Zajęcia 21

Temat: Grafy
Czas trwania: 2x45 min
Cel zajęć:

projektuje i programuje proste problemy z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje warunkowe, instrukcje iteracyjne, tablice, rekurencję, pisze własne funkcje rekurencyjne, zbiory i operacje na zbiorach, struktury danych, biblioteka STL, grafy nieskierowane, testuje poprawność programów dla różnych danych, posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

Efekty:

- umie uruchomić potrzebne oprogramowanie,
- umie napisać program z wykorzystaniem struktury danych – graf nieskierowany, podgrafy, wierzchołek, krawędzie, ścieżki,
- zna metody przechodzenia grafów,
- zna strukturę Find-Union,

Formy i metody pracy: praca samodzielna, omówienie, pokaz, wykład

Zadania do wykonania na zajęciach	Treści programowe
1. Dwukolorowanie	M.5, P.2.18, A.3.7, A.3.8, A.4.3
2. Prehistoria	M.5, P.2.18, A.3.7, A. 3.8, A.4.3, A.4.5

Materiały do zajęć:

https://www.main2.edu.pl/main2/courses/show/7/26/

Zadania do wykonania w domu:

Gildie

https://szkopul.edu.pl/problemset/problem/Oys6jiVOIap59lYCHRwDMbNT/site/

Żabka Bajtozja

https://szkopul.edu.pl/problemset/problem/Pbxxhq_YBgWCAXYvKCfJb_HA/site/

ZADANIA I ROZWIĄZANIA

Zadanie 1. Dwukolorowanie

Dostępna pamięć: 32MB

Twierdzenie o czterech barwach mówi, że każdą mapę przedstawioną na płaszczyźnie można pokolorować z użyciem czterech kolorów w taki sposób, żeby żadne dwa sąsiadujące obszary nie były pomalowane na ten sam kolor. Problem ten był otwarty przez ponad sto lat i został udowodniony dopiero w roku 1976 za pomocą komputera.

Twoim zadaniem jest rozwiązanie prostszego problemu. Sprawdź, czy dany graf (spójny, nieskierowany i bez pętli) jest dwukolorowalny, tzn. czy jego wierzchołki mogą być pomalowane na kolory czerwony i czarny w taki sposób, aby sąsiadujące ze sobą wierzchołki nigdy nie były tego samego koloru.

Wejście

Dane wejściowe składają się z pewnej liczby zestawów testowych. W pierwszym wierszu każdego zestawu znajduje się liczba wierzchołków n ($1 \le n \le 100\,000$). Etykietą każdego wierzchołka jest liczba z zakresu od 0 do n–1. Drugi wiersz zawiera liczbę krawędzi k ($1 \le k \le 1\,000\,000$). Każdy z kolejnych k wierszy zawiera numery dwóch wierzchołków – numery te opisują krawędź.

Wyjście

Sprawdź, czy graf jest dwukolorowalny, i wypisz wynik: TAK – jeśli graf jest dwukolorowalny, lub NIE – w przeciwnym wypadku.

Przykład

Wejście	Wejście
3 3 0 1 2 0 1 2 Wyjście NIE	9 8 0 1 0 2 0 3 0 4 4 5 4 6 4 7 4 8 Wyjście TAK

Rozwiązanie

Zadanie rozpoczyna cykl zajęć związanych z zadaniami grafowymi.

W jaki sposób reprezentować graf w pamięci komputera? W wielu zadaniach będzie używane podobne rozwiązanie jak w zadaniu Wodzirej z zajęć nr 18.

Do zapamiętywania ścieżek pomiędzy wierzchołkami wykorzystamy tablicę dynamiczną (można wykorzystać `vector <int>` z biblioteki STL). Poniżej kod w C++.

```
//lista sąsiedztwa
vector <int> V[100001];
odwiedzony[1000001];
cin >> n >> k;
//dla wszystkich par wierzchołków
for (int i=0; i<m; i++) {
```

```
  //wczytaj numery wierzchołków
  // i dodaj je naprzemiennie do listsąsiedzwa
  cin >> u >> v;
  V[u].push_back(v); //dodaj do listy znajomych u znajomego v
  V[v].push_back(u); //dodaj do listy znajomych v znajomego u
}
```

Teraz musimy pokolorować graf na dwa kolory. W tym celu użyjemy tablicy `odwiedzony[]`, która będzie przechowywała informację o stanie wierzchołka: 0 – nieodwiedzony, 1 lub 2 – kolor wierzchołka.

W zadaniu wykorzystamy metodę przeszukiwania grafu wszerz. Zaczniemy przeglądanie wierzchołków od dowolnego z nich (na przykład pierwszego) i nadamy mu kolor 1. Odwiedzimy jego sąsiadów. Każdy nieodwiedzony zostanie dodany do kolejki do sprawdzenia w przyszłości oraz nadamy mu kolor przeciwny do wierzchołka, z którego do niego dotarliśmy (1 lub 2). Co w przypadku, kiedy trafimy na wierzchołek już odwiedzony? Jeżeli jego kolor jest taki sam, jak wierzchołka, z którego przyszliśmy, nie jest możliwe dwukolorowanie grafu. W przeciwnym wypadku sprawdzamy dalej.

