

1. A tárgyról

Ezen a kurzuson **problémákat** fogunk megtanulni „bonyolultság” szerint osztályozni. Más kurzusokon, mint pl. az *Algoritmusok és adatszerkezetek*en, nem magukat a problémákat, hanem az azokat megoldó **algoritmusokat** vizsgáljuk olyan szempontból, hogy vajon az adott algoritmus minden lehetséges bemeneten megáll-e, a specifikációnak helyes outputot ad-e vissza, mindeközben mennyi az időigénye (lépésszáma egy n méretű inputon, n függvényében), vagy a tárigénye.

Például, hallhattunk olyat már, hogy egy n méretű int tömb **rendezése buborékredezéssel** legrosszabb esetben n^2 lépésszámot (=időt) igényel, **gyorsrendezéssel** átlagosan $n \log n$ -et, legrosszabb esetben n^2 -est, **összefésülő rendezéssel** legrosszabb esetben is $n \log n$ -et, ha (mondjuk) d -bites inteket rendezünk, akkor **radix sort**tal dn időben végzünk, vagy ha tudjuk, hogy az intjeink a $[0, N]$ intervallumba esnek, akkor a **leszámláló rendezés** $N + n$ időigényű rajta, nagyságrendben. Az **algoritmusok** tárgyban számos konkrét algoritmusnak vizsgáljuk ilyen módon a hatékonyságát.

A **bonyolultságelméletben** a kérdés inkább az lenne, hogy maga a **rendezési probléma** „milyen nehéz”, vagy első közelítésben: a leghatékonyabb rendezési algoritmusnak mennyi az időigénye? (A fentiek szerint legfeljebb $n \log n$, de vajon lehet-e lineáris időben rendezni?) Tudunk-e adni olyan függvényt, ami *alsó korlátot* ad a legjobb algoritmus időigényére? (Nyilván mivel minden tömbelemet meg kell nézzünk, n lépés mindenképp kell. Lehetséges, hogy $n \log n$ is szükséges, az alá nem lehet menni általános rendező algoritmussal?)

A fenti kérdések ugyan egyszerűnek **hangzanak**, de cseppet sem azok: ha ilyen szinten próbáljuk vizsgálni a problémák nehézségét, nagyon komoly akadályokba ütközünk, jelenleg **nincsenek** igazán jó (matematikai) **eszközök**, melyekkel ennyire precízen meg tudnánk határozni a „legjobb algoritmus” idő- vagy tárigényét, például azért sem, mert nincsenek jó **alsó korlát**-módszerek, amikkel olyan állításokat be tudnánk látni, hogy „ennek a problémának a megoldására **legalább** ennyi idő feltétlenül szükséges”.

Vegyük például az $n \times n$ -es mátrixok **összeszorzásának kérdését** (vagy problémáját): a **triviális algoritmus** n^3 művelettel meghatározza két ekkora mátrix szorzatát. Elsőre az ember azt gondolná, ennél kevesebb művelet **„egyszerűen nem lehet elég”** a szorzat kiszámítására, de pl. a **Strassen-algoritmus** mégis kb. $n^{2.8074}$ lépésben kiszámítja ezt. Még olyan „egyszerű” problémákról **sem ismert egy precíz korlát**, mint a mátrixszorzás: annyit tudunk, hogy létezik olyan algoritmus, mely $n^{2.3728639}$ -cel arányos lépésszámban kiszámítja két mátrix szorzatát (a **Coppersmith-Winograd** algoritmus egy 2014-es változata), a triviális alsó korlát pedig, hogy **legalább n^2 lépés kell** (ennyi cellája van egy $n \times n$ -es mátrixnak), de a kettő közötti gap még mindig jelen van. Itt is az a helyzet, mint a rendezési problémánál: nagyon nehéz **nemtriviális alsó korlátokat** adni problémák megoldásának időigényére. További kérdés az **architektúra**, amire az alsó korlát vonatkozik, van-e egyáltalán tömbcímezés benne, nagy egész számok szorzását-összeadását egy lépésnek tekinthetjük-e, stb.

A **Bonyolultságelmélet kurzuson** nem kísérünk meg ennyire pontosan „nehézséget” tulajdonítani problémáknak, hanem (a tárgy első felében) a következő **három kategóriába** próbáljuk meg besorolni őket: egy probléma lehet

1. **elméletileg is megoldhatatlan**: ezek azok a problémák, melyekre nincs olyan algoritmus (a „nincs” nálunk nem azt jelenti, hogy „még nincs”, hanem azt, hogy matematikai-

lag is igazolható, hogy **nem is lehet készíteni**), mely minden inputra mindig véges sok lépésben megáll, és mindig helyes választ ad;

2. **gyakorlatilag megoldhatatlan**: ezek azok a problémák, melyekre van ugyan megoldó algoritmus, de jelen ismereteink szerint még a legjobb algoritmus időigénye is „túl sok”;
3. **gyakorlatilag megoldható**: ezekre pedig létezik hatékony megoldó algoritmus.

Az első kategóriát a másik kettőtől elválasztó határvonal éles: egy probléma vagy megoldható elméletileg, vagy nem, nincs köztes út. (Habár azon el lehet gondolkodni, hogy vajon lehetséges-e, hogy egy probléma valamilyen architektúrán vagy **valamilyen programozási nyelven megoldható, egy másikon pedig nem az?**) Technikailag ezzel a kérdéssel a **kiszámíthatóság-elmélet** foglalkozik, de ilyen nevű kurzusunk nem lesz; a Bonyolultságelmélet kurzus keretein belül néhány előadás erejéig tanulni fogunk olyan módszereket, melyekkel egyes problémákról **be fogjuk tudni bizonyítani, hogy megoldhatatlanok**.

Onnan kezdve viszont, hogy egy problémáról tudjuk, hogy (elméletileg) megoldható, persze rögtön felmerül a kérdés, hogy **mi számít „hatékony” algoritmusnak**, vagy mi számít „túl sok” időigénynek? Amíg erre nincs „objektív” fogalmunk, addig nem tudunk egy megoldható (ebben a tárgyban a megoldható problémákat **eldönthető**, vagy **rekurzív** problémának is fogjuk hívni) problémát besorolni a második vagy harmadik kategóriák egyikébe sem. Adott környezetben pl. lehetséges, hogy akkora nagy inputokon kell dolgoznunk (ilyen pl. a DNS szekvenciák vizsgálatakor előfordulhat), hogy még egy $n \log n$ -es algoritmus is túl sokáig futna, más esetekben pedig csak olyan kisméretű inputokra kell megoldjuk a problémát, hogy egy n^3 -ös vagy akár egy 2^n -es algoritmus is „időben” lefut.

Mielőtt tehát beszélni tudnánk arról, hogy valami megoldható-e egyáltalán, és ha igen, akkor hatékonyan-e vagy sem, előbb le kell rögzítenünk, hogy ezek a szavak (matematikailag) mit jelentenek, erről szól az első előadás.

2. Kiszámíthatóság- és bonyolultságelmélet

Az 1900-as évben David Hilbert a Matematikusok Nemzetközi Kongresszusán olyan matematikai problémákról tartott előadást, melyeket a XX. század matematikusai számára egyfajta „célként” tűzött ki mint fontos megoldandó problémákat. Hilbert problémái (a kongresszuson tízről beszélt, később egy 23 elemű lista jelent meg nyomtatásban) közül számos a mai napig megoldatlan, a megoldottak közül pedig többnek a megoldása is egészen új matematikai irányvonalak kifejlődését tette szükségessé.

A problémák közül a 10. számú, **Hilbert tizedik problémája** a mi terminológiánkban a következőképp szól:

Definíció: Hilbert 10. problémája

Adjunk meg egy olyan algoritmust, melynek inputja egy többváltozós egész együtthatós polinom, outputja pedig **igen**, ha a polinomnak van egész értékű zérushelye, **nem** egyébként.

Például, egy ilyen algoritmusnak az $2x^2 + 2xy + 2x + y^2 + 1$ polinomra **igen** választ kell adnia, hiszen $x = -1$, $y = 1$ egy zérushely; a $4x^2 + 4x + 1$ polinomra pedig **nem** választ, mert ennek egyetlen zérushelye a valósok körében az $x = -0.5$, ami nem egész.

Hilbert kérdésfelvetésének megfogalmazásából nem világos, számolt-e ő azzal a lehetőséggel, hogy esetleg ilyen algoritmus nincs (erről a matematikatörténetesek közt sincs teljes egyetértés). Mindenesetre ilyen módszert találni sokáig nem sikerült. Hilbert 1928-ban egy rokon problémát is felvetett:

Adjunk meg egy olyan algoritmust, melynek inputja egy elsőrendű logikai formula, outputja pedig **igen**, ha a formula tautológia, **nem** egyébként.

Szintén nem világos, hogy Hilbert a kérdés megfogalmazásakor számolhatott-e azzal, hogy esetleg ilyen algoritmus sincs.

Mindenesetre egyik problémára sem sikerült sokáig algoritmust találni, elegendően sokáig ahhoz, hogy felmerüljön a kérdés:

Tulajdonképpen mit jelent az, hogy valami (algoritmikusan) kiszámítható?

Hiszen ahhoz, hogy olyan állítást tudjunk igazolni, hogy „erre a problémára nincs megoldó algoritmus”, először meg kell állapodnunk abban, hogy mit tekintünk egyáltalán „megoldó algoritmus”-nak, milyen műveleteket engedünk meg és hogyan kombinálhatjuk őket.

