apto

Lab 3: Problem SAT (część 1)

W ramach laboratorium należy:

1. Wykonać zadanie pierwsze: przejście fazowe SAT
2. Wykonać redukcję z problemu X3C do SAT (+ testy)
3. Wykonać redukcję problemu kolorowania grafów do X3C [Dodatkowo]

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)
- Biblioteka pycoSAT
- Solwery glucose lub maple.

pycoSAT

Biblioteka pycoSAT tworzy interface do solwera PicoSAT dla języka Python (PicoSAT odnosił sukcesy na turniejach rozwiązywania instancji SAT—SAT Competition—ok. roku 2007; dzięki wygodnemu interface'owi będzie świetnym narzędziem na początek)

Instalacja pycoSAT. Należy zainstalować pakiet `pycosat` wykonując polecenie `pip install pycosat` (w niektórych systemach `pip3` zamiast `pip`)

Reprezentacja formuł. pycoSAT reprezentuje formuły jako listy klauzul, gdzie każda klauzula to lista numerów zmiennych, które w niej występują. Wartość ujemna oznacza zanegowanie danej zmiennej. Na przykład lista:

```
cnf = [ [-1,2,3], [2,-3], [1,-2,-3] ]
```

reprezentuje formułę $(-x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee -x_3) \wedge (x_1 \vee -x_2 \vee -x_3)$. Zmienna o numerze 0 nie istnieje.

Szukanie rozwiązania. Aby sprawdzić czy formuła jest spełnialna i znaleźć odpowiednie wartościowanie należy wykonać funkcję `solve( cnf )`. Poniższy kod pokazuje przykład:

```
import pycosat
cnf = [ [-1,2,3], [2,-3], [1,-2,-3] ]
sol = pycosat.solve( cnf )
print( sol )
```

Ten kod jako wyjście wypisze `[-1, 2, -3]`, co oznacza, że formuła jest spełnialna i świadczy o tym wartościowanie:

- $x_1 =$ `fałsz`,
- $x_2 =$ `prawda`,
- $x_3 =$ `fałsz`.

Formalnie, funkcja `solve` zwraca następujące wartości:

- napis `u'UNSAT'` jeśli formuła jest niespełnialna
- listę numerów zmiennych, gdzie zmienna występuje jako ujemna jeśli w rozwiązaniu ma logiczną wartość `fałsz` oraz dodatnią jeśli w rozwiązaniu ma wartość `prawda`.

dimacs.py

Tę mikrobibliotekę znają Państwo z Laboratorium 1: VertexCover. W ramach tego laboratorium pojawiły się dwie nowe funkcja:

```
saveCNF( name, cnf )
```

która nagrywa formułę ze zmiennej `cnf` (reprezentowaną jako lista klauzul) do pliku o nazwie `name` w formacie DIMACS ascii. Taki plik można bezpośrednio przekazać do zewnętrznego solwera. Druga nowa funkcja to:

```
loadX3C( name )
```

która wczytuje instancję problemu X3C (jest omówiona przy redukcji X3C do SAT).

glucose i maple

Solwerami zewnętrznymi należy zająć się po zakończeniu wszystkich innych prac

glucose i maple to dwa solwery SAT (wywodzące się z tej samej bazy kodu), które odnosiły sukcesy w ostatnich turniejach SAT. Po ściągnięciu ich źródeł należy je skompilować.

Odpowiednie instrukcje znajdują się w pliku README. Sprowadza się to do wykonania poleceń:

```
cd simp
make rs
```

W przypadku glucose może być konieczna podmiana opcji kompilatora w pliku `mtl/template.mk` (zmiana linii zaczynającej się od `CFLAGS` na `CFLAGS ?= -Wall -Wno-parentheses -std=c++0x`).

