamit[w] + 1);
10:     ogień[v] := min(ogień[v], ogień[w] + 1);
11:   end
12:   if dynamit[v] + ogień[v] ⩽ t then begin
13:   { istniejące wierzchołki startowe zdetonują wszystkie pozostałe ładunki }
14:     dynamit[v] := −∞;
15:   end
16:   if dynamit[v] = t or (v to korzeń and dynamit[v] ≠ −∞) then begin
17:   { wybieramy v jako startowy }
18:     m := m − 1;
19:     dynamit[v] := −∞;
20:     ogień[v] := 0;
21:   end
22: end
```

Cały algorytm sprawdzania, czy istnieje rozwiązanie (czyli m wierzchołków, w zasięgu których jest każdy ładunek wybuchowy), polega po prostu na wykonaniu przedstawionego algorytmu dla każdego wierzchołka. Czynimy to od liści do korzenia, ponieważ korzystamy z wartości dla synów. Rozwiązanie istnieje wtedy i tylko wtedy, gdy udało nam się zużyć co najwyżej m wierzchołków startowych, czyli gdy na końcu m jest nieujemne.

Złożoność czasowa takiej funkcji sprawdzającej jest liniowa. Wobec tego cały algorytm, jeśli korzysta z wyszukiwania binarnego, działa w czasie $O(n \log n)$. Na tej zasadzie działało rozwiązanie wzorcowe, zaimplementowane je w plikach `dyn.c`, `dyn1.cpp` i `dyn2.pas`.

Rozwiązanie siłowe

Jak głosi treść zadania, do otrzymania 10 punktów wystarcza program działający dla $n \leqslant 10$, czyli w praktyce brutalne rozwiązanie wykładnicze. Dobrze napisane takie rozwiązanie działa w czasie $O(n \cdot \binom{n}{m})$. Wystarczała jednak złożoność czasowa $O(2^n + nm \cdot \binom{n}{m})$. Takie rozwiązanie polegało na wygenerowaniu wszystkich podzbiorów zbioru wierzchołków, odfiltrowaniu m-elementowych i dla każdego z nich przeprowadzeniu symulacji. Można ją przeprowadzić przy użyciu jednego wywołania algorytmu BFS o wielu źródłach (w czasie $O(n)$) lub m zwykłych wywołań tego algorytmu (czas łącznie $O(mn)$). Przykładowe rozwiązania wykładnicze można znaleźć w plikach `dyns6.c` i `dyns7.pas`.

Testy

Zadanie było sprawdzane na 10 grupach testów, zawierających łącznie 34 przypadki testowe. Krótkie opisy testów są zawarte w następującej tabeli.

Nazwa	n	m	Opis
dyn1a.in	1	1	minimalny
dyn1b.in	1	1	minimalny
dyn1c.in	6	2	prosty poprawnościowy
dyn1d.in	10	3	poprawnościowy
dyn2a.in	50	3	mało rozgałęzień i podpaleń
dyn2b.in	97	8	dużo rozgałęzień
dyn2c.in	80	10	dużo podpaleń
dyn3a.in	500	10	mało rozgałęzień i podpaleń
dyn3b.in	600	40	dużo rozgałęzień i dynamitu
dyn4a.in	800	8	mały test losowy, mało podpaleń
dyn4b.in	1000	50	mały test, dużo podpaleń
dyn5a.in	10 000	100	losowy
dyn5b.in	23 450	1 000	dużo dynamitu
dyn5c.in	30 000	2	mało dynamitu, drzewo binarne
dyn5d.in	30 000	30	linia
dyn6a.in	50 000	5	losowy
dyn6b.in	70 000	13	dużo dynamitu
dyn6c.in	80 000	1 000	jedno duże rozgałęzienie
dyn6d.in	76 500	66	kilka wierzchołków z dużą liczbą krawędzi
dyn7a.in	100 000	600	losowy
dyn7b.in	100 000	16	dużo dynamitu
dyn7c.in	100 000	11	mało dynamitu, małe rozgałęzienia
dyn7d.in	110 000	23	losowy, dynamit w każdej komnacie
dyn8a.in	200 000	17	losowy
dyn8b.in	230 000	27	dużo dynamitu
dyn8c.in	250 000	14	dynamit w każdej komnacie
dyn8d.in	250 000	14	dużo dynamitu
dyn9a.in	299 900	7	dużo rozgałęzień
dyn9b.in	299 990	33	dużo dynamitu
dyn9c.in	299 999	12 033	regularne drzewo złożone ze ścieżek długości 5

Nazwa	n	m	Opis
dyn10a.in	300 000	1	maksymalny test
dyn10b.in	300 000	49	dużo dynamitu
dyn10c.in	300 000	30	drzewo binarne o dużej wysokości
dyn10d.in	300 000	299 999	maksymalna gwiazda