Logică și structuri discrete

Logică propoziţională

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

6 noiembrie 2017

În cursul de azi

Cum codificăm probleme în logică propoziţională

Cum determinăm dacă o formulă e *realizabilă*
algoritm folosit în rezolvarea multor probleme

Ce înseamnă o *demonstraţie* logică?

Unde aplicăm logica booleană?

Calculatoarele sunt construite din *circuite logice*
⇒ realizează aceleași *funcții* ca în logică (ȘI, SAU, NU)

Numerele sunt reprezentate în calculator *în baza 2*
⇒ valori *boolene* / biți: (0 sau 1, F sau T)

Aritmetica pe numere e implementată prin circuite logice

```
unsigned add(unsigned a, unsigned b) {
  return b ? add(a^b, (a&b) << 1) : a;
}
```

```
let rec add a b =
  if b = 0 then a else add (a lxor b) ((a land b) lsl 1)
```

Mulțimile pot fi reprezentate prin *vectori* de valori *boolene*
pentru fiecare element: T/F, face sau nu parte din mulțime ?

⇒ reprezentăm noțiuni (reale sau matematice) în *logică booleană*

Aplicații: Căutare / explorarea stărilor / planificarea

Putem ajunge la o poziție câștigătoare într-un joc ?

Există un drum între două noduri într-un graf ?

Planificare = găsirea unui șir de *acțiuni* care duc la o *țintă*
de la ordonarea unor acțiuni între care există constrângeri
până la comportamentul unor roboți inteligenți / autonomi

În general: într-un sistem descris prin *stări* și *acțiuni* (tranziții),
cum găsim o cale de la o *stare inițială* la o *stare țintă* (finală) ?

Exemplu: jocul cu ordonarea a 3x3 piese

Se poate reface ordinea? Din câte mutări?

2		5
1	3	4
8	6	7

Numerotăm pozițiile:

123
456
789

Trebuie să codificăm *starea* (configurația): poziția pieselor

Putem alege un vector $\bar{v} = (p_1, p_2, \ldots, p_9)$, $p_i \in [0..8]$ (0 = liber)

fiecare valoare p_i poate fi reprezentată boolean (cu 4 biți)

$p_1 = 2 \wedge p_2 = 0 \wedge p_3 = 5 \wedge \ldots$

sau direct cu booleni p_{ij} = piesa i (1..8) e pe poziția j (1..9)

$\neg p_{11} \wedge \neg p_{12} \wedge \neg p_{13} \wedge p_{14} \wedge \ldots$ piesa 1 e pe poz. 4 (doar acolo)

$p_{21} \wedge \neg p_{22} \wedge \neg p_{23} \wedge \neg p_{24} \wedge \ldots$ piesa 2 e pe poz. 1

...

O *stare* poate fi descrisă printr-o *formulă propozițională*

Cum reprezentăm o mutare?

O mutare se face între 2 stări: *starea curentă* și *starea următoare*, reprezentate prin *vectori de stare* $\bar{v}$ și $\bar{v}'$, fiecare cu propozițiile sale:

$\bar{v} = (p_{11}, p_{12}, ...)$ și $\bar{v}' = (p'_{11}, p'_{12}, ...)$ și

Sunt 12 perechi de poziții vecine: $\{(1,2),(1,4),\ldots(8,9)\}$

123
456
789

Introducem 12 propoziții: m_{12} = mutarea între poz. 1 și 2, etc.

Dacă facem mutarea m_{12}:

piesa de pe poz. 1 în $\bar{v}$ va fi pe poz. 2 în $\bar{v}'$ și reciproc

piesele de pe alte poziții (3, 4, ... 9) rămân pe loc

Reprezentarea unei mutări (cont.)

p_{ij} =piesa i e pe poziția j

(1) *Dacă* facem mutarea m_{12}:

piesa de pe poz. 1 în $\bar{v}$ va fi pe poz. 2 în $\bar{v}'$ și reciproc

$(m_{12} \to p'_{11} = p_{12})$ 1 va fi pe poz. 1 doar dacă era pe poz. 2

$\wedge\ (m_{12} \to p'_{12} = p_{11})$ 1 va fi pe poz. 2 doar dacă era pe poz. 1

$\wedge\ (m_{12} \to p'_{21} = p_{22})$ 2 va fi pe poz. 1 doar dacă era pe poz. 2

$\wedge\ (m_{12} \to p'_{22} = p_{21})$ 2 va fi pe poz. 2 doar dacă era pe poz. 1

...

