algograf

Lab 3: Spójność krawędziowa

W ramach laboratorium należy zaimplementować algorytmy obliczające spójność krawędziową grafu

Zadanie 1

Dany jest graf nieskierowany *G = (V,E)*. *Spójnością krawędziową* grafu *G* nazywamy minimalną liczbę krawędzi, po których usunięciu graf traci spójność. Przykładowo:

- spójność krawędziowa drzewa = 1
- spójność krawędziowa cyklu = 2
- spójność krawędziowa *n*-kliki = *n-1*

Opracuj i zaimplementuj algorytm obliczający spójność krawędziową zadanego grafu *G*, wykorzystując algorytm Forda-Fulkersona oraz następujący fakt:

(*Tw. Mengera*) Minimalna ilość krawędzi które należy usunąć by zadane wierzchołki *s*, *t* znalazły się w różnych komponentach spójnych jest równa ilości krawędziowo rozłącznych ścieżek pomiędzy *s* i *t*

Wskazówka: jak można zinterpretować ilość krawędziowo rozłącznych ścieżek jako problem maksymalnego przepływu? Proszę wykorzystać kod opracowany w ramach Laboratorium 2.

Zadanie 2

Proszę zaimplementować program obliczający spójność krawędziową grafu nieskierowanego *G* przy użyciu algorytmu Stoera-Wagnera.

Algorytm stworzony w ramach zadania 1 nie jest optymalny (nie sprawdza się zbyt dobrze np. dla dużych klik). Głównym jego składnikiem jest algorytm Forda-Fulkersona zaprojektowany z myślą o znajdowaniu maksymalnych przepływów pomiędzy dwoma określonymi wierzchołkami. Problem znajdowania spójności krawędziowej sprowadza się natomiast do poszukiwania pary wierzchołków, pommiędzy którymi maksymalny przepływ jest najmniejszy, stąd używając algorytmu przeznaczonego do tego problemu dostaniemy rozwiązanie o mniejszej złożoności obliczeniowej.

Implmentacja algorytmu Stoera-Wagnera

Dokładny opis algorytmu znajduje się na stronach powyżej. Tutaj przedstawiamy porady jak go implementować.

Reprezentacja grafu

Graf należy wczytać jak zwyke, korzystając z funkcji `loadWeightedGraph` (nasze grafy są nieskierowane). Tym razem wejściowo wszystkie krawędzie mają wagi 1, ale w trakcie działania algorytmu będą się one zmieniać i należy to uwzględnić:

```
from dimacs import *

(V,L) = loadWeightedGraph( "g1" )        # wczytaj graf
```

```
for (x,y,c) in L:                        # przeglądaj krawędzie z listy
  print( "krawedz miedzy", x, "i", y,"o wadze", c )   # wypisuj
```

Algorytm Stoera-Wagnera opiera się na dwóch głównych operacjach:

- znajdowanie minimalnego przecięcia dla pewnych dwóch wierzchołków (algorytm podobny w działaniu do algorytmu Dijkstry)
- scalaniu wierzchołków

Należy przyjąć reprezentację grafów, która pozwala na obie operacje. W szczególności przydatna będzie reprezentacja przez listy sąsiedztwa. Można ją zrealizować np. jako listę wierzchołków:

```
class Node:
  def __init__(self):
    self.edges = {}    # słownik  mapujący wierzchołki do których są krawędzie na ich wagi

  def addEdge( self, to, weight):
    self.edges[to] = self.edges.get(to,0) + weight  # dodaj krawędź do zadanego wierzchołka
                                                    # o zadanej wadze; a jeśli taka krawędź
                                                    # istnieje, to dodaj do niej wagę

  def delEdge( self, to ):
    del self.edges[to]                              # usuń krawędź do zadanego wierzchołka


G = [ Node() for i in range(V) ]

for (x,y,c) in L:
  G[x].addEdge(y,c)
  G[y].addEdge(x,c)
```

Warto sobie zaimplementować funkcję wypisującą graf, żeby móc łatwo oglądać co się dzieje na małych grafach. Można też zrealizować osobną klasę przechowującą cały graf—będzie to bardziej eleganckie, ale z punktu widzenia ćwiczenia algorytmiki, niekonieczne.

Scalanie wierzchołków

Algorytm Stoera-Wagnera wykorzystuje operację łączenia wierzchołków. Jeśli łaczymy wierzchołek `x` z wierzchołkiem `y` to dla każdego wierzchołka `z`, który jest połączony krawędzią z przynajmniej jednym z nich, mamy teraz krawędź łączącą nowy wierzchołek `xy` z wierzchołkiem `z` o wadze będącej sumą wag krawędzi między `x` i `z` oraz między `y` i `z`. Jeśli występowała jakaś krawędź między `x` i `y` to znika.

