AZ ALGORITMIKUS BONYOLULTSÁGELMÉLET ÚJ EREDMÉNYEI

Mann Zoltán Ádám

Budapesti Műszaki és Gazdaságtudományi Egyetem,
Számítástudományi és Információelméleti Tanszék
1117 Budapest, Magyar tudósok körútja 2.

Ez a cikk itt jelent meg: *Pro Scientia Aranyérmesek X. Konferenciája, Budapest, 2010*

1. Bevezetés

Számos területen találkozhatunk nehéz algoritmikus problémákkal, pl. a logisztikában, számítógépes rendszerek tervezésében vagy az órarendkészítésben. Egy adott problémára sok különböző algoritmus adható, amelyek hatékonysága nagyon eltérő lehet.
Példaként tekintsük az összeadás problémáját: inputként kapunk két pozitív egész számot, és ki kell számolnunk az összegüket. Erre egy lehetséges algoritmus a jól ismert írásbeli összeadás; nevezzük ezt ALG1-nek. Egy másik lehetőség a következő: az első számot eggyel növeljük, a másodikat eggyel csökkentjük. Ettől az összegük nyilván nem változott. Ismételjük ezt addig, amíg a második szám 0 nem lesz; ekkor az első szám épp a keresett összeg. Legyen ez az algoritmus ALG2.
Nyilván ALG1 és ALG2 egyaránt helyes, hiszen mindkettő véges sok lépésben véget ér és megadja a keresett összeget. Azonban gondoljuk meg, hogy a két algoritmus mennyi idő alatt ér célba. A pontos futási idő természetesen nagyon sok tényezőtől függ, de a nagyságrendjét azért meg tudjuk becsülni. Tegyük fel, hogy két n számjegyű számot kell összeadni. Ekkor ALG1 lépésszáma n-nel arányos, míg ALG2 lépésszáma magával az egyik inputként kapott számmal, ami nagyságrendileg 10^n.

1. Táblázat: Példa algoritmusok lépésszáma az input méretének függvényében

Input mérete (n):	**3**	**20**	**28**
ALG1 lépésszáma:	3	20	28
ALG2 lépésszáma:	1000	10^{20}	10^{28}

Mint az 1. Táblázatban látható, a két algoritmus lépésszáma n növekedésével drasztikusan eltér egymástól. ALG2 futtatása egy mai számítógépnek n=20 esetén kb. 1000 évébe, míg n=28 esetén kb. 100 milliárd évébe telne, míg ALG1 ennél lényegesen nagyobb inputok esetén is a másodperc törtrésze alatt lefut. Tehát nagyon fontos, hogy az adott problémára milyen hatékonyságú algoritmust használunk. Éppen ezzel foglalkozik az algoritmikus bonyolultságelmélet.

2. „Klasszikus” bonyolultságelmélet

Mint a fenti példából is látható, a fő kérdés egy algoritmus bonyolultságának meghatározásánál, hogy az algoritmus lépésszáma az input méretétől hogy függ. A bonyolultságelmélet egyik alapfeltevése, hogy azok az algoritmusok tekinthetők hatékonynak, melyek lépésszáma az input méretének valamely hatványával becsülhető felülről. Az ilyen algoritmusokat polinomiális algoritmusnak nevezik. Ilyen például a fentiekben szereplő ALG1, melynek lépésszáma lineáris az input hosszában, de ide tartoznak pl. a négyzetes vagy

köbös lépésszámú algoritmusok is. A fenti ALG2 lépésszáma exponenciális az input méretének függvényében, és az exponenciális függvény gyorsabban nő minden polinomnál, így ALG2 nem polinomiális algoritmus.
Az algoritmikus bonyolultság klasszikus elmélete bonyolultsági osztályokba sorolja a problémákat annak megfelelően, hogy a problémára adható-e hatékony algoritmus. Az elmondottak szerint például az összeadás problémája egy könnyű probléma, mivel létezik rá hatékony algoritmus (ALG1).
Sajnos a gyakorlatban előkerülő problémáknak egy jelentős részére nem ismerünk hatékony algoritmust. Mivel kb. fél évszázada nagy erőkkel keresünk ilyen algoritmust, de eddig hiába, általánosan elterjedt sejtés, hogy ezekre a problémákra – az úgy nevezett NP-nehéz problémákra – nem is létezik hatékony algoritmus, tehát ezek valóban nehéz problémák. Azonban ezt mindezidáig nem sikerült bizonyítani. Pusztán annyit sikerült megmutatni, hogy e problémák kölcsönösen visszavezethetők egymásra, így ha valamelyikre volna hatékony algoritmus, akkor automatikusan volna a többire is.
Néhány példa NP-nehéz problémákra:

