Michał Pilipczuk
Treść zadania, Opracowanie

Adam Iwanicki
Program

OI, Etap I, 22.10–19.11.2007

Cło

Król Bajtazar postanowił uporządkować kwestie związane z opłacaniem cła przez kupców Bajtocji. Bajtocja składa się z n miast połączonych m dwukierunkowymi drogami. Każda droga w Bajtocji łączy dwa różne miasta. Żadne dwa miasta nie są połączone więcej niż jedną drogą. Drogi mogą prowadzić przez tunele i estakady.

Dotychczas każde miasto w Bajtocji pobierało cło od każdego, kto do niego przyjeżdżał, i od każdego, kto z niego wyjeżdżał. Niezadowoleni z tej sytuacji kupcy wnieśli oficjalny protest, w którym sprzeciwili się wielokrotnemu pobieraniu cła. Król Bajtazar postanowił ograniczyć przywileje miast. Wedle nowego królewskiego edyktu, każde miasto może pobierać cło od kupców podróżujących dokładnie jedną z dróg prowadzących do niego (bez względu na kierunek ich podróży). Ponadto, dla każdej z dróg, podróżni podróżujący tą drogą nie mogą być zmuszeni do płacenia cła obu miastom, które ta droga łączy. Należy jeszcze podjąć decyzję, które miasto ma pobierać cło z której drogi. Rozwiązanie tego problemu król zlecił Tobie.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis układu dróg w Bajtocji,*
- *dla każdego miasta wyznaczy, na której drodze dane miasto będzie pobierać od kupców cło, lub stwierdzi, że wprowadzenie w życie edyktu nie jest możliwe,*
- *wypisze wynik na standardowe wyjście.*

Wejście

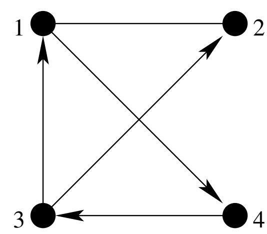
W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m ($1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 200\,000$), oznaczające odpowiednio liczbę miast oraz dróg w Bajtocji. Miasta są ponumerowane od 1 do n. W kolejnych m wierszach znajdują się opisy kolejnych dróg. W wierszu i znajdują się dwie liczby całkowite a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$) oznaczające, że miasta a_i i b_i są połączone bezpośrednią drogą.

Wyjście

Jeśli pobieranie cła zgodnie z wymaganiami królewskiego edyktu nie jest możliwe, to w pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać słowo `NIE`*. W przeciwnym przypadku, w pierwszym wierszu Twój program powinien wypisać słowo* `TAK`*, a w kolejnych n wierszach powinny się znaleźć informacje, które miasto, z jakiej drogi pobiera cło. W wierszu $i+1$ powinien znaleźć się numer miasta, do którego prowadzi droga, na której miasto numer i pobiera od kupców cło. W przypadku, gdy istnieje wiele rozwiązań, należy podać dowolne z nich.*

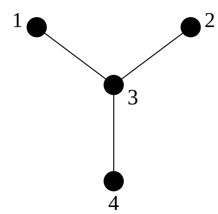
Przykład

Dla danych wejściowych:
```
4 5
1 2
2 3
1 3
3 4
1 4
```
poprawnym wynikiem jest:
```
TAK
3
3
4
1
```



Strzałki na rysunku wskazują miasta pobierające cło od kupców podróżujących daną drogą. Zwróć uwagę, że kupcy podróżujący drogą łączącą miasta 1 i 2 nie płacą w ogóle cła.

Dla danych wejściowych:
```
4 3
1 3
3 4
2 3
```
poprawnym wynikiem jest:
```
NIE
```



Rozwiązanie

Wprowadzenie

Opiszmy problem z zadania w języku teorii grafów. Dany jest graf nieskierowany $G = (V, E)$, w którym wierzchołkami są miasta znajdujące się w Bajtocji, a krawędziami — drogi pomiędzy nimi. Każdemu wierzchołkowi należy przyporządkować jedną spośród incydentnych[1] z nim krawędzi, co odpowiada nadaniu miastu prawa do pobierania cła z drogi. Każda krawędź może być przyporządkowana co najwyżej jednemu wierzchołkowi — tak, aby podatek nie był pobierany na żadnej drodze dwukrotnie. Jeśli opisane przyporządkowanie nie istnieje, to również powinniśmy umieć to stwierdzić.

