# apto

# Lab 4: Problem SAT (część 2: Solwer DPLL)

W ramach laboratorium należy:

1. Zaimplementować elementarny solwer SAT typu DPLL
2. Przetestować solwer na różnych formułach
3. Użyć własnego solwera w programach z poprzedniego laboratorium

W szczególności realizacja tych zadań może się przeplatać. W ramach laboratorium będziemy coraz bardziej poprawiać podstawowy solwer, dzięki czemu będzie się dało rozwiązać coraz więcej formuł. Podobnie będzie można go równolegle testować w Państwa wcześniejszych programach.

## 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 wczytywanie i nagrywanie formuł logicznych CNF w formacie DIMACS ascii (z którego korzystają solwery SAT)
- sat.zip - zestaw formuł CNF do testów

### Reprezentacja formuł Formuły reprezentujemy tak jak w pycoSAT,

jako listy klauzul, gdzie każda klauzula to lista numerów zmiennych (literałów), które w niej występują. Wartość ujemna oznacza zanegowanie danej zmiennej. Na przykład lista:

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

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

### dimacs.py

Tę mikrobibliotekę znają już Państwo. W ramach tego laboratorium pojawiła się funkcja:

```
loadCNF( name )
```

która wczytuje formułę CNF z pliku w formacie DIMACS ascii.

# Podstawowy solwer

Algorytm DPLL (czyli algorytm Davisa, Putnama, Logemanna oraz Lovelanda) to prosty algorytm z powrotami, uzupełniony o kilka optymalizacji. W ramach laboratorium zaimplementujemy podstawowy algorytm z powrotami, następnie dodamy optymalizacje DPLL i kilka innych pomysłów.

## Podstawowy algorytm

Nasz bazowy algorytm z powrotami ma następującą strukturę:

```
def solve( CNF, V ):
  # CNF to rozważana formuła
  # V to wartościowanie zmiennych

  if CNF jest spełniona przez V:
    return V

  v = zmienna występująca w CNF

  if solve( CNF-z-v-ustawionym-na-1, V-z-v-ustawionym-na-1 ):
    return V
  if solve( CNF-z-v-ustawionym-na-0, V-z-v-ustawionym-na-0 ):
    return V

  return "UNSAT"
```

Zaczniemy od zaimplementowania go. W tym celu potrzebujemy rozwiązać kilka pomniejszych problemów.

## Reprezentacja wartościowania zmiennych

Proponujemy reprezentować wartościowanie zmiennych jako słownik `V` zawierający wartość dla każdego ustalonego literału. Wartość `1` oznacza prawdę a wartość `-1` oznacza fałsz.

Zamiast słownika można oczywiście wykorzystać listę wartości, co poprawi część wielomianową algorytmu, ale utrudni trochę implementację, a przekonają się Państwo, że dużo ważniejsze od części wielomianowej jest ograniczenie rekursji.

W szczególności następujący kod pokazuje przykład kilku operacji na zaproponowanym wartościowaniu zmiennych:

```
V = {}     # stwórz puste wartościowanie

# ustaw x_1 na prawdę
V[ 1] =  1
```

```
V[-1] = -1

# ustaw x_3 na fałsz
V[ 3] = -1
V[-3] =  1

# usuń wartościowanie zmiennej x_1
del V[1]

# skopiuj wartościowanie
newV = V.copy()
```

## Konwencja dla pustych klauzul/formuł

Przyjmujemy następującą konwencję co do wartości `None` i `[]` dla klauzul i formuł.

Klauzule:

- klauzula `None` jest spełniona (bo to brak klauzuli)
- klauzula `[]` jest niespełniona (bo to pusta alternatywa; tak jak suma zera elementów to zero)

Formuły:

- formuła `[]` jest spełniona (bo to pusta koniunkcja; tak jak iloczyn pustego zbioru elementów to jeden)
- formuła `None` jest niespełniona (bo tak)

## Upraszczanie klauzul

Proszę zaimplementować funkcję upraszczającą klauzulę na podstawie wartościowania (tj. tworzy nową kopię klauzuli, pozbawioną literałów ustawionych na False; jeśli klauzula jest spełniona przez V to zwraca None):

```
def simplifyClause( C, V ):
  # C - klauzula, czyli lista literałów
  # V - wartościowanie zmiennych

  ...

  return uproszczona-C lub None jeśli klauzula jest spełniona
```

## Upraszczanie formuł

Proszę zaimplementować funkcję, która upraszcza formułę CNF na podstawie wartościowania (czyli upraszcza każdą klauzulę).

```
def simplifyCNF( CNF, V ):
  # CNF - formuła do uproszczenia
  # V   - wartościowanie zmiennych

  uprość każdą klauzulę pamietając, że:
  - jeśli uproszczona klauzula to None, to ją pomijamy
  - jeśli uproszczona klauzula to [] to ta klauzula jest niespełniona i zwracamy None

  return uproszczona-CNF
```

### Implementacja solwera

Mając funkcje upraszczające klauzule i formuły oraz reprezentację wartościowania zmiennych, można łatwo zaimplementować podstawowy solwer.

### Testowanie solwera

Proszę przetestować solwer na niewielkich formułach, w szczególności na plikach:

- 5.{yes/no}.sat
- 10.{yes/no}.sat
- 20.{yes/no}.sat
- 30.{yes/no}.sat (może wymagać kilkunastu do kilkudziesięciu sekund)

Większe formuły są raczej poza zasięgiem tego solwera.

