Zbigniew J. Czech
Treść zadania, Opracowanie

Piotr Stańczyk
Program

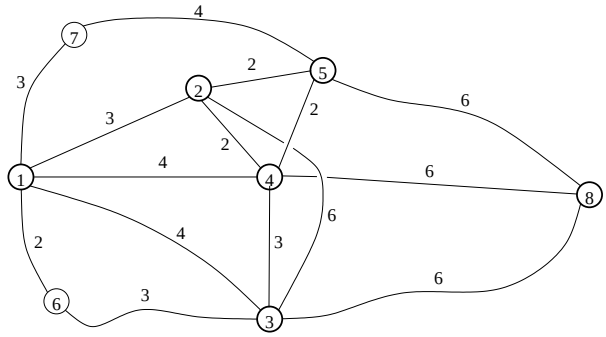
OI, Etap I

Atrakcje turystyczne

Bajtazar jedzie z Bitowic do Bajtogrodu. Po drodze chce odwiedzić kilka wybranych miejscowości, w których znajdują się interesujące zabytki, dobre restauracje czy inne atrakcje turystyczne. Kolejność odwiedzania wybranych miejscowości nie jest całkowicie obojętna. Na przykład, Bajtazar wolałby nie wspinać się na wieżę zamku Bitborku po sutym obiedzie zjedzonym w Cyfronicach, a do Zipowic (na słynną kawę Compresso) chciałby wpaść raczej po obiedzie, a nie przed. Jednak kolejność odwiedzania wybranych miejscowości nie jest całkowicie ustalona i do pewnego stopnia elastyczna. Ze względu na obłędne ceny benzyny, Bajtazar chce tak zaplanować trasę przejazdu, żeby była jak najkrótsza. Pomóż mu wyznaczyć długość najkrótszej trasy spełniającej jego wymagania.

Sieć drogowa składa się z n miejscowości i łączących je m dróg. Miejscowości są ponumerowane od 1 do n, a drogi od 1 do m. Każda droga łączy dwie różne miejscowości i jest dwukierunkowa. Drogi spotykają się tylko w miejscowościach (w których mają końce) i nie przecinają się poza nimi. Drogi mogą prowadzić przez estakady i tunele. Każda droga ma określoną długość. Parę miejscowości może łączyć co najwyżej jedna bezpośrednia droga.

Oznaczmy przez k liczbę wybranych miejscowości, które chce odwiedzić Bajtazar. Numeracja miast jest taka, że Bitowice mają numer 1, Bajtogród numer n, a miejscowości, które chce odwiedzić Bajtazar, mają numery $2, 3, \ldots, k+1$.



Na rysunku przedstawiono przykładową sieć dróg. Powiedzmy, że Bajtazar chce odwiedzić miejscowości 2, 3, 4 i 5, przy czym miejscowość 2 chce odwiedzić przed miejscowością 3, a miejscowości 4 i 5 po miejscowości 3. Wówczas najkrótsza trasa biegnie przez miejscowości 1, 2, 4, 3, 4, 5, 8 i ma ona długość 19.

Zauważmy, że miejscowość 4 pojawia się na tej trasie przed i po miejscowości 3. Jednak przed odwiedzeniem miejscowości 3 Bajtazar nie zatrzyma się w niej, gdyż takie przyjął ograniczenia. Nie oznacza to jednak, że w ogóle nie może wcześniej przejeżdżać przez tę miejscowość.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci drogowej, listę wybranych miejscowości, które chce odwiedzić Bajtazar oraz ograniczenia, co do kolejności, w jakiej chce je odwiedzić,*
- *wyznaczy długość najkrótszej trasy, przechodzącej przez wszystkie wybrane przez Bajtazara miejscowości w odpowiedniej kolejności,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m i k pooddzielane pojedynczymi odstępami, $2 \leqslant n \leqslant 20\,000$, $1 \leqslant m \leqslant 200\,000$, $0 \leqslant k \leqslant 20$; ponadto jest spełniona nierówność $k \leqslant n-2$.