Poszukiwane rozwiązanie jest *funkcją różnowartościową* $f : V \to E$ taką, że dla każdego wierzchołka $v \in V$ wartość $f(v)$ jest krawędzią incydentną z v.

[1]Mówimy, że krawędź jest incydentna z wierzchołkiem, jeśli ten wierzchołek jest jednym z jej końców.

Rozwiązanie wzorcowe — proste przeszukiwanie

Zauważmy, że możemy zająć się poszukiwaniem funkcji przyporządkowującej krawędzie wierzchołkom oddzielnie dla każdej spójnej składowej grafu. Niech $G = H_1 \cup H_2 \cup \ldots \cup H_k$, gdzie H_i to spójne składowe G. Wtedy znajdując poszukiwane w zadaniu przyporządkowanie f_i dla każdej składowej H_i, możemy zdefiniować przyporządkowanie f dla całego grafu G jako f_i na każdym H_i. Oczywiście, jeśli dla pewnej składowej nie istnieje przyporządkowanie spełniające warunki zadania, to nie istnieje także przyporządkowanie dla całego grafu.

Przypadki bez rozwiązania

Zastanówmy się wpierw, dla jakich grafów odpowiedź na pytanie postawione w zadaniu jest negatywna. Pomocne będzie następujące proste spostrzeżenie.

Fakt 1. *Jeśli spójna składowa H grafu G jest drzewem, to nie istnieje dla niej poszukiwane przyporządkowanie.*

Dowód: Niech $H = (W, F)$ będzie drzewem. Oznaczmy przez $|W|$ i $|F|$ odpowiednio moc zbioru wierzchołków i krawędzi. Wiemy, że w każdym drzewie liczba krawędzi jest o jeden mniejsza niż liczba wierzchołków, więc $|F| = |W| - 1$. Zauważmy także, że jeśli funkcja f ma być różnowartościowa, to musi $|W|$ wierzchołkom przypisać dokładnie $|W|$ różnych krawędzi. Ponieważ w składowej H nie ma tylu krawędzi, więc jest to niemożliwe — stąd f nie istnieje.

■

Okazuje się, że fakt 1 zawiera w sobie już wszystkie przypadki, dla których odpowiedź jest negatywna. Dla grafów, w których żadna spójna składowa nie jest drzewem, pokażemy algorytm konstrukcji przyporządkowania krawędzi wierzchołkom dla poszczególnych składowych, a tym samym — przyporządkowania dla całego grafu.

Konstrukcja

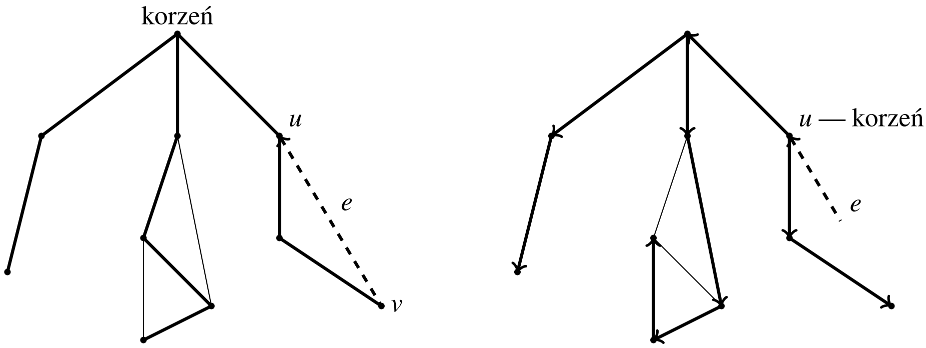
Rozważmy spójną składową $H = (W, F)$. Dla składowej tej istnieje *drzewo rozpinające*, które możemy *ukorzenić* w dowolnym wierzchołku. Wybór korzenia określa *orientację* w składowej — korzeń jest na samej górze, jego dzieci poniżej, ich dzieci jeszcze niżej itd. Każdemu z wierzchołków, oprócz korzenia, możemy przypisać krawędź łączącą go z jego ojcem w drzewie. W ten sposób dostajemy prawie dobre rozwiązanie — każdemu wierzchołkowi, oprócz korzenia, przypisaliśmy inną krawędź.