Proszę zmodyfikować kod tak, żeby zliczał liczbę wywołań rekurencyjnych `solve` i wypisywał ją

## Inna strategia backtrackingu

Na wykładzie pokazaliśmy, że mając klauzulę *(x ∨ y ∨ z ∨ ... ∨ a)* lepszą strategią backtrackingu od próbowania, na przykład, *x=1* a potem *x=0* jest próbowanie wartościowań:

- *x=1*
- *x=0*, *y=1*,
- *x=0*, *y=0*, *z=1*,
- ...
- *x=0*, *y=0*, *z=0*, ..., *a=1* Proszę zmodyfikować podstawowy solwer tak, by wykorzystywał tę strategię (proszę zwrócić uwagę, że klauzule mogą mieć różne długości i proszę nie zakładać, że zawsze będzie to trzy).

### Testowanie

Proszę spróbować tych plików co poprzednio i porównać liczbę wejść do funkcji `solve` . Można się pokusić o policzenie pliku:

- 30.yes.sat (nasza implementacja wymagała 3237348 wywołań rekurencyjnych i obliczenia trwały blisko trzy minuty)

Warto spróbować też wybierania w solwerze odpowiednio najmniejszej lub największej klauzuli do zejścia rekurencyjnego. Przy niektórych formułach może to dać bardzo dobre efekty i pokazuje, że kolejność przetwarzania klauzul ma duże znaczenie. W szczególności rozważając klauzule od najmniejszej nasz wzorcowy solwer rozwiązywał formuły `100.{yes/no}.sat` poniżej 5 sekund.

# Solwer DPLL

Podstawowa różnica między poprzednim solwerem a solwerem DPLL to wykonywanie propagacji jednostkowej (ang. unit propagation) oraz ustawiania wartości zmiennych występujących tylko z jednym znakiem przed zejściami rekurencyjnymi

```
def solve( CNF, V ):
  # CNF to rozważana formuła
  # V to wartościowanie zmiennych

  CNF = unit_propagation( CNF )
  CNF = fix_const( CNF )

  if CNF jest spełniona przez V:
    return V

  v = zmienna występująca w CNF

  if solve( CNF-z-v-ustawionym-na-1, V-z-v-ustawionym-na-1 ):
    return V
  if solve( CNF-z-v-ustawionym-na-0, V-z-v-ustawionym-na-0 ):
    return V

  return "UNSAT"
```

## Unit Propagation

Optymalizacja ta polega na tym, że przeglądamy wszystkie klauzule i jeśli znajdziemy klauzulę składającą się z jednego literału, to wiadomo, że należy mu przypisać wartość *1*. Oczywiście przypisanie wartości jednemu takiemu literałowi może spowodować, że powstają kolejne, tak więc operację tę należy wykonywać dopóki (a) formuła nie zostanie spełniona, (b) nie dojdziemy do sprzeczności, lub (c) nie ma więcej klauzul jednostkowych.

```
def unitPropagate( CNF, V ):

  while CNF zawiera klauzulę postaci C = [L]:
    if L ustawione w V na 0:
      formuła niespełnialna
```

```
      ustaw L na 1 w V
      uprość formułę CNF

    return CNF
```

### Testowanie

Unit propagation jest bardzo silną heurystyką i zastosowanie jej powinno istotnie przyspieszyć solwer. Proszę spróbować formuł:

- *x*.{yes/no}.sat dla wszystkich wartości *x*

### Literały o ustalonym znaku

Druga optymalizacja to wykrywanie literałów, które w formule występują albo tylko zanegowane, albo tylko niezanegowane i przypisywanie im odpowiedniej wartości.

### Testowanie

Algorytm DPLL z obiema heurystykami powinien sobie poradzić, m.in., z formułami:

- anna.11.sat
- anna.15.sat

## Solwer + strategie wyboru zmiennej

Na wydajność solwera istotny wpływ ma strategia wyboru zmiennej względem której przebiega rekursja. Proszę wypróbować następujące strategie:

- Wybieranie zmiennej, która pojawia się w największej liczbie klauzul
- Wybieranie zmiennej, która pojawia się w najmniejszej liczbie klauzul

### Testowanie

Proszę wykonać testy, które sprawdzą:

- Jak strategie wpływają na formuły x.{yes/no}.sat? .
- Czy da się zweryfikować spełnialność formuły `anna.5.sat` ?

Podpowiedź: Nasza implementacja jednej z tych strategii rozwiązuje formułę `anna.5.sat` (niespełnialna) po 353 wywołaniach funkcji `solve` w wariancie DPLL.

## Solwer + SAT2CNF

#### Wymaga biblioteki networkx

W ramach przedmiotu algorytmy grafowe implementowali Państwo solwer wielomianowy dla problemu SAT-2CNF. Można uzupełnić nasz solwer DPLL o testowanie, czy formuła ma dokładnie 2 literały w każdej klauzuli (po propagacji jednostkowej nie ma klauzul rozmiaru 1) oraz uruchamianie takiego solwera.

Przykładowy solwer SAT-2CNF dostępny jest tutaj: <a href="faliszew.github.io/apto/sat2cnf.py>sat2cnf.py</a>:

```
sat2cnf( CNF, V )  # sprawdza czy formuła CNF z dokładnie dwoma zmiennymi na klauzule jest speł
                   # jeśli tak, zwraca wartościowanie V uzupełnione o spełniające CNF
                   # jeśli nie, zwraca "UNSAT"
```

## Testowanie

To usprawnienie czasem poprawia czas działania solwera, ale czasem istotnie pogorsza.

- Proszę sprawdzić formuły z rodziny `r30`
- Proszę sprawdzić wpływ wybierania najdłuższej lub najkrótszej klauzuli w funkcji rekurencyjnej