Marcin Pilipczuk
Treść zadania, Opracowanie

Marcin Andrychowicz, Adam Karczmarz
Program

Dostępna pamięć: 256 MB.

OI, etap III, dzień pierwszy, 2.04.2014

Turystyka

Król Bajtazar jest przekonany, że pełna turystycznych atrakcji Bajtocja powinna przyciągać rzesze turystów, a ci powinni wydawać pieniądze, przyczyniając się do wypełnienia królewskiego skarbca. Tak się jednak nie dzieje. Król polecił swojemu doradcy przyjrzeć się bliżej sytuacji. Ten odkrył, że przyczyną małej popularności królestwa wśród obcokrajowców jest niedostatecznie rozwinięta sieć dróg.

Nadmieńmy, że w Bajtocji jest n miast i m dwukierunkowych dróg, z których każda łączy dwa różne miasta. Drogi mogą prowadzić tunelami i estakadami. Nie ma gwarancji, że z każdego miasta da się dojechać do każdego innego.

Doradca zauważył, że obecna sieć dróg nie pozwala na zorganizowanie żadnej długiej wycieczki. Zaczynając wycieczkę w dowolnym z miast i poruszając się drogami, nie jesteśmy w stanie odwiedzić więcej niż 10 miast bez przejeżdżania przez to samo miasto dwukrotnie.

Z braku funduszy, zamiast budować nowe drogi, Bajtazar postanowił zbudować w Bajtocji sieć punktów informacji turystycznej (PIT), w których odpowiednio przeszkoleni pracownicy będą reklamować zalety krótkich wycieczek. Dla każdego miasta, PIT powinien znajdować się albo w tym mieście, albo w którymś z miast bezpośrednio połączonych z nim drogą. Dla każdego miasta znany jest koszt wybudowania PIT-u w tym mieście. Pomóż Bajtazarowi znaleźć najtańszy sposób zbudowania sieci PIT-ów.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n, m ($2 \leqslant n \leqslant 20\,000$, $0 \leqslant m \leqslant 25\,000$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę miast i dróg w Bajtocji. Miasta są ponumerowane liczbami od 1 do n. Drugi wiersz wejścia zawiera n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($0 \leqslant c_i \leqslant 10\,000$) pooddzielanych pojedynczymi odstępami; liczba c_i oznacza koszt zbudowania PIT-u w mieście o numerze i.

Dalej następuje opis dróg w Bajtocji. W i-tym z kolejnych m wierszy znajdują się dwie liczby całkowite a_i, b_i ($1 \leqslant a_i < b_i \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że miasta o numerach a_i i b_i są połączone drogą. Pomiędzy każdą parą miast istnieje co najwyżej jedna droga.

W testach wartych łącznie 20 % punktów zachodzi dodatkowy warunek $n \leqslant 20$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą łączny koszt budowy wszystkich PIT-ów.

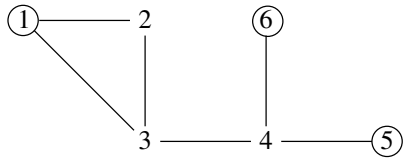
Przykład

Dla danych wejściowych:

```
6 6
```

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

poprawnym wynikiem jest:
7



Wyjaśnienie do przykładu: *Aby uzyskać minimalny koszt budowy, PIT-y powinny zostać zbudowane w miastach o numerach 1, 5 i 6 (koszt wyniesie 3 + 2 + 2 = 7).*

Testy „ocen":

`1ocen`: *$n = 10$, $m = 9$, sieć dróg tworzy ścieżkę długości 10;*

`2ocen`: *$n = 10$, $m = 45$, między każdą parą miast jest droga;*

`3ocen`: *$n = 20\,000$, $m = 19\,998$, ze wszystkich miast z wyjątkiem 1 i 2 wychodzi dokładnie jedna droga – do miasta 1 lub 2.*

Rozwiązanie

Na początek przetłumaczmy treść zadania na język teorii grafów. Mamy dany na wejściu nieskierowany graf $G = (V, E)$ o n wierzchołkach (reprezentujących miasta) i m krawędziach (reprezentujących drogi), wraz z wagami $c : V \to \mathbb{Z}_{\geqslant 0}$. Naszym celem jest znalezienie podzbioru $X \subseteq V$ o minimalnym możliwym koszcie tak, by każdy wierzchołek $v \in V$ należał do X lub miał sąsiada należącego do X. Zbiór X będzie oczywiście zawierał wierzchołki reprezentujące miasta, w których zbudowane zostaną punkty informacji turystycznej.

