P=NP?

Vedran Šego[1]

1. Uvod

The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) odabrao je 7 važnih, a dugo vremena neriješenih problema, ponudivši nagradu od po milijun dolara za rješenje svakog od njih. Ti problemi zovu se *Milenijski problemi*, a više o njima možete pronaći u [5].

U ovom tekstu bavit ćemo se jednim od tih problema, tzv. "P=NP?" ili "P vs. NP". Riječ je o problemu iz teorije algoritama, a koji – laički rečeno – znači "postoje li, za naizgled složene probleme, jednostavni algoritmi koji ih rješavaju?".

Na web stranici CMI-a problem "P=NP?" prikazan je sljedećim slikovitim primjerom:

> Pretpostavimo da želimo organizirati smještaj za grupu od 400 studenata. Prostor je ograničen, te će samo 100 studenata dobiti sobu. Da ne bi bilo prejednostavno, dekan je dao listu parova "nekompatibilnih" studenata – onih koji ne mogu zajedno biti u domu (tj. ako jedan dobije smještaj, drugi ga ne smije dobiti).

Ovo je tipični NP-potpuni problem, jer je za već gotovo rješenje jednostavno provjeriti zadovoljava li dane uvjete, ali do samog rješenja općenito nije jednostavno doći.

Naravno, ako je zadana lista parova "lijepa" (npr. skoro prazna ili sadrži skoro sve parove studenata), vodit će nas gotovo deterministički prema nekom rješenju. No, općenito, može se dogoditi da moramo pokušati sve kombinacije.

Matematičkim rječnikom, to znači da treba pogledati sve 100-člane podskupove skupa studenata, te provjeriti koji od njih (ako ijedan!) zadovoljava uvjet "nekompatibilnosti". Iz kombinatorike, poznato je da m-članih podskupova n-članog skupa ima

$$\binom{n}{m} = \frac{n!}{(n-m)!\,m!}.$$

U našem slučaju, traženih podskupova ima

$$\begin{aligned}\binom{400}{100} &= \frac{400!}{(400-100)!100!} \\ &= 300742969389401893510717432 0 \approx 3 \cdot 10^{27}.\end{aligned}$$

Za usporedbu, danas najbrže računalo[2] uspjelo je postići 2331 TFLOPS[3], odnosno oko $2.33 \cdot 10^{15}$ operacija s pomičnim zarezom u sekundi. Provjera uvjeta za svako od $3 \cdot 10^{27}$ mogućih rješenja nužno zahtijeva puno više od jedne operacije s pomičnim zarezom, no

[1] Autor je viši asistent na Matematičkom odjelu PMF-a Sveučilišta u Zagrebu, e-pošta: vsego@math.hr. Članak je objavljen šk. god. 2009/10.

[2] Cray XT5 s 224162 Opterona

[3] FLOPS znači *floating point operations per second*, tj. broj operacija s pomičnim zarezom (osnovne računske operacije na realnim brojevima) koje računalo može obaviti u jednoj sekundi. TFLOPS (čita se "tera flops") je 10^{12} FLOPS.

čak i kad bi nam jedna operacija bila dovoljna, navedenom super-računalu trebalo bi više od 10^{12} sekundi, odnosno više od 30 000 godina, da riješi zadani problem.

Opisani način rješavanja problema naziva se *brute force* (rješavanje grubom silom) i često je neizvedivo zbog ogromnog vremena izvršavanja.

Godine 1971., neovisno jedan o drugome, Cook i Leonid definirali su P-probleme kao "one koje je jednostavno riješiti" i NP-probleme kao "one čije je rješenje jednostavno provjeriti". Što u ovom kontekstu znači "jednostavno", vidjet ćemo u poglavlju 4.

Do danas nije poznato je li riječ o istoj klasi problema ili postoji problem čije je rješenje jednostavno provjeriti, ali ga nije jednostavno naći. To pitanje zovemo "P=NP?" problem.