Rescriem în CNF:

$(\neg m_{12} \vee \neg p'_{11} \vee p_{12}) \wedge (\neg m_{12} \vee p'_{11} \vee \neg p_{12})$

$\wedge\ (\neg m_{12} \vee \neg p'_{12} \vee p_{11}) \wedge (\neg m_{12} \vee p'_{12} \vee \neg p_{11})$

$\wedge\ (\neg m_{12} \vee \neg p'_{21} \vee p_{22}) \wedge (\neg m_{21} \vee p'_{21} \vee \neg p_{22})$ ...

Reprezentarea unei mutări (cont.)

p_{ij} =piesa i e pe poziția j

(1) *Dacă* facem mutarea m_{12}:

piesa de pe poz. 1 în $\bar{v}$ va fi pe poz. 2 în $\bar{v}'$ și reciproc

$(m_{12} \to p'_{11} = p_{12})$ 1 va fi pe poz. 1 doar dacă era pe poz. 2

$\wedge\ (m_{12} \to p'_{12} = p_{11})$ 1 va fi pe poz. 2 doar dacă era pe poz. 1

$\wedge\ (m_{12} \to p'_{21} = p_{22})$ 2 va fi pe poz. 1 doar dacă era pe poz. 2

$\wedge\ (m_{12} \to p'_{22} = p_{21})$ 2 va fi pe poz. 2 doar dacă era pe poz. 1

...

Rescriem în CNF:

$(\neg m_{12} \vee \neg p'_{11} \vee p_{12}) \wedge (\neg m_{12} \vee p'_{11} \vee \neg p_{12})$

$\wedge\ (\neg m_{12} \vee \neg p'_{12} \vee p_{11}) \wedge (\neg m_{12} \vee p'_{12} \vee \neg p_{11})$

$\wedge\ (\neg m_{12} \vee \neg p'_{21} \vee p_{22}) \wedge (\neg m_{21} \vee p'_{21} \vee \neg p_{22})$ ...

(2) *Dacă* facem mutarea m_{12}

piesele de pe alte poziții (3, 4, ... 9) rămân pe loc

$(\neg m_{12} \vee \neg p'_{13} \vee p_{13}) \wedge (\neg m_{12} \vee p'_{13} \vee \neg p_{13})$ piesa 1 pe poz. 3

$\wedge\ (\neg m_{12} \vee \neg p'_{23} \vee p_{23}) \wedge (\neg m_{12} \vee p'_{23} \vee \neg p_{23})$ piesa 2 pe poz. 3

...

la fel pentru toate cele *12 mutări*

Constrângeri pentru mutări

(3) Nu putem face ~~două mutări~~ odată.

Deci pentru orice două mutări distincte, $12 \cdot (12-1)/2$ perechi:

$\neg(m_{12} \wedge m_{23})$ adică $(\neg m_{12} \vee \neg m_{23})$

$\neg(m_{12} \wedge m_{14})$ adica $(\neg m_{12} \vee \neg m_{14})$ ...

Constrângeri pentru mutări

(3) Nu putem face ~~*două mutări*~~ odată.
Deci pentru orice două mutări distincte, $12 \cdot (12-1)/2$ perechi:

$\neg(m_{12} \wedge m_{23})$ adică $(\neg m_{12} \vee \neg m_{23})$
$\neg(m_{12} \wedge m_{14})$ adica $(\neg m_{12} \vee \neg m_{14})$ ...

(4) *Trebuie* să facem una din mutările asociate unei poziții libere:

$(\neg p_{11} \wedge \neg p_{21} \wedge ... \wedge \neg p_{81} \rightarrow m_{12} \vee m_{14})$ poz. 1 liberă
$\wedge\ (\neg p_{12} \wedge \neg p_{22} \wedge ... \wedge \neg p_{82} \rightarrow m_{12} \vee m_{23} \vee m_{25})$ ... poz. 2 liberă

sau în CNF:

$(p_{11} \vee p_{21} \vee ... \vee p_{81} \vee m_{12} \vee m_{14})$
$\wedge\ (p_{12} \vee p_{22} \vee ... \vee p_{82} \vee m_{12} \vee m_{23} \vee m_{25})$ ...

