algograf

Lab 5 & 6: Grafy przekątniowe

W ramach laboratorium należy zaimplementować rozpoznawanie grafów przekątniowych i przedziałowych oraz rozwiązywanie dla tej klasy grafów pewnych problemów obliczeniowych. Algorytmy, które zostaną do tego wykorzystane są opisane w następującym artykule:

M. Habib, R. McConnell, C. Paul, L. Viennot *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing* (tekst artykułu)

Zadanie 1

Dany jest graf nieskierowany $G = (V,E)$. Graf G nazywamy grafem przekątniowych (*chordal*), jeśli nie istnieje w nim żaden cykl długości większej niż 3, w którym żadne dwa wierzchołki nie są połączone krawędzią nie należącą do cyklu (taki cykl nazywany jest czasem dziurą – *graph hole*).

Zaimplementuj algorytm sprawdzający, czy zadany graf G jest grafem przekątniowym. Należy wykorzystać w tym celu algorytm LexBFS opisany w dalszej części konspektu, oraz następującą alternatywną definicję grafu przekątniowego:

G jest przekątniowy wtedy i tylko wtedy gdy można uszeregować jego wierzchołki w ciąg $v1$, $v2$, …, vn taki, że każdy wierzchołek vi wraz ze swoimi sąsiadami którzy występują w tym ciągu przed vi tworzą klikę (graf pełny). Takie uporządkowanie wierzchołków nazywamy *kolejnością idealnej eliminacji* (*perfect elimination ordering* – PEO).

Wskazówka

Można pokazać, że dla grafu przekątniowego algorytm LexBFS odwiedza wierzchołki w kolejności PEO. Aby więc sprawdzić, czy graf jest przekątniowy, wystarczy sprawdzić czy ciąg wierzchołków zwrócony przez algorytm LexBFS spełnia definicję PEO (opis jak to zrobić w dalszej części konspektu).

Zadanie 2

Dany jest graf przekątniowy $G = (V,E)$. Zaimplementuj algorytm znajdujący rozmiar największej kliki w G.

Wskazówka

Dla grafu przekątniowego kolejność wierzchołków zwrócona przez LexBFS gwarantuje, że dla każdego wierzchołka *v* zbiór *RN(v)* + *{v}* jest kliką (sekcja "Kolejność idealnej eliminacji" niżej). Niech *K* będzie największą kliką w grafie, a *v* jej wierzchołkiem odwiedzonym przez LexBFS jako ostatni. Wówczas wszystkie pozostałe wierzchołki *K* zostały już odwiedzone i są sąsiadami *v*, zatem *K* jest podzbiorem *RN(v)* + *{v}*. Ponadto, jako że cały zbiór *RN(v)* + *{v}* jest kliką i jest rozmiaru co najmniej takiego jak *K*, to musi być równy *K*, czyli *K* = *RN(v)* + *{v}*. Stąd, w celu znalezienia największej kliki w grafie przekątniowym wystarczy wziąć największy ze zbiorów *RN(v)* + *{v}*.

Zadanie 3

Dany jest nieskierowany graf przekątniowy *G* = *(V,E)*. *Kolorowanie* grafu *G* to przyporządkowanie każdemu wierzchołkowi koloru tak, by wierzchołki sąsiadujące ze sobą miały różne kolory. *Liczba chromatyczna* grafu *G* to minimalna ilość kolorów wymagana do pokolorowania grafu *G*.

Zaimplementuj algorytm znajdujący optymalne (używające minimalnej liczby kolorów) kolorowanie grafu *G* i tym samym obliczający liczbę chromatyczną *G*.

Wskazówka

Kolorowanie grafu można zrealizować przy pomocy prostego algorytmu zachłannego:

```
color = tablica przechowujaca kolory wierzcholkow jako liczby naturalne, poczatkowo same 0

for v in V:
  N = zbior sasiadow v
  used = {color[u] for u in N}
  c = najmniejszy kolor > 0 ktory nie wystepuje w zbiorze used
  color[v] = c
```

W ogólności takie kolorowanie używa większej liczby kolorów, niż jest to konieczne – kolorowanie nie jest optymalne. W przypadku grafów przekątniowych można pokazać, że algorytm ten daje kolorowanie optymalne, jeśli kolorujemy wierzchołki w kolejności idealnej eliminacji (PEO) zwróconej przez LexBFS.

Zadanie 4

Dany jest nieskierowany graf przekątniowy *G* = *(V,E)*. *Pokrycie wierzchołkowe* grafu *G* to zbiór wierzchołków *S* taki, że każda krawędź w *E* jest incydentna z jakimś wierzchołkiem z *S*, tj. co najmniej jeden z jej końców leży w *S*. Zbiór wierzchołków *I* jest *niezależny*, jeśli żadne dwa wierzchołki w *I* nie są połączone krawędzią. Zbiór *I* jest niezależny wtedy i tylko wtedy, gdy jego dopełnienie *V-I* jest pokryciem wierzchołkowym.