Aby uzyskać pełne rozwiązanie, musimy jeszcze znaleźć krawędź dla korzenia. Możemy w tym celu przed konstrukcją drzewa wybrać krawędź $e = (u, v) \in F$, której usunięcie nie rozspójni rozważanej składowej, a następnie:

- skonstruować drzewo rozpinające dla $H' = (W, F \setminus \{e\})$;
- za korzeń skonstruowanego drzewa wybrać jeden z wierzchołków końcowych krawędzi e, na przykład u;
- przypisać $f(u) = e$, a dla pozostałych wierzchołków $w \in W \setminus \{u\}$ za $f(w)$ przyjąć krawędź łączącą w z jego ojcem w drzewie.

Wszystkie powyższe operacje możemy wykonać, posługując się jednym z algorytmów przeszukiwania grafu: w głąb (DFS) lub wszerz (BFS), które są opisane w [20]. Najpierw musimy wybrać krawędź e, czyli dowolną krawędź należącą do jakiegokolwiek cyklu (co gwarantuje, że jej usunięcie nie spowoduje rozspójnienia składowej). Takie krawędzie łatwo rozpoznać w trakcie algorytmu przeszukiwania grafu — próba przejścia taką krawędzią prowadzi do wierzchołka już wcześniej odwiedzonego i zamyka cykl złożony z krawędzi drzewa. Poszukiwana krawędź nie istnieje zatem wtedy i tylko wtedy, gdy cała spójna składowa jest drzewem — jednak wówczas mamy do czynienia z przypadkiem negatywnym na mocy faktu 1. W przeciwnym razie znajdujemy odpowiednią krawędź $e = (u, v)$ i za korzeń drzewa rozpinającego wybieramy jeden z jej końców, na przykład u.

Teraz możemy usunąć krawędź e z grafu i ponownie uruchomić algorytm DFS lub BFS, startując z wierzchołka u — w ten sposób skonstruujemy drzewo rozpinające. Każdemu wierzchołkowi w możemy przypisać krawędź $f(w)$ — tę, którą weszliśmy do tego wierzchołka w trakcie przechodzenia grafu. Jest to krawędź łącząca wierzchołek w z jego ojcem w skonstruowanym drzewie rozpinającym.



(a) (b)

Rys. 1: Przykład działania opisanej procedury dla spójnej składowej. W pierwszej iteracji (a) za pomocą przeszukiwania DFS znajdowane jest drzewo rozpinające (grubsze linie), a także wybierana jest pewna krawędź powrotna $e = (u, v)$, która staje się $f(u)$. W drugiej iteracji (b), w wyniku przeszukiwania rozpoczętego w u, każdemu wierzchołkowi poza u zostaje przypisana krawędź, którą do niego wchodzimy (przypisanie krawędzi oznaczono na rysunku strzałkami).

Algorytm wzorcowy

Opisany wyżej pomysł to nasz algorytm wzorcowy:

```
for s ∈ V
  if s nie przypisano jeszcze krawędzi then
  begin
    wykonaj przeszukiwanie z wierzchołka s:
      wyznacz krawędź e należącą do cyklu;
    if krawędź e nie istnieje then
    begin
      wypisz NIE;
```

```
            zakończ działanie programu;
        end else
        begin
            niech u będzie jednym z końców e;
            przypisz f(u) = e;
            usuń krawędź e z grafu;
            wykonaj przeszukanie z wierzchołka u:
                przypisz każdemu wierzchołkowi krawędź,
                którą do niego wchodzimy;
        end;
    end;
end;
wypisz TAK oraz zapamiętane przyporządkowanie f.
```

Algorytm automatycznie rozpoznaje spójne składowe grafu — przetwarzając wierzchołek s, odwiedzamy i przetwarzamy wszystkie wierzchołki należące do tej samej składowej, co on. Potem wracamy do głównej pętli algorytmu i poszukujemy nieprzetworzonego wierzchołka grafu, który wyznacza jeszcze nierozpatrzoną spójną składową.

