Michał Włodarczyk
Treść zadania, Opracowanie

Mirosław Michalski
Program

Dostępna pamięć: 32 MB.

OI, Etap III, dzień próbny, 31.03.2009

Wiedźmak

Bajtazar został wiedźmakiem — pogromcą potworów. Przyszło mu wracać do swojego rodzimego Bajtogrodu przez krainę pełną potworów. Szczęśliwie, ludzie zamieszkujący tę krainę, walczący od pokoleń z potworami, opanowali sztukę wytwarzania specjalistycznych mieczy pomocnych w konfrontacji z nimi. W krainie, przez którą wędruje Bajtazar, znajduje się wiele miejscowości i łączących je dróg. Drogi nie krzyżują się poza miejscowościami, choć niektóre z nich prowadzą nie tylko po ziemi, ale i przez podziemia.

Bajtazar zebrał informacje o tej krainie (jako wiedźmak lubi wiedzieć różne rzeczy). Wie, jakiego rodzaju potwory można spotkać na każdej drodze i ile czasu potrzeba na jej przebycie. Wie też, w których miejscowościach można znaleźć kowali i na jakie rodzaje potworów potrafią oni wykuwać miecze. W tej chwili nie posiada żadnych mieczy i chce jak najszybciej dostać się do Bajtogrodu. Wstyd się przyznać, ale choć jest wiedźmakiem, to nie wie, jak to zrobić. Pomóż mu i znajdź taką trasę, która jak najszybciej doprowadzi go do Bajtogrodu i na której, zawsze gdy może spotkać potwory danego rodzaju, ma wcześniej możliwość zaopatrzenia się w miecz odpowiedni do walki z nimi. Bajtazar jest bardzo silny i może nosić ze sobą dowolnie wiele mieczy.

Wejście

Pierwszy wiersz standardowego wejścia zawiera cztery liczby całkowite: n, m, p i k ($1 \leqslant n \leqslant 200$, $0 \leqslant m \leqslant 3\,000$, $1 \leqslant p \leqslant 13$, $0 \leqslant k \leqslant n$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę miejscowości, liczbę łączących je dróg, liczbę gatunków potworów i liczbę kowali. Miejscowości są ponumerowane od 1 do n, przy czym Bajtogród ma numer n, a miejscowość, z której wyrusza Bajtazar — numer 1. Rodzaje potworów są ponumerowane od 1 do p.

W kolejnych k wierszach opisani są kowale, po jednym w każdym z tych wierszy. Wiersz $(i+1)$-szy zawiera liczby całkowite w_i, q_i, $r_{i,1} < r_{i,2} < \ldots < r_{i,q_i}$ ($1 \leqslant w_i \leqslant n$, $1 \leqslant q_i \leqslant p$, $1 \leqslant r_{i,j} \leqslant p$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: numer miejscowości, w której mieszka kowal, liczbę rodzajów potworów, przeciwko którym potrafi on wykuć miecze, oraz rodzaje tych potworów (w kolejności rosnącej). Kowale mogą mieszkać w tych samych miejscowościach.

W kolejnych m wierszach opisane są drogi. Wiersz $(k+i+1)$-szy zawiera liczby całkowite v_i, w_i, t_i, s_i, $u_{i,1} < u_{i,2} < \ldots < u_{i,s_i}$ ($1 \leqslant v_i < w_i \leqslant n$, $1 \leqslant t_i \leqslant 500$, $0 \leqslant s_i \leqslant p$, $1 \leqslant u_{i,j} \leqslant p$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: miejscowości, które łączy droga, czas potrzebny na przebycie drogi (jest on taki sam w każdym kierunku), liczbę rodzajów potworów, które można spotkać na danej drodze, oraz rodzaje tych potworów (w kolejności rosnącej). Żadne dwie drogi nie łączą tej samej pary miejscowości.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą — minimalny łączny czas potrzebny na dotarcie do Bajtogrodu. W przypadku, gdy dotarcie do Bajtogrodu jest niemożliwe, należy wypisać −1.