2. Algoritmi

Da bismo mogli razmatrati algoritme koji rješavaju pojedine probleme, potrebno je prvo reći što je to uopće algoritam. Najkraće rečeno, algoritam je popis jednoznačnih uputa koje treba slijediti da bismo u konačnom vremenu riješili problem iz neke zadane klase problema (ili, općenitije, obavili neki posao). Ovdje je bitno naglasiti:

1. konačnost niza uputa, da bismo algoritam uopće mogli zapisati;
2. do rješenja treba doći u konačnom vremenu, tj. algoritam se ne smije vječno izvršavati;
3. algoritam rješava probleme iz određene klase problema, dakle, čak i ako ga možemo "zaposliti" s nekim problemom za koji on nije predviđen, ne možemo očekivati da će ga korektno riješiti.

Pogledajmo primjer iz matematike: zadana je jednadžba

$$ax^2 + bx + c = 0,$$

za neke zadane $a, b, c \in \mathbf{R}$. Traži se kompleksni broj $x \in \mathbf{C}$ koji zadovoljava zadanu jednadžbu. Jednostavno, ali i pogrešno rješenje ovog problema bi bilo:

> Jednadžba $ax^2+bx+c = 0$, pri čemu su $a, b, c \in \mathbf{R}$, u skupu kompleksnih brojeva ima dva rješenja, dana formulom
>
> $$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Prikazana formula je poznata za rješavanje kvadratnih jednadžbi, no u njoj se kriju zamke koje – kad računamo "na prste" – obično zaobiđemo "u hodu". No i ta zaobilaženja treba ugraditi u algoritam:

1. Ako je $a = 0$, jednadžba nije kvadratna, a primjena gornje formule rezultirala bi dijeljenjem s nulom. Tada imamo slučajeve:
   (a) Ako je $b = c = 0$, svaki kompleksni broj $x \in \mathbf{C}$ je rješenje zadane jednadžbe.
   (b) Ako je $b = 0$ i $c \neq 0$, zadana jednadžba nema rješenja.

(c) Ako je $b \neq 0$, zadana jednadžba ima jedinstveno rješenje

$$x = -\frac{c}{b}.$$

2. Ako je $a \neq 0$ i $b^2 - 4ac = 0$, zadana jednadžba je kvadratna, ali ima jedinstveno rješenje

$$x = -\frac{b}{2a}.$$

3. Ako je $a \neq 0$ i $b^2 - 4ac \neq 0$, zadana jednadžba je kvadratna i ima dva rješenja, dana formulom

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Tek sada imamo jednoznačnu i točnu proceduru rješavanja zadane jednadžbe koja će uvijek u konačno mnogo koraka dati ispravno rješenje zadanog problema.

3. Koliko je neki algoritam brz?

Kad se postavi pitanje brzine algoritma, prvo što nekome padne na pamet obično je "štopericu u ruke i pokrenimo program". Ovakav pristup iz mnogo razloga nije dobar.

Kao prvo, program treba napisati u nekom jeziku. Bitno je naglasiti da implementacija algoritma (u određenom programskom jeziku, na određenoj arhitekturi računala i sl.) nije isto što i algoritam. Jezici se međusobno razlikuju i po mogućnostima (npr. FORTRAN podržava kompleksne brojeve, dok C i Pascal ne) i po brzini (programi pisani u C-u i Pascalu su općenito daleko brži od onih pisanih u C++ i Delphiju). Dapače, mnogi jezici imaju više od jednog prevodioca koji se međusobno opet razlikuju (C je posebno poznat po ovome, no slično vrijedi i za Pacal, C++, Perl, SmallTalk i mnoge druge). Zbog svega navedenog, postavlja se pitanje kako usporediti dva algoritma ako imamo njihove programske implementacije napisane u različitim jezicima?

Kao drugo, računala se međusobno razlikuju. Neka imaju brže procesore, memorije, sabirnice; neka imaju više radne memorije ili međumemorije; različita može biti i programska podrška (npr. operativni sustav); itd. Kako usporediti rezultate dobivene "štopericom" za program koji je testiran na Athlonu na 2.7 GHz s 1 GB RAM-a pod nekim Linuxom i program testiran na 64-bitnom Pentiumu na 3 GHz i s 2 GB RAM-a pod Mac OS X-om?

Računala se razvijaju velikom brzinom i novi modeli dolaze na tržište svakodnevno. Kako bi se izbjegla potreba stalnog biranja idealne platforme za testiranja algoritama (što bi opet onemogućilo usporedbe starih i novih testova), analiza algoritama bazira se na teorijskom računalu koje je 1937. godine u [9] predstavio Alan Turing. To računalo zovemo Turingov stroj, a njegovu formalnu definiciju možete pronaći, na primjer, u [6]. Ovdje ćemo dati neformalni opis.