Zauważmy, że w każdej spójnej składowej wykonujemy dwa przeszukiwania — każde o złożoności czasowej liniowej względem jej rozmiaru. Cały algorytm ma zatem złożoność liniową względem rozmiaru grafu, czyli $O(n+m)$. Złożoność pamięciowa jest taka sama, gdyż musimy pamiętać jedynie reprezentację grafu, zaznaczać odwiedzane wierzchołki i zapisywać konstruowane przyporządkowanie f. Wszystkie te dane mieszczą się w strukturach o rozmiarze $O(n+m)$.

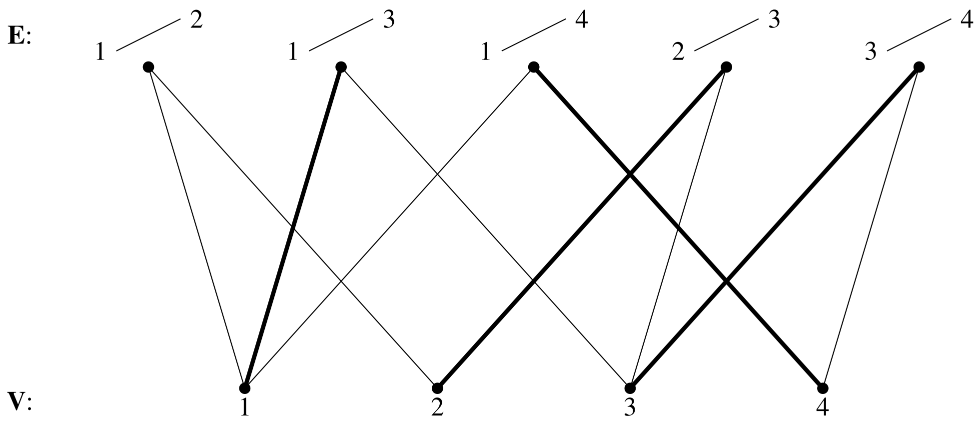
Rozwiązania bazujące na powyższym schemacie mogą różnić się zastosowanym algorytmem przeszukiwania. Implementacje z przeszukiwaniem w głąb znajdują się w plikach: `clo6.c`, `clo7.cpp`, `clo8.pas` oraz `clo11.java`, natomiast z przeszukiwaniem wszerz — w plikach `clo.c`, `clo1.cpp`, `clo2.pas` oraz `clo9.java`.

Rozwiązanie alternatywne — zastosowanie skojarzeń

Problem postawiony w zadaniu możemy rozwiązać także inną metodą — stosując bardziej skomplikowane pojęcia i otrzymując w wyniku algorytm nie lepszy niż wzorcowy. Uważamy jednak, że warto przedstawić także ten pomysł, gdyż pozwala on poznać kolejne ważne pojęcia i ciekawe algorytmy z teorii grafów.

Sprowadzimy zadanie do problemu znajdowania *maksymalnego skojarzenia w grafie dwudzielnym*. Oznaczmy oryginalny graf przez $G=(V,E)$. Zbudujmy dla niego graf dwudzielny $H=(V\cup E,F)$, gdzie $(v,e)\in F$ dla $v\in V$ oraz $e\in E$ wtedy i tylko wtedy, gdy v jest incydentne z e. Mówiąc prościej, graf H ma dwa zbiory wierzchołków: w jednym z nich są wierzchołki grafu G, a w drugim — krawędzie G. Wierzchołek v i krawędź e są połączone w grafie H krawędzią, jeśli v jest jednym z końców e w grafie G. A zatem H zawiera łącznie $n+m$ wierzchołków oraz $2m$ krawędzi.

