SZAKDOLGOZAT

Major Sándor Roland

Debrecen

2009

Debreceni Egyetem

Informatikai Kar

A P vs. NP probléma vizsgálata

Témavezető:

Dr. Herendi Tamás

Egyetemi adjunktus

Készítette:

Major Sándor Roland

Programtervező informatikus

Debrecen

2009

Tartalomjegyzék

1. Bevezetés

A P vs. NP probléma a bonyolultságelmélet területének meghatározó problémája. A kérdés önmagában véve nagyon egyszerű, mindössze két halmaz egyenlőségére vonatkozik. A kimondásához nem szükséges sok matematikai előismeret. Jelentőségének és igazi összetettségének megértése azonban minden, csak nem egyszerű.

A szóban forgó két halmaz két bonyolultsági osztály, (amelyek pontos definícióját a későbbiekben fogjuk megadni), azaz a halmazok elemei problémák. A bonyolultságelmélet a problémák bonyolultságát vizsgálja abból a szempontból, hogy a megoldásukhoz mennyi szükséges egy adott erőforrásból, például időből. Az, hogy egy problémát "könnyűnek" vagy "nehéznek" nyilvánítunk, szorosan összefügg ezen két osztállyal.

A probléma sok más kérdéssel áll kapcsolatban. Nem kevés problémáról sikerült már belátni, hogy valójában ekvivalensek a P vs. NP-vel. A kérdés központi voltát mi sem bizonyítja jobban, mint hogy számtalan különböző megfogalmazásban látott már napvilágot. Nem csupán az elméleti jelentősége hatalmas, a megfejtésének nagyon nagy gyakorlati következményei lennének. A P=NP állítás konstruktív bizonyítása óriási változásokat hozhatna a számítástudomány minden területén. Képesek lennénk olyan feladatok gyors és hatékony megoldására, amelyek ma reménytelenül nehéznek tűnnek. Vegyük például az elektronikus pénzügyi akcióknál elterjedt RSA titkosítási rendszert. Jelenleg a rendszer biztonságos, mivel a megfelelő kulcs nélkül a dekódolásra nem ismert olyan hatékony algoritmus, amely belátható időn belül működne. Egy ilyen bizonyítás egyben egy ilyen algoritmus leírását is megadná. Sajnos mint azt látni fogjuk, egy ilyen bizonyítás egyre valószínűtlenebbnek és elérhetetlenebbnek tűnik. Ha a bizonyítás nem lenne konstruktív, aminek jelenleg igen nagy esélyt tulajdonítanak, az is elég lenne ahhoz, hogy megbizonyosodjunk a keresett hatékony megoldások létezéséről. A $P \neq NP$-re mutató jelek komoly túlsúlyban vannak, de a tény, hogy rengeteg ráutaló jel ellenére sem sikerült mindent eldöntő bizonyítást találni, azt mutatja, hogy a kérdés jóval mélyebb, mint azt első ránézésre meg lehetne róla mondani.

A P vs. NP fontosságát jól példázza, hogy helyet kapott a 2000-ben összeállított Millennium Prize Problems között. Ezen hét probléma mindegyike a matematika nagy megoldatlan kérdése, jelentőségükben a Hilbert-problémákhoz mérhetőek, amelyek meghatározták a huszadik század matematikájának fejlődési irányát. Ez a hét kérdés hasonló szerepet játszhat a huszonegyedik század tudományos történelmében, és ennek elősegítése érdekében a Clay Mathematics Institute bármelyik megoldásáért egymillió dolláros jutalmat ajánlott fel. A problémák megoldásának gyakorlati jelentősége ennél a "jelképes összegnél" is jóval többet ér.

A P vs. NP nagyjából negyven-ötven évnyi történelme során már az egyes részmegoldások felé vezető út is meglehetősen rögös volt. A megoldására tett kísérletekhez eszközök és módszerek széles skáláját találták ki. Ezek közül nem kevésről sikerült belátni, hogy vagy csak speciális esetekbe nyerhetünk velük betekintést, vagy teljesen alkalmatlanok a probléma eldöntésére. Hamar felmerült annak a lehetősége is, hogy a kérdés eldönthetetlen.

A P vs. NP probléma komoly figyelemnek örvend. Ebben a szakdolgozatban össze kívánom gyűjteni a megismeréséhez szükséges főbb ismereteket, és a terület hozzá kapcsolódó fontosabb eredményeit. A bonyolultságelmélet szükséges alapfogalmainak áttekintése után szót ejtek a probléma történelméről, majd a megoldási kísérletek fontosabb irányvonalairól. A dolgozatban szó esik az NP-teljes problémák jelentőségéről, a Chomsky-hierarchia P illetve NP-hez képesti viszonyáról, az orákulumos Turing-gépek tulajdonságairól valamint néhány eredményről a Boolean-hálózatok területén. Ezek mellett mellékelek egy Java nyelven írt programot, amellyel Turing-gép (mind determinisztikus, mind nemdeterminisztikus) és veremautomata működését lehet szimulálni. A programhoz írt osztályok dokumentációját mellékletként csatolom a dolgozathoz.

2. Alapvető ismeretek

A P vs. NP probléma megfogalmazásához mindenekelőtt szükséges néhány számításelméleti alapfogalom ismerete. Ebben a részben áttekintem azokat az alapvető definíciókat, melyek a probléma kimondásához elengedhetetlenek.

Az algoritmus fogalmának nem létezik matematikai definíciója. Algoritmus alatt általánosan fogalmazva egy feladat megoldására alkalmas, meghatározott lépések véges sorozatát értjük. Ez önmagában nem elegendő arra, hogy az algoritmusok precíz matematikai vizsgálatát lehetővé tegye, ehhez szükségünk lesz az algoritmusok egy (lehetőleg minél általánosabb) modelljének megadására.

Definíció: Turing-gép

A T Turing-gép alatt értsük a

$$T =< k, Q, \Sigma, q_0, \Phi >$$

ötöst, ahol k a gép szalagjainak száma ($k \geq 1$ egész), Q egy nem üres véges halmaz, a belső állapotok halmaza, Σ szintén egy nem üres véges halmaz, a szalagábécé, $q_0 \in Q$ a kezdőállapot, Φ pedig az átmenetfüggvény, melynek alakja

$$\Phi : Q \times \Sigma^k \longrightarrow 2^{Q \times \Sigma^k \times M^k}$$

M={balra, helyben, jobbra}
A gép szalagjai cellákra vannak osztva, melyek mindegyike egy szimbólumot tartalmaz a szalagábécé készletéből. A szalagokat mindkét irányba végtelen hosszúnak tekintjük. Minden szalag egy "író-olvasó fejjel" van ellátva, amely a szalag "aktuális" celláját jelöli ki. Ezen kívül a gép rendelkezik egy belső állapottal, amelynek lehetséges értékeit a Q

halmaz tartalmazza. A Turing-gépet determinisztikusnak hívjuk, ha átmenetfüggvénye egyértelmű (azaz egy adott baloldalhoz csak legfeljebb egy jobboldal létezik). Ez valódi részhalmaza a nemdeterminisztikus Turing-gépeknek, amikre ezt a feltételt nem szabjuk ki. (azaz az átmenetfüggvény egy baloldalához akárhány jobboldal tartozhat)

Definíció: Turing-gép konfigurációja

A Turing-gép egy konfigurációja alatt egy szót értünk, melynek szerkezete:

$$w \in Q(\Sigma^*\#)^k\mathbb{Z}^k$$

A szó tartalmazza a gép jelenlegi belső állapotát, a szalagok tartalmát valamilyen módon (pl. "#" jellel) elválasztva, és tartalmazza, hogy az egyes szalagokon az I/O (író/olvasó) fejek hanyadik cellán állnak.

A Turing-gép a Φ által meghatározott módon megy át egyik konfigurációból a másikba. A T Turing-gép egy lépésben átmegy a K_1 konfigurációból a K_2-be ($K_1 \to K_2$) , ha

$$K_1 = q_1 a_{11} a_{12} ... a_{1n_1} \# a_{21} a_{22} ... a_{2n_2} \# ... \# a_{k1} a_{k2} ... a_{kn_k} \# \alpha_1 \alpha_2 ... \alpha_k$$

$$\mathrm{K}_2 = q_2 b_{11} b_{12} ... b_{1n_1} \# b_{21} b_{22} ... b_{2n_2} \# ... \# b_{k1} b_{k2} ... b_{kn_k} \# \beta_1 \beta_2 ... \beta_k$$

$$\Phi(q_1, a_{1,\alpha_1}, a_{2,\alpha_2}, ..., a_{k,\alpha_k}) = (q_2, b_{1,\alpha_1}, b_{2,\alpha_2}, ..., b_{k,\alpha_k}, m_1, m_2, ..., m_k)$$

$\beta_i = \alpha_i + m_i \quad (i = 1, ..., k) \quad m_i = -1, 0, 1$

$b_{ij} = a_{ij}$ *ha* $j \neq \alpha_i$ $j = 1, ..., n_i$

Egy lépés során a Turing-gép jelenlegi belső állapotának és az I/O fejek alatti szimbólumok függvényében megváltoztatja belső állapotát valamint a beolvasott cellák tar-

talmát, majd az I/O fejeket (egymástól függetlenül) vagy egy cellával balra mozdítja, vagy helyben hagyja, vagy eggyel jobbra mozdítja.

Definíció: Számítás

A K_1, K_2, K_n konfiguráció sorozatot a T Turing-gép egy számításának nevezzük, ha mind szabályos konfigurációk, és $K_i \to K_{i+1}$ teljesül $i = 0, ..., n-1$-re.

Egy konfigurációt kezdőkonfigurációnak nevezünk, ha

$$K = q_0 w_1 \# w_2 ... w_k 1^k$$

ahol $w_1, ..., w_k$ a T gép bemenete.

A Turing-gép megáll K-n, ha $\nexists K' : K \to K'$.

A T Turing-gép a w bemenetre a w' választ adja, ha $\exists K_0, ..., K_n$ számítás, ahol K_0 kezdő konfiguráció, K_n-n megáll a gép és $K_n = qw' \# \lambda \# ... \# \lambda \# 1^k$

Ha a Turing-gép determinisztikus, akkor a számítás egyértelmű, egy bemeneti szóra csak egy úton halad. Ha a gép nemdeterminisztikus, akkor számítási fa épül fel, ahol az elágazások a nemdeterminisztikus választási lehetőségeket reprezentálják.

Definíció: Elfogadó Turing-gép

A

$$T =< k, Q, \Sigma, q_0, \Phi, F >$$

Turing-gépet elfogadó Turing-gépnek nevezzük, ahol $F \subseteq Q$.
F az elfogadó állapotok halmaza. Ha a gép F-beli állapotba kerül, akkor a számítás véget ér. Determinisztikus esetben a Turing-gép elfogadja a w szót, ha a számítás végén elfogadó állapotban áll meg. Nemdeterminisztikus esetben akkor fogadja el, ha létezik olyan levélelem a számítási fában, amely elfogadó állapotban állt meg.

Definíció: A Turing-gép által felismert nyelv

A T Turing-gép felismeri az L nyelvet, ha pontosan $L \in \Sigma^*$ szavain áll meg elfogadó állapotban. Tehát:
$T\ elfogadja\ w - t \leftrightarrow w \in L\ \ (L = L(T))$
Ha a géptől megköveteljük, hogy minden bemeneten megálljon, akkor *rekurzív* nyelvről beszélünk. Ha csupán annyit követelünk meg tőle, hogy az L nyelv szavait biztosan álljon meg elfogadó állapotban, az L-en kívül eső szavakat pedig vagy utasítsa el, vagy fusson végtelen ideig, akkor *rekurzív felsorolható* nyelvről beszélünk.

Ha a T Turing-gép a w bemeneten megáll és közben a $K_0, ..., K_n$ számítást végzi, akkor azt mondjuk, hogy a T a w szón n lépésben megáll.

Definíció: Időbonyulultság

Legyen a $t : \mathbb{N} \to \mathbb{N}$ függvény a T Turing-gép időbonyolultsága, ha $\forall |w| \leq n$-re T legfeljebb t(n) lépésben megáll. Ha a gép nemdeterminisztikus, akkor ennek a feltételnek a legrövidebb elfogadó számításra kell teljesülnie.

Definíció: Tárbonyululsság

Legyen a $s : \mathbb{N} \to \mathbb{N}$ függvény a T Turing-gép tárbonyolultsága, ha $\forall |w| \leq n$-re T működése közben a szalagokon szereplő szavak összhossza legfeljebb s(n). Nemdeterminisztikus esetben itt is a legrövidebb elfogadó számításnak kell ezt teljesítenie.

Az idő- és tárbonyolultság fogalmának segítségével képesek vagyunk bonyolultsági osztályokat definiálni. Ezekkel az osztályokkal a nyelveket csoportosítjuk aszerint, hogy az őket megoldó algoritmusok idő- illetve tárbonyolultsága milyen.

Definíció: Time(f)

Legyen $f : \mathbb{N} \to \mathbb{R}^+$ monoton növekvő függvény. Definiáljuk a Time(f) halmazt a következőképpen:

$$Time(f) = \{L | \exists T \; determinisztikus \; Turing-gép, \; L = L(T), \; T \; időbonyolultsága \; O(f(n))\}$$

azaz legyen Time(f) azon nyelvek halmaza, melyekhez létezik determinisztikus Turing-gép amely pontosan az adott nyelvet ismeri fel és az időbonyolultsága O(f).

Definíció: Space(f)

A Time(f) mintájára definiáljuk a következőképpen:

$$Space(f) = \{L | \exists T \; determinisztikus \; Turing-gép, \; L = L(T), \; T \; tárbonyolultsága \; O(f(n))\}$$

azaz legyen Space(f) azon nyelvek halmaza, melyekhez létezik determinisztikus Turing-

gép amely pontosan az adott nyelvet ismeri fel és a tárbonyolultsága O(f).

Ezen halmazokhoz hasonló módon definiáljuk még az NTime(f) és NSpace(f) halmazokat, melyek csak annyiban térnek el a fenti két definíciótól, hogy nem kötjük ki, hogy a felismerő Turing-gépnek determinisztikusnak kell lennie. Nyilvánvalóan $Time(f) \subseteq NTime(f)$ és $Space(f) \subseteq NSpace(f)$, hiszen a determinisztikus Turing-gépek a nemdeterminisztikus Turing-gépek részhalmaza, minden determinisztikus gép felfogható speciális nemdeterminisztikus gépként is. Érdemes megjegyezni az $Time(f) \subseteq Space(f)$ összefüggést is, amely egyszerűen abból következik, hogy f(n) idő alatt az algoritmus biztosan nem tud k*f(n)-nél több tárat felhasználni, hiszen minden lépésben minden szalagon legfeljebb egy cellát mozdulhat el. Ezzel a tármennyiséggel definíció szerint beleesik a Space(f) halmazba. Vegyük még észre, hogy mivel a definíciókban O(f) szerepel, $Time(a_0 * n_1^k + ... + a_m * n_m^k) = Time(n^k)$ és $Space(a0 * n^k 1 + ... + am * n^k m) = Space(n^k)$, azaz polinomok esetében nem számítanak az együtthatók, és a legnagyobb kivételével a kitevők sem. Ha az n kellően nagy, a polinom növekedési ütemét a legnagyobb kitevő fogja dominánsan meghatározni, az együtthatók és a kisebb kitevők pedig elhanyagolhatóvá válnak.

Végre pontosan definiálhatjuk a P illetve NP osztályokat.

Definíció:

$$P = \bigcup_{k=0}^{\infty} Time(n^k)$$

$$NP = \bigcup_{k=0}^{\infty} NTime(n^k)$$

Azaz legyen P a determinisztikus Turing-gépeken a bemenet hosszához képest polinomiális idő alatt felismerhető nyelvek halmaza, NP pedig a nemdeterminisztikus Turing-gépeken a bemenet hosszához képest polinomiális idő alatt felismerhető nyelvek halmaza. Mivel

minden determinisztikus gép tulajdonképpen egy speciális nemdeterminisztikus gép, azonnal következik, hogy $P \subseteq NP$.

Hasonló módon definiálhatók az L és NL osztályok, melyek a logaritmikus tárbonyolultságú osztályokat jelölik:

L=Space(log n)
NL=NSpace(log n)

Valamint az EXP és NEXP osztályok, melyek az exponenciális idejű problémákat tartalmazzák:

$$EXP = \bigcup_{k=0}^{\infty} Time(2^{n^k})$$

$$NEXP = \bigcup_{k=0}^{\infty} NTime(2^{n^k})$$

A probléma első megfogalmazása tehát így hangzik: *a polinomiális idő alatt felismert nyelvek tekintetében különbözik-e egymástól a determinisztikus és nemdeterminisztikus Turing-gépek esete?*

Először is nézzük meg, miért pont az időbonyolultság az, ami ennyire fontos számunkra, és nem a tárbonyolultság? Az algoritmusok tekintetében az idővel jóval "bőkezűbben" szoktunk bánni, mint a tárral. A nagy tárigényt a gyakorlatban nehezebb teljesíteni, mint a nagy időigényt. Természetesen definiálhatunk a P és NP osztályoknak analóg, polinomiális tárigényű osztályokat:

$$PSPACE = \bigcup_{k=0}^{\infty} Space(n^k)$$
$$NPSPACE = \bigcup_{k=0}^{\infty} NSpace(n^k)$$

Az rögtön nyilvánvaló, hogy $P \subseteq PSPACE$, a $Time(f) \subseteq Space(f)$ állítás egyenes következménye. Hasonlóképpen, $NP \subseteq NPSPACE$.

A kérdés tehát a determinizmus és a nemdeterminizmus közötti különbségről szól. Lássuk be most azt, hogy ez a különbség a tárigény esetében nem számottevő.

Tétel: Savitch-tétel

Ha f(n) jól számolható függvény, és $f(n) \geq log\ n$, akkor $NSpace(f) \subseteq Space(f^2)$.

(Egy függvényt jól számolhatónak nevezünk, ha létezik olyan Turing-gép, amely f(n)-et O(f(n)) idő alatt kiszámítja.)

Bizonyítás:

Legyen T egy olyan egy szalagos nemdeterminisztikus Turing-gép, amelynek tárbonyolultsága f(n). Rögzítsük a bemenetek n hosszát. A gép helyzetén értsük a (g,p,h) hármast, ahol g a gép jelenlegi állapota, p az I/O fej pozíciója ($-f(n) \leq p \leq f(n)$), h pedig a szalag tartalma. Elegendő azt a 2f(n) nagyságú darabját vizsgálni a szalagnak, amelyet a p "megcímez", az f(n) tárbonyolultságú számítás során a gép sosem fogja ezt a területet elhagyni. A helyzetek száma $|Q| * 2f(n) * |\Sigma|^2 f(n) \in O(f(n)|\Sigma|^2 f(n))$, ahol Q az állapotok halmaza, Σ pedig a szalagábécé. Minden helyzet kódolható egy O(f(n)) hosszúságú szóval. Készítsük el azt a G irányított gráfot, amelynek csúcsai a helyzetek, és az u csúcsból a v csúcsba akkor vezet él, ha az u helyzetből a v helyzetbe a gép egy legális lépése vezet. Vegyünk fel még egy v_0 csúcsot, amelybe vezessen él minden olyan csúcsból, amely olyan helyzetet kódol, amiben a gép elfogadó állapotban áll meg. Legyen u_x az x bemenetnek megfelelő induló helyzet. Az x szó akkor és csak akkor van L(T)-ben, ha a G gráfban az u_x csúcsból irányított út vezet a v_0 csúcsba.

Mivel a G gráf igen nagy, tekintsük úgy, hogy egy "orákulummal" van megadva. Ettől az orákulumtól egy Turing-gép egy lépésben meg tudja kérdezni, hogy két csúcs között

létezik-e irányított él az elsőből a másodikba. Tegyük föl, hogy orákulummal adott a G irányított gráf a p hosszú szavak halmazán. Ekkor létezik olyan Turing-gép, amely adott u, v csúcsokhoz és q természetes számhoz legfeljebb qp+q*log q felhasználásával eldönti, hogy G-ben vezet-e u-ból v-be legfeljebb 2^q hosszú irányított út. Bizonyítsuk ezt be q szerinti teljes indukcióval. A $q = 0$ esetben a megoldás triviális, csak az orákulumot kell megkérdezni u és v csúcsokra. Legyen $q \geq 1$. Készítsünk egy Turing-gépet, amely a következő módon működik: másolja át v-t és q-t egy másik szalagjára, v helyére írjon egy w szót, q-t csökkentse eggyel, és rekurzív módon döntse el, hogy vezet-e 2^{q-1} hosszú út u-ból w-be. Ha nem, akkor próbálkozzon új w-vel. Ha igen, akkor döntse el, hogy vezet-e 2^{q-1} hosszú út w-ből v-be. Ha nem, írjon fel egy új w-t és kezdje újra a folyamatot. Ha igen, akkor beláttuk, hogy vezet legfeljebb 2^q hosszú út u-ból v-be. Ha minden w csődöt mondott, akkor nem létezik ilyen út. A megfelelő w szavakat könnyen lehet például lexikografikus sorrendben generálni, ehhez nem kell külön tár. Az algoritmus két rekurzívan működő része használhatja ugyanazt a tárt, így ez csak $(q-1)p + (q-1)log\ (q-1) \leq (q-1)p + (q-1)log\ q$ tár. Ehhez jön még a p és q "kimentéséhez" szükséges p+log q tár. Tehát a gép képes eldönteni $qp + q * log\ q$ tárral, hogy létezik-e megfelelő hosszú út u és v között.