Kolejne m wierszy zawiera opisy dróg, po jednej w wierszu. Wiersz $i+1$-szy zawiera trzy liczby całkowite p_i, q_i i l_i, pooddzielane pojedynczymi odstępami, $1 \leqslant p_i < q_i \leqslant n$, $1 \leqslant l_i \leqslant 1\,000$. Liczby te oznaczają drogę łączącą miejscowości p_i i q_i, o długości l_i. Możesz założyć, że dla każdych danych testowych można z Bitowic dojechać do Bajtogrodu oraz do wszystkich miejscowości, które Bajtazar chce odwiedzić.

W $m+1$-szym wierszu znajduje się jedna liczba całkowita g, $0 \leqslant g \leqslant \frac{k \cdot (k-1)}{2}$. Jest to liczba ograniczeń dotyczących kolejności odwiedzania wybranych przez Bajtazara miast. Ograniczenia te są podane w kolejnych g wierszach, po jednym w wierszu. Wiersz $m+i+1$-szy zawiera dwie liczby całkowite r_i i s_i oddzielone pojedynczym odstępem, $2 \leqslant r_i \leqslant k+1$, $2 \leqslant s_i \leqslant k+1$, $r_i \neq s_i$. Para liczb r_i i s_i oznacza, że Bajtazar chce odwiedzić miejscowość r_i przed odwiedzeniem miejscowości s_i. Nie oznacza to, że nie może przejechać przez s_i przed odwiedzeniem r_i ani że nie może przejechać przez r_i po odwiedzeniu s_i, jednak nie będzie się on wtedy zatrzymywał ani zwiedzał żadnych atrakcji turystycznych. Możesz założyć, że dla każdych danych testowych istnieje przynajmniej jedna kolejność zwiedzania wybranych miejscowości spełniająca wszystkie ograniczenia.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, będącą długością najkrótszej trasy z Bitowic do Bajtogrodu, przechodzącej przez wszystkie wybrane przez Bajtazara miejscowości w odpowiedniej kolejności.

Przykład

Dla danych wejściowych:

```
8 15 4
1 2 3
1 3 4
1 4 4
1 6 2
1 7 3
2 3 6
2 4 2
2 5 2
3 4 3
3 6 3
3 8 6
4 5 2
4 8 6
5 7 4
5 8 6
3
2 3
3 4
3 5
```

poprawnym wynikiem jest:

```
19
```

Rysunek i wyjaśnienie do przykładu znajdują się w treści zadania.

Rozwiązanie

Problem przestawiony w zadaniu możemy w naturalny sposób wyrazić w języku grafów — miasta to wierzchołki grafu, a drogi to nieskierowane krawędzie o określonych długościach. Wówczas poszukiwana trasa to najkrótsza ścieżka w grafie łącząca wierzchołek początkowy z końcowym i przebiegająca w określonym porządku przez pozostałe wierzchołki.

Zadanie polegające na wyznaczaniu najkrótszej ścieżki łączącej dwa wierzchołki i przechodzącej przez zadany zbiór wierzchołków jest problemem NP–trudnym, powiązanym z zagadnieniem wyznaczania drogi Hamiltona w grafie. Po dodaniu ograniczeń na kolejność odwiedzania miejscowości przez Bajtazara, zadanie pozostaje NP–trudne — każde znane rozwiązanie tego zadania działa w czasie wykładniczym ze względu na k.

Rozwiązanie zadania można podzielić na dwie niezależne części. W części pierwszej wyznaczamy odległości między wszystkimi parami miejscowości, które Bajtazar chce odwiedzić (włączając w to Bitowice i Bajtogród), czyli pomiędzy wierzchołkami o numerach $\{1, 2, \ldots, k+1, n\}$. Druga część rozwiązania zadania polega na wyznaczeniu najkrótszej drogi z Bitowic (wierzchołka o numerze 1) do Bajtogrodu (wierzchołka o numerze n), przechodzącej przez wszystkie wybrane miejscowości (wierzchołki o numerach $\{2, 3, \ldots, k+1\}$) w kolejności spełniającej zadane warunki.

