# MI-PAA 2015 5.ukol
# Řešení problému vážené splnitelnosti booleovské formule pokročilou iterativní metodou

Tomas Nesrovnal
nesrotom@fit.cvut.cz
utery 16:15

January 30, 2016

# 1 Uvod do problemu

## 1.1 Definice

Je dána booleovská formule $F$ proměnnných $X = (x1, x2, \ldots, xn)$ v konjunktivní normální formě (tj. součin součtů). Dále jsou dány celočíselné kladné váhy $W = (w1, w2, \ldots, wn)$. Najděte ohodnocení $Y = (y1, y2, \ldots, yn)$ proměnných $x1, x2, \ldots, xn$ tak, aby $F(Y) = 1$ a součet vah proměnných, které jsou ohodnoceny jedničkou, byl maximální.

Je přípustné se omezit na formule, v nichž má každá klauzule právě 3 literály (problém 3 SAT). Takto omezený problém je stejně těžký, ale možná se lépe programuje a lépe se posuzuje obtížnost instance (viz Selmanova prezentace v odkazech).

## 1.2 Priklad

$x1'$ značí negaci $x1$.

$n = 4$

$F = (x1 + x3' + x4).(x1' + x2 + x3').(x3 + x4).(x1 + x2 + x3' + x4').(x2' + x3).(x3' + x4')$

$W = (2, 4, 1, 6)$

Přípustné konfigurace, kde $F = 1$ (řešení):

$X = \{x1, \ldots, xn\} = \{0, 0, 0, 1\}, S = 6$

$X = \{x1, \ldots, xn\} = \{1, 0, 0, 1\}, S = 2 + 6 = 8$ (optimální)

$X = \{x1, \ldots, xn\} = \{1, 1, 1, 0\}, S = 2 + 4 + 1 = 7$

## 1.3 DIMACS CNF format

Instance z prikladu zapsana v DIMACS CNF formátu vypada nasledovne:

```
c Priklad CNF
c 4 promenne a 6 klauzuli
c kazda klauzule konci nulou (ne novym radkem)
p cnf 4 6
1 -3 4 0
-1 2 -3 0
3 4 0
1 2 -3 -4 0
-2 3 0
-3 -4 0
```

Tento format neobsahuje vahy. Symbol c znamena komentar. Radek zacinajici p ve formatu "p cnf nbvar nbclauses" rika, ze instance je v CFN formatu, ma nbvar promennych a obsahuje nbclauses klauzoli. Pridame proto radek zacinajici w za kterem budou nasledovat vahy. Ukazova instance z predhoziho prikladu ma tedy tento format:

```
c Priklad CNF
c 4 promenne a 6 klauzuli
c kazda klauzule konci nulou (ne novym radkem)
p cnf 4 6
w 2 4 1 6
1 -3 4 0
-1 2 -3 0
3 4 0
1 2 -3 -4 0
-2 3 0
-3 -4 0
```

### 1.4 3SAT

Protoze je 3SAT stejne tezky, budeme pracovat pouze s nim. Lepe se bude posuzovat obtiznost instanci a take se snadneji vytvareji, hledaji testovaci data.

## 2 Algoritmus a implementace

### 2.1 Simulovana evoluce

Nejprve jsem implementoval zakladni simulovanou evoluci. Pouzita byla jen mutace, dvoubodove krizeni a turnajova selekce. Protoze algoritmus nedosahoval dobrych vysledku a protoze jsem simulovanou evoluci resil predchozi ulohu batohu, rozhodl jsem se od tohoto reseni upustit a zkusit simulovane zihani.

### 2.2 Simulovane zihani

Algoritmus simulovaneho zihani vychazi z algoritmu Hill climbing.

#### 2.2.1 Hill climbing

Hill climbing vyuziva dosud nejlepsiho nalezeneho reseni. Analogicky k splhani do kopce se rozhlizime (generujeme dalsi body z nalezeneho reseni) a

jdeme nahoru (tedy pokud je vygenerovany bod vys, jdeme na nej). Generovanim dalsich bodu se v nasem pripade mysli prohozeni hodnoty nektere promenne.

Nejvetsim problemem Hill climbingu je uvaznuti v lokalnich extremech. Resenim by bylo s nejakou pravdepodobnosti prijmout i horsi reseni. Na tomto principu funguje simulovane zihani.

### 2.2.2 Simulovane zihani

Anglicky Simulated annealing, jinak cesky Simulovane propousteni nebo i simulovane ochlazovani.