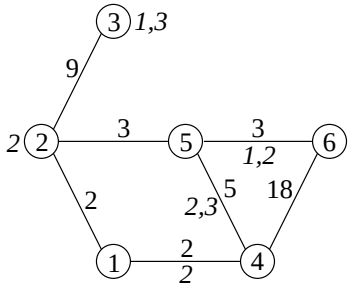
Przykład

Dla danych wejściowych:

```
6 7 4 2
2 1 2
3 2 1 3
1 2 2 0
2 3 9 0
1 4 2 1 2
2 5 3 0
4 5 5 2 2 3
4 6 18 0
5 6 3 2 1 2
```

poprawnym wynikiem jest:

```
24
```



Natomiast dla danych:

```
2 1 1 1
2 1 1
1 2 1 1 1
```

poprawnym wynikiem jest:

```
-1
```

Komentarz do przykładu

W pierwszym przykładzie na rysunku kółka reprezentują miejscowości, a liczby w ich wnętrzu to ich numery. Krawędzie reprezentują drogi, a liczby umieszczone nad nimi — czas potrzebny na ich przebycie. Liczby znajdujące się obok kółek (pisane kursywą) oznaczają rodzaje potworów, przeciwko którym kowal z danej miejscowości potrafi wykuć miecze. Liczby pod krawędziami (pisane kursywą) oznaczają rodzaje potworów, które można spotkać na danej drodze.

Wiedźmak Bajtazar powinien najpierw pójść do miejscowości nr 2, zdobyć miecz przeciwko potworom nr 2, wrócić do miejscowości nr 1, potem do miejscowości nr 4 i w końcu do Bajtogrodu.

W drugim przykładzie Bajtazar nie jest w stanie zdobyć miecza na potwora nr 1, więc nie może też dojść do Bajtogrodu.

Rozwiązanie

Analiza problemu

Jak każdy Czytelnik już z pewnością zauważył, nasze zadanie ma naturę grafową. Rozważmy graf nieskierowany G, w którym miejscowości są wierzchołkami, a krawędzie to drogi pomiędzy nimi, przy czym jako wagę krawędzi przyjmujemy czas potrzebny na przebycie odpowiadającej jej drogi. W takim grafie mamy znaleźć najkrótszą bezpieczną ścieżkę między pewnymi dwoma wierzchołkami. Ścieżka będzie *bezpieczna*, jeśli zawsze przed wejściem na drogę zbierzemy komplet mieczy pozwalający rozprawić się z występującymi na niej potworami.

Na początek spróbujmy uprościć, co tylko się da. Zauważmy, że:

- nie warto pamiętać, którzy kowale wykuwają które miecze — w zupełności wystarczy nam informacja, jakie rodzaje mieczy można zdobyć w poszczególnych miejscowościach;
- jeżeli Bajtazar znajduje się w jakiejś miejscowości, to na pewno dobrym pomysłem jest dla niego zebranie od razu wszystkich dostępnych w niej rodzajów mieczy;
- jako że wszystkie miecze przeznaczone do walki z danym rodzajem potworów są dla Bajtazara zupełnie jednakowe, w dowolnym momencie wędrówki liczy się dla niego tylko informacja, jakie rodzaje potworów może on pokonać za pomocą posiadanych mieczy.

Dalej, oznaczmy przez $M(v)$ zbiór mieczy dostępnych w miejscowości v, natomiast przez $P(e)$ — zbiór mieczy potrzebnych do bezpiecznego pokonania drogi e.

Rozwiązanie wzorcowe

Wejdźmy na chwilę w rolę Bajtazara — może pomoże nam to wymyślić dobrą strategię rozwiązania problemu. Załóżmy, że doszliśmy właśnie do miejscowości v i mamy w plecaku zbiór mieczy S. Zgodnie z dotychczasowymi spostrzeżeniami, w tym momencie opłaca nam się zebrać wszystkie miecze dostępne w v. Jeśli teraz e jest pewną drogą wychodzącą z v, to możemy nią wyruszyć, jeżeli

$$P(e) \subseteq S \cup M(v).$$

Oznaczmy przez $t(v,S)$ minimalny czas potrzebny, by znaleźć się w miejscowości v, posiadając zbiór mieczy S — łącznie z mieczami wykuwanymi w v. Jeżeli sytuacja określana przez parę (v,S) nie jest możliwa, to przyjmujemy, że $t(v,S) = \infty$. Oczywiście $t(1,\ M(1)) = 0$, bo na samym początku wędrówki Bajtazar może od razu zdobyć wszystkie miecze z miejscowości nr 1.