Turingov stroj zamišljamo kao računalo koje ima beskonačnu traku. Po toj traci čita i piše glava stroja koja se u svakom koraku (izvršavanju jedne naredbe) može pomaknuti jedno mjesto lijevo ili desno ili može ostati na mjestu. Stroj prati i stanje u kojem se nalazi, svojevrstan pandan vrijednostima varijabli u programima.

U svakom koraku, Turingov stroj nalazi se u nekom stanju q, te čita s trake (na mjestu na kojem se nalazi glava stroja) znak x. Ovisno o stanju i pročitanom znaku, stroj prelazi u stanje q', na traku zapisuje znak x', te glava vrši pomak p (za jedno

mjesto lijevo ili desno ili ostaje na mjestu). Matematički zapisano, jedan korak je zapravo obično pridruživanje:

$$(q,x) \mapsto (q',x',p).$$

Kad popišemo sve takve korake, definirali smo Turingov stroj.

Primijetimo: ako su svi parovi (q,x) međusobno različiti, opisani Turingov stroj je zapravo matematička funkcija. To znači da ponašanje stroja ovisi isključivo o njegovom stanju i znaku pročitanom na traci. Takve strojeve zovemo deterministički Turingovi strojevi.

Ako se neki od parova (q,x) preslikavaju u više različitih trojki (q',x',p), opisani skup pravila nije funkcija, nego samo relacija. Za takav stroj kažemo da je nedeterministički. Iako je to naizgled u koliziji sa zahtjevom da koraci algoritma moraju jednoznačno opisivati što algoritam treba raditi, kod nedeterminističkih Turingovih strojeva pretpostavljamo da će stroj pogoditi ispravni put k rješenju.

Recimo da se stroj nalazi u stanju q, te je s trake pročitao znak x i pravila dopuštaju korake

$$(q,x) \mapsto (q'_i,x'_i,p_i), \quad i = 1,2,\ldots,n,$$

za neki $n > 1$. Tada pretpostavljamo da će stroj odabrati upravo takav i da ga prelazak u stanje q'_i uz zapisivanje znaka x'_i i pomak glave p_i vode prema rješenju.

Zanimljivo je da su nedeterministički Turingovi strojevi ekvivalentni deterističkima (tj. mogu riješiti iste probleme): dovoljno je napraviti deterministički stroj koji će nekim redom isprobati sve mogućnosti kroz koje bi mogao proći njegov nedeterministički ekvivalent. Naravno, stroj koji isprobava sve mogućnosti obavlja posao puno sporije od onoga koji odmah pogađa ispravni put. Upravo to će biti bitna razlika između P- i NP-problema, što ćemo vidjeti u poglavlju 5.

P- i NP-problemi obično se definiraju kao problemi odlučivanja (oni koji daju odgovor na da/ne-pitanje), no mi ćemo ih ovdje promatrati u širem kontekstu traženja odgovora i na općenitija pitanja.

4. P-problemi

Kao P-probleme (ili probleme klase P) definiramo one probleme za koje je moguće definirati deterministički Turingov stroj koji rješenje nalazi u polinomnom vremenu, što znači da postoji neki polinom $p(X)$ takav da je broj koraka Turingovog stroja uvijek manji od $p(X)$, gdje je X duljina ulaznih podataka zapisanih na traci Turingovog stroja.

Turingov stroj nam služi za opis osnovnih operacija. Jedan korak ovdje jasno označava: jedno čitanje s trake, jedno pisanje na traku, jedan pomak glave stroja i jednu promjenu stanja. Kod analize algoritama često brojimo osnovne računske operacije, što se jednostavno prevede u terminologiju determinističkih Turingovih strojeva.

Primjer 4.1. *Zadan je niz brojeva koje treba poredati (sortirati) od najmanjeg prema najvećem.*

Očito, općenito je nebitno koje su vrijednosti brojeva koje želimo poredati, ali je itekako bitno koliko ih ima. Označimo duljinu niza s n.

