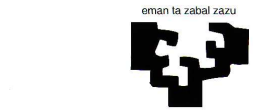


Lengoaiak, Konputazioa eta Sistema Adimendunak

Kudeaketaren eta Informazio Sistemen Informatikaren Ingeniaritzako Gradua

Bilboko Ingeniaritza Eskola (UPV/EHU)

2. maila

2019-2020 ikasturtea

5. gaia: Konputazioaren konplexutasuna

JOSÉ GAINTZARAIN IBARMIA

Lengoaia eta Sistema Informatikoak Saila

Azken eguneraketa: 2019 - 08 - 30

Aurkibidea

5. gaia

Konputazioaren konplexutasuna

5.1.

Sarrera

Problema batzuk ez dira konputagarriak, baina konputagarriak diren problemen kasuan ere kalkuluen konplexutasunaren arazoa agertzen da. Kalkuluen konplexutasuna kalkuluak burutzeko behar den denborarekin edo espazioarekin (memoriarekin) lotuta egoten da. Berez konplexutasuna algoritmoei lotuta dago eta ez problemei. Problema bat ebazteko dugun algoritmoa konplexua izan arren, behar bada askoz hobea eta errazagoa den beste algoritmo bat existi daiteke. Problema batzuen kasuan, problema horiek ebazteko balio duten algoritmo ezagun denak konplexuak dira (denbora asko edo memoria asko behar dute) baina ez dakigu algoritmo hoberik existitzen al den ala ez. Problema batzuen kasuan, beraien soluzioa aurkitzeko balio duten algoritmo ezagun denak denbora asko edo espazio asko behar dutenez, soluzioa kalkulatzen hasteak ez du zentzurik (denbora gehiegi edo espazio gehiegi beharko delako). Denbora edo espazio gehiegi behar duten problemei "landuezinak" edo "trataezinak" deitzen zaie. Informatikaren arlo bat problemak beraien zailtasuna edo konplexutasunaren arabera sailkatzeaz arduratzen da. Sailkapen hori egiteko problema horiek ebazteko balio duten algoritmo ezagunak hartzen dira kontuan.

Gai honetan sailkapen hori era sinplifikatuan aurkeztuko da.

5.2.

Denbora-konplexutasuna eta espazio-konplexutasuna

5.2.1 Denbora-konplexutasuna

Denbora-konplexutasunaren barruan hiru kasu bereiztuko ditugu: algoritmo linealak, algoritmo polinomikoak eta algoritmo esponentzialak.

5.2.1.1 Algoritmo linealak eta polinomikoak

Har dezagun A(1..j) bektoreko elementuen batura s aldagaian kalkulatzen duen algoritmoa. Batura hori kalkulatzeko nahikoa da i aldagaia indize bezala erabiliz bektorea behin zeharkatzearekin. Demagun bektoreko elementu bat irakurtzeko behar den denbora c_1 dela, i-ren eguneraketa bakoitza egiteko behar den denbora c_2 dela eta s aldagaiaren eguneraketa bakoitza egiteko behar den denbora c_3 dela. Konstante horien balioa oso txikia izan ohi da eta horregatik c_1, c_2 eta c_3 konstanteek ez diote algoritmoaren konplexutasunari eragiten. Konplexutasunean benetako eragina dutenak honako hauek dira: bektorearen posizio-kopurua eta bektorea zenbat aldiz zeharkatzen den. Adibide honetan batura kalkulatzeko nahikoa da bektorea behin zeharkatzearekin eta ondorioz algoritmoaren konplexutasuna lineala da, izan ere konstante bat ($c_1 + c_2 + c_3$) bider posizio-kopurua (j) baita: $(c_1 + c_2 + c_3) \times j$

A(1..j) bektoreko balio txikiena t aldagaian lagatzen duen algoritmoaren konplexutasuna ere lineala da. Balio txikiena zein den kalkulatzeko t aldagaia $A(1)$ balioarekin hasieratu beharko da eta gero i aldagaia indize bezala erabiliz bektorea zeharkatu beharko da. Demagun bektoreko elementu bat irakurtzeko behar den denbora c_1 dela, i-ren eguneraketa bakoitza egiteko behar den denbora c_2 dela eta t aldagaia $A(i)$ balioarekin konparatzeko eta t eguneratzeko (eguneratu behar baldin bada) behar den denbora c_3 dela. Konstante horien balioa oso txikia izan ohi da eta horregatik c_1, c_2 eta c_3 konstanteek ez diote algoritmoaren konplexutasunari eragiten. Bektorearen posizio-kopurua eta bektorea zenbat aldiz zeharkatzen den hartzen dira kontuan konplexutasuna kalkulatzean, bi balio horiek baitira konplexutasunean benetako eragina dutenak. Adibide honetan balio txikiena kalkulatzeko nahikoa da bektorea behin zeharkatzearekin

eta ondorioz algoritmoaren konplexutasuna lineala da, izan ere konstante bat ($c_1+c_2+c_3$) bider posizio-kopurua (j) baita: $(c_1 + c_2 + c_3) \times j$