Zauważmy teraz, że jeśli $t(v,S) = c \neq \infty$ i w grafie G istnieje krawędź (v,w), to wiedźmakowi wystarczy informacja, czy $P\big((v,w)\big) \subseteq S$. Jeśli tak, to może przejść do miejscowości w i od razu zebrać wszystkie nowe miecze, jakie tam znajdzie. Niech c' oznacza czas potrzebny na przebycie drogi (v,w). Wtedy wiedźmak może znaleźć się w miejscowości

w z mieczami $S \cup M(w)$ w czasie $c + c'$. Nie wiemy jednak, czy jest to najszybszy sposób, więc otrzymujemy nierówność

$$t(w,\ S \cup M(w)) \leqslant c + c'.$$

Bardzo podobną nierówność otrzymalibyśmy, próbując wymyślić algorytm znajdowania najkrótszej ścieżki w grafie G. Okazuje się, że zbieżność ta jest nieprzypadkowa, tyle że nasza nierówność odpowiada szukaniu najkrótszej ścieżki w nieco innym grafie...

Parę $[v,S]$ nazwijmy *stanem*. Skonstruujmy teraz nowy graf H, którego wierzchołkami są stany, zaś zbiór krawędzi definiujemy jak wyżej. Formalnie,

$$\big([v,A],[w,B]\big) \in E(H) \quad \Longleftrightarrow \quad (v,w) \in E(G) \ \wedge \ P\big((v,w)\big) \subseteq A \ \wedge \ B = A \cup M(w), \quad (1)$$

przy czym $E(H)$ i $E(G)$ oznaczają zbiory krawędzi odpowiednio grafów H i G, natomiast wagi pozostają takie jak oryginalnych krawędzi. Zauważmy, że H jest grafem skierowanym. Zadanie sprowadza się teraz do znalezienia najkrótszej ścieżki w grafie H z wierzchołka $[1,M(1)]$ do dowolnego wierzchołka postaci $[n,X]$, bo nieważne jest, z jakimi mieczami dotrzemy do Bajtogrodu. Powszechnie znanym i efektywnym algorytmem rozwiązującym ten problem dla grafu o nieujemnych wagach jest algorytm Dijkstry. Czytelników niezaznajomionych z tym algorytmem odsyłamy do [18] czy [20], a praktycznie do dowolnej książki o algorytmach. Algorytm Dijkstry korzysta ze struktury danych zwanej kolejką priorytetową lub kopcem, której opis również można znaleźć w każdej z wymienionych książek.

Miecze i maski

Dotychczas pisaliśmy dużo o zbiorach i relacjach między nimi, nie zastanawiając się wcale nad implementacją rozwiązania na komputerze. Chcemy reprezentować zbiory w taki sposób, by można było efektywnie wykonywać na nich następujące operacje:

- badanie równości zbiorów,
- obliczanie sumy zbiorów,
- stwierdzanie, czy zbiór jest podzbiorem drugiego.

Okazuje się, że jeśli umiemy wykonać dwie pierwsze operacje, to umiemy też trzecią. Wynika to z poniższej tożsamości, której dowód pozostawiamy jako proste ćwiczenie:

$$A \subseteq B \Leftrightarrow A \cup B = B.$$

W tym miejscu w istotny sposób skorzystamy z ograniczenia $p \leqslant 13$. Otóż skojarzymy miecze z liczbami w systemie dwójkowym w następujący sposób:

$$\begin{aligned}
\text{pierwszy miecz} &\equiv [0000000000001]_2 &&= 1\\
\text{drugi miecz} &\equiv [0000000000010]_2 &&= 2\\
&\dots\\
\text{piąty miecz} &\equiv [0000000010000]_2 &&= 16\\
&\dots
\end{aligned}$$