W języku teorii grafów mówimy, że taki zbiór X jest *zbiorem dominującym*: wierzchołek v *dominuje* wierzchołek w jeśli $v = w$ lub $vw \in E$, zbiór X dominuje zbiór Y jeśli dla każdego $y \in Y$ istnieje $x \in X$ dominujący y, a zbiór dominujący to taki podzbiór V, który dominuje wszystkie wierzchołki grafu. Podsumowując, naszym zadaniem jest znalezienie najtańszego (względem wag c) zbioru dominującego w danym grafie.

Zauważmy, że naiwne rozwiązanie, sprawdzające wszystkie możliwe podzbiory $X \subseteq V$, działa w czasie $O(2^n(n + m))$: sprawdzenie, czy dany zbiór X jest zbiorem dominującym, i wyznaczenie jego kosztu można łatwo wykonać w czasie $O(n+m)$. To rozwiązanie, zaimplementowane w plikach `turs3.cpp` i `turs6.pas`, będzie skuteczne dla $n \leqslant 20$, otrzymując około 20% punktów, lecz na pewno nie poradzi sobie dla większych grafów. Co więcej, problem znalezienia najtańszego zbioru dominującego

jest problemem bardzo trudnym i jedyne znane algorytmy rozwiązujące go działają w czasie wykładniczym od liczby wierzchołków w grafie ($O(\gamma^n)$ dla pewnej stałej $1 < \gamma < 2$). Jest to szybciej niż algorytm naiwny, ale wciąż niewystarczająco.

Pozostaje nam wykorzystać ostatnią informację zawartą w treści zadania, którą być może część czytelników uznała tylko za tło fabularne: w Bajtocji nie można odwiedzić więcej niż $t = 10$ miast, nie przejeżdżając przez jedno miasto dwukrotnie. W języku teorii grafów oznacza to, że w grafie G nie ma ścieżki prostej (takiej, na której nie powtarzają się wierzchołki) dłuższej niż $t = 10$.[1] Zachęcam do próby narysowania sobie przykładu „skomplikowanego" grafu, który nie ma długiej ścieżki – jest to dość trudne. Pozwala to mieć nadzieję, że ta własność grafu G powoduje, że musi mieć on określoną strukturę, przez co problem znajdowania najtańszego zbioru dominującego staje się istotnie prostszy.

Załóżmy, że dany graf G jest spójny – w przeciwnym razie, możemy osobno rozważyć każdą spójną składową G. Weźmy dowolny wierzchołek $r \in V$ i przeszukajmy graf w głąb (DFS), zaczynając z wierzchołka r. Niech T będzie drzewem rozpinającym grafu G, wygenerowanym przez to przeszukiwanie (tzn. drzewo T jest ukorzenione w wierzchołku r i dla każdego wierzchołka $v \in V$ kolejne dzieci v w drzewie T odpowiadają kolejnym rekurencyjnym wywołaniom procedury przeszukiwania grafu w głąb, wywoływanym w czasie rozpatrywania wierzchołka v). Następująca obserwacja okazuje się kluczowa:

Obserwacja 1. Głębokość drzewa T (liczba wierzchołków na najdłuższej ścieżce od liścia do korzenia r) jest **nie większa niż** t.

Dowód: Ścieżka od liścia do korzenia w drzewie T jest również ścieżką prostą w grafie G. ■

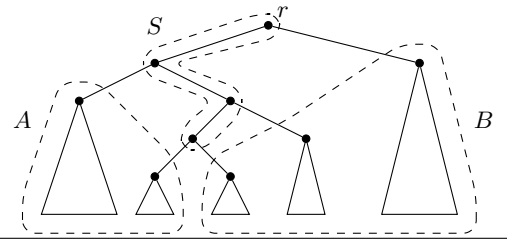
Zastanówmy się teraz, jak wykorzystać to spostrzeżenie. Pomyślmy o przeszukiwaniu grafu w głąb jako o procedurze, która, rozpatrując wierzchołek v, posiada pewien zbiór wierzchołków S na stosie (są to przodkowie wierzchołka v w drzewie T, wraz z wierzchołkiem v), odwiedziła już całkowicie pewne wierzchołki (oznaczmy ich zbiór przez A) i jeszcze nie odwiedziła wierzchołków $B = V \setminus (A \cup S)$. Warto myśleć o zbiorze A jako o zbiorze wierzchołków *na lewo* od ścieżki S w drzewie T, a o zbiorze B jako o zbiorze wierzchołków *na prawo* od tej ścieżki; przykładową konfigurację zbiorów S, A i B przedstawia rysunek 1. Poczyńmy następującą obserwację:

Obserwacja 2. Nie ma żadnej krawędzi między zbiorem A a zbiorem B.