Vegyük még észre hogy az, hogy a G gráf két csúcsa között vezet-e él, tár igénybevétele nélkül könnyen eldönthető. Tehát a fenti eljárás alkalmazható $p, q \in O(f(n))$ értékekkel. Ilyen nagyságú értékekkel meghívva az algoritmus a teljes G gráfot képes lesz átvizsgálni. Tehát az algoritmus $pq + q * log\ q \in O(f^2(n))$ tárral eldönti, hogy u_x-ből v_0-be létezik-e irányított út, azaz hogy x eleme-e L(T)-nek. Ezzel az állítást beláttuk.

Savitch tételének egyenes következménye, hogy **PSPACE=NPSPACE**. Tárbonyolultság tekintetében tehát semmit sem nyerünk a nemdeterminizmussal, azaz nem leszünk képesek több nyelvet felismerni, ha determinisztikus gépek helyett nemdeterminisztikusakat használunk. Sajnos mint azt látni fogjuk, az időbonyolultság tekintetében a helyzet közel sem ilyen egyszerű.

Miért is fontos számunkra pont a polinomiális időbonyolultság? Erre a kérdésre adott válasz szorosan összefügg azzal, hogy mit is tekintünk "értelmesen kezelhető" problémának, mi az ami megkülönbözteti a "könnyű" feladatokat a "kezelhetetlenül nehéz" feladatoktól.

Érdemes megjegyezni, hogy a bonyolultságelmélet fejlődésével a "jól kezelhető probléma" fogalma egyre csak szűkült. Eleinte minden probléma amely nem eldönthetetlen ilyennek minősült. Később ez leszűkült a rekurzív nyelvek osztályára, hiszen egy ilyen nyelvet eldöntő algoritmus biztos nem fut végtelen ideig, biztosan választ ad bármilyen feltett kérdésre. Megint később a véges magasságú exponenciális tornyokkal kifejezhető időbonyolultság lett az, amit elfogadhatónak véltek.

Végül a figyelem középpontjába a P bonyolultsági osztály került. A gondolat, miszerint azt az algoritmust tekintjük használhatónak gyakorlati szempontból, amelynek időbonyolultága polinomiális, *Cobham-tézis* néven ismert. Emellett több gyakorlati érv is szól. Mindenekelőtt minden p(n) polinomiális függvény nyilvánvalóan lassabban nő minden e(n) exponenciális függvénynél, ha n elég nagy, ezáltal a függvény hosszabb bemeneteken is használható marad. A legtöbb gyakorlati jelentőssséggel bíró polinomiális algoritmusnak elég kicsi a bonyolultság függvényben szereplő kitevője ahhoz, hogy a számunkra érdekes bemenet hosszakon a futási idő kezelhető maradjon. Ez az exponenciális esetben nem mondható el. A P osztály ezen kívül robosztus sok átalakítással szemben. Az egyszalagos Turing-gép $f^2(n)$ idő alatt képes szimulálni egy többszalagos, f(n) időbonyolultságú gép működését, így a szalagok száma nem befolyásolja azt, hogy a felismert nyelv P-ben van-e. Az osztály akkor sem változik, ha a Turing-gép helyett egy másik gyakran használt számítási modellre, a RAM-gépre térünk át (amelynek számítási ereje bizonyítottan ekvivalens a Turing-géppel, fő különbsége a Turing-géptől a közvetlenül elérhető memória, amely által a modell közelebb áll a valós számítógépek működéséhez). Ezek a tulajdonságok hatékonyan vizsgálhatóvá teszik az osztályt. Az osztály sok természetesen felmerülő, nagy gyakorlati jelentőségű problémát tartalmaz (ezekre később példákat

is látni fogunk), így viszonylag jól lefedi az "értelmesen kezelhető" probléma fogalmától elvártakat. Ezen kívül a P vs. NP probléma több más felmerült problémával is ekvivalens, azaz igaz vagy hamis volta eldöntené a kapcsolódó problémát is és viszont (a probléma különböző egyenértékű megfogalmazásairól a későbbiekben még szó esik).

Mindezek ellenére a Cobham-tézis ellen is szólnak érvek. Egyfelől vannak nagyon is jelentős problémák, melyekről bizonyított (vagy erősen sejtett), hogy megoldásuk kívül esik a polinomiális időn. Ilyen például az utazó ügynök probléma, amely során egy gráf minden éléhez egy költséget rendelünk és keressük a minimális költségű Hamilton-kört. Másfelől egyes exponenciális időt igénylő algoritmusok időigényének növekedése nem jelentősen nagyobb a polinomiális algoritmusokénál, így "használhatatlannak" minősítése gyakorlati szempontból félrevezető. Hasonlóképpen egyes polinomiális algoritmusok bonyolultságában a polinom kitevője bizony olyan nagy, hogy az algoritmus használhatatlan a számunkra érdekes bemenethosszakon. Sőt, bizonyított, hogy az időhierarchia semmilyen magasságban nem omlik össze, vagyis tetszőleges k kitevőre létezni fog olyan nyelv, amely eleme P-nek, de nem eleme $Time(n^k)$-nak. Ezt fogalmazza meg az *Időhierarchia tétel.*

Tétel: Ha f(n) teljesen időkonstruálható, és g(n)(log g(n))=o(f(n)), akkor van olyan nyelv Time(f)-ben, amely nem eleme Time(g)-nek.

(Egy függvényt teljesen időkonstruálhatónak nevezünk, ha létezik olyan Turing-gép, amely minden n hosszú bemeneten pontosan f(n) lépést végez. Ez minden legalább lineáris polinomiális függvényre teljesül.)

Bizonyítás: Konstruáljunk olyan M determinisztikus Turing-gépet, amelyre $L(M) \in Time(f)$ és $L(M) \notin Time(g)$. M a következőképpen működjön: a w input szót kezelje úgy, mint az M' Turing-gép kódolását, majd szimulálja az M' működését w-n. Itt egy nehézségbe ütközünk: M szalagjainak száma rögzített, de a szimulálandó M' szalagjainak száma tetszőleges lehet. Tudjuk viszont azt, hogy egy kétszalagos Turing-gép akárhány

szalagos gép működését képes szimulálni c*g(n)(log g(n)) idő alatt, ahol g(n) a szimulálandó gép időbonyolultsága és c a szimulálandó géptől függő konstans.

Hogy biztosítsuk, hogy M' szimulációja nem lépi túl az f(n) időkorlátot, M ezzel párhuzamosan szimulálja egy olyan gép működését, amely pontosan f(n) lépést tesz n hosszú bemeneteken. (ilyen biztosan létezik, hiszen f(n) teljesen időkonstruálható). Pontosan f(n) lépés után M megáll. M csak akkor fogadja el w-t, ha M' szimulációja már befejeződött és az elvetette w-t. M' kódolását megadhatjuk úgy, hogy a hossza tetszőlegesen nagy legyen, tehát biztosan létezik olyan hosszú w kódolása, hogy $c * g(|w|)(log\ g(|w|)) \leq f(|w|)$ teljesüljön, hiszen $g(|w|)(log\ g(|w|)) = o(f(|w|))$. Ekkor a szimuláció be fog fejeződni időben. Ilyenkor w akkor és csak akkor lesz eleme L(M)-nek, ha nem eleme L(M')-nek. Tehát $L(M) \neq L(M')$ minden g(n) időbonyolultságú M'-re. Következésképpen L(M) eleme $Time(f) \backslash Time(g)$-nek. Ezzel az állítást beláttuk.

Az Időhierarchia tétel bizonyos szempontból ellenérvként is felfogható a Cobham-tézissel szemben, hiszen kimondja, hogy bizony létezik olyan nyelv, amelynek felismeréséhez például n^{1000} idő szükséges. Nem tehetünk tehát egyenlőséget a P osztály és a "jól kezelhető probléma" fogalma közé, viszont eléggé közel áll hozzá, hasznos tulajdonságokkal bír és kellően intuitív ahhoz, hogy az algoritmusok használhatóságának mércéje legyen.

Mielőtt továbbhaladnánk, lássunk még egy tételt, amely fontos a nemdeterminisztikus számítások jelentésének megértésében, és amely segít a problémát egy más, bizonyos szempontból sokkal elegánsabb módon megfogalmazni.

<u>Tétel:</u> Tanú tétel

Az L rekurzív nyelv akkor és csak akkor eleme NP-nek, ha létezik olyan L' P-beli nyelv, hogy $\forall w \in L : \exists v, hogy |v| \leq c|w|^k$ valamilyen L-től függő c,k>0-ra, és $w\#v \in L'$. Azaz, ha egy nyelv NP-beli, akkor létezik hozzá egy olyan rövid (a szóhoz képest polinomiálisan

hosszú) "tanú", amellyel kiegészítve a nyelv P-beli. A "tanú" gyakorlatilag azt akarja bebizonyítani a bírónak (az algoritmusnak), hogy a szó eleme a nyelvnek. Ha a szó nem eleme a nyelvnek, akkor ez a bizonyíték nem létezik.

Bizonyítás:

Tegyük fel, hogy L eleme NP-nek. Ekkor definíció szerint létezik olyan T' nemdeterminisztikus Turing-gép, amely pontosan L-et ismeri fel és az időbonyolultsága legfeljebb n^k valamilyen k-ra. Ha a w szó eleme L-nek, akkor létezik olyan számítása T'-nek, ami w-t legfeljebb n^k lépésben elfogadja. A w-hez tartozó v tanú $i_0, ..., i_t$ $t \leq n^k$, ahol i_j azt mutatja meg, hogy a legrövidebb elfogadó számítás j-edik lépésében a nemdeterminisztikus számítás a továbbhaladási lehetőségei közül melyiket választotta. (a lehetőségek maximális száma a T' géptől függő valamilyen d konstans). Ennek hossza, nyilvánvalóan polinomiális, hiszen a polinomiális hosszúságú számítás minden lépéséhez egy elemet rendel. Most egy olyan determinisztikus T gépet kell alkotnunk, amely szimulálja T'-t polinomiális időben. Legyen ez a gép eggyel több szalagos, mint T', a működése során erre a szalagra írja át a szó tanúját. Az átmenetfüggvénye nézzen ki a következőképpen:

$\Phi(q, a_1, ..., a_k, b) = (q', c_1, ..., c_k, b, m_1, ..., m_k, jobbra)$

Ahol b legyen a tanúból beolvasott betű, $(q', c_1, ..., c_k, m_1, ..., m_k)$ pedig legyen eleme $\Phi'(q, a_1, ..., a_k)$-nak (T' átmenetfüggvényének) , mégpedig pontosan a b-edik választási lehetőség. Azaz
$\Phi'(q, a_1, ..., a_k) = \{q_1 c_{11} c_{12} ... c_{1k} m_{11} m_{12} ... m_{1k}, ..., q_s c_{s1} c_{s2} ... c_{sk} m_{s1} m_{s2} ... m_{sk}\}$ $q' = q_b, c_i = c_{bi}, m_i = m_{bi}$ $i = 1, 2, ...k$

T működése determinisztikus, időbonyolultsága a tanú hosszával arányos, és csak akkor fogadja el $w\#v$-t ha T' elfogadja w-t. Tehát ha a nyelv NP-beli, akkor biztosan létezik

hozzá polinomiálisan hosszú tanú. Most fordítsuk meg az állítást: ha a nyelvhez létezik polinomiálisan hosszú tanú, akkor NP-beli.

Állítsunk elő egy nemdeterminisztikus gépet, amely a következő módon működik: először is a szalagján kezdje el előállítani az összes lehetséges tanút, amit írjon a bemeneti szava után: írjon nemdeterminisztikus módon betűket egymás után, és ennek nemdeterminisztikus úton vessen véget egy tetszőleges ponton, de legkésőbb amikor a szó hossza elérte a megfelelően nagy $|w|^k$ hosszt valamilyen a nyelvtől függő k-ra. Ez polinomiális időt vesz igénybe, hiszen v hossza polinomiális. A számítási fa egyes ágain tehát előáll az összes lehetséges szóba jöhető $w\#v$ szó. Ezután a gép szimulálja az előbb leírt determinisztikus gép működését a $w\#v$ szavakon. Mivel az összes lehetséges szót felírtuk, a fa egyik ágán a helyes $w\#v$ szó fog szerepelni, amit a determinisztikus gép elfogad, és következésképpen a nemdeterminisztikus gép elfogadja a w szót. Ha $w \notin L$, akkor helyes tanú nem létezik, így a számítási fa egyik ágán sem fog elfogadó számítás születni, a szót pedig a gép elveti. A működés első (v szavakat felíró) és második (T-t szimuláló) szakasza is polinomiális időben fut, tehát a kettő összege szintén polinomiális lesz. Ezzel az állítást beláttuk.

Látható, hogy az NP-beli problémához így előállított determinisztikus gép nem tesz mást, mint leköveti a tanúban megadott számítást, tehát a bíró egyszerűen ellenőrzi a tanú bizonyítékait. A gyakorlatban a tanúnak nem feltétlenül szükséges a számítási folyamatot kódolnia. Célszerűen használhatjuk tanúként az adott probléma megoldását, ilyenkor az algoritmus feladata pedig a megoldás helyességének ellenőrzése. Például legyen a feladat az, hogy egy számról belássuk, hogy nem prímszám. Ilyenkor a tanú lehet a szám egy valódi osztója. Ez a tanú nyilván kellően rövid, az algoritmus pedig egyszerűen elosztja az adott számot a tanúval, és megnézi, hogy a maradék nulla-e, ami könnyűszerrel elvégezhető polinomiális időben. Ha pedig a szám prímszám, akkor ilyen tanú egyértelműen nem létezik. Ezzel be is láttuk, hogy egy szám összetett szám voltának eldöntése NP-beli probléma.

Ha a P osztályt a "könnyen megoldható" problémák osztályának tekintettük, akkor ezen tétel értelmében az NP-t tekinthetjük a "könnyen ellenőrizhető" problémák osztályának. Ekkor a P=NP kérdés a következőképpen fogalmazható meg: *Bebizonyítani egy állítást nehezebb-e, mint leellenőrizni egy állítást?*

Áttekintettük a probléma kimondásához szükséges alapvető fogalmakat, és kimondtuk a problémát két alapvető alakjában. A későbbiekben még látni fogjuk a probléma más szemszögből történő megfogalmazásait, és a megoldására tett próbálkozások eszközeit, az ezekhez szükséges definíciókat pedig velük párhuzamosan fogjuk megadni.

Most, hogy már ismerjük a problémát magát, tekintsünk ki a probléma történelmére.

3. A P vs. NP probléma történelme

A problémák megoldásának triviális módja az összes elképzelhető lehetőség átvizsgálása. A legtöbb természetes úton felmerülő probléma megoldható ilyen módon, de az is nyilvánvaló, hogy ha a számba veendő lehetőségek száma kellően nagy, akkor ez a megoldás nem praktikus. A "brute force" módszer ezen hiányosságát hamar felismerték, és egyes speciális esetekben sikeresen találtak megfelelően hatékony keresőalgoritmust. Az algoritmusok területének egy korai eredménye volt az a felismerés, hogy a brute force keresés folyamata független a konkrét problémától.

Gödel egy Neumannhoz írt 1956-os levelében már említést tesz erről: arról elmélkedett, hogy mennyi időbe telne egy Turing-gép számára leelenőrizni, hogy egy logikai formulának létezik-e n hosszú levezetése. Afelől is érdeklődik, hogy általános esetben mennyire lehetséges javítani a keresési folyamat teljesítményén a brute forcehoz képest. Neumann válasza nem ismert, de munkássága alapján valószínűleg nem volt ismeretlen előtte a probléma, néhány évvel korábban megjelent cikkében a brute force elkerülésének módjáról ír speciális esetekben. Az ötvenes években a kutatók ezen az úton haladtak tovább:

algoritmusokat alkottak az egyes problémák hatékony megoldásához, de egyáltalán nem vetették el a brute force használhatóságát, sőt, bizonyos helyzetekben elkerülhetetlennek tartották.

A hatvanas években nagyszámú cikk született (pl. [1], [2], [3]), amely az algoritmusok bonyolultságát a szükséges lépések számaként értelmezte a bemenet hosszának függvényében. Ekkoriban született meg és kezdett terjedni Cobham felfogása, ami a P osztályt a könnyen megoldható problémák osztályaként azonosította. Az NP osztály jelentősége is ekkor kezdett növekedni, ahogy elkezdték felismerni, hogy milyen nagy számban találhatóak benne fontos gyakorlati problémák. Ugyanekkor született a nemdeterminisztikus számításokhoz kapcsolódó tanú-bíró szemlélet (bár Edmonds eredeti 1965-ös megfogalmazásában az NP metszet coNP osztályt írja le). A hetvenes évek elején bizonyították be az NP-teljes problémák létezését (ezek olyan NP-beli problémák, amelyekre minden NP-beli probléma "visszavezethető", [4], [5]), és Karp ekkor mutatta be huszonegy különböző problémáról, hogy NP-teljesek. [6] (Köztük közismert és igen fontos problémákról is, mint a Boolean formulák kielégíthetősége és olyan első ránézésre meglepő feladatokról is, mint a Hátizsák probléma, ahol egy adott teherbírású hátizsákba pakolunk adott súlyú és értékű tárgyakat, a feladat pedig a bepakolt érték maximalizálása.) Cook volt az első, aki explicit módon fogalmazta meg a P vs. NP kérdést egy 1971-es cikkében.

Ekkortájt bizonyították be a diagonizálás módszerével a tetszőlegesen sok időt igénylő problémák létezését. Ezt a módszert láttuk az Időhierarchia tétel bizonyításában, amely egy olyan nyelvet épít fel, ami definíció szerint különbözik minden nyelvtől, aminek alacsonyabb a bonyolultsága. Ez a nyelv eléggé mesterséges, gyakorlati jelentősége nincs a tétel bizonyításán kívül. A módszer azonban használható például annak bizonyítására is, hogy egy konkrét probléma kívül esik a P osztályon. Például a diagonizálás segítségével talán bizonyítható lenne, hogy egy NP-beli probléma determinisztikus megoldásához legalább exponenciális idő szükséges. Elképzelhetnénk egy polinomiális nemdeterminisztikus gépet, amely a számítási ágain szimulálja az összes polinomiális determinisztikus gép működését, majd úgy építi fel az elfogadott nyelvet, hogy az biztosan különbözzön ezektől. A gon-

dolatmenet hibás, ugyanis a nemdeterminisztikus gép nyilván nem szimulálhat olyan determinisztikus gépet, aminek az időbonyolultsága nagyobb a sajátjánál, márpedig ilyen akármilyen magas polinomra biztosan létezik. Ehelyett a naiv megoldás helyett talán létezik olyan ravasz nemdeterminisztikus algoritmus, amely ezt a diagonizálást képes lenne véghezvinni. Ha létezik is ilyen, az nem konstruálható meg egyszerű módon, erről az orákulumok viselkedése biztosít minket.

Hasznos eszköz az osztályok vizsgálatára az orákulum. Az orákulum egy nyelv, gyakorlatilag "ingyen tudás", amivel ellátjuk a Turing-gépet. A gép egy lépésben képes megkérdezni az orákulumától, hogy egy bizonyos szó eleme-e az orákulum nyelvének. Minden bonyolultsági osztályhoz alkotható egy hozzá tartozó orákulumos osztály, amely azon nyelveket tartalmazza, amelyeket az adott orákulummal ellátott, az eredeti osztálynak megfelelő gépek képesek felismerni. A hetvenes években bebizonyították, hogy létezik olyan orákulum, amellyel ellátva P=NP, és olyan is, amellyel ellátva $P \neq NP$ [7]. Azok az eredmények, amelyek lépésről lépésre történő szimulációt használnak, mint a diagonizálás módszere, orákulummal ellátva is változatlanok lesznek, hiszen az ilyen szimulációban az orákulumot pontosan arra fogják használni, mint a gép, amelyet éppen szimulálnak. Tehát egy olyan algoritmus, amely lépésről lépésre történő szimulációt használ arra, hogy bebizonyítja P és NP egyenlő vagy nem egyenlő voltát, akármilyen orákulum mellett működőképes kell, hogy legyen. Ez azonban ellentmond az előbb leírt eredménynek. Ilyen szimulációt használó módszer biztos nem lesz képes eldönteni a kérdést, azonban létezik másféle, "indirekt" szimuláció is, amely nem minden orákulum esetében marad változatlan ([8],[9]). Elméletileg tehát egy ilyen szimulációt használó algoritmus képes lehet eldönteni P és NP helyzetét.