Formalnie, i-temu mieczowi przyporządkowujemy liczbę 2^{i-1}. Zbiór traktujemy jako sumę reprezentacji jego elementów. Na przykład, zbiór złożony z mieczy $1, 2, 5$ to $[0000000010011]_2 = 19$. Jak widać, reprezentacja zbioru o zadanych elementach jest jednoznaczna, dzięki czemu możemy zbiory bez problemu porównywać. Taka reprezentacja nazywana jest *maską bitową* zbioru. Jak łatwo zauważyć, maska bitowa żadnego zbioru mieczy nie przekroczy

$$\sum_{k=0}^{p-1} 2^k = 2^p - 1 \leqslant 8\,191, \tag{2}$$

co umożliwia wykorzystanie do ich przechowywania (z solidnym zapasem) chociażby liczb całkowitych 32-bitowych. Dzięki temu do obliczania sumy zbiorów można użyć operacji sumy logicznej na bitach, w Pascalu zapisywanej jako **or**, a w C/C++ jako „|”. Oznaczenia te nie są przypadkowe — suma na pojedynczych bitach to nic innego jak alternatywa logiczna. Przykładowo:

$$19 \mid 48 = [0000000010011]_2 \mid [0000000110000]_2 = [0000000110011]_2 = 51.$$

Ostatnie szlify

Jedyną rzeczą dzielącą nas od rozwiązania wzorcowego jest konstrukcja grafu H. Ustaliliśmy już, że zbiory będziemy reprezentować jako ich maski bitowe. Na mocy (2), liczba wierzchołków wyniesie wówczas $2^p n \approx 1{,}6 \cdot 10^6$, co nie sprawi nam większego kłopotu. Jednak liczba krawędzi może osiągnąć $2^p m \approx 2{,}5 \cdot 10^7$, a taka ich ilość prawie na pewno nie zmieści się w skromnym limicie pamięciowym wynoszącym 32 MB. Aby temu zaradzić, możemy konstruować krawędzie dynamicznie w oparciu o krawędzie G i zależność (1). Oto pseudokod rozwiązania (przyjmujemy, że na początku $t[v][S] = \infty$ dla wszystkich v, S):

```
 1: t[1][M(1)] := 0;
 2: queue.push((1, M(1)));
 3: while not queue.empty() do
 4: begin
 5:   (v,S) := queue.top();
 6:   queue.pop();
 7:   for w : (v,w) ∈ E(G) do
 8:     if P((v,w)) | S = S then
 9:       if t[v][S] + time((v,w)) < t[w][S | M(w)] then
10:       begin
11:         t[w][S | M(w)] := t[v][S] + time((v,w));
12:         queue.update((w, S | M(w)));
13:       end
14: end
```

Zakładamy, że operacja $queue.top()$ zwraca taką parę (v,S) znajdującą się w kolejce priorytetowej, dla której wartość $t[v][S]$ jest minimalna, $queue.pop()$ usuwa taką parę z kolejki, zaś $queue.update(x)$ wstawia element x do kolejki lub aktualizuje jego położenie w kolejce, jeśli x już się tam znajduje. Algorytm Dijkstry z użyciem kolejki priorytetowej działa w czasie $O(|E| \log |V|)$, co w naszym przypadku daje złożoność $O(2^p m \cdot p \log n)$.

Jest to złożoność pesymistyczna i w przypadku losowych testów omijamy większość wierzchołków, więc program działa szybciej, niż można by się spodziewać z analizy złożoności czasowej. Natomiast złożoność pamięciowa jest rzędu $O(2^p n)$, ponieważ dominującym czynnikiem jest rozmiar tablicy t. Rozwiązanie to zostało zaimplementowane w plikach `wie.cpp` i `wie3.cpp`.

Inne rozwiązania

Jak zrobiłby to wiedźmak i dlaczego zrobiłby to źle

Znając wiedźmakowe podejście, wybierałby on pierwszą drogę, którą może przejść, i szedłby tak naprzód, aż napotka przeszkodę. Wtedy zawróciłby w poszukiwaniu odpowiedniego miecza i powtarzałby całą tę procedurę do skutku. Innymi słowy, próbowałby rozwiązać problem metodą prób i błędów. Nadzieje na skonstruowanie w ten sposób najkrótszej ścieżki są jednak bardzo mizerne — takiemu wiedźmakowi cała wędrówka zapewne zajęłaby z 7 tomów powieści.