Algoritam "Isprobaj sve mogućnosti". *Jedan način sortiranja brojeva je pronaći sve njihove moguće redoslijede i za svaki provjeriti je li dobar. Sama provjera je jednostavna: potrebno je jednom proći kroz cijeli niz i provjeriti jesu li svaka dva susjedna broja u ispravnom poretku. To se, očito, izvede u $n-1$ koraka (preciznije, usporedbi brojeva), što je polinom prvog stupnja, pa se provjera izvodi u polinomnom*

vremenu. Dapače, pošto je polinom prvog stupnja (dakle, linearan), kažemo da je vrijeme izvršavanja linearno.

No, svih mogućnosti ima

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1.$$

Lako se provjeri da za proizvoljni polinom $p(x)$ postoji dovoljno veliki n takav da je $n! > p(n)$, što znači da takav algoritam ima nadpolinomno vrijeme izvršavanja.

No, činjenica da smo pronašli spori algoritam koji rješava naš problem, naravno, ne znači da je sam problem težak (u smislu da nema polinomno rješenje), nego samo da je pristup "isprobaj sve mogućnosti" spor.

Algoritam "Usporedi svaka dva". *Možemo usporediti svaka dva broja, te ako su oni u pogrešnom redoslijedu, onda ih zamijeniti. Riječ je o tzv. selection sortu koji je jednostavan za implementaciju i analizu složenosti, no sporiji od ostalih koji su u upotrebi (v. [7]).*

Primijetimo da brojeva ima n, a svakog možemo usporediti (i eventualno zamijeniti) s njih najviše $n-1$ (jer nema smisla uspoređivati broj sa samim sobom), pa će broj koraka očito biti manji ili jednak vrijednosti polinoma

$$p(n) = n(n-1) = n^2 - n.$$

Zbog jednostavnosti, prikažimo uzlazni sort (dakle, onaj koji želi elemente niza poredati od najmanjeg do najvećeg):

```
za sve x[i]
 min = i
 za sve x[j] takve da je i razlicito od j
  ako je x[k] > x[j]
   min = j
 ako je min razlicito od i
  zamijeni vrijednosti x[i] i x[min]
```

Očito je da ovakav algoritam zaista sortira niz. Naime, on pronalazi index `min` *takav da je $x_{\texttt{min}}$ najmanji element podniza $x_i, x_{i+1}, \ldots, x_n$, te ga smješta na mjesto x_i (dakle, ispred svih elemenata koji su veći od njega). Tako će u prvom prolazu vanjske petlje najmanji element biti postavljen na prvo mjesto u nizu. U drugom prolazu, drugi najmanji će biti postavljen na drugo mjesto, itd.*

U stvarnosti, usporedbi i potencijalnih zamjena ima $n(n-1)/2$, no taj detalj utječe samo na koeficijente polinoma, ali ne mijenja činjenicu da je riječ o polinomu drugog stupnja. Zamjena brojeva `x1` *i* `x2` *vrši se u tri koraka:*

```
privremeni = x1;
x1 = x2;
x2 = privremeni;
```

Dakle, broj koraka u zamjeni dva broja ne ovisi o n, pa i zamjene utječu samo na koeficijente, ali ne i stupanj polinoma kojim ograničavamo broj koraka.

Konačno, jer je polinom kojim možemo ograničiti broj koraka ovog algoritma drugog stupnja (tj. kvadratni), kažemo da je algoritam kvadratni.

Kao što smo vidjeli u prethodnom primjeru, problem sortiranja brojeva je klase P. Opisano rješenje problema je daleko od optimalnog, te postoje mnogo efikasniji sortovi, kao i usko specijalizirani sortovi koji donose dodatna ubrzanja (npr. Radix sort koji postiže linearnu složenost, ali po cijenu ograničavanja veličine svakog elementa niza).

5. NP-problemi

NP-probleme definiramo slično P-problemima, uz dodatak onog "N" koje znači "nedeterministički". Dakle, NP-problemi (ili problemi klase NP) su oni problemi za koje je moguće definirati nedeterminističke Turingove strojeve koji ih rješavaju u polinomnom vremenu.

U poglavlju 3 smo rekli da nedeterministički stroj, kad ima "dvojbu", ispravno pogađa što će od ponuđenih akcija napraviti. Kad bismo pokušali implementirati takav stroj, morali bismo isprogramirati isprobavanje svih mogućnosti, pa bi broj koraka jako narastao (u pravilu barem eksponencijalno).

