Zbigniew Czech
Treść zadania, Opracowanie

Michał Pilipczuk
Program

OI, Etap III, dzień pierwszy, 02.04.2008

Podział Królestwa

Król Bajtocji, Bajtazar, postanowił przejść na emeryturę. Ma on dwóch synów. Nie może się jednak zdecydować, który z nich powinien być jego następcą. Postanowił więc podzielić królestwo na dwie połowy i uczynić swoich synów ich władcami.

Po podziale królestwa, na drogach łączących obie połowy należy zbudować strażnice. Ponieważ wiąże się to z kosztami, dróg łączących obie połowy powinno być możliwie najmniej.

Bajtocja składa się z parzystej liczby miast połączonych drogami. W wyniku podziału w każdej połowie powinna znaleźć się połowa miast. Każda droga łączy dwa miasta. Drogi nie łączą ani nie krzyżują się poza miastami, mogą natomiast występować wiadukty tudzież tunele. Każde dwa miasta mogą być bezpośrednio połączone co najwyżej jedną drogą.

Przy podziale królestwa istotne jest, które miasta znajdą się w której połowie. Możesz założyć, że teren poza miastami można tak podzielić, że drogi łączące miasta leżące w tej samej połowie nie będą przecinać granicy. Natomiast na każdej drodze łączącej miasta z różnych połówek trzeba wybudować jedną strażnicę.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis miast i łączących je dróg,*
- *wyznaczy taki podział królestwa na dwie połowy, że w każdej połowie będzie tyle samo miast, a liczba dróg łączących miasta leżące w różnych połowach będzie minimalna,*
- *wypisze wyznaczony wynik na standardowe wyjście.*

Jeżeli istnieje wiele poprawnych podziałów królestwa, Twój program powinien wyznaczyć którykolwiek z nich.

Wejście

W pierwszym wierszu wejścia zapisane są dwie liczby całkowite n i m oddzielone pojedynczym odstępem, równe odpowiednio liczbie miast i liczbie łączących je dróg, $2 \leqslant n \leqslant 26$, $2|n$, $0 \leqslant m \leqslant \frac{n \cdot (n-1)}{2}$. Miasta są ponumerowane od 1 do n. W kolejnych m wierszach zapisane są po dwie liczby całkowite oddzielone pojedynczym odstępem. W wierszu $(i+1)$-szym (dla $i = 1, 2, \ldots, m$) zapisane są liczby u_i i v_i, $1 \leqslant u_i < v_i \leqslant n$. Reprezentują one drogę łączącą miasta u_i i v_i.

Wyjście

Twój program powinien wypisać jeden wiersz zawierający $\frac{n}{2}$ liczb całkowitych pooddzielanych pojedynczymi odstępami. Powinny to być numery miast należących do tej połówki królestwa, do której należy miasto nr 1, podane w rosnącej kolejności.