Skojarzenie w grafie H to podzbiór krawędzi tego grafu, które nie mają wspólnych wierzchołków. Wierzchołki incydentne z krawędziami wybranymi do skojarzenia nazywamy *skojarzonymi* lub *pokrytymi* przez skojarzenie. Każde skojarzenie w grafie H możemy zinterpretować jako różnowartościowe przyporządkowanie krawędzi wierzchołkom grafu G. Oczywiście zależy nam na znalezieniu jak największego skojarzenia — w końcu chcemy przypisać krawędź każdemu wierzchołkowi. Będziemy więc szukać skojarzenia maksymalnego pod względem liczności. Jeśli okaże się, że zawiera ono $|V|$ krawędzi, to będziemy mieli poszukiwane przyporządkowanie. W przeciwnym razie będziemy wiedzieli, że takie przyporządkowanie nie istnieje.



Rys. 2: Graf dwudzielny H skonstruowany dla grafu G z pierwszego przykładu z treści zadania. Dolne cztery wierzchołki H odpowiadają wierzchołkom G, natomiast górne pięć odpowiada krawędziom G. Pogrubione krawędzie reprezentują najliczniejsze skojarzenie w H, odpowiadające przyporządkowaniu wierzchołkom G takich samych krawędzi, jak w pierwszym przykładowym wyjściu.

Więcej o skojarzeniach i związanych z nimi algorytmach można przeczytać w [23].

Trochę kombinatoryki

Warunek określający istnienie skojarzenia w grafie H pokrywającego cały zbiór V jest oczywiście analogiczny do przedstawionego przy okazji algorytmu wzorcowego. Wystarczy, by żadna spójna składowa grafu G nie była drzewem — wtedy istnieje poszukiwane przyporządkowanie krawędzi wierzchołkom w grafie G, które z kolei odpowiada skojarzeniu w grafie H o mocy $|V|$. Warto jednak, korzystając z okazji, przytoczyć także inny warunek, oparty na twierdzeniu Halla, o którym można przeczytać np. w [31]. Pozwala on rozstrzygnąć o istnieniu skojarzenia pokrywającego jeden ze zbiorów wierzchołków także w szerszej klasie grafów, niż tutaj rozważane.

Twierdzenie 1 (Hall). *Niech $H = (A \cup B, E)$, gdzie $E \subseteq A \times B$, będzie grafem dwudzielnym. W H istnieje skojarzenie, w którym każdy wierzchołek ze zbioru A jest pokryty, wtedy i tylko wtedy, gdy zachodzi następujący warunek:*

dla każdego podzbioru $X \subseteq A$, zbiór tych wierzchołków $y \in B$, dla których istnieje $x \in X$, takie że $(x,y) \in E$, ma moc równą co najmniej $|X|$.

Korzystając z twierdzenia Halla, można napisać algorytm konstrukcji skojarzenia — niestety, jest on daleki od optymalnego. Wynaleziono jednak sporo innych, znacznie efektywniejszych rozwiązań tego problemu.

Algorytm Hopcrofta-Karpa

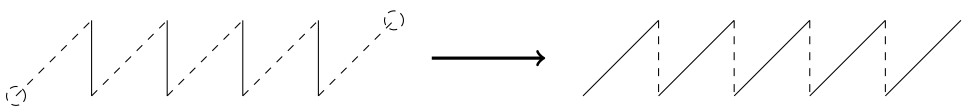
Najszybszą i nietrudną w implementacji metodą znajdowania maksymalnego skojarzenia w grafie dwudzielnym w ogólnym przypadku jest algorytm Hopcrofta-Karpa, opisany w [23]. Działa on w czasie $O(|E|\sqrt{|V|})$, gdzie V i E to odpowiednio zbiory wierzchołków i krawędzi. Przypomnijmy, że graf dwudzielny H zbudowany na podstawie grafu danego w zadaniu ma $n+m$ wierzchołków oraz $2m$ krawędzi. Zastosowanie do niego metody Hopcrofta-Karpa pozwala więc wyznaczyć rozwiązanie w czasie $O(m^{\frac{3}{2}})$, czyli istotnie gorszym od czasu działania algorytmu wzorcowego. Takie rozwiązanie jest zaimplementowane w pliku `clos1.cpp` — ze względu na kiepską złożoność nie uzyskuje ono kompletu punktów.

Turbo-matching

Turbo-matching to nieoficjalna nazwa nadana algorytmowi otrzymanemu przez drobną, acz brzemienną w skutki modyfikację podstawowego algorytmu znajdowania maksymalnego skojarzenia. Algorytm podstawowy jest dokładnie opisany w [20], więc tutaj tylko krótko omówimy schemat jego działania. Startujemy w nim od skojarzenia pustego, które stopniowo zwiększamy, wykorzystując tak zwane *ścieżki powiększające* (inaczej nazywane także *naprzemiennymi*). Ścieżkę w grafie nazwiemy powiększającą, jeśli:

- rozpoczyna się ona od wierzchołka nieskojarzonego jeszcze z żadnym innym;
- zawiera na przemian krawędzie nienależące do skojarzenia i należące do skojarzenia;
- kończy się także na wierzchołku nieskojarzonym.

Jeśli przez E_1 oznaczymy zbiór krawędzi ścieżki należących do skojarzenia, a przez E_2 — zbiór jej pozostałych krawędzi, to łatwo zauważyć, że $|E_1| = |E_2| - 1$. Ponadto, jeśli ze skojarzenia wyrzucimy krawędzie ze zbioru E_1, a dodamy krawędzie ze zbioru E_2, to dostaniemy nowe, większe skojarzenie! (patrz rys. 3) Można udowodnić, że dla dowolnego nie najliczniejszego skojarzenia istnieje ścieżka powiększająca, czyli że powiększając skojarzenie za pomocą ścieżek naprzemiennych, w końcu uzyskuje się najliczniejsze skojarzenie.



(a) (b)

Rys. 3: Przykład ścieżki naprzemiennej (a) dla $|E_1| = 4$ i $|E_2| = 5$ (krawędzie ze skojarzenia oznaczone są liniami ciągłymi, a pozostałe — przerywanymi) oraz wyniku dołączenia krawędzi z E_2 do skojarzenia oraz „odkojarzenia” krawędzi z E_1 (b). Wierzchołki nieskojarzone zaznaczone są kółkami.

Niech $G = (V_1 \cup V_2, E)$ będzie danym grafem dwudzielnym. Schemat samego algorytmu wykorzystującego ścieżki powiększające jest następujący:

```
podstaw za skojarzenie zbiór pusty;
repeat
   znalezione := false;
   for v ∈ V1 do
      if v jest nieskojarzony then
      begin
         poszukaj ścieżki powiększającej z wierzchołka v;
         { w tym celu stosujemy przeszukiwanie w głąb (DFS) z wierzchołka v }
         if znaleziono ścieżkę then
         begin
            popraw skojarzenie, wykorzystując ścieżkę;
            znalezione := true;
         end;
         zaznacz odwiedzone wierzchołki jako nieodwiedzone;
      end;
   if not znalezione then przerwij;
end;
```

Powyższy algorytm działa w czasie $O(|V| \cdot |E|)$.
Algorytm turbo-matching niewiele różni się od poprzedniego:

```
1:  podstaw za skojarzenie zbiór pusty;
2:  repeat
3:     znalezione := false;
4:     for v ∈ V1 do
5:        if v jest nieskojarzony i nieodwiedzony then
6:        begin
7:           poszukaj ścieżki powiększającej z wierzchołka v procedurą DFS;
8:           if znaleziono ścieżkę then
9:           begin
10:             popraw skojarzenie, wykorzystując ścieżkę;
11:             znalezione := true;
12:          end;
13:       end;
14:    if not znalezione then przerwij;
15:    zaznacz odwiedzone wierzchołki jako nieodwiedzone;
16: end;
```

Istotną modyfikacją dokonaną w algorytmie podstawowym jest zmiana momentu „zerowania tablicy odwiedzin”. W algorytmie turbo-matching zostało to przeniesione na zewnątrz pętli **for** (wiersz 15). To oznacza, że w jednej iteracji tej pętli w kolejnych wywołaniach przeszukiwania w głąb (wiersz 7) nie wchodzimy wielokrotnie do tych samych wierzchołków. Ta zmiana nie psuje poprawności algorytmu — jeśli istnieje ścieżka powiększająca, to ją znajdziemy. Istotnie, jeśli w jednej iteracji pętli **for**, przechodząc przez jakiś wierzchołek, nie znaleźliśmy ścieżki powiększającej, to ponowne wchodzenie do niego w tej samej iteracji pętli nie ma sensu. Jeśli zatem istnieją ścieżki powiększające, to pierwsza z nich zostanie wyszukana bez potrzeby wchodzenia do odwiedzonych wcześniej wierzchołków.