Gödel munkássága nyomán ismert, hogy akármilyen axiómarendszert állítunk is fel, mindig létezni fog olyan kérdés, amely megválaszolhatatlan lesz az axiómák segítségével. Felmerült a kutatókban, hogy talán a P vs. NP kérdés is ilyen, a jelenlegi axiómáktól független kérdés. Ez azt jelentené, hogy ha P nem egyenlő NP-vel, akkor a P=NP bizonyításának nemlétét sohasem tudnánk belátni, és hogy ha P egyenlő NP-vel, akkor a

P=NP állítás igazát nem tudnánk soha bebizonyítani. Az orákulumok viselkedése önmagában is egyfajta függetlenséget sugall. Születtek eredmények, amelyek bizonyítják a kérdés függetlenségét néhány jóval gyengébb, mesterséges axiómarendszertól ([10],[11]), és olyanok is, amelyek orákulumokat adnak meg, amivel ellátva a kérdés független néhány erősebb axiómarendszertől [12]. A hetvenes években a kérdés matematikai logikai szemlélete meglehetősen elterjedt volt, ugyanis a bonyolultsági osztályok egy teljesen más szemszögből történő definiálását tette lehetővé. Több olyan eredmény is napvilágot látott, amely bizonyos logikai formulák halmazával azonosítja az ismerős bonyolultsági osztályokat. ([13],[14])

Ha egy probléma bonyolultságának alsó határát akarjuk meghatározni, abba a nehézségbe ütközünk, hogy az algoritmusok egyszerűen túl sok mindenre képesek. Egy lehetőség, amit ilyenkor tehetünk, az az algoritmusok képességeinek valamilyen módon történő korlátozása. Ezt tehetjük egyfelől úgy, hogy meghagyjuk a számítási modell általánosságát, de egyéb módon korlátozzuk le úgy, számunkra érdekes alsó bonyolultsági határokat kaphassunk, másfelől lekorlátozhatjuk az algoritmus mozgásterét úgy, hogy csak az adott problémához szükséges lépésekre legyen képes. Ez utóbbira példa az a módszer, amivel egy logikai formuláról azt akarjuk belátni, hogy kielégíthetetlen. Ha a formulának vannak olyan tagjai, amelyek $(x \wedge \alpha)$ és $(\neg x \wedge \beta)$ alakúak, akkor bevezethetünk egy új $(\alpha \wedge \beta)$ tagot. A rezolúció akkor ér véget, amikor az üres tagot adjuk a formulához, és minden kielégíthetetlen formula levezethető ilyen módon. Ha minden ilyen levezetésnek polinomiális lenne a hossza, abból az NP=coNP állítás következne, ami maga után vonná, hogy P=NP.

A problémák bonyolultságának meghatározásának egy másik hasznos eszköze a Boolean hálózatok. A hálózatok nagysága és a hozzájuk kapcsolódó Turing-gép időbonyolultsága között szoros összefüggés van [15]. Egyszerű felépítésük és könnyen manipulálható voltuk miatt jól kutathatóak, több kutató is a hálózatokat a legreálisabb esélyesnek a P vs. NP kérdés eldöntését illetően. A hálózatok logikati kapukból állnak, amiket valamilyen bázisból választhatunk ki (például ÉS, VAGY és NEM), a kapuk pedig logikai értékeket

állítanak elő azokból a logikai értékekből, amiket bemenetként kapnak. A folyamat az inputként megadott értékekből indul, ezt olvassák be azok a kapuk, amelyek közvetlen összeköttetésben állnak velük, az így kapott értékekkel dolgoznak azok a kapuk, amelyek ezekkel állnak kapcsolatban, és így tovább, amíg egy olyan kapuhoz nem érünk, aminek nincs rákövetkezője. Ezen kapu értékét tekintjük a kimenetnek, azaz hogy elfogadta-e az inputot vagy sem. Egy problémához nem egy hálózatot, hanem egy hálózat-családot rendelünk, minden inputszó hosszhoz egy külön hálózatot, amely a problémát az adott hosszú bemenetekre megoldja. Tudjuk, hogy minden P-beli problémához létezik polinomiális méretű hálózat-család. Tehát ahhoz, hogy bebizonyítsuk, hogy egy probléma nincs P-ben elég megmutatni, hogy szuperpolinomiális méretű hálózatokra van szükség a megoldásához. Egy egyszerű számolás alapján megmutathatjuk, hogy a legtöbb Boole függvény kiszámításához exponenciális méretű hálózatok szükségesek. Ennek ellenére, jelenleg a legjobb alsó korlát, amit egy számunkra érdekes problémára sikerült megadni is csak lineáris [16], tehát ez a közvetlen hozzáállás nem tűnik célravezetőnek. Természetesen itt is lehetőség van olyan megszorításokat bevezetni, amelyekkel ellátva a hálózatok eléggé lekorlátozódnak ahhoz, hogy számunkra érdekes eredményekkel szolgáljanak. A bázis megfelelő megválasztásával vagy a hálózatok mélységének korlátozásával sikerült jóval nagyobb alsó határokat sikerült megkapni. Például egy olyan egyszerű feladat, mint hogy egy bináris szó egyes bitjeinek száma páros-e vagy sem, szuperpolinomiális nagyságú hálózatokat igényel ha a hálózatok mélysége fix [17]. A többség függvény, ami egy bináris szóról megadja, hogy egyes vagy nullás bitből tartalmaz-e többet, exponenciális méretű hálózatokkal adható meg, ha a lehetséges bázis ÉS, VAGY, NEM, és PÁROS (az előbb említett párosság függvény) kapukból áll [18].

Érdekesek még a monoton hálózatok, olyan hálózatok, amelyek nem tartalmaznak NEM kaput. A monoton hálózatok pontosan a monoton függvényeket képesek kiszámítani (az olyan Boole függvényeket, amelyekre $x \leq y$ esetén $f(x) \leq f(y)$ áll fenn). Ilyen hálózatok segítségével a klikk probléma (egy gráfról megmondani, hogy található-e benne olyan részgráf, amely teljes gráf és a csúcsainak száma egy megadott k, nem mellesleg

egy NP-teljes probléma) exponenciális méreteket igényel [19]. Ez az eredmény, mint a terület több más eredménye is, bizonyítás gyanánt egyfajta közelítési módszert használ. A hálózat működésének teljes elemzése helyett az egyes kapuk működését változtatjuk meg úgy, hogy a kimenetük változása a végeredményt csak kis mértékben változtassa meg. Ezután belátjuk, hogy a végeredmény nagyon is nagy mértékben különbözik az eredetitől, tehát a hálózatban nagyon sok kapu található. Felmerült a gondolat, hogy ez a közelítő eljárás használható lenne általános hálózatok esetén is, nem csak monoton vagy korlátos mélységű hálózatok esetén. A módszer általánosításának két lehetőségét vázolták fel. Az egyik esetében a közelítő függvények, amelyeket használunk konstans módon választjuk ki és minden hálózatnál ugyanúgy alkalmazzuk. Ez a módszer túl gyenge ahhoz, hogy számunkra érdekes alsó korlátokat kaphassunk a segítségével. Ha a közelítéshez használt függvények halmazát függővé tesszük az adott hálózattól, akkor már kaphatunk érdekes eredményeket, de ennek a változatnak az alkalmazása a gyakorlatban jóval nehezebb. Ha az általános hálózatokat át lehetne alakítani monoton hálózatokká úgy, hogy a méretük polinomiálisan növekszik, akkor a monoton hálózatokon meghatározott alsó határokat használhatóak lennének általános bonyolultsági osztályok jellemzésekor is. Sajnos ez nem teljesül, a fent elmített közelítési módszerrel sikerült belátni azt, hogy bizonyos P-beli problémák (amelyekhez tehát biztosan létezik polinomiális méretű hálózat) szuperpolinomiális méretű monoton hálózatokat igényelnek [20]. Léteznek azonban olyan függvények, amelyekhez tartozó hálózatok átalakíthatók monoton hálózatokká polinomiális növekedés árán. Egy hálózat, amely valamilyen k-ra nulla értéket ad, ha a bementi szóban az egyesek száma kevesebb, mint k és egyet különben, átalakítható monoton hálózattá ilyen módon [21]. Ha erre a problémára sikerülne szuperpolinomiális alsó korlátot bizonyítani, az azt jelentené, hogy $P \neq NP$.

Érdekesek még számunkra az olyan polinomiális méretű hálózatok, amelyek a szokásos bázist használják, de az egyes kapuk bemeneteinek számát kettőre korlátozzuk, a hálózat mélységére tett korlát pedig $O(log^i\ n)$. Ezt az osztályt NC^i-nek nevezzük, az összes nemnegatív i-re történő egyesítését pedig NC-nek ("Nick's class", Nick Pippenger után).

Az NC osztályt a párhuzamos algoritmusokkal jól kezelhető problémákkal szokás azonosítani. Az NC^1 osztályt a következőképpen lehet még jellemezni: legyen L egy olyan nyelv, amely eleme NC^1-nek. Képzeljünk el egy kommunikációs játékot két játékossal, ahol az egyik játékosnál egy olyan szó van, amely eleme L-nek, a másiknál pedig egy olyan, amely nem eleme. Ezután a két játékos egymással kommunikál, és egy olyan pozíciót keresnek, ahol a két szó értéke különbözik. Azon bitek minimális száma, amelyre ehhez szükségük van bármilyen n hosszú szavak esetén lesz az f(n) bonyolultsága a játéknak. Ez pontosan egyenlő a nyelv felismeréséhez szükséges hálózatok minimális mélységével. Ez a játék kapcsolatban áll még a monoton hálózatok bonyolultságával is. Ha kikötjük azt, hogy a talált különböző pozícióban az L-beli szó értéke egy kell hogy legyen, akkor a játék bonyolultsága pontosan a megfelelő monoton hálózatok minimális mélysége.

Könnyen elképzelhetünk olyan bonyolultsági osztályokat, amelyek a számunkra már ismerős osztályok monoton analógjai, például mP, vagy mNP osztályok. Tudjuk, hogy $mP \neq mNP$ [20], $mP \neq P \cap mono$ [20], ahol mono az összes monoton nyelv osztálya, valamint hogy $mNC^1 \neq mNL$ [22], $mNC^1 \neq mL$ [23], és $mNL \neq m - coNL$.

Áttekintettük a P vs. NP probléma történetének főbb pontjait, láttuk a probléma jelentőségének kialakulását, a probléma főbb megközelítéseit, és a megoldására tett kísérletek meghatározó gondolatait, módszereit. Nézzük most meg ezeket részletesebben. Elsőként ejtsünk szót azokról a problémákról, amelyeknek talán a legnagyobb a jelentősége a P vs. NP kérdés ügyében, azon problémákról, amelyek segítségével a legegyszerűbben lehetne eldönteni a két osztály viszonyát.

4. NP-teljes problémák

Mindenek előtt definiáljuk azt, mit jelent egy nyelvet visszavezetni egy másik nyelvre. Alapvetően egy L' nyelv visszavezetése L-re annyit tesz, hogy létezik egy olyan g függvény, amelyet egy olyan Turing-gép számít ki, amely minden bemeneten megáll, hogy minden x szóra teljesül, hogy x akkor és csak akkor eleme L'-nek, ha g(x) eleme L-nek. Ilyen módon, ha L rekurzív nyelv, akkor L' felismerhető olyan módon, hogy kiszámítjuk az x bemenethez tartozó g(x)-et, majd megnézzük, hogy $g(x) \in L$ teljesül-e. Ha a g függvényt megfelelően lekorlátozzuk úgy, hogy kiszámítása kellően egyszerű legyen, akkor a visszavezetés segítségével képesek vagyunk bizonyítani a nyelvek valamilyen számunkra érdekes bonyolultsági osztályba való tartozását (például a P, NP vagy PSPACE osztályba tartozást).

Azt mondjuk, hogy L' polinomiális időben visszavezethető L-re, ha létezik olyan polinomiális időbonyolultságú Turing-gép, amely az x inputra ad egy y outputot, és $y \in L$ akkor és csak akkor fog teljesülni, ha $x \in L'$. Ekkor teljesül, hogy ha L eleme NP-nek, akkor L' is eleme NP-nek és ha L eleme P-nek, akkor L' is eleme P-nek. Tegyük fel, hogy a függvény, amely visszavezeti L'-t L-re $p_1(n)$ időbonyolultságú, az L-t felismerő Turing-gép időbonyolultsága pedig $p_2(n)$, ahol p_1 és p_2 polinomiális függvények. Ekkor L'-t a következőképpen ismerhetjük fel: először az x bemenetet vezessük vissza az y szóra. Mivel a visszavezetés $p_1(n)$ ideig tart, $|y| \leq p_1(|x|)$, hiszen $p_1(|x|)$ lépés alatt ennél hosszabb szót nem kaphatunk. Ezután megvizsgáljuk, hogy y eleme-e L-nek $p_2(|y|) \leq p_2(p_1(n))$ idő alatt. Tehát $x \in L'$ eldönthető összesen $p_1(n) + p_2(|y|) \leq p_2(p_1(n))$ idő alatt, ami polinomiális n-ben.

Egy másik visszavezetés, amely érdekes számunkra a log-space visszavezetés. Ez abban különbözik a polinomiális idejű visszavezetéstől, hogy a visszavezetést végző Turing-gép csak logaritmikus tárat használhat. Az ilyen Turing-gépre kikötjük, hogy az input szalagján a bemeneti szót soha nem írhatja felül (azaz csak olvashatja), az output szalagon soha nem léphet balra (azaz amit egyszer leírt többet nem vonhatja visz-

sza), a számításokhoz használt tárterülete pedig legyen O(log n) méretű az n hosszú bemeneti szóhoz képest. Ekkor ha L' log-space visszavezethető L-re akkor ha L eleme P-nek akkor L' is eleme P-nek, ha L eleme $NSpace(log^k\ n)$-nek akkor L' is eleme $NSpace(log^k\ n)$-nek és ha L eleme $Space(log^k\ n)$-nek akkor L' is eleme $Space(log^k\ n)$-nek. Az első eset bizonyításához elég belátni, hogy a logaritmikus táron végzett visszavezetés legfeljebb polinomiális ideig tarthat. Az outputszalag tartalma nem befolyásolhatja a számítást, így a gép lehetséges konfigurációinak számára felső határt szab a gép állapotainak számának (a géptől függő konstans), a számítási szalag tartalmának lehetséges kombinációinak számának ($O(|\Sigma|^{O(log\ n)})$ nagyságrendű mennyiség, ahol $|\Sigma|$ a szalagábécé elemeinek száma. $|\Sigma|$ a géptől függő konstans,legyen c_1. Ekkor ez a mennyiség $O(c_1)^{O(log\ n))} = O(2^{(log\ c_1 * O(log\ n)))} = O(2^{O(log\ n)} = O(2^{(log^k\ n)}) = O(n^k)$ méretű.) és az input és számítási szalagok I/O fejeinek lehetséges pozíciónak számának (O(log n) nagyságrendű mennyiségek) szorzata. Ez a szorzat polinomiális méretű n-ben, és ha ennél több lépést tenne a gép, az azt jelentené, hogy legalább egy konfiguráció kétszer is szerepelt a számítás során, vagyis a gép egy végtelen ciklusba lépett, ami ellentmond annak a kikötésnek, hogy minden bemenetre outputot kell, hogy produkáljon. A második és harmadik eset belátása valamivel bonyolultabb. Legyen M_1 az a gép, amely a log-space visszavezetést végzi, és legyen M_2 a $log^k\ n$ tárbonyolultságú gép, amely felismeri L-et. Mint láttuk, M_1 outputja polinomiálisan hosszú a bemenethez képest, amit nem tudunk $log^k\ n$ táron tárolni. Tehát M_1 és M_2 nem szimulálható úgy, hogy M_1 outputját eltároljuk egy szalagon. Ehelyett az outputot szimbólumonként egyesével fogjuk átadni M_2-nek. Ez működik amíg M_2 csak jobbra halad az input szalagján, ha balra lép, akkor M_1 szimulációját újra kell indítani, mivel az output korábbi szimbólumai nem lettek "elmentve". Az L'-t elfogadó M_3 gépet a következő módon építjük fel. M_3 egyik szalagján tároljuk, hogy M_2 az input éppen hanyadik pozícióján áll. Ezen szám hossza log n valamilyen konstansszorosa, azaz O(log n) nagyságrendű. A többi szalagjain M_3 szimulálja M_1 és M_2 működését. Tegyük fel, hogy M_2 épp az i pozíción áll, és arrébb akar lépni balra vagy jobbra. M_3 szimulálja M_1 működését, amíg az a megfelelő számú (i-1 vagy i+1) output

szimbólumot elő nem állította. Ha M_2 balra akart lépni, akkor ehhez M_1 szimulációjának elölről kezdése szükséges. Ha előállt a szükséges szimbólum, M_2 szimulációja folytatódhat. Különleges helyzetekben, amikor M_2 balra akar lépni amikor $i = 1$, vagy M_2 jobbra akar lépni és M_1 megáll mielőtt a megfelelő számú output jelet előállította volna, a szimulált M_2 azt a jelzést kapja, hogy elérte az input bal vagy jobb szélét. M_3 pontosan akkor fogadja el az inputját, amikor M_2 elfogadja a szimulált inputját. M_3 tehát egy $log^k\ n$ tárat használó Turing-gép, amely felismeri L'-t.

Két log-space visszavezetés kompozíciója szintén log-space visszavezetés, és két polinomiális idejű visszavezetés kompozíciója szintén polinomiális idejű visszavezetés.

A visszavezetések ezen tulajdonságait használjuk bonyolultsági osztályok viszonyainak eldöntésére. A P vs. NP kérdés alapvetően arról szól, hogy NP több nyelvet tartalmaz-e, mint P, azaz hogy $NP \backslash P$ üres halmaz-e vagy sem. Intuitívan érezzük, hogy ha ez a halmaz nem üres, akkor a tagjai az NP osztály legnehezebb problémái. Ez a "megérzés" kivételesen helyes.

Tegyük fel egy L_0 nyelvről, hogy az NP-ben található összes nyelv visszavezethető rá valamilyen egyszerűen számítható függvénnyel. A visszavezetés típusától függően kimondhatunk bizonyos dolgokat L_0-ról. Ha log-space vagy polinomiális idejű visszavezetésről van szó, és L_0-ról belátnánk, hogy P-beli, akkor $P = NP$. Ha a visszavezetés log-space, és L_0 $Space(log^k\ n)$-beli, akkor az $NP = Space(log^k\ n)$ állítás is teljesülne.

Ha egy nyelvre igaz, hogy valamilyen bonyolultsági osztály összes nyelve visszavezethető rá, akkor az mondjuk, hogy ez a nyelv "nehéz" erre az osztályra nézve. Ha a nyelv maga is eleme az osztálynak, amely visszavezethető rá, akkor a nyelv "teljes" az osztályra nézve. Ha fontos az, hogy milyen típusú visszavezetést használunk, hozzátesszük, hogy például a polinomiális idejű visszavezetést használva teljes/nehéz a nyelv az adott osztályra nézve. Amikor először próbálunk egy osztályban teljes nyelvet találni, akkor meg kell adnunk egy visszavezetést, amivel minden az osztályba tartozó nyelv visszavezethető rá. Ezután ha egy másik nyelvre akarjuk bizonyítani a teljességet, elég ha megadunk egy

megfelelő visszavezetést, amely az előző teljes nyelvet vezeti vissza az új nyelvre.

Természetesen minket elsősorban az NP-teljes problémák fognak érdekelni. Ezekre a problémákra nem ismert polinomiális idejű determinisztikus algoritmus, és ha az egyikre sikerülne találni, akkor mindegyikre sikerülne találni. Ezért fontos egy problémáról belátni, hogy NP-teljes vagy sem. Ha igen, akkor valószínűleg egy nagyon nehéz problémával állunk szemben, amihez általánosan használható polinomiális algoritmust keresni valószínűleg időpazarlás, és érdemes más módszerekkel közelíteni hozzá. Az első probléma amiről belátjuk, hogy NP-teljes az lesz, amiről történelmileg is először sikerült ezt belátni, a Boolean kifejezések kielégíthetősége.

Satisfiability (SAT):

Egy Boolean kifejezésben találhatók változók, zárójelek és háromféle logikai operátor: az ÉS ($\wedge$), a VAGY ($\vee$) és a NEM ($\neg$). A változók értéke 0 vagy 1 (igaz vagy hamis), akárcsak a kifejezéseké. Az operátorok precedenciája legerősebbtől a leggyengébb felé: NEM, ÉS, VAGY. A változók önmagukban is legális kifejezések. Ha E_1 és E_2 kifejezések, akkor $E_1 \wedge E_2$, $E_1 \vee E_2$, $\neg E_1$ és (E_1) is azok. Egy kifejezés kielégíthető, ha a benne szereplő változóknak lehet úgy értéket adni, hogy végül minden változónak legyen értéke és a kifejezés értéke legyen 1. A SAT nyelv legyen a kielégíthető Boolean kifejezések halmaza.

A kifejezésben szereplő változókat jelöljük x_i-vel, i-t pedig írjuk binárisan. Ekkor a SAT nyelv ábécéje:
$\{\wedge, \vee, \neg, (,), x, 0, 1\}$
Egy n szimbólumból álló kifejezés hossza ilyen jelölések mellett legfeljebb $\lceil n * log_2\ n \rceil$, mivel a változókon kívül minden szimbólum egy karakter hosszú, egy n hosszú kifejezésben pedig legfeljebb $\lceil n/2 \rceil$ változó található, mivel minden változó kódolása legalább két karakter hosszú. Így a kifejezésben egy változó kódjának hossza legfeljebb $\lceil 1 + log_2\ n \rceil$ hosszú. Ezentúl amikor n hosszú kifejezésekről beszélünk, nem fogjuk megkülönböztetni a kifejezés hosszát és a kifejezést kódoló szó hosszát, mivel az eredmény nem fog függni at-

tól, hogy a bemenet hosszát n-nek vagy $n*log\ n$-nek tekintjük. ($log(n*log\ n) \leq 2*log\ n$, vagyis a log-space visszavezetés, amit használni fogunk ugyanúgy helyesen fogja majd kezelni)

Először lássuk be, hogy a SAT probléma NP-beli. Ehhez elég elképzelni egy nemdeterminisztikus gépet, amely a kifejezésben szereplő változókhoz nemdeterminisztikusan hozzárendeli az értékeket az összes lehetséges kombinációban (tehát a számítási fa egyes ágain egy-egy változó kiértékelés fog elhelyezkedni), majd egyszerűen kiszámítja a kifejezés értékét, ami könnyen megtehető polinomiális időben. Ha volt olyan ága, amelyen helyes kiértékelés szerepelt, akkor a kifejezést el fogja fogadni, különben pedig elutasítja. Másképp megfogalmazva, a helyes változó kiértékelést használhatjuk mint tanút, aminek segítségével determinisztikusan leellenőrizhetjük a helyességét.