Erre az első javaslatot Kurt Gödel tette 1931-ben, a primitív rekurzív függvényeket tekintette kiszámíthatónak. Ezek a függvények természetes számokat várnak argumentumként és a következőképp definiáljuk halmazukat:

Definíció: Primitív rekurzív függvények

- A **konstans 0** függvény: $0(n) := 0$ primitív rekurzív.
- A **rákövetkezés** függvény: $s(n) := n + 1$ is primitív rekurzív.
- A **projekció** függvények, melyek visszaadják valahányadik argumentumukat: $\pi_k^i(x_1, \dots, x_k) := x_i$ is primitív rekurzívak.
- Primitív rekurzív függvények **kompozíciója** is primitív rekurzív: ha f egy n -változós primitív rekurzív függvény, g_1, \dots, g_n pedig k -változós primitív rekurzív függvények, akkor a $h = f \circ \langle g_1, \dots, g_n \rangle$ összetett függvény egy k -változós rekurzív függvény: $h(x_1, \dots, x_k) := f(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$.
- Ha f egy k -változós és g egy $k + 2$ -változós primitív rekurzív függvény, akkor a következő h , $k + 1$ -változós függvény is az:

$$h(x_1, \dots, x_{k+1}) := \begin{cases} f(x_1, \dots, x_k) & , \text{ ha } x_{k+1} = 0; \\ g(x_1, \dots, x_k, x_{k+1} - 1, h(x_1, \dots, x_k, x_{k+1} - 1)) & , \text{ egyébként.} \end{cases}$$

Más primitív rekurzív függvény nincs.

Az utolsó konstrukció maga a **primitív rekurzió**. Az első négy esetről mindenki elfogadja, hogy ezek „kiszámítható” függvények (és nem utolsó sorban: véges idő alatt kiszámíthatóak), a primitív rekurzió esetén pedig elég azt meggondolnunk, hogy a h függvény utolsó paraméterének értéke minden egyes „hívásnál” (vagy kiértékelésnél) csökken eggyel, végül 0 lesz, ekkor az f függvényt kell kiszámítsuk, majd letről felfelé a g függvényt x_{k+1} -szer, a feltevés szerint ezek

mind kiszámíthatóak. (Azaz, az algoritmusok terminológiájával, az x_{k+1} értéke a rekurziónak egy **leszállási feltétele**.)

Lássunk pár **példát**.

Ha szeretnénk pl. előállítani az $(x, y, z) \mapsto z + 1$ függvényt, akkor ez két függvény kompozíciójaként áll elő: először az (x, y, z) hármastól kiválasztjuk a z -t, a háromból a harmadik argumentumot, ez a π_3^3 projekciós függvény, majd ezen az eredményen alkalmazzuk az egyváltozós rákövetkezés s függvényt, azaz az $s \circ \pi_3^3$ pontosan ez a függvény.

Ha a primitív rekurzió képzésekor az $f = \pi_1^1$ egyváltozós projekciós függvényből indulunk ki (azaz az $f(x) = x$ identikus függvényből), g függvényünk pedig a $g(x, y, z) = s \circ \pi_3^3$, vagyis a $z + 1$ függvény, akkor ezekből primitív rekurzióval kapjuk azt a $h(x, y)$ függvényt, melyre $h(x, 0) = x$ és $h(x, y + 1) = g(x, y, h(x, y)) = h(x, y) + 1$. Ez a függvény az **összeadás**, jelölje mondjuk $\text{add}(x, y)$.

Hasonlóan tudunk **szorzást** is előállítani: ekkor $f(x) = 0$ az (egyváltozós) konstans 0 függvény, a g háromváltozós függvény pedig legyen $g(x, y, z) = \text{add}(x, z)$ (ez előállítható ismét két projekcióval: ha kirészletezzük, akkor az $\text{add} \circ \langle \pi_3^1, \pi_3^3 \rangle$ függvényről van szó), ezekből primitív rekurzióval a $h(x, 0) = 0$, $h(x, y + 1) = g(x, y, h(x, y)) = x + h(x, y)$ függvényt kapjuk, ez pedig a **szorzás**.

Teljesen analóg módon tudunk előállítani **hatványozást**, **faktoriált**, vagy akár **egyenlőség tesztelést** (mely 1-et ad, ha két argumentuma megegyezik, ellenkező esetben 0-t), **kivonást** (azzal pl, hogy ha $x < y$, akkor $x - y = 0$, hogy maradjunk a természetes számok halmazán), **egészosztást**, **maradékképzést** stb.

Azt látjuk tehát, hogy a primitív rekurzív függvényekkel egyrészt csupa intuitíve „kiszámítható” függvényt tudunk előállítani (hiszen az alapfüggvényekről mindről úgy gondoljuk, hogy ezek egyszerűen kiszámíthatóak, a két összetételi műveletről pedig szintén látjuk, hogy „ha ezek a függvények mind kiszámíthatóak, akkor ez az eredmény is”). A kérdés, hogy

vajon van-e olyan függvény, amit „intuitíve” kiszámíthatónak „érezünk”, **de mégse áll elő ilyen alakban?**

Mint kiderült, van, pl. az alábbi $A(m, n)$ ún. **Ackermann-függvény**:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)). \end{aligned}$$

Például $A(1, 1) = A(0, A(1, 0)) = A(0, A(0, 1)) = A(0, 2) = 3$ és $A(2, 2) = A(1, A(2, 1)) = A(1, A(1, A(2, 0))) = A(1, A(1, A(1, 1))) = A(1, A(1, 3)) = A(1, A(0, A(1, 2))) = A(1, A(0, A(0, A(1, 1)))) = A(1, A(0, A(0, 3))) = A(1, A(0, 4)) = A(1, 5) = A(0, A(1, 4)) = A(0, A(0, A(1, 3))) = A(0, A(0, 5)) = A(0, 6) = 7$. Úgy általában $A(1, n) = n + 2$ és $A(2, n) = 3 + 2n$, de tovább növelve az első argumentumot már nem lineáris függvényeket kapunk: $A(3, 0) = A(2, 1) = 5$ -ből és $A(3, n + 1) = A(2, A(3, n)) = 3 + 2 \cdot A(3, n)$ -ből azt kapjuk, hogy $A(3, n) = 2^{n+3} - 3$. Ezek után pl. $A(4, 0) = A(3, 1) = 2^4 - 3$, $A(4, 1) = A(3, A(4, 0)) = 2^{2^4} - 3$,

$A(4, 2) = 2^{2^4} - 3$ és így tovább, $A(4, n)$ -nél már $n + 1$ *darab* 2-es van egymásra „tornyozva”! Ezek nagy számok már ilyen kis értékekre is, pl. $A(4, 1) = 65533$ és $A(4, 2) = 2^{65536} - 3$. Az első paramétert tovább növelve még gyorsabban fog növekedni a függvény. . . ez érezhetően olyan típusú függvény, melyet **gyakorlatilag** nem akarunk kiszámíthatónak minősíteni (hiszen pl. már az $A(5, 1)$ leírásához bitenként egyetlen elektront felhasználva is több atom kellene, mint amennyi az ismert univerzumban létezik), ugyanakkor **elméletileg**, végtelen mennyiségű erőforrást feltételezve (időt is, tárat is) a függvény kiszámítható, „csak” iterálni kell a fenti helyettesítést. (Mivel a paraméterek lexikografikusan csökkennek, azaz vagy az első argumentum csökken, vagy az ugyanannyi marad és a második csökken, így garantált, hogy nem eshetünk végtelen ciklusba.)

Pontosan ezt a szinte felfoghatatlanul gyors növekedést kihasználva igazolható, hogy az Ackermann-függvény **nem primitív rekurzív**. Ami azt is jelenti, hogy a primitív rekurzív függvények **nem felelnek meg a(z elméleti) „kiszámíthatóság” fogalmának**, abba ez a függvény is bele kell férjen.

Ezért vezette be Gödel 1934-ben az **általános rekurzív függvényeket**: a képzési szabályok ugyanazok, mint a primitív rekurzívaké, plusz:

Ha f egy $k + 1$ -változós általános rekurzív függvény, akkor az a h k -változós függvény is az, melyre $h(x_1, \dots, x_k)$ értéke az a legkisebb y , melyre $f(y, x_1, \dots, x_k) = 0$.

Ha létezik ennek az $f(_, x_1, \dots, x_k)$ függvénynek zérushelye. Ha nem létezik, akkor $h(x_1, \dots, x_k)$ definiálatlan, ezek a függvények **parciálisak**.

Az Ackermann-függvény már általános rekurzív függvény.

Természetesen nem csak Gödel próbálta megfogni matematikailag a „kiszámítható függvény” intuitív fogalmát. A teljesség igénye nélkül, az 1930-as évek elején Church, Kleene és Rosser a Princeton-i Egyetemről szintén bevezettek egy formalizmust Gödeltől függetlenül, az ún. **λ -kalkulust**. (A λ -kalkulus egyébként a **funkcionális programozási nyelvek** alapja.) Alan Turing pedig a **Turing-gépet**.

Ez három **teljesen különböző**képp (és más célokra) definiált matematikai formalizmus arra, hogy mi is az, hogy valami „kiszámítható” . . . és mégis, **ez a három modell ekvivalens** abban az értelemben, hogy ami kiszámítható az egyikben, az kiszámítható a másik kettőben is! Ezt 1936-ban bizonyította be Church (azt a részt, hogy a λ -definiálhatóság megegyezik az általános rekurzíóval) és Turing (azt, hogy a λ -definiálhatóság megegyezik a Turing-géppel való kiszámíthatósággal). Ezek az ekvivalenciák **tételek**: matematikai objektumokról állítanak valamit, amit formálisan be is lehet bizonyítani.

Azt, hogy „mi is az számunkra, hogy valami kiszámítható”, nem lehet tételként bebizonyítani, mert nem tisztán matematikai fogalomról beszél, hanem egy intuitív fogalomra próbál adni egy matematikai modellt, ezt **tézis**nek hívjuk.

Definíció: Church-Turing tézis

A **Church-Turing tézis** azt mondja ki, hogy az általános rekurzív függvények pontosan azok, melyeket „algorithmikusan kiszámítható”-nak tartunk^a.