- Logikai fejtörők, pl. Sudoku
- Adott feltételeknek megfelelő órarend készítése
- Szabályrendszer konzisztenciájának ill. teljességének eldöntése
- Áramkör minimális területen történő huzalozása
- Raktárak helyének és a szállítási utak optimális kijelölése

A sor még hosszan folytatható a legkülönbözőbb területeken gyakorlati relevanciával bíró NP-nehéz problémákkal [9].

3. A klasszikus elmélet korlátai

A fent vázolt klasszikus elmélet fő korlátja, hogy az NP-nehéz problémák kezelésében nem ad segítséget. Számos tudós fáradozik azon, hogy bebizonyítsa: e problémákra valóban nem létezik hatékony algoritmus. De persze, ha ezt egyszer sikerül bebizonyítani, az nem fog segíteni abban, hogy NP-nehéz problémákat megoldjunk, márpedig erre a gyakorlatban szükség van.
Nem túl valószínű ugyan, de elvileg lehet, hogy egy napon sikerül polinomiális algoritmust adni az NP-nehéz problémákra. Ám az eddigi tapasztalatok alapján úgy tűnik, hogy ezek a problémák tényleg nehezek, így egy ilyen algoritmus jó eséllyel csak elméletben lenne hatékony, a gyakorlatban nem. Ez sem segítene tehát az NP-nehéz problémák gyakorlati megoldásában, hanem csak arra mutatna rá, hogy hibás a bonyolultságelméletnek az a feltételezése, miszerint a polinomiális algoritmusok a hatékony algoritmusok.

4. Bonyolultságelmélet új megközelítésben

A fentiek miatt az NP-nehéz problémák tanulmányozására és kezelésére az utóbbi években egy új megközelítés van kialakulóban. Ennek lényege, hogy az algoritmusok legrosszabb esetben mutatott viselkedése helyett a tipikusnak tekinthető probléma példányokon mutatott viselkedését vizsgáljuk. Ha ugyanis egy probléma NP-nehéz, ez csupán annyit jelent, hogy valószínűleg nincs olyan algoritmus, mely minden probléma példányon – vagy másképp fogalmazva: a legrosszabb esetben is – hatékony lenne. Ez azonban nem zárja ki azt, hogy a probléma példányok jelentős hányadát – szerencsés esetben éppen a gyakorlatban előfordulókat – hatékonyan meg tudjuk oldani. Gondoljunk például a Sudokura: nincs olyan módszerünk, mely minden esetben hatékonyan megoldja a rejtvényt, de vannak könnyebb és nehezebb példányok, és bizonyos módszerekkel a könnyűeket valóban gyorsan meg tudjuk oldani.

Mivel az új megközelítés nem problémák, hanem probléma példányok bonyolultságát vizsgálja, így fontos szerep jut az empirikus vizsgálatoknak, valamint a mérési eredményeket megmagyarázó – általában statisztikus – modelleknek. Itt tehát hasonló szereposztás és együttműködés van kialakulóban, mint a természettudományokban (pl. kísérleti fizika és elméleti fizika).
Az új bonyolultságelméleti vizsgálatok egyik központi kérdése, hogy min múlik az egyes probléma példányok bonyolultsága, vagyis mitől lesz az egyik probléma példány lényegesen bonyolultabb, mint egy másik, méretében hasonló példány. A tapasztalt bonyolultságbeli eltérés bizonyos esetekben algoritmusfüggő, azaz az adott algoritmus számára az egyik példány nagyobb kihívást jelent a másiknál. Más esetekben azonban úgy tűnik, hogy egy általánosabb jelenségről van szó, mivel nagyon különböző elven működő algoritmusok is ugyanazt a példányt találják nehéznek illetve könnyűnek.
A bonyolultság forrásának megértése természetesen nem csupán elméleti érdekesség, hanem hosszútávon hozzásegít minket ahhoz, hogy hatékonyabb algoritmusokat készítsünk.