Ahhoz, hogy megmutassuk, hogy minden NP-beli nyelv visszavezethető SAT-ra, meg fogunk adni egy log-space visszavezetést, ami az x bemeneti szót átalakítja az E_x Boolean kifejezéssé, ami akkor és csak akkor lesz kielégíthető, ha x eleme a nyelvnek, amit éppen visszavezettünk a SAT-ra. Nézzük meg E_x felépítését.

Legyen $\#\beta_0\#\beta_1...\#\beta_{p(n)}$ az M nemdeterminisztikus, p(n) polinomiális időbonyolultságú Turing-gép egy számítása. β_i álljon pontosan p(n) szimbólumból, és legyen $\alpha_1 q \alpha_2$ alakú, ahol $\alpha_1, \alpha_2 \in \Sigma^*$ legyen a szalag tartalma (az általánosság elvesztése nélkül feltételezhetjük, hogy a gép egyszalagos), α_1 az I/O fejtől balra eső rész, α_2 az I/O fejtől jobbra eső rész, q pedig egy összetett szimbólum, amely tartalmazza a gép állapotát és az éppen beolvasott szimbólumot, valamint tartalmazza, hogy az adott lépésben a továbbhaladási lehetőségei közül a gép melyiket választja. Ha a számítás elfogadással véget a p(n). lépés előtt, akkor a fennmaradó β_i-k az utolsó elfogadó konfigurációt fogják ismételni, hogy pontosan p(n)+1 β_i tag legyen.

Minden x szimbólumra, ami előfordulhat a számításban és minden i értékre $0 \leq i < (p(n)+1)^2$, létrehozunk egy logikai változót, c_{ix}-et, ami megmondja, hogy a számítás i. szimbóluma x-e. A nulladik szimbólum legyen a #. Az ezekből a változókból álló E_x

kifejezést úgy építjük fel, hogy akkor és csak akkor lesz igaz értékű, ha a változókhoz rendelt értékek alapján egy helyes számítást tudunk előállítani. Az E_x a következőket fogja állítani:

1. Az igaz értékű c_{ix} változók egy szimbólumsorozatot kódolnak, azaz minden i-re pontosan egy igaz c_{ix} létezik.
2. A β_0 tag az M gép x inputtal ellátott kezdőkonfigurációját kódolja.
3. Az utolsó tagban a gép végállapotban van.
4. Minden tag olyan módon következik az előtte levőből, ahogy az az előtte levő tagban meg van adva.

Az E_x négy kifejezés ÉS-sel történő összekapcsolása lesz, a négy kifejezés mindegyike a fenti feltételek egyikét fogja állítani.

Az első kifejezés, amely szerint minden $0 \leq i \leq (p(n)+1)^2 - 1$ értékre pontosan egy c_{ix} igaz:

$$\bigvee_i [\bigwedge_x C_{iX} \vee \neg(\bigwedge_{X \neq Y}(C_{iX} \wedge C_{iY}))]$$

Minden i értékre a $\bigwedge_x C_{iX}$ meghatározza, hogy legalább egy c_{ix} igaz, a $\neg(\bigwedge_{X \neq Y}(C_{iX} \wedge C_{iY}))$ pedig azt, hogy legfeljebb egy c_{ix} igaz.

A második kifejezés, amely β_0-ról mond ki állításokat, legyen megint csak négy kifejezés ÉS-sel történő összekapcsolása, a négy részkifejezés pedig nézzen ki így: (a bemenet x legyen $a_1 a_2 ... a_n$ alakú)

1. $c_{0\#} \wedge c_{p(n)+1,\#}$. A nulladik és p(n)+1-edik szimbólum #.
2. $c_{1Y_1} \vee c_{2Y_2} \vee ... \vee c_{kY_k}$, ahol $Y_1, Y_2, ..., Y_k$ azok az összetett szimbólumok, amelyekben a a_1 szalagszimbólum, a q_0 kezdőállapot és az összes ezek alapján lehetséges legális lépési lehetőség van kódolva. Ez azt mondja ki, hogy β_0 első szimbóluma helyes.

3. $\bigwedge_{2\leq i\leq n} c_{ia_i}$. 2-től n-ig β_0 szimbólumai helyesek.

4. $\bigwedge_{n\leq i\leq p(n)} c_{iB}$. β_0 minden további szimbóluma az üres szimbólum.

A harmadik kifejezés szerint az utolsó β_i tagban végállapot van kódolva. Ezt így adhatjuk meg:

$$\bigvee_{p(n)(p(n)+1)<i<(p(n)+1)^2}\left(\bigvee_{X\in F} c_{iX}\right)$$

ahol F azon összetett szimbólumok halmaza, amelyekben végállapot van kódolva.

A negyedik kifejezés szerint minden β_i tag ($i \geq 1$) megkapható az előtte álló β_{i-1} tagból, mégpedig azon lépés által, amely β_{i-1} összetett szimbólumában kódolva van. Vegyük észre, hogy minden β_i-beli szimbólumot megkaphatunk a neki megfelelő és mellette levő pozíciókban található β_{i-1}-beli szimbólumok segítségével. A szimbólum változatlan, kivéve ha az adott pozícióban állt az I/O fej, vagy az I/O fej szomszédos pozíción volt és a szimbólum felé mozdult el. Vegyük még észre, hogy mivel egy összetett szimbólum csak egy lehetséges továbblépési lehetőséget kódol, ezért egy β_i tagban több legális összetett szimbólum is lehetséges, amelyek mind a nemdeterminisztikus számítás különböző lehetséges útvonalaihoz tartoznak. Arról se feledkezzünk meg, hogy ha a β_{i-1} tagban elfogadó állapot volt, akkor β_{i-1} és β_i teljesen megegyeznek. Alkossunk egy f(W,X,Y,Z) predikátumot, amely akkor és csak akkor igaz, ha a Z szimbólum legálisan szerepelhez valamilyen β_i tag j. pozíciójában, ha W, Y, és X a β_{i-1} tag j-1, j, és j+1 pozíciójú szimbólumai. W legyen # ha $j = 1$ és Y legyen # ha $j = p(n)$. Legyen még $f(W, \#, X, \#)$ igaz, hogy a β_i tagokban előforduló # szimbólumokat és a β_i tagokat elválasztó # szimbólumokat egységesen tudjuk kezelni. A negyedik részkifejezés legyen a következő:

$$\bigwedge_{p(n)<j<(p(n)+1)^2}\left(\bigvee_{W,X,Y,Z \text{ ahol } f(W,X,Y,Z)}\left(c_{j-p(n)-2,W} \wedge c_{j-p(n)-1,X} \wedge c_{j-p(n),Y} \wedge c_{jZ}\right)\right)$$

Ha adott M egy elfogadó számítása az x bemeneten, könnyű olyan változókiértékelést készíteni, amely kielégíti E_x-et. Legyen c_{iy} igaz akkor és csak akkor, ha a számítás i. szimbóluma y. Ez a négy részkifejezés biztosítja, hogyha valamilyen változókiértékelés

kielégíti E_x-et, akkor M-nek létezik x-en elfogadó számítása. Bár M nemdeterminisztikus, a β_i tagok úgy lettek definiálva, hogy az átmenetek β_i-ből β_{i+1}-be mindig legális lépései lesznek M-nek. Az E_x-et alkotó kifejezések $O(p^2(n))$ nagyságúak, vagyis egy logaritmikus tárral ellátott Turing-gép képes őket megkonstruálni. A gépnek még el kell tudnia számolni $(p(n)+1)^2$-ig. Mivel egy polinom logaritmusa valamilyen konstansszor log n, ez megtehető O(log n) tárral. Ezzel beláttuk, hogy minden NP-beli nyelv log-space visszavezethető SAT-ra. Tehát SAT NP-teljes.

Ha a SAT-hoz létezne determinisztikus polinomiális algoritmus, akkor a visszavezetéseknél imertetett módon minden NP-beli nyelv felismerhető lenne determinisztikusan polinomiális időben, azaz P=NP teljesülne. Ezért a P vs. NP kérdés egy gyakran használt megfogalmazása még a következő: *Létezik-e olyan algoritmus, amely determinisztikus módon polinomiális időben felismeri a SAT nyelvet?*

Az NP-teljességet az évek során rengeteg problémáról látták már be. Az, hogy ezek között a problémák között sok nagy fontosságú, természetes probléma is helyet kap, mutatja, hogy az NP-teljesség tulajdonsága meghatározó egy probléma szempontjából. Ráadásul ezek a problémák sok különböző területen találhatóak. Ilyen például a logika, ahol nemcsak a SAT, de a SAT bizonyos szűkített formái is NP-teljesek. Ilyen a 3-SAT: egy problémák k-SAT-nak nevezünk, ha a formula, aminek a kielégíthetőségét vizsgáljuk konjunktív normálformában van (azaz az egyes tagjai ÉS műveletekkel vannak összekötve, és minden tag változók vagy azok tagadásának VAGY művelettel való összekapcsolásából áll), és minden tagjában pontosan k darab változó szerepel. Nagyon sok NP-teljes probléma található a gráfelmélet területén. Ilyen a Hamilton-út problémája, ami azt kérdezi, hogy létezik-e egy gráfban olyan út, ami az összes csúcsát bejárja. A 3-színezés (ami arra kíváncsi, hogy egy gráf csúcsait ki lehet-e színezni három színnel úgy, hogy ne legyen két szomszédos egyező színű csúcs) szintén NP-teljes, akárcsak az utazóügynök-probléma eldöntési változata (ez azt kérdezi, hogy ha adott n város és a közöttük lévő távolságok, és adott egy K szám, vajon létezik-e olyan körút, amely az összes várost érinti és az

össztávolsága kisebb, mint K). Ezeknél a problémáknál az NP-teljesség bizonyítása során azt, hogy a probléma NP-beli, könnyű belátni. Csupán a tanú-tétel alapján belátjuk, hogy a megoldás könnyen ellenőrizhető. Például a Hamilton-út probléma során egy konkrét megadott útról gyorsan be tudjuk látni, hogy valóban minden csúcsot érint, a 3-színezésnél egy megadott színezés helyességéről gyorsan meg tudunk győződni. A bizonyítás nehezebbik része ezeknél a problémáknál egy olyan visszavezetés megadása, amely egy már bizonyítottan NP-teljes problémát vezet vissza rájuk. Ez azonban nem mindig van így, ellenpélda erre az egészértékű lineáris programozás problémája. Ez a probléma a következőt kérdezi: adott egy csak egészelemű $m \times n$-es A mátrix és egy n elemű b vektor (szintén egész értékekkel), létezik-e olyan x vektor, hogy x egész értékű és $Ax \geq b$? Erről a feladatról könnyű belátni, hogy NP-nehéz, de belátni azt, hogy maga is NP-beli (tehát következésképpen NP-teljes) már bonyolultabb.

Az NP-teljes problémák időigényére tett megállapítások akkor is fontosak lehetnek, ha nem polinomiálisak. Például ha egy ilyen problémáról belátnánk, hogy eleme $Time(n^{log\ n})$-nek, akkor tudnánk, hogy $NP \subseteq Time(n^{c*log\ n})$ valamilyen c konstansra. Általánosan, ha tudnánk, hogy az L NP-teljes nyelv eleme Time(T(n))-nek, akkor $NP \subseteq \bigcup_{c>0} DTIME(T(n^c))$.

Láthattuk, hogy az NP-teljes problémák vizsgálata egy egyszerű, természetes úton felmerülő útja a P vs. NP kérdés tisztázásának. Most nézzünk meg egy másik utat, amely egy érdekes segédeszközt vezet be a bonyolultság vizsgálatához: az orákulumokat.

5. Orákulumos Turing-gépek

Az orákulumok hasznos eszközök a nyelvek vizsgálatánál. Gyakorlatilag plusz tudást szolgáltatnak a gépnek, amiért nem kell "megdolgoznia". Definiáljuk most az orákulum fogalmát.

Legyen A egy nyelv. Az A orákulummal ellátott Turing-gép legyen egy olyan Turing-

gép, amelynek létezik három különleges állapota és egy kijelölt szalagja. Legyenek ezek $q_?$, q_{igen} és q_{nem}. A számítása során, amikor a Turing-gép a $q_?$ állapotba kerül, kérdést intéz az A orákulumhoz. Megkérdezi tőle, hogy a kijelölt szalagján található szó eleme-e az A nyelvnek. Ha a válasz igen, akkor a gép a q_{igen} állapotba kerül, különben pedig a q_{nem} állapotba. Mindezt egy lépésben tette, ezután pedig a számítás ennek a válasznak a függvényében folytatódik. Ezzel a gép megspórolja a kérdés megválaszolásához szükséges időt, sőt, ha az A nem rekurzív nyelv, akkor akár eldönthetetlen problémákat is képes lehet eldönteni.

Definiálhatunk orákulummal ellátott bonyolultsági osztályokat. Legyen C^A egy ilyen osztály, ahol C egy bonyolultsági osztály, A pedig egy orákulum. C^A-ba fognak tartozni az olyan problémák, amelyek megoldhatóak úgy, hogy egy C-be tartozó Turing-gépet ellátjuk az A orákulummal. Például legyen P^A azon nyelvek halmaza, amelyek felismerhetőek A orákulummal ellátott, polinomiális időbonyolultságú determinisztikus Turing-géppel.

Lássuk most az orákulumok P vs. NP kérdéshez kapcsolódó két talán legfontosabb eredményét.

Tétel:

Létezik olyan A orákulum, amelyre $P^A = NP^A$.

Bizonyítás:

Legyen A egy PSPACE-teljes nyelv. Legyen M^A egy nemdeterminisztikus, polinomiális időbonyolultságú A orákulummal ellátott Turing-gép és legyen $L = L(M^A)$. Az időbonyolultsága miatt M^A polinomiális sokszor fog kérdést intézni az A orákulumhoz, és a kérdések mindig polinomiálisan hosszú szavakra fognak vonatkozni. Emiatt a gép teljes működése (az orákulum "számítását" is beleértve) szimulálható polinomiális tár segítségével. Ebből következik, hogy $NP^A \subseteq PSPACE$. Viszont minden PSPACE-

beli L nyelv felismerhető olyan determinisztikus A orákulummal ellátott géppel, amely L-et polinomiális időben visszavezeti A-ra, majd kérdést intéz az orákulumhoz. Tehát $PSPACE \subseteq P^A$. Mivel $P^A \subseteq NP^A$ definíció szerint teljesül, ezért $P^A = NP^A$.

Mielőtt megörülhetnénk ennek az eredménynek, lássuk a másik fontos tételt.

Tétel:

Létezik olyan B orákulum, amelyre $P^A \neq NP^A$.

Bizonyítás:

A B orákulumot olyan módon fogjuk felépíteni, hogy minden lehetséges n szóhosszra csak legfeljebb egy eleme lesz, amelynek a hossza n. B ábécéje legyen $\{0, 1\}$. A nyelv, amely segítségével a tételt bizonyítani fogjuk a
$L = \{0^i | B - ben\ van\ i\ hosszú\ szó\}$
lesz. Könnyen elképzelhetünk egy olyan nemdeterminisztikus, B orákulummal ellátott gépet, amely a számítási fájának ágaira felírja az összes $\{0, 1\}$ feletti i hosszú szót, majd elfogadja a bemenetet, ha van olyan ág, amelyen lévő szót az orákulum felismeri. Tehát $L \in NP^B$. Viszont B-t meg tudjuk úgy konstruálni, hogy az elemeit egy B-vel ellátott determinisztikus Turing-gép nem tudja megtalálni polinomiális idő alatt.

B felépítése történjen a következőképpen: miközben B-t generáljuk, fenntartunk egy listát azokról a szavakról, amelyeket kizártunk a B nyelvből. Vegyünk a felsorolását az összes orákulummal ellátott determinisztikus Turing-gépnek, amelynek a szalagábécéje és az orákulum ábécéje $\{0, 1\}$. Legyenek ezek a gépek M_i, $i = 1, 2, ...$, ezeket fogjuk sorra vizsgálni. Tegyük fel, hogy minden gép végtelen sokszor szerepel a listán. Ezt könnyen megtehetjük, hiszen bármilyen géphez hozzáadható olyan új állapot, amely a számítás lefolyását soha nem befolyásolja, viszont ennek az "új" gépnek is szerepelnie kell valahol a

listán. Amikor a vizsgálat M_i-nél tart, már lesz egy listánk azokról a szavakról, amelyeket már kizártunk B-ből, és egy B_i nyelvünk, amelyben minden szó i-nél rövidebb. Ez a nyelv tartalmazza a B-hez eddig hozzáadott szavakat. Ezen a ponton i-nél rövidebb szavak már nem kerülhetnek be B-be. Szimuláljuk a $M_i^{B_i}$ működését 0^i inputon. Ha az M_i gép kérdést intéz az orákulumhoz egy olyan y szóról, amelynek a hossza i vagy nagyobb, akkor a válasz nyilvánvalóan "nem" lesz, és hogy megbizonyosodjunk róla, hogy később sem kerül be B-be, hozzáadjuk a tiltott szavak listájához. Ezzel elérjük azt, hogy ha egy akármilyen szóra az M_i gép a B_i orákulumtól a "nem" választ kapja, arra a szóra a B orákulum is a "nem" választ fogja adni. Ha a szó rövidebb, mint i, akkor semmilyen plusz intézkedésre nincs szükség. $M_i^{B_i}$ szimulációját az 0^i inputon $i^{log\ i}$ lépésig folytatjuk. Ezután attól függetlenül, hogy M_i megállt-e vagy sem, döntést hozunk, hogy hozzáadjunk-e egy szót B-hez vagy sem. Ha a gép megállt és elutasította a bemenetét, akkor hozzáadunk B-hez egy olyan i hosszú szót, amely nem szerepel a tiltott listán, feltéve, hogy létezik ilyen. A megfelelő hosszú nem tiltott szavak közül bármelyiket szabadon kiválaszthatjuk. Ha $M_i^{B_i}$ nem utasította el a bemenetét ennyi lépés után, akkor B-be nem kerül i hosszúságú szó. Természetesen akkor sem kerül szó B-be, ha M_i szimulációjának végére nem marad megfelelő hosszú nem tiltott szó. Mivel azonban $M_j^{B_j}$ szimulációján $j^{log\ j}$ lépést hajtunk végre, ezért a tiltott szavak maximális száma az $M_1, M_2, ..., M_i$ szimulációinak végére
$\sum_{j=1}^{i} j^{log\ j} \leq i(i^{log\ i}) \leq i^{1+log\ i}$
Mivel az i hosszú szavak száma 2^i, tudjuk, hogy nem lehet minden i hosszú szó tiltott, ha $2^i > i^{1+log\ i}$, azaz $i > (1 + log\ i) log\ i$. Ez igaz, ha $i \geq 32$, vagyis csak véges sok, igen kicsi i-re lehetséges, hogy minden szó tiltott legyen.

Miután M_i szimulációját befejeztük, ha szükséges, legeneráljuk a B-hez adandó i hosszú szót. Ezáltal megkapjuk az új, B_{i+1} halmazt. Ezután a folyamatot újrakezdjük a $M_{i+1}^{B_{i+1}}$ géppel a 0^{i+1} bemenettel.

Most pedig lássuk be, hogy a fent definiált L nyelv eleme $NP^B \backslash P^B$-nek. Azt már tudjuk, hogy L eleme NP^B-nek. Tegyük fel, hogy L eleme P^B-nek. Tegyük fel, hogy M_k^B p(n) polinomiális időbonyolultságú B orákulummal ellátott determinisztikus Turing-gép

felismeri L-t. Mivel ezzel ekvivalens gépek végtelen sokszor fordulnak elő a felsorolásban, ezért k-t biztosan tudjuk úgy megválasztani, hogy $k \geq 32$ és $k^{log\ k} \geq p(k)$. Ha M_k^B elfogadja 0^k-t, az azt jelenti, hogy B-nek létezik k hosszú szava. Vagyis $M_k^{B_k}$ elutasítja 0^k-t, hiszen csak ilyenkor kerülhet k hosszú szó B-be. Viszont M_k^B és $M_k^{B_k}$ azonos módon kell, hogy viselkedjen 0^k-n. A két orákulum egyetért minden szón, aminek a hossza kisebb, mint k, és B-ben nem lehet olyan k vagy hosszabb szó, amelyre $M_k^{B_k}$ rákérdez az 0^k-n folytatott működése közben, hiszen pontosan ezeket a szavakat zártuk ki B felépítése közben. Vagyis M_k^B-nak el kellene utasítania a bemenetet, ami ellentmondás. Ha M_k^B elutasítja 0^k-t, vagyis 0^k nem eleme L-nek, akkor $M_k^{B_k}$ nem utasíthatja el 0^k-t $k^{log\ k}$ lépésen belül. Ez abból következik, hogy $k \geq 32$ és ha $M_k^{B_k}$ elutasította volna $k^{log\ k}$ lépés alatt, akkor még kell, hogy legyen olyan k hosszú szó, amely nincs a tiltott szavak listáján, tehát helyet kaphat B-ben. Emiatt $0^k \in L$ kell hogy teljesüljön. Tehát M_k^B nem utasítja el 0^k-t $k^{log\ k}$ lépés alatt. Mivel $k^{log\ k} \geq p(k)$, ezért M_k^B számítása eddigre véget ér, vagyis egyáltalán nem utasítja el 0^k-t. Vagyis újabb ellentmondáshoz jutottunk.