^alásd még a wikit

Ez persze azt is jelenti, hogy a Turing-gép és a λ -definiálhatóság is pontosan ragadja meg az algorithmikus kiszámíthatóságot:

A **Church-Turing tézist** úgy is kimondhatjuk, hogy a Turing-géppel kiszámítható függvények pontosan azok, melyeket „algoritmikusan kiszámítható”nak tartunk.

A tézist addig fogadjuk el, míg valaki nem mutat egy olyan függvényt, mely intuitíve „kiszámítható”, de mégsem általános rekurzív. (Ahogy ez történt pl. az Ackermann-függvénnyel: az kiszámítható, de nem primitív rekurzív, ezért egy másik, bővített modellt kellett definiálni, melyben az Ackermann-függvény is beletartozik.) Mindenesetre a tézis 1936-os megszületése óta **ilyen függvényt senki nem tudott mondani**, ezért a Church-Turing tézist széles körben elfogadottnak tekintjük.

Mivel így az 1930-as évek közepétől már volt egy matematikai fogalom arra, hogy mi is az, hogy valami „kiszámítható”, így már volt rá esély, hogy valaki bebizonyítsa egy problémáról, hogy az nem kiszámítható. Hilbert tizedik problémája és a később megfogalmazott kérdése ilyenek:

Állítás: (Church 1936, Turing 1937)

Nem létezik olyan algoritmus, mely eldöntené, hogy egy input elsőrendű logikai formula tautológia-e.

Állítás: (Matijasevič, 1971)

Hilbert tizedik problémája algoritmikusan megoldhatatlan.

A kiszámíthatóság elméletéhez képest a **bonyolultságelmélet** azzal foglalkozik (kb), hogy megoldható-e egy probléma **korlátozott mennyiségű erőforrással**, a fő motiváció elkülöníteni a megoldható problémák közül a **gyakorlatilag** megoldhatóakat.

Ismét meg kell ehhez egyezni abban, hogy

mit is tekintünk gyakorlatilag megoldhatónak?

Mivel matematikai elméletről van szó, nem lenne szerencsés, ha olyan definícióját adnánk a „gyakorlatilag megoldható” problémának, ami megváltozik, ahányszor kijön egy új processzor, gyorsabb RAM vagy egy nagyobb hard drive.

Először is, hogy egy **algoritmus** hatékonynak számít-e vagy sem, arra is van több megközelítés (függhet attól is, hogy párhuzamos architektúrán dolgozunk-e, vagy attól is, hogy jellemzően mennyi adaton), de a legelterjedtebb az ún. Cobham-Edmonds féle tézis:

Definíció: Cobham-Edmonds tézis

A Cobham-Edmonds tézis szerint

- egy algoritmus akkor számít hatékonynak, ha polinom idejű,
- egy probléma pedig akkor oldható meg hatékonyan, ha létezik rá hatékony megoldó algoritmus.

Itt a „polinom idejű” azt jelenti, hogy van egy olyan p polinom, melyre igaz, hogy az algoritmus tetszőleges n -bites (!) inputon legfeljebb $p(n)$ **időben megáll** (helyes válasszal). Ezt

legrosszabbeset-analízisnek nevezik: ha az összes n -bites input közül csak egyre (vagy csak néhányra) fut az algoritmus (mondjuk) 2^n ideig, az összes többire pedig mondjuk n lépésben megáll, az algoritmus akkor sem polinomidejű, mert az a 2^n nem korlátozható felülről egy polinommal sem: egy exponenciális függvény mindig gyorsabban nő, mint egy polinom.

Azért, hogy algoritmusainknak ne kelljen kiszámítanunk precízen az időigényét és hogy pl. lásuk, hogy „az exponenciális függvény mindig gyorsabban nő”, felidézzük a függvények aszimptotikus, vagy nagyságrendi összehasonlítására szolgáló O , Ω , o , ω és Θ jelöléseket.

Definíció: O , Ω , Θ , o , ω

Ha f és g (monoton növekvő) függvények, akkor mondjuk, hogy

- $f = O(g)$, ha van olyan $c > 0$, amire $f(n) \leq c \cdot g(n)$ majdnem minden n -re;
- $f = \Omega(g)$, ha $g = O(f)$;
- $f = \Theta(g)$, ha $f = O(g)$ és $g = O(f)$;
- $f = o(g)$, ha minden $c > 0$ -ra $f(n) \leq c \cdot g(n)$ teljesül majdnem minden n -re;
- $f = \omega(g)$, ha $g = o(f)$.

Itt a „majdnem minden n -re” azt jelenti¹, hogy „létezik olyan $N > 0$ küszöbszám, hogy minden $n > N$ -re”, ami ugyanaz, mint ha azt mondanánk, hogy csak véges sok n -re nem igaz, de az elég nagy n -ekre már mindre igen.

Ezek a jelölések rendre megfelelnek kb. a \leq , \geq , $<$ és $>$ relációknak függvények közt. Igaz, hogy **nem minden** függvényt lehet ezek valamelyikével összehasonlítani, de azok a függvények, melyekkel dolgozni fogunk, kevés kivételtől eltekintve páronként összehasonlíthatóak lesznek majd ezen relációk mentén. Aki tud határértéket számítani², annak hasznos a következő állítás:

Állítás

- Ha $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, akkor $f = o(g)$.
- Ha $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, akkor $f = O(g)$.

Így például ha p és q két polinom (pozitív főegyütthatókkal), akkor

- $p = O(q)$ pontosan akkor igaz, ha p foka legfeljebb akkora, mint q foka;
- $p = \Theta(q)$ pontosan akkor, ha fokaik megegyeznek.

Továbbá, (mondjuk) L'Hopital-szabállyal belátható, hogy ha p polinom, $a > 1$ pedig konstans, akkor $p = o(a^n)$, azaz **egy exponenciális függvény majdnem minden n -re nagyobb, mint egy polinom.**

¹igen, ez egy létező matematikai fogalom

²lásd kalkulus

Egy kicsit formálisabban a Cobham-Edmonds tézis tehát azt mondja, hogy egy algoritmus akkor hatékony, ha $O(p)$ időigényű valamely p polinomra.

Persze ezzel a tézissel (mint minden tézissel) lehet vitatkozni:

- Egy n^{100} időigényű algoritmus valójában nem mondható hatékonnak.

Ez jogos. Azonban, a **gyakorlatban előforduló** problémákra az a jellemző, hogy amire sikerül polinomidejű algoritmust találni, annak a polinomnak a fokszáma is kezelhető (vé válik némi optimalizálás után). Például a lineáris programozás³ feladatára a közismert szimplex módszer (1947) exponenciális időigényű, mégis a gyakorlatban előforduló méretű feladatokra gyorsabban futott, mint az ún. ellipszoid módszer (1979), mely kb. n^6 időigényű volt. Azonban a később kitalált ún. projektív módszer (1984) variánsai már a kb. $n^{3.5}$ időigényükkel felveszik a versenyt a szimplex variánsokkal a gyakorlatban is. (Mivel egy polinom „rendje” a fokszáma, ezt mondják úgy is, hogy „a gyakorlatban előforduló polinom időben megoldható problémák rendje általában kicsi”).

- Az O jelölés által „elrejtett” **együtthatók nagysága** is számíthat, pl. egy $10^{23}n$ időigényű algoritmus a gyakorlatban szintén nem hatékony.

Ez is jogos ellenvetés, azonban ez sem kifejezetten jellemző. Tény, hogy ez a probléma már a gyakorlatban is előfordul, pl. a korábban említett Coppersmith-Winograd mátrixszorzó algoritmust éppen azért nem használják a valóságban, mert akkora együtthatók jelennek meg az időigényben, hogy a gyakorlatban előforduló méretű mátrixokra az *aszimptotikusan* lassabb Strassen-algoritmus gyorsabb. Továbbá az is igaz, hogy „élég kicsi” mátrixokra a naiv köbös algoritmus a leggyorsabb. Vannak területek, ahol két azonos nagyságrendű, pl. két köbös algoritmus közt kell különbséget tenni, ilyen pl. a mátrixok felbontásai⁴; pl. a Cholesky felbontás időigényét ezért adják meg a legtöbb helyen $\frac{1}{3}n^3 + O(n^2)$ alakban, ez így több információt tartalmaz, mintha csak annyit mondanának róla, hogy $O(n^3)$ -ös; az LU-felbontás időigényéhez képest, ami $\frac{2}{3}n^3$, a Cholesky felbontás ebben az értelemben pl. „majdnem kétszer gyorsabb”, a mi terminológiánkban meg mindkettő $\Theta(n^3)$ -ös algoritmus lévén „egyforma jók” lennének. Léteznek tehát erre megoldások, de ezen a bevezető kurzuson megelégszünk az aszimptotikus nagyságrenddel.

- A legrosszabb eset-analízis nagyon pesszimista, a **várható érték** vizsgálata is indokolt lehet.

Ez is jogos. A legrosszabb eset-korlát mindig egyfajta „garanciát” szolgáltat: garantálja, hogy bárki is ad egy n -bites inputot, az algoritmus garantáltan lefut $f(n)$ lépésben. Van számos olyan kutatás, mely egyfajta várhatóérték-analízist végez: felteszik, hogy minden n -bites input egyforma valószínűséggel érkezik és kiátlagolják a futásidőket. Ez a megközelítés is hasznos lehet, azonban ennek is vannak hátulütői: egyrészt még egyszerű feladatok esetén is nagyon nehézé válik kiszámítani a várható értéket, még nagyságrendben is, másrészt ha a valóságot akarjuk modellezni, akkor nem indokolt feltételezni az egyenletes eloszlást az inputban. Viszont a tényleges eloszlás modellezése a legtöbbször lehetetlen feladat.