Dowód: Gdyby taka krawędź ab istniała dla pewnych wierzchołków $a \in A$, $b \in B$, to algorytm przeszukiwania w głąb poszedłby tą krawędzią i odwiedziłby wierzchołek b, zanim by całkowicie opuścił wierzchołek a. ■

Z obserwacji 1 wynika, że $|S| \leqslant t$, gdyż zbiór S odpowiada ścieżce od wierzchołka v do korzenia r w drzewie T. Inaczej mówiąc, zbiór S jest małym *separatorem*, którego usunięcie, zgodnie z obserwacją 2, dzieli graf na dwie niepołączone części A i B. Co więcej, *każda* ścieżka z wierzchołka do korzenia w drzewie T jest takim separatorem.

[1] Dla zachowania ogólności rozumowania, w dalszej części będziemy używać litery t na oznaczenie ograniczenia na długość ścieżki w grafie G i na koniec „przypomnimy sobie", że $t = 10$.



Rys. 1: Przykładowy zbiór S, dzielący graf na część już odwiedzoną A i część jeszcze nie odwiedzoną B. Po lewej: zgodna równoodległa trójka. Po prawej: niezgodna równoodległa trójka.

Ideą rozwiązania jest, by wykorzystać te separatory do zaprojektowania algorytmu opartego o programowanie dynamiczne, niejako „zamiatając" drzewo T od lewej do prawej.

Programowanie dynamiczne

Przypomnijmy, że mamy ustalone drzewo T wyznaczone przez algorytm przeszukiwania grafu w głąb. Stan algorytmu przeszukiwania grafu w głąb po każdym jego kroku (tj. po nowym wywołaniu rekurencyjnym lub po zakończeniu aktualnego wywołania) odpowiada pewnej trójce zbiorów (S, A, B), zdefiniowanej wcześniej. Niech $(S_1, A_1, B_1), (S_2, A_2, B_2), \ldots, (S_p, A_p, B_p)$ będzie ciągiem kolejnych takich trójek, które wystąpiły w czasie przeszukiwania w głąb. Zauważmy, że $(S_1, A_1, B_1) = (\{r\}, \emptyset, V \setminus \{r\})$ to początkowa trójka, powstała po pierwszym wywołaniu procedury przeszukiwania w głąb dla wierzchołka r, zaś $(S_p, A_p, B_p) = (\emptyset, V, \emptyset)$ to ostatnia trójka, powstała po zakończeniu wywołania rekurencyjnego dla wierzchołka r. Co więcej, $p = 2n$, gdyż dla każdego wierzchołka $v \in V$, powstają dwie trójki (S_i, A_i, B_i) – pierwsza gdy przeszukiwanie w głąb wchodzi do tego wierzchołka, a druga jak z niego wychodzi. Do powyższych trójek dołóżmy jeszcze trójkę $(S_0, A_0, B_0) = (\emptyset, \emptyset, V)$, odpowiadającą stanowi sprzed pierwszego wywołania procedury przeszukiwania w głąb. Będziemy rozpatrywać każdą trójkę (S_i, A_i, B_i) po kolei i znajdywać pewną liczbę kandydatów na częściowe rozwiązania $Y \subseteq S_i \cup A_i$.

Przyjrzyjmy się jednej trójce (S_i, A_i, B_i). Z naszych dwóch obserwacji z początku rozdziału wynika, że mamy $|S_i| \leqslant t$ oraz nie ma żadnej krawędzi łączącej A_i i B_i. Weźmy jakiegoś „dobrego kandydata" na część rozwiązania $Y \subseteq S_i \cup A_i$, do którego będziemy później dobierać wierzchołki ze zbioru B_i. Jakie własności powinien mieć taki zbiór Y i co powinniśmy o nim pamiętać?

Po pierwsze, zauważmy, że żaden wierzchołek ze zbioru B_i nie dominuje żadnego wierzchołka ze zbioru A_i. Tak więc, nasi kandydaci Y na część zbioru dominującego muszą dominować cały zbiór A_i. Po drugie, ważne jest dla nas zachowanie zbioru Y na zbiorze S_i:

1. Pełen zbiór dominujący musi dominować zbiór B_i. Część elementów B_i może być dominowana przez wierzchołki z S_i (ale nie z A_i). Zatem istotne jest dla nas przecięcie $S_i \cap Y$, gdyż mówi nam ono, co w zbiorze B_i mamy już zdominowane.

2. Musimy też zdominować zbiór S_i. Część wierzchołków z S_i może być zdominowana przez wierzchołki z B_i, tak więc nie możemy wymagać, by Y dominował też cały zbiór S_i, ale musimy zapamiętać, które wierzchołki pozostały do zdominowania.