Ezzel beláttuk, hogy $L \in NP^B \backslash P^B$.

Mit is jelentenek számunkra ezek az eredmények? Sajnos igen nagy következményük van a P vs. NP kérdés eldöntésére irányuló módszereink hatásosságára nézve.

Minden olyan módszer, amely változatlan eredményt ad akkor, ha a benne használt Turing-gépekhez tetszőleges orákulumot adunk, egyáltalán nem lesz képes eldönteni P és NP viszonyát. Minden olyan szimuláció, amely lépésről lépésre szimulál egy másik gépet, akkor is ugyanúgy fog működni, ha orákulumos gépeket használunk, elvégre az orákulum nem tesz mást, mint megspórolja a gép számára a számítás egyes szakaszait. Ha egy ilyen szimuláció eredményeként kapnánk választ P és NP viszonyára, annak akármilyen orákulum jelenlétében is teljesülnie kell, ami a fenti eredmények tükrében lehetetlen. Az a diagonalizációs módszer, amit olyan tételek felállításához használtunk, mint az időhierarchia tétel, szintén működik bármilyen orákulumot is kapcsolunk a gépekhez. Ha ezzel a módszerrel látnánk be, hogy egy nyelv $NP \backslash P$-ben van, az működne annak belátásához

is, hogy egy nyelv $NP^A \backslash P^A$-ban van, ami újfent csak nem lehetséges. Hasonló helyzetbe ütközünk abban az esetben, ha találnánk egy műveletet, amelyre nézve a P osztály zárt, de az NP osztály nem. Elképzelhető, hogy P valamilyen zártsági tulajdonsága öröklődik tetszőleges orákulum esetén is, de lehet, hogy NP esetében az eredmény nem lenne változatlan akármilyen orákulum mellett. Ellenben NP nem zártságának bizonyításához egy nyelvről be kellene látni, hogy nem NP-beli, ami valószínűleg valamilyen diagonalizációs módszerrel történne.

Lényeges ellenpélda, hogy a teljes problémák felhasználásával működő módszerek nem érzéketlenek az orákulumokkal szemben, és emiatt nem esnek áldozatul az esetleges egymásnak ellentmondó orákulumos eredményeknek. Tehát az NP-teljes problémák P-hez való viszonyának vizsgálata nem "értelmetlen" a P vs. NP kérdés szempontjából.

Az orákulumok ezen enyhe pesszimizmusra okot adó tulajdonságai ellenére hasznos eszközei lehetnek a bonyolultsági osztályok viszonyainak tisztázásához. Ha két osztályhoz kapcsolódóan olyan orákulumot találunk, amellyel ellátva nem egyenlőek, az egy komoly jelzés afelé, hogy az osztályok valóban nem egyenlőek, vagy legalább is afelé, hogy az egyenlőségüknek nem létezik egyszerű bizonyítása. Annak megmutatása, hogy valamilyen orákulummal ellátva a két osztály egyenlő, szinte mindig nagyon egyszerű. Egy olyan PSPACE-teljes orákulum, mint amit az első tételben használtunk, bármilyen osztályok esetén működik, amik PSPACE-en belül vannak. Az orákulumos eredmények remekül alkalmasak arra, hogy a segítségükkel belássuk, hogy két osztály, például a P és NP osztály viszonya mennyire tartalmas kérdés.

Az orákulumok után most nézzük meg a bonyolultságok vizsgálatának egy, a Turing-gépektől merőben eltérő segédeszközét, a Boolean-hálózatokat.

6. Hálózati bonyolultság

A hálózatok segítségével igen érdekes módon tudjuk vizsgálni az egyes feladatok bonyolultságát. Lássunk először néhány szükséges definíciót.

Egy n változós Boole-függvény alatt egy $f : \{igaz, hamis\}^n \rightarrow \{igaz, hamis\}$ függvényt értjük. Nyilvánvalóan az olyan logikai műveleteket mint az ÉS, VAGY, és NEM szintén tekinthetjük Boole-függvényeknek. A Boole-formulákat szintén tekinthetjük függvénynek, amely a benne szereplő változók logikai értékeiből előállít egy új, igaz vagy hamis értéket. Továbbá minden n változós Boole-függvény leírható legfeljebb n változós Boole-formulával. Ez könnyen belátható: legyen F az $\{igaz, hamis\}^n$ azon vektorainak halmaza, amelyre f igaz. Minden $t = (t_1, t_2, ..., t_n) \in F$ vektorra legyen D_t olyan n-tagú konjunkció (azaz az n változó ÉS művelettel való összekapcsolása), amelynek i. tagja az i. változó, ha t_i igaz, vagy az i. változó tagadása ha t_i hamis. A keresett formula ekkor az összes ilyen D_t VAGY művelettel való összekapcsolása (diszjunkciója). Ha az F halmaz üres lenne, akkor a formula egyszerűen legyen $x_1 \wedge \neg x_1$. Látható, hogy az elkészített formula $O(n^2 2^n)$ hosszú. A számunkra érdekes függvények általában jóval rövidebb formulával is leírhatók, de a létező Boole-függvények nagy többsége nem ilyen. A Boolean-hálózatok leírása után ennek az állításnak egy még erősebb változatát fogjuk látni.

A Boolean-hálózatok a formulákhoz képest egy jóval takarékosabb reprezentációja a Boole-függvényeknek. Legyen a hálózat egy $C = (V, E)$ gráf, ahol $V = \{1, 2, ..., n\}$ a csúcsok halmaza, ezeket kapuknak hívjuk. A gráfban nem lehet irányított kör, így feltehetjük, hogy minden (i,j) élre $i < j$ teljesül. Feltesszük, hogy a csúcsokba vezető élek száma (befoka) csak 0, 1 vagy 2 lehet. Ez a feltevés valójában attól függ, hogy milyen kapukat kívánunk majd használni a hálózatban. Azok a kapuk, amiket mi használni fogunk, csak 0, 1 vagy 2 befokúak lehetnek. Minden i kapuhoz hozzárendeljük az s(i) típust, amelyre $s(i) \in \{igaz, hamis, \text{É}S, VAGY, NEM\} \cup \{x_1, x_2, ...\}$. Ha a kapu típusa igaz, hamis, vagy valamelyik változó, akkor a befoka 0, azaz nem vezet bele él, ezek lesznek a hálózat

bemeneti kapui. Ha a kapu típusa NEM, akkor a befoka 1, ha ÉS vagy VAGY, akkor 2. Természetesen szükségünk lesz egy kimeneti kapura is, ez a legnagyobb sorszámú kapu lesz, amelynek már nincs rákövetkezője. Olyan hálózat is elképzelhető, amely egyszerre több függvényt számít ki, ilyenkor minden olyan kaput kimeneti kapunak veszünk, aminek nincs kimenő éle.

Most nézzük meg, hogyan határozza meg a hálózat a kimenetét. Minden i kapura definiáljuk a T(i) igazságértéket a következőképpen: vegyük sorra az i kapukat növekvő sorrendben. Ha $s(i) = igaz$ akkor $T(i) = igaz$, és ha $s(i) = hamis$ akkor $T(i) = hamis$. Ha $s(i) \in X$, azaz a kapu valamilyen változót jelöl, akkor $T(i) = T(s(i))$, vagyis a kapu értéke legyen a változó értéke. Ha $s(i) = NEM$, akkor létezik pontosan egy olyan $j < i$ kapu, amire $(j, i) \in E$. T(j) értéke ekkor már biztosan definiált. Legyen $T(i) = \neg T(j)$.Ha i befoka 2, akkor a két i-be vezető él legyen (j, i) és (j', i). Ha $s(i) = VAGY$, akkor legyen $T(i) = T(j) \vee T(j')$. Hasonlóképpen, ha $s(i) = S$, akkor $T(i) = T(j) \wedge T(j')$. A C hálózat T(C) értéke pedig legyen egyenlő T(n)-nel, ahol n a kimeneti kapu.

Egy formulához tartozó hálózat könnyen megkonstruálható induktív módon, úgy, hogy minden részformulához egy új kaput vezetünk be. Ekkor a hálózat mérete pontosan egyenlő lesz a formuláéval, vagyis nem lett takarékosabb. Viszont ha a kapukból kimenő élek számát (kifokát) nem korlátozzuk, akkor az egyes részformulákat "újrahasznosíthatjuk", azaz a többször is előforduló részkifejezésekhez nem szükséges új kapukat bevezetni. Ezáltal a hálózatokkal lehetségessé válik a függvények takarékosabb reprezentációja. Minden hálózat kiszámít egy Boole-függvényt. Azt mondjuk, hogy az n változós C hálózat kiszámítja az f n-változós Boole-függvényt, ha az összes lehetséges $t = (t_1, t_2, ..., t_n)$-re $f(t) = T(C)$, úgy, hogy a hálózat minden x_i változójára $T(x_i) = t_i$. Minden n változós Boole-függvény kiszámítható hálózattal, hiszen a függvény leírható formulával. A hálózati bonyolultság fő kérdése az, hogy n-től függően mekkorának kell lennie ennek a hálózatnak.

Akármilyen takarékosak is a hálózatok, biztosan léteznek olyan függvények, amelyek kiszámításához nagyon sok kapura van szükség. Ezt egyszerű számolással be tudjuk látni.

Tétel:

Minden $n \geq 2$ egész számhoz létezik olyan n változós Boole-függvény, amely nem számítható ki $\frac{2^n}{2n}$ vagy kevesebb kaput tartalmazó hálózattal.

Bizonyítás:

Tegyük fel, hogy létezik $n \geq 2$ szám, amelyre minden n változós Boole függvény kiszámítható maximum $m = \frac{2^n}{2n}$ kaput tartalmazó hálózattal. Összesen 2^{2^n} n változós Boole-függvény létezik. Adjunk egy felső becslést arra, hogy hány hálózat van, amelyben legfeljebb m kapu található. A hálózatot egyértelműen megadhatjuk úgy, ha tudjuk minden kapu típusát, és hogy melyik másik kapukból vezet vozzá él. A kapuk típusának száma n+5, hiszen a típus vagy változó, vagy konstans, vagy a három művelet egyike. Minden kapuba legfeljebb két másikból tart él, és az m kapuból ezt a két élt m^2 módon választhatjuk ki. Vegyük észre, hogy ebbe a felső becslésbe sok olyan gráfot is beleszámoltunk, amelyek nem felelnek meg a hálózat fogalmának. Az m darab kapu mindegyikére így $(n+5)m^2$ lehetőséget kapunk, vagyis összesen a hálózatok száma semmiképpen sem lehet több mint $((n+5)m^2)^m$. Most lássuk be azt, hogy az n változós Boole-függvények száma még ennél a felső korlátnál is nagyobb. Vegyük mindkét mennyiség kettes alapú logaritmusát. Ekkor a függvények számára 2^n-t kapunk, a felső becslésünkre pedig $2^n(1 - \frac{log\ \frac{4n^2}{n+5}}{2n})$-t kapunk. Mivel $n \geq 2$, ezért azt kapjuk, hogy a lehetséges n változós függvények száma nagyobb, mint a legfeljebb $\frac{2^n}{2n}$ kaput tartalmazó hálózatok száma. Ezzel az állítást beláttuk, azaz biztosan léteznek olyan függvények, amelyek kiszámításához a hálózatoknak exponenciálisan sok kapura van szüksége.

A hálózatokat tekinthetjük úgy, hogy bizonyos $x = x_1, x_2, ..., x_n \in \{0,1\}^n$ szavakat elfogad, másokat pedig elutasít. Ahhoz, hogy tetszőleges 0,1 ábécé feletti nyelveket fel tudjunk velük ismerni, minden lehetséges bementhosszhoz alkotnunk kell egy hálózatot.

A C hálózatcsalád alatt értsük a $C = (C_0, C_1, ...)$ hálózatok sorozatát, ahol C_n n változós. Az $L \subseteq \{0,1\}^*$ nyelv eldönthető polinomiális hálózatcsaláddal, ha létezik valamilyen p polinom, hogy C_n mérete (kapuinak száma) legfeljebb p(n), és egy x szó csak akkor eleme az L nyelvnek, ha $C_{|x|}$ igaz kimenetet ad az x bemeneten.

A változómentes hálózatok kiértékelésének problémája P-teljes. Létezik olyan visszavezetés, amely bármilyen p(n) időben eldönthető L nyelvre és x bemenetre megad egy $O(p(|x|)^2)$ méretű változómentes hálózatot, amely az x-en történő számítási folyamatot kódolja. Ennek a hálózatnak a kiértékelése polinom időben elvégezhető a hálózat igazságértékének definíciójában megadott módon.

Minden P-beli nyelv felismerhető polinomiális hálózatokkal, de az állítás megfordítása nagyon is meglepő eredményt ad. Léteznek polinomiális hálózatokkal eldönthető nem rekurzív nyelvek! Ennek a váratlan állításnak a belátásához lássuk a következőt: legyen L egy tetszőleges eldönthetetlen nyelv a $\{0,1\}$ ábécé felett, és legyen az $U \subseteq \{1\}^*$ unáris nyelv a következő: $U = \{1^n : \ n \ bináris \ alakja \ L - ben \ van\}$. U nyilván eldönthetetlen, hiszen L visszavezethető rá. Igaz, hogy a visszavezetés exponenciális idejű, de ez az eldönthetetlenségen nem változtat. Az U felismerhető egy elég triviális polinomiális hálózatcsaláddal. Ha $1^n \in U$, akkor a C_n hálózat álljon n-1 ÉS kapuból, amelyek a bemenet konjunkcióját számítják ki. Ez akkor és csak akkor lesz igaz, ha a bemenet 1^n. Ha $1^n \notin U$, akkor a hálózatunkban egyáltalán ne legyenek élek, csak a bemeneti kapukból, és a hamis típusú kimenetből álljon, ami minden bemenetet el fog utasítani. Ezzel fény derült a hálózatcsaládok egy komoly gyengéjére. Az egyes hálózatok megszerkesztéséhez korlátlan sok számítási kapacitás áll a rendelkezésünkre. Az előbbi példa során gyakorlatilag feltettük, hogy a hálózatok megkonstruálása során megoldunk egy megoldhatatlan problémát. Valamilyen módon korlátoznunk kell a hálózatok erejét. Egy hálózatcsaládot uniformnak nevezünk, ha létezik olyan log n tárat használó Turing-gép, amely az 1^n bemenetre a C_n-t adja kimenetként. Ez pontosan az a feltétel, amire szükségünk van, hogy a polinomiális számításokat azonosítani tudjuk a polinomiális hálózatokkal: egy L nyelv akkor és csak akkor dönthető el uniform polinomiális hálózatokkal, ha $L \in P$. A P-beli

nyelvek nyilván felismerhetőek ilyen hálózatokkal, pont ilyenek segítségével látható be, hogy a változómentes hálózatok kiértékelése P-teljes. Másfelől tegyük fel, hogy L felismerhető ilyen hálózatokkal. Ekkor az x bemenetről el tudjuk dönteni, hogy eleme-e L-nek úgy, hogy $log\ |x|$ tárral (ami nem futhat tovább polinomiális időnél) előállítjuk $C_{|x|}$-et, majd polinom időben kiértékeljük azt az x bemeneten.

A hálózatok segítségével újabb módon tudjuk megfogalmazni a P vs. NP kérdést: *Kiszámíthatóak-e az NP-teljes problémák uniform polinomiális hálózatokkal?*

Ez ekvivalens az eddigi megfogalmazásokkal, hiszen ha egy NP-teljes problémáról bebizonyosodna, hogy kiszámítható ilyen módon, akkor az azt jelentené, hogy P-beli, vagyis $P = NP$, ha pedig belátnánk, hogy nem, akkor tudnánk, hogy létezik NP-beli probléma, ami nem P-beli, vagyis $P \neq NP$. Valójában egy még ennél is erősebb sejtést is megfogalmaztak már: az NP-teljes problémák nem oldhatók meg polinomiális hálózatokkal, attól függetlenül, hogy azok uniformak-e vagy sem. Ez a sejtés egyáltalán nem túlzó, hiszen nemrég láttuk, hogy valójában igen kevés függvény valósítható meg kisméretű hálózatokkal. A P vs. NP-hez kapcsolódó kutatások jelentős hányada foglalkozott ennek a sejtésnek a vizsgálatával. Ennek ellenére a haladás afelé, hogy szuperpolinomiális alsó korlátot bizonyítsunk egy problémára, nagyon lassú. Jelenleg a legjobb bizonyított korlátok is csak lineárisak, pedig tudjuk, hogy a legtöbb függvénynek exponenciális méretre van szüksége! Emiatt célszerűnek tűnik valamilyen egyszerűbb, gyengébb modellel foglalkozni. A legtermészetesebb ilyen gyengített modell a monoton hálózatok modellje. Ezek a hálózatok nem tartalmazhatnak NEM kaput. Ez a modell nem annyira gyenge, hogy érdektelen lenne a számunkra. A monoton változó nélküli hálózatok kiértékelése még mindig P-teljes feladat. Monoton hálózatokkal csak monoton függvényeket lehet kiszámítani. Ezekre az a jellemző, hogy értéke nem változhat igazról hamisra ha a bemenet egy változója hamisról igazra vált. Vannak NP-teljes problémák, amelyek nem monotonok, tehát ezeket ez a modell nem képes megoldani. Ilyen például a Hátizsák probléma. Viszont léteznek monoton NP-teljes problémák is, mint a Hamilton-út probléma és a Klikk probléma. A Klikk problémára már sikerült is exponenciális alsó korlátot bizonyítani.

A bizonyítás során a problémát egyfajta "durva hálózattal" közelíti a Klikket kiszámító monoton hálózatot. A közelítés lépésről lépésre halad, a monoton hálózat minden kapuja egy lépést jelent. Minden lépés során csak kevés hibát, azaz téves pozitív vagy negatív válaszokat hoz be, viszont az eredményül kapott durva hálózat mégis exponenciálisan sokszor hibázik, amiből következik, hogy az eredeti hálózat exponenciálisan sok kapuból állt. Ha sikerülne belátni, hogy minden P-beli monoton nyelv eldönthető monoton polinomiális hálózatokkal, akkor ez az eredményt azt jelentené, hogy P nem egyenlő NP-vel. Ez azonban nem igaz, létezik olyan P-beli probléma, amelyre ez nem teljesül, például a Párosítás probléma. Ennél a problémánál egy gráfot veszünk, amelynek páros számú csúcsa van, amiket két egyenlő halmazra osztunk ("fiúkra" és "lányokra"). Egy élhalmazt párosításnak nevezünk, ha a halmazban minden él egy fiú csúcsból egy lány csúcsba vezet, és nincs két különböző él, amelyek ugyanarra a csúcsra illeszkednének. A Párosítás probléma azt kérdezi, hogy létezik-e a gráfban teljes párosítás, azaz olyan párosítás, amely minden csúcsot lefed. Mivel a Párosítás megoldásához szuperpolinomiális monoton hálózat szükséges, nem tudjuk a monoton hálózatok eredményeit közvetlenül általánosítani a hálózatok vizsgálatánál. A Párosság ezen tulajdonsága miatt lehetséges, hogy a NEM kapuk használata akár exponenciálisan gazdaságos is lehet.

A bonyolultság vizsgálatának különféle eszközei után most nézzünk meg néhány olyan nyelvosztályt, amelyek definíciója nem a bonyolultságon alapszik, viszont gyakorlati jelentőségük, valamint a P és NP osztályokhoz való viszonyuk miatt érdekesek lehetnek számunkra. Ezek az osztályok Noam Chomsky nevéhez fűződnek.

7. A Chomsky-hierarchia és P vs. NP

A Chomsky-hierarchia által definiált nyelvosztályok fontos összekötő hidat képeznek a nyelvészet, az algoritmuselmélet és az informatika között. Nézzük meg most ezen nyelvosztályokat, és helyüket a bonyolultságelméletben.

Az osztályok definícióját egyrészt megadhatjuk egy olyan géppel, amely felismeri ezeket a nyelveket (ahogy az eddig definiált osztályokkal is tettük), másrészt definiálhatjuk őket egy nyelvtan segítségével, amire különböző megszorításokat teszünk, és ezáltal különítjük el az osztályokat. A G nyelvtan alatt értsük a $G =< N, T, S, H >$ négyest. N és T két diszjunkt végez halmaz, a nemterminális és a terminális szimbólumok halmaza. A terminális szimbólumok halmaza megegyezik a nyelvtan által felismert nyelv ábécéjével, míg a nemterminálisok voltaképpen a nyelv "szerkezetét" reprezentálják. Az $S \in N$ nemterminális a kezdőszimbólum. A H a helyettesítési szabályok véges halmaza, ahol minden helyettesítés $\alpha \rightarrow \beta$ alakú, ahol α és β terminálisok és nemterminálisok véges sorozata, és α legalább egy nemterminálist tartalmaz. Definiáljuk a nyelvtanhoz tartozó mondatformákat a következőképpen: legyen S mondatforma, és ha $\alpha\beta\gamma$ mondatforma és $\beta \rightarrow \delta$ eleme a helyettesítési szabályok halmazának, akkor $\alpha\delta\gamma$ is mondatforma. Ekkor mondjuk azt, hogy $\alpha\beta\gamma$-ból egy lépésben levezethető az $\alpha\delta\gamma$, ezt jelöljük $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$-val. Egy A mondatformából n lépésben levezethető egy B mondatforma, léteznek olyan M_i mondatformák, hogy $M_i \Rightarrow M_{i+1}$ $i = 0, 1, ..., n$, és $M_0 = A$, valamint $M_n = B$. A nyelvtan által felismert nyelv legyen azon mondatformák halmaza, amelyek nem tartalmaznak nemterminális szimbólumot.