- A polinomidőben megoldható problémák egy része **hatékonyan párhuzamosítható**, más része pedig nem, legalábbis vannak olyan hatékonyan megoldható problémák, melyekre nem sikerült még „jó” párhuzamos algoritmust megadni.

³lásd opkut

⁴lásd köszi

Ez is jogos. A kurzus végén érintőlegesen **foglalkozunk párhuzamosíthatósággal is**, hiszen erre napjainkban, a GPGPU-programozás egyre szélesebb körben való elterjedésével nagy igény mutatkozik. Sokan úgy képzelik a „párhuzamosítás”t, hogy batch taskokat szétosztanak a konstans sok egyforma processzor között és általában kihozzák, hogy négy processzorra lehet szétosztani „optimálisan” a feladatot, majdnem négyszeresére gyorsítva a feladatmegoldást; a bonyolultságelmélet kurzuson a párhuzamosítással (megfelelően sok processzort feltételezve) **tényleges nagyságrendi** párhuzamosításra látunk majd példákat, pl. n -ről $(\log n)^2$ -re javítva egyes algoritmusok időigényét.

3. Architektúrák: RAM-gép és Turing-gép

Ezen a kurzuson két architektúrán fogunk algoritmusokat vizsgálni, **Turing-gépen** és **RAM-gépen**. A két számítási modell ekvivalens, nem csak számítási kapacitásukat tekintve (mindkettővel ugyanazt lehet elvégezni, mint pl. az általános rekurzív függvényekkel vagy λ -kalkulussal), de a felhasznált erőforrásokat (idő, tár) tekintve is⁵. A Turing-gép történetileg érdekes, és a „ezt a problémát **nem lehet** megoldani” típusú állításokat Turing-gépekkel könnyebb bebizonyítani (azért, mert a Turing-gépnek **nagyon** kevés elemi utasítása van). A RAM-gép, és az azon keresztül definiált pszeudokódok, strukturált imperatív programozás közelebb áll egy mai tényleges program struktúrájához, ezzel az „ezt a problémát **így lehet** megoldani” típusú állításokat könnyebb bebizonyítani.

3.1. Turing-gép

Egy (egyszalagos, determinisztikus) Turing-gépet a következőképp képzelünk el: adott egy **cellákra osztott, egy irányba végtelen szalag**. Egy cellába egy Γ ábécé⁶ egy-elemét írhatjuk, ez a **szalagábécé**, mely tartalmazza a \triangleright (háromszög, startjel) és a \sqcup (blank, space) szimbólumokat. Az inputot, mely egy $\Sigma \subseteq \Gamma - \{\triangleright, \sqcup\}$ **input ábécé** fölötti szó, a gép erre a szalagra írva kapja, a következőképpen: ha az input szó az $a_1 a_2 \dots a_n \in \Sigma^*$, akkor a végtelen szalag első cellájába kerül a \triangleright jel, a következőbe a_1 , a következőbe a_2 , \dots , majd az input utolsó betűjét a_n -t követően az összes többi cellába a \sqcup jel kerül. Pl. ha $\Sigma = \{0, 1\}$ a bináris ábécé, és 1101 az input, akkor a szalagtartalom iniciálisan $\boxed{\triangleright} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{\sqcup} \boxed{\sqcup} \boxed{\sqcup} \boxed{\sqcup} \dots$

A gép rendelkezik még egy **író-olvasófejjel**, ami egyszerre mindig pontosan egy cella fölött áll a szalagon, és az ott levő szimbólumot tudja elolvasni, majd átírni és lépni egyet balra, jobbra vagy helyben maradni. Iniciálisan a fej a szalag legelején lévő (\triangleright jelet tartalmazó) cellán áll.

A gépnek van továbbá egy véges **állapothalmaza**, jelölhetjük mondjuk Q -val, ezek közül van egy kitüntetett $s \in Q$ **kezdőállapota**. A gép számítás közben mindig pontosan egy állapotban van, iniciálisan az s -ben.

A gép működését az **átmenetfüggvénye** vagy **átmenettáblázata** határozza meg: ez egy $\delta : Q \times \Gamma \rightarrow (Q \times \Gamma \times D) \cup \{\text{ACCEPT}, \text{REJECT}, \text{HALT}\}$ leképezés, ahol $D = \{\leftarrow, -, \rightarrow\}$ a **mozgási irányok** halmaza. Ezt a következőképp alkalmazzuk: ha a számítás egy pillanatában a gép a q állapotban van és az olvasófej alatti cellában az a szimbólum áll, és $\delta(q, a) = (p, b, d)$, akkor a gép átmegy a p állapotba, az a szimbólumot ebben a cellában átírja b -re (persze lehet $a = b$ is), és a d irányba lépteti a fejet (egyet balra, jobbra vagy helyben marad). Amennyiben

⁵Ez nem jelenti azt, hogy ami RAM gépen megoldható pl. lineáris időben, az Turing-gépen is az lenne; az viszont igaz, hogy ha valami megoldható RAM gépen **polinomidőben**, akkor Turing-gépen is – csak lehet, hogy nagyobb fokszámú lesz a polinom azért, mert a Turing-gép nem támogat konstans időben tömbcímkést.

⁶szimbólumok véges, nemüres halmaza

$\delta(q, a) = \text{ACCEPT, REJECT vagy HALT}$, úgy a számítás leáll. **ACCEPT** esetben a gép **elfogadja** az inputot, **REJECT** esetben **elutasítja** az inputot (ekkor az output ez az egy bit), **HALT** esetben pedig egy **függvényt számított ki**, az output pedig a szalag aktuális tartalma a kezdő \triangleright és a záró (végtelen sok) \sqcup nélkül.

Nézzünk pár példát: ha $\Gamma = \{\triangleright, \sqcup, 0, 1\}$, $\Sigma = \{0, 1\}$ és az átmenettáblázat

Q	Γ	Q	Γ	D
s	\triangleright	s	\triangleright	\rightarrow
s	0	s	0	\rightarrow
s	1	p	1	\rightarrow
p	\sqcup	ACCEPT		

és minden más, a táblázattal nem lefedett esetben REJECT (ez lesz a konvenciónk, a nem szereplő $Q \times \Gamma$ sorokba REJECT-et feltételezünk).

Szimuláljuk le a számítást, ha ez a gép megkapja az 001 szót inputként. A szimulálásban a szalagtartalmat leírjuk a végtelen sok záró \sqcup közül csak párat leírva, aláhúzzuk azt a cellát, ahol az olvasófej épp tartózkodik, és eléírjuk az állapotot, tehát a gép kezdő **konfigurációja** $s, \sqcup 001 \sqcup$. (Ahogy más számítási modelleknél – véges automata, veremautomata – megszokhatuk, egy konfiguráció a gép aktuális működéséről egy snapshot, amit ha elmentenénk, onnan tudnánk folytatni a számítást, ahol abbahagytuk. Turing-gépnél a konfigurációba tartozik az **aktuális állapot**, a **fej pozíciója** és a **szalag aktuális tartalma**.)

Annak jele, hogy a gép egy lépésben egy C konfigurációból átmegy egy C' -be, a $C \vdash C'$ lesz; ha nulla, egy vagy több lépésben, annak a szokott módon $C \vdash^* C'$.

Tehát a példa számítás:

$$s, \sqcup 001 \sqcup \vdash s, \triangleright \sqcup 001 \sqcup \vdash s, \triangleright \sqcup 01 \sqcup \vdash s, \triangleright 001 \sqcup \vdash p, \triangleright 001 \sqcup \vdash \text{ACCEPT},$$

minden egyes lépésben annyit teszünk, hogy megnézzük az állapotot és az aláhúzott betűt, kikeressük a táblázat megfelelő (ezzel a két jellel kezdődő) sorát, és aktualizáljuk az állapotot, a szalag aktuális jelét (ez a gép sose változtat a szalagtartalmon), és léptetjük a fejet (az aláhúzást – ez a gép mindig jobbra lép).

Mivel a számítás **ACCEPT**-ra végződik, ezért ez a gép **elfogadja** a 001 inputot. Könnyű látni, hogy a gép jobbra teker a háromszögön és a 0-kon, közben végig az s állapotban marad, majd amikor megérkezik az első 1-es (ha nincs 1-es és s -ben lát \sqcup -t, akkor elutasít), akkor átmegy a p állapotba és tovább lép jobbra; ha ekkor \sqcup -ot lát (azaz ekkor fogy el az input), elfogadja az inputot, ha nem (mert van még az inputból hátra az első 1-es után), akkor elutasítja. Tehát a gép **a $0^n 1$ alakú szavakat fogadja el**, $n \geq 0$, a többi elutasítja.

Azt mondjuk, hogy az M (mint Machine) Turing-gép **eldönti** az $L \subseteq \Sigma^*$ nyelvet, ha az **L -beli** szavakon **ACCEPT**-ben áll meg, a **nem L -beli** szavakon pedig **REJECT**-ben áll meg. Tehát ez a gép a $0^* 1$ nyelvet dönti el. Ha az inputot mint kis endiában (kis helyiérték balra) ábrázolt bináris számot fogjuk fel, akkor a gép eldönti, hogy kettő-hatványt kapott-e vagy sem (feltételezve, hogy nincsenek leading 0-k).

Definíció

Egy nyelv **eldönthető** vagy **rekurzív**, ha van őt eldöntő Turing-gép.

Aki ismeri a formális nyelveket és a véges automatákat, annak könnyű végiggondolni, hogy minden véges automatával felismerhető nyelv (azaz minden **reguláris** nyelv) eldönthető: a

Turing-gép átmenettáblázata ugyanerre a mintára készülhet, vagyis mindig jobbra lép a szalagon, a szalagtartalmat sose írja felül, a kezdő \triangleright jelet olvasva a véges automata kezdőállapotába lép, onnan követi a véges automata átmenettáblázatát, majd a végén ha elérkezik a \sqcup jelhez, akkor ACCEPT, ha épp végállapotban van és REJECT, egyébként.

