Kraków, 11.02.2015

Implementacja grafów oparta na liście sąsiedztwa

Dokumentacja

Maciej Michalec
Projekt z przedmiotu **Algorytmy i struktury danych z językiem Python**

1. Wprowadzenie

Graf to struktura, której zastosowanie we współczesnej informatyce jest niezwykle rozległe. Chociaż sama jego idea oraz definicja są bardzo proste, to nawet tak na pozór nieskomplikowane zagadnienie jak sposób reprezentacji tej abstrakcyjnej struktury danych znajduje wiele rozwiązań. Zbiór wierzchołków i łączących ich krawędzi można przedstawić między innymi za pomocą rysunku, macierzy sąsiedztwa, macierzy incydencji oraz listy sąsiedztwa. W tym projekcie przedstawiam implementację tej ostatniej koncepcji.

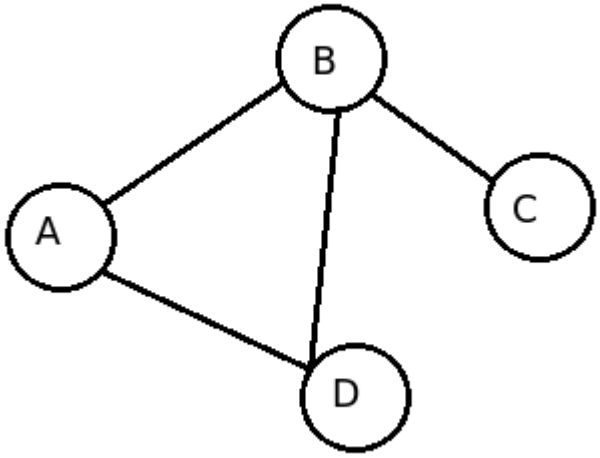
1.1 Pojęcia wstępne

Wierzchołek sąsiadujący, sąsiad - dwa wierzchołki grafu sąsiadują ze sobą, jeśli istnieje krawędź pomiędzy nimi

Graf skierowany – graf, którego krawędzie można wyobrazić sobie za pomocą jednokierunkowych strzałek

1.2 Przykład

Dany mamy nieskierowany graf w postaci rysunku:



Jego lista sąsiedztwa będzie wyglądała następująco:
A: B, D
B: A, C, D

C: B
D: A, B

Możemy przy tym przyjąć, że kolejność sąsiadów danego wierzchołka jest na takiej liście dowolna – mogą być one w jakiś sposób uporządkowane (na przykład alfabetycznie, jeśli wierzchołki są etykietowane literami, bądź niemalejąco jeżeli wierzchołki identyfikujemy za pomocą liczb), ale na ogół taka funkcjonalność nie jest potrzebna w implementacji – sortowanie list sąsiadów sprowadza się jedynie do potrzebnego kosztu czasowego i pamięciowego. Oznacza to, że równie dobrze możemy powyższy graf przedstawić jako listę sąsiedztwa:

A: D, B
B: A, D, C
C: B
D: A, B

2. Implementacja – spojrzenie techniczne

Projekt został zrealizowany z wykorzystaniem języka Python w wersji 2.7.3. Do zarządzania kodem źródłowym użyty został system kontroli wersji Git, a repozytorium projektu dostępne jest w serwisie GitHub.com.

2.1 Pliki

W skład aplikacji wchodzą pliki:

- *node.py* – klasa reprezentująca wierzchołek grafu
- *graph.py* – klasa reprezentująca graf
- *main.py* – niewielki skrypt udostępniający interfejs użytkownika

2.2 Wierzchołek

Najbardziej podstawowymi elementami grafu są wierzchołki. W niniejszym programie za ich implementację odpowiada osobna klasa *Node*. Każda z jej instancji zawiera dane (liczby, etykiety wierzchołków) przechowywane w atrybucie *data* oraz listę wierzchołków sąsiadujących jako atrybut *neighbours*. To ostatnie pole zostało opakowane w funkcje

pozwalające na dodawanie nowych (*add_neighbour)* sąsiadów do listy określonego wierzchołka oraz pobieranie już wpisanych wcześniej sąsiadów (*get_neighbour*). Funkcje sprawdzają poprawność argumentów – indeksów oraz typów, a także nie dopuszczają do wstawiania na listę duplikatów.