Ha a helyettesítési szabályokra semmilyen más kitételt nem teszünk, akkor meg is kaptuk a 0. típusú nyelvek osztályát. Az a gép, ami ezt a nyelvosztályt képes felismerni nem más mint a Turing-gép, vagyis a nyelvtanok nyelvek felismerése szempontjából pont annyira képesek, mint a Turing-gépek. Ez azt mutatja számunkra, hogy a nyelvtanok kifejezőereje nagyon is nagy, hiszen ekvivalens az ismert legerősebb számítási modellel. A 0. típusú nyelvek osztálya tehát megegyezik a rekurzív felsorolható nyelvek osztályával.

Az 1. típusú nyelvek osztályát egy egyszerű megszorítással kaphatjuk: a nyelvhez tartozó nyelvtanok között léteznie kell olyannak, hogy minden $\alpha \rightarrow \beta$ helyettesítési szabályra teljesül, hogy β nem rövidebb α-nál. Ebből következik, hogy egy levezetés során $|M_i| \leq |M_j|$ teljesül a mondatformákra, ha $i \leq j$. Ezt az osztályt a környezetfüggő nyelvek osztályának hívjuk. Világos, hogy a környezetfüggő nyelvek a 0. típusú nyelvek valódi részhalmazát képezik, hiszen könnyen elképzelhetünk olyan nyelvtant, amely megengedi, hogy a levezetésben egy mondatformát egy nála rövidebb kövessen, például ha a helyettesítési szabályok között szerepel $A \rightarrow \lambda$ valamilyen nemterminális A-ra. E szerint a definíció szerint környezetfüggő nyelv nem tartalmazhatná az üres szót, hiszen minden levezetés az egy hosszúságú S szimbólumból indul, amiből nem lehetne levezetni a nulla hosszú λ-t. Emiatt tesszük azt a kitételt, hogy ha egy 1. típusú nyelvnek eleme a λ, akkor ezt egyedül egy $S \rightarrow \lambda$ szabállyal lehet levezetni, és ekkor S nem szerepelhet más szabályok jobboldalán, hiszen ez rövidülésekhez vezethetne. A "környzetfüggő" elnevezés abból a normálformából ered, amibe minden ilyen nyelvhez tartozó nyelvtan hozható. Ilyenkor kikötjük azt, hogy minden helyettesítési szabály a nyelvtanban legyen $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ alakú, ahol A egy nemterminális, α_1, α_2 és β pedig terminálisok és nemterminálisok véges sorozata úgy, hogy β nem lehet az üres szó. Ekkor az A nemterminális egy mondatformában akkor helyettesíthető a β sorozattal, ha A balról az α_1, jobbról az α_2 környezetben van. Ezeket a nyelveket a lineárisan korlátolt nemdeterminisztikus Turing-gépek ismerik fel. Pontosabban, kikötjük, hogy a felismerő Turing-gépek a számítás során soha nem léphetnek le az input celláiról. Ezt könnyen elérhetjük, ha úgy vesszük, hogy az inputot két különleges szimbólum fogja közre, amit a gép nem írhat felül és az I/O fej nem léphet balra, ha elérte a baloldali jelet és nem léphet jobbra ha elérte a jobboldali jelet. Tudjuk továbbá, hogy gép pontosan annyit képes kiszámítani, mintha a tárkorlátja a bemenet hosszának valamilyen lineáris függvénye lenne. Minden algoritmus felgyorsítható ugyanis lineáris mértékben. A gyorsítás ahhoz hasonlítható, mintha egy számítógép szóhosszát növelnénk meg, úgy alakítjuk át a gépet, hogy minden betűje egyszerre több betűt kódoljon a régi gép ábécéjéből, és ezáltal a lépési során a régi gép valamilyen konstansszor több lépését tudja

szimulálni. Tehát ha a gép tárkorlátja $f(n) = cn$ valamilyen $c > 1$-re, akkor c tetszőlegesen közel vihető 1-hez. Ennek következményeképpen a környezetfüggő nyelvek felismerésére használt gépekkel bármilyen nyelv felismerhető, amihez lineáris tárbonyolultság szükséges, vagyis a környezetfüggő nyelvek osztálya egyenlő NSpace(n)-nel. Savitch tételének köszönhetően tudjuk, hogy $NSpace(n) \subseteq Space(n^2)$, tehát az 1. típusú nyelvek benne vannak PSPACE-ben. Számunkra ez egyrészt azért érdekes, mert PSPACE viszonya a P és NP osztályokhoz szintén nem ismert, vagyis akár az is megeshet, hogy P=PSPACE (ami nyilván maga után vonná a P=NP kijelentést, hiszen $NP \subseteq PSPACE$). Másfelől a környezetfüggő nyelvek igen közeli kapcsolatban állnak egy PSPACE-teljes problémával: ha adott egy G környezetfüggő nyelvtan és egy w szó, w eleme-e L(G)-nek? Ez első hallásra meglepő lehet, hiszen a lineáris tárat használó környezetfüggő nyelvek a PSPACE "alján" helyezkednek el. A PSPACE-teljesség belátásához először vegyük a nyelvtanok valamilyen kódolását. Elegendő számunkra az, ha a nyelvtanban szereplő minden szimbólumot ugyanolyan hosszú szavakkal kódolunk. Álljon az L_{kf} nyelv $x\#w$ alakú szavakból, ahol x a G_x környezetfüggő nyelvtan kódolása, w pedig egy G_x terminálisaiból álló szó. El tudunk képzelni egy olyan kétszalagos lineárisan tárkorlátos Turing-gépet (amit azután szimulálhatunk egyszalagos géppel), ami a $x\#w$ bemeneten nemdeterminisztikusan az x segítségével kitalálja az összes lehetséges levezetést addig, amíg a mondatforma el nem éri w hosszát, majd összeveti w-vel, és ha talál egyezést, akkor elfogadja a bemenetét. Ez a gép nem használ több tárat, mint az input hossza, vagyis eleme NSpace(n)-nek, azaz PSPACE-nek. Most lássuk, hogyan vezethető vissza minden PSPACE-beli nyelv L_{kf}-re. Legyen L egy tetszőleges PSPACE-beli nyelv, és legyen M egy olyan determinisztikus Turing-gép ami felismeri L-t p(n) tár felhasználásával. Legyen $L' = y\$^{p(|y|)} | yelemeL - nek$, ahol \$ egy új szimbólum. L' eleme Space(n)-nek, hiszen y-t felismerni $p(|y|)$ ideig tart, utána pedig csak az új \$ szimbólumokat kell megszámolni, és mindkét tevékenység lineáris időt vesz igénybe $y\$^{p(|y|)}$-en. L' tehát környzetfüggő nyelv. Legyen G egy környezetfüggő nyelvtan L'-höz, és legyen x G kódolása. Ekkor a polinomiális idejű visszavezetés, ami y-hoz $x\#w$-t rendeli, ahol w $y\$^{p(|y|)}$ kódolása pontosan L_{kf}-re vezeti vissza L-t. Ezzel beláttuk, hogy

L_{kf} PSPACE-teljes.

A Chomsky-hierarchia számunkra legérdekesebb osztálya a 2. típusú, azaz környezetfüggetlen nyelvek osztálya. Ezek a nyelvek egyrészt jó közelítését adják a természetes nyelvek felépítésének, másrészt (ami számunkra még fontosabb) még jobb közelítését adják a programozási nyelvek felépítésének. A legtöbb programozási nyelvről belátható, hogy ha nem is teljes egészében, de főbb részeiben generálható 2. típusú nyelvtannal, és ennek köszönhetően igen komoly szerepet kapnak a fordítóprogramok területén. A 2. típusú nyelvekhez tartozó nyelvtanok helyettesítési szabályai az 1. típusú nyelvre vonatkozó megszorítás további szigorítását jelenti: minden szabály legyen $A \rightarrow \beta$ alakú, ahol A egy nemterminális, β pedig egy terminálisokból és nemterminálisokból álló szó. Ez az 1. típusú szabályok azon részesete, amikor $\alpha_1 = \alpha_2 = \lambda$, vagyis a nemterminálisokat bármikor helyettesíthetjük egy mondatformában függetlenül attól, hogy milyen szimbólumok találhatók a környezetében, vagyis a nyelv környezetfüggetlen. A 2. típusú nyelveket veremautomatával tudjuk felismerni. A P veremautomatát a következőképpen tudjuk definiálni: legyen $P =< Q, \Sigma, \Gamma, \delta, q_0, Z_0, F >$, ahol Q az állapotok véges halmaza, Σ az input szimbólumok véges ábécéje, Γ a veremábécé, $\delta : Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^* \times \{-1, 0, 1\}$ az átmenetfüggvény, q_0 a kezdőállapot, Z_0 a kezdeti veremszimbólum, F pedig Q részhalmaza, az elfogadó állapotok halmaza. A veremautomata áll egy szalagból, amire az input szó van felírva és amit csak olvasni tud, egy belső állapotból, valamint egy veremből, amiben véges sok elem szerepel. Minden lépés során beolvassa az aktuális betűt az inputból és a verem legfelső elemét (ezt az elemet le is veszi a verem tetejéről), majd ezek és a belső állapot alapján meghatározza, hogy milyen állapotba kerüljön, merre mozduljon az olvasófej az inputszalagon és hogy a verem tetejére milyen a veremábécé betűiből álló szót írjon. Akárcsak a lineárisan korlátos gépek esetén, most is feltesszük, hogy az input két különleges szimbólum között áll, amik megakadályozzák, hogy az automata túlhaladjon rajtuk. A számítás kezdetén a gép a kezdőállapotban van, a veremben pedig egyedül a kezdeti veremszimbólum szerepel. Az input elfogadása kétféleképpen történhet, végállapot vagy üres verem szerint. Az első esetben akkor fogadjuk el a szót, ha a számítás

végén a gép elfogadó van, a másik esetben akkor, ha a számítás végén a verem üres. A felismerés ezen két módja ekvivalens, mindkét esetet át tudjuk alakítani a másikba. Ha a felismerés a végállapot alapján történt, akkor ahhoz, hogy üres verem szerinti elfogadáshoz alakítsuk át, elég egyszerűen kiürítenünk a vermet akkor, ha a szót elfogadta. Üres verem esetén pedig megtehetjük azt, hogy azt az állapotot, amiben akkor van, amikor a verem kiürült elfogadó állapotnak nevezzük ki. A veremautómatáknál is megkülönböztethetünk determinisztikus és nemdeterminisztikus gépeket attól függően, hogy minden lehetséges helyzetben egyértelmű-e a továbblépés vagy sem. Ezen kívül fontos megkülönböztetnünk az egyirányú és kétirányú veremautomatákat: ha az olvasófej sosem léphet balra, tehát csak helyben maradhat, vagy egy betűt "feldolgozva" jobbra léphet, akkor az automata egyirányú, ha a szalagon a balra lépés is megengedett, akkor pedig kétirányú. A környezetfüggetlen nyelvek osztályát pontosan az egyirányú nemdeterminisztikus veremautomaták ismerik fel.

Ez egy újabb fontos dolgot jelent a számunkra: az egyirányú determinisztikus veremautomaták csak egy szűkebb nyelvosztályt, a determinisztikus környezetfüggetlen nyelvek osztályát képesek felismerni. Ebben az esetben tehát elválik egymástól a determinizmus és a nemdeterminizmus. Lássunk egy példát egy olyan nyelvre, amely környezetfüggetlen, de nem determinisztikus környezetfüggetlen. Legyen $L = \{a^n b^n\} \cup \{a^n b^{2n}\}\ n \geq 0$. Ez a nyelv környezetfüggetlen, ugyanis generálható a következő nyelvtannal:

$S \to \lambda, S \to S_1, S \to S_2$

$S_1 \to aS_1b, S_1 \to ab$

$S_2 \to aS_2bb, S_2 \to abb$

Indirekt módon tegyük fel, hogy a nyelv determinisztikus környezetfüggetlen, tehát létezik olyan M determinisztikus veremautomata, ami felismeri. Tegyük fel, hogy a gép bemenetén $a^n b^n$ áll. A gép elfogadja a szót, és megáll valamilyen konfigurációban. Mivel a gép konfigurációinak száma véges, ezért biztosan létezik egy olyan $a^{n+k}b^{n+k}$ szó valamilyen $k > 0$-ra, hogy a gép ugyanolyan konfigurációban állt meg, mint $a^n b^n$ esetén. Most tegyük fel, hogy a bemeneten $a^n b^{2n}$ áll. Amikor a gép elér $a^n b^n$-hez, azaz még b^n betű van

hátra, a gép nyilván pontosan abban a konfigurációban lesz, mintha a bemeneten $a^n b^n$ szerepelt volna. Ekkor a gép nem tudja, hogy eddig $a^n b^n$-t vagy $a^{n+k} b^{n+k}$-t olvasott be. Emiatt a gép azt is elfogadná, ha inputként $a^{n+k} b^{2n+k}$-t kapott volna, hiszen a fennmaradó b^n betű beolvasása után nem tudja megkülönböztetni a $a^n b^{2n}$-től. Viszont $a^{n+k} b^{2n+k}$ nem eleme L-nek, tehát a gép nem fogadhatná el, ezért ellentmondáshoz jutottunk. L tehát olyan környezetfüggetlen nyelv, amelyet nem lehet determinisztikus veremautomatával felismerni. Ennél a nyelvosztálynál tehát elkülönül a determinizmus és a nemdeterminizmus.

Egy másik tény, ami fontos számunkra az, hogy a 2. típusú nyelvekről tudjuk, hogy részhalmaza P-nek. Ennek belátáshoz meg fogunk adni egy algoritmust, ami polinomiális időben felismer bármilyen nyelvet, amit egy kétirányú nemdeterminisztikus veremautomata felismer, tehát valójában a környezetfüggetlen nyelveknél egy bővebb halmazra alkalmazható. Először hozzuk a veremautomatákat egyfajta normálformába. Tegyük fel, hogy a veremautomata a szó felismerését mind az elfogadó állapotba lépéssel, mind a verem kiürítésével jelzi, valamint hogy egy lépés során a verem tetejére írt szó vagy az üres szó, vagy két betűből áll, amelyek közül az alsó megegyezik azzal a betűvel, amelyet levett a verem tetejéről a lépés során (tehát egy lépésben vagy letöröl egy betűt a veremről, vagy a tetejére ír egy újat). Minden veremautomata átalakítható ilyen formába. Legyen a bemeneti szó $w = a_2 a_{n-1}$, és az input szalagon álljon $[w]$, ahol $a_1 = [$ és $a_n =]$ a bal- és jobboldali határoló szimbólumok. Legyen az elfogadó állapot q_f, a kezdőállapot q_0, a kezdeti veremszimbólum pedig Z_0. A veremautomata pillanatnyi helyzetét írjuk le $(p, [w], i, \gamma)$ módon, ahol p az automata állapota, w az inputszalagon szereplő szó, i az olvasófej helyzete és γ pedig a verem tartalma. A felismerő algoritmus fő eszköze egy $n \times n$-es R mátrix lesz, amit felismerő mátrixnak hívunk. A mátrix minden $r(i,j)$ $1 \leq i, j \leq n$ eleme $Q \times \Gamma \times Q$ részhalmaza. Az $r(i,j)$ akkor és csak akkor fog tartalmazni egy (p, Z, q) elemet, ha a veremautomata $(p, [w], i, Z)$ állásából elérhető a $(q, [w], j, \lambda)$ állás. Tehát az r(1,n) akkor és csak akkor fogja tartalmazni a (q_0, Z_0, q_f) elemet, ha a kezdő konfigurációból elérhető egy olyan konfiguráció, amiben az automata elfogadó állapotban van és a verme

üres, tehát ha az automata felismerte a bemenetét. A mátrix elkészülte után elég lesz az r(1,n) elemet megvizsgálni ahhoz, hogy eldöntsük a bemenet eleme-e az automata által felismert nyelvnek. A δ átmenetfüggvényt bontsuk kétfelé, δ_1-re és δ_2-re. A δ_1 függvény képezzen $\{1,2,...,n\}^2$-ből $Q\times\Gamma\times Q$ részhalmazaiba, ez tartalmazza azokat az átmeneteket, amik során veremszimbólumot törlünk. Pontosabban $\delta_1(i,i+d)$ tartalmazza (p,Z,q)-t akkor és csak akkor, ha $\delta(p,a_i,Z)$ tartalmazza (q,λ,d)-t. A δ_2 képezzen $\{1,2,...,n\}^2$-ből $Q\times\Gamma\Gamma\times Q$-ba, és tartalmazza azokat az átmeneteket, amik a verem tetejére új szimbólumot írnak. Pontosabban $\delta_2(i,i+d)$ tartalmaza (p,ZY,q)-t akkor és csak akkor, ha $\delta(p,a_i,Z)$ tartalmazza (q,ZY,d)-t. Definiáljunk még egy Θ függvény a következőképpen: legyen három összetett argumentuma, az első legyen $Q\times\Gamma\Gamma\times Q$ részhalmaza, a második és harmadik argumentuma, valamint az értékkészlete pedig legyen $Q\times\Gamma\times Q$ részhalmaza. Legyen $\Theta(A_1,A_2,A_3)$ azon (p,Z,q)-k halmaza, amelyekre teljesül, hogy létezik olyan q_1,q_2,Z', hogy (p,ZZ',q_1) eleme A_1-nek, (q_1,Z',q_2) eleme A_2-nek és (q_2,Z,q) eleme A_3-nak. Például ha a $\delta_2(i,i+d)$ tartalmaza (p,ZY,q_1)-t, $r(i+d,j)$ tartalmazza (q_1,Y,q_2)-t és r(j,k) tartalmazza (q_1,Z,q)-t, akkor $\Theta(\delta_2(i,i+d),r(i+d,j),r(j,k))$ tartalmazza (p,Z,q)-t. Ez azt jelenti, hogy a $(p,[w],i,Z)$ konfigurációból elérhető a $(q,[w],k,\lambda)$ konfiguráció. Egy ilyen átmenetet fogunk egy elemi lépésnek tekinteni, tehát ezt a függvényt fogjuk használni az algoritmus bonyolultságának mérésére. Vegyük észre, hogy $\Theta(A_1,A_2,A_3)$ kiértékelése akármilyen argumentumokra nem függ a w bemeneti szótól, csakis a P veremautomatától függ. Az R mátrix kiszámításához egy A vermet is felhasználunk segítségül. A-nek a legalsó kivételével minden eleme (i,j,(p,Z,q)) alakú, ahol $1\leq i,j\leq n$, p és q elemei Q-nak és Z eleme Γ-nak. Ha (i,j,(p,Z,q)) eleme A-nak, akkor (p,Z,q) eleme r(i,j)-nek. R kiszámítása során meg fogjuk határozni, hogy milyen új (p,Z,q) hármasok kerülhetnek be R-be azon elemek alapján amik már R-ben vannak. Minden új (p,Z,q) amit hozzáadunk valamilyen r(i,j)-hez hozzáadódik A-hoz is (i,j,(p,Z,q))-ként. Az R-ben található hármasok maximális száma $|Q|^2|\Gamma|n^2$. Kezdéskor R minden eleme üres és A egyetlen eleme a (0,0,0). A(1), A(2) és A(3) jelölje A legfelső elemének első, második és harmadik komponensét. Az M Turing-gép amit használni fogunk négy szalaggal

rendelkezik, az egyiken fogja tárolni az inputot és a másik három szalagon fog dolgozni. Ebből a három szalagból kettőt osszunk fel $|Q|^2|\Gamma|$ sávra (vehetjük úgy, hogy a gép ennyi külön szalagon dolgozik, amiket végül egy szalagon szimulál), ugyanis ennyi a különböző (p,Z,q) hármasok maximális száma. Ezeken a szalagokon minden sávban [i,(p,Z,q)] alakú szimbólumok lesznek, az i prefix lehetséges értékei 0, 1 vagy 2 attól függően, hogy (p,Z,q) benne van az A veremben, az A verem legfelső eleme vagy nem szerepel az A veremben. A harmadik szalagon végzi majd a szükséges egyéb számításokat. Az algoritmus működjön a következőképpen:

1. M az első két szalagján töltsön ki n^2 cellát úgy, hogy minden cella minden sávjában álljon a $[2, \lambda]$ elem. Minden cella R egy r(i,j) elemét fogja reprezentálni.

2. Ezután M kiszámítja $r(i, i+d) = \delta_1(i, i+d)$-t minden $i \in \{1, ..., n\}$-re és $d \in \{-1, 0, 1\}$-re ahol $1 \leq i + d \leq n$. Ha $\delta_1(i, i+d)$ tartalmazza (p,Z,q)-t, akkor M az első két szalag azon cellájába, amely $r(i, i+d)$-t reprezentálja egy üres sávba beírja a $[0, (p, Z, q)]$ szimbólumot.

3. M megkeresi az első szalagon a legbaloldalibb olyan cellát, amely valamely sávjában tartalmaz 0 indexű elemet Ha ilyen nincs, akkor az A verem kiürült. Ekkor M megnézi az r(1,n)-et reprezentáló cellát, hogy tartalmazza-e a $[2, (p_0, Z_0, q_f)]$ szimbólumot. Ha igen, akkor M elfogadja a bemenetet, ha nem, akkor elveti. Viszont ha talált egy $[0, (p, Z, q)]$ alakú elemet valamilyen r(i,j) cellában, akkor M változtassa meg ennek az elemnek a prefixét 1-re.

