Problema SAT

Context:

- $X = \{ x_1, x_2, \dots, x_n \}$ o mulțime de variabile boolene (pot lua valori True/False)
- literal = o variabilă sau negația unei variabile (exemple: x_1, $\neg x_2$)
- clauză = disjuncție de literali (este alcătuită din mai mulți literali cu simbolul „∨” ("SAU" logic) - între ei); exemplu: $(x_1 \vee \neg x_2 \vee x_3)$
- FNC = Forma Normală Conjunctivă
- formulă în FNC = o expresie de forma $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ unde C_i este o clauză (practic o conjuncție de clauze, aka clauze cu "ȘI" între ele)

Exemple de formule în FNC:

$$(p \vee q) \wedge (p \vee \neg q)$$

$$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_7) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_2 \vee x_5 \vee x_7)$$

Enunțul problemei:

Ni se dă o formulă în CNF (nu NEAPĂRAT în CNF, dar forma canonică a problemei SAT e în CNF, iar orice altă formă are la bază aducerea formulei în CNF - evident, orice formulă poate fi adusă în CNF). Există o configurație de valori pentru variabilele din X, astfel încât formula să fie evaluată la True? (aka Este formula satisfiabilă?)

Exemplu:

$X = \{ x_1, x_2, x_3 \}$

$F = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$

clauza1 $= (x_1 \vee x_2)$

clauza2 $= (\neg x_2 \vee x_3)$

Evident, ca F să fie adevărată, ambele clauze trebuie să fie adevărate (deoarece au "ŞI" între ele). Ne propunem să găsim ce valoare dăm fiecărei variabile astfel încât F să fie adevărată. Dacă toate sunt True, evident că F este adevărată. La fel şi dacă x_1 şi x_3 sunt True, iar x_2 False.

=> am găsit valori pentru care F este adevărată => F satisfiabilă

Opţional, să găsim numărul MINIM de variabile care trebuie să fie True ca F să fie True. Observăm că numărul minim este 1 (dacă x_2 este True, atunci neapărat şi x_3 trebuie să fie True; dacă x_2 este False, atunci neapărat x_1 trebuie să fie True, iar x_3 poate fi False). Anul trecut am avut temă la seminar la AA cu un algoritm aproximativ pentru problema asta (unde nu aveam si negaţie la variabile)!

Despre SAT (evident, vine de la SATisfiability):

Versiunea problemei în care fiecare clauză este limitată să aibă cel mult 3 literali, se numeste 3-SAT. Generalizând, există noţiunea de k-SAT. Există multe derivări/extinderi ale problemei SAT.

Problema se mai numeşte şi Problema Satisfiabilităţii Boolene şi este prima problemă despre care s-a demonstrat că este **NP-complete** (k-SAT este de asemeneaza NP-complete)! Deşi încă nu s-a demonstrat matematic, se crede că nu există un algoritm polinomial pentru a rezolva SAT, iar găsirea acestui algoritm/demonstrarea că nu există este echivalentă cu a răspunde la întrebarea **P vs NP** (despre care ştim că încă este *open problem*).

Există programe numite **SAT-solver** care încearcă să rezolve problema/cât mai multe instanţe ale problemei bazându-se pe euristici si noţiuni complexe. Evident că problema nu poate fi "rezolvată" (în timp polinomial), însă în prezent aceste programe sunt capabile să rezolve instanţe cu dimensiuni suficient de mari pentru a fi folosite in practică în domenii precum inteligenţa artificială, demonstrarea automată de teoreme.

Dacă tot suntem aici, un mic memory refresher despre **clase de complexitate** e mereu bine venit :)

În teoria complexităţii computaţionale, problemele sunt împărţite în clase pe

baza resurselor, cum ar fi timpul și spațiul, necesare pentru a le rezolva. Aceste clase ne ajută să înțelegem dificultatea problemelor și eficiența algoritmilor. Cele mai comune sunt:

- Clasa **P** (Polynomial):

  - probleme care pot fi rezolvate eficient în timp polinomial.

- Clasa **NP** (Nondeterministic Polynomial):

  - probleme care pot fi rezolvate de o mașină nedeterministă în timp polinomial.

  - probleme pentru care o soluție poate fi verificată eficient în timp polinomial.

  - nu este clar dacă aceste probleme pot fi rezolvate **și** în timp polinomial (P vs NP).

  - exemple: ciclu hamiltonian, SAT

- Clasa **NP-hard**:

  - probleme care sunt cel puțin la fel de dificile ca cele din clasa NP.

  - exemplu: travelling salesman

- Clasa **NP-complete**:

  - probleme care sunt în NP și sunt, într-un fel, cele mai dificile probleme din NP.

