apto

Lab 2: Algorytmy aproksymacyjne dla problemu VertexCover

W ramach laboratorium należy zaimplementować przynajmniej jeden algorytm aproksymacyjny dla problemu VertexCover. Następnie można zająć się albo implementacją kolejnych algorytmów aproksymacyjnych, albo implementacją algorytmów dokładnych z Lab 1

Przykłady algorytmów, które można implementować:

- algorytm 2-aproksymacyjny (wybieramy niepokrytą krawędź, dodajemy oba jej wierzchołki do rozwiązania)
- algorytm O(logn)-aproksymacyjny (dodajemy do rozwiązania wierzchołek o najwyższym aktualnym stopniu)
- dowolna realizacja symulowanego wyżarzania (simulated annealing)
- jeśli ktoś ma pomysły na usprawnienia, tricki, lub inne podejścia to też są mile widziane (np. usuwanie w losowej kolejności wierzchołków z rozwiązania obliczonego przez dany algorytm, o ile nie powoduje to, że tracimy pokrycie wierzchołkowe)

Celem laboratorium jest przekonanie się w praktyce jak dobre są algorytmy, których jakość udało nam się oszacować teoretycznie oraz jak wypadają na tle naturalnej heurystyki.

Grafy

- graph2.zip
- Grafy z rodziny `fxx` pochodzą z benchmarku BHOSLIB.

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, nagrywanie wyników, oraz najbardziej podstawowe operacje na grafach
- verify.py – program weryfikujący poprawność rozwiązania
- grademe2.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`

grademe2.py

W swoim katalogu należy umieścić plik `graph2.zip` i rozpakować go. W podkatalogu `graph` pojawi się zbiór grafów. Następnie dla wybranych plików `input-graph` w tym katalogu należy nagrać rozwiązania (patrz opis `dimacs.py`) w plikach `input-graph.sol` (również w tym katalogu).

Następnie wywołanie polecenia: `python grademe2.py` wypisze spis ocenianych grafów oraz poda dla których znaleźli Państwo poprawne, optymalne rozwiązania (program sprawdzający jest taki sam jak w Lab 1, ale obejmuje więcej grafów).

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 = "%d :" % v
  for u in G[v]:
    s += " %d" % u
  print s
```

Konwersja do listy krawędzi:

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

Sprawdzenie rozwiązania:

Funkcja `isVC( E, C )` sprawdza czy zbior (`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

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 tem celu, jesli `C` jest zbiorem wystarczy wykonać operację:

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