Constrângeri pentru mutări

(3) Nu putem face ~~două mutări~~ odată.
Deci pentru orice două mutări distincte, $12 \cdot (12-1)/2$ perechi:

$$\neg(m_{12} \wedge m_{23}) \quad \text{adică} \quad (\neg m_{12} \vee \neg m_{23})$$
$$\neg(m_{12} \wedge m_{14}) \quad \text{adica} \quad (\neg m_{12} \vee \neg m_{14}) \quad \ldots$$

(4) *Trebuie* să facem una din mutările asociate unei poziții libere:

$$(\neg p_{11} \wedge \neg p_{21} \wedge \ldots \wedge \neg p_{81} \rightarrow m_{12} \vee m_{14}) \quad \text{poz. 1 liberă}$$
$$\wedge\ (\neg p_{12} \wedge \neg p_{22} \wedge \ldots \wedge \neg p_{82} \rightarrow m_{12} \vee m_{23} \vee m_{25}) \ldots \quad \text{poz. 2 liberă}$$

sau în CNF:

$$(p_{11} \vee p_{21} \vee \ldots \vee p_{81} \vee m_{12} \vee m_{14})$$
$$\wedge\ (p_{12} \vee p_{22} \vee \ldots \vee p_{82} \vee m_{12} \vee m_{23} \vee m_{25}) \quad \ldots$$

Putem face o mutare *doar* dacă una din cele două poziții e *liberă*:
e implicată de constrângerile de mai sus: *trebuie* făcută o mutare pentru o poziție liberă și nu putem face două mutări

Relația de tranziție

Constrângerile de mai sus definesc *legătura* dintre starea curentă (vectorul de propoziții $\bar{v}$) și starea următoare (vectorul $\bar{v}'$).

am introdus și propozițiile auxiliare m_{ij} (care nu țin de stare), dar le-am putea elimina

Legătură = *relație* . Nu e o ~~*funcție*~~ pentru că putem face mai multe mutări (2, 3 sau 4) și ajunge în mai multe stări succesor.

Combinând constrângerile (1)-(4) avem deci o formulă $R(\bar{v}, \bar{v}')$ care depinde de propozițiile p_{ij} din vectorul de stare $\bar{v}$ și p'_{ij} din $\bar{v}'$:

Câte clauze are formula, așa cum am scris-o?

$R(\bar{v}, \bar{v}')$ e *relația de tranziție*: descrie cum evoluează sistemul.

Relația de tranziție poate fi descrisă printr-o formulă propozițională

Soluția problemei e un șir de mutări

Să presupunem că am reușit să ordonăm piesele în k mutări.

Avem deci $k+1$ stări (configurații): $\bar{v}^0 \to \bar{v}^1 \to \ldots \to \bar{v}^k$.

- *Starea inițială* $\bar{v}^0 = (p^0_{11}, p^0_{12}, \ldots)$ satisface o formulă:
$S_i(\bar{v}^0) = p^0_{14} \wedge p^0_{21} \wedge \ldots$ (1 pe poz. 4, 2 pe poz. 1, ...)

- *Starea finală* $\bar{v}^k = (p^k_{11}, p^k_{12}, \ldots)$ e descrisă tot printr-o formulă:
$S_f(\bar{v}^k) = p^k_{11} \wedge p^k_{22} \wedge \ldots$ (1 pe poz. 1, 2 pe poz. 2, ...)

- Stările succesive sunt legate prin mutări (*relația de tranziție*):

$$\boxed{S_i(\bar{v}^0) \wedge R(\bar{v}^0, \bar{v}^1) \wedge R(\bar{v}^1, \bar{v}^2) \wedge \ldots \wedge R(\bar{v}^{k-1}, \bar{v}^k) \wedge S_f(\bar{v}^k)}$$

Există soluție în k mutări *dacă și numai dacă* formula e *realizabilă.*

Soluția problemei

Putem rezolva deci problema (și multe altele) exprimând-o ca o problemă de *logică propozițională*:

Descriem o *stare* ca formulă propozițională.
în particular, starea inițială S_i și cea țintă S_f

Descriem o *mutare* între stări ca formulă propozițională.
relația de tranziție $R(\bar{v}, \bar{v}')$ între doi vectori de stare

$\Rightarrow$ Găsim un *plan* de lungime minimă căutând succesiv soluții pentru formule tot mai complexe: 1, 2, 3, ... pași

$S_i(\bar{v}^0) \wedge R(\bar{v}^0, \bar{v}^1) \wedge S_f(\bar{v}^1)$ 1 pas: $\bar{v}^0 \rightarrow \bar{v}^1$

$S_i(\bar{v}^0) \wedge R(\bar{v}^0, \bar{v}^1) \wedge R(\bar{v}^1, \bar{v}^2) \wedge S_f(\bar{v}^2)$ 2 pași: $\bar{v}^0 \rightarrow \bar{v}^1 \rightarrow \bar{v}^2$

etc.