Tehát:

Állítás

Minden reguláris nyelv eldönthető.

Elgondolkodhatunk rajta, hogy mi történik akkor, ha az olvasófej „leesne” a szalagról; számos Turing-gép variáns van, mi azt követjük itt, amelyik szintaktikusan akadályozza ezt meg: \triangleright **olvasásakor balra lépni tilos, és csak \triangleright -ot lehet visszaírni ilyenkor a szalagra.** Ezzel megakadályozzuk, hogy a fej „leessen”.

A Turing-gép azért többet tud, mint egy véges automata (hiszen ahogy korábban mondtuk, minden algoritmikusan kiszámítható/eldönthető problémát meg lehet oldani Turing-géppel is, bármilyen hihetetlennek is látszik ez ezen a ponton), pl. a $\{0,1\}$ ábécé fölötti páros hosszú palindrom szavak nyelvét is el lehet dönteni a következő géppel:

Q	Γ	Q	Γ	D
s	\triangleright	s	\triangleright	\rightarrow
s	0	q_0	\triangleright	\rightarrow
s	1	q_1	\triangleright	\rightarrow
s	\sqcup	ACCEPT		
q_0	0	q_0	0	\rightarrow
q_0	1	q_0	1	\rightarrow
q_0	\sqcup	c_0	\sqcup	\leftarrow
q_1	0	q_1	0	\rightarrow
q_1	1	q_1	1	\rightarrow
q_1	\sqcup	c_1	\sqcup	\leftarrow
c_0	0	r	\sqcup	\leftarrow
c_1	1	r	\sqcup	\leftarrow
r	0	r	0	\leftarrow
r	1	r	1	\leftarrow
r	\triangleright	s	\triangleright	\rightarrow

A gép a következő algoritmust implementálja: az s állapotban a szalag elején van, egyet jobbra lép és megjegyzi az ott levő bitet: ha ez 0, akkor a q_0 , ha 1, akkor a q_1 állapotba kerül (ezt mondhatjuk úgy, hogy **megjegyzi a bitet** – így lehet egy gépnek az állapotába megjegyeztetni korlátos sok mennyiségű információt), és átírja a bitet \triangleright -re (technikailag „levágja” a szalag elejét). Majd a q_0 vagy q_1 állapotban a szalag **végére teker**, átlépve a 0-kat és 1-eket, amint elér a szó végére a \sqcup jelre, visszalép (így az utolsó karakteren áll) és a c_0 vagy c_1 „check” állapotba kerül. Ha ebben épp a megjegyzett bitet látja, akkor törli a cella tartalmát (mondhatni levágja az input végét), átmegy az r „rewind” állapotba, amiben a szalag **elejére teker vissza** és ott kezdi a ciklust előlről. Ha az s állapotban látja, hogy elfogyott az input, akkor fogadja el (mert ekkor sikerült összepárosítani a megfelelő betűket), egyébként elutasít.

Nézzünk egy **példa futást**: a 0110 inputon a számítás:

$$\begin{aligned}
 s, \triangleright 0110 \sqcup &\vdash s, \triangleright 0110 \sqcup \vdash q_0, \triangleright \triangleright 110 \sqcup \vdash q_0, \triangleright \triangleright 110 \sqcup \vdash q_0, \triangleright \triangleright 110 \sqcup \vdash q_0, \triangleright \triangleright 110 \sqcup \\
 &\vdash c_0, \triangleright \triangleright 110 \sqcup \vdash r, \triangleright \triangleright 11 \sqcup \sqcup \vdash r, \triangleright \triangleright 11 \sqcup \sqcup \vdash r, \triangleright \triangleright 11 \sqcup \sqcup \\
 &\vdash s, \triangleright \triangleright 11 \sqcup \sqcup \vdash q_1, \triangleright \triangleright \triangleright 1 \sqcup \sqcup \vdash q_1, \triangleright \triangleright \triangleright 1 \sqcup \sqcup \vdash c_1, \triangleright \triangleright \triangleright 1 \sqcup \sqcup \\
 &\vdash r, \triangleright \triangleright \triangleright 1 \sqcup \sqcup \vdash s, \triangleright \triangleright \triangleright \sqcup \sqcup \vdash \text{ACCEPT},
 \end{aligned}$$

ezt (elég sok szaladgálás árán) a gép elfogadja és valóban, mindig az első és az utolsó karaktert hasonlítja össze, törli azokat, ha nem egyformák, elutasít, egyébként ezt ismétli, amíg el nem fogy az input. Ez a gép valóban a páros hosszú palindromák nyelvét dönti el.

3.2. Eldöntés, felismerés, kiszámítás

Egy M Turing-gépet mint „függvényt” is írhatunk: ha $x \in \Sigma^*$ egy input szó, akkor $M(x)$ jelöli a gép outputját az x inputon. Láttuk, hogy ez lehet **ACCEPT** vagy **REJECT**, lehet akár egy

$y \in \Gamma^*$ output szó is (ha a gép **HALT**tal állt meg), de az is elképzelhető, hogy a gép **végtelen ciklusba esik** (pl. egy $\delta(q, a) = (q, a, -)$ átmenetet követve folyamatosan, de persze ennél sokkal komplikáltabb esetek is vannak), ennek jele $M(x) = \nearrow$.

Ha egy M Turing-gép minden szón megáll, mégpedig **ACCEPT**tel vagy **REJECT**tel, akkor **eldönt** egy nyelvet, az $L(M) = \{x \in \Sigma^* : M(x) = \text{ACCEPT}\}$ nyelvet. Amennyiben van olyan szó, amin végtelen ciklusba esik, akkor pedig **felismer** egy nyelvet, szintén az $L(M) = \{x \in \Sigma^* : M(x) = \text{ACCEPT}\}$ nyelvet. Tehát felismeréskor az L -beli szavak mindegyikét **ACCEPT**olni kell, a nem L -beli szavakon pedig vagy **REJECT**elni, vagy végtelen ciklusba esni.

Definíció

Egy nyelv **eldönthető** vagy **rekurzív**, ha van őt eldöntő és **felismerhető** vagy **rekurzívan felsorolható**, ha van őt felismerő Turing-gép.

Amennyiben a gép mindig **HALT**tal áll meg, akkor kiszámít egy függvényt, mégpedig nyilván az $x \mapsto M(x)$ függvényt (emlékezzünk, ilyenkor $M(x)$ az a szó, amit a szalagot záró végtelen sok \sqcup és a szalag elején levő \triangleright levágásával kapunk).

Definíció

Egy f függvény **kiszámítható** vagy **rekurzív**, ha van őt kiszámító Turing-gép.

Háromszalagos offline Turing-gép.

Később fogjuk algoritmusok idő- és tárigényét is elemezni.

Egy Turing-gép **időigénye egy adott inputon** egyszerűen a megállásig **meg tett átmenetek száma**. Ha az M **gép időigényéről** akarunk beszélni, akkor azt egy $f : \mathbb{N} \rightarrow \mathbb{N}$ függvénnyel írjuk le: az M gép időigénye akkor $f(n)$, ha **tetszőleges n hosszú input szón $O(f(n))$ lépésben megáll**. Így például a fenti példák közül a 0^*1 -et eldöntő gép időigénye lineáris, n , a palindromákat eldöntő gép időigénye pedig négyzetes (a sok oda-vissza tekerces miatt, számtani sor összegeként kijön), azaz n^2 .

Az teljesen validnak hangzik, hogy egy „értelmes” **Turing-gép időigénye legalább n** , hiszen különben végig se tudná olvasni az inputot, mielőtt döntést hozna.

Tárigény szempontjából viszont nem ennyire egyértelmű a helyzet. Ha a gép az inputot csak olvassa és sose írja át, és úgy dönt el egy nyelvet, akkor az a megközelítés is validnak hangzik (és ezt is fogjuk használni), hogy ebben az esetben **az input ne számítson bele a tárigénybe**. Ez megfelel a „hívó fél foglalja a memóriát” modellnek, szemben a „hívott fél foglalja a memóriát” modellel.

Tehát ha egyszalagos gépben gondolkodunk, akkor annak a tárigénye egy adott inputon **a számítás során látott összes cella száma** lesz (ez a fej mozgása miatt mindig a leghátsó, valaha meglátogatott cellának az indexe), ez a „hívott fél foglal” memóriamodell, és ez mindig legalább n lesz (a legrosszabb eset-korlát).

Tárigénynél viszont a „hívó fél foglal” memóriamodell támogatására bevezetjük a „**háromszalagos offline**” Turing-gépet. Ez az egyszalagoséhoz képest a következőképp fest:

- **Három darab**, jobbra végtelen szalaggal rendelkezik, ezek az **input**, a **munka** és az **output** szalagok.
- Mindhárom szalag ugyanúgy cellákra osztott, mint az előbb, minden cellába pontosan

egy szalagszimbólum kerül.

- Inicialisan az input szalag van úgy feltöltve, mint egyszalagos esetben (az input szalagon érkezik az input), a másik két szalag üres, azaz csak a kezdő \triangleright van az első cellájukban, a többi cellában \sqcup .
- **Mindhárom szalagon** áll egy-egy (egymástól függetlenül mozgatható) **olvasófej**.
- Az átmenetfüggvény $\delta : Q \times \Gamma^3 \rightarrow Q \times (\Gamma \times D)^3 \cup \{\text{ACCEPT}, \text{REJECT}, \text{HALT}\}$ alakú. Azaz: a gép tudja, milyen állapotban van és látja a három cellát, amik fölött a fejek vannak, ez alapján eldönti, hogy melyik állapotba menjen át és melyik szalagon mire írja át az épp olvasott szimbólumot, merre vigye az olvasófejet, vagy esetleg álljon meg.
- Továbbra is, \triangleright helyébe csak \triangleright kerülhet, és onnan balra lépni tilos.
- Az **input szalag „csak olvasható”**: ott a gép köteles mindig ugyanazt a szimbólumot visszaírni, mint amit olvasott. Továbbá, az input szalagon \sqcup -ról jobbra lépni tilos (ez azért kell, hogy az olvasófej ne bóklásszon el az input szalagon az inputtól messzire).
- Az **output szalagon balra lépni tilos**. (Ez azért kell, hogy a gép az output szalagot „stream módban” tudja írni.)
- Ha a gép a **HALT**-ban áll meg, akkor az **output szalagon található a kimenet** (levágva persze a háromszöget és a záró blankokat).