$A(1..j)$ bektorea ordenatzeko $A(1..j)$ bektoreko balio txikiena aurkitu eta 1 posizioan ipini (1 posiziokoa $A(1..j)$ bektoreko txikiena zegoen lekura eramanez), gero $A(2..j)$ tarteko balio txikiena aurkitu eta 2 posizioan kokatu (2 posiziokoa $A(2..j)$ bektoreko txikiena zegoen lekura eramanez). Kontuan izan dagoeneko 1 posizioan bektore osoko balio txikiena egongo dela. Gero $A(3..j)$ tarteko balio txikiena aurkitu eta 3 posizioan kokatu (3 posiziokoa $A(3..j)$ bektoreko txikiena zegoen lekura eramanez). Berriro ere kontuan izan dagoeneko 1 eta 2 posizioetan bi balio txikienak egongo direla. Beste posizioekin ere horrela jarraitzen duen algoritmoa hartuko dugu orain. Lehenengo aldian j posizio zeharkatzen dira, bigarrengo aldian $j-1$ posizio zeharkatzen dira, hirugarrengoan $j-2$ eta azkeneko aldian posizio 1 zeharkatzen da. Guztira $j + (j-1) + (j-2) + \cdots + 1$ posizio zeharkatzen dira, hau da, $(j^2+j)/2$ posizio. Beraz algoritmo hau balio txikiena bilatzen duen algoritmoa baino konplexuagoa da. Une bakoitzean h aldagaiak ordenatu gabeko zatia non hasten den adieraziko du eta i aldagaia h posiziotik j posiziorainoko tartea zeharkatzeko erabiliko da. Demagun bektoreko elementu bat irakurtzeko behar den denbora c_1 dela, i-ren eguneraketa bakoitza egiteko behar den denbora c_2 dela eta $A(h)$ osagaia $A(i)$ balioarekin konparatzeko eta $A(h)$ eguneratzeko (eguneratu behar baldin bada) behar den denbora c_3 dela. Algoritmoaren konplexutasuna polinomikoa da, 2. mailako polinomio bat ateratzen baita: $(c_1 + c_2 + c_3) \times ((j^2+j)/2)$.

Denbora neurtzerakoan batzutan kasu okerrena eta batezbesteko kasua bereiztea komeni izaten da. Adibidez x elementua $A(1..j)$ bektorean agertzen al den erabakitzerakoan, behar bada x balioa 3 posizioan aurkituko dugu eta ez dugu bektoreko j posizioak zeharkatu beharrik izango. Bestalde, x balioa $A(1..j)$ bektorean ez bada agertzen, hori horrela dela jakiteko bektore osoa zeharkatu beharko da. Indize bezala i aldagaia erabiliko da. Demagun bektoreko elementu bat irakurtzeko behar den denbora c_1 dela, i-ren eguneraketa bakoitza egiteko behar den denbora c_2 dela eta x osagaia $A(i)$ balioarekin alderatzeko behar den denbora c_3 dela. Kasu okerrenean $(c_1 + c_2 + c_3) \times j$ denbora unitate beharko genituzke baina batezbesteko denbora $((c_1+c_2+c_3) \times j)/2$ izango litzateke. x elementua $A(1..j)$ bektore ordenatuan agertzen al den erabakitzerakoan ere antzekoa gertatuko litzateke. Elementu bat dagoeneko ordenatuta dauden elementuen artean bilatzea nahi denean, datuak gordetzeko zuhaitz bitarrak erabiltzen baditugu bektoreen ordez, algoritmo azkarragoak lortuko ditugu.

Algoritmo linealak oso onak dira eta algoritmo polinomikoak nahiko onak dira orokorrean, baina problema batzuentzat ez da ezagutzen ez algoritmo linealik eta ez algoritmo polinomikorik ere.