Je to tedy algoritmus podobny Hill climbingu, ktery ale s urcitou a postupne klesajici pravdepodobnosti prijima horsi stavy, cimz je schopen vyvaznout z lokalnich extremu.

Algoritmus zacne s nejakou pocatecni telplotou a po ekvilibrium krocich ji snizi vynasobenim ochlazovanim faktorem. V kazdem kroku pak vygeneruje novy stav. Pokud je jeho cena lepsi, prijme se jako soucasne nejlepsi nalezene reseni. Pokud ne, je jeste $e^{-d/t}$ (kde d je rozdil cen mezi soucasnym nejlepsim resenim a nove vygenerovanym a t je soucasna teplota) sance, ze se take stav prijme. To nam umozni vyvaznuti z lokalnich extremu.

- ti je pocateni teplota
- te je konecna teplota ($te < ti$)
- eq je hodnota equlibria ($0 < eq$)
- cf je ochlazovaci faktor ($0 < cf < 1$)

Zde je zjednoduseny zdrojovy kod v jazyce C:

```
for (double t = ti; te < t; t *= cf) {
  for (int i = 0; i < eq; i++) {
    state_next = state_genenerate_next(state);
    double d = cost(state) - cost(state_next);
    if (d < 0 || randd() < pow(M_E, -d / t)) {
      state_swap(&state, &state_next);
      continue;
    }
  }
}
```

# 3 Reseni

Je nekolik velmi dulezitych veci nad kterymi je potreba se zamyslet pro dobrou implementaci algoritmu. Za prve je to spravne nastaveni parametru pro simulovane zihani - tedy pocatecni a koncove teploty, chladici faktor a hodnota ekvilibria. Za druhe je to pocatecni reseni. Dale je to vhodne zvolena cenova funkce, ktera ohodnoti stav cislem.

## 3.1 Cenova funkce

Stavy jsou reprezentovany binarnim vektorem, ktery znaci ohodnoceni literalu. Stav muze byt bud validni, nebo nevalidni. Pokud se $S$ ohodnoceni formule $F$ a $F(S) = 0$, znamena to, ze formule neni splnena, coz je podle zadani nevadlidni reseni. Naopak pokud $F(S) = 1$, je formule splena a stav je tedy validni.

Cilem je najit takove validni reseni, ktere ma nejvetsi soucet vah. Validni reseni s velkou vahou by tedy mely mit velkou cenu. Je ale potreba zohlednit to, ze nejaky stav muze byt nevalidni, ale je blizko nejakeho validniho reseni s velkou cenou.

Empiristicky byly vymysleny 3 cenove funkce, ktere budou dale zkoumany.

### 3.1.1 Promenne v cenovych funkcich

Nasleduje vycet cenovych funkci. Malym c budu oznacnovat pocet splnenych klausoli a velkym C celkovy pocet klausoli. Velkym W oznacim sumu vsech vah: $W = \sum_{i=1}^{n} w_i$. Malym w oznacim sumu vah pro splnene klauzole $w = \sum_{i=1}^{n} y_i w_i$ MaxW oznacuje maximalni ze vsech vah.

### 3.1.2 Cenova funkce $cost1$

Cenova funkce $cena1$ je jednoducha funkce, ktera diky nasobeni $F(Y)$ zvyhodni validni reseni. Tedy kazde validni reseni bude lepsi nez nevalidni.

$$cost1(Y) = F(Y)W + \frac{c}{C}w$$

### 3.1.3 Cenova funkce $cost2$

Cenova funkce $cena2$ deli podle validnich a nevalidnich stavu. Pro validni reseni sectu vahy splnenych klauzoli a maximalni moznou cenu. Pro nevalidni reseni je to pak maximalni cena vynasobena pomerem splnenych a vsech klauzoli.

$$cost2(Y) = \begin{cases} W + w & \text{if } F(Y) = 1 \\ \frac{c}{C}W & \text{if } F(Y) = 0 \end{cases}$$

### 3.1.4 Cenova funkce *cost*3

Cenova funkce *cena*3 se velmi lisi od ostatnich tim, ze nijak nezvyhodnuje validni reseni. Tato funkce zvyhodnuje lepsi reseni, nikoliv vsak validni. Diky teto funkci snadneji nalezneme lepsi vysledek, ale je vice pravdepodobne, ze vysledek nebude validni.

$$cost1(Y) = c * MaxW + w$$

## 3.2 Nastaveni parametru

Nastaveni parametru je tezky ukol. Cilem bylo parametry vypocitat z instanci pomoci nejakeho vzorce. Cilem je vytvorit obecny 3SAT resic, ne jen pro nejake pevne zadane parametry.

