apto

Lab 1: Algorytmy dokładne dla problemu VertexCover

W ramach laboratorium należy zaimplementować przynajmniej jeden algorytm dokładny (wykładniczy, np. przegląd wszystkich rozwiązań) dla problemu VertexCover.

Algorytmy, do implementacji (w tej kolejności):

- pełny brute-force O(n^k)
- rekurencja z powrotami o złożoności O(2^k)
- rekurencja z powrotami o złożoności O(1.618^k)
- rekurencja z powrotami o złożoności O(1.47^k)
- jeśli ktoś ma pomysły na usprawnienia, tricki, lub inne podejścia to też są mile widziane
- kernelizacja (jako dodatek, nie była jeszcze omawiana na wykładzie)

Wszystkie powyższe algorytmy wymagają podania rozmiaru poszukiwanego pokrycia wierzchołkowego. Niezła strategia, to poszukiwanie liniowe od najmniejszego do największego (wyszukiwanie binarne może strzelać w za duże rozwiązania, które liczyłyby się zbyt długo)

Celem laboratorium jest przekonanie się w praktyce jak dużo zyskuje się na poprawianiu algorytmów wykładniczych.

VertexCover

W problemie VertexCover mamy dany graf $G = (V,E)$ oraz liczbę naturalną k. Należy sprawdzić, czy istnieje zbiór k wierzchołków takich, że każda krawędź dotyka co najmniej jednego wybranego wierzchołka. Jest to jeden z klasycznych problemów NP-zupełnych.

Dokładniejszy opis algorytmów (omówienie złożoności na wykładzie)

Poniżej znajduje się opis algorytmów, które należy zaimplementować w trakcie zajęć.

Brute-Force: $O(n^k)$

W tym algorytmie próbujemy każdego podzbioru k wierzchołków i sprawdzamy, czy jest pokryciem wierzchołkowym dla zadanego grafu

Rekurencja z powrotami: $O(2^k)$

Ten algorytm opiera się na obserwacji, że dla każdej krawędzi $e = \{u,v\}$ należy wybrać co najmniej jeden z wierzchołków u i v. Algorytm działa więc następująco:

```
def VC( G, k, S ):
   # G to graf wejściowy, k liczba wierzchołków, które możemy użyć
   # S to zbiór wierzchołków, który budujemy

   wybierz dowolną krawędź e = {u,v}, która nie jest
   jeszcze pokryta (czyli ani u ani v nie jest wybrany)

   if nie ma takiej krawędzi:
     return S  # rozwiązanie znalezione

   if k == 0:
     return None # nie ma rozwiązania

   S1 = VC( G - {u}, k-1, S + {u} )
   S2 = VC( G - {v}, k-1, S + {v} )

   if S1:
     return S1
   else:
     return S2
```

Oczywiście prawdziwą implementację można wykonać lepiej niż opis idei powyżej (w szczególności warto reprezentować graf jako listę krawędzi).

Rekurencja z powrotami: $O(1.618^k)$

Ten algorytm opiera się na obserwacji, że jeśli rozważamy wierzchołek u, to do rozwiązania należy albo wziąć u, albo wszystkich jego sąsiadów (inaczej nie ma jak pokryć krawędzi łączących u z sąsiadami).

Algorytm działa więc następująco:

```
def VC( G, k, S ):
   # G to graf wejściowy, k liczba wierzchołków, które możemy użyć
   # S to zbiór wierzchołków, który budujemy

   if k < 0:
     return None  # porażka (zabezpieczenie przed tym, że drugie wywołanie rekurencyjne może

   if G nie ma żadnych krawędzi:
     return S     # udało się
   if k == 0:
     return None  # porażka---tu nie znaleźliśmy pokrycia wierzchołkowego
```

```
wybierz dowolny wierzchołek u o stopniu większym od 0

S1 = VC( G - u, k-1, S + {u} )   # zwróćmy uwagę, że usunięcie u usuwa
                                 # też wszystkie krawędzie, które do niego
                                 # dochodziły

# poniżej przez N(u) rozumiemy zbiór wierzchołków, które są połączone
# krawędzią z u (czyli zbiór sąsiadów)
S2 = VC( G - N(u), k-|N(u)|, S + N(u) )

if S1:
  return S1
else:
  return S2
```