Primjer 5.1. (Problem trgovačkog putnika.) *U nekoj državi postoji n gradova. Gradovi A i B povezani su, a udaljenost između njih je $d(A,B)$ (metara, kilometara,... svejedno je; može biti i vrijeme, cijena,...). Traži se najkraći put kojim trgovac može obići sve gradove, te se vratiti na početak, na način da svaki grad posjeti točno jednom. Drugim riječima, traže se gradovi*

$$g_1, g_2, \ldots, g_n$$

takvi da je

$$\{g_1, g_2, \ldots, g_n\} = \{1, 2, \ldots, n\}$$

i da je

$$d(g_1, g_2) + d(g_2, g_3) + \ldots + d(g_{n-1}, g_n) + d(g_n, g_1)$$

najmanje moguće. Takvi putovi koji obiđu svaki vrh točno jednom te se vrate u polazišni vrh grafa zovu se Hamiltonovi ciklusi.

Slično kao kod sortiranja iz primjera 4.1, i ovdje možemo pribjeći traženju svih mogućih putova. Sva argumentacija navedena tamo, vrijedi i ovdje, pa je riječ o izuzetno sporom rješenju.

Ono što nije poznato je postoji li polinomni algoritam koji će ovaj problem riješiti točno. Postoje razni algoritmi koji problem rješavaju brzo, ali rješenja su približna ili ograničena na posebne situacije (npr. zadovoljenost nejednakosti trokuta i sl).

Spomenuli smo da se P- i NP-problemi obično bave odlučivanjem. Problem trgovačkog putnika lako je svesti na da/ne-pitalicu: "Za zadani $x \in \mathbf{R}^+$, postoji li put kojim trgovac može putovati tako da prijeđe udaljenost manju od x?" Ako na to pitanje nađemo odgovor u polinomnom vremenu, onda jednostavnim postavljanjem tog pitanja za nekoliko vrijednosti x_i (ali ne naročito mnogo njih) možemo pronaći i optimalni put.

Primjedba 5.2. *Za koliko različitih vrijednosti x_i bismo trebali postaviti pitanje iz prethodnog paragrafa? Primijetimo, odgovor na pitanje je jednak za vrijednosti x_i i x_j ($x_i < x_j$) ako ne postoji put u grafu duljine x' takve da je $x_i < x' \leq x_j$. Drugim riječima, zanimljivi su nam samo oni x_i koji su jednaki duljini nekog puta u grafu. No, skup svih putova u grafu je sigurno podskup skupa $\mathcal{P}(E)$, gdje je E skup svih bridova u grafu, a $\mathcal{P}(E)$ njegov partitivni skup. Pošto je graf potpun (svaka dva grada su povezana), skup E ima $\frac{n}{2}(n-1)$ elemenata, pa njegov partitivni skup ima*

$$\operatorname{card} \mathcal{P}(E) = 2^{\frac{n}{2}(n-1)}$$

elemenata, a broj kandidata x_i je sigurno manji od toga. Traženi x_i možemo lako naći binarnim traženjem. Prvo sortiramo sve vrijednosti x_i uzlazno po veličini (u primjeru 4.1 smo vidjeli da to možemo u polinomnom vremenu), te provjeravamo može li trgovac

obići sve gradove tako da prijeđe udaljenost najviše x_k *za* $k = \left\lfloor \frac{n}{2} \right\rfloor$ *(drugim riječima, postavljamo pitanje za srednji* x_i*). Ako može, odbacujemo sve* x_i *takve da je* $i \geq \left\lfloor \frac{n}{2} \right\rfloor$ *(jer znamo da i za njih može). U protivnom, odbacujemo one za koje je* $i \leq \left\lfloor \frac{n}{2} \right\rfloor$ *(jer znamo da niti za njih ne može). Zatim ponavljamo postupak za preostale kandidate. Broj kandidata se u svakom koraku raspolavlja, pa je najveći mogući broj koraka jednak* $\log_2(\operatorname{card} \mathcal{P}(E))$*. Konačno, imamo najveći broj postavljanja pitanja o obilasku uz duljinu najviše* x_i *(za odabrane* x_i*):*

$$\log_2(\operatorname{card} \mathcal{P}(E)) = \log_2 2^{\frac{n}{2}(n-1)} = \frac{n}{2}(n-1) = \frac{n^2}{2} - \frac{n}{2},$$