5.2.1.2 Algoritmo esponentzialak

Formula boolear bat True egiten duen baloraziorik ba al dagoen erabakitzearen problema hartuko dugu orain. Problema hau SAT bezala ezagutzen da (ingelesezko "satisfiable" hitzean oinarrituz). Formulak eraikitzeko aldagai boolearrak erabiliko dira (q, r, s, eta abar) eta eragile be-

zala ukapena, konjuntzioa eta disjuntzioa ($\neg$, $\wedge$, $\vee$) erabiliko ditugu. Beraz $\varphi = (q \vee r) \wedge (q \vee \neg s)$ formula bat da.

Adibidez $q = True$, $r = False$ eta $s = True$ direnean φ formula $True$ da. Bestalde $\psi = (\neg q \vee r) \wedge (q \vee s) \wedge \neg r \wedge \neg s$ formula hartzen badugu, ψ True egiten duen balorazioriK ez da existitzen. ψ True egiten duen balorazioriK ez dela existitzen frogatzeko konbinazio posible denak eratu beharko genituzke, 2^3 konbinazio guztira, hau da, bi ber ψ formulan agertzen diren aldagai desberdinen kopurua (q, r, s).

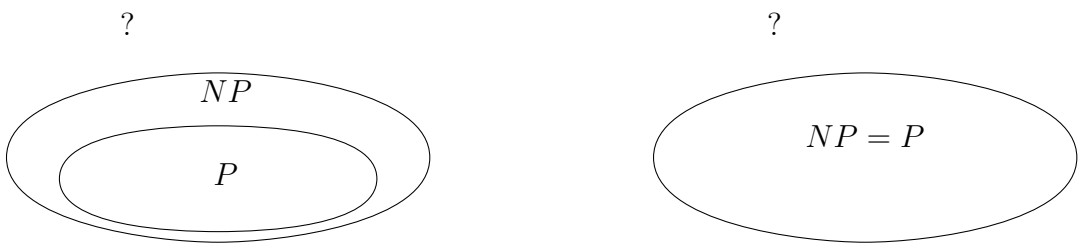
q	r	s	φ	ψ
$True$	$True$	$True$	$True$	$False$
$True$	$True$	$False$	$True$	$False$
$True$	$False$	$True$	$True$	$False$
$True$	$False$	$False$	$True$	$False$
$False$	$True$	$True$	$False$	$False$
$False$	$True$	$False$	$True$	$False$
$False$	$False$	$True$	$False$	$False$
$False$	$False$	$False$	$False$	$False$

Formula bat emanda, formula hori True egiten duen balorazioriK existitzen al den kalkulatzearen konplexutasuna esponentziala da, izan ere j aldagai desberdin dituen formula bat emanda, 2^j konbinazio sortu eta egiaztatu behar baitira orokorrean. Metodo hau oso txarra da. Esate baterako 100 aldagai dituen formula batentzat 2^{100} konbinazio daude, hau da, gutxi gorabehera 10^{30} konbinazio dira. Konbinazio bakoitza aztertzeko segundu bat beharko bagenu, guztira 10^{22} urte beharko genituzke. Konbinazio bakoitza aztertzeko behar den denbora $1/10^{10}$ segundukoa dela kontsideratuta ere, 10^{12} urte beharko genituzke. Beraz problema hau landuezina edo trataezina da.

Formula bat emanda, formula True egiten duen balorazioren bat existitzen al den erabakitzen duen algoritmo polinimikorik ez da aurkitu orain arte, baina algoritmo polinimikorik ez dela existitzen ere ez da frogatu.

5.2.1.3 P eta NP

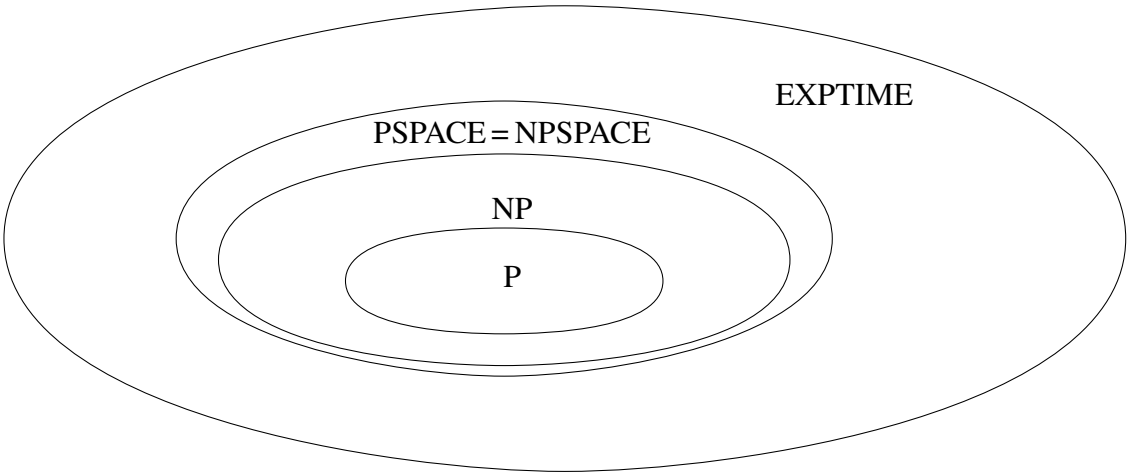
Algoritmoak polinimoko eta esponentzial bezala sailkatzeak P eta NP multzoak definitzea ekarri du berarekin (Polynomial, Non-Polynomial). Problema batentzat algoritmo polinimikoren bat ezagutzen bada orduan problema hori P multzoan dago baina SAT bezalako problemak NP multzoan daude. Oraindik inork ez daki P eta NP berdinak al diren ala ez, baina desberdinak izango direla uste da.