Zaimplementuj algorytm znajdujący rozmiar najmniejszego pokrycia wierzchołkowego w zadaym grafie *G*.

Wskazówka

Żeby znaleźć najmniejsze pokrycie wierzchołkowe, wystarczy znaleźć największy zbiór niezależny – szukane pokrycie będzie jego dopełnieniem. Zbiór niezależny można znaleźć przy pomocy prostego algorytmy zachłannego:

```
I = zbior pusty

for v in V:
  N = zbior sasiadow v
  if I oraz N sa rozlaczne:
    dodaj v do I
```

Podobnie jak w przypadku poprzedniego zadania, w ogólności taki algorytm nie znajduje *największego* zbioru niezależnego, ale jest tak w przypadku grafów przekątniowych, jeśli przeglądamy wierzchołki w kolejności **odwrotnej** do kolejności idealnej eliminacji (PEO) zwróconej przez LexBFS.

Algorytmy potrzebne do wykonania zadań

Algorytm LexBFS

Algorytm LexBFS działa podobnie jak zwykły BFS – przegląda wierzchołki grafu wszerz, przy czym kolejność w jakiej wierzchołki są odwiedzane spełnia pewien dodatkowy warunek. O ile BFS przez użycie kolejki jako następny do odwiedzenia wybiera dowolny nieodwiedzony wierzchołek którego poprzednik był odwiedzony najwcześniej, LexBFS dodatkowo rozstrzyga "remisy" porównując dla każdego nieodwiedzonego *v* zbiory wszystkich jego poprzedników (tj. wierzchołków połączonych z *v* które zostały już odwiedzone) i wybiera ten, którego zbiór poprzedników jest "leksykograficznie najmniejszy".

Konkretnie – niech *u*, *v* będą nieodwiedzonymi jeszcze wierzchołkami, *P(u)*, *P(v)* będą ich zbiorami poprzedników. Jeśli najwcześniej odwiedzony wierzchołek z *P(u)* był odwiedzony wcześniej niż najwcześniej odwiedzony wierzchołek z *P(v)*, to *P(u)* jest leksykograficznie mniejszy niż *P(v)*. Jeśli najwcześniej odwiedzone wierzchołki w obu zbiorach są równe, porównujemy drugie najwcześniej odwiedzone itd. Jeśli dojdziemy do momentu w którym jeden ze zbiorów się "skończy", to ten drugi jest leksykograficznie mniejszy.

Realizacja

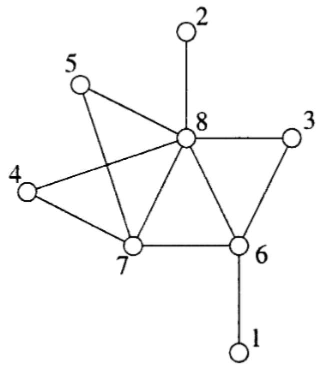
Algorytm odwiedzający wierzchołki w kolejności LexBFS (algorytm 2 z artykułu, lekko zmodyfikowany) działa następująco. Przechowujemy nieodwiedzone wierzchołki w liście zbiorów zbudowanej w taki sposób, że wierzchołki w dalszych zbiorach mają leksykograficznie mniejsze zbiory poprzedników. W każdej iteracji wybieramy do odwiedzenia dowolny wierzchołek *v* z ostatniego zbioru, usuwamy go z tego zbioru i uaktualniamy listę zbiorów.

Aby uaktualnić listę zbiorów, po kolei dla każdego zbioru *X* dzielimy go na dwa podzbiory – *Y*, który zawiera te elementy *X* które są sąsiadami nowo odwiedzonego wierzchołka *v*, oraz *K* – reszta *X*, tj. *K* = *X* - *Y*. Nowo stworzone zbiory wstawiamy do listy zamiast *X* w kolejności *K*, *Y* (czyli wierzchołki sąsiadujące z *v* będą odwiedzone wcześniej).

Na początku działania algorytmu zaczynamy od listy zawierającej jeden element – zbiór wszystkich wierzchołków. Aby zacząć przeglądanie grafu od wybranego wierzchołka *v*, wystarczy zamiast tego zacząć od listy z dwoma elementami – *V* - *{v}* oraz *{v}*.

Przykład

Prześledzimy działanie LexBFS na grafie poniżej (Fig. 5 z podlinkowanego artykułu).