Wywołanie solwera. Jeśli solwer ma nazwę `solver` oraz mamy plik `cnf` z zapisaną formułą, to należy wykonać polecenie:

```
solver cnf
```

Zadanie 1: Przejście fazowe SAT

W ramach tego zadania należy sporządzić wykres, który pokazuje prawdopodobieństwo, że losowo wygenerowana formuła w formacie k-CNF (k literałów na klauzulę) jest spełnialna, w zależności od ilorazu liczby klauzul i liczby zmiennych.

Generowanie losowej klauzuli. Załóżmy, że mamy *n* zmiennych do dyspozycji (o numerach *1, ..., n*). Losowa klauzula rozmiaru *k* składa się z losowo wybranych *k* zmiennych (z powtarzaniem), z których każda jest zanegowana/niezanegowana z prawdopodobieństwem 1/2.

Od strony technicznej przydatny mozę być następujący fragment kodu:

```
import random          # biblioteka liczb pseudolosowych
n = 5
S = [1,-1]             # lista +/-
V = range(1,n+1)       # lista zmiennych 1...n
x = random.choice(V)*random.choice(S)  # losowo wybrana zmienna z losowym negowaniem
print( x )
```

Wykonanie eksperymentu. Ustalmy *k=3*, czyli problem 3CNF-SAT. Proszę ustalić pewną liczbę zmiennych *n* (np. 10, 50, 100) oraz pewną liczbę powtórzeń *T* (np. *T = 100*). Następnie, dla wartości *a* z przedziału 1 do 10 (np. z krokiem 0.1) proszę:

- Wygenerować *T* formuł zawierających po *n* zmiennych oraz *a * n* klauzul.
- Dla każdej wygenerowanej formuły sprawdzić, czy jest spełnialna.
- Zapisać liczbę *S* spełnialnych formuł.
- Wypisać wartość *a* oraz iloraz *S/T*.

Następnie proszę wygenerować wykres funkcji *S/T* w zależności od *a*.

Proszę powtórzyć eksperyment dla kilku wartości *n* oraz dla innej wartości *k*.

Rysowanie wykresu

Jeśli dane są w pliku tekstowym `dane`, np. o treści:

```
1 0.1
2 0.4
3 0.8
4 0.16
5 0.32
```

to można zrobić ich wykres przy pomocy narzędzia `gnuplot`, wpisując w jego konsoli:

```
set yrange [-0.1:1.1]
plot "dane" using 1:2 with lines
```

Alternatywa: Wykorzystanie pakietu OpenOffice lub Google Sheets.

Zadanie 2: Exact Cover by 3-Sets (X3C)

W tym zadaniu należy zaimplementować redukcję pewnego wariantu problemu X3C do SAT oraz wykorzystać solwer SAT do rozwiązywania przykładowych instancji X3C.

W problemie X3C mamy dany zbiór elementów $N = \{1, ..., 3k\}$ oraz rodzinę zbiorów $S = \{S_1, ..., S_m\}$ gdzie $m <= 3k$. Każdy zbiór S_i zawiera trzy elementy ze zbioru N a każdy element z N występuje najwyżej w trzech zbiorach z rodziny S. Pytanie brzmi czy da się wybrać k zbiorów tak, że każdy element zbioru N należy do dokładnie jednego wybranego zbioru.

Redukcja do SAT

Pomysł redukcji polega na tym, że dla każdego zbioru S_i tworzymy zmienną logiczną x_i, której wartość interpretujemy następująco:

- jeśli x_i = 1 to zbiór *S_i należy do rozwiązania,
- jeśli x_i = 0 to zbiór *S_i nie należy do rozwiązania.

Następnie tworzymy następujące klauzule. Dla każdego elementu j ze zbioru N tworzymy klauzulę, która składa się ze zmiennych odpowiadających zbiorom, do których należy j. Na przykład, jeśli element 1 należy do zbiorów S_6, S_9 oraz S_{11} to tworzymy klauzulę:

$((x_6 \vee x_9 \vee x_{11}))$

która wymusza, że co najmniej jeden z tych zbiorów jest wybrany. Następnie dla każdej takiej klauzuli należy zapewnić, że żadne dwa zbiory zawierające ten sam element nie są wybrane jednocześnie. Dla powyższego przykładu wystarczy dodać trzy klauzule:

$((-x_6 \vee -x_9) \wedge (-x_6 \vee -x_{11}) \wedge (-x_9 \vee -x_{11}))$

Każda z powyższych klauzul mówi, że spośród dwóch zbiorów co najwyżej jeden może być wybrany.

