apto

Lab 7: Szerokość drzewowa i Vertex Cover

W ramach laboratorium należy zaimplementować algorytm obliczający rozmiar minimalnego pokrycia wierzchołkowego w oparciu o programowanie dynamiczne i dekompozycję drzwową grafu.

VertexCover

W problemie VertexCover mamy dany graf *G = (V,E)* oraz jego dekompozycję drzewową *T*. Należy obliczyć rozmiar najmniejszego pokrycia wierzchołkowego, w oparciu o programowanie dynamiczne przebiegające po drzewie dekompozycji.

Algorytm

Dla każdego węzła *y* w drzewie dekompozycji przez *B[y]* rozumiemy jego torbę wierzchołków oryginalnego grafu. Algorytm opiera się na obliczaniu pewnej funkcji *f*. Niech *y* będzie pewnym węzłem w drzewie dekompozycji oraz niech *C* będzie pewnym podzbiorem *B[y]*. Wówczas definiujemy:

```
f(y, C) = minimalny rozmiar pokrycia wierzchołkowego
          dla fragmentu grafu opisywanego przez poddrzewo
          dekompozycji zakorzenione w y, takiego że
          to pokrycie zawiera wszystkie wierzchołki z C i nie
          zawiera wierzchołków z B[y]-C

          lub

          +nieskończoność, jeśli takie pokrycie
          wierzchołkowe nie istnieje
```

Jeśli *r* to korzeń drzewa dekompozycji, to wynikiem naszego algorytmu powinno być:

```
min{ f(y,C) | C jest podzbiorem B[r] }
```

Wartość *f(y, C)* można obliczać następująco. Niech z_1, ..., z_t to dzieci *y* w drzewie dekompozycji (niestety nasze drzewa nie są “fajne”). Stosujemy poniższy wzór oraz programowanie dynamiczne:

```
jeśli C nie stanowi pokrycia wierzchołkowego dla grafu indukowanego w G przez B[y] to:

f(y, C) = +nieskończoność
```

```
  w przeciwnym razie:

  f(y, C ) = |C| +  min( f( z1, D1 )-|B[z1] n C|  | D1 to zbiór wierzchołków zawierający się w
                 +  min( f( z2, D2 )-|B[z2] n C|  | D2 to zbiór wierzchołków zawierający się w
                 ...
                 +  min( f( z_t, D_t )-|B[z_t] n C|  | D_t to zbiór wierzchołków zawierający s
```

Grafy używane w laboratorium:

- graphtw.zip – proszę zwrócić uwagę, że grafy kodowane są minimalnie inaczej niż zwykle. Każdy graf występuje jako plik .gr z opisem grafu oraz plik .tw z opisem dekompozycji drzewowej (dekompozycje były obliczone przy użyciu programów z wyzywania PACE 2017).

Szczegóły techniczne

Wszystkie programy powinny być implementowane w języku Python. Mogą Państwo (i powinni) korzystać z poniższych programów:

- dimacs.py – mikrobiblioteka pozwalająca na wczytywanie grafów, zapisywanie wyników, oraz najbardziej podstawowe operacje na grafach

dimacs.py

W ramach biblioteki `dimacs.py` pojawiły się następujące nowe funkcje:

```
G = loadGRGraph( name )       # wczytaj graf z pliku o nazwie name (zachowuje się tak
                              # jak poprzednio loadGraph, ale używa formatu PACE 2017)

B = loadDecomposition( name)  # wczytuje opis dekompozycji drzewowej; zwraca listę obiektów
                              # klasy Bag (element o numerze 0 nie jest używany)
# B[1] to korzeń drzewa dekompozycji
```

```
class Bag:                    # klasa Bag opisuje węzły dekompozycji drzewowej
  def __init__(self, id):
    self.id = id              # number węzłą na liście
    self.parent = ..          # numer rodzica tego węzła w dekompozycji drzewowej (nie będzie
    self.children = set()     # zbiór numerów węzłów, które są dziećmy tego węzła w dekompozyc
    self.bag = set()          # zbiór wierzchołków oryginalnego grafu, które znajdują się w ty
```

Sugerowana kolejność implementacji

Proponujemy implementować algorytm w następującej kolejności:

- wczytaj graf `e5.gr` oraz jego dekompozycję drzewową, wypisz je na ekran, sprawdź że wszystko jest zgodne z oczekiwaniami
- zaimplementuj funkcję `checkVC( G, X, Y)`, która sprawdza czy jeśli ograniczymy się w grafie G do wierzchołków ze zbioru `X`, to wierzchołki ze zbioru `Y` stanowią pokrycie wierzchołkowe
- rozpocznij implementację funkcji *f*. Wygodnie będzie oprzeć się na programowaniu dynamicznym ze spamiętywaniem.
- zaimplementuj funkcję, która zamienia zbiór wierzchołków i numer węzła np. na napis, którym można będzie indeksować słownik przechowujący wartości funkcji *f*
- dokończ implementację *f*
- zaimplementuje obliczanie wyniku

Przydatne fragmenty kodu

Pomocne są operacje na zbiorach:

```
A = set([1,2,3,4])     # stwórz zbiór {1,2,3,4}
B = set([3,5])         # stwórz zbiór {3,5}
A & B                  # przecięcie zbiorów
A | B                  # suma zbiorów
A - B                  # różnica zbiorów
len(A)                 # rozmiar zbioru

# generowanie wszystkich podzbiorów
from itertools import *
for s in range(len(S)+1):
   for c in combinations( A, s ):
      print(set(c))
```