## 3.3 Kriterium ukonceni

Vypocet skonci po tom, co se teplota snizi na predem stanovenou mez. Je tedy mozne, ze zadne validni reseni nebude nalezeno, prestoze jich v instanci existuje hodne. Je to hlavne z toho duvodu, ze na beh algoritmu ma z velke casti nahoda.

## 3.4 Opakovani vypoctu

Protoze je algoritmus nahodny na nahode, budeme vypocet opakovat 100 krat a vysledkem bude prumerna hodnota.

## 3.5 Pocatecni stav

Existuje nekolik moznosti jak zvolit pocatecni stav.

Prvni moznosti je zacit se stavem, ktery vsechny literaly odhonoti bud 0, nebo 1. Vzhledem k povaze vstupnich dat, ktere muzou byt jakekoliv bysme ale pro nejake instance mohli touto taktikou vypocet velice znekvalitnit. Lepsim resenim bude priradit kazdemu literalu nahodne bud 0 nebo 1.

Druhou moznosti je pokusit se vygenerovat nejaky validni stav, nebo pouzit sat resic a vychazet z neho. Timto zpusobem bysme pravdepodobne zacinali v lokalnim extremu, kterym se ale chceme vyvarovat.

Pocatecni stav bude tedy vygenerovan zcela nahodne.

## 3.6 Generovani naslednika

Generovani naslednika znamena vzit nejaky stav a nejakou modifikaci vytvorit novy stav, velice blizky tomu prvnimu.

Nejjednodussi metodou je negace soucasne hodnoty nahodneho literalu.

Dalsi moznosti je ta, ktera se snazi vygenerovat validni nasledniky. Vezme nahodnou nesplnenou klauzoli a zneguje nejaky nahodny literal tak, aby klauzole byla validni.

## 3.7 Optimalizace algoritmu

Po sepsani textu vyse mi doslo, ze algorimus se da naprogramovat optimalneji.

### 3.7.1 Generovani jedinecnych naslednika

Naslednici se generuji nahodne, je tedy mozne, ze se za jedno ekvilibrium (tedy za stejne teploty) vyzkousi stejny naslednik nekolikrat. To nicemu nevadi, pokazde se vygeneruje sance, s jakou se pripadne horsi reseni prijme. Prakticky mi to prijde ale velice neprakticke, proto po zmene teploty nahodne vygeneruji indexy bitu, ktere se budou menit. Tim se take vytvori horni mez pro hodnotu ekvilibira a to pocet promennych. Vetsi pocet iteraci v ekvilibriu nema cenu, protoze bysme zkouseli stejne stavy vicekrat.

### 3.7.2 Kopirovani stavu

Ciste technickou zalezitosti je pak kopirovani stavu pri generovani jeho naslednika. Prestoze je funkce memcpy v C rychla a optimalizovana, muzeme se ji vyhnout tim, ze pri generovani naslednika budeme rovnou upravovat nejlepsi stav a v pripade, ze naslednika neprijmeme, vratime nejlepsi reseni do puvodniho stavu.

### 3.7.3 Dalsi mozne optimalizace

Dalsi optimalizaci by mohlo byt drzeni si uplne nejlepsiho nalezeneho stavu. Nebo nepocitani ceny soucasneho stavu, pokud se nezmenil. To jsem ale neimplementoval.

### 3.7.4 Optimalizovany algoritmus

Zjednoduseny zdrojovy kod v jazyce C:

```
for (double t = ti; te < t; t *= cf) {
  p = generate_random_permutation();
  for (int i = 0; i < eq; i++) {
    double cost_state = cost(state);
    state[p[i]] = swap_bit(state[p[i]]);
    double d = cost_state - cost(state);
    if (d < 0 || randd() < pow(M_E, -d / t)) {
      continue;
    }
    state[p[i]] = swap_bit(state[p[i]]);
  }
}
```

# 4 Instance

## 4.1 Generator G2

Pro generovani zkusebnich dat jsem si nejprve stahnul generator G2, ktery byl na studentskem webu fit-wiki. Generator pracuje na tom principu, ze nahodne vygeneruje nejake reseni a pak podle nej dopocita klauzole, literaly i vahy. Upravene zdrojove kody tohoto generatoru jsou soucasti zdrojovych kodu. Po par experimentech bylo ale jasne, ze generator nezarucuje, ze nalezene reseni je globalni optimium. Proto jsem si naprogramoval brute force resic, ktery mi vzdy zarucene globalni optimum najde.

## 4.2 SATLIB

SATLIB knihovna obsahuje take instance 3SAT. Testovaci data maji navic pomer poctu klauzoli a poctu literalum blizici se cislu 4.3. Podle clanku Stochastic Search And Phase Transitions:AI Meets Physics od Barta Selmana jsou to tedy ty nejtezsi instance problemu.

Pro kazdy literal byla nahodne pomoci bashoveho skriptu vygenerovana vaha v rozsahu 20 az 60.

# 5 Experimentalni mereni

## 5.1 Postup mereni

Mereno bylo na notebooku s Intel(R) Core(TM) i3-2328M Processor (3M Cache, 2.20 GHz), 8GB RAM, gcc 4.9.2 (-Ofast), OS GNU/Linux Lubuntu 14.04.3 64bit.

Mereny byly instance 3SAT problemu z knihovny SATBLIB.

## 5.2 Volba parametru

### 5.2.1 Koncova teplota

Koncovou teplotu jsem empiricky zvolil na $te = 0.01$. Je to podle me dostatecne male cislo na to, aby se ke konci vypoctu prijmulo horsi reseni.

### 5.2.2 Pocatecni teplota

U pocatecni teploty jsem nezvolil zadne pevne cislo. Zalezi totiz na velikosti vah. Proto jsem empiricky zvolil tento vzorec na vypocet pocatecni teploty:

$$ti = PocetLiteralu * MaximalniVaha * 10$$

### 5.2.3 Ochlazovaci faktor

Ochlazovaci faktor $cf$, je cislo mezi 0 a 1. Me spis zajima pocet kroku, tedy kolikrat se snizi teplota a bude se pocitat v ekvilibriu. Proto pocitam ochlazovaci faktor pomoci promenne *steps*, ktera mi urci, kolikrat se snizi teplota o ochlazovaci faktor, nez se dostanu na koncovou teplotu:

$$\left(\frac{te}{ti}\right)^{\frac{1}{steps}}$$

### 5.2.4 Hodnota ekvilibria

Hodnota ekvilibria, nebo jinak kolik stavu bude vygenerovano nez se snizi teplota. V nasem pripade, kdy garantujeme to, ze se nasledujici stavy generuji nahodne, ale je zaruceno to, ze se zadny stav neopakuje nema smysl generovat vice stavu, nez je pocet promennych v instanci. Empiristicky byla vybrana hodnota rovna polovine poctu promennych, tedy:

$$ekvilibrium = pocet_promennych/2$$

## 5.3 Vyber cenove funkce

### 5.3.1 Relativni chyba

Pro testovani relativni chyby jsem pouzil dataset uf20, ktery ma 20 promennych a 91 klauzoli. Pocet klauzoli jsem ale omezil postupne na 90, 70 a 50 a zkoumal relativni chybu u cenovych funkci.

Kazdy vypocet byl opakovan 5x a byla z neho vybran nejlepsi a prumerny vysledek od kterych byla vypoctena relativni chyba. To bylo opakovano pro 200 instanci a zprumerovano.

Table 1: Relativni chyba. avg prumerna, bst nejlepsi chyba. obe v procentech

| pomer | c1avg | c1bst | c2avg | c2bst | c3avg | c3bst |
|---|---|---|---|---|---|---|
| 4.5 | 4.23 | 0.48 | 4.23 | 0.48 | 4.23 | 1.07 |
| 3.5 | 5.85 | 0.84 | 5.85 | 0.84 | 5.85 | 0 |
| 2.5 | 1.49 | 0.22 | 1.49 | 0.22 | 1.49 | 0 |

Prekvapilo mne, ze prumerne hodnoty jsou stejne pro vsechny 3 ceny. Je ale videt, ze cenova funkce *cena*3 se snazi vice nalezt globalni optimum, zatimco cenove funkce *cena*1 a *cena*2 hledaji spise nejake validni reseni a kvuli tomu mohou uvaznout v lokalnich extremech.

### 5.3.2 Prubeh vypoctu

Chovani cenovych ukazu na prikladu o 200 literalech a postupne 200, 400 a 600 klauzolich. Na grafech je znazornen postup vyvoje nejlepsiho reseni. Zelene hodnta zkoumane cenove funkce a cervene skutecna cena reseni podle zadani.

DULEZITE:
V grafech je zelene oznacena cenova funkce pouzita v simulovanem zihani.
Cervene pak skutecna cena reseni podle zadani.

Figure 1: 200 liter. / 200 klauz. cena1

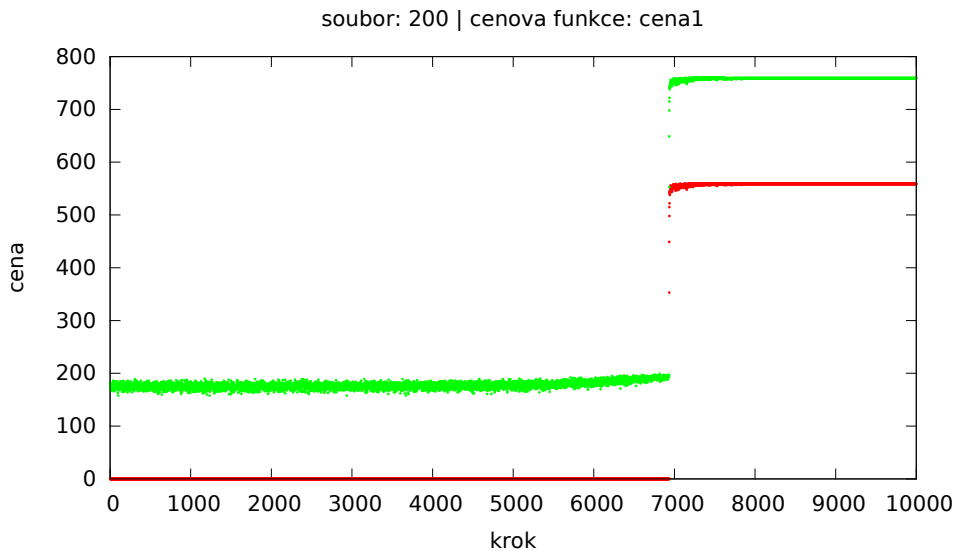



Figure 2: 200 liter. / 200 klauz. cena2

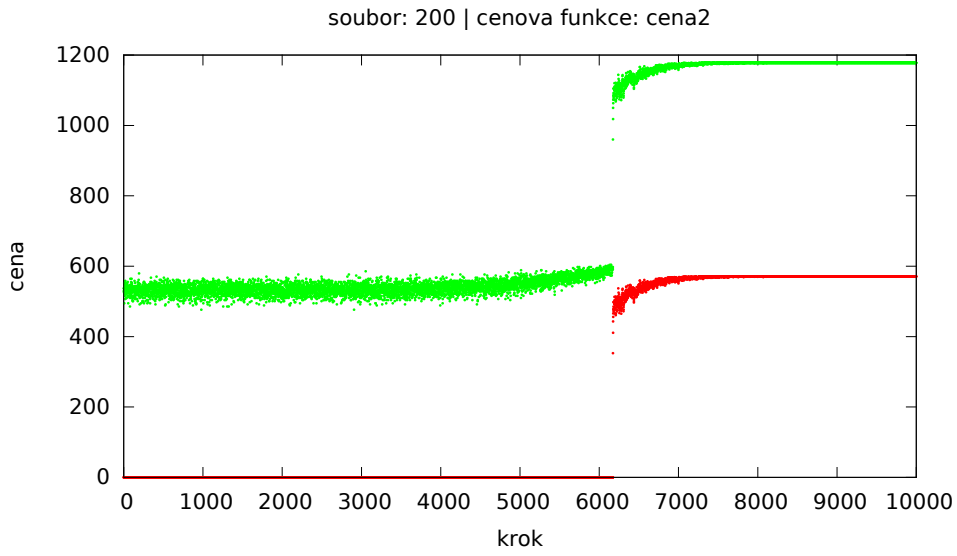


Figure 3: 200 liter. / 200 klauz. cena3

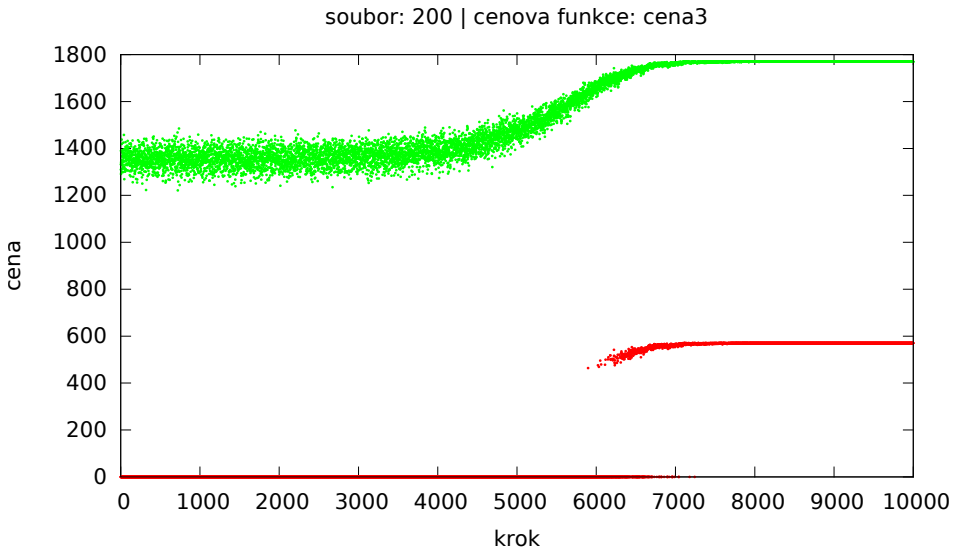



Figure 4: 200 liter. / 400 klauz. cena1

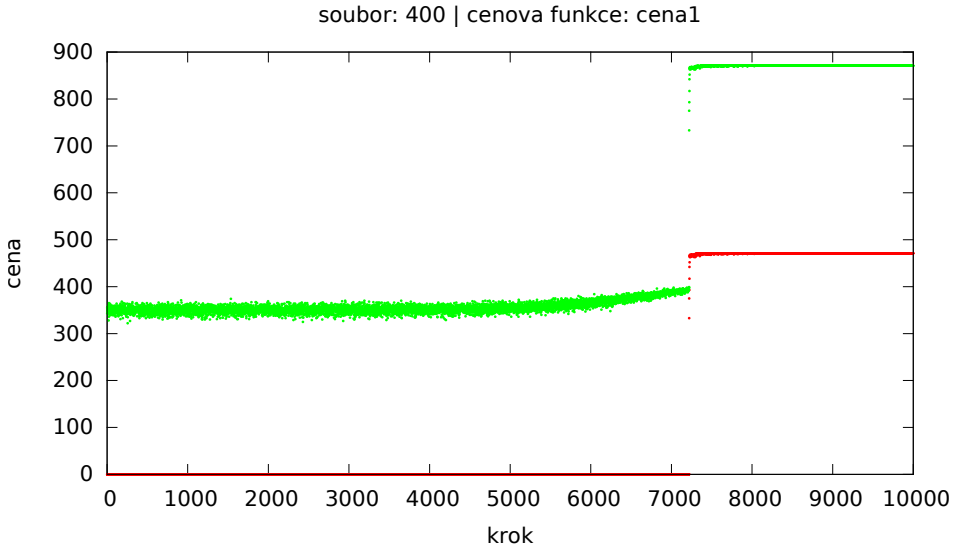


Figure 5: 200 liter. / 400 klauz. cena2

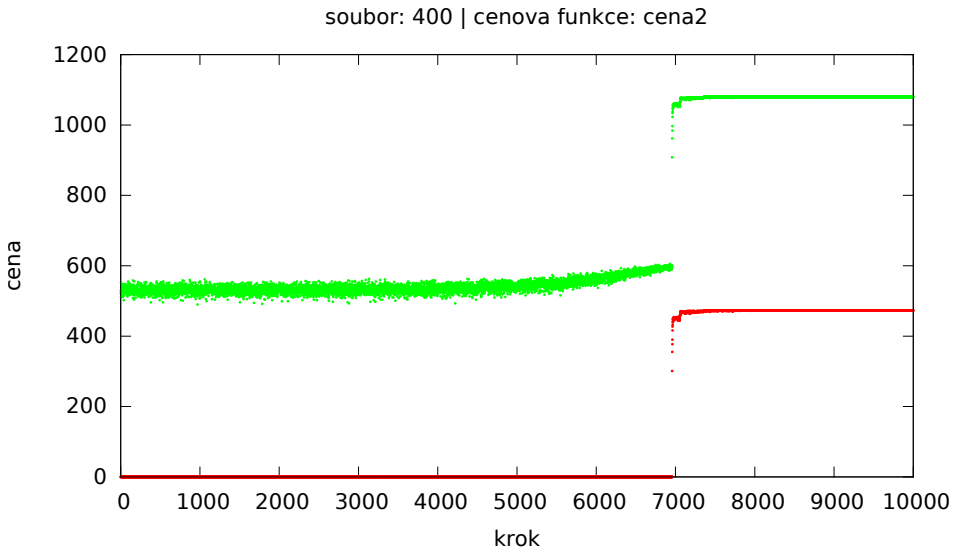



Figure 6: 200 liter. / 400 klauz. cena3

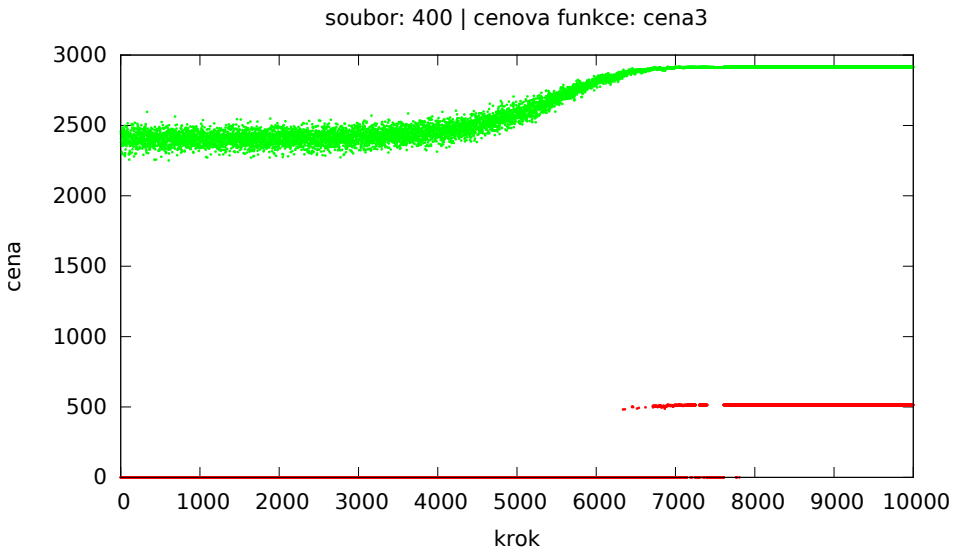


Figure 7: 200 liter. / 600 klauz. cena1

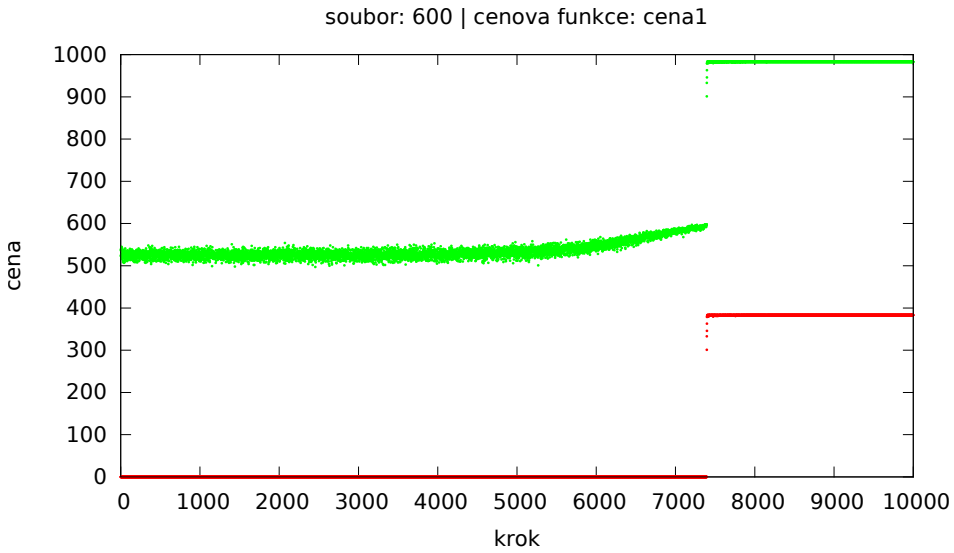