što je polinom. Primijetimo i da je riječ o gruboj gornjoj ogradi jer skup $\mathcal{P}(E)$ *sadrži mnoge podskupove od* E *koji nisu kružni putovi u grafu, pa je stvarni broj postavljanja pitanja puno manji (pravi broj mogućih putova je* $(n-1)!/2$*).*

6. Kako dalje?

Očito, svaki deterministički Turingov stroj je ujedno i specijalna vrsta nedeterminističkog (koji nema niti jednu situaciju u kojoj mora pogađati što dalje), pa su svi P-problemi ujedno i NP-problemi. Milenijsko pitanje je vrijedi li i obrat, tj. jesu li NP-problemi ujedno i P-problemi?

Odgovor na to pitanje, kako smo već rekli, nije poznat. No, ako je $\mathrm{P} \subsetneq \mathrm{NP}$, kako uopće pokazati da među svim tim silnim algoritmima postoji neki kojeg ne možemo dovoljno ubrzati da upadne u klasu P? Ili, ako je ipak $\mathrm{P} = \mathrm{NP}$, kako dokazati da se takvo ubrzavanje može napraviti za svaki algoritam?

Očito, baratanje s beskonačno mnogo poznatih algoritama i onih koje ćemo tek otkriti nije moguće. Zato uvodimo posebnu klasu problema, tzv. NP-potpuni problemi. Kažemo da je problem NP-potpun ako zadovoljava sljedeća dva uvjeta:

1. ako je klase NP (tj. ako se za svako ponuđeno rješenje može u polinomnom vremenu provjeriti rješava li problem), te
2. ako rješavanjem tog problema u polinomnom vremenu možemo i sve ostale NP-probleme riješiti u polinomnom vremenu.

Dakle, ako za neki NP-potpuni problem pronađemo polinomno rješenje, znamo da je $\mathrm{P} = \mathrm{NP}$. Naravno, ako za bilo koji NP (potpuni ili ne) problem dokažemo da nema polinomno rješenje, pokazali smo da je $\mathrm{P} \subsetneq \mathrm{NP}$.

Problem trgovačkog putnika iz poglavlja 5 je jedan od najpoznatijih i najviše proučavanih NP-potpunih problema sa širokim spektrom primjena. Primjer s rasporedom studenata (iz uvodnog poglavlja) također je NP-potpuni problem, usko povezan s problemom bojanja grafa u kojem se traži najmanji broj boja kojima se može obojati vrhove grafa tako da nikoja dva susjedna (povezana) vrha ne budu obojani istom bojom. Naravno, vrhovi grafa predstavljaju studente, a veze u grafu povezuju parove studenata koji ne smiju zajedno biti u domu.

NP-potpuni problemi su izuzetno važni za teoriju složenosti, a mnogi imaju i praktične primjene. O onim najpoznatijima možete više pročitati, na primjer, u [4].

Problem sortiranja (v. primjer 4.1) brojeva je primjer NP-problema koji nije NP-potpun. Za početak, problem je klase NP jer ima jednostavni linearni (dakle, polinomni) algoritam za provjeru ponuđenog rješenja (poretka brojeva). No, problemom sortitanja

n brojeva ne možemo riješiti, na primjer, problem trgovačkog putnika s $p(n)$ gradova, pri čemu je $p(\cdot)$ neki polinom.

Naravno, postavlja se pitanje kako pomoću polinomnog rješenja jednog (NP-potpunog) problema riješiti bilo koji NP-problem u polinomnom vremenu? Osnovna ideja je svođenje drugih problema, uz neke transformacije, na problem koji analiziramo. Očito, te transformacije moraju se izvršavati u polinomnom vremenu. Na primjer, recimo da imamo probleme X i Y. Ako možemo problem X svesti na problem Y u polinomnom vremenu, te ako znamo polinomno rješenje problema Y, tada imamo i polinomno rješenje problema X: u najgorem slučaju, svođenjem na problem Y.

Pogledajmo jedan NP-potpuni problem:

Primjer 6.1. (SAT) *Neka je zadana formula logike sudova (varijable mogu poprimati vrijednosti* ISTINA *i* LAŽ*, a povezane su operatorima* I, ILI *i* NE*). Je li moguće postaviti vrijednosti varijabli u formuli tako da njena vrijednost na kraju bude* ISTINA*?*

SAT je prvi problem za kojeg je dokazano da je NP-potpun. Njegovu NP-potpunost su, neovisno jedan o drugom, otkrili i dokazali Cook [1] i Levin (prijevod na engleski jezik dostupan je u [8]).