2.3 Graf

Sam graf został zaimplementowany w klasie *Graph*. Utworzenie obiektu tego typu wymaga określenia rozmiaru grafu już przy inicjalizacji. Tworzona jest wtedy lista wierzchołków etykietowanych liczbami od *0 do N-1*. Dostęp do konkretnego węzła można uzyskać za pomocą funkcji *get_node*, która jako argument przyjmuje jego indeks (i sprawdza jego poprawność).

Ważną rolę w procesie tworzenia konkretnego grafu pełni funkcja *add_edge*, która pozwala na utworzenie krawędzi łączącej dwa (istniejące już) wierzchołki. Jej parametrami są indeksy węzłów (a nie same węzły!), ich poprawność jest kontrolowana, a w razie błędu wyrzucany jest odpowiedni wyjątek. Ponieważ *Graph* implementuje graf nieskierowany, to podanie jako argumentów np. indeksów (0, 3) spowoduje, że węzeł o indeksie *3* pojawi się na liście sąsiadów wierzchołka *0*, a wierzchołek *0* na liście sąsiadów węzła *3*.

Dodanych, za pomocą wyżej opisanej funkcji, krawędzi nie można usuwać pojedynczo. Czyszczenie grafu odbywa się dzięki metodzie *clear()*.

Za właściwe wyświetlanie uzupełnionego grafu odpowiada implementacja funkcji *__str__()*. Sposób formatowania listy sąsiedztwa wygląda tak, jak w podanym w rozdziale *1.2* przykładzie.

```
Lista sasiedztwa grafu:
0: 1, 2
1: 0, 3, 2
2: 0, 1
3: 1, 4
4: 3
```

2.4 Przeszukiwanie grafu

W programie zostały zaimplementowane dwa najpopularniejsze algorytmy przeszukiwania grafu: w głąb oraz wszerz.

2.4.1 Przeszukiwanie w głąb

Algorytm DFS (ang. *Depth-first search*) polega na badaniu wszystkich krawędzi wychodzących z podanego wierzchołka. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka algorytm powraca do wierzchołka, z którego dany wierzchołek został odwiedzony[0].

Złożoność czasowa tego algorytmu to O(|V|+|E|), a złożoność pamięciowa to O(h), gdzie *h* oznacza długość najdłuższej prostej ścieżki, *V* liczbę wierzchołków, a *E* liczbę krawędzi grafu.

Aplikacja zawiera dwie różne implementacje tego algorytmu: rekurencyjną oraz z wykorzystaniem stosu. Ta druga działa w oparciu o tworzoną tablicę (listę) odwiedzonych już wierzchołków, natomiast algorytm rekurencyjny przekazuje w swoich kolejnych wywołaniach aktualnie obliczoną ścieżkę.

2.4.2 Przeszukiwanie wszerz

Algorytm BFS (ang. *Breadth-first search*) przechodzi graf rozpoczynając od zadanego wierzchołka *s* i polega na odwiedzeniu wszystkich osiągalnych z niego wierzchołków. Wynikiem działania algorytmu jest drzewo przeszukiwania wszerz o korzeniu w s, zawierające wszystkie wierzchołki osiągalne z s. Do każdego z tych wierzchołków prowadzi dokładnie jedna ścieżka z s, która jest jednocześnie najkrótszą ścieżką w grafie wejściowym[1].

Zarówno złożoność pamięciowa, jak i czasowa, algorytmu BFS to O(|V|+|E|), gdzie *V* oznacza liczbę wierzchołków, a *E* liczbę krawędzi grafu.