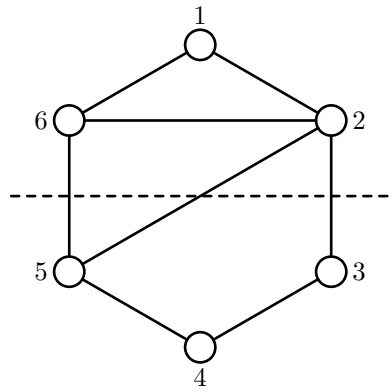
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 2 6
```



Na rysunku linią przerywaną zaznaczono optymalny podział, który wymaga zbudowania 3 strażnic.

Rozwiązanie

Rozwiązanie wzorcowe

Zadanie, jakie ma do rozwiązania król Bajtocji, dotyczy problemu minimalnego połowienia (bisekcji) grafu, który formułuje się następująco. Dany jest graf $G = (V, E)$, gdzie V jest zbiorem wierzchołków, a E zbiorem krawędzi. Zakładamy, że liczba miast $|V| = n$ jest parzysta. Należy podzielić zbiór wierzchołków grafu na dwa równoliczne podzbiory Y_1 i Y_2 tak, aby liczba krawędzi łączących wierzchołki podzbiorów Y_1 i Y_2 była minimalna. Problem minimalnego połowienia grafu należy do klasy NP-zupełnych. Przedstawienie programu rozwiązującego większe przykłady problemu w sposób optymalny w rozsądnym czasie jest więc niemożliwe, o ile P≠NP[1]. Jednak dla tak niewielkich wartości n, jakie występują w treści zadania, można zaryzykować rozwiązanie w czasie wykładniczym polegające na przejrzeniu wszystkich możliwych bisekcji. Oznaczmy liczbę wierzchołków grafu przez

[1]Do klasy P należą zadania, które potrafimy rozwiązać w czasie wielomianowym: $O(n^2)$, $O(n^3)$ czy nawet $O(n^{100})$. Do klasy NP należą zadania, których na razie nie potrafimy tak szybko rozwiązywać — potrafimy poradzić sobie w czasie około 2^n (przekonaj się sam, od jakiego n zaczyna się przewaga 2^n nad n^{100} i jak szybko potem się ona zwiększa). W klasie NP występują zadania, które nieformalnie można nazwać „kluczowymi", a formalnie określa się je jako NP-zupełne. Gdybyśmy dysponowali programem rozwiązującym dowolne zadanie NP-zupełne w czasie wielomianowym, to wykorzystując go jako procedurę, potrafilibyśmy w czasie wielomianowym rozwiązać każde zadanie z klasy NP i okazałoby się, że P=NP. Naukowcy skłaniają się ku podejrzeniom, że klasy P i NP są różne. Gdy więc napotkasz na problem NP-zupełny, rozsądek nakazuje wierzyć, że uda Ci się napisać program, który znajduje rozwiązanie tego problemu: dla małych przypadków w czasie wykładniczym, dla charakterystycznych przypadków w czasie wielomianowym, ewentualnie z pewnym przybliżeniem (za pomocą algorytmu aproksymacyjnego) lub gdy przypisze Ci szczęście (za pomocą algorytmu losowego). Więcej o klasach P i NP oraz problemach NP-zupełnych można przeczytać w [15] i [20] (przyp. red.).

$|V| = n = 2N$. Wówczas liczba możliwych połowień grafu, jakie należy sprawdzić, wynosi:

$$\binom{n}{n/2} = \binom{2N}{N} = \frac{(2N)!}{N!N!}.$$

Powyższą wartość można oszacować, stosując wzór Stirlinga:

$$\frac{(2N)!}{N!N!} = \frac{\sqrt{2\pi 2N}\frac{(2N)^{2N}}{e^{2N}}(1+o(1))}{2\pi N\frac{N^{2N}}{e^{2N}}(1+o(1))^2} = \frac{2^{2N}(1+o(1))}{\sqrt{\pi N}} = \Theta\left(\frac{2^{2N}}{\sqrt{N}}\right) \tag{1}$$

Poszukując jak najlepszego rozwiązania, zastosujemy przeszukiwanie wyczerpujące. Nie możemy liczyć na istotne zmniejszenie liczby przypadków do rozważenia, ale możemy starać się uzyskać jak najniższy współczynnik proporcjonalności poprzedzający czynnik wykładniczy w funkcji złożoności. Poniższy lemat pokazuje, jak w efektywny sposób generować wszystkie interesujące nas połowienia grafu.

Lemat 1. Dla dowolnych liczb naturalnych k,l istnieje ciąg (a_i), którego elementami są wszystkie słowa zerojedynkowe długości $k+l$ mające dokładnie k jedynek, gdzie każde takie słowo występuje dokładnie raz, a_1 jest dowolnym takim słowem oraz każde dwa kolejne elementy ciągu różnią się na dokładnie dwóch pozycjach.