Teorem 6.2 (Cook-Levin). *SAT je NP-potpun.*

Dokazivanje Cook-Levinovog teorema nadilazi područje ovog članka, te ga ovdje nećemo izvoditi. Jedan od dokaza Cook-Levinovog teorema možete pronaći, na primjer, u [2].

Osnovna ideja dokaza je konstruiranje logičkog izraza koji poprima vrijednost ISTINA ako neki zadani nedeterministički Turingov stroj završava u konačnom vremenu s potvrdnim odgovorom.

Drugim riječima, ako imamo NP-problem (da/ne-pitalicu), onda imamo i nedeterministički Turingov stroj koji taj problem rješava u konačnom vremenu. Taj stroj možemo (pomoću konstrukcije u dokazu) prevesti u logički izraz koji je istina ako i samo ako stroj vraća potvrdan odgovor. Na taj je način traženje vrijednosti varijabli tako da izraz bude istinit ekvivalentno izvršavanju nedeterminističkog Turingovog stroja. Riješimo li SAT u polinomnom vremenu, riješili smo i početni (proizvoljni!) NP-problem u polinomnom vremenu.

S Cook-Levinovim teoremom obavljen je početni dio posla oko NP-potpunih problema: pronađen je jedan takav. Pretpostavimo sada da za neki proizvoljni NP-problem X želimo pokazati da je NP-potpun. Kako ćemo to napraviti?

Odgovor je jednostavan: polinomnim svođenjem drugog NP-potpunog problema (npr. SAT-a) na X. Na taj način, ako nađemo polinomno rješenje problema X, našli smo i polinomno rješenje svih NP-potpunih problema (jer se svaki može polinomno svesti na svakog od njih).

Vratimo se na trenutak na problem sortiranja brojeva i problem trgovačkog putnika. Očito, ako imamo zadan potpuni graf s n gradova, možemo popisati sve putove u grafu, te uzlazno sortirati njihove duljine. Prvi put u tako sortiranom nizu putova bit će najkraći, pa na taj način problem trgovačkog putnika možemo svesti na problem sortiranja brojeva. Problem leži u tome da je broj Hamiltonovih ciklusa u potpunom grafu (dakle, broj mogućih putova trgovačkog putnika) jednak $(n-1)!/2$. Drugim riječima, da bismo riješili problem trgovačkog putnika za n gradova, moramo riješiti problem sortiranja za $(n-1)!/2$ brojeva, što je nadpolinomno veliko, pa problem trgovačkog putnika nismo uspjeli **polinomno** svesti na problem sortiranja brojeva.

Kako se problemi svode jedan na drugi, možete pročitati u članku [3] u kojem se detaljno analizira veza poznate igre Minesweeper i SAT-a.

7. Zaključak

Očito, klasu NP-potpunih problema čine problemi koje opravdano možemo zvati "teškima", jer ih ne znamo riješiti u polinomnom vremenu (niti znamo je li to moguće).

Problem trgovačkog putnika samo je jedan primjer problema kombinatorne optimizacije koji ima široku primjenu. Jedna očita primjena je organizacija dostave neke robe nizu prodajnih mjesta. No, taj problem i njegove razne modifikacije imaju široku primjenu u izradi računalnih sklopova, robotici, rudarstvu, organizaciji pogona za proizvodnju međusobno različitih proizvoda,... Jasno je da bi pronalaženje efikasni(ji)h rješenja ovog problema dovelo do mnogo korisnih efekata. Slično, naravno, vrijedi i za mnoge druge NP-potpune probleme.

Kako rješavamo mnoge probleme koji su direktno vezani uz NP-potpune probleme? Pošto nema efikasnih egzaktnih algoritama, pribjegava se raznim "slabijim" algoritmima koji

- rješavaju samo neku usku podklasu problema koji proučavamo, ili
- ne garantiraju optimalnost rješenja (ali za gotovo sve ulazne podatke daju rješenje blisko optimalnom), ali garantiraju brzinu ili
- garantiraju da je dobiveno rješenje uvijek proizvoljno blizu optimalnom, ali ne garantiraju brzinu (tj. u pravilu su brzi, ali je moguće konstruirati ulazne podatke za koje nisu).