4. Ekkor minden $d \in \{-1, 0, 1\}$-re és i-re ahol $1 \leq i - d \leq n$, valamint minden k-ra, $k \in \{1, 2, ..., n\}$ M kiszámítja $r(i-d, k) = \Theta(\delta_2(i-d, i), \{(p, Z, q)\}, r(j, k))$-t. Ehhez M eltárolja (p,Z,q) értékét a belső állapotában, és beállítja az első és második szalag I/O fejeit az r(i-d,1) és r(j,1) cellák felé. M ezután szinkronban mozgatja a két fejet jobbra n lépésnyit. Minden $k = 1, 2, ...n$-re M kiolvassa r(j,k) értékét a

második szalagról és az első szalag üres sávjaiba beír minden új (p', Z', q') elemet az $r(i-d,k)$ cellába $[0,(p',Z',q')]$ formában. Amint M kiszámította $r(i-d,n)$-et, M átmásolja az első szalag tartalmát a második szalagra és megismétli ezt d következő értékére.

5. Amikor M végzett a 4. pont számításával, M megkeresi az $[1,(p,Z,q)]$ elemet r(i,j)-ben az első szalagon, és megváltoztatja a prefixét 1-ről 2-re.

6. Ekkor minden $d \in \{-1,0,1\}$-re és h-ra ahol $1 \leq h+d \leq n$, M kiszámítja $r(h,j) = \Theta(\delta_2(h,h+d), r(h+d,i), \{(p,Z,q)\})$-t. M most is a második szalagról olvassa $r(h+d,i)$ értékeit és beírja az első szalag r(h,j) celláiba azokat a hármasokat, amelyek nem szerepelnek benne. Minden ilyen új elem prefixe 0. Amikor M kiszámította $r(n,j)$-t, M átmásolja az első szalag tartalmát a másodikra.

7. M megismétli az algoritmust a 3. lépéstől kezdve.

Az algoritmus minden lépése legfeljebb $O(n^2)$ lépésébe kerül a Turing-gépnek, és mivel a 3-6 lépéseket legfeljebb $|Q|^2|\Gamma|n^2$-szer kell megismételni, ezért az algoritmus időbonyolultsága $O(n^4)$. Az is látható, hogy nem használ $O(n^2)$-nél több tárat.

A hierarchia legalsó fokán a 3. típusú nyelvek, más néven a reguláris nyelvek állnak. Az olyan nyelvtanokat nevezzük regulárisnak, amelyek bal- vagy jobblineárisak. Ez azt jelenti, hogy minden helyettesítési szabály vagy $A \to Bw$, vagy $A \to w$ alakú (ahol A és nemterminálisok, w pedig egy terminálisokból álló szó) ha ballineáris, és $A \to wB$, vagy $A \to w$ alakú ha jobblineáris. Látható, hogy ezek a nyelvek a 2. típusú nyelvek részesete, hiszen itt már nem csak a szabályok baloldalára, hanem a jobboldalára is megszorításokat tettünk. Ezeket a nyelveket a véges automaták ismerik fel, amelyek olyan gráfok, ahol a csúcsok állapotokat tartalmaznak, az élek pedig átmeneteket: az inputból minden beolvasott szimbólum átviszi az automatát egyik állapotból a másikba, feltéve ha létezik a szimbólumnak megfelelő él. Az automata elfogadja a bemenetet, ha az input végigolvasása után elfogadó állapotba kerül. Voltaképpen egy olyan Turing-gépként is elképzelhetőek, mint ami az inputon és az állapotain kívül semmit sem használ

a számításához. Érezhetően számítás szempontjából ezek a nyelvek már meglehetősen egyszerűek.

Mint arról szó esett, a Chomsky-hierarchia osztályai az informatika gyakorlatában meglehetősen nagy szerepet játszanak. Szakdolgozatom mellékleteként egy olyan Java nyelven írt programot adok be, amely a Turing-gép mellett az egyirányú nemdeterminisztikus veremautomata (vagyis pontosan a környezetfüggetlen nyelveket felismerő veremautomata) működését mutatja be, mivel mint azt láthattuk, mindkettő szerepe igen jelentős a bonyolultságelmélet és a számítástudomány szempontjából. Lássuk most a program felépítését és működését.

8. Implementációk

Programom írásakor azért döntöttem a Java nyelv használata mellett, mert ezáltal az elkészült osztályok könnyen újrafelhasználhatóak és meglehetősen rugalmasak, valamint a Java nyelv jó hordozhatósága miatt széles körben felhasználható. Ráadásul az objektum orientált felépítésnek köszönhetően a gép szükség esetén könnyen kezelhető egyetlen jól határolható egységként, valamint ha kell, egyes alkotóelemei is képesek önálló viselkedésre.

Az osztályok használatához szükséges dokumentációt mellékletként csatolom a szakdolgozathoz, itt pedig áttekintem felépítését és működése néhány jellemzőjét.

Az algoritmusok vizsgálatához használt olyan modellek, mint a Turing-gép, több tekintetben is különböznek a számítógépektől. Mindenekelőtt a Turing-gépről feltételezzük, hogy végtelen mennyiségű tárterület áll rendelkezésére. Ha a gépnek csak egy konstans, véges nagyságú "memóriája" lenne, az az algoritmusok elemzését igencsak megnehezítené, például teljesen elfelejthetnénk az olyan inputokat, amelyek hossza nagyobb a gép befogadóképességénél. A végtelen memória azonban nyilvánvalóan nem implementálható. A Turing-gép implementációja a szalagokat a következőképpen kezeli: minden szalag legelső

szimbóluma egy kitüntetett "szalag eleje szimbólum", amelytől a szalag I/O feje nem léphet balra. Ez nem korlátozza a Turing-gép általánosságát, minden gép átalakítható úgy, hogy amikor balra akar lépni a szalag "elejéről", akkor a szalag teljes tartalmát eggyel jobbra tolja. Ez az átalakítás legfeljebb a négyzetére emeli az időbonyolultságot (hiszen ha az időbonyolultság f(n), akkor minden alkalommal legfeljebb f(n) szimbólumot kell megmozgatnia, mivel ennél több nem lehet a szalagon, és erre legfeljebb f(n)-szer kerülhet sor), tehát nem vezet ki P-ből. Az I/O fej mozgása során a szalagot reprezentáló objektum észleli, ha a fej elérte a szalag jobb szélét, és ekkor egy új, üres szimbólumot tartalmazó cellát ad a szalaghoz, ezáltal a fej szabadon mozoghat jobbfelé. A determinisztikus és nemdeterminisztikus Turing-gépek implementációi az átmenetfüggvény kezelésében térnek el. Mindkét esetben a függvény egy HashMap reprezentálja, amelyben a kulcsok a szabályok baloldalai. Determinisztikus esetben minden baloldalhoz csak egy jobboldal tartozhat, nemdeterminisztikus esetben viszont jobboldalak halmaza. A Turing-gép példányosítása lehetséges minden definíció szerinti tag explicit megadásával, de néhány közülük elhagyható, amikhez rendelhető alapértelmezett érték. Ilyen nem szükséges megadni az állapotot (a kezdőállapotot állítja be jelenlegi állapotként), ellenőrzi, hogy a kezdő- és végállapotok elemei-e az állapotok halmazának és hozzáadja ha nem így lenne, valamint a k darab szalagot mind üresnek inicializálja. A Szimbolum osztály két kitüntetett tagja, a szalag elejét jelző szimbólum és az üres szimbólum minden szimbólumhalmaznak (az állapotok halmazának, a szalagábécének, veremautomata esetén a veremábécének) alapértelmezetten eleme. Példányosítás után az initInput(String s) metódussal adható meg az input String formában, ahol minden betűt külön szimbólumnak tekint. A létrehozott Turing-gép minden eleme külön objektum, így a program működése során bármikor könnyen lecserélhetők vagy módosíthatók. A Turing-gép egy lépését a lepes() metódus hajtja végre, nemdeterminisztikus gép esetén a metódus bemenő paraméterként egy átmenet-jobboldalt kér, ami tartalamazza a lépéshez szükséges adatokat. Determinisztikus esetben erre a paraméterre nincs szükség, hiszen a végrehajtandó lépés egyértelmű a gép mindenkori állásából. A szamitas() metódus nemdeterminisztikus gép esetén a teljes számítási fát bejárja, viszont

az első elfogadó számítás megtalálása után megáll és visszatér igaz értékkel. A determinisztikus gép esetén ez a metódus egyszerűen végiglépked a determinisztikus számításon. A szamitasLogged() metódus a szamitas()-hoz hasonlóan működik, viszont egy plusz paramétert igényel, egy Turing-gép konfigurációkból álló ArrayList-et. A számítás során ebbe a tömbbe elmenti az összes végrehajtott lépés után a Turing-gép konfigurációját. Ezeken kívül még több kisebb metódus tartalmaznak a Turing-gép implementációs osztályai, amelyek főként különféle lekérdezéseket hajtanak végre, és amelyek segítségével a számítási folyamat nagymértékben irányítható.

A veremautomata implementációja egyirányú nemdeterminisztikus gépek létrehozását teszi lehetővé. Csak egy szalaggal rendelkezik, amelyre a lépései során nem képes írni, valamint egy veremmel, amelynek legalsó eleme ugyanaz a szimbólum, amely a szalag elejét is jelzi. Az elemek itt is megadhatók mind explicit módon, vagy az állapot, szalag és verem elhagyásával is létrehozható a gép kezdőállapotban, üres inputszalaggal és veremmel. Az átmenetfüggvényben szereplő szabályok baloldalán az inputról beolvasott szimbólum lehet üres, ekkor a szalag olvasófeje helyben marad, különben pedig eggyel jobbra lép. A lepes() metódus itt egy átmenet szabály baloldalát és jobboldalát egyaránt kéri paraméterként, hiszen az olvasófej mozgása itt a szabályok baloldalából olvasható ki. A szamitas() metódus a Turing-géppel hasonló módon működik, és itt is lehetőség van a számítás menetének ArrayList objektumba mentésére. A számítás akkor ér véget, ha az olvasófej üres szimbólumon áll, tehát végigolvasta a teljes inputot, ekkor dönti el az állapota alapján, hogy a szót elfogadja-e vagy sem.

Az osztályok használatának példájaként mellékelek két osztályt, a TuringFrame.java és PDAFrame.java osztályokat, amelyek egy fájlból képesek beolvasni, majd ezek alapján összeállítani egy Turing-gépet illetve veremautomatát. Az így létrehozott gépekkel egyszerűen szemmel követhetjük a számítás folyamatát. Két .txt formátumú fájlt is mellékelek, amelyek példaalgoritmusként szolgálnak ehhez a két osztályhoz: a pda.txt alapján létrehozott veremautomata a $\{0^n1^n\}$ $n \geq 0$ nyelvet ismeri fel, a turing.txt fájl alapján létrehozott Turing-gép pedig a $\{0, 1\}$ ábécé feletti palindromok nyelvét ismeri fel.

A TuringFrame.java futtatása során a betöltés mellett lévő szövegdobozba írhatjuk be a beolvasni kívánt .txt fájl elérési útvonalát. Az OK gombra kattintva a fájlból beolvassa a szükséges adatokat (úgymint az állapotok halmaza, a veremábécé, az átmenetfüggvényben található szabályok, stb.), amelyeket megjelenít a különböző panelekben. Ha elkészült, vagy a betöltés valamilyen oknál fogva sikertelen, arról hibaüzenetet kapunk a "Log" feliratú panelben. A "Szalagok" panelben látható a gép szalagjainak tartalma, az "Állapot" feliratúban a jelenlegi állapota, a "Lehetséges szabályok" feliratúban pedig az adott helyzetben alkalmazható szabályok vannak felsorolva. Az "előre" gombra kattintással a gép megtesz egy lépést. Ha több lehetséges szabály közül választ, akkor a "Választás" mezőbe egy megfelelő nagyságú számot vár, aminek megfelelő sorszámú szabály alapján fog tovább lépni. Ha a szám nem megfelelő, de a számítást tekintve nem ez volt az utolsó már bejárt konfiguráció (azaz a logban, amely az eddigi lépéseket tartalmazza, nem a jelenlegi az utolsó bejegyzés), akkor a gép a log segítségével lép előre egyet A "vissza" gombra kattintással visszalép egy lépést. A "Futtatás" gomb akkor ajánlott, ha a számítás végeredménye érdekel minket és nem a számítás menete. Erre a gombra kattintva a gép végrehajtja a teljes számítást loggolva, majd a végeredményt és a logot kiírja a Log feliratú panelbe. A futtatásra kattintással a log eddigi tartalma elvész, azaz visszafelé csak addig lesz bejárható, amíg el nem érünk arra a pontra, ahol kiadtuk a futtatás parancsot.

A beolvasott .txt módosítása egyszerű, formátuma legyen a következő: A fájl első sora az input szó, a második sor a szalagok száma, a harmadik az állapotok halmaza (szóközzel elválasztva), a negyedik a szalagábécé ugyanilyen módon megadva, az ötödik a kezdőállapot, a hatodik pedig a végállapotok halmaza, szintén szóközzel elválasztva. Ezt követően minden sor az átmenetfüggvény egy-egy szabályát tartalmazza. Ezen sorok megadása legyen a következő formátumú:

$(q, a_1a_2...a_n) -> (q'_1, b_{11}b_{12}...b_{1n}, m_{11}m_{12}...m_{1n}); ...; (q'_m, b_{m1}b_{m2}...b_{mn}, m_{m1}m_{m2}...m_{mn})$

A szabályok megadása során a szóköz az üres szimbólumot, a "$>$" jel pedig a szalag eleje szimbólumot jelöli. A fejek mozgásai a B,H,J betűkkel adhatók meg.

A PDAFrame.java osztály teljesen analóg módon működik a TuringFrame.java

osztállyal, kivéve, hogy veremautomatát alkot és annak megfelelő adatokat jelenít meg. Ebben az esetben a beolvasott .txt fájl legyen a következő alakú: az első három sor rendeltetése változatlan, a negyedik sor a veremábécé, az ötödik a kezdőállapot, a hatodik az elfogadó állapotok halmaza, a hetedik pedig a verem kezdeti tartalma. Ha azt akarjuk, hogy a verem csak a kezdeti veremszimbólumot tartalmazza, ez a sor legyen üres. Az ezt követő sorok mind egy-egy átmeneti szabályt kódolnak. Ezek formátuma: $(q, a, Z) -> (q', \alpha)$, ahol q és q' állapotok, a egy input szimólum (lehet üres sztring, ekkor a gép nem olvas be szimbólumot), Z a verem tetején lévő szimbólum, $\alpha \in \Gamma^*$ pedig egy veremszimbólumokból álló szó, amely a verem tetejére kerül (ez szintén lehet üres sztring).

Reményeim szerint ezek a példaprogramok megmutatják, hogy a felhasznált osztályok hasznos eszközei lehetnek az algoritmusok gyakorlati implementálásának és megértésének. Habár az elméleti szempontból fontosabb algoritmusok implementálása Turing-gépen meglehetősen hálátlan feladat lehet, egyszerűbb algoritmusok segítségével szemléltethetők a gépek alapvető működései és tulajdonságai. Bár a bonyolultságelmélet területe helyenként igencsak absztrakt, a gyakorlati szemlélet segítségünkre lehet a megértésében.

9. Összefoglalás

A szakdolgozatomban a P vs. NP probléma megértéséhez szükséges ismereteket kívántam összegyűjteni, valamint a megoldására tett főbb kísérletek eszközeit szándékoztam ismertetni.

Az első fejezetben az alapvető fogalmak kimondása mellett láthattunk néhány olyan tételt, amelyekkel jobb rálátást nyerhetünk a problémára. Az ezt követő történeti áttekintés során olvashattunk a probléma eredetéről, korai eredményekről és a megoldási kísérletek fontosabb vonalairól. Az NP-teljes problémákkal foglalkozó fejezet során ezen problémák jelentőségéről és felismerésük módjáról esett szó. Ezt követte az orákulumok első ránézésre ellentmondó eredményeivel és ezek a probléma kutatásának módszereire gyakorolt hatásával foglalkozó rész. A Boolean-hálózatok az aktív kutatásbeli fontosságuk miatt szintén helyet kaptak a dolgozatban. Végül a Chomsky-hierarchia, ami mind a természetes, mind a formális nyelvek kutatásának fontos eleme, szintén ismertetésre kerül tekintettel a P és NP osztályokhoz való kapcsolatára.

A dolgozathoz mellékelt program és dokumentációja alapvetően algoritmusok formális implementálására alkalmas. Segítségével egyfelől könnyebben képzelhetjük magunk elé az algoritmusok vizsgálatához használt absztrakt gépeket, ami segítséget jelenthet egy ilyen meglehetősen "elvont" terület megértésében, másfelől a Java nyelvű osztályok hordozhatósága és újrafelhasználhatósága miatt alkalmas eszközök formális nyelvekkel vagy algoritmusokkal foglalkozó célirányos rendszerek felépítésére.

Természetesen alaptalan lenne azt gondolni, hogy a szakdolgozatban összefoglaltak elolvasásával komoly szaktudásra tehetnénk szert a P vs. NP probléma kapcsán. A probléma az itt leírtaknál jóval terebélyesebb, ahogy az illik egy olyan kérdéshez, amely központi szerepet élvez a bonyolultságelmélet, és ennek köszönhetően az informatika területén. A dolgozat hossza így is meghaladja valamivel az ajánlott terjedelmet, amiért ezúton szeretnék elnézést kérni. Bár a szakdolgozat talán csak a "jéghegy csúcsát" mu-

tatja be a problémának, reményeim szerint alkalmas lehet arra, hogy megsejtsük az alatta fekvő jéghegy nagyságát.

A problémában való elmélyedéssel töltött idő alatt arra lettem figyelmes, hogy az egyes kutatók cikkeinek hangvétele optimizmus tekintetében nagyjából monoton csökkenő tendenciát mutatnak az idő előrehaladtával. Voltaképpen ez egyáltalán nem meglepő, hiszen bármilyen komoly probléma esetén a probléma "fiatalkorában" még nem lehet tudni, hogy a megoldásához mennyi erőforrásra, például időre lesz szükség, vagy hogy a megoldás mennyire lesz rövid és elegáns.

A P vs. NP meglehetősen "makacsnak" bizonyult. Nemcsak az derült ki róla, hogy sok szerteágazó területen jelenik meg valamilyen formában, de gyakorlati jelentősége is tovább növelte a neki szentelt figyelmet. Ennek ellenére a története során nem egyszer tűnt úgy, hogy akármerről is közelítenek felé, csupán újabb akadályokat találnak. Például a hálózati bonyolultság felől közelítve a célunk az lenne, hogy bizonyos NP-beli problémák bonyolultságára szuperpolinomiális alsó korlátokat bizonyítsunk, de több évtized után is az általános esetben a legjobb korlát is csak lineáris! Ezt csak tetézi az a tény, amit egy egyszerű számolás alapján tudunk: hogy a problémák túlnyomó többségének exponenciális bonyolultsága van. A hierarchia-tételek segítségével szintén érdekes megvilágításba kerül a tudásunk a bonyolultsági osztályokról: tudjuk, hogy $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$, és tudjuk azt, hogy L valódi részhalmaza PSPACE-nek. Tehát a fenti részhalmaz relációk közül legalább az egyik valódi, (sőt, a sejtésünk szerint mindegyik valódi) de jelenleg egyikről sem tudjuk ezt bizonyítani. Az orákulumok segítségével pedig sikeresen beláttuk, hogy sok "egyszerű" módszer alkalmatlan a P és NP elválasztására, gyakorlatilag garantálva a megoldás nehézségét. A kérdésre egy rövid, egyszerű, konstruktív válasz ötlete a jelenlegi eszközeink mellett szinte naivnak hat.

Az orákulumok eredményei tulajdonképpen azt mutatják, hogy a probléma bizonyos szinten független azoktól a rendszerektől, amikkel vizsgálni kívánjuk. Nem meglepő, hogy a kérdés függetlensége, azaz bizonyíthatatlansága is felmerült az évek során. Tekintve, hogy a kérdés gyakorlatilag a kiszámíthatóság egy fontos tulajdonságát kívánja kiszámí-

tani, ezt sem vethetjük el, hiszen az eldönthetetlenség határán több olyan matematikai probléma is van, ami magáról a matematikáról tesz fel kérdést. A jelenlegi eredmények is afelé mutatnak, hogy a megoldáshoz merőben újszerű eszközökre lehet szükségünk.

Tegyük fel, hogy a kérdés mégis eldönthető. Egy megtalált bizonyításról be tudnánk látni annak helyességét, tehát alkalmas tanúja lenne a P vs. NP kérdésnek. A jelenlegi eredmények és a probléma több évtizedes történelme nem azt sugallják, hogy a bizonyítás ellenőrzésének nehézsége összemérhető lenne a bizonyítás megtalálásának nehézségével. A $P \neq NP$ állítás érdekes iróniája ez: ha igaz, azzal szinte borítékolja saját bizonyításának nehézségét.

Köszönetnyilvánítás

Köszönöm Herendi Tamás tanár úrnak amiért érdekes óráival megszerettette velem a bonyolultságelméletet és rokonait, és amiért mindenben végig a segítségemre volt.