Dowód: Przeprowadzimy dowód indukcyjny ze względu na sumę $k+l$. Najpierw zauważmy, że jeśli $k = 0$ lub $l = 0$, to teza jest oczywista — mamy po prostu jedno słowo składające się z samych zer lub samych jedynek. Załóżmy teraz, że dla wszystkich par liczb k',l' o sumie mniejszej od $k+l$ teza jest prawdziwa oraz że $k > 0$ i $l > 0$. Weźmy jako a_1 dowolne słowo ze zbioru, który mamy wygenerować. Bez straty ogólności możemy przyjąć, że ostatnim znakiem a_1 jest 1. Następnie:

- Zastosujmy założenie indukcyjne dla pary $k-1,l$, generując w pożądany sposób wszystkie słowa zerojedynkowe ciągu (a_i) zawierające na końcu jedynkę.
- Niech a będzie ostatnim wygenerowanym dotychczas słowem. Wyszukajmy pierwszy znak zero w słowie a (na pewno taki istnieje, bo $l > 0$). Utwórzmy kolejne słowo ciągu a', zamieniając miejscami w a znalezione zero i jedynkę z ostatniej pozycji.
- Następnie zastosujmy założenie indukcyjne dla pary $k,l-1$ i słowa początkowego a', uzyskując w ten sposób wszystkie szukane słowa z ostatnim znakiem równym zero.

Widzimy, że generując ciąg w podany wyżej sposób, dbamy o zachowanie wszystkich koniecznych warunków:

- każde wymagane słowo pojawia się w ciągu,
- żadne słowo nie powtarza się,
- każde dwa kolejne słowa ciągu różnią się dokładnie dwoma znakami.

■

Wniosek 1. Przedstawiony w powyższym dowodzie sposób konstrukcji ciągu (a_i) jest w istocie algorytmem rekurencyjnym, za pomocą którego można wygenerować taki ciąg w czasie $O\left((k+l)\frac{(k+l)!}{k!l!}\right)$. Drugi człon w funkcji złożoności jest liczbą wygenerowanych elementów ciągu (a_i), zaś pierwszy wynika z liniowego czasu szukania zera zamienianego z ostatnim znakiem.

Algorytm wzorcowy

1. *Wczytaj dane wejściowe i zbuduj graf o* $2N$ *wierzchołkach.*
2. *Ustal pierwszy podział, wybierając do pierwszej połówki wierzchołki* $1,2,\ldots,N$*, a resztę — do drugiej. Dla każdego wierzchołka oblicz, ilu ma sąsiadów w pierwszej połówce, a ilu w drugiej. Wyznacz także liczbę krawędzi przebiegających pomiędzy połowami.*
3. *Generuj kolejne podziały królestwa (właściwie* N*-elementowe podzbiory zbioru wierzchołków stanowiące pierwszą połowę), wykorzystując procedurę z Lematu 1. Zauważ, że dwa kolejne podziały różnią się wymianą jednej pary wierzchołków* (u,v) *pomiędzy połowami.*
   a. *Popraw liczby sąsiadów z odpowiednich połówek dla wierzchołków sąsiadujących z* u *i* v *oraz dla samych* u *i* v*.*
   b. *Uaktualnij także liczbę krawędzi pomiędzy połowami. Jeśli jest mniejsza niż dotychczasowe minimum, to zapamiętaj aktualny podział.*
4. *Wypisz jako wynik najlepszy znaleziony w kroku* 3. *podział.*

W algorytmie wykorzystaliśmy procedurę z Lematu 1, więc widać, że rozważyliśmy wszystkie możliwe podziały i wybraliśmy optymalny. Zastanówmy się jeszcze, ile czasu nam to zajęło. Dla każdego podziału aktualizacja statystyk dla wierzchołków i wyniku (liczby krawędzi łączących połówki) mogą być wykonane w czasie $O(N)$. Tyle samo czasu potrzebujemy, by przejść do kolejnego podziału. Wszystkich podziałów jest oczywiście $\frac{(2N)!}{N!N!}$. To oznacza, że mamy algorytm o złożoności czasowej $O\left(N\frac{(2N)!}{N!N!}\right)$ — ze wzoru (1) wynika, że jest to funkcja rzędu $O(\sqrt{N}2^{2N}) = O(\sqrt{|V|}2^{|V|})$.