În funcție de problemă, există și alți algoritmi, dedicați.
Aici am redus problema la o exprimare *simplă*, fundamentală:
determinarea realizabilității unei formule boolene (problema SAT)

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:

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

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

(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)

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

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

Reguli pentru determinarea realizabilității (cont.)

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

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

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 $\cup\{a\}$, clauses) (* R1: a trebuie să fie T *)
else
  try solve (truelit $\cup\{\neg a\}$, clauses); (* R4: încearcă a=F *)
  with Unsat $\rightarrow$ solve (truelit $\cup\{a\}$, 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 $p_1 = p_2 = p_3 = \mathsf{F}$

Complexitatea realizabilității

O formulă cu n propoziții are 2^n atribuiri
$\Rightarrow$ timp *exponențial* încercând toate

O atribuire dată se verifică în timp *liniar* (în dimensiunea formulei)
parcurgem formula o dată și obținem valoarea

În general, a *verifica* o soluție e (mult) mai simplu decât a o *găsi*.

P = clasa problemelor care pot fi *rezolvate* în timp polinomial (relativ la dimensiunea problemei)
căutare în tablou nesortat: liniar, $O(n)$
sortare $O(n \log n)$ (eficient), $O(n^2)$ (nu folosiți)
toate drumurile minime în graf $O(n^3)$

NP (nondeterministic polynomial time) = clasa problemelor pentru care o soluție ("ghicită", dată) poate fi *verificată* în timp polinomial

Probleme NP-complete

Unele probleme din clasa *NP* sunt mai dificile decât altele.

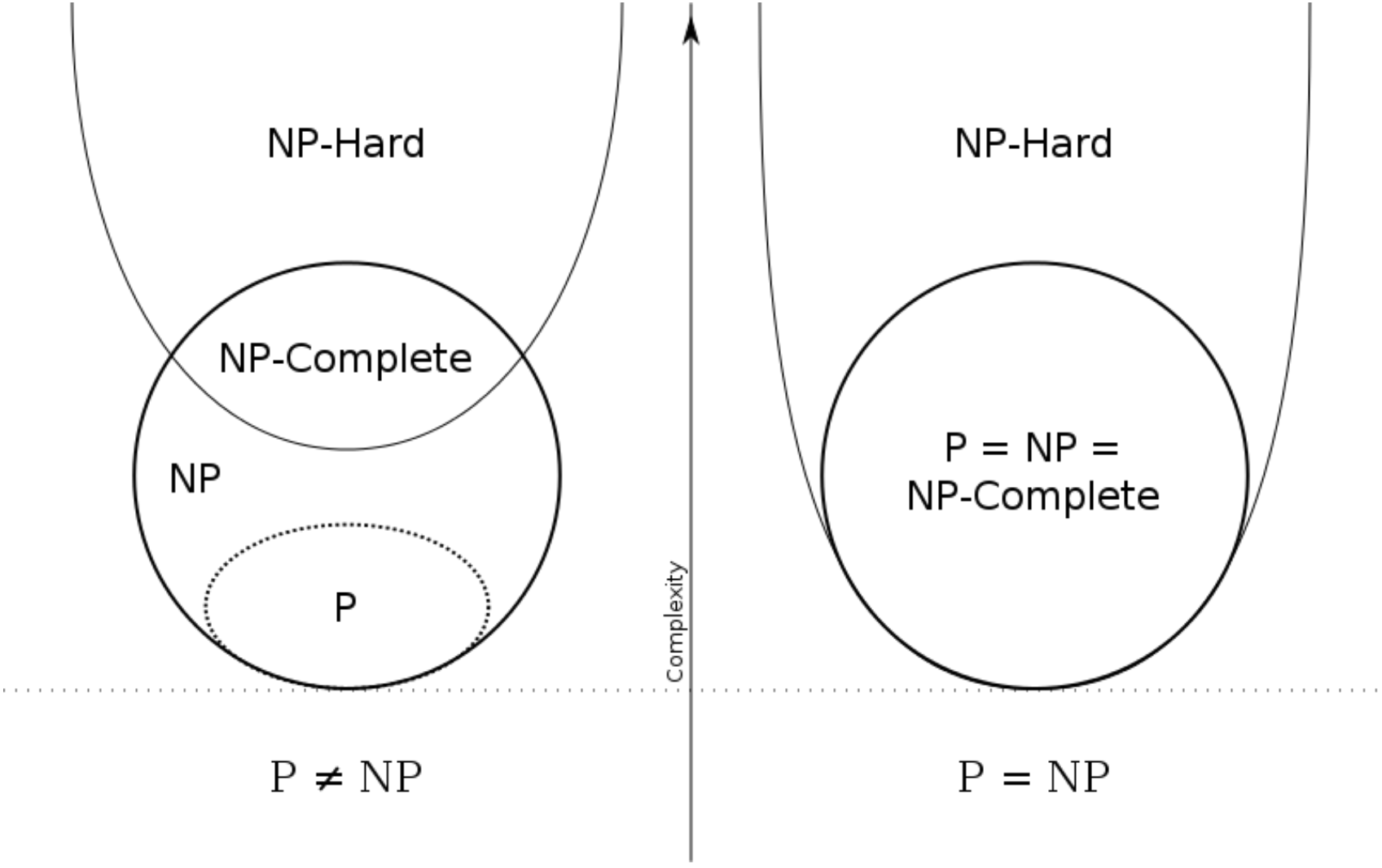
Probleme *NP-complete*: cele mai dificile probleme din clasa NP
dacă una din ele s-ar rezolva în timp polinomial,
orice altă problemă din NP s-ar rezolva în timp polinomial
$\Rightarrow$ am avea P = NP (se crede P $\neq$ NP)