W odróżnieniu od algorytmu DFS, nie używa się stosu, ale kolejki FIFO. Również tutaj używa się natomiast listy wierzchołków już odwiedzonych (czasami jest ona także nazywana tablicą kolorowania, bowiem odwiedzone już wierzchołki można rozróżniać za pomocą kolorowania je

np. na biało/czarno/szaro).

3. Interfejs użytkownika

Uruchamiając skrypt *main.py* użytkownik ma możliwość zdefiniowania własnego grafu nieskierowanego w postaci listy sąsiedztwa, a następnie przeszukania go wszerz lub w głąb.

3.1 Argumenty wiersza poleceń

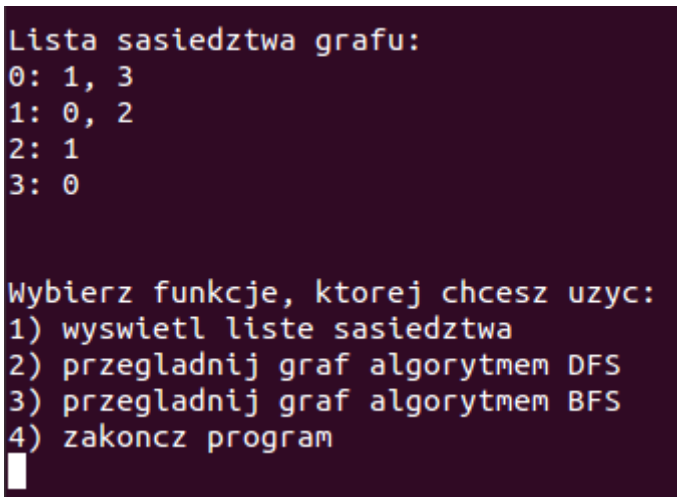
Aby uzyskać dostęp do głównej funkcjonalności programu należy uruchomić *main.py* wykonanym w terminalu poleceniem:
python main.py
lub też (w systemie Linux) nadać plikowi uprawnienia do wykonywania (*+x*) i wpisać w konsoli:
./main.py
Aplikacja udostępnia również dwa argumenty, które można podawać przy uruchamianiu:

- --*help*, -*h* – wyświetlona zostaje pomoc programu
- --*version*, -*v* – wyświetlona zostaje numer wersji aplikacji

3.2 Obsługa

Po uruchomieniu pliku głównego najpierw należy podać liczbę wierzchołków definiowanego grafu oraz liczbę jego krawędzi. Następnie program prosi o opisanie każdej z krawędzi. Należy tego dokonać wpisując indeks pierwszego wierzchołka, przechodząc do nowej linii (naciśnięcie przycisku *enter*) i wreszcie podając indeks drugiego wierzchołka oraz zatwierdzając go *enterem*. Należy przy tym pamiętać, że indeksy wierzchołków są liczbami w zakresie od *0* do *N-1*, więc określając rozmiar grafu (liczbę węzłów) jako 5, pierwszy węzeł będzie oznaczony numerem *0*, a ostatni liczbą *4*. Prawidłowe dodanie każdej krawędzi jest potwierdzane przez komunikat *Dodany*.

Po zdefiniowaniu grafu program wyświetla listę sąsiedztwa oraz menu pozwalające na wybór funkcji programu (zob. rys. poniżej).



Po wybraniu opcji przeszukiwania grafu (dokonuje się tego przez wpisanie odpowiedniej liczby i zatwierdzenie jej *enterem*), program prosi użytkownika o podanie indeksu wierzchołka, od którego algorytm ma rozpocząć przeglądanie grafu. Za każdym razem w menu dostępna jest również funkcja ponownego wyświetlenia listy sąsiedztwa (numer 1) oraz zakończenia działania aplikacji (numer *4*).

4. Przypisy

- [0] *Przeszukiwanie w głąb*, Wikipedia (http://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b)
- [1] *Przeszukiwanie wszerz*, Wikipedia (http://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz)