Implementace DPLL solveru (předp, že vstup neobsahuje unit klauzule)

reprezentace proměnných : 1, 2, 3, 4, ...

reprezentace literálů : proměnná k, literály $2*k$ pozitivní
$2*k+1$ negativní

operace s literály: var(l) ⇒ l/2
neg(l) ⇒ l ⊕ 1 (⊕ je XOR)
sgn(l) ⇒ (l & 1) == 0 (kontrola dolního bitu)

~~////~~

klauzule : pole literálů
první dva literály jsou sledovaný (tj. watched)

watched literál : ~~je~~ není False (ev. s jednou výjimkou)
sledujeme, jestli se nestane False. Pokud ano, najdeme jiný watched literál pro danou klauzuli: když to nejde, je klauzule unit

Pamatujeme si

vars # proměnných
clauses # klauzule

watches # pro každý literál ~~ten~~ p je watches[p] seznam klauzulí, ve kterých sledujeme $\bar{p}$ (tj. $\bar{p}$ je watched v dané klauzuli
watches je tedy pole s (vars*2 + 2) prvky.

prop_q # fronta literálů, které jsme ohodnotily ~~do~~ na true, a je nutné toto ohodnocení promítnout do formule

assigns # pro proměnnou i je assigns[i] ohodnocení této proměnné, i to hodnota z množiny {True, False, None}
assigns je pole s vars+1 prvky

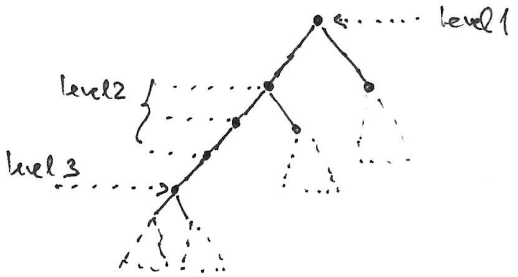
level # level[i] je úroveň proměnné i (viz dále)

tries # tries[i] je počet pokusů o ohodnocení proměnné i (z mezi množiny {0, 1, 2, 3}.

trail # zásobník obsahující posloupnost literálů ohodnocených na true, uložen pomocí pole
trail_lim # ~~množina indexů~~
zásobník indexů do trail, na kterých jsou tam literály rozhodovacích proměnných (viz dále)

Proměnná je rozhodovací, je-li ohodnocena ve větvení ve stromu rekurze. Jinak je proměnná ohodnocena unit propagací.

Pojmy výše dávají smysl v nějakém uzlu ve stromu rekurze algoritmu DPLL. Level proměnné je počet rozhodovacích proměnných ~~ocho~~ ohodnocených na cestě z kořene do vrcholu, kde je daná proměnná ohodnocena.



Procedury

search()

```
while
    conflict ← propagate()
    if conflict
        ok ← backtrack()
        if not ok then return false                    // UNSAT
    else
        if  Pen(trail) = vars  then return true        // SAT
        else
            pick a literal p to be satisfied
            assume(p).
```

```
assume (p)      # p ... literál
    push (trail_lim, len (trail))      # push ... vlož na zásobník
    return enqueue (p)
```

```
enqueue (p)     # p ... literál
    if value (p) != False then return false
    if value (p) = True  then return true
    else    # tady je value (p) = None
        v ← var(p)
        assigns [v] ← sgn (p) ? True : False
        level [v] ← len (trail_lim)
        tries [v] += 1
        insert p into prop-q ; add p to trail
        return true
```

value(p) = { None : assigns [var(p)] = None ; True : ? ; False : ? } — pravdivost p podle ohodnocení var(p) v assigns

```
propagate ()
    while prop-q is not empty
        p ← dequeue (prop-q)
        tmp ← watches (p)      # copy
        remove all from watches[p]
        for i in 0..<len(tmp)
            ok ← propagate_clause (tmp[i], p)
            if not ok    # conflict: the clause tmp[i] is empty
                For k in (i+1)..<len(tmp)
                    insert tmp[k] into watches[p]
                remove all from prop-q
                return false
        return true
```

klausule, ve kterých jsme nepřiřazovali $\bar{p}$ z pravidel dvou pozic musí zůstat ve watches[p]

propagate_clause (cl, p)

```
# make sure  cl[1] = ¬p
if cl[1] ≠ neg(p)  then swap(cl[0], cl[1])

if value(cl[0]) = True  then return true

# look for a new watched literal
for i in 2.. len(cl)
    if value(cl[i]) ≠ False
        swap(cl[i], cl[1])
        insert cl into watches[neg(cl[i])]
        return true
# cl may be unit
insert cl into watches[p]
return enqueue(cl[0]).
```

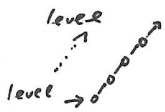
// poland je cl[0] false, vrati equeue false a nasla konflikt, poluzi je cl[0] : None, pak je to unit propagace.

undo_one()

```
p ← pop(trail)
v ← var(p)
if p is decision variable then pop(trail_lim)      # decision? len(trail) = top(trail_lim) + 1
assigns[v] ← None
level[v] ← -1
tries[v] ← 0
```

backtrack()

```
while
    if len(trail_lim) = 0 then return false          // vrátil jsem se nad kořen
    while top(trail) is not decision variable
        undo_one()                                   } ⟶
    p ← top(trail)
    if tries(var(p)) < 2 and enqueue(neg(p))
        return true                                  ⟶
    else
        undo_one()
```



Hlavní myšlenky CDCL solveru [Conflict driven clause learning]

Rozšíření DPLL solveru o učení se nových klausulí a nechronologický backtracking

konflikt = klausule je vyprázdněna (enqueue vrátil false) a tedy propagate clause

→ konfliktní klausule

Nechť je konfliktní klausule $(\ell_1 \vee \ell_2 \vee \ell_3 \dots \vee \ell_k)$. Konflikt je způsoben pravdivostí literálů $\bar{\ell}_1, \bar{\ell}_2, \dots \bar{\ell}_k$.

tyto zpracujeme následovně:

(A) – je-li $var(\bar{\ell}_1)$ rozhodovací proměnná, přidáme ℓ_1 do naučené klausule.

(B) – je-li $var(\bar{\ell}_1)$ ohodnocena unit propagací, označme $source(\bar{\ell}_1) = \{x_1 \vee x_2 \vee x_3 \dots\}$ klausuli, která byla transformována na $\{\bar{\ell}_1\}$, což vyvolalo unit propagaci literálu $\bar{\ell}_1$.
Potom k literálům z konfliktní klausule přidáme $\bar{x}_1, \bar{x}_2, \bar{x}_3 \dots$

předchozí dva kroky opakujeme (přičemž na začátku je naučená klausule prázdná; ~~kaž~~ každá proměnná může mít v naučené klausuli nejvýše jeden literál).

Dostaneme naučenou klauzuli $(v_1 \vee v_2 \vee v_3 \dots) = C$.

→ přidáme ji ke vstupní formuli [vidíme, že C musí být splněna, abychom předešli konfliktu způsobenému konfliktní klausulí]

→ všechny proměnné v C jsou rozhodovací; vybereme tu která má maximální level. Provedeme backtracking až do místa, kde tuto proměnnou ohodnocujeme [nonchronological backtracking].

K literálu, který byl splněn unit propagací, a příslušnou klauzuli z bodu (B) můžeme v solveru pamatovat.

Více informací lze nalézt v článku:

N. Eén, N. Sörensson. An extensible SAT-solver.

Tento článek popisuje solver MiniSAT, jehož zdrojové kódy (v C, C++) jsou také k dispozici. Odkaz na stránku solveru MiniSAT je na webu předmětu.