Testowanie redukcji

Jako dane testowe proszę wykorzystać instancje X3C z pliku x3c.zip. Nazwa pliku mówi, czy instancja ma rozwiązanie czy nie, oraz daje pogląd na temat rozmiaru danych (np. plik 10.yes.x3c ma rozwiązanie i zawiera zbiór N składający się z 3*10 elementów). Instancje z tego pliku można wczytać przy pomocy funkcji

```
loadX3C( name )
```

która wczytuje instancję z pliku *name* i zwraca parę *n, sets*, gdzie *n* to liczba elementów do pokrycia (numerowanych od *1* do *n*) a *sets* to lista zbiorów, gdzie każdy zbiór jest listą trzyelementową. N

Zadanie 3: Kolorowanie grafów

W tym zadaniu rozważamy problem kolorowania grafów, zdefiniowany następująco:

- Wejście: Graf nieskierowany *G*, liczba naturalna *k*.
- Pytanie: Czy da się przypisać każdemu z wierzchołków jeden z *k* kolorów tak, żeby żadne dwa wierzchołki połączone krawędzią nie miały tego samego koloru.

W ramach zadania proszę napisać program, który wczytuje graf oraz dostaje liczbę *k* dopuszczalnych kolorów, oblicza formułę logiczną, która jest spełnialna wtedy i tylko wtedy gdy graf posiada odpowiednie kolorowanie, sprawdza spełnialność tej formuły (i w przypadku spełnialności wypisuje numery kolorów przypisanych wierzchołkom).

Redukcja do SAT

Można wykorzystać następującą redukcję do SAT. Mamy graf $G = (V,E)$, gdzie $V = \{v_1, ..., v_n\}$ oraz *k* kolorów do wykorzystania. Dla każdego wierzchołka v_i oraz koloru *j* tworzymy zmienną $x_{i,j}$, której wartość interepretujemy następująco:

- $x_{i,j}$ = prawda – wierzchołek v_i ma kolor *j*,
- $x_{i,j}$ = fałsz – wierzchołek v_i nie ma koloru *j*.

Dla każdego wierzchołka x_i tworzymy serię klauzul, które mówią, że ten wierzchołek ma dokładnie jeden kolor:

$(x_{i,1} \vee x_{i,2} \vee ... \vee x_{i,k}) \wedge (-x_{i,1} \vee -x_{i,2}) \wedge (-x_{i,1} \vee -x_{i,3}) \wedge ... \wedge (-x_{i,1} \vee -x_{i,k}) \wedge (-x_{i,2} \vee -x_{i,3}) \wedge (-x_{i,2} \vee -x_{i,4}) \wedge ... \wedge (-x_{i,2} \vee -x_{i,k}) \wedge ... \wedge (-x_{i,k-1} \vee -x_{i,k})$

Dla każdej krawędzi $\{ v_i,v_t \}$ i dla każdego koloru *j* tworzymy klauzulę, która mówi, że oba wierzchołki nie mogą mieć jednocześnie koloru *j*:

$(-x_{i,j} \vee -x_{t,j})$

Testowanie redukcji

Jako dane testowe proszę wykorzystać zestaw grafów pochodzący ze zbioru benchmarków Graph Coloring Benchmarks (wygasły link). Proszę pamiętać o odczytaniu kolorów wierzchołków z wartościowania formuły oraz zrobienia wewnętrznego testu w programie, sprawdzającego czy kolorowanie jest poprawne.

Zewnętrzne solwery

Jeśli `pycoSAT` okaże się za wolny, można próbować nagrywać formułę na dysk (`saveCNF`) oraz uruchamiać solwery `glucose` lub `maple` .