Ha egy Turing-gép ilyen típusú, akkor a tárigénybe nem kell beszámítani sem az input, sem az output szalagon „használt” cellákat (megfelelően annak a modellnek, mikor ezeket a hívó foglalja, ha lesz hívó, majd annak a tárigényébe beszámoljuk), csak **a munkaszalagon a számítás során látott összes cella száma lesz a tárigény**.

3.3. Egyéb Turing-gép variánsok

A Turing-gépekkel kapcsolatban nincs egységes konvenció, hogy milyen a „standard” Turing-gép. Vannak, akik nem különböztetik meg az input- és a szalagábécét, vannak, akik mindkét irányba végtelen szalaggal dolgoznak (és esetleg \triangleright marker nélkül), 3 szalag helyett akár mennyi k konstans sok szalagot megengednek, vagy egy mindkét irányba végtelen $2D$ „füzetet” és konstans sok író-olvasó fejet, amik eredetileg mind az origóból indulnak...

...de minden ekvivalens mindennel.

3.4. RAM-gép

Turing-gépet akkor fogunk használni (mégpedig egyszalagosat), amikor valamiről azt akarjuk megmutatni, hogy nem lehet megcsinálni. Amikor pedig valamire adott tár- vagy időigényű algoritmust adunk, akkor RAM-gép alapú pseudokódot fogunk adni, mert sokkal közelebb áll a strukturált imperatív programozási nyelvekhez a Turing-gépnél.

A RAM-gép konkrét utasításkészletéről vagy regiszter-szettjéről sincs egységesség az irodalomban, az itt ismertetettől eltérő is teljesen elfogadható.

A RAM-gép az előző modellek közül hasonlít a háromszalagos Turing-gépre abban az értelemben, hogy **háromféle regiszterrel** rendelkezik: **Input**, **Working** és **Output** regiszterekkel. A regiszterek **0-tól** vannak indexelve, **mindből végtelen sok** áll rendelkezésre, tehát input regisztereink $I[0], I[1], I[2], \dots$, munkaregisztereink $W[0], W[1], W[2], \dots$, és output regisztereink $O[0], O[1], \dots$.

Minden egyes regiszterbe egy-egy **tetszőlegesen nagy természetes számot** írhatunk bele. A RAM-gép inputja **pozitív egészeknek egy 0-terminated sorozata**, mely az input regiszterekben érkezik: tehát ha pl. az input a 17, 3, 31, 11 sorozat, akkor $I[0] = 17, I[1] = 3, I[2] = 31, I[3] = 11$ és $I[4] = 0$. Inicialisan az összes többi regiszter tartalma 0.

A RAM-gép szintaktikai előnye a Turing-géphez képest a **tömbcímzés**: pl. a 4 jelentése a 4 konstans, $W[4]$ értéke a 4. munkaregiszter tartalma (ez a **direkt címzés**), $W[I[4]]$ értéke pedig az $I[4]$ -edik munkaregiszter tartalma (ez az **indirekt címzés**). Persze tetszőlegesen kombinálható W, I és O direkt és indirekt címzésre, valid még pl. $W[W[0]], I[W[42]]$ és $O[W[1]]$ is. Azért, hogy a lehetséges műveletek számát egy limit alatt tartsuk, kettőnél mélyebb címzést nem engedünk meg szintaktikai szinten (persze több utasítást egymás után fűzve, a köztes értéket kirakva egy új munkaregiszterbe el lehet érni ilyen címzést is).

A RAM-gép **programja** sorszámozott utasítások tetszőleges sorozata. A sorszámokat nem kell nullától felfelé haladva egyesével kiosztani, de minden sorszám maximum egy utasításhoz tartozhat⁷. A RAM-gépnek egy **utasítása** a következők egyike lehet:

- **Kifejezés** lehet egy konstans, direkt vagy indirekt címzés, vagy két ilyen összege ill. különbsége, pl. $W[1] + I[W[0]]$ egy kifejezés;
- $L := R$, ahol L egy direkt vagy indirekt címzett érték, R egy tetszőleges kifejezés; ez az **értékadás** művelete, az R által mutatott érték belekerül az L -be, pl. $W[1] := I[2] + 1$ az első munkaregiszterbe másolja a második input regiszter tartalmát, plusz egyet.
A kivonás kiértékelésekor (mivel a regiszterekben természetes számokat szánunk) ha **az eredmény negatív lenne, akkor az eredmény 0**.
- **JZ(X, L)**, ahol X egy kifejezés, L egy (sor)szám: ez a **„jump-if-zero”** utasítás, ha X értéke 0, akkor a számítás az L . sorra ugrik majd, ellenkező esetben nincs ugrás és a következő sorszámú utasítást hajtjuk végre;
- **HALT**, **ACCEPT** vagy **REJECT** – funkciójukban ugyanaz, mint Turing-gép esetén, leállítja a számítást.

A gép az elvárt módon hajtja végre a programot: rendelkezik egy **programszámlálóval** (PC), ami mindig az aktuálisan végrehajtandó utasításra mutat, inicialisan az elsőre; ha megkapta az inputot az input regiszterekben a fent leírt módon, akkor végrehajtja a mutatott utasítást, és a feltételes ugrást kivéve minden esetben eggyel növeli a PC-t a következő utasításra (ha nincs ilyen, akkor REJECTel), jump-if-zeronál ha a kifejezés 0-ra értékelődik ki, akkor értelemszerűen az L . sorra ugrik.

Amennyiben az aktuális utasítás ACCEPT vagy REJECT, úgy a program elfogadja vagy elutasítja az input számsorozatot; ha HALT, akkor a program kiszámított egy függvényt, az **eredmény pedig az $O[0], O[1], O[2], \dots, O[n]$ számsorozat**, ahol $O[n + 1]$ az első olyan output regiszter, amiben a megálláskor 0 áll. (Tehát az output is egy zero-terminated sorozat). Ellenkező

⁷egyelőre

esetben folytatja a végrehajtást az aktuális utasítással.

Mint Turing-gépnél is, **az M program outputját az a_1, \dots, a_m inputon $M(a_1, \dots, a_m)$ jelöli**, ami \nearrow , ha a program ezen az inputon nem áll meg.

3.5. A RAM gép és a strukturált programozás

A RAM-gépünk elemi utasításaiból tudunk összetettebbeket is készíteni, ezeket fel is fogjuk használni később, mikor „RAM-programot” írunk valamire (az így kapott kód jobban fog emlekeztetni egy pszeudokódra).

Az **IF ($X == Y$) GOTO L** (ha az X és az Y kifejezések értéke megegyezik, ugorj az L számú sorra, egyébként a következőre) pl. előállítható így:

```
1:  $Z := X + 1$ 
2:  $Z := Z - Y$ 
3: JZ( $Z, 6$ )
4:  $Z := Z - 1$ 
5: JZ( $Z, L$ )
6: ...
```

ahol Z egy új (másra nem használt) munkaregiszter. Valóban: ha $X < Y$, akkor a harmadik sorban kiszámított $X + 1 - Y$ értéke 0 lesz (mert a természetes számok körében a kivonás ha negatívba menne, akkor 0 lesz), ha pedig az 5. sorban lesz a Z értéke 0, az pont azt jelenti, hogy $X + 1 - Y - 1 = X - Y \leq 0$, de $X \geq Y$ -nal együtt (különben elugrottunk volna a 3. sorról a 6-ra) épp azt jelenti, hogy $X == Y$.

Hasonló módon készíthetünk **feltétel nélküli ugrást** (arra jó pl. a $JZ(0, L)$ utasítás is), vagy egyenlőség helyett tesztelhetünk a $\leq, \geq, <, >$ relációkra is.

Ha már van feltételes GOTO, abból készíthetünk WHILE, DO vagy FOR ciklust is:

1: IF ($F == 0$) GOTO 4	1: R	1: $i := 1$
2: R	2: IF ($F == 0$) GOTO 4	2: WHILE($i \leq n$)DO {
3: GOTO 1	3: GOTO 1	3: R
4: ...	4: ...	4: $i := i + 1$ }
		5: ...
WHILE(F) DO R	DO R WHILE(F)	FOR $i := 1 \dots n$ DO R

Pszeudokódban megszokott konstrukció a **függvénydeklarálás**, és az egy függvényen belüli **lokális változók** deklarálása egy **stack** memóriaterületre. Ezt RAM géppel is megtehetjük pl. a következő módon:

1. Egy fix munkaregisztert, pl. $W[0]$ -t, kinevezünk SP-nek, Stack Pointernek.
2. Egy „megszokott” imperatív nyelven a függvényhíváskor először a verembe lenyomjuk a visszatérés címét, majd szépen sorban az argumentumokat, aztán ugrunk a függvény belépési pontjára.
3. Ezt RAM gépen is meg lehet tenni: először $W[SP] := k$, ahol k az a programsor, ahova a függvény végrehajtása után szeretnénk kerülni. Ezután SP egyesével növelése mellett $W[SP]$ -nek sorban értékül adjuk az argumentumokat. Végül GOTOzunk a függvény belépési pontjára.