Najprostszy sposób implementacji to stworzenie funkcji, która usuwa wszystkie krawędzie np. z wierzchołka `y` i dodaje je do `x`:

```
def mergeVertices( G, x, y ):
  ...
```

Może być przydatne (dla weryfikacji algorytmu) przechowywanie w każdym wierzchołku informacji, czy został "deaktywowany" w wyniku scalania wierzchołków.

Warto także w każdym aktywnym wierzchołku mieć informację jakie wierzchołki zostały z nim scalone (pozwala to odczytać optymalne przecięcie grafu).

Znajdowanie minimalnego przecięcia dla pewnych dwóch wierzchołków (MinimumCutPhase)

Podstawowa operajca w algorytmie Stoera-Wagnera to znalezienie pewnych dwóch wierzchołków `s` i `t` oraz takiego minimalnego przecięcia `C = (S,T)`, że `s` należy do `S` a `t` należy do `T`.

Jest to realizowane przez następującą pętlę:

```
def minimumCutPhase( G ):

  a = dowolny wierzcholek # może to zawsze być wierzchołek numer 1 (lub 0 po przenumerowaniu)
  S = {a}

  while S nie zawiera wszystkich wierzcholkow:
    znajdz taki wierzcholek v, ze suma wag krawedzi z v do
    wierzcholkow w S jest maksymalna

    dolacz v do S (zapamietujac kolejnosc dodawania)

  s = ostatni wierzcholek dodany do S
  t = przedostatni wierzcholek dodany do S

  # tworzone przecięcie jest postaci S = {s}, T = V - {s}
  zapamietaj sume wag krawedzi wychodzacych z s jako potencjalny_wynik

  mergeVertices(G,s,t)

  return potencjalny_wynik
```

Podobieństwo do algorytmu Dijkstry

Powyższy algorytm można efektywnie zrealizować tak samo, jak implementuje się algorytm Dijkstry:

- Dla każdego wierzchołka `v` trzymamy w tablicy aktualną wartość sumy wag krawędzi między wierzchołkami z `S` a `v`
- Utrzymujemy kolejkę priorytetową, w której znajdują się wierzchołki, a priorytetem jest obecna suma wag dochodzących do `S`
- W każdej iteracji:
  - Wyciągamy wierzchołek `v` z kolejki
  - Jeśli `v` był już rozważany to pomijamy go
  - Uaktualniamy sumy wag wszystkich wierzchołków połączonych krawędzią z `v` (każdy wierzchołek, którego sumę wag uaktualniliśmy umieszczamy w kolejce—stąd w kolejce może być wiele kopii tego samego wierzchołka)

Kolejka priorytetowa w Pythonie

Aby zrealizować algorytm podobny do algorytmu Dijkstry, będziemy potrzebować kolejki priorytetowej:

```
from queue import PriorityQueue

Q = PriorityQueue()         # stwórz pustą kolejkę

Q.put( (10,"Henryk"   ) )    # wstaw parę (10, "Henryk") do kolejki (priorytetem jest 10)
Q.put( (5 ,"Hermiona")  )
Q.put( (20,"Harold"   ) )
```

```
Q.get()                   # wyjmij z kolejki (da (5, "Hermiona"))

Q.empty()                 # sprawdza czy kolejka jest pusta
```

Ponieważ potrzebujemy kolejki, która najpierw zwraca elementy o większym priorytecie a nie mniejszym, to należy sumę wag umieszczać ze znakiem ujemnym.

Główny algorytm

Główny algorytm sprowadza się do wykonywania funkcji `minimumCutPhase` aż zostanie tylko jeden wierzchołek. Jako rozwiązanie należy zwrócić minimalny z uzyskanych potencjalnych wyników.

Jeśli przechowujemy listę wierzchołków reprezentowanych przez dany wierzchołek, to możemy także odtworzyć minimalne przecięcie. Wykonanie `minimumCutPhase` , które daje minimalny wynik może w tym celu zwrócić wierzchołek `s` ; minimalne przecięcie tworzą reprezentowane przez niego (w tym momencie) wierzchołki.

Dodatkowe Informacje

Bardziej obszerny opis algorytmu wraz z jego działaniem na konkretnym, przykładowym grafie (pomocne przy implementacji) jest dostępny tutaj.

Proponowana kolejność prac

- zaimplementuj algorytm obliczający spójność krawędziową w oparciu o znajdywanie maksymalnych przepływów
- przetestuj powyższy algorytm na przykładowych grafach
- zaimplementuj algorytm Stoera-Wagnera
    - zaimplementuje reprezentację grafu i jego wczytywanie
    - zaimplementuj wypisywanie grafu
    - zaimplementuj scalanie wierzchołków (i sprawdź czy działa)
    - zaimplementuj funkcję `minimumCutPhase` (i sprawdź czy działa)
    - wstań, zrób trzy przysiady
    - zaimplementuj główny algorytm

Pomocne pliki

W ramach laboratorium należy wykorzystać:

- dimacs.py – wczytywanie grafów (teraz również funkcja do wczytywania grafów skierowanych)
- graphs-lab3.zip – grafy testowe