Powyższe rozważania prowadzą nas do następującej definicji. Wprowadźmy symbol $\mathtt{T}$, oznaczający należenie do zbioru Y, symbol $\mathtt{N}_1$, oznaczający nienależenie, ale bycie już zdominowanym, oraz symbol $\mathtt{N}_0$, oznaczający zarówno nienależenie do Y, jak i nie bycie zdominowanym przez Y.

Definicja 1. Dla ustalonej trójki (S_i, A_i, B_i) powiemy, że zbiór $Y \subseteq S_i \cup A_i$ jest *częściowym rozwiązaniem*, jeśli Y dominuje cały zbiór A_i. Co więcej, jeśli dodatkowo mamy daną funkcję $\sigma : S_i \to \{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, to powiemy, że częściowe rozwiązanie Y *ma interfejs* σ, jeśli zachodzą następujące warunki dla każdego $v \in S_i$:

- $\sigma(v) = \mathtt{T}$ wtedy i tylko wtedy gdy $v \in Y$,
- $\sigma(v) = \mathtt{N}_1$ wtedy i tylko wtedy gdy $v \notin Y$, ale v jest zdominowany przez Y, oraz
- $\sigma(v) = \mathtt{N}_0$ wtedy i tylko wtedy gdy $v \notin Y$ i v nie jest zdominowany przez Y.

Nasze dotychczasowe rozważania można podsumować następującym lematem:

Lemat 1. Ustalmy $0 \leqslant i \leqslant p$. Jeśli X jest zbiorem dominującym w grafie G, to zbiór $Y = X \cap (S_i \cup A_i)$ jest częściowym rozwiązaniem. Co więcej, jeśli X jest najtańszym zbiorem dominującym w G, a Y ma interfejs σ, to Y jest najtańszym częściowym rozwiązaniem o interfejsie σ.

Dowód: Pierwsza część lematu wynika wprost z obserwacji 2: elementy zbioru A_i mogą być zdominowane tylko przez elementy $S_i \cup A_i$. By udowodnić drugą część lematu, załóżmy nie wprost, że $Y' \subseteq S_i \cup A_i$ jest częściowym rozwiązaniem o interfejsie σ i o koszcie mniejszym niż koszt rozwiązania Y. Rozważmy zbiór $X' = (X \setminus Y) \cup Y' = (X \cap B_i) \cup Y'$. Skoro Y' ma mniejszy koszt niż Y, to X' jest tańszy od X. By otrzymać sprzeczność, pokażemy, że X' jest też zbiorem dominującym w grafie G. Rozważmy dowolny wierzchołek $v \in V$ i niech $w \in X$ będzie dowolnym wierzchołkiem dominującym wierzchołek v w rozwiązaniu X. Rozważmy następujące przypadki:

1. Jeśli $v \in A_i$, to v jest zdominowany przez Y', gdyż Y' jest częściowym rozwiązaniem.
2. Jeśli $w \in A_i$, lecz $v \notin A_i$, to mamy $v \in S_i$ oraz $\sigma(v) \neq \mathtt{N}_0$. Wówczas Y' dominuje v, gdyż ma również interfejs σ.
3. Jeśli $w \in S_i$, to $w \in S_i \cap X = S_i \cap Y = S_i \cap Y' \subseteq X'$, gdyż Y' też ma interfejs σ.
4. Jeśli $w \in B_i$, to $w \in B_i \cap X = B_i \cap X' \subseteq X'$.

W każdym przypadku otrzymujemy, że v jest zdominowany przez X'. Z dowolności wyboru v wynika, że X' jest zbiorem dominującym w G. ■

Definicja tabeli i obliczenia

W naszym programowaniu dynamicznym dla każdej trójki (S_i, A_i, B_i) oraz interfejsu $\sigma : S_i \to \{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, obliczamy wartość $M[i, \sigma]$ będącą minimalnym możliwym kosztem częściowego rozwiązania $Y \subseteq S_i \cup A_i$ o interfejsie σ. Dla $i = 0$, gdy $(S_0, A_0, B_0) = (\emptyset, \emptyset, V)$, początkową wartością jest $M[0, \emptyset] = 0$: jedynym częściowym rozwiązaniem jest zbiór pusty[2]. Wartość najtańszego zbioru dominującego w grafie G odczytamy z pola $M[p, \emptyset]$: stan $(S_p, A_p, B_p) = (\emptyset, V, \emptyset)$ ma tylko jeden możliwy interfejs $\sigma = \emptyset$ i częściowe rozwiązania dla tej trójki to dokładnie zbiory dominujące w grafie G.