Realizabilitatea (SAT) e prima problemă demonstrată a fi *NP-completă* (Cook, 1971). Sunt multe altele (21 probleme clasice: Karp 1972).

Cum demonstrăm că o problemă e NP-completă (grea) ?
reducem o problemă cunoscută din NP la problema studiată
$\Rightarrow$ dacă s-ar putea rezolva în timp polinomial problema nouă,
atunci ar lua timp polinomial problema cunoscută

P = NP?

Una din cele mai fundamentale probleme în informatică



Se crede că $P \neq NP$, dar nu s-a putut (încă) demonstra

Imagine: http://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg

Sintaxă și semantică

Pentru logica propozițională, am discutat:

Sintaxa: o formulă are *forma*:
propoziție sau $(\neg$ *formulă*$)$ sau $($*formulă* $\rightarrow$ *formulă*$)$

Semantica: calculăm *valoarea de adevăr* (înțelesul), pornind de la cea a propozițiilor

$$v(\neg\alpha) = \begin{cases} \mathsf{T} & \text{dacă } v(\alpha) = \mathsf{F} \\ \mathsf{F} & \text{dacă } v(\alpha) = \mathsf{T} \end{cases}$$

$$v(\alpha \rightarrow \beta) = \begin{cases} \mathsf{F} & \text{dacă } v(\alpha) = \mathsf{T} \text{ și } v(\beta) = \mathsf{F} \\ \mathsf{T} & \text{în caz contrar} \end{cases}$$

Deducții logice

Deducția ne permite să demonstrăm o formulă în mod *sintactic* (folosind doar structura ei)

E bazată pe o *regulă de inferență* (de deducție)

$$\frac{A \qquad A \to B}{B} \qquad \textit{modus ponens}$$

(din A și $A \to B$ deducem/inferăm B; A, B formule oarecare)

și un set de *axiome* (formule care pot fi folosite ca premise/ipoteze)
A1: $\alpha \to (\beta \to \alpha)$
A2: $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$
A3: $(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$
în care α, β etc. pot fi înlocuite cu *orice* formule

Exercițiu: arătați că A1 - A3 sunt tautologii

Deducție (demonstrație)

Informal, o deducție (demonstrație) e o înșiruire de afirmații în care fiecare rezultă (poate fi derivată) din cele anterioare.