10. Irodalomjegyzék

Könyvek:

- Introduction to Automata Theory, Languages, and Computation; John Edward Hopcroft, Jeffrey David Ullman; 1979 Addison-Wesley Publishing Company, Inc.
- Handbook of Formal Languages Volume 1 Word, Language, Grammar; Grzegorz Rozenberg, Arto Salomaa; 1997 Springer-Verlag Berlin Heidelberg
- Handbook of Formal Languages Volume 2 Linear Modeling: Background and Application; Grzegorz Rozenberg, Arto Salomaa; 1997 Springer-Verlag Berlin Heidelberg
- Algoritmusok bonyolultsága; Lovász László; 1990 Budapest ELTE
- Számítási bonyolultság; Christos Harilaos Papadimitriou; 1999 Győr Novadat
- Algoritmusok; Rónyai Lajos, Ivanyos Gábor, Szabó Réka; 1998 Budapest Typotex

Cikkek:

- Time and Tape Complexity of Pushdown Automaton Languages; Alfred Vaino Aho, John Edward Hopcroft, Jeffrey David Ullman; 1968 Information and Control
- The History and Status of the P versus NP Question; Michael Sipser; 1992 Proceedings of ACM STOC'92
- Is P Versus NP Formally Independent? Scott Aaronson; 2003 Bulletin of the EATCS
- The Hardest Context-free Language Revised; Julia Stoll; 2002 Report A/2002/4 University of Kuopio

Hivatkozások:

- [1] M. O. Rabin: Degree of difficulty of computing a function and a partial ordering of recursive sets; Tech. Rep. No. 2, Hebrew University, 1960
- [2] J. Hartmanis, R. E. Stearns: On the computational complexity of algorithms; Transactions of the AMS 117, 285-306, 1965
- [3] M. Blum: A machine-independent theory of the complexity of recursive functions; Journal of the ACM, 14, no. 2, 322-336, 1967
- [4] S. A. Cook: The complexity of theorem proving procedures; Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151-158, 1971
- [5] L. Levin: Universal sequential search problems; Problems of Information Trans 9, 3, 265-266, 1973
- [6] R. M. Karp: Reducibility among combinatorial problems; Complexity of Computer Computations, Plenum Press, 85-103, 1972
- [7] T. P. Baker, J. Gill, R. Solovay: Relativizations of the P=?NP question; SIAM Journal on Computing, 4:4, 431-442, 1975
- [8] C. Lund, L. Fortnow, H. Karloff, N. Nisan: Algebraic methods for interactive proof systems; Proceedings of the 22th Annual Symposium on Theory of Computing, 2-10, 1990
- [9] A. Shamir: IP=PSPACE; Proceedings of the 22th Annual Symposium on Theory of Computing, 11-15, 1990
- [10] R. J. Lipton: Model theoretic aspects of complexity theory; Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 193-200, 1978

- [11] R. E. DeMillo, R. J. Lipton: The consistency of P=NP and related problems within fragments of number theory; Proceedings of the 11th ACM Symposium on Theory of Computing, 153-159, 1979
- [12] J. Hartmanis, J. Hopcroft: Independence results in computer science; ACM SIGACT News 8, no. 4, 13-24, 1976
- [13] R. Fagin: Generalized first-order spectra and polynomial-time recognizable sets; Complexity of Computation, SIAM-AMS Proceedings 7, 43-73, 1974
- [14] N. D. Jones, A. L. Selman: Turing machines and the spectra of first-order formulas; Journal of Symbolic Logic 39, 139-150, 1974
- [15] J. E. Savage: Computational work and time on finite machines; Journal of the ACM 19:4, 660-674, 1972
- [16] Iwama K., Morizumi H.:An Explicit Lower Bound of $5n - o(n)$ for Boolean Circuits; Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science, 353-364, 2002
- [17] M. Furst, J. Saxe, M. Sipser: Parity, circuits and the polynomial time hierarchy; Mathematical Systems Theory 17, 13-27, 1984
- [18] A. A. Razborov: Lower bounds on the size of bounded depth networks over a complete basis with logical addition; Mathematical notes of the Academy of Sciences of the USSR 41:4, 333-338, 1987
- [19] A. E. Andreev: On a method for obtaining lower bounds for the complexity of individual monotone functions; Soviet Mathematics Doklady 31:3, 530-534, 1985
- [20] A. A. Razborov: A lower bound on the monotone network complexity of the logical permanent; Mathematical notes of the Academy of Sciences of the USSR 37:6, 485-493, 1985

- [21] S. J. Berkowitz: On some relationships between monotone and non-monotone circuit complexity; Technical Report, Computer Science Department, University of Toronto, 1982

- [22] M. Karchmer, A. Widgerson: Monotone circuits for connectivity require super-logarithmic depth; Proceedings of the 20th ACM Symposium on Theory of Computing, 539-550, 1988

- [23] M. Grigni, M. Sipser: Monotone separation of NC^1 from logspace; Proceedings of the 6th Annual Symposium on Structure in Complexity Theory, 294-198, 1991

11. Függelék

11.1. Áttekintés

A csomagokban nemdeterminisztikus veremautomata és nemdeterminisztikus Turing-gép, valamint az ezek felépítéséhez szükséges elemek Java nyelvű implementációja található. Külön található még determinisztikus Turing-gép implementáció is, amely belső reprezentációja kihasználja az általános nemdeterminisztikus esethez viszonyított egyszerűsítési lehetőségeket.

11.1.1. A csomagok áttekintése

package elemek

Az ebben a csomagban található osztályok a modellek alapvető építőkövei, a modellspecifikus osztályok jórészt ezekből épülnek fel.

-Szimbolum.java
-Cella.java
-Abece.java
-Szalag.java
-FejMozgas.java

package turingspec

A csomag osztályai a determinisztikus és nemdeterminisztikus Turing-gépek modelljeihez használt speciális adatszerkezetek.

-AtmenetBaloldal.java

-AtmenetJobboldal.java

-AtmenetFuggvenyDTG.java

-AtmenetFuggvenyNDTG.java

-TGKonfiguracio.java

-TuringGep.java

package pdaspec

Ezek az osztályok a nemdeterminisztikus veremautomata modelljéhez használt speciális adatszerkezetek.

-AtmenetBaloldalPDA.java

-AtmenetJobboldalPDA.java

-AtmenetFuggvenyPDA.java

-PDAKonfiguracio.java

package modellek

Ez a csomag tartalmazza a számítási modelleket reprezentáló osztályokat.

-DTuringGep.java

-NDTuringGep.java

-PushdownAutomaton.java

default package

Ebben a csomagban a modellek tesztelésére használt példák találhatóak.

-TuringFrame.java
-PDAFrame.java

11.1.2. Megjegyzések

A használt adatszerkezetek törekednek a modellek minél általánosabban történő megadására. A Turing-gép szalagjai mind egy kitüntetett szimbólummal kezdődnek, aminek szerepe az író/olvasó fejek negatív irányba történő lefutásának megakadályozása. Mivel minden algoritmus átalakítható olyan formába, amely ezt teljesíti, ez nem jelent elvi megszorítást.

11.2. package elemek

11.2.1. Szimbolum.java

Általános szimbólumot képvisel, értéke Object típusú. Az egyszerűség kedvéért a példákban a szimbólumok értékei mind String típusúak. Két szimbólum egyenlőségének vizsgálata értékeik String formájának összehasonlításával történik, ezt a *getValueString()* metódussal kérhetjük le.
Két kitüntetett szimbólum a **szalagElejeSzimbolum**, amelynek értéke a **">"** String, és az **uresSzimbolum**, amelynek értéke **null**.

11.2.2. Abece.java

Szimbólumok halmazát reprezentálja, HashSet<Szimbolum> adatszerkezettel. A két kitüntetett szimbólum minden ábécének automatikusan az eleme. Létrehozáskor elemei

megadhatók HashSet<Szimbolum> és String[] adatszerkezetekkel. Az *eleme(Szimbolum sz)* metódus eldönti paraméteréről, hogy eleme-e az ábécének. Az *addSzimbolum(Szimbolum sz)*, *addSzimbolumok(HashSet<Szimbolum> sz)* és *removeSzimbolum(Szimbolum sz)* metódusok hozzáadnak az ábécéhez egy ill. több elemet, valamint eltávolítanak szimbólumot belőle.

11.2.3. Cella.java

Egy szimbólumot csomagol be. Értéke az egyszerűség kedvéért megadható közvetlenül String-gel is, és String formában lekérdezhető az *erteke()* metódussal.

11.2.4. Szalag.java

Egy ArrayList<Cella> típusú adatszerkezettel reprezentálja a szalagot és egy int típusú számmal az író/olvasó fej helyzetét. Létrehozás során az első eleme automatikusan a **szalagElejeSzimbolum**, és a (String vagy ArrayList<Cella>) formában megadott szimbólumsor után mindig **uresSzimbolum**-mal zár. A *fejBalraLep()* metódus ügyel arra, hogy ne menjen negatív értékbe, ekkor false értékkel tér vissza. A *fejJobbraLep()* metódus ha észleli, hogy a szalag végére ért, automatikusan kiegészíti egy **uresSzimbolum**-mal a szalag végén.

11.2.5. FejMozgas.java

Az író/olvasó fejek mozgását reprezentáló **enum** típus. Három eleme a **BALRA**, **HELYBEN**, **JOBBRA**. Metódusai számokat és betűket konvertálnak FejMozgas típussá és vissza: negatív szám vagy "b" betű esetén **BALRA**, nulla vagy "h" esetén **HELYBEN**, pozitív szám vagy "j" esetén **JOBBRA**. A kis- és nagybetűkre nem érzékeny.

11.3. package turingspec

11.3.1. TuringGep.java

Egy interfész, melyet a determinisztikus és nemdeterminisztikus Turing-gép modell is implementál. Négy metódust definiál, a *lepes()* metódust, amely a számítás egy lépése, az *elfogadas()* metódust, ami logikai értékkel tér vissza aszerint, hogy jelenleg elfogadó állapotban van-e a gép vagy sem,a *megallas()* metódust, amely szintén egy logikai értékkel megmondja, hogy a gép a jelenlegi helyzetében megáll-e vagy sem, valamint a *getKonfiguracio()* metódust, ami a gép jelenlegi konfigurációját adja vissza.

11.3.2. TGKonfiguracio.java

A Turing-gép egy konfigurációját reprezentálja, egy Cella segítségével az állapot megadására és egy Szalag[] szerkezettel a gép szalagjainak megadására. Az író/olvasó fejek helyzete közvetett módon adott, mivel a Szalagok belső tulajdonsága.

11.3.3. AtmenetBaloldal.java

A Turing-gép átmenetfüggvényének baloldalát reprezentáló osztály, egy Szimbolum példányt használ az állapot, és egy Szimbolum[] szerkezetet a beolvasott jelek megadására. Létrehozáskor paraméterei megadhatók két String-gel is, ekkor az első String az állapot lesz, a második String-et pedig karakterekre bontja és minden karaktert egy-egy beolvasott szimbólumnak vesz. Ekkor a beolvasott üres szimbólumot szóközzel lehet megadni. A *getJel(int i)* és *setJel(int i,Szimbolum sz)* metódusokkal az egyes beolvasott jelek különkülön is lekérhetők illetve felülírhatók.

11.3.4. AtmenetJobboldal.java

A Turing-gép átmenetfüggvényének jobboldalát reprezentáló osztály, az AtmenetBaloldal osztályhoz képest kiegészül egy FejMozgas[] szerkezettel, amely az író/olvasó fejek mozgatására vonatkozó információkat tartalmazzák. Létrehozáskor paraméterei három String-

gel adhatók meg, állapot-mozgatások-jelek sorrendben. A mozgatásokat ekkor a jelekhez hasonlóan értelmezi, a **b**,**h** és **j** betűket feleltetve meg a **BALRA**, **HELYBEN** és **JOBBRA** mozgásoknak. Metódusai a *getMozgas(int i)* és a*setMozgas(int i,FejMozgas f)* metódusokkal egészülnek ki, melyekkel az i. szalag mozgatására vonatkozó információ kérhető le és írható felül.

11.3.5. AtmenetFuggvenyNDTG.java

A nemdeterminisztikus Turing-gép átmenetfüggvénye. A szabályokat egy HashMap<AtmenetBaloldal,HashSet<AtmenetJobboldal» adatszerkezet adja meg. A függvény baloldalai tehát egyediek (kulcsok), amikhez egy jobboldal-halmaz tartozik. A *vanSzabaly(AtmenetBaloldal b)* metódus megadja, hogy az adott baloldalhoz tartozik-e szabály a táblázatban. Az *addSzabaly(AtmenetBaloldal b,AtmenetJobboldal j)* és *removeSzabaly(AtmenetBaloldal b,AtmenetJobboldal j)* metódusokkal egy szabály adható hozzá illetve vehető el a függvényből. Az *addSzabalyok(AtmenetBaloldal b,HashSet<AtmenetJobboldal> j)* és *removeSzabalyok(AtmenetBaloldal b,HashSet<AtmenetJobboldal> j)* metódusokkal közös baloldalú szabályok halmazával tehető ugyanez. A *getSzabalyok(AtmenetBaloldal a)* metódus visszaadja a megadott baloldalhoz tartozó összes szabályt. A *valaszt(AtmenetBaloldal b)* metódus véletlenszerűen választ egyet a megadott baloldalhoz tartozó szabályok közül, és annak jobboldalával tér vissza.

11.3.6. AtmenetFuggvenyDTG.java

Ez az osztály kihasználja a determinizmusból származó egyszerűsítést, adatszerkezete HashMap<AtmenetBaloldal,AtmenetJobboldal>, azaz minden baloldalhoz egyetlen jobboldal tartozik. Nem rendelkezik az egyszerre több közös baloldalú szabályt kezelő metódusokkal, valamint a *valaszt(AtmenetBaloldal b)* metódussal (hiszem a következő használható szabály mindig egyértelmű), ettől eltekintve működése egyezik az AtmenetFuggvenyNDTG osztállyal.

11.4. package pdaspec

11.4.1. PDAKonfiguracio.java

A veremautomata egy konfigurációját reprezentáló osztály. Tartalmaz egy Cella elemet az állapot tárolására, egy Szalag elemet az automata inputszalagjának tárolására, és egy Stack<Szimbolum> szerkezetet a verem tárolására.

11.4.2. AtmenetBaloldalPDA.java

Az osztály példányai a veremautomata átmenetfüggvényének baloldalát adják meg. Három Szimbolum-ot tartalmaznak, amely az állapotot, az input aktuális betűjét és a verem tetején lévő szimbólumot jelentik. Üres inputbetű esetén az **uresSzimbolum**-ot használja. Létrehozáskor szintén ebben a sorrendben kell őket megadni. A könnyebb használhatóság miatt a megadás történhet három String-gel is, ezekből létrehozza a nekik megfelelő Szimbolum-okat. Az *allapotVeremEgyezes(AtmenetBaloldalPDA a)* metódus logikai értékkel tér vissza, ami igaz, ha a paraméterként megadott AtmenetBaloldalPDA példány állapota és veremszimbóluma egyezik a hívóéval, és az inputbetű vagy szintén egyezik, vagy **uresSzimbolum**. Ennek az automata adott helyzetből való továbblépési lehetőségeinek megtalálásában van szerepe.

11.4.3. AtmenetJobboldalPDA.java

A veremautomata átmenetfüggvényének jobboldalát reprezentáló osztály. Egy Szimbolum-ot tartalmaz az állapot megadására, és egy Szimbolum[] szerkezetet a verem tetejére kerülő szónak. A megadása két String-gel is történhet, ekkor a második String-et karakterekre bontja és mindet a verem tetejére kerülő szó egy Szimbolum-aként veszi. A verembe kerülés sorrendje balról jobbra történik. Ha a verem tetejére nem kerül semmi, azt az **uresSzimbolum** Szimbolum-mal vagy **null** értékű String-gel lehet megadni.

11.4.4. AtmenetFuggvenyPDA.java

A veremautomata átmenetfüggvényét reprezentálja a Turing-géphez hasonló módon, egy HashMap<AtmenetBaloldalPDA,HashSet<AtmenetJobboldalPDA» adatszerkezettel. A nemdeterminisztikus Turing-gépnél használt metódusok itt is megjelennek, de paraméterként AtmenetBaloldalPDA illetve AtmenetJobboldalPDA példányokat használnak. A *getLehetosegek(AtmenetBaloldalPDA b)* metódus egy baloldal-halmazzal tér vissza, melynek elemei az AtmenetBaloldalPDA.java osztályban leírt *allapotVeremEgyezes()* metódus szerint megegyeznek a paraméterrel. A *vanFolytatas(AtmenetBaloldalPDA b)* metódus azt adja meg, hogy ez a halmaz üres-e vagy sem. A *getAtmenetek(AtmenetBaloldalPDA b)* metódus egy jobboldal-halmazzal tér vissza, amelyek a *getLehetosegek()* metódus által talált baloldalakhoz tartozó összes jobboldal uniója.

11.5. package modellek

11.5.1. NDTuringGep.java

Az osztály példányai nemdeterminisztikus Turing-gépek: adatai egy k szám (a szalagok száma), egy Szalag[], egy Abece az állapotok halmazának megadására, egy Abece a szalagábécé megadására, egy Cella az állapot tárolására, egy Szimbolum a kezdőállapot tárolására, egy Abece a végállapotok tárolására, és egy AtmenetFuggvenyNDTG példány az átmenetfüggvény megadására. Létrehozáskor lehetőség van egyes adatok elhagyására, ekkor az állapotot automatikusan a kezdőállapotnak állítja be, létrehoz k darab új szalagot, és a kezdő- és végállapotokat hozzáadja az állapotok halmazához ha esetleg nem szerepelnének benne. Az *initInput(String s)* metódussal adható meg a gép számára az input. Az *osszerakas()* metódus összeállít egy AtmenetBaloldal példányt a jelenlegi helyzet alapján. A *lepes(AtmenetJobboldal j)* metódus egyet lép a számítási folyamatban a paraméterként megadott AtmenetJobboldal objektumban találtaknak megfelelően. Az *elfogadas()* és *megallas()* metódusok a TuringGep interfészben specifikált módon működnek. A *lehetosegekSzama(AtmenetBaloldal b)* metódus megadja, hogy az adott baloldal-

hoz hány jobboldal tartozik az átmenetfüggvényben. A *szamitas(TGKonfiguracio tk)* metódus a megadott konfigurációtól kezdve bejárja a teljes számítási fát, és a igaz értékkel tér vissza, ha talált elfogadó számítást, azaz a gép elfogadta az inputot. A gép jelenlegi konfigurációja a *getKonfiguracio()* metódussal kérhető le. A *szamitasLogged(TGKonfiguracio tk, ArrayList<TGKonfiguracio> log)* metódus a számítás folyamatában felmerülő konfigurációkat a log paraméterként megadott listába menti.

11.5.2. DTuringGep.java

A determinisztikus Turing-gépet reprezetáló osztály adatai egyedül az átmenetfüggvényben térnek el a nemdeterminisztikustól. (egy AtmenetFuggvenyDTG példányt használ az AtmenetFuggvenyNDTG helyett). Megadása egyezik a nemdeterminisztikus gépével. A *lepes()* metódus ez esetben determinisztikusan működik, a *szamitas()* metódus pedig nem igényel paramétert, egyszerűen a gép jelen állásától kezdve végigmegy a determinisztikus számításon, és az elfogadás/elutasításnak megfelelően tér vissza logikai értékkel. A *szamitasLogged(ArrayList<TGKonfiguracio> log)* metódus a determinisztikus számítás során létrejövő minden konfigurációt elment a log paraméterként megadott listába. Ezen különbségektől eltekintve működése egyezik a nemdeterminisztikus gépével.

11.5.3. PushdownAutomaton.java

A nemdeterminisztikus veremautomatát reprezentáló osztály. Adatai egy Cella az állapot tárolására, egy Abece a veremábécé megadására, egy Abece az állapothalmaz megadására, egy Abece az inputábécé megadására, egy AtmenetFuggvenyPDA az átmenetfüggvénynek, egy Szimbolum a kezdőállapotnak, egy Abece az végállapotok halmazának, egy Szalag az inputszalag számára és egy Stack<Szimbolum> a veremnek. A létrehozás során lehetőség van egyes elemek elhagyására, ekkor a kezdőállapotban jön létre a gép,üres inputszalaggal és veremmel, (egyetlen eleme a **szalagElejeSzimbolum**,a működés során a gép feltételezi, hogy a verem legalsó eleme mindig ez a szimbólum), valamint kezdő- és végállapotokat automatikusan hozzáadja az állapotok halmazához ha az nem tartal-

mazná azokat. Az *initInput(String s)* metódussal adható input a gépnek. Az *elfogadas()*, *megallas()* metódusok a nemdeterminisztikus Turing-gépnél megadott módon viselkednek. A *lepes(AtmenetBaloldalPDA b,AtmenetJobboldalPDA j)* metódus a paraméterekben megadottak szerint lépteti tovább a gépet. A következő metódusok a számítási folyamat implementálásának megkönnyítése miatt szerepelnek: A *baloldalValasztas()* metódus véletlenszerűen választja ki egy alkalmazható szabály baloldalát. Az *allas()* metódus szintén egy baloldalt ad vissza, a gép jelenlegi állásának megfelelőt. A *lehetosegekSzama()* metódus megadja, hogy a jelenlegi helyzetben alkalmazó szabályok halmazában hányféle különböző baloldal található (0, 1, vagy 2). A számítási folyamatot a *szamitas(PDAKonfiguracio pk)* metódus hajtja végre, amely a megadott konfigurációtól kezdve bejárja a teljes számítási fát, és az elfogadásnak/elutasításnak megfelelő logikai értékkel tér vissza. A *getKonfiguracio()* metódus itt is a gép jelenlegi konfigurációjával tér vissza.