Figure 8: 200 liter. / 600 klauz. cena2

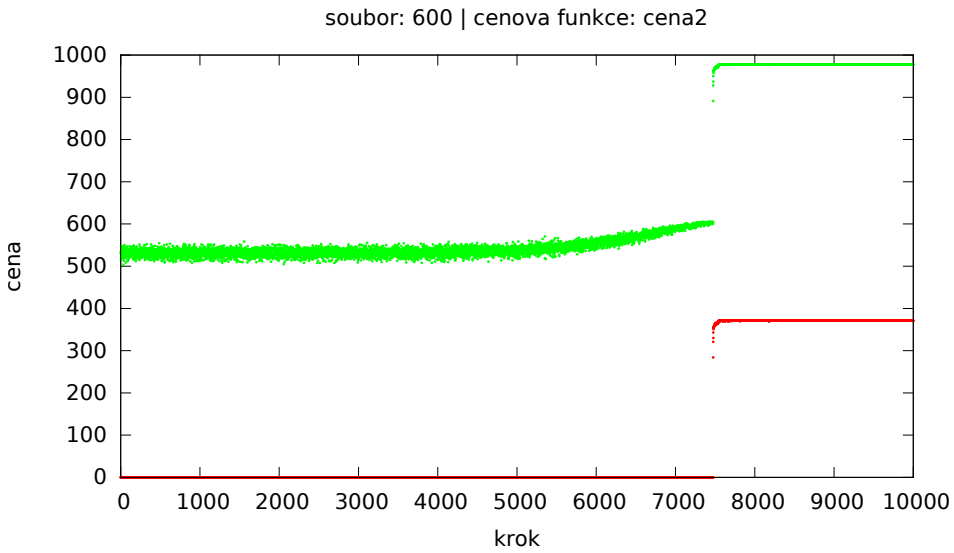
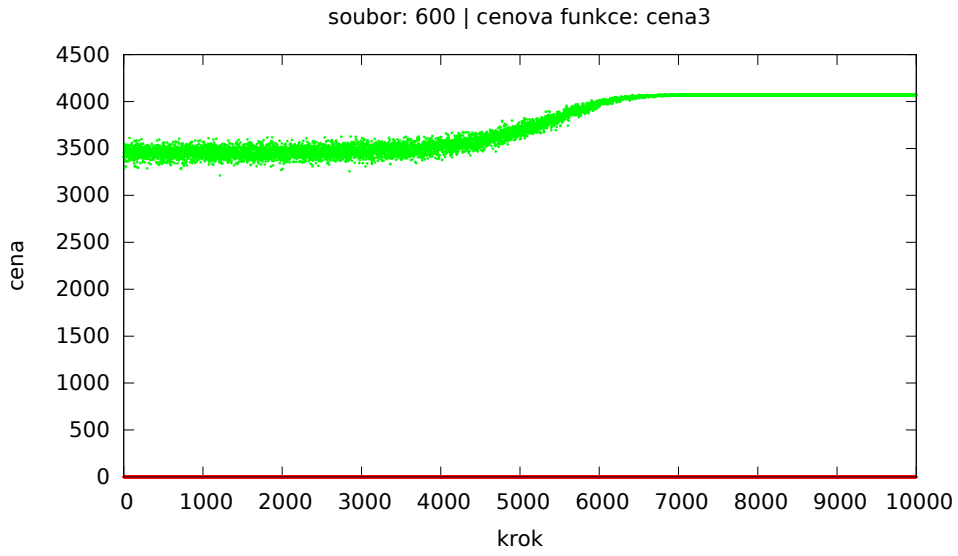


Figure 9: 200 liter. / 600 klauz. cena3




Jsou zde videt predpoklady pri definich cenovych funkci. Cenova funkce *cena*3 se odlisuje tim, ze se snazi jit po lepsim vysledku, misto hledani splnitelne formule.

## 5.4 Splnitelnost instanci

Nasledujici tabulka je vysledkem experimentu, kdyz se pro sadu dat hledalo reseni. Vypocet byl opakovan 10x a z kazde sady bylo vzato 50 instanci.

Table 2: Vliv cenove funkce na splnitelnost