5.2.2 Espazio-konplexutasuna

Algoritmo batek behar duen denborak konputazio-konplexutasuna eragiten duen era berean, algoritmoak behar duen espazioak (memoriak) ere konplexutasuna eragiten du. Problema batzuk denbora esponentziala behar izateagatik landuezintzat edo trataezintzat jotzen diren bezala, beste problema batzuk landuezinak edo trataezinak dira espazio (memoria) asko behar dutelako. Adibidez, zenbatutako edo kuantifikatutako formula boolear baten balioa True al den aztertzeak espazio polinomikoa behar du. Zenbatutako edo kuantifikatutako formulen adibide bat $\forall x(Q(x) \vee \exists y(R(x,y) \wedge S(y))$ da. PSPACE eta NPSPACE berdinak direla frogatuta dago.

Konplexutasun klaseen arteko erlazioa honako hau dela uste da:



NP eta EXPTIME multzoetako problementzat ezagutzen diren algoritmoak denbora esponentziala behar dute baina NP-koak diren problemen kasuan, erantzuna aurkitzeko denbora esponentziala behar da baina erantzuna ematen badigute, erantzuna zuzena al den ala ez erabakitzea erraza da (denbora gutxi behar da). Baina EXPTIME multzoko problemen kasuan erantzuna aurkitzeko denbora asko behar da eta erantzuna emanda ere erantzuna benetan zuzena dela egiaztatzeko ere denbora esponentziala behar da.

5.3.

Konputazio-konplexutasunaren teoriaren aplikazioak

Konputazio-konplexutasunaren aplikazio garrantzitsuenetako bat kriptografia da. Problema batzuk ebazteko denbora asko behar dela badakigunez, gakoak eratzeko orduan problema horietan oinarritzen da kriptografia, izan ere horrela bai baitakigu gakoa zein den aurkitzeko denbora asko beharko dela. Esate baterako osoak diren zenbaki handiak faktorizatzeko (zenbaki lehenetan deskonposatzeko) denbora asko behar da eta horregatik kriptografoak batzutan faktoriazioaren probleman oinarritu izan dira aurkitzen zailak diren gakoak eratzeko.

5.4.

Beste konputazio-eredu batzuk

Gai honetan azaldu den bezala, problema batzuk ebazteko ezagunak diren algoritmoek hainbeste denbora edo hainbeste espazio behar dutenez, ez dauka zentzurik algoritmo horiek erabiltzeak. Arazo edo muga horren aurrean beste konputazio-eredu bat probabilitatean (estatistikan) oinarritzen den eredua da. Eredu honetan emaitza zehatza kalkulatu beharrean emaitza hurbildu egiten da. Emaitza ona edo nahiko ona izateko probabilitatea handia denean gelditu egiten dira algoritmo probabilistikoak. Horrela konputazio-denbora askoz txikiagoa da eta emaitza nahiko ona izateko probabilitatea handia da.

Adibidez, zenbaki handiak lehenak al diren erabakitzeko denbora asko behar da baina bada zenbaki lehen denek eta zenbaki ez-lehen gutxi batzuk betetzen duten propietate bat. Eredu probabilistikoan zenbaki bat lehena al den ala ez erabakitzeko kalkulu zehatzak egin beharrean, egiaztatzen askoz errazagoa den beste propietate hori aztertzen da. Emandako zenbakiak beste propietate hori betetzen badu, zenbakia lehena dela esango du algoritmoak. Metododo honekin askotan asmatu egiten da baina hala ere kasu batzuetan ez da asmatzen, propietate hori betetzen duten zenbaki ez-lehen batzuk existitzen direlako. Hala ere asmatzeko probabilitatea handia da.