Przeanalizujmy teraz, jak obliczyć pojedynczą wartość $M[i+1, \sigma]$, mając dane wszystkie wartości $M[j, \sigma']$ dla $0 \leqslant j \leqslant i$. Po pierwsze, zauważmy, że nie wszystkie interfejsy σ mają sens: jeśli dwa wierzchołki $u, v \in S_{i+1}$ są połączone krawędzią w grafie G, ale $\sigma(u) = \mathtt{T}$ oraz $\sigma(v) = \mathtt{N}_0$, to wówczas nie ma żadnego częściowego rozwiązania o tym interfejsie, i w tym przypadku $M[i+1, \sigma] = +\infty$. W przeciwnym przypadku, mamy dwa podprzypadki: stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał albo w wyniku nowego wywołania rekurencyjnego procedury przeszukiwania w głąb, albo w wyniku zakończenia aktualnego wywołania.

Nowe wywołanie. Załóżmy, że stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał w wyniku wywołania procedury przeszukiwania rekurencyjnego na wierzchołku v. Zauważmy, że wówczas $S_{i+1} = S_i \cup \{v\}$, $A_{i+1} = A_i$ oraz $B_{i+1} = B_i \setminus \{v\}$. Niech Y będzie dowolnym częściowym rozwiązaniem o interfejsie σ. Rozważmy trzy przypadki, zależnie od wartości $\sigma(v)$.

Załóżmy wpierw, że $\sigma(v) = \mathtt{T}$, czyli $v \in Y$, i zastanówmy się, jaki interfejs σ' ma zbiór $Y' = Y \setminus \{v\}$ względem trójki (S_i, A_i, B_i). Oczywiście, dla każdego $w \in S_i$ mamy $\sigma'(w) = \mathtt{T}$ wtedy i tylko wtedy gdy $w \in Y$, co jest równoważne $\sigma(w) = \mathtt{T}$. Podobnie, jeśli $\sigma(w) = \mathtt{N}_0$, czyli Y nie dominuje w, to również Y' nie dominuje w i $\sigma'(w) = \mathtt{N}_0$. Jeśli zaś $\sigma(w) = \mathtt{N}_1$, to albo Y' dominuje w, albo Y' nie dominuje w, lecz v dominuje w ($vw \in E$). Tak więc, $\sigma'(w) = \mathtt{N}_1$ lub $\sigma'(w) = \mathtt{N}_0$, ale ten drugi przypadek może zajść tylko wtedy, gdy $vw \in E$. Oznaczmy więc $\sigma' \prec \sigma$, jeśli dla każdego $w \in S_i$ mamy:

1. jeśli $\sigma(w) = \mathtt{T}$, to $\sigma'(w) = \mathtt{T}$;
2. jeśli $\sigma(w) = \mathtt{N}_0$, to $\sigma'(w) = \mathtt{N}_0$;
3. jeśli $\sigma(w) = \mathtt{N}_1$, to $\sigma'(w) = \mathtt{N}_1$ lub $\sigma'(w) = \mathtt{N}_0$ oraz $vw \in E$.

Wówczas z powyższych rozważań wynika, że:

$$M[i+1, \sigma] = c(v) + \min_{\sigma' \prec \sigma} M[i, \sigma'].$$

Rozważmy teraz przypadek $\sigma(v) = \mathtt{N}_1$. Przez $\sigma|_{S_i}$ oznaczmy funkcję σ ograniczoną do dziedziny S_i. Zauważmy, że v nie ma żadnych sąsiadów w zbiorze A_i, tak więc Y musi dominować v jakimś wierzchołkiem w $S_i = S_{i+1} \setminus \{v\}$. Jeśli istnieje $w \in S_i$ taki, że $vw \in E$ oraz $\sigma(w) = \mathtt{T}$, to każde częściowe rozwiązanie dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ będzie dominowało v, gdyż będzie zawierało wierzchołek w. Z drugiej

[2] Zastosowaliśmy tutaj konwencję, że $\emptyset$ oznacza też pustą funkcję.

strony, jeśli taki wierzchołek w nie istnieje, to żadne częściowe rozwiązanie dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ nie będzie dominowało v, co oznacza, że nie istnieje żadne częściowe rozwiązanie dla trójki $(S_{i+1}, A_{i+1}, B_{i+1})$ o interfejsie σ.

Mamy więc, że w pierwszym przypadku $M[i+1, \sigma] = M[i, \sigma|_{S_i}]$, zaś w drugim przypadku $M[i+1, \sigma] = +\infty$.