Rekurencja z powrotami: $O(1.47^k)$

To ten sam algorytm, co powyżej ale z rozsądniejszym wyborem wierzchołków *u*. Za każdym razem wybieramy wierzchołek o jak największym stopniu (a przynajmniej o stopniu większym od 2). Gdy zostaną same wierzchołki o stopniu 1 i 2, to rozwiązujemy problem w czasie wielomianowym (wówczas graf to zbiór ścieżek i cykli i powinni Państwo być w stanie wskazać algorytm wielomianowy samodzielnie).

Alternatywne podejście (łatwiejsze w implementacji):

- gdy mamy choć jeden wierzchołek o stopniu 1 to wiadomo, że należy do rozwiązania dodać jego jedynego sąsiada (i usunąć z grafu *u* i *N(u)*)
- jeśli nie ma żadnego wierzchołka o stopniu 1, to wybieramy wierzchołek o jak najwyższym stopniu (zwróćmy uwagę, że jeśli zostały nam same wierzchołki o stopniu 2, to tworzą one cykle i każdy cykl rozetniemy w jednym zejściu rekurencyjnym tak, że pojawią się wierzchołki o stopniu 1)

Kernelizacja (to już dla koneserów)

Kernelizacja polega na modyfikacji wejścia tak, że zamiast *(G,k)* dostajemy instancję *(G',k')*, która jest spełnialna wtedy i tylko wtedy gdy spełnialna jest *(G,k)*, ale dodatkowo rozmiar grafu *G'* jest wyłącznie funkcją *k'*. Kernelizacja nie rozwiązuje problemu, ale oblicza jego "jądro trudności". Wynik algorytmu kernelizaji można zapamiętać i uruchomić na nim któryś z pozostałych algorytmów.

W ramach laboratorium można użyć algorytmu, który stosuje następujące reguły na wejściowym grafie, dopóki powodują one zmiany:

- jeśli jest wierzchołek o stopniu 0, usuń go
- jeśli jest wierzchołek o stopniu 1, weź do rozwiązania jego sąsiada (i zmniejsz *k* o 1)
- jeśli jest wierzchołek o stopniu większym od *k* to weź go do rozwiązania (i zmniejsz *k* o 1); wiadomo, że nie można wziąć wszystkich jego sąsiadów, a pokryć krawędzie trzeba

Jeśli w efekcie powstanie graf, który ma więcej niż k^2 krawędzi to znaczy, że nie ma pokrycia wierzchołkowego o rozmiarze k (bo każdy wierzchołek ma stopień między 2 a k, więc wybierając k wierzchołków nie da się pokryć więcej niż k^2 krawędzi)

Grafy

Grafy do testów:

- graph.zip
- Grafy z rodziny `fxx` pochodzą z benchmarku BHOSLIB (nie używane w tym laboratorium)

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
- verify.py – program weryfikujący poprawność rozwiązania
- grademe.py – program oceniający testowe grafy

verify.py

Weryfikacja rozwiązania dla grafu `graph` zapisanego w pliku `solution` polega na wykonaniu polecenia: `python verify.py graph solution`

grademe.py

W swoim katalogu należy umieścić plik `graph.zip` i rozpakować go. W podkatalogu `graph` pojawi się zbiór grafów. Następnie dla wybranych plików `input-graph` w tym katalogu należy zapisać rozwiązania (patrz opis `dimacs.py`) w plikach `input-graph.sol` (również w tym katalogu).

Następnie wywołanie polecenia: `python grademe.py` wypisze spis ocenianych grafów oraz poda dla których znaleźli Państwo poprawne rozwiązania. Ostatnią wypisaną linijkę `=split(...)` należy zignorować.

dimacs.py

Biblioteka `dimacs.py` zapewnia następujące operacje:

Wczytywanie grafu.

Grafy są zapisane w formacie DIMACS ascii:

- Plik zaczyna się od 0 lub więcej linii zaczynających się od znaku `c` i odstępu (komentarze, w których można umieścić dodatkowe informacje o grafie)
- Następnie występuje linia postaci `p edge V E` gdzie `V` to liczba wierzchołków w grafie a `E` to liczba krawędzi.
- Następnie występuje `E` linii postaci `e X Y` oznaczających krawędź nieskierowaną między wierzchołkami o numerach `X` i `Y`.

`dimacs.py` dostarcza funkcję `loadGraph(name)`, która wczytuje graf w formacie DIMACS ascii i zwraca graf w reprezentacji list sąsiedztwa. Na przykład:

```
G = loadGraph( "graph/e5" )
```

wczyta graf z pliku `graph/e5`. Zbiór wierzchołków reprezentowany jest jako liczby całkowite `0, 1, ..., len(G)-1`. Dla wierzchołka `v`, `G[v]` to zbiór (w pythonie: `set`) numerów wierzchołków, do których są krawędzie z `v`. (W formacie DIMACS wierzchołek 0 nie występuje i jest dopisywany do grafu dla zapewnienia jednolitego numerowania wierzchołków w plikach i w kodzie programów.)

Przykład:

Poniższy kod wypisuje jakie krawędzie występują w grafie `graph/e5`:

```
from dimacs import *

G = loadGraph( "graph/e5" )
V = len(G)
for v in range(V):
  s = f"{v} :"
  for u in G[v]:
    s += f" {u}"
  print( s )
```

Konwersja do listy krawędzi:

Funkcja `edgeList( G )` zamienia graf w formacie zbiorów sąsiedztwa na listę krawędzi postaci `(u,v)`, gdzie `u < v` (np. `[(1,2), (1,5), (2,4), ... ]`).

Sprawdzenie rozwiązania:

Funkcja `isVC( E, C )` sprawdza czy zbiór (`set`) wierzchołków `C` jest pokryciem wierzchołkowym dla grafu reprezentowanego jako lista krawędzi.

Zapisanie rozwiązania:

Funkcja: `saveSolution( name, C )` zapisuje do pliku o nazwie `name` zbiór C jako rozwiązanie VertexCover. Nazwa pliku powinna być nazwą grafu z dodanym rozszerzeniem `.sol`.

Dodawanie i usuwanie wierzchołków:

`dimacs.py` nie wspiera bezpośrednio usuwania i dodawania wierzchołków, ale można te operacje łatwo zrealizować przez usuwanie i dodawanie krawędzi.

```
# G to graf reprezentowany jako lista zbiorów sąsiedztwa

# usuwanie wierzchołka G
N = []                    # lista na usunięte krawędzie
for u in G[v].copy():     # przegladaj sąsiadów v
  N += [(u,v)]            # dopisz usuwaną krawędź do N
  G[u].remove(v)          # usun krawędź z u do v
  G[v].remove(u)          # usun krawędź z v do u

# zrób coś ze zmodyfikowanym grafem

# przywracanie wcześnej usuniętego wierzchołka na podstawie listy usuniętych krawędzi
for (u,v) in N:
  G[u].add(v)
  G[v].add(u)
```

Przeglądanie wszystkich podzbiorów:

Do przeglądania wszystkich podzbiorów danej listy można wykorzystać funkcję `combinations` z pakietu `itertools`. Poniższy kod wypisuje wszystkie 3 elementowe podzbiory listy `[0,1,2,3,4,5,6,7,8,9]`:

```
from itertools import *
for C in combinations( range(10), 3 ):
  print C
```

Wykonanie kopii zbioru:

Czasem może być konieczne wykonanie kopii zbioru (obiektu `set` w Pythonie). Jest tak np. wtedy, gdy iterujemy po elementach zbioru ale jednocześnie chcemy go zmieniać. W tym celu, jeśli `C` jest zbiorem wystarczy wykonać operację:

```
D = C.copy()
```