Riguros, definim:
Fie H o mulțime de formule (ipoteze). O *deducție* (demonstrație) din H e un șir de formule $A_1, A_2, \cdots, A_n$, astfel ca $\forall i \in \overline{1, n}$
1. A_i este o *axiomă*, sau
2. A_i este o *ipoteză* o formulă din H), sau
3. A_i rezultă prin *modus ponens* din A_j, A_k anterioare $(j, k < i)$

Spunem că A_n *rezultă* din H (e *deductibil*, e o *consecință*).
Notăm: $H \vdash A_n$

Exemplu de deducție

Demonstrăm că $A \to A$ pentru orice formulă A

(1) $A \to ((A \to A) \to A))$ A1 cu $\alpha = A, \beta = A \to A$

(2) $A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$
A2 cu $\alpha = \gamma = A, \beta = A \to A$

(3) $(A \to (A \to A)) \to (A \to A)$ MP(1,2)

(4) $A \to (A \to A)$ A1 cu $\alpha = \beta = A$

(5) $A \to A$ MP(3,4)

Verificarea unei demonstrații e un proces simplu, mecanic
(verificăm motivul indicat pentru fiecare afirmație;
o simplă comparație de șiruri de simboluri).

Găsirea unei demonstrații e un proces mai dificil.

Alte reguli de deducție

Modus ponens e suficient pentru a formaliza logica propozițională dar sunt și alte reguli de deducție care simplifică demonstrațiile

$$\frac{p \to q \qquad \neg q}{\neg p}$$ *modus tollens (reducere la absurd)*

$$\frac{p}{p \vee q}$$ *generalizare (introducerea disjuncției)*

$$\frac{p \wedge q}{p}$$ *specializare (simplificare)*

$$\frac{p \vee q \qquad \neg p}{q}$$ *eliminare (silogism disjunctiv)*

$$\frac{p \to q \qquad q \to r}{p \to r}$$ *tranzitivitate (silogism ipotetic)*

Deducția (exemplu)

Fie $H = \{a,\ \neg b \vee d,\ a \to (b \wedge c),\ (c \wedge d) \to (\neg a \vee e)\}$.
Arătați că $H \vdash e$.

(1) a	ipoteză, H_1
(2) $a \to (b \wedge c)$	ipoteză, H_3
(3) $b \wedge c$	modus ponens (1, 2)
(4) b	specializare (3)
(5) d	eliminare (4, H_2)
(6) c	specializare (3)
(7) $c \wedge d$	(5) și (6)
(8) $\neg a \vee e$	modus ponens (7, H_4)
(9) e	eliminare (1, 8)

Consecința logică (semantică)

Interpretare = atribuire de adevăr pentru propozițiile unei formule.
O formulă poate fi adevărată sau falsă într-o interpretare.

Def.: O mulțime de formule $H = \{H_1, \ldots, H_n\}$ *implică* o formulă C
(C e o *consecință logică* / consecință semantică a ipotezelor H)
dacă orice interpretare care satisface (formulele din) H satisface C

Notăm: $H \models C$

Ca să stabilim consecința semantică trebuie să *interpretăm* formule
(cu valori/funcții de adevăr)

$\Rightarrow$ lucrăm cu *semantica* (înțelesul) formulelor

Exemplu: arătăm $\{A \vee B, C \vee \neg B\} \models A \vee C$ Fie interpretarea v.
Cazul 1: $v(B) = \mathrm{T}$. Atunci $v(A \vee B) = \mathrm{T}$ și $v(C \vee \neg B) = v(C)$.
Dacă $v(C) = T$, atunci $v(A \vee C) = T$, deci afirmația e adevărată.
Cazul 2: $v(B) = \mathrm{F}$. La fel, reducem la $\{A\} \models A \vee C$ (adevărat).

Consistență și completitudine

$H \vdash C$: *deducție* (pur sintactică, din axiome și reguli de inferență)
$H \models C$: *implicație, consecință semantică* (valori de adevăr)
Care e legătura între ele ?

Logica propozițională e *consistentă și completă*:

Consistență: Dacă H e o mulțime de formule, și C este o formulă astfel ca $H \vdash C$, atunci $H \models C$ (Orice teoremă e validă; orice afirmație obținută prin deducție e întotdeauna adevărată).

Completitudine: Dacă H e o mulțime de formule, și C e o formulă astfel ca $H \models C$, atunci $H \vdash C$. (Orice tautologie e o teoremă, orice consecință semantică poate fi dedusă din aceleași ipoteze).

Ca să demonstrăm o formulă, putem arăta că e *validă*.
Pentru aceasta, verificăm că *negația ei nu e realizabilă*.
(am văzut algoritmul DPLL; vom discuta metoda rezoluției).