Zacznijmy od wierzchołka 1 – w wyniku jego odwiedzenia pozostałe wierzchołki mają następujące zbiory poprzedników:

```
odwiedzone - 1
6 - {1}
2,3,4,5,7,8 - {}
lista - [{2,3,4,5,7,8}, {6}]
```

Tylko jeden element ma poprzednika, więc jego zbiór poprzedników jest leksykograficznie najmniejszy. Stąd, jako następny odwiedzimy wierzchołek 6:

```
odwiedzone - 1, 6
7,8,3 - {6}
2,4,5 - {}
lista - [{2,4,5}, {7,8,3}]
```

Następny zostanie odwiedzony zatem któryś z wierzchołków 7, 8, 3. Tutaj możemy wybrać dowolny – LexBFS nie narzuca kolejności w przypadku równych zbiorów poprzedników. Wybierzmy np. wierzchołek 8:

```
odwiedzone - 1, 6, 8
7,3    - {6,8}
2,5,4  - {8}
lista - [{2,4,5}, {7,3}]
```

Jako następny wierzchołek odwiedzony zostać musi 7 lub 3. Wybierzmy 3:

```
odwiedzone - 1, 6, 8, 3
7     - {6,8}
2,4,5 - {8}
lista - [{2,4,5}, {7}]
```

Kolejno odwiedzić należy 7:

```
odwiedzone - 1, 6, 8, 3, 7
2 - {8}
5 - {8, 7}
4 - {8, 7}
lista - [{2}, {4,5}]
```

Wierzchołki 2, 4 i 5 mają takiego samego najwcześniej występującego poprzednika, ale tylko 4 i 5 mają drugiego w kolejności poprzednika, ponadto jest on taki sam – stąd, odwiedzić musimy albo 4, albo 5. Wybierzmy 4:

```
odwiedzone - 1, 6, 8, 3, 7, 4
2 - {8}
5 - {8, 7}
lista - [{2}, {5}]
```

Jak wyżej, najwcześniejszy poprzednik jest taki sam, ale 5 ma drugiego poprzednika, zostanie zatem odwiedzony jako pierwszy:

```
odwiedzone - 1, 6, 8, 3, 7, 4, 5
2 - {8}
lista - [{2}]
```

W przypadku ostatniego wierzchołka nie mamy wyboru, ostateczna kolejnośc to

```
1, 6, 8, 3, 7, 4, 5, 2
```

Warto zauważyć, że LexBFS nie ustala kolejności odwiedzania wierzchołków jednoznacznie.

Kolejność idealnej eliminacji (Perfect Elimination Ordering)

Dla danego uporządkowania wierzchołków *O = (v1, v2, ..., vn)* możemy sprawdzić, czy jest to kolejność idealnej eliminacji w następujący sposób (algorytm 3 w podlinkowanym artykule). Niech *RN(v)* będzie zbiorem sąsiadów *v* pojawiających się w *O* wcześniej niż *v*, zaś *parent(v)* niech będzie najpóźniej pojawiającym się elementem *RN(v)*. Z definicji kolejności idealnej eliminacji, zbiór *RN(v) + {v}* powinien być kliką, więc w szczególności *RN(v)* poza *parent(v)* powinien zawierać się w *RN(parent(v))*.

Można pokazać, że jest to również warunek wystarczający. Jeśli *v* jest pierwszym wierzchołkiem dla którego zbiór *RN(v) + {v}* nie jest kliką, to istnieją w *RN(v)* wierzchołki *x* i *y* nie połączone krawędzią. Jako że *parent(v)* jest najpóźniejszym elementem *RN(v)*, co najmniej jeden z nich – powiedzmy, że *x* – musi pojawiać się przed *parent(v)* (drugi albo pojawia się przed *parent(v)*, albo jest to *parent(v)*). Jeśli *x* jest elementem *RN(parent(v))*, to *x*, *y* leżą w *RN(parent(v)) + {parent(v)}* które z założenia jest kliką (*v* jest pierwszym wierzchołkiem który łamie to założenie), więc muszą być połączone – sprzeczność. Stąd, *x* nie jest sąsiadem *parent(v)*, ale jest sąsiadem *v*, a zatem *RN(v) - {parent(v)}* nie jest podzbiorem *RN(parent(v))*.

Żeby sprawdzić więc, czy zadany porządek jest kolejnością idealnej eliminacji, wystarczy dla każdego wierzchołka *v* policzyć *RN(v)* oraz *parent(v)*, i sprawdzić czy zbiór *RN(v) - {parent(v)}* jest zawarty w *RN(parent(v))*.