| literalu/klauzoli | cena1 | cena2 | cena3 |
|---|---|---|---|
| 20/91 | 100 % | 100 % | 100 % |
| 50/218 | 100 % | 100 % | 65 % |
| 75/218 | 100 % | 100 % | 14 % |

Jak je z tabulky videt, cenova funkce *cena*3 uz reseni pro vetsi instance nalezne s malou pravdepodobnosti.

## 5.5 Rychlost vypoctu

Protoze se cenove funkce lisi velmi malo a jinak algoritmy funguji stejne, rychlost vypoctu byla temer totozna pro stejne velke instance.

# 6 Zaver

3SAT je velmi slozity problem, pokud mame velke instance. Pro male instance muzeme pouzit brute force reseni. Pro vetsi muzeme pouzit cenovou funkci *cena*3, ktera neznevyhoduje validni reseni a tim nam pomuze najit lepsich vysledku. Nebo je mozne pouzit jinou cenovou funkci na podobnem principu.

Cenove funkce *cena*1 a *cena*2 agresivneji hledaji validni reseni. Pokud nechceme vypocet provadet obzvlaste dlouho, je to temer nutnost pro nalezeni validniho reseni.

Cely problem bych shrnul na to, ze je potreba brat v potaz velikost instanci, kolik mam casu na vypocet a jestli mi staci horsi, ale validni reseni, nebo se budu snazit o lepsi reseni i za cenu toho, ze dlouho nemusim validni reseni nalezt.