Pozostał nam najprostszy przypadek $\sigma(v) = \mathtt{N}_0$. Przypomnijmy, że rozpatrujemy tylko takie interfejsy σ, dla których żadne dwa sąsiadujące wierzchołki w S_{i+1} nie mogą mieć wartości $\mathtt{T}$ i $\mathtt{N}_0$. Razem z faktem, że v nie ma sąsiadów w A_i, oznacza to, że rodziny częściowych rozwiązań dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ i dla trójki $(S_{i+1}, A_{i+1}, B_{i+1})$ o interfejsie σ są dokładnie takie same. Mamy więc $M[i+1, \sigma] = M[i, \sigma|_{S_i}]$.

Zakończenie wywołania. Załóżmy, że stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał w wyniku zakończenia procedury przeszukiwania rekurencyjnego na wierzchołku v. Zauważmy, że wówczas $S_{i+1} = S_i \setminus \{v\}$, $A_{i+1} = A_i \cup \{v\}$ oraz $B_{i+1} = B_i$. Niech Y będzie dowolnym częściowym rozwiązaniem o interfejsie σ.

Jeśli $v \in Y$, to Y jest częściowym rozwiązaniem zgodnym z interfejsem $\sigma_{v \mapsto \mathtt{T}}$ względem trójki (S_i, A_i, B_i), gdzie $\sigma_{v \mapsto \mathtt{T}}$ jest interfejsem σ rozszerzonym o przyporządkowanie $v \mapsto \mathtt{T}$. Koszt najtańszego takiego częściowego rozwiązania przechowywany jest w komórce $M[i, \sigma_{v \mapsto \mathtt{T}}]$.

Jeśli $v \notin Y$, to z faktu, że Y jest częściowym rozwiązaniem, wynika, że Y dominuje v i Y jest częściowym rozwiązaniem zgodnym z interfejsem $\sigma_{v \mapsto \mathtt{N}_1}$ względem (S_i, A_i, B_i). Koszt najtańszego takiego częściowego rozwiązania przechowywany jest w komórce $M[i, \sigma_{v \mapsto \mathtt{N}_1}]$. Zatem otrzymujemy następującą zależność:

$$M[i+1, \sigma] = \min\left(M[i, \sigma_{v \mapsto \mathtt{T}}], M[i, \sigma_{v \mapsto \mathtt{N}_1}]\right).$$

Analiza złożoności

Tabela M ma $O(3^t n)$ komórek, gdyż indeks i przebiega zakres $0 \leqslant i \leqslant p = 2n$, a $|S_i| \leqslant t$ dla każdego i. Taka też jest złożoność pamięciowa powyższego algorytmu. Zastanówmy się teraz nad złożonością czasową.

Obliczenie nowej wartości $M[i+1, \sigma]$ prawie zawsze wymaga sięgnięcia tylko do kilku poprzednich wartości oraz obejrzenia krawędzi incydentnych z „kluczowym" wierzchołkiem v. Wydawać się może, że więcej czasu wymaga obliczenie $M[i+1, \sigma]$ w przypadku, gdy v jest nowym wierzchołkiem ($S_{i+1} = S_i \cup \{v\}$) i $\sigma(v) = \mathtt{T}$, gdyż wtedy musimy przejrzeć wszystkie interfejsy $\sigma' \prec \sigma$. Zauważmy jednak, że dla każdego interfejsu σ' względem (S_i, A_i, B_i), istnieje co najwyżej jeden interfejs σ, dla którego zachodzi $\sigma' \prec \sigma$: by otrzymać σ z σ', należy położyć $\sigma(v) = \mathtt{T}$ oraz zamienić wszystkie wartości $\mathtt{N}_0$ na $\mathtt{N}_1$ wśród sąsiadów wierzchołka v w zbiorze S_i. Tak więc, przy uważnej implementacji, powyższy algorytm programowania dynamicznego można wykonać w czasie $O(3^t(n+m))$; to rozwiązanie zostało zaimplementowane w plikach `turs1.cpp` i `turs4.pas`.

Możliwe jest jednak pewne dodatkowe usprawnienie. Przypomnijmy, że możemy pomijać obliczenia dla interfejsów σ, w których istnieją dwa wierzchołki u, v połączone krawędzią, dla których $\sigma(u) = \mathtt{T}$ i $\sigma(v) = \mathtt{N}_0$. Co więcej, przypomnijmy, że każdy

zbiór S_i odpowiada ścieżce w drzewie T, a więc i w grafie G. Okazuje się, że te dwie obserwacje istotnie zmniejszają liczbę interfejsów σ, które musimy rozważać.