5. Néhány konkrét eredmény

A következőkben néhány példát láthatunk a bonyolultságelmélet újfajta eredményeire.

5.1. Kiugróan nagy futási idők

Korábban szó esett arról, hogy egy probléma megoldására szolgáló különböző algoritmusok futásideje között óriási eltérés lehet. A tapasztalat azonban azt mutatja, hogy már egyetlen algoritmus esetén is, ha azt különböző, egyébként hasonló méretű és struktúrájú inputokra futtatjuk, sok esetben nagyon nagy szórást tapasztalhatunk a futásidőben. Sőt, randomizált (tehát véletlen döntéseket is használó) algoritmusok esetén még egyetlen probléma példány esetén is, az algoritmus különböző futásai nagyon eltérő futásidőket produkálhatnak.
A gyakorlatban ez azt jelenti, hogy pl. egy algoritmus bizonyos méretű probléma példányokat általában 1-2 perc alatt old meg, de előfordul, hogy egy másodperc alatt elkészül, és az is, hogy a futás napokat vesz igénybe. Az ilyen kiugró idők persze nem gyakoriak, de közel sem annyira ritkák, mint azt várnánk. A futási idő eloszlása jellemzően messze van a normális eloszlástól: a Gauss-féle haranggörbe helyett sokkal szélesebben elnyúló görbékkel találkozunk [2].
Ez szemléletesen azt jelenti, hogy az algoritmusnak néha szerencséje van: véletlenül pont a megfelelő döntéseket hozva, nagyon hamar eljut a megoldáshoz. Máskor épp fordítva, véletlenül szerencsétlen irányokba kezd kutatni, és csak nagyon sokára derül ki, hogy a keresési térnek egy meddő területére tévedt.

5.2. Gyakori újraindítás

A fenti jelenségnek egy érdekes felhasználását jelenti a gyakori újraindítás stratégiája. Az előző gondolatmenetet folytatva, tegyük fel, hogy van egy randomizált algoritmusunk, mely adott méretű probléma példányokat általában 1-2 perc alatt old meg. Tegyük fel, hogy az algoritmus már 2 perce fut. A javaslat az, hogy állítsuk le és futtassuk újra. Ez tulajdonképpen egy meglepő ötlet, hiszen azt gondolhatnánk, hogy az algoritmus már sok tudást összegyűjtött a probléma példányról, és esetleg pár másodperc múlva el is készülne, így pedig ismét a nulláról indulunk és kárba vész a felhasznált idő.
A tapasztalat azonban azt mutatja, hogy az intuícióval szemben, a gyakori újraindítás stratégiája általában jelentős megtakarítást eredményez. Ennek oka, hogy minél hosszabb ideje fut már az algoritmus, annál valószínűbb, hogy véletlenül pont egy kiugróan hosszú

ideig tartó futással van dolgunk. Az újraindítás révén ez elkerülhető; sőt, arra is van esélyünk, hogy a következő futás esetleg egy kiugróan rövid lesz.
Természetesen nem mindegy, hogy mennyi idő után indítjuk újra az algoritmust. Ha ismert a futásidő eloszlása, akkor pontosan kiszámítható az optimális újraindítási idő [3]. Ha a futásidő eloszlásáról nem tudunk semmit, akkor érdemes viszonylag kis újraindítási idővel kezdeni, majd minden újraindítás után növelni az újraindítási időt. Az algoritmus ismételt futtatásai és újraindításai arra is lehetőséget adnak, hogy menet közben gépi tanulási módszerekkel tanuljunk meg minél többet a futásidő eloszlásáról, és a bővülő tudást felhasználva, egyre jobban közelítsük az optimális újraindítási időt [10].
Ha az algoritmus determinisztikus, vagyis ugyanazon inputon futtatva mindig ugyanazt csinálja, az újraindítás természetesen nem segít. Másképpen fogalmazva, érdemes a determinisztikus algoritmust véletlen elemek beépítésével randomizálni, mert így már használható lesz az újraindítás eszköztára.

