Zajęcia 22

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, wierzchołek, krawędzie, ścieżki,
- zna metody przechodzenia grafów, cykle

Formy i metody pracy: praca samodzielna, omówienie, dyskusja

Zadania do wykonania na zajęciach	Treści programowe
1. Mysz w labiryncie	M.5, P.2.18, A.3.7, A.4.3
2. Grand Prix Bajtocji	M.5, P.2.18, A.3.7, A.4.3

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

Zadania do wykonania w domu:

Szpiedzy

https://szkopul.edu.pl/problemset/problem/RiTxbbjmgXZ84FuD9qo15Aq8/site/

Nadajniki

https://szkopul.edu.pl/problemset/problem/sKmyIHBMNi9EV3WO6GQ4xoFt/site/

ZADANIA I ROZWIĄZANIA

Zadanie 1. Mysz w labiryncie

Dostępna pamięć: 32MB

Do labiryntu trafiła mysz. Pomóż jej odnaleźć wyjście!

Dane:

W pierwszej linii wejścia znajdują się dwie liczby całkowite n i m (2 <= n, m <= 1000) oznaczające rozmiar labiryntu: n kolumn i m wierszy. W kolejnych liniach znajdują się znaki oznaczające kolejno:

- - możliwość przejścia (korytarz);
- x - brak przejścia (ściana);
- o - początkowe położenie myszy;
- w - wyjście (końcowe położenie myszy).

Wynik

Jedna liczba całkowita oznaczająca ilość pół odwiedzonych przez mysz, lub słowo „NIE” – jeżeli taka ścieżka nie istnieje.

Przykładowe dane:

Wejście	Wyjście
8 8 w--x-x-- xx-x-x-- ---x---- --xxxx-- --x--x-- -----x-o xxxx---- ------x-	17

Rozwiązanie

Graf w tym zadaniu do kwadratowa siatka – dwuwymiarowa tablica `mapa[][]`. Z każdego pola o współrzędnych (x,y) możemy przejść do czterech pól: na prawo i na lewo oraz w górę i w dół. Ze względu na konieczność znalezienia najkrótszej ścieżki między dwoma polami rozwiązanie zadania wymaga użycia algorytmu BFS. W kolejce wierzchołków do odwiedzenia będziemy przechowywali współrzędne kolejnych pól do odwiedzenia (punkty p składające się z dwóch pól: x i y). Wszystkie niedostępne pola na mapie oznaczymy dowolną liczbą (w naszym algorytmie -1). Każde odwiedzone pole będzie zawierało informację o odległości od pola startowego, zaś pola nieodwiedzone – wartość 0. Dodatkowo (aby ułatwić sobie pilnowanie brzegów planszy) w kolumnach i wierszach 0 oraz n+1 ustawimy „ścianę” (wartownika, wartości -1).

Poniżej uproszczony algorytm:

```
kolejka Q
mapa[][]
punkt p, pom
BFS (x,y)
  dla i=0,1,2,…,n+1
    mapa[0][i] ← mapa[n+1][i] ← mapa[i][0] ← mapa[i][n+1] ← -1
  mapa[x][y] ← 1
  p.x ← 1, p.y ← 1
  Q.dodaj(p)
  dopóki (Q nie jest pusta)
    p ← Q.zdejmij_pierwszy_punkt()
```

```
jeżeli (mapa[x-1][y]=0)
  mapa[x-1][y] ← mapa[x][y]+1 //zwiększamy odległość o 1
  pom.x ← p.x-1, pom.y ← pom.y
  Q.dodaj(pom)
jeżeli (mapa[x+1][y]=0)
  mapa[x+1][y] ← mapa[x][y]+1 //zwiększamy odległość o 1
  pom.x ← p.x+1, pom.y ← pom.y
  Q.dodaj(pom)
jeżeli (mapa[x][y-1]=0)
  mapa[x][y-1] ← mapa[x][y]+1 //zwiększamy odległość o 1
  pom.x ← p.x, pom.y ← pom.y-1
  Q.dodaj(pom)
jeżeli (mapa[x][y+1]=0)
  mapa[x][y+1] ← mapa[x][y]+1 //zwiększamy odległość o 1
  pom.x ← p.x, pom.y ← pom.y+1
  Q.dodaj(pom)
```

W głównej części programu poza wczytaniem mapy musimy wywołać algorytm BFS dla punktu startowego myszy, a następnie sprawdzić wartość w punkcie wyjścia (dla 0 mysz nigdy nie dotrarła, w przeciwnym wypadku w polu `mapa[x_wyjścia][y_wyjścia]` znajdziemy minimalną odległość od wejścia.

Zadanie 2. Grand Prix Bajtocji

Dostępna pamięć: 64 MB

W Bajtocji wybudowano nowoczesny tor wyścigowy. Składa się on połączonej ze sobą sieci dróg, które mogą się łączyć ze sobą wyłącznie na skrzyżowaniach. Pomiędzy każdą parą skrzyżowań zaprojektowana została dokładnie jedna droga. Drogi (i skrzyżowania) są na tyle szerokie, że można na nich wygodnie wyprzedzać inne samochody (nawet jeżdżąc w dwóch różnych kierunkach), ale nie można na nich zawracać.

Aby urozmaicić wyścigi właściciele toru starają się umieścić pole startu / mety na różnych skrzyżowaniach. Zastanawiają się teraz, czy w ogóle jest możliwe wytyczenie takiej zamkniętej trasy.

Wejście

Skrzyżowania ponumerowane są kolejnymi liczbami naturalnymi od 1 do n ($1 \le n \le 1000$).

Pierwszy wiersz danych zawiera dwie liczby naturalne n (liczba skrzyżowań) oraz k – liczba dróg ($1 \le k \le 10^6$). Kolejne k wierszy zawiera po dwie oddzielone pojedynczym odstępem liczby naturalne u i w oznaczające numery skrzyżowań połączonych drogą ($1 \le u, w \le 1000$, $u \ne w$).

Wyjście

Program powinien wypisać słowo TAK, jeśli można wytyczyć trasę o własnościach opisanych w temacie, lub słowo NIE – w przeciwnym przypadku.

Przykład

Wejście	Wejście
4 3 1 2 1 3 2 4 Wyjście NIE	4 4 1 2 1 3 1 4 2 4 Wyjście TAK

Rozwiązanie

Reprezentacja grafu – lista sąsiedztwa. Zadanie sprowadza się jedynie do sprawdzenia, czy graf reprezentujący drogi zawiera jakikolwiek cykl. Jednym ze sposobów odnalezienia takiego cyklu jest przeszukiwanie grafu (wykorzystamy w naszym wypadku ponownie algorytm przeszukiwania grafu wszerz – BFS). Zmiana standardowego algorytmu, którą wprowadzimy, wymaga sprawdzania, czy w przypadku trafienia na wierzchołek już odwiedzony jest to inny wierzchołek niż ten, z którego przyszliśmy. W tablicy `odwiedzony[]` będziemy przechowywać dodatkowo numer wierzchołka, z którego przyszliśmy do danego wierzchołka.

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] ← w //przyszliśmy z wierzchołka w
        Q.dodaj(v)
      przeciwnie
        jeżeli (odwiedzony[w] = w)
          zwróć PRAWDA i zakończ funkcję
  zwróć FAŁSZ // przeszukaliśmy graf i nie znaleźliśmy cyklu
```