  - o problemă este NP-complete dacă este atât NP, cât și NP-hard

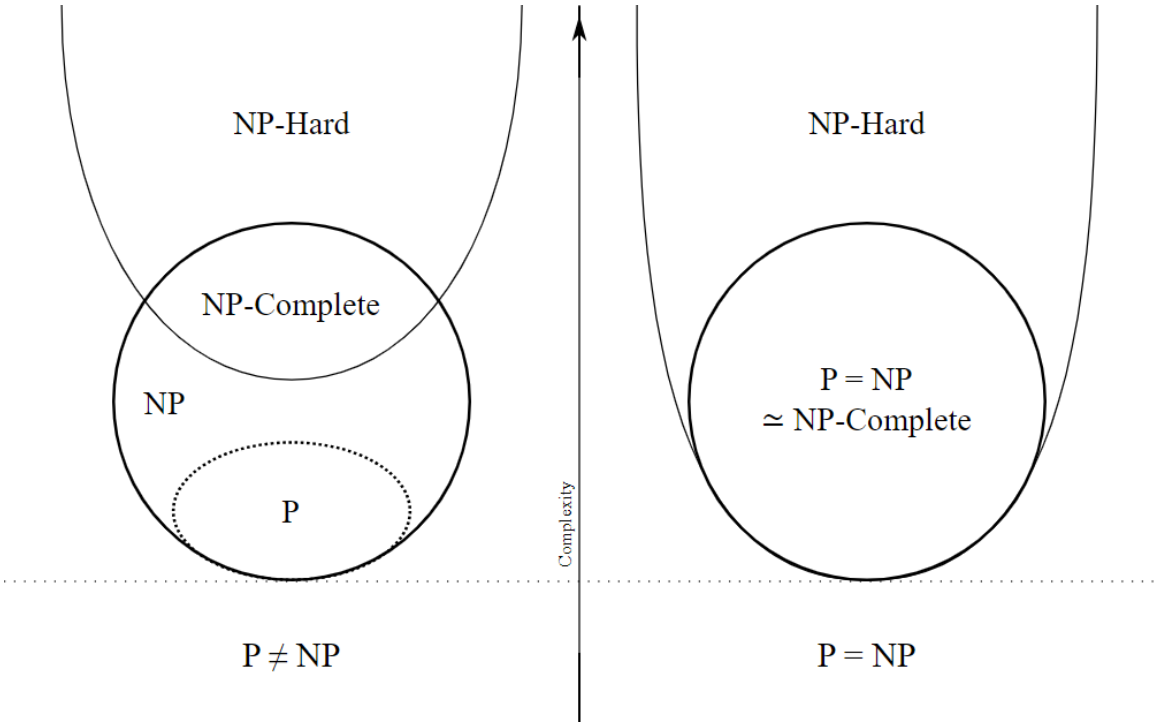
  - exemple: SAT, vertex cover

Dacă găsim o soluție eficientă pentru o problemă NP-completă, atunci putem rezolva eficient oricare altă problemă din NP, deoarece acestea sunt echivalente (pot fi reduse una la alta; NP-complete e inclus în NP).

P vs NP:

Este una din cele mai mari și importante probleme nerezolvate în informatica

teoretică (are bounty de un milion de dolari!), primind și titlul de problema mileniului (există 7 astfel de probleme). Informal, întrebarea este dacă orice problemă pentru care putem verifica rapid dacă o soluție dată este corectă, poate să fie de asemenea și rezolvată rapid. Se crede în general că NU (deși evident nu a fost demonstrat).



Mai sus observăm ierarhia claselor de complexitate în ambele cazuri (e important să ne uităm și să înțelegem).

Problema 3-colorării unui graf

Ni se dă un graf (m muchii și n noduri) și 3 culori (să zicem roșu, verde, albastru - RGB). Vrem să verificăm dacă nodurile grafului pot fi colorate cu cele 3 culori, fără ca 2 vecini să aibă aceeași culoare.

Problema poate fi redusă la SAT!

Mai exact, pentru ca graful să fie colorabil (cu restricțiile din enunț), trebuie să impunem următoarele condiții:

1. Un nod are o singură culoare

2. Nicio muchie nu leagă două noduri care au aceeaşi culoare

Pentru a traduce prima condiţie în limbaj de logică, vom lua pentru fiecare nod 3 variabile boolene astfel: nod_rosu, nod_verde, nod_albastru, cu semnificaţia că dacă *nod_rosu=True*, atunci nodul are culoarea roşie. Vom impune condiţia ca, pentru fiecare nod, doar una dintre cele 3 să fie adevărată:

- verificăm ca fiecare nod să aibă asignată cel puţin o culoare

$$\bigwedge_{i=1}^{n} nod[i].rosu \ \vee \ nod[i].verde \ \vee \ nod[i].albastru$$

- luăm culorile 2 câte 2 şi vedem că nu sunt adevărate ambele simultan

$$\bigwedge_{i=1}^{n} \begin{array}{l} \neg(nod[i].rosu \ \wedge \ nod[i].verde) \\ \wedge \ \neg(nod[i].rosu \ \wedge \ nod[i].albastru) \\ \wedge \ \neg(nod[i].albastru \ \wedge \ nod[i].verde) \end{array}$$

Mai departe, pentru a îndeplini condiţia conform căreia *nicio muchie nu leagă 2 noduri care au aceeaşi culoare*:

$$\bigwedge_{i=1}^{m} \begin{array}{l} \neg\Big(muchie[i].nod1\_rosu \ \wedge \ muchie[i].nod2\_rosu\Big) \\ \wedge \ \neg\Big(muchie[i].nod1\_verde \ \wedge \ muchie[i].nod2\_verde\Big) \\ \wedge \ \neg\Big(muchie[i].nod1\_albastru \ \wedge \ muchie[i].nod2\_albastru\Big) \end{array}$$

Pentru fiecare muchie, luăm nodurile pe care le leagă şi impunem pentru fiecare culoare să nu fie True la ambele noduri.

Făcând AND între toate acestea (şi eventual aducând rezultatul la FNC - este simplu, dar am lăsat aşa pentru că e mai uşor de vizualizat), am găsit formula SAT care corespunde 3-colorării grafului.

Bonus fact: ca să arătăm că o problemă este NP-complete, trebuie să arătăm că:

1. Este NP, adică dată fiind o soluție, putem verifica în timp polinomial dacă este corectă.
2. O problemă NP-complete se poate reduce la problema noastră (nu invers!); de exemplu, dacă știm că SAT este NP-complete și vrem să arătăm că 3-coloring este NP-complete, vom reduce SAT la 3-coloring.
3. Algoritmul de reducere de la 2) rulează în timp polinomial