Logică și structuri discrete

Logică propozițională

Algoritmul Davis-Putnam-Logemann-Loveland

https://tinyurl.com/lecturesLSD

Cum determinăm dacă o formulă e *realizabilă*?

algoritm folosit în rezolvarea multor probleme

Realizabilitatea unei formule propoziționale (satisfiability)

Se dă o formulă în *logică propozițională*.

Există vreo atribuire de valori de adevăr care o face adevărată ?
= e *realizabilă* (engl. *satisfiable*) formula ?

$$
\begin{aligned}
&(a \vee \neg b \vee \neg d)\\
\wedge\ &(\neg a \vee \neg b)\\
\wedge\ &(\neg a \vee c \vee \neg d)\\
\wedge\ &(\neg a \vee b \vee c)
\end{aligned}
$$

Găsiți o atribuire care satisface formula?

Formula e în *formă normală conjunctivă* (conjunctive normal form)
= conjuncție de disjuncții de *literali* (pozitiv sau negat)

Fiecare conjunct (linie de mai sus) se numește *clauză*

Reguli în determinarea realizabilității

Simplificăm problema, știind că vrem formula adevărată
(NU se aplică la simplificarea formulelor în formule echivalente!)

R1) Un literal *singur într-o clauză* are o singură valoare utilă:

în $a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$ — a trebuie să fie T

în $(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$ — b trebuie să fie F

(altfel formula are valoarea F)

Reguli pentru determinarea realizabilității (cont.)

R2a) Dacă un literal e T, *pot fi șterse clauzele* în care apare
(ele sunt adevărate, le-am rezolvat)

R2b) Dacă un literal e F, *el poate fi șters* din clauzele în care apare
(nu poate face clauza adevărată)

Exemplele anterioare se simplifică:

$$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad \stackrel{a=\mathsf{T}}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$$

$$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \quad \stackrel{b=\mathsf{F}}{\rightarrow} \quad a$$

(și de aici $a = T$, deci formula e realizabilă)

Reguli pentru determinarea realizabilității (cont.)

R3) Dacă *nu mai sunt clauze*, formula e realizabilă
(cu atribuirea construită)

Dacă obținem o *clauză vidă*, formula *nu e realizabilă*
(fiind vidă, nu putem s-o facem T)

$(a \vee b) \wedge a \wedge (a \vee \neg b \vee c) \quad \stackrel{a=\mathrm{T}}{\rightarrow} \quad (\mathrm{T} \vee b) \wedge \mathrm{T} \wedge (\mathrm{T} \vee \neg b \vee c) \xrightarrow{R2a}$
ștergem toate clauzele (conțin T, le-am rezolvat)
$\Rightarrow$ formulă realizabilă (cu $a = \mathrm{T}$)

$a \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$
$\stackrel{a=\mathrm{T}}{\rightarrow} b \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$
$\stackrel{b=\mathrm{T}}{\rightarrow} c \wedge \neg c \quad \stackrel{c=\mathrm{T}}{\rightarrow} \quad \emptyset \quad$ ($\neg c$ devine clauza vidă $\Rightarrow$ nerealizabilă)

Reguli pentru determinarea realizabilității (cont.)

Dacă *nu mai putem face reduceri* după aceste reguli ?

$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \stackrel{a=T}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$??

R4) Alegem o variabilă și *despărțim pe cazuri* (încercăm):

- cu valoarea F
- cu valoarea T

O soluție pentru *oricare* caz e bună (nu căutăm o soluție anume).

Dacă *nicicare caz* nu are soluție, formula *nu e realizabilă*.

Un algoritm de rezolvare

Problema are ca date:

- lista clauzelor (formula)
- mulțimea variabilelor deja atribuite (inițial vidă)

Regulile 1 și 2 ne *reduc problema la una mai simplă*
(mai puține necunoscute sau clauze mai puține și/sau mai simple)

Regula 3 spune când ne oprim (avem răspunsul).

Regula 4 reduce problema la rezolvarea a *două probleme mai simple*
(cu o necunoscută mai puțin)

Reducerea problemei la *aceeași problemă cu date mai simple*
(una sau mai multe instanțe) înseamnă că problema e *recursivă*.