Izbor metoda jako ovisi o primjenama. Na primjer, možemo fiksirati neke parametre i tako smanjiti složenost, uz cijenu ograničavanja algoritma na jako specifični problem (slično ograničavanju Radix sorta na cijele brojeve). Možemo si pomoći i randomizacijama (izvođenje pojedinih koraka algoritma u ovisnosti o pseudoslučajnim brojevima), što je posebno korisno za traženje početne točke iterativnih algoritama.

Od općenitijih metoda široko su rasprostranjene razne heuristike: algoritmi koji se općenito "dobro ponašaju" (eksperimentalno je pokazano da su brzi i daju optimalna ili njima bliska rješenja), no moguće je konstruirati i primjere za koje se dugo izvršavaju i/ili daju rješenja daleko od optimalnog.

Primjedba 7.1 *Ponekad je poželjno da problemi budu teški. Pogledajmo problem kriptiranja podataka na kojem se bazira sigurnost raznih sustava. Na primjer, plaćanja preko weba koriste enkripciju kako bi se spriječile krađe podataka o kreditnim karticama i sl. Očito, da bi sustav funkcionirao, ekripcija i dekripcija moraju biti brzi algoritmi za one koji imaju potrebne podatke (npr. lozinku), a izuzetno spori (dakle "teški") za one koji te podatke nemaju.*

No, recimo da imamo neki niz znakova L za koji mislimo da bi mogla biti lozinka za dekripciju nekog teksta T. Kako provjeriti da je L ispravna lozinka za T? Najjednostavnije je pokušati dekriptirati tekst T s nizom znakova L. Ako dobijemo smislene podatke, gotovo je sigurno da je L lozinka. Drugim riječima, problem dekripcije bez lozinke možemo smatrati NP-problemom (ima brzu provjeru i potencijalno sporo traženje). Kad bi ispalo da je $P=NP$, to bi značilo da postoje i polinomni algoritmi za otkrivanje lozinki, neovisno o primijenjenim metodama enkripcije (dok god se držimo razumne pretpostavke da brzo radimo ako znamo lozinku).

Treba naglasiti da su ovo teorijski koncepti. Između dokazivanja da postoji polinomni algoritam za rješavanje nekog problema i pronalaženja efikasnog algoritma (tj. onog koji je koristan u praksi), dug je put. Iako se polinomnost obično veže uz efikasnost, algoritam koji ima polinomnu složenost za polinom, na primjer, sedamnaestog stupnja

i velikih koeficijenata i dalje neće biti brz (iako će za dovoljno velike ulazne podatke sigurno biti bitno brži od, recimo, algoritama eksponencijalne složenosti).

Većina polinomnih algoritama koji se koriste u praksi, ima složenost u ovisnosti o polinomu najviše četvrtog stupnja.

Problem "P = NP?" i dalje je otvoreni problem čije proučavanje donosi mnoga korisna i zanimljiva otkrića. Čak i ako ne osvojite glavnu nagradu od milijun dolara, možda – rješavajući taj problem – riješite neki od brojnih srodnih problema ili pronađete novu metodu za približno rješavanje teških problema. Sretno!

Literatura

[1] S. A. Cook, The complexity of theorem-proving procedures. In *STOC'71: Proceedings of the third annual ACM symposium On Theory of computing*, pages 151–158, New York, USA, 1971. ACM.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[3] V. Kojić, Minesweeper problem je NP-potpun. *math.e*, (12), 2008. `http://e.math.hr/old/minesweeper/index.html`.

[4] List of NP-complete problems. `http://en.wikipedia.org/wiki/List_of_NP-complete_problems`.

[5] Millennium problems. `http://www.claymath.org/millennium/`.

[6] V. Šego, *Programiranje 1 (vježbe)*. preprint, 2009. `http://degiorgi.math.hr/prog1/materijali/p1-vjezbe.pdf`.

[7] Selection sort. `http://en.wikipedia.org/wiki/Selection_sort`.

[8] B. Trakhtenbrot, A survey of russian approaches to perebor (bruteforce searches) algorithms. *IEEE Annals of the History of Computing*, 6(4):384–400, 1984.

[9] A. M. Turing, On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.