Dla ustalonej trójki (S_i, A_i, B_i), zakodujmy interfejs σ jako ciąg $|S_i|$ symboli nad alfabetem $\{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, oznaczających wartości $\sigma(v)$ dla kolejnych wierzchołków na ścieżce S_i. W tym ciągu, symbole $\mathtt{T}$ i $\mathtt{N}_0$ nie mogą wystąpić obok siebie. To sprowadza nas do następującego pytania dotyczącego kombinatoryki na słowach:

> Ile jest t-literowych słów nad alfabetem $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ takich, że litery $\mathtt{b}$ i $\mathtt{c}$ nie występują obok siebie?

Nazwijmy powyższe słowa *dobrymi* i niech $F(t)$ oznacza liczbę dobrych słów o t literach. W dobrym słowie, dla każdych dwóch kolejnych pozycji, mamy tylko 7 możliwości wyboru liter na tych pozycjach: kombinacje $\mathtt{bc}$ i $\mathtt{cb}$ są zabronione. Łącząc litery kolejno w pary, otrzymujemy następującą obserwację:

Obserwacja 3.

$$F(2t) \leqslant 7^t < 2{,}65^{2t}.$$

Oznacza to, że dobrych słów jest istotnie mniej niż wszystkich słów o określonej długości: w powyższej obserwacji mamy ograniczenie $2{,}65^{2t}$ zamiast naiwnego 3^{2t}.

W dowodzie obserwacji 3 jest jednak dużo luzu. Niestety do precyzyjnego określenia wartości $F(2t)$ potrzebne są trochę bardziej zaawansowane narzędzia. Zamiast nich zadowolimy się tutaj następującym, dość dokładnym, oszacowaniem.

Lemat 2.

$$F(t) \leqslant \sqrt{2} \cdot (1 + \sqrt{2})^t.$$

Dowód: Dla $\alpha \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, przez $F_\alpha(t)$ oznaczmy liczbę dobrych t-literowych słów, kończących się na literę α. Oczywiście, $F(t) = F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t)$. Zdefiniujmy $G(t) = \sqrt{2} \cdot F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t)$. Będziemy, przez indukcję po wartości t, dowodzić następującej równości, która natychmiast daje tezę lematu (gdyż $F(t) \leqslant G(t)$):

$$G(t) = \sqrt{2} \cdot (1 + \sqrt{2})^t.$$

Dla $t = 1$ mamy $F_\mathtt{a}(1) = F_\mathtt{b}(1) = F_\mathtt{c}(1) = 1$ i powyższa równość zachodzi.

Załóżmy, że równość ta jest spełniona dla pewnego $t \geqslant 1$. Niech w będzie $(t+1)$-literowym dobrym słowem, niech α będzie ostatnią literą w, i niech w' będzie słowem w z obciętą ostatnią literą. Oczywiście, w' jest t-literowym dobrym słowem. Jeśli $\alpha = \mathtt{a}$, to w' może kończyć się na dowolną literę. Jeśli $\alpha = \mathtt{b}$, to w' nie może kończyć się na $\mathtt{c}$. Podobnie, jeśli $\alpha = \mathtt{c}$, to w' nie może kończyć się na $\mathtt{b}$. Uzyskujemy więc następujące zależności:

$$\begin{aligned}
F_\mathtt{a}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t),\\
F_\mathtt{b}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{b}(t),\\
F_\mathtt{c}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{c}(t).
\end{aligned}$$

Podsumowując:

$$\begin{aligned} G(t+1) &= \sqrt{2}\cdot F_{\mathtt{a}}(t+1) + F_{\mathtt{b}}(t+1) + F_{\mathtt{c}}(t+1) \\ &= \sqrt{2}\big(F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t) + F_{\mathtt{c}}(t)\big) + \big(F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t)\big) + \big(F_{\mathtt{a}}(t) + F_{\mathtt{c}}(t)\big) \\ &= (2+\sqrt{2})F_{\mathtt{a}}(t) + (1+\sqrt{2})F_{\mathtt{b}}(t) + (1+\sqrt{2})F_{\mathtt{c}}(t) \\ &= (1+\sqrt{2})(\sqrt{2}\cdot F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t) + F_{\mathtt{c}}(t)) \\ &= (1+\sqrt{2})G(t) \\ &= \sqrt{2}\cdot(1+\sqrt{2})^{t+1}. \end{aligned}$$

■

Tak więc przy uważnej implementacji omówionego algorytmu programowania dynamicznego, rozważającego tylko „istotne" interfejsy σ, otrzymujemy rozwiązanie o złożoności czasowej $O((1+\sqrt{2})^t(n+m))$. To rozwiązanie zostało zaimplementowane w plikach `tur.cpp` i `tur0.pas`. Zauważmy, że dla ograniczenia $t = 10$ mamy $(1+\sqrt{2})^t < 7000$ i rozwiązanie powinno spokojnie mieścić się w limitach czasowych.