Zniecierpliwiony wiedźmak zapewne szukałby najkrótszej ścieżki do Bajtogrodu w grafie G, ignorując potwory i licząc na łut szczęścia. Oczywiste jest, że takie podejście, choć efektowne, praktycznie nie ma szans zadziałać. Amatorzy mocnych wrażeń znajdą je w pliku `wieb1.cpp`.

Zapominalski wiedźmak zapomniałby kupić wszystkie dostępne miecze przed wyruszeniem w podróż i rozpoczęłby wyszukiwanie z wierzchołka $[1,0]$, a nie $[1,M(1)]$. Błąd ten popełnić bardzo łatwo, a takie rozwiązanie zdobywało jedynie 40 punktów. Można je znaleźć w pliku `wieb2.cpp`.

Rozwiązania wolniejsze

Wszystkie przygotowane przez jurorów rozwiązania wolniejsze opierają się w mniejszym lub większym stopniu na schemacie wykorzystanym w rozwiązaniu wzorcowym. Ze względu na specyficzną strukturę grafu H, część z nich, mimo teoretycznie gorszej złożoności, uzyskiwała maksymalną punktację (rozwiązania `wie1.cpp` i `wie2.pas`, korzystające ze zwykłej kolejki z wielokrotnym wstawianiem elementów zamiast kopca), a część nieco niższą (rozwiązanie `wies1.cpp`, w którym dla rozważanego wierzchołka za każdym razem zostają przetworzone wszystkie maski bitowe — 60 punktów — i rozwiązanie `wies2.cpp`, implementujące siłową wersję algorytmu Dijkstry niewykorzystującą kopca — 80 punktów).

Testy

Do oceny rozwiązań przygotowano 10 zestawów testów. Pierwsze trzy zestawy zawierają nieduże testy poprawnościowe wygenerowane ręcznie (w każdym z tych zestawów zadbano o to, żeby w miejscowości numer 1 występowały jakieś potrzebne miecze). Wszystkie pozostałe zestawy składają się wyłącznie z testów losowych: w każdym z nich obierano losowy graf, po czym w części testów wybierano pewną ścieżkę o zadanej długości i rozmieszczano miecze w grafie w taki sposób, aby ta właśnie ścieżka była najkrótszą ścieżką dla Bajtazara, natomiast w pozostałych testach miecze rozmieszczano w grafie zupełnie losowo.

Oznaczenia w poniższej tabelce są zgodne z tymi w treści zadania, tzn. reprezentują odpowiednio: liczbę miejscowości, liczbę dróg, liczbę rodzajów potworów (a zarazem mieczy) oraz liczbę kowali. Niniejsze zadanie nie sprawiło zawodnikom większych problemów i stanowiło rozgrzewkę przed właściwymi zawodami III stopnia.

Nazwa	n	m	p	k	Opis
wie1a.in	1	0	4	1	mały test poprawnościowy
wie1b.in	4	4	4	4	mały test poprawnościowy
wie1c.in	1	0	1	0	mały test poprawnościowy
wie1d.in	2	1	13	2	mały test poprawnościowy
wie2.in	12	14	4	8	mały test poprawnościowy
wie3.in	70	115	7	9	średni test poprawnościowy
wie4.in	101	166	10	10	test losowy
wie5.in	140	573	10	20	test losowy
wie6.in	150	684	11	40	test losowy
wie7a.in	150	2624	12	9	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie7b.in	150	2624	12	9	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie8a.in	180	2718	13	30	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie8b.in	180	2705	13	30	test losowy
wie9a.in	200	778	13	60	test losowy
wie9b.in	200	2796	13	60	test losowy
wie10a.in	200	2779	13	30	test losowy
wie10b.in	200	2793	13	200	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie10c.in	200	2778	13	200	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie10d.in	200	2781	13	1	test losowy