5.3. Fázisátmenet

Tekintsünk egy eldöntési problémát. Pl. tegyük fel, hogy egy Sudoku-tábla egyes celláiba véletlenszerűen beírunk számokat, és a kérdés, hogy befejezhető-e a Sudoku úgy, hogy a végeredmény megfeleljen a szabályoknak. Ha csak kevés cellába írtunk kezdetben számot, akkor nagy valószínűséggel kiegészíthető a Sudoku. Ha pedig a tábla nagy részét kitöltjük véletlen módon, akkor nagy valószínűséggel nem oldható meg a rejtvény.
Az érdekes az, hogy a kitöltött cellák számának növelése eleinte alig csökkenti a befejezhetőség valószínűségét, ami aztán egy kritikus tartományban hirtelen zuhan majdnem 1-ről majdnem 0-ra, és utána már ismét alig csökken a kitöltött cellák számának további növelésétől. A jelenséget fizikai analógiára fázisátmenetnek hívják, mivel az olvadás/fagyás jelenségére emlékeztet [8]. A Sudoku mellett számos más algoritmikus problémánál is megfigyelhető ugyanilyen fázisátmenet: a korlátok számának növelésével az alul-korlátozott tartományból a kritikusan korlátozott tartományon keresztül a túl-korlátozott tartományba jutunk. Az alul-korlátozott tartományban a probléma példányok majdnem mindig megoldhatóak, a túl-korlátozott tartományban majdnem mindig megoldhatatlanok, a kettő között pedig egy keskeny sávban változik a megoldhatóság valószínűsége majdnem 1-ről majdnem 0-ra.
A fázisátmenet a bonyolultság változásával is együtt jár. Az alul-korlátozott tartományban általában könnyű megtalálni egy megoldást, a felül-korlátozott tartományban pedig általában könnyen látható a megoldhatatlanság. A bonyolultság ezért általában a kritikusan korlátozott tartományban tetőzik. Ráadásul ebben a tipikusan igen keskeny tartományban nagyságrendekkel nehezebbek a problémák, mint máshol, vagyis egy nagyon markáns könnyű-nehéz-könnyű mintázatot mutat a bonyolultság.

6. Bonyolultságelméleti kutatások a BME-n

Ilyen és ehhez hasonló jelenségek kutatásával (is) foglalkozunk a BME Számítástudományi és Információelméleti Tanszékén. Kutatómunkánknak három fő területe van:

- Optimalizálási algoritmusok empirikus vizsgálata [7]
- A tapasztalt jelenségeket magyarázó elméleti modellek [5,6]
- Műszaki alkalmazások (vezeték nélküli hálózatok, számítógép architektúrák) [1,4]

Az implementációs és mérési feladatok támogatására létrehoztunk egy szoftver keretrendszert (BCAT, Budapest Complexity Analysis Toolkit), mely kutatási célokra ingyenesen letölthető a http://www.sourceforge.net/projects/bcat címről [7].

Irodalomjegyzék

[1] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 10(1), pp. 136-156, 2005.
[2] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24, pp. 67-100, 2000.
[3] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 29, pp. 173-180, 1993.
[4] Zoltán Ádám Mann, András Orbán, and Péter Arató. Finding optimal hardware/software partitions. *Formal Methods in System Design*, 31(3), pp. 241-263, 2007.
[5] Zoltán Ádám Mann and Anikó Szajkó. Determining the expected runtime of exact graph coloring. *Mini-Conference on Applied Theoretical Computer Science*, 2010.
[6] Zoltán Ádám Mann and Anikó Szajkó. Improved bounds on the complexity of graph coloring. *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2010.
[7] Zoltán Ádám Mann and Tamás Szép. BCAT: A framework for analyzing the complexity of algorithms. *8th IEEE International Symposium on Intelligent Systems and Informatics*, pp. 297-302, 2010.
[8] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 400, pp. 133-137, 1999.
[9] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
[10] Yongshao Ruan, Eric Horvitz, and Henry Kautz. Restart policies with dependence among runs: a dynamic programming approach. *8th International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pp. 573-586, 2002.