Wolniejsze implementacje

Rozwiązania zaimplementowane w plikach `turs2.cpp` i `turs5.pas` mają nieoptymalnie zaimplementowane interfejsy – niezależnie dla każdego wierzchołka S_i wybieramy, czy jest on wzięty do częściowego rozwiązania i czy jest on już zdominowany. Prowadzi to do złożoności czasowej $O(4^t(n+m))$.

Rozwiązania wolne w pozostałych plikach stosują trochę inną strukturę zamiatania drzewa T. Oparte one są o następujące spostrzeżenie: dla każdego wierzchołka v, jeśli oznaczymy przez C_v zbiór wszystkich potomków wierzchołka v w drzewie T, a przez S_v zbiór jego przodków (wliczając w to v), to S_v separuje C_v od $V \setminus (C_v \cup S_v)$ oraz oczywiście $|S_v| \leqslant t$. Niestety, w tym przypadku nie mamy tak ładnych przejść między kolejnymi stanami, jak w rozwiązaniu wzorcowym, i te algorytmy działają istotnie wolniej od rozwiązania wzorcowego.

Testy

Wygenerowanie ciekawych testów dla tego zadania okazało się pewnym wyzwaniem. Testy były podzielone na następujące kategorie:

1. drzewo o wysokości 3 i średnicy 5; do niektórych wierzchołków dodano krawędzie do niektórych przodków;
2. pełen graf dwudzielny, posiadający po jednej stronie 4 wierzchołki, a po drugiej stronie dużo wierzchołków; dodatkowo jedna z krawędzi została zastąpiona ścieżką długości 2;
3. losowy graf o 10 wierzchołkach;
4. losowy graf o 8 wierzchołkach, do którego podoczepiano liście;
5. losowy graf o 6 wierzchołkach, do którego podoczepiano poddrzewa o wysokości 2 lub trójkąty;

6. drzewo o średnicy 10.

Warto zwrócić uwagę, że weryfikacja, czy graf wejściowy spełnia warunki zadania (czy nie posiada ścieżki długości dłuższej niż $t = 10$), jest również bardzo trudnym problemem. Można go rozwiązać przez podobny algorytm programowania dynamicznego jak rozwiązanie wzorcowe, lecz jest to bardziej techniczne i żmudne. Zainteresowanych czytelników zachęcamy do zastanowienia się, jak taki algorytm mógłby wyglądać – w szczególności, jaka byłaby definicja interfejsu i jak by wyglądała tablica programowania dynamicznego.

Szersze spojrzenie: strukturalne miary grafu

W szerszym ujęciu opisane rozwiązanie wzorcowe jest tak naprawdę algorytmem programowania dynamicznego na dość specyficznej *dekompozycji ścieżkowej* (ang. *path decomposition*) wejściowego grafu. Bez podawania formalnej definicji, dekompozycja ścieżkowa grafu G opisuje, jak „zamieść" graf G, używając małej miotły. Obserwację 1 można przeformułować następująco: jeśli graf nie posiada ścieżki dłuższej niż t, to *głębokość drzewiasta* (ang. *treedepth*) grafu G wynosi co najwyżej t; głębokość drzewiasta spójnego grafu G jest najmniejszą możliwą wysokością drzewa T o tych samych wierzchołkach co G, takiego, że każda krawędź G łączy przodka i potomka w T (nie wymagamy tu, by T był podgrafem G).

Dużo bardziej znaną miarą strukturalną grafów jest *szerokość drzewiasta* (ang. *treewidth*), która odpowiada zamiataniu grafu G z użyciem małej miotły, ale tak, że w poszczególnych krokach zamiatanie może „rozgałęzić się" na różne części grafu G (czyli struktura zamiatania przypomina drzewo, a nie ścieżkę). Mając daną dekompozycję drzewiastą (czyli taką receptę, jak zamiatać graf) o małej szerokości (czyli małej miotle), można rozwiązać efektywnie wiele problemów, używając programowania dynamicznego w podobny sposób, jak to zrobiliśmy w rozwiązaniu wzorcowym. Umiejętność rozwiązywania trudnych problemów na grafach o ograniczonej szerokości drzewiastej okazuje się istotnym elementem w wielu zastosowaniach, zarówno w teoretycznej jak i praktycznej informatyce.

Zainteresowanych czytelników odsyłamy do rozdziału 12 w książce R. Diestla [41], gdzie znajduje się dokładny opis dekompozycji drzewiastych i szerokości drzewiastej, oraz do rozdziału 7 w nowo powstałej książce o algorytmach parametryzowanych [42], który opisuje różne algorytmy programowania dynamicznego na dekompozycjach[3].

[3]Książka powinna się ukazać na początku 2015 roku.