Złożoność pamięciowa jest $O(N^2)$, gdyż największą strukturą, którą należy zapamiętać, jest reprezentacja grafu. Podziały zbioru wierzchołków mogą być zapamiętane w postaci wektora-słowa długości n, więc nie mają istotnego wpływu na złożoność pamięciową. Bez problemu mieścimy się więc nie tylko w limicie pamięciowym 64 MB, ale także w znacznie mniejszych.

W rozwiązaniu wzorcowym zostało dodane jeszcze jedno usprawnienie. Ze względu na symetrię połówek grafu, można na początku ustalić przynależność jednego wierzchołka i rozpatrywać podzbiory N-elementowe zbioru $(2N-1)$-elementowego. Nie poprawia to asymptotycznej złożoności algorytmu, ale dwukrotnie przyspiesza jego działanie.

Implementacja algorytmu wzorcowego znajduje się w plikach:

- `pod.cpp` — implementacja w języku C++,
- `pod1.pas` — implementacja w języku Pascal,
- `pod2.cpp` — implementacja w języku C++ z użyciem biblioteki STL.

Inne rozwiązania

Rozwiązania alternatywne

Algorytm 2.1. Algorytm ten jest implementacją pomysłu wzorcowego z zastosowaniem operacji na bitach do zapisu i modyfikacji podziału zbioru wierzchołków. Jeśli zapiszemy podział jako słowo maszynowe o $O(|V|)$ bitach (nazwijmy je *maską podziału*), wymianę wierzchołków i znajdowanie pierwszego zera w reprezentacji zbioru możemy wykonać za pomocą kilku operacji procesora.

W tym celu dla każdego wierzchołka zapisujemy jego stopień i maskę bitową reprezentującą jego sąsiadów (nazwijmy to słowo *maską sąsiedztwa wierzchołka*). Po przeniesieniu wierzchołka z jednej połowy do drugiej, chcielibyśmy w czasie stałym odpowiedzieć na pytanie, ilu sąsiadów z danej połówki ma przeniesiony wierzchołek. Można to zrobić, mnożąc bitowo (tj. stosując operację AND) maskę podziału przez maskę sąsiedztwa tego wierzchołka i następnie znajdując liczbę jedynek w tak otrzymanym słowie. Aby nie zliczać jedynek w słowie za każdym razem od nowa, można wcześniej, na początku algorytmu, utworzyć tablicę $L[0..2^{16}-1]$, zawierającą na pozycji i liczbę jedynek w reprezentacji binarnej liczby i (z przedziału $[0..2^{16}-1]$.) Wówczas aby obliczyć liczbę jedynek w podwójnym słowie a, wystarczy podzielić je na dwa słowa: a_1 złożone z początkowych 16 bitów i a_2 złożone z kolejnych 16 bitów, a następnie odczytać i dodać wartości $L[a_1]$ i $L[a_2]$. Przygotowanie tablicy L wymaga 2^{16} prostych operacji, a sprawdzenie liczby jedynek w każdej masce sąsiedztwa wierzchołka wykonujemy w czasie $O(1)$.

Operację przejścia do kolejnego podziału zbioru wierzchołków także realizujemy, wykorzystując operacje maszynowe i wykonując odpowiednie obliczenia wstępne. Obliczamy tablicę $P[0..2^{16}-1]$, gdzie $P[i]$ to pierwsza pozycja, na której w binarnej postaci liczby i występuje zero. Chcąc odszukać pozycję pierwszego zera w liczbie 32-bitowej, sprawdzamy wpierw, czy dolne 16 bitów liczby nie składa się z samych jedynek, i odpowiednio odczytujemy wartość z tablicy P dla dolnych lub górnych bitów liczby[2].

Przedstawiony sposób pozwala nam uniknąć liniowego kosztu wyszukiwania związanego z aktualizacją wyniku i przejściem do kolejnego podziału w zamian za kilka operacji bitowych i wstępne, jednorazowe obliczenia. Otrzymujemy w ten sposób algorytm o złożoności $O\left(2^{|V|}/\sqrt{|V|}\right)$. Jest on obarczony jednak sporą stałą, więc w rezultacie okazuje się tylko około 2 razy szybszy od programów wzorcowych. Implementacja tego rozwiązania znajduje się w plikach `pod3.cpp` i `pod4.pas`.

Algorytm 2.2. Kolejnym pomysłem jest przejrzenie wszystkich możliwych podziałów zbioru wierzchołków, a nie tylko tych, w których liczby wierzchołków w obu częściach są jednakowe. Taka ewidentna „rozrzutność" (rozważamy w końcu wiele niepotrzebnych podziałów) pozwala jednak łatwo śledzić liczby wierzchołków w obu częściach i inne własności masek podziału, gdyż zmieniają się one w sposób bardzo regularny. Obliczając koszty tylko tych podziałów, które są połowieniami (stosując w tym celu ten sam pomysł,

[2]Ciekawostka: w C/C++ w kompilatorach z serii GCC istnieją bardzo wygodne wbudowane instrukcje realizujące obie powyższe operacje. Funkcja `__builtin_popcount(x)` zlicza liczbę zapalonych bitów w liczbie `x` — jest ona realizowana podobnie, jak to zostało opisane powyżej. Z kolei funkcja `__builtin_ctz(x)` (ang. *count trailing zeros*) zwraca liczbę zer znaczących na końcu `x`, np. dla `x`=8 zwraca 3 (uwaga: funkcja jest niezdefiniowana dla `x`=0). W przeciwieństwie do poprzedniej, ta funkcja w procesorach z architekturą x86 jest realizowana za pomocą pojedynczej instrukcji procesora. Z kolei liczbą zawierającą jedynie najmniejszy zapalony bit liczby `x` jest `x & ~(x - 1)` (przyp. red.).

co w poprzednim algorytmie) uzyskujemy ostatecznie rozwiązanie o złożoności takiej jak rozwiązanie wzorcowe, choć w praktyce nieco wolniejsze. Podobnie jak poprzednie metody, ta także pozwalała uzyskać 100% punktów. Jej implementacja znajduje się w pliku `pod5.cpp`.

Rozwiązania wolniejsze

Rozwiązania wolniejsze są w zasadzie różnymi wariantami siłowego przeszukiwania zbioru podziałów grafu, w związku z czym ich złożoność różni się złożoności algorytmu wzorcowego jedynie czynnikiem poprzedzającym wyraz 2^n.

Algorytm 3.1. Algorytm polega na podobnym jak w rozwiązaniu wzorcowym przeszukiwaniu tylko tych podziałów, które są połowieniami. Dla każdego połowienia w czasie $O(|E|)$ obliczana jest liczba krawędzi pomiędzy połówkami grafu. W rezultacie złożoność algorytmu wynosi $O\left(|E|2^{|V|}/\sqrt{|V|}\right) = O(|V|^{\frac{3}{2}}2^{|V|})$. Przekłada się to na kilkunastokrotnie gorszy czas działania w stosunku do algorytmu wzorcowego.

Algorytm pozwalał uzyskać 30%–40% punktów, a jego implementacja znajduje się w plikach `pods1.cpp` i `pods4.pas`.

Algorytm 3.2. Algorytm jest prostszy, choć na pierwszy rzut oka bardziej rozrzutny niż poprzedni. Przeglądamy wszystkie podziały zbioru wierzchołków grafu i dla każdego podziału będącego połowieniem od nowa zliczamy liczbę krawędzi pomiędzy połówkami. W ten sposób dla $(2N)!/(N!N!)$ podziałów będących połowieniami wykonujemy operacje w czasie $O(|V|+|E|)$, a dla pozostałych — w czasie $O(|V|)$ (generacja kolejnego podziału). W rezultacie dostajemy algorytm o złożoności $O\left(|V|2^{|V|} + \frac{|E|}{\sqrt{|V|}}2^{|V|}\right)$, czyli $O(|V|^{\frac{3}{2}}2^{|V|})$. Złożoność tego algorytmu jest taka sama jak poprzedniego, lecz stała proporcjonalności jest nieco gorsza. Pozwalał on jednak także uzyskać 30%–40% punktów. Jego implementacja znajduje się w plikach `pods2.cpp` i `pods5.pas`.

Algorytm 3.3. Algorytm polega na przeglądaniu wszystkich podzbiorów zbioru wierzchołków, ale w uporządkowany sposób. Wszystkie słowa zerojedynkowe określonej długości można ułożyć w ciąg w ten sposób, by kolejne dwa różniły się dokładnie jednym bitem (podobny efekt uzyskaliśmy w Lemacie 1, ale dotyczył on słów o ustalonej liczbie jedynek i wymagał zmiany dwóch bitów). Wówczas śledzenie liczby krawędzi pomiędzy częściami podziału oraz kontrolę, czy podział jest połowieniem, można wykonać w czasie $O(|V|)$. Oczywiście aktualizację znalezionego minimum wykonujemy tylko wtedy, gdy aktualny podział jest połowieniem. W rezultacie otrzymujemy algorytm o złożoności $O(|V|2^{|V|})$, czyli tylko trochę gorszej od złożoności algorytmu wzorcowego. Dla większych testów nie mieścił się on już w założonych ograniczeniach czasowych — pozwalał uzyskać około 80% punktów.

Implementacja znajduje się w plikach `pods3.cpp` i `pods6.pas`.

Rozwiązania niepoprawne

Większość rozwiązań niepoprawnych polega na losowaniu dużego zbioru połowień i próbie „złożenia” z nich możliwie dobrego rozwiązania, niekoniecznie optymalnego. Ze względu na stosunkowo małą przestrzeń stanów, niektóre z algorytmów mogą trafić na rozwiązanie optymalne i podać dobrą odpowiedź.

Algorytm 4.1. Losujemy wiele podziałów i dla każdego z nich sprawdzamy, czy jest to połowienie. Jeśli tak, to obliczamy liczbę krawędzi pomiędzy połówkami grafu. Wypisujemy ten spośród wylosowanych podziałów, który ma najmniejszą liczbę krawędzi. Jest to algorytm bardzo podobny do algorytmu 3.2 — zamiast sprawdzać przestrzeń podziałów w uporządkowany sposób, testujemy ją losowo.

Takie rozwiązanie pozwalało uzyskać 0–20% punktów. Jego implementacja znajduje się w pliku `podb1.cpp`.

Algorytm 4.2. To rozwiązanie opiera się na obserwacji, że fragment grafu tworzący połówkę w optymalnym połowieniu powinien być jak najbardziej zagęszczony, czyli jak najbardziej przypominać klikę. Dlatego w algorytmie najpierw losuje się wierzchołek, a następnie dołącza do niego inne wierzchołki zachłannie tak, aby w każdym ruchu maksymalizować liczbę krawędzi pomiędzy nowo dodanym a poprzednio wybranymi. Jeśli istnieje kilka możliwości optymalnych ruchów, to algorytm wybiera losowo jedną z nich. Zauważmy, że w algorytmie zakłada się, że jeśli graf jest spójny, to co najmniej jedna część połowienia jest spójna. Tak być nie musi, na co kontrprzykładami są wszystkie testy oznaczone literką `b`. Stąd algorytm, jako błędny, nie pozwala zdobyć żadnych punktów.

Implementacja znajduje się w pliku `podb2.cpp`.

Algorytm 4.3. Jest to drobna modyfikacja algorytmu 4.2. Tym razem, zamiast zachłannie budować gęsty podgraf, dobieramy wierzchołki tak, aby w każdym kroku minimalizować liczbę krawędzi pomiędzy wierzchołkami wybranymi i pozostałymi. Podstawową wadą tego algorytmu jest to, że ma tendencję do dobierania najpierw wierzchołków o małych stopniach, a dopiero potem o większych, co może już na początku przekreślić optymalność rozwiązania. Większość kontrprzykładów na heurystykę 4.2 jest również kontrprzykładami na ten pomysł, ale w trzech przypadkach to testy z literką `e` obnażają jego błędy. Implementacja znajduje się w pliku `podb3.cpp`. Program nie dostaje żadnych punktów.

Algorytm 4.4. W algorytmie losuje się połowienie i poprawia się je dopóty, dopóki jest to możliwe. Dla bieżącego połowienia sprawdza się, czy istnieją dwa wierzchołki z różnych połówek, których wymiana zmniejsza liczbę krawędzi pomiędzy połówkami. Jeśli tak, to wierzchołki zostają wymienione, w przeciwnym razie uznaje się rozwiązanie za lokalnie optymalne i losuje kolejne. Każdą poprawkę realizujemy w czasie $O(|V|^2)$, więc przy każdym losowaniu wykonujemy operacje o złożoności $O(|E||V|^2)$. Jako wyjście wypisujemy najlepsze znalezione połowienie. Dość dobrymi kontrprzykładami, dla których ta heurystyka zawodzi, są dopełnienia prostych grafów takich jak cykl, ścieżka itp. oraz rzadkie grafy strukturalne, najlepiej o dużej liczbie wierzchołków o stopniach parzystych. Algorytm ten pozwalał uzyskać około 20% punktów.

Implementacja znajduje się w pliku `podb4.cpp`.

Algorytm 4.5. Jest to drobna modyfikacja algorytmu 4.4. Zamiast wybierać dowolną parę wierzchołków, wybieramy zawsze tę, której zamiana najbardziej zmniejsza liczbę krawędzi między połówkami. W rzeczywistości zmniejsza to losowość algorytmu 4.4, co powoduje, że heurystyka ta daje nieznacznie gorsze wyniki.

Implementacja znajduje się w pliku `podb5.cpp`. Algorytm pozwalał uzyskać około 20% punktów.

Testy

Przygotowano następujące testy:

Nazwa	n	Opis
pod1abcde.in	2, 18, 20	mała grupa poprawnościowa
pod2abcd.in	18, 20	mała grupa poprawnościowa
pod3abcde.in	18, 20	mała grupa poprawnościowa
pod4abc.in	22, 24	średnia grupa poprawnościowa
pod5abc.in	24	średnia grupa poprawnościowa
pod6abc.in	24	średnia grupa poprawnościowo-wydajnościowa
pod7abcde.in	24	średnia grupa poprawnościowo-wydajnościowa
pod8abcde.in	26	duża grupa poprawnościowo-wydajnościowa
pod9abcd.in	26	duża grupa poprawnościowo-wydajnościowa
pod10abc.in	26	duża grupa poprawnościowo-wydajnościowa

Testy `a` są ogólnymi testami poprawnościowymi, sprawdzającymi błędy implementacyjne. Nie są kontrprzykładami na żadną konkretną heurystykę. Są w zasadzie strukturalnymi testami z elementami losowości, raczej z grafami rzadkimi.

Testy `b` stanowią kontrprzykłady dla rozwiązań niepoprawnych typu 2, jak i innych wymyślonych przez zawodników. Są „złośliwymi" testami z małą liczbą połowień optymalnych, przy czym we wszystkich połowieniach optymalnych mimo spójności całego grafu, podgrafy indukowane przez połówki są niespójne. Sprawia to, że heurystyka 2 generuje niepoprawne rozwiązania. Testy te stanowią raczej grafy rzadkie.

Testy `c` i `d` są testami poprawnościowo-wydajnościowymi. Są tworzone na zasadzie znalezienia dopełnienia pewnego prostego grafu, np. cyklu, ścieżki, gwiazdy itp. Są to grafy gęste, dzięki czemu testują szybkość działania programu. Ponadto takie grafy często stanowią dobre kontrprzykłady na heurystyki typu 4 i 5.

Testy `e` stanowią (oprócz 1*e*) kontrprzykłady dla heurystyki typu 3. Są to zazwyczaj grafy, w których wierzchołki o małych stopniach nie mogą należeć do tej samej połówki grafu. W momencie, gdy jest klika wierzchołków o małych stopniach, a reszta ma duże stopnie, heurystyki typu 3 mają tendencję do dobierania na początku wierzchołków o małych stopniach, co z reguły nie jest optymalne.

Testy są podzielone na 10 grup. Grupy od 1. do 3. zawierają tylko testy z $n \leqslant 20$ — powinny je przejść również rozwiązania nieefektywne. Na testach od 4. do 10. rozwiązania efektywne mieściły się w $\frac{1}{7}$–$\frac{1}{4}$ limitu czasowego, w zależności od implementacji i języka programowania. Rozwiązania nieefektywne tracą punkty za przekroczenie połowy limitu czasowego lub nie kończą działania w określonym czasie. Grupy od 4. do 7. zawierają testy o $n = 22, 24$, zaś grupy od 8. do 10. dostępne tylko dla algorytmów efektywnych $n = 26$.

W praktyce okazało się, że na ocenę rozwiązań miały wpływ także drobne różnice implementacyjne, wybór języka programowania itp.