Pomocne fragmenty kodu

Zbiory w Pythonie

Implementacja zbiorów w bibliotece standardowej Pythona pozwala na proste wykonywanie operacji teoriomnogościowych przy użyciu operatorów.

```
A = set()              # zbiór pusty
B = {1, 2, 3}          # zbiór z elementami 1, 2, 3
C = {2, 4, 5}

if A: ...              # zbiór pusty jest traktowany jako False
if B: ...              # zbiór niepusty - jako True

B | C                  # suma zbiorów - {1, 2, 3, 4, 5}
B & C                  # przecięcie zbiorów - {2}
B - C                  # różnica zbiorów  - {1, 3}

A |= B, A &= B ...     # modyfikacja zbioru w miejscu

2 in B, 5 not in B     # przynależność do zbioru
{1, 3} <= B            # zawieranie - True
```

Więcej przykładów można znaleźć w dokumentacji.

Reprezentacja grafu

Wszystkie grafy używane w tym laboratorium są nieskierowane Bardzo pomocna w implementacji będzie reprezentacja oparta o listy, bądź zbiory sąsiedztwa – w wielu miejscach konieczne jest wykonanie pewnych operacji teoriomnogościowych (suma, przecięcie) z użyciem zbioru sąsiadów danego wierzchołka.

```
class Node:
  def __init__(self, idx):
    self.idx = idx
    self.out = set()              # zbiór sąsiadów

  def connect_to(self, v):
    self.out.add(v)

...

(V, L) = loadWeightedGraph(name)

G = [None] + [Node(i) for i in range(1, V+1)]  # żeby móc indeksować numerem wierzchołka

for (u, v, _) in L:
  G[u].connect_to(v)
  G[v].connect_to(u)
```

Testowanie porządku LexBFS

Jeśli *v1, v2, ..., vn* jest jedną z możliwych kolejności, z jaką odwiedził wierzchołki grafu algorytm LexBFS, to spełniony jest następujący warunek:

Jeśli *k* < *i* < *j* są takie, że *vk* jest sąsiadem *vj*, ale nie jest sąsiadem *vi*, to istnieje również *m* < *k* takie, że *vm* jest sąsiadem *vi*, ale nie jest sąsiadem *vj*.

Można tej własności użyć do skonstruowania prostej funkcji sprawdzającej, czy kolejność zwrócona przez implementację LexBFS jest poprawna:

```
def checkLexBFS(G, vs):
  n = len(G)
  pi = [None] * n
  for i, v in enumerate(vs):
    pi[v] = i

  for i in range(n-1):
    for j in range(i+1, n-1):
      Ni = G[vs[i]].out
      Nj = G[vs[j]].out

      verts = [pi[v] for v in Nj - Ni if pi[v] < i]
```

```
      if verts:
        viable = [pi[v] for v in Ni - Nj]
        if not viable or min(verts) <= min(viable):
          return False
  return True
```

Proponowana kolejność prac

- zaimplementuj algorytm LexBFS
- upewnij się, że działa poprawnie używając sposobu opisanego wyżej – jeśli LexBFS jest zaimplementowany źle, nic innego nie będzie działać poprawnie
- zaimplementuj algorytm sprawdzania, czy kolejność wierzchołków zwrócona przez LexBFS jest *kolejnością idealnej eliminacji* (PEO) – jeśli tak, graf jest przekątniowy
- Zaimplementuj zachłanny algorytm kolorowania wierzchołków odwiedzając je w kolejności zwróconej przez LexBFS
- Zaimplementuj algorytm znajdowania największej kliki w grafie przekątniowym
- Zaimplementuj zachłanny algorytm znajdowania niezależnego zbioru wierzchołków w kolejności odwrotnej do tej zwróconej przez LexBFS – najmniejsze pokrycie wierzchołkowe to zbiór pozostałych wierzchołków
- Zaimplementuj algorytm LexBFS tak, żeby działał w czasie *O(nlogn)* (gdzie *n* to rozmiar danych wejściowych; pierwsza impelmentacja, o ile opiera się na zbiorach i listach z Pythona, działa w czasie *O(n^2logn)* albo *O(n^2)*)
- Zaimplementuj algorytm LexBFS tak, żeby działał w czasie *O(n)* (gdzie *n* to rozmiar danych wejściowych)

Pomocne pliki

W ramach laboratorium należy wykorzystać:

- dimacs.py – wczytywanie grafów, teraz dodatkowo z funkcją do odczytywania rozwiązania z pliku z grafem
- graphs-lab4.zip – grafy testowe do następujących zadań:
  - chordal/ – rozpoznawanie grafów przekątniowych (chordal)
  - maxclique/ – znajdowanie największej kliki
  - coloring/ – kolorowanie minimalną liczbą kolorów (liczba chromatyczna)
  - vcover/ – znajdowanie minimalnego pokrycia wierzchołkowego
  - interval/ – rozpoznawanie grafów przedziałowych