Faza 1

W celu wyznaczenia długości najkrótszych ścieżek między wszystkimi parami miast, które Bajtazar chce odwiedzić, można wykonać $k+2$ razy algorytm Dijkstry, zaczynając kolejno z wierzchołków $1, 2, \ldots, k+1, n$. Za każdym razem algorytm wykonywany jest na całym grafie wejściowym, zatem czas jego działania to $O(m \cdot \log n)$. Ponieważ algorytm powtarzamy $O(k)$ razy, pierwszą fazę jesteśmy w stanie wykonać w sumarycznym czasie $O(k \cdot m \cdot \log n)$. Po każdym wykonaniu algorytmu Dijkstry zapamiętujemy odległości od wierzchołka startowego do pozostałych wierzchołków ze zbioru $\{1, 2, \ldots, k+1, n\}$.

Zamiast korzystać z algorytmu Dijkstry, w fazie pierwszej można zastosować również algorytm Floyda-Warshalla, uzyskując złożoność rzędu $O(n^3)$. Biorąc jednak pod uwagę możliwy rozmiar danych, rozwiązanie to ma istotnie gorszą złożoność.

Faza 2

Druga faza algorytmu polega na wyznaczeniu najkrótszej ścieżki wychodzącej z wierzchołka 1 i kończącej się w wierzchołku n. Ścieżka ta musi odwiedzić wszystkie wierzchołki ze zbioru $\{2, 3, \ldots, k+1\}$ i dodatkowo muszą być uwzględnione wymagania dotyczące kolejności odwiedzania tych wierzchołków.

Prostym sposobem wyznaczenia najkrótszej ścieżki jest wygenerowanie wszystkich możliwych ścieżek, a następnie wybranie najkrótszej spośród nich. Rozwiązanie takie sprowadza się do wygenerowania wszystkich permutacji zbioru k miast, które Bajtazar chce odwiedzić, sprawdzenia dla każdej permutacji, czy zachowane są ograniczenia dotyczące kolejności odwiedzania miast, a następnie obliczenia długości ścieżki. Na koniec, spośród wszystkich dozwolonych ścieżek, wybieramy najkrótszą i jej długość zwracamy jako wynik. Algorytm taki może być zaimplementowany w prosty sposób, by działał w czasie $O(k^2 \cdot k!)$, gdyż możemy mieć $O(k^2)$ ograniczeń, które należy sprawdzić dla każdej z wygenerowanych permutacji. Można rozwiązanie to nieco poprawić, osiągając złożoność $O(k \cdot k!)$. W tym celu wykorzystujemy fakt, iż zależności między kolejnością odwiedzania wierzchołków możemy reprezentować dla pojedynczego wierzchołka przy użyciu pojedynczej zmiennej typu całkowitego 32-bitowego (`int` w C/C++, `longint` w Pascalu), dzięki temu, że $k \leqslant 20 < 32$. Dla każdego wierzchołka przechowujemy jedną taką zmienną, której m-ty bit jest ustawiony na 1, o ile rozpatrywany wierzchołek jest zależny od wierzchołka o numerze m. Podczas sprawdzania zgodności permutacji wierzchołków z zadaną kolejnością ich odwiedzania, konstruujemy 32-bitową zmienną odwiedzenia, której m-ty bit jest ustawiony na 1, o ile wierzchołek o numerze m został już odwiedzony. Sprawdzenie zależności dla m-tego wierzchołka sprowadza się w takiej sytuacji do zweryfikowania, czy wszystkie zapalone bity w zmiennej zależności są zapalone w zmiennej odwiedzenia (czyli wykonaniu operacji logicznej AND). Dodatkowo, stosując algorytm generowania permutacji w kolejności wymagającej wykonywania jedynie transpozycji sąsiednich elementów[1], jesteśmy w stanie zaimplementować rozwiązanie tak, by działało w czasie $O(k!)$.

Możliwe jest także inne rozwiązanie, także oparte na pomyśle generowania wszystkich permutacji odwiedzanych wierzchołków. Możemy mianowicie generować stopniowo rozwiązanie (permutację), zatrzymując się w momencie, gdy dochodzimy do miejsca,

[1] Ten i inne algorytmy generowania permutacji można znaleźć w [22].

z którego nie można już kontynuować poprawnej ścieżki. Algorytm ten można zaimplementować tak, by działał w czasie $O(k \cdot p)$, gdzie p to liczba poprawnych permutacji.

W tym celu zaczynamy od pustego ciągu i dokładamy do niego kolejne wierzchołki spośród miast do odwiedzenia — takie, które są zależne jedynie od wierzchołków już znajdujących się w ciągu (czyli powinny być odwiedzone później niż one). W kroku pierwszym możemy umieścić w ciągu tylko wierzchołki niezależne — załóżmy, że wybraliśmy wierzchołek w. W drugim kroku możemy dołączyć wierzchołki niezależne (inne niż w) oraz wierzchołki zależne wyłącznie od wierzchołka w itd.

Powinniśmy jeszcze uzasadnić, że opisana procedura pozwoli nam przejrzeć wszystkie poprawne permutacje. Wystarczy zauważyć, że w każdej poprawnej permutacji, którą chcielibyśmy otrzymać, wierzchołki występują w kolejności zgodnej z zasadą, z jaką są umieszczane w ciągu — przed wierzchołkiem u nie może wystąpić żaden wierzchołek zależny od u. Dodatkowo, w zadaniu jest zagwarantowane, że dla przedstawionych danych istnieje rozwiązanie, więc przynajmniej jedną permutację znajdziemy.

Niestety, poprawnych permutacji może być wiele. W przypadku, gdy mamy niewiele warunków ograniczających kolejność odwiedzania miast (w szczególności, gdy nie ma ich wcale) liczba poprawnych permutacji może być bliska $k!$ i czas działania opisanego algorytmu może wynosić $O(k!)$.

Okazuje się jednak, że w zadaniu można zastosować technikę programowania dynamicznego. Dla każdego podzbioru $A \subseteq \{2,3,\ldots,k+1\}$ już odwiedzonych wierzchołków[2] oraz wierzchołka $v \in A$ (o ile A jest niepusty) odwiedzonego jako ostatni, możemy zapamiętać długość najkrótszej ścieżki z wierzchołka 1 do v odwiedzającej wszystkie wierzchołki A (i tylko przez te, spośród wybranych). Poszukiwana wartość dla zadanych A i v jest równa

$$\mathcal{D}(A,v) = \min(\mathcal{D}(A-\{w\},w) + d(w,v)),$$

gdzie minimum obliczamy po wszystkich wierzchołkach $w \in A$, a $d(w,v)$ to minimalna odległość między wierzchołkami w i v obliczona w pierwszej fazie. Gdy $A = \emptyset$, wówczas dla każdego $v \in \{2,3,\ldots,k+1\}$ przyjmujemy, że

$$\mathcal{D}(\emptyset,v) = 0.$$

Po obliczeniu wszystkich wartości $\mathcal{D}$, odpowiedzią na postawione w zadaniu pytanie będzie najmniejsza wartość $\mathcal{D}(A_0,v)$, dla $A_0 = \{2,\ldots,k+1\}$ oraz dowolnego $v \in A_0$, powiększona o odległość między wierzchołkiem v a wierzchołkiem n. Uzyskujemy w ten sposób algorytm działający w czasie $O(k^2 \cdot 2^k)$, gdyż mamy do rozpatrzenia nie więcej niż $k \cdot 2^k$ par złożonych z wierzchołka i podzbioru, a dla każdej pary obliczenia wykonujemy w czasie $O(k)$.

Złożoność pamięciowa rozwiązania dynamicznego

W warunkach zadania określono ograniczenie na pamięć w wysokości 64 MB. Rozwiązanie dynamiczne o złożoności czasowej $O(k^2 \cdot 2^k)$ wymaga zapamiętania $k \cdot 2^k$ rozwiązań — na

[2]Dowolny podzbiór $A \subseteq \{2,3,\ldots,k+1\}$ możemy reprezentować przy użyciu liczby naturalnej $m \in \{0,1,\ldots,2^k-1\}$ — jeśli i-ty bit liczby m jest ustawiony na 1, oznacza to, że do rozpatrywanego zbioru A należy wierzchołek o numerze $i+2$. Sprawdzanie przynależności elementu do zbioru można wówczas zrealizować za pomocą operacji bitowej AND oraz przesunięcia bitowego.

każde z nich potrzebne są 4 bajty. Oznacza to, że dla maksymalnego zestawu danych potrzeba około 80 MB pamięci ($20 \cdot 2^{20} \cdot 4 \approx 80\,000\,000$). Nie mieści się to w zadanych ograniczeniach, więc musimy zmodyfikować nasze rozwiązanie tak, by zmniejszyć zapotrzebowanie na pamięć. Zauważmy, że przechowywanie przez cały czas wszystkich podrozwiązań nie jest konieczne. W celu obliczenia rozwiązań częściowych dla wszystkich zbiorów o określonej mocy, wystarczy, że będziemy pamiętali rozwiązania częściowe dla zbiorów o mocy o jeden mniejszej. Tak więc na początku, zaczynając od zbioru pustego, generujemy rozwiązania dla zbiorów jednoelementowych, następnie możemy wygenerować rozwiązania dla zbiorów dwuelementowych itd. Najwięcej pamięci potrzebujemy w czasie przetwarzania zbiorów o mocach $k/2$ i $k/2+1$. Stąd zapotrzebowanie na pamięć w całym algorytmie można ograniczyć do $2k \cdot \binom{k}{k/2}$ (dla uproszczenia zakładamy, że $2 \mid k$), co jest znaczną poprawą w stosunku do początkowej wartości $k \cdot 2^k$ i mieści się w zadanych limitach. Podejście takie utrudnia nieco sposób przechowywania wyliczanych wyników — nie możemy sobie pozwolić na użycie tablicy wielkości $k \cdot 2^k$. Dlatego też konieczne jest wprowadzenie dodatkowej listy, do której będą wstawiane analizowane zbiory o kolejnych rosnących mocach. W celu szybkiego wyszukiwania zadanego podzbioru w liście, możemy zastosować tablicę o wielkości 2^k, w której będziemy przechowywali pozycje poszczególnych zbiorów na liście.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe, którego implementacja jest zawarta w plikach `atr.cpp` i `atr0.pas`, polega na wielokrotnym zastosowaniu algorytmu Dijkstry w fazie 1, a algorytmu dynamicznego w fazie 2. Jego złożoność czasowa to $O(k \cdot m \cdot \log n + k^2 \cdot 2^k)$.

Testy

Programy zawodników zostały przetestowane na następującym zestawie danych testowych (kolumny zawierają wartości n, m, k, g oraz ograniczenie na długość krawędzi w grafie):

Nazwa	n	m	k	g	ogr
atr1a.in	10	9	5	0	10
atr1b.in	10	30	5	3	10
atr1c.in	4	5	0	0	10
atr2a.in	100	99	9	30	100
atr2b.in	100	1000	9	15	100
atr3a.in	300	600	11	10	200
atr3b.in	300	5000	12	16	200
atr4a.in	500	499	15	105	500
atr4b.in	500	9000	16	90	500
atr5a.in	1000	3000	20	0	500

Nazwa	n	m	k	g	ogr
atr5b.in	1 000	5 000	20	40	500
atr6a.in	2 000	1 999	12	10	10
atr6b.in	2 000	5 000	12	20	10
atr7a.in	3 000	30 000	14	3	100
atr7b.in	7 000	120 000	14	2	100
atr8a.in	12 000	200 000	16	0	200
atr8b.in	5 000	4 999	16	10	200
atr9a.in	10 000	50 000	18	105	500
atr9b.in	10 000	2 000 000	18	50	500
atr10a.in	20 000	200 000	20	0	1 000
atr10b.in	20 000	5 000	20	50	1 000
atr11a.in	20 000	20 000	20	19	1 000
atr11b.in	20 000	170 000	20	2	1 000

Wszystkie testy poza *atr11a.in* są pseudolosowe, z podanymi wyżej parametrami. Test *atr11a.in* jest ścieżką, po której trzeba chodzić ciągle od jednego końca do drugiego; odpowiedź dla tego przypadku jest bardzo duża — wynosi około 500 000 000. „Złośliwy” jest także test *atr1c.in*, w którym $k = 0$.