4. Ha a függvény lefutott, ki kell vegyük a veremből az argumentumok számának megfelelő elemet: ha a függvénynek ℓ argumentuma volt, akkor ehhez csak SP-t kell csökkentjük ℓ -lel.
5. A visszaugrás egy GOTO a verem tetején lévő sorszámmra, azaz GOTO $W[SP]$. Végül csökkentjük SP-t, hogy a visszatérési cím is kikerüljön a veremből.
6. Ha a függvény használ lokális változókat x, y , stb. fantázianévvel, ezeknek is a veremben csinálunk helyet, és $W[SP + 1], W[SP + 2], \dots$ -ként érjük el őket. Kilépéskor persze ekkor az argumentumok száma plusz a lokális változók számával csökkentjük SP-t.

Amennyiben SP-t mindig (mondjuk) kettővel növeljük/csökkentjük, úgy ennek a veremnek a páros munkaregiszterek fognak megfelelni, a páratlan indexűeket pedig továbbra is használhatjuk bármire.

Hasonló módon **kezelhetünk heap memóriát**, dinamikus memóriafoglalással, ha arra van épp szükség.

A továbbiakban mikor valamilyen algoritmust írunk le, a RAM elemi utasítások helyett a „megszokott” pszeudokód konstrukciókat fogjuk használni és láthatjuk, hogy ezek az utasítások mind RAM gépen is implementálhatók.

Amiről eddig nem volt szó: a **RAM program futásának idő- és tárigényéről**. Az **időigény egy adott inputon** egyszerűen a futás során összesen végrehajtott **elemi** utasításoknak a száma. Ha pedig a **program időigényéről** beszélünk, az egy $f : \mathbb{N} \rightarrow \mathbb{N}$ függvény lesz: $f(n)$ akkor az M időigénye, ha az M program **tetszőleges n méretű** inputon $O(f(n))$ lépésben garantáltan megáll. (Ebben az is benne van, hogy M -nek minden inputon meg kell állnia.)

Itt most le kell tisztáznunk, mi is az **input mérete**. Turing-gépnél egyszerű volt a kérdés, ott az input szónak a hossza megfelelő volt méretnek. RAM gépnél azonban az input egy a_1, a_2, \dots, a_k számsorozat, pozitív egész számok egy sorozata. Mivel bármelyik szám tetszőleges nagy lehet, ezért nem definiáljuk az input méretét k -nak, hanem:

Definíció: Az input mérete

Az a_1, \dots, a_k input mérete a benne szereplő számok bináris reprezentációinak a hossza.

amit képlettel $n = \sum_{i=1}^k (1 + \log a_i)$ -ként írhatunk le⁸. Pl. ha az input $(1, 4, 2, 8, 5, 7)$, azaz binárisan $1, 001, 01, 0001, 101, 111$ (kis endián!), akkor ennek mérete $1 + 3 + 2 + 4 + 3 + 3 = 16$. Néha „16-bites”-ként is fogjuk emlegetni a 16 méretű inputokat.

Egy RAM gép **tárigényét** kissé komplikáltabban definiáljuk.

Először is, egy **regiszter tárigénye egy adott inputon**: a regiszterbe a futás során kerülő **legnagyobb szám** bináris reprezentációjának a hossza, plusz a regiszter **címének** (indexének) a bináris reprezentációjának a hossza. Tehát pl. ha egy adott futás során a $W[3]$ regiszterbe a legnagyobb valaha beírt érték a 20, akkor ennek a regiszternek a tárigénye: 20 bináris alakja 00101 , 3 bináris alakja 11 , összesen tehát a tárigény $5 + 2 = 7$.

A **program egy adott inputon** való futásakor össze kell adjuk az összes **használt** regiszter tárigényét, hogy megkapjuk a program tárigényét ezen az inputon. Kérdés, mi számít használatnak? Itt is, mint Turing-gépnél, **kétféle memóriamodell** van:

⁸ebből a tárgyból a log *mindig* kettes alapú.

Definíció

- Ha a RAM gép az input regisztereknek soha nem ad új értéket, az output regiszterekben pedig „stream módban” ír, tehát először $O[1]$ kap értéket, majd $O[2]$, stb., akkor **csak a munkaregiszterek** tárigényét kell összeadni: ami munkaregiszter valaha is címzésre kerül a program futása során (írásra vagy olvasásra), az használatnak számít.
- Ellenkező esetben az input és output regiszterek is „használatnak” számítanak, így pl. be kell számítsuk az input méretét is a tárigénybe (az összes valaha megcímezett input, output vagy munkaregiszter tárigényét össze kell adjuk).

Egy **program tárigénye** pedig megint egy függvény: az M tárigénye akkor $f(n)$, ha tetszőleges n méretű inputon legfeljebb $f(n)$ tárat használ.

(Caveat: tárigénynél nem baj, ha az algoritmus végtelen ciklusba esik, ha ugyanazt a memóriaterületet írja újra és újra, egy fix korlát alatt maradó számokkal, attól még a tárigény lehet véges is.)

3.6. Turing-gép szimulálása RAM géppel

A két géptípus nem ugyanolyan típusú inputot és outputot vár: Turing-gép egy Σ ábécé feletti szavakat, a RAM-gép pedig természetes számok egy sorozatát. Ez nem baj: a Σ (sőt a teljes Γ szalagábécé) minden betűjéből készíthetünk egy számot 0 és $|\Gamma| - 1$ közt, \sqcup kódja lesz a 0.

Hogy RAM géppel lehet (mondjuk egyszalagos) Turing-gépet szimulálni, az nem olyan meglepő: az input regiszterekben kerül a Turing-gép (számokkal elkódolt) inputja, majd a munkaregiszterekben írjuk át a Turing-gép konfigurációját: az állapotot, az olvasófej pozícióját és a szalagtartalmat. Az állapothalmazról is feltehetjük, hogy az $1, 2, \dots, |Q|$ számok az állapotok, ez kerül $W[0]$ -ba (mondjuk). Az olvasófej pozíciója is egy egész szám lesz, ez kerül $W[1]$ -be. $W[3]$ -ba a \triangleright számkódja kerül, $W[4]$ -be az input első jele, $W[5]$ -be a második, \dots , így elkészítettük a munkaregiszterekben a Turing-gép egy konfigurációját. $W[0]$ -t beállítjuk a kezdőállapotra, az olvasófej pozícióját pedig 2-re (mivel a második regiszterben kezdődik a tényleges szalagtartalom).

Ezek után nincs más dolgunk, mint az átmenettáblázatból egy nagy IF szerkezetet készíteni, egy $\delta(q, a) = (p, b, d)$ átmenetből pl. lesz egy $\text{IF}(W[0] == q \text{ AND } W[W[1]] == b)\{\dots\}$, a blokkon belül pedig aktualizáljuk $W[W[1]]$ -et b -re, $W[0]$ -t p -re és $W[1]$ -et növeljük vagy csökkentjük eggyel, a mozgásiránytól függően. Minden átmenetből létrehozunk egy ilyen IF struktúrát, az egészet pedig betesszük egy $\text{WHILE}(\text{TRUE})$ végtelen ciklusba. Amennyiben pedig termináló átmenetet hajtunk végre ACCEPT vagy REJECT válasszal, a RAM gép is ezt adja vissza; ha HALT válasszal, akkor pedig először a szalag tartalmát másoljuk ki az output regiszterekbe és ezután a RAM gép is leállhat HALT tal.

Háromszalagos Turing-gépet is tudunk teljesen hasonló módon szimulálni annyi különbséggel, hogy a három szalagot (mondjuk) úgy osztjuk szét a munkaregiszterek közt, hogy $W[0]$ lesz az állapot, $W[1]$ a pozíció az első szalagon, $W[2]$ a másodikon, $W[3]$ a harmadikon, $W[4]$, $W[7]$, $W[10]$, \dots az első szalag cellái, $W[5]$, $W[8]$, $W[11]$, \dots a második és $W[6]$, $W[9]$, $W[12]$, \dots a harmadiké; ily módon az egész konfigurációt el tudjuk tárolni a RAM gép munkaregisztereiben. Az átmenetfüggvény szimulálása pedig összetettebb IF-ekkel zajlik, hiszen tesztelnünk kell $W[0]$ -t, $W[W[1]]$ -et, $W[W[2]]$ -t és $W[W[3]]$ -at, és ezek alapján aktualizálni őket (léptetéskor hárommal növelve-csökkentve a pozíciók regisztereit).

Érdekes megfigyelni, hogy a szimuláció során sem az idő-, sem a tárigény nem romlik számottevően: ha az eredeti Turing-gép időigénye $f(n)$ volt, akkor a szimuláló RAM-gépé $O(f(n))$

(konstans sok IFet kell végigpörgetnie minden egyes átmenetnél, az elején egy input, a végén egy output másolással, de ez még mindig csak $O(f(n))$ lépés lesz), és az $f(n)$ tárigényből szintén $O(f(n))$ tárigény lesz: a RAM-gép ebben az esetben offline, így csak a munkaregiszterek tartalmát kell számolnunk, egy cellába a valaha írt legnagyobb érték $|\Gamma|$ lehet, és ha összeadjuk a használt regiszterek összes tárigényét, szintén egy $O(f(n))$ értéket kapunk⁹.

3.7. RAM gép szimulálása Turing-géppel

Az elsőre talán kicsit nehezebben hihető állítás, hogy RAM gépet is lehet szimulálni Turing-géppel, ráadásul az idő- és a tárigény *számottevő* romlása nélkül.