Poniżej uproszczony algorytm:

```
kolejka Q
BFS (w)
  odwiedzony[w] ← 1
  Q.dodaj(w)
  dopóki (Q nie jest pusta)
    w ← Q.zdejmij_pierwszy_wierzchołek()
    dla wszystkich sąsiadów v wierzchołka w
      jeżeli (odwiedzony[v] = 0)
        odwiedzony[v] ← 1+odwiedzony[w] mod 2 //kolor przeciwny do w
        Q.dodaj(v)
      przeciwnie
        jeżeli (odwiedzony[v] = odwiedzony[w])
          zwróć FAŁSZ i zakończ funkcję
  zwróć PRAWDA // przeszukaliśmy graf i nie znaleźliśmy kolizji
```

W głównej części programu (po wcześniejszym wczytaniu struktury grafu) wystarczy wywołać funkcję BFS i sprawdzić zwracany przez nią wynik:

```
jeżeli (BFS (1) = PRAWDA)
  wypisz TAK
przeciwnie
  wypisz NIE
```

Zadanie dla uczniów: Jaka jest złożoność obliczeniowa tego programu (przeanalizuj przede wszystkim, ile razy zostanie odwiedzony każdy z wierzchołków).

Zadanie 2. Prehistoria

Limit pamięci: 64 MB

Na dziewiczych, zamieszkałych przez dinozaury, mamuty, wiewiórki biegające za żołędziem i inne dziwne stwory terenach rozciąga się kraina zwana przez jej pierwotnych mieszkańców

Bajtolandią. Na owym obszarze żyje *n* plemion. Miedzy niektórymi z nich są dwukierunkowe drogi, których jest w sumie *m*. Wodzowie postanowili połączyć ich plemiona w sojusze, aby łatwiej było im przetrwać. Zasada łączenia jest prosta: plemię *a* zawiąże sojusz z plemieniem *b* jeśli:

- istnieje droga łącząca plemię *a* i plemię *b*,

lub

- istnieje takie plemię *c*, że zarówno plemię *a* i plemię *b* mają z nim sojusz.

Proces zawierania sojuszy trwa tak długo, aż nie da się zawrzeć żadnego innego.

Ludzie zamieszkujący te tereny są prymitywni, aczkolwiek bardzo inteligentni, więc zgadali się i założyli Koło Informatyczne Bajtolandii (KIB). Zasada przyjęcia do KIBu jest prosta. Obecni członkowie zadają kandydatowi *q* pytań o treści "Czy plemię *p* jest w sojuszu z plemieniem *k*?". Jaskiniowcy mają już swoje komputery stworzone z kamieni oraz śladowych ilości drewna, lecz mimo to odpowiadanie na owe pytania sprawia im trudność. Napisz odpowiedni program i pomóż im dostać się do KIBu.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia ilość plemion, dróg, opisy tych dróg, ilość pytań KIBu, oraz ich opisy.
- wyliczy odpowiedzi na zapytania,
- wypisze wynik na standardowe wyjście.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite *n* i *m* ($1 \le n, m \le 1000000$) oznaczające odpowiednio ilość plemion i dróg w Bajtolandii. W następnych m wierszach znajdują się opisy tych dróg, składające się z dwóch liczb całkowitych *a* i *b* ($1 \le a, b \le n$, $a \ne b$) oznaczające, że plemię *a* i plemię *b* łączy droga. W następnym wierszu znajduje się jedna liczba całkowita *q* ($1 \le q \le 1000000$), a w następnych *q* wierszach po dwie liczby całkowite *p* i *k* ($1 \le p, k \le n$, $p \ne k$), oznaczające pytanie KIBu. W testach wartych około 40% punktów zachodzi warunek ($1 \le n, m, q \le 1000$).

Wyjście

Twój program powinien wypisać na wyjście *q* wierszy, w *i*-tym z nich powinno znajdować się pojedyncze słowo "TAK" (bez cudzysłowów), jeśli odpowiedź na *i*-te pytanie KIBu jest twierdząca, lub "NIE" w przeciwnym wypadku.

Przykład

Wejście	Wyjście
8 6 1 2 5 7 3 2 5 8 1 3 2 4 4	TAK NIE TAK NIE

1 4 6 7 7 5 3 8	

Rozwiązanie

Reprezentacja grafu – lista sąsiedztwa (jak w zadaniu Dwukolorowanie). Aby dowiedzieć, się, które plemiona są ze sobą w sojuszu, będziemy przeglądać graf od dowolnego wierzchołka. Wszyscy sąsiedzi badanego wierzchołka oraz ich sąsiedzi będą częścią jednego sojuszu. Każde wywołanie przeszukiwania grafu odnajdzie nam jeden sojusz. Zamiast kolorować wierzchołki na dwa kolory będziemy w tablicy `odwiedzony[]` przechowywać numer sojuszu, do którego należy nasze plemię.

```
BFS (w, nr_sojuszu)
  kolejka Q
  Q.dodaj(w)
  dopóki (Q nie jest pusta)
    w ← Q.zdejmij_pierwszy_wierzchołek()
    odwiedzony[w] ← nr_sojuszu
    dla wszystkich sąsiadów v wierzchołka w
      jeżeli (odwiedzony[v] = 0)
        odwiedzony[v] ← odwiedzony[w] + 1
        Q.dodaj(v)
```

W głównej części programu (po wcześniejszym wczytaniu struktury grafu) każde plemię przydzielimy do sojuszu:

```
liczba_sojuszy ← 0
dla i=1,2,…,n wykonaj
  jeżeli (odwiedzony[i]=0)
    liczba_sojuszy ← liczba_sojuszy + 1
    BFS(i, liczba_sojuszy)
```

Teraz wystarczy dla q zapytań sprawdzić, czy plemiona p oraz k zostały oznaczone tym samym numerem sojuszu:

```
wczytaj q
dla i=1,2,…,q wykonaj
wczytaj p, k
jeżeli (odwiedzony[p]=odwiedzony[k])
  wypisz TAK
przeciwnie
  wypisz NIE
```