Teoretycznie złożoność turbo-matchingu jest taka sama jak algorytmu podstawowego — w każdej iteracji pętli **for** przetwarzamy każdą krawędź i każdy wierzchołek co najwyżej raz, na co potrzebujemy czasu $O(|V|+|E|)$. Jednocześnie każde wykonanie tej pętli generuje nową ścieżkę powiększającą, zatem obrotów będzie co najwyżej $|V|$ — tyle, ile maksymalnie krawędzi ma skojarzenie. Cały algorytm ma złożoność $O(|V| \cdot (|V| + |E|))$, a ponieważ w sensownych grafach mamy $|V| = O(|E|)$, to daje ostatecznie złożoność turbo-matchingu równą $O(|V| \cdot |E|)$.

W praktyce turbo-matching okazuje się jednak dużo szybszy. Wprowadzona modyfikacja sprawia, że poszukując ścieżek powiększających, często pomijamy wierzchołki, o których wiadomo, że i tak nie ma sensu do nich wchodzić. Ze względu na małą stałą, algorytm ten dla rozsądnych danych zachowuje się nie gorzej niż algorytm Hopcrofta-Karpa. Jest za to prostszy i łatwiejszy do zaprogramowania.

Grafy dwudzielne, które występują w naszym zadaniu, są dość proste — wszystkie wierzchołki z jednej części (odpowiadające krawędziom z oryginalnego grafu) mają stopień równy 2. W takim przypadku turbo-matching działa dużo szybciej od algorytmu Hopcrofta-Karpa i nie udało się znaleźć przykładów, w których byłby istotnie wolniejszy od rozwiązań wzorcowych, liniowych. Dlatego jego poprawna implementacja, zawarta w pliku `clos3.cpp`, otrzymuje maksymalną liczbę punktów.

Testy

W powyższym opracowaniu zauważyliśmy, że rozwiązania szukamy praktycznie oddzielnie w każdej spójnej składowej. Dlatego grafy występujące w testach zostały skonstruowane z różnego rodzaju składowych:

- grafów losowych o zadanej gęstości;
- dużych cykli;
- drzew — używanych do testów z odpowiedzią negatywną;
- klik, czyli grafów pełnych;
- „drzew z cyklem" — w tej kategorii znalazły się dwa rodzaje grafów: drzewa z cyklem losowej długości i drzewa z cyklem długości 3.

Rozwiązania zawodników były sprawdzane na 10 grupach testów.

Nazwa	**n**	**m**	**Opis**
clo1a.in	40	60	mały test poprawnościowy z odpowiedzią pozytywną
clo1b.in	10	9	małe drzewo
clo2a.in	175	175	większy test poprawnościowy złożony z dwóch „drzew z cyklem”
clo2b.in	175	11 199	klika i drzewo
clo3.in	600	600	większy test poprawnościowy: cykl i dwa „drzewa z cyklami”
clo4.in	1 000	1 000	„drzewo z cyklem”
clo5.in	5 000	9 850	dwa „drzewa z cyklami”, cykl i klika
clo6.in	15 000	15 000	„drzewo z cyklem” i cykl
clo7.in	35 000	35 000	dwa „drzewa z cyklami” i cykl
clo8.in	50 000	50 000	dwa większe „drzewa z cyklami” i cykl
clo9.in	80 000	158 800	cykl, cztery małe kliki oraz dwa „drzewa z cyklami”
clo10a.in	100 000	180 000	„drzewo z cyklem” oraz losowy graf o dużej gęstości
clo10b.in	100 000	199 233	klika i duże drzewo