A módszer lényege az, hogy egy háromszalagos offline Turing-gép munkaszalagját tároljuk le a szimulált RAM gép aktuális konfigurációját *számpárok* formájában: az (i, j) – binárisan leírt – számpár jelentése az, hogy $W[i] = j$. Ilyen párokat írunk le, egymástól mondjuk vesszővel elválasztva a munkaszalagra¹⁰. Leírjuk továbbá a RAM gép programszámlálóját is (hogy hanyadik sorban járunk), mondjuk a munkaszalag legelejére. Mivel a RAM programnak tudjuk, hogy hány sora van, ennek a programszámlálónak csak véges sok értéke lehet.

A szimuláció egy lépésében pedig először is végigolvassuk és letesszük állapotba (a palindromás példához hasonlóan, csak nem egyetlen bitet, hanem egy egész szót) az aktuális sorszámot. A sorszámából tudjuk, hogy most milyen utasítást kell végrehajtsunk; a köztes műveletek eredményeit a szalag végére írjuk. Ha kifejezést akarunk kiértékelni, azt a következőképp tesszük: ha a kifejezés egy k konstans, akkor a szalag végére írjuk k bináris alakját. Ha a kifejezés egy $W[k]$ direkt címzés, akkor először a szalag végére írjuk k bináris alakját, majd megkeresünk egy (k, x) alakú párt a listán (ezt megint oda-vissza tekerve a szalagot, és mondjuk aláhúzással jelölve az éppen egyeztetett biteket meg lehet tenni, sok oda-vissza tekercselés árán), és ha találunk ilyet, akkor a szalag végétől töröljük k -t és másoljuk oda x -et. Ha nem találunk ilyet, akkor felvesszük a $(k, 0)$ párt a többi mögé és a 0-t másoljuk a szalag végére. Indirekt címzés esetén ezt ismétljük kétszer.

Ha összeadást kell végrehajtanunk, azt is meg lehet tenni Turing-géppel, ahogy a kivonást is (a szalag végére másolt két bináris szám összegét vagy különbségét szintén oda-vissza tekercselésekkel és aláhúzások mozgatásával elő lehet állítani), így tehát értékadás jobb oldalán szereplő értéket elő tudunk állítani. Ha tényleges értékadás a feladatunk, akkor ezek után a keresett célregiszternek megfelelő (k, x) párt is meg tudjuk keresni a szalagon (ha ilyen van, ha nincs, létrehozuk $(k, 0)$ -t a szalag végére), és az x eredeti értéke helyére az új értéket írjuk. Ekkor még arra kell figyelni, hogy ha az új érték hosszabb, mint x volt, akkor kellő sokszor jobbra kell shiftelnünk a szalag maradék részét, de (szintén a megjegyzem-jobbra lépek-megjegyzem-felülírom módszer ismétlésével) ez is megoldható. Feltételes ugrásnál a 0-ra tesztelés könnyű, és ha ugrani kell, akkor csak a programszámlálót kell átírjuk a megfelelő értékre. Ellenkező esetben növelni eggyel, de Turing-géppel azt is lehet.

Összességében ez egy nagyon lassú szimulációnak tűnik, de (aki nem hiszi, dolgozza ki a részleteket) tárigénye nagyságrendben pontosan megegyezik a szimulált RAM-gép tárigényével, időigénye pedig minden intuíció ellenére nem *sokkal* nagyobb: egy $f(n)$ időigényű RAM programot ilyen módon lehet szimulálni kb. $O((f(n))^6)$ időben háromszalagos Turing-gépen. Ez

⁹Valójában ha a szimulált Turing-gép offline, akkor az input szalagként ténylegesen az input regisztereket, output szalagként ténylegesen az output regisztereket kell használnunk, hogy kijöjjön ugyanaz a térigény, de ez is megoldható.

¹⁰Ez egyből érthetővé teszi a RAM gép komplikáltnak tűnő tárigényét is: így lesz a tárigénye kb. ugyanannyi, mint egy Turing-gépé ugyanarra a feladatra, hiszen a Turing-gép szalagjára pont leírjuk binárisan a cella címét és tartalmát

ugyan nagyságrendi eltérés, de ne felejtjük el, ebből a tárgyból akkor nevezzük *hatékonnak* egy algoritmust, ha polinom időigényű, a hatodik hatványra emelés pedig polinomból még mindig polinomot képez. Tehát

pontosan ugyanazokat a problémákat lehet polinom időben megoldani Turing-gépen,
mint amiket RAM gépen

sőt, ezek megint ugyanazok, mint amiket JAVA, C vagy bármelyik másik létező (nem feltétlenül imperatív) programozási nyelven meg lehet oldani polinomidőben.

4. Első bonyolultsági osztályaink

Láttuk, hogy bár a Turing-gép egy Σ ábécé feletti stringeket, a RAM gép pedig pozitív egész számok egy sorozatát várja inputként, ezeket oda-vissza tudjuk konvertálni és maga a konkrét reprezentáció nem lényeges. Hasonló módon a kurzuson vizsgált problémáink **input**jának meg fogunk engedni **egyéb struktúrákat** is (gráfokat, formulákat, stb.), anélkül, hogy konkrétan megegyeznének egy kódolásban (pl. gráfokat a szomszédsági mátrixukkal vagy az éllistájukkal reprezentálunk); csak annyit teszünk fel, hogy a kódolás **„elfogadhatóan kompakt”** (nem ábrázolunk pl. számokat egyes számrendszerben¹¹).

Definíció

Egy problémát **eldöntési problémának** nevezzük, ha várt outputja egy bites (igen vagy nem, accept vagy reject, ...).

Az összes eldöntési probléma közül

Definíció

- **R** jelöli az eldönthető problémák osztályát (hogyan RAM gépen, vagy Turing-gépen, az láttuk, hogy mindegy) és
- **P** jelöli a polinomidőben eldönthető problémák osztályát (ismét csak mindegy, hogy Turing-gépen vagy RAM gépen, ugyanazt kapjuk).

Azaz **P**-ben az összes olyan eldöntési **probléma** van, amire **létezik** $O(n^k)$ időigényű eldöntő algoritmus, valamilyen konstans k -ra. A Cobham-Edmonds tézis szerint ezeket tartjuk hatékonyan megoldható problémáknak.

Néhány konkrét probléma

Az első probléma, ami még számos alkalommal visszatér a kurzuson:

Definíció: ELÉRHETŐSÉG

Input: egy $G = (V, E)$ irányított gráf. Feltehető, hogy $V = \{1, \dots, N\}$.

Output: vezet-e G -ben (irányított) út 1-ből N -be?

¹¹egészen a pseudopolinomiális algoritmusokig és az erősen **NP**-teljes problémákig, stay tuned.

Erre a problémára létezik algoritmus: vezetünk egy X és egy Y halmazt, inicializáljuk $X = \{1\}$, $Y = \{1\}$ -gyel őket, az elgondolás az, hogy $X \cup Y$ -ban lesznek azok a csúcsok, akik elérhetők 1-ből (1 biztosan az), és a „ha valaki elérhető, akkor szomszédjai is azok” fixpontiterációt hajtjuk végre, vagyis minden lépésben kivesszünk X -ből (a még nem „kifejtett” csúcsok halmazából) egy csúcsot, és ennek az összes szomszédjai közül aki nincs még Y -ben, az belekerül X -be és Y -ba is. Ha N belekerül X -be, akkor elérhető; ha nem, és X kiürül, akkor nem az.

Keresési algoritmus az ELÉRHETŐSÉG problémára

```

1: IF ( $N == 1$ ) RETURN TRUE
2:  $X := \{1\}$ ,  $Y := \{1\}$ 
3: WHILE ( $X \neq \emptyset$ )
4:   választunk egy  $i \in X$  csúcsot
5:    $X := X - \{i\}$ 
6:   FOR  $j = 1 \dots N$ 
7:     IF  $(i, j) \in E$  AND  $j \notin Y$ 
8:       IF  $j == N$  RETURN TRUE
9:        $X := X \cup \{j\}$ ,  $Y := Y \cup \{j\}$ 
10: RETURN FALSE

```

A fenti algoritmus specifikációjából pár részlet kimaradt. Például, a gráfot lehet a szomszédsági mátrixával is reprezentálni (ekkor az input mérete $n = N^2$), vagy éllistával (ekkor pedig mondjuk $n = |E| \cdot \log N$, élenként kell két csúcsot letárolni, egy csúcs egy 1 és N közti szám, amit kb. $\log N$ biten tudunk ábrázolni). Egy további kérdés lehet, hogy az $(i, j) \in E$ lekérdezésnek mennyi az időigénye? Szomszédsági mátrixos reprezentációban ez egy konstans lenne (mert a tömbcímzés RAM gépen konstans idejű), rendezetlen éllistas reprezentációban mondjuk $O(|E|)$ (végigmenve az éllista összes elemén), a „szokásos” éllistas reprezentációban pedig a csúcsok foksámával lenne talán arányos, az $O(N)$, ha pedig kezdőcsúcsenként rendezett tömbben tartjuk a szomszédokat, akkor $O(\log N)$.

Az is kérdés lehet, hogy az X és Y halmazokat hogyan reprezentáljuk; ha pl. bitvektorral (azaz RAM gépen N regisztert használunk X -hez is és Y -hoz is, mindegyik regiszterbe egy bit kerül, ami TRUE akkor, ha az elem benne van a halmazban és FALSE, ha nincs¹²), akkor az inicializálás $O(N)$ lépés, az üresség-tesztelés szintén $O(N)$, egy csúcs kiválasztása szintén $O(N)$, egy elem hozzáadása és elvétele pedig $O(1)$, konstans idejű. Ha láncolt listával tesszük, akkor az inicializálás, az üresség-tesztelés és egy elem kiválasztása-kivétele lesz $O(1)$, a bővítés pedig $O(N)$. Kiegyensúlyozott bináris keresőfával pedig az inicializálás és az üresség-tesztelés $O(1)$, a hozzáadás-kivétel pedig $O(\log N)$ időigényű műveletek.

A fenti kérdések algoritmikus szempontból lényegesek egy **tényