apto

# Lab 5 Problem SAT (część 3: VertexCover i funkcja progowa)

W ramach laboratorium należy:

1. Wykonać podstawową część redukcji Vertex Cover do SAT
2. Wypróbować różne sposoby realizacji funkcji progowej

Grafy do testów należy wziąć z wcześniejszych laboratoriów.

## Szczegóły techniczne

Wszystkie programy powinny być implementowane w języku Python (wersja 3.x.y). Mogą Państwo (i powinni) korzystać z poniższych programów:

- dimacs.py - mikrobiblioteka pozwalająca na wczytywanie grafów, nagrywanie wyników, najbardziej podstawowe operacje na grafach, oraz nagrywanie formuł logicznych CNF w formacie DIMACS ascii (z którego korzystają solwery SAT)
- sortnet.py - mikrobiblioteka pozwalająca tworzyć formuły CNF implementujące sieci sortujące
- Biblioteka pycoSAT
- Solwery glucose lub maple.

## Zadanie 1: VertexCover (podstawa redukcji)

Proszę zrealizować redukcję problemu VertexCover do SAT. Najpierw zajmiemy się elementarną częścią redukcji, a potem zrealizujemy funkcję progową.

### Redukcja SAT

Mamy graf $G = (V,E)$, gdzie $V = \{ v_1, \ldots, v_n \}$ oraz pytamy, czy istnieje pokrycie wierzchołkowe wykorzystujące $k$ wierzchołków. Dla każdego wierzchołka $v_i$ tworzymy zmienną $x_i$, której wartość interpretujemy następująco:

- $x_i$ = prawda – wierzchołek $v_i$ należy do pokrycia,
- $x_i$ = fałsz – wierzchołek $v_i$ nie należy do pokrycia.

Każda krawędź musi być pokryta przez jakiś wierzchołek. Stąd dla każdej krawędzi $\{ v_i, v_j \}$ tworzymy klauzulę wymagającą, że co najmniej jeden z jej końców został wybrany:

$( x_i \lor x_j )$

Taka redukcja wystarcza, żeby solwer SAT znalazł *jakieś* pokrycie wierzchołkowe, ale nic go nie zmusza do minimalizacji liczby użytych wierzchołków.

# Zadanie 2: Implementacja funkcji progowej

Trudniejszą częścią redukcji VertexCover do SAT jest wymuszenie, że najwyżej $k$ zmiennych spośród $x_1$, ..., $x_n$ może mieć wartość `prawda` .

## Rozwiązanie a'la programowanie dynamiczne

W pierwszym podejściu tworzymy zmienne $y_{i,j}$, które interpretujemy następująco:

- $y_{i,j}$ = prawda – wśród zmiennych $x_1, \dots, x_i$, co najmniej j ma wartość `prawda` ,
- $y_{i,j}$ = fałsz – powyższe nie zachodzi.

Wymuszamy poprawną wartość zmiennych następująco. Przede wszystkim zmienne $y_{i,0}$ są prawdziwe dla wszystkich $i$, a zmienne $y_{0,j}$ są fałszywe dla wszystkich $j > 0$:

$( y_{0,0} ) \land ( y_{1,0} ) \land \dots \land ( y_{n,0} ) \land (-y_{0,1} ) \land ( -y_{0,2} ) \land \dots \land ( -y_{0,n} )$

Następnie dla każdej pary wartości $0 < i,j \le n$ dodajemy następujące implikacje (każda z nich może być zapisana jako pojedyncza klauzula):

$( y_{i-1,j} \Rightarrow y_{i,j} ) \land ( ( y_{i-1,j-1} \land x_i ) \Rightarrow y_{i,j} )$

Na samym końcu dodajemy pojedynczą klauzulę:

$( -y_{n,k+1} )$

która wymusza, że nie używamy więcej niż $k$ wierzchołków do pokrycia.

## Indeksowanie zmiennych

W tym zadaniu należy odróżnić zmienne $x_i$ od zmiennych $y_{i,j}$. Niestety pycoSAT (oraz inne solwery) stosują prostą numerację zmiennych. W związku z tym konieczne jest tłumaczenie numerów "dwuwymiarowych" na jednowymiarowe. W tym celu przydatna może być poniższa funkcja, która zamienia parę indeksów $i, j$ na jedną, jednoznacznie określoną, liczbę naturalną:

```
def index( i, j ):
  return (i+j)*(i+j+1)/2+i
```

### Testowanie redukcji

Jako dane testowe proszę wykorzystać zestaw grafów z poprzedniego laboratorium: graph.zip

## Zadanie 2: Funkcja progowa zbudowana na sieci sortującej

Alternatywne rozwiązanie polega na zastosowaniu sieci sortujące (patrz wykład 6). Sieć sortującą zrealizujemy w oparciu o clasę `sorterNet` :

```
class sorterNet:
    def __init__( self, start, lines, equiv ):
      # stwórz sieć sortującą
      # start -- numer pierwszej dostępnej zmiennej
      # lines -- tablica numerów zmiennych, które są sortowane
      # equiv -- czy komparatory mają być realizowane jako równoważności,
                 czy jako implikacje


    def comp( self, i, j ):
      # dodaj komparator między liniami i oraz j
      # większa wartość wędruje do linii o niższym numerze


    def getCNF( self )
      # odczytaj formułę realizującą zadane komparatory

    def getLines( self ):
      # odczytaj obecne numery zmiennych przechowujących kolejne linie
```

### Elementarny test

Sprawdź, czy powyższa klasa działa poprawnie. Stwórz formułę, która zmiennym 1, 2 i 3 przypisuje—na przykład—wartości False, True, False, stwórz obiekt `sorterNet` dla `lines = [1,2,3]` (oraz używającej równoważności) i dodaj komparatory (1,2), (2,3), (1,2). Przekonaj się, że połączenie Twojej formuły oraz tej, którą zwraca `sorterNet` poprawnie sortuje zmienne 1,2,3 (żeby to sprawdzić, musisz wywołać `getLines` po dodaniu komparatorów, żeby wiedzieć jaka zmienna realizuje jaką linię).

### Sieci sortujące

Najprostszą siecią sortującą jest sieć realizująca sortowanie przez wstawianie. W powyższym zadaniu stworzyliśmy taką sięc dla trzech linii. Można to łatwo uogólnić do sieci dla dowolnej liczby linii.

Wykład 6 zawiera także opis sieci opartej o MergeSort.

### Użycie sieci sortującej w redukcji VertexCover

Proszę połączyć sieci sortujące z poprzedniego punktu z redukcją z VertexCover. Najpierw wykonujemy elementarną część redukcji, która używa zmiennych od *1* do *n* do opisu wykorzystania wierzchołków. Następnie dodajemy sieć sortującą, która traktuje te zmienne jako linie wejściowe. Na koniec wymuszamy, że *k+1*-sza linia ma wartość *False*

## Lepsze kodowanie funkcji progowej

Przedstawione metody kodowania warunku liczności nie są optymalne. Istnieje wiele lepszych metod, z których jedna jest opisana w pracy:


- M. Karpiński, M. Piotrów, Encoding cardinality constraints using multiway merge selection networks, Constraints, Vol. 24, str. 234–251, 2019. PDF