Obligatoriu: trebuie să avem și o *condiție de oprire*

Algoritmul Davis-Putnam-Logemann-Loveland (1962)

```
function solve(truelit: lit set, clauses: clause list)
(truelit, clauses) = simplify(truelit, clauses) (* R1, R2 *)
if clauses = lista vidă then
   return  truelit; (* R3: realizabila, returneaza atribuirile *)
if clauses conține clauza vidă then
   raise Unsat;    (* R3: nerealizabila *)
if clauses conține clauză cu unic literal a then
   solve (truelit ∪{a}, clauses)        (* R1: a trebuie să fie T *)
else
   try solve (truelit ∪{¬a}, clauses);    (* R4: încearcă a=F *)
   with Unsat → solve (truelit ∪{a}, clauses);   (* încearcă T *)
```

Rezolvitoarele (*SAT solvers/checkers*) moderne pot rezolva formule cu milioane de variabile (folosind optimizări)

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literali)
- *mulțimea* literalilor cu valoare T

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite
- *adăugarea* unui literal la mulțimea celor atribuite
- *parcurgerea* literalilor dintr-o listă (clauză)
- *eliminarea* unui literal dintr-o listă (clauză)
- *eliminarea* unei clauze dintr-o listă (formula)

Cum reprezentăm un literal ?

un șir (numele variabilei) etichetat cu P (pozitiv) / N (negativ)

```
module L = struct
  type t = P of string | N of string (* pozitiv / negat *)
  let compare = compare  (* fct. std. Pervasives.compare *)
  let neg = function   (* negare = schimba eticheta *)
    | P s -> N s
    | N s -> P s
 end
```

(cod după Conchon et. al, SAT-MICRO, 2008)

Sau reprezentăm o propoziție p_k prin indicele întreg $k \in \mathbb{N}^*$ și folosi numere negative pentru negație (formatul standard DIMACS)

```
module L = struct
  type t = int
  let compare = compare
  let neg x = -x
end

module S = Set.Make(L)  (* pentru multimi de literali *)
```

Simplificarea unei clauze

`tlits` = mulțimea literalilor (cunoscuți deja ca) adevărați

R2a: când găsim un literal adevărat, putem elimina clauza (e T)
⇒ nu mai continuăm prelucrarea, semnalăm *excepția* `Exit`
va fi tratată de funcția apelantă

Altfel, eliminăm un literal dacă e fals (R2b)
i.e., dacă apare negat în mulțimea celor adevărate, `tlits`
deci dacă nu apare negat în `tlits` îl păstrăm

```
let filter_clause tlits =
  List.filter (fun lit ->
    if S.mem lit tlits then raise Exit (* clauza adevarata *)
    else not (S.mem (L.neg lit) tlits)) (* retine daca nu e F *)
```

Simplificarea listei de clauze

Acumulăm cu `List.fold_left` o *pereche* de valori:
mulțimea literalilor adevărați `tlits`, lista clauzelor simplificate `clst`

```
let rec simplify truelits = List.fold_left
  (fun (tlits, clst) cl -> (* (lit,clauze) acumul.+clauza crt.*)
    try match filter_clause tlits cl with
     | [] -> raise Unsat (* clauza vida -> nerealizabila *)
     | [lit] -> simplify (S.add lit tlits) clst (*reia cu lit=T*)
     | rstcl -> (tlits, rstcl::clst) (* adauga clauza simplif.*)
    with Exit -> (tlits, clst) (* ignora clauza care a fost T *)
  ) (truelits, [])
```

Dacă `filter_clause` dă un unic literal, se adaugă la cele adevărate
și reluăm simplificarea clauzelor deja prelucrate

Dacă returnează lista vidă, toată formula e nerealizabilă

Dacă produce excepția `Exit`, eliminăm clauza (e adevărată)

Altfel, adăugăm clauza simplificată la listă

Verificarea propriu-zisă

Dacă simplificând obținem lista vidă de clauze, returnăm mulțimea literalilor adevărați (restul nu contează)

Altfel, cu primul literal din prima clauză încercăm ambele valori
dacă prima încercare dă excepția `Unsat`, încercăm și a doua

```
let sat =
  let rec sat1 tlits clist = match simplify tlits clist with
    | (tlits, (lit::cl)::clst) -> (* luam primul literal *)
      S.union tlits (* return.lit. deja T + aflati mai jos *)
        (try sat1 (S.singleton (L.neg lit)) (cl::clst) (*lit=F*)
        with Unsat -> sat1 (S.singleton lit) clst) (*sau lit=T*)
    | (tlits, _) -> tlits (* _ va fi []; return. literalii T *)
  in sat1 S.empty          (* initial nu stim lit. T *)

let res = sat [[1;2;-3]; [-1;3]; [1;-2]] |> S.elements
val res : S.elt list = [-3; -2; -1]
```

$(p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_3) \vee (\neg p_1 \vee p_2)$ e SAT cu $p_1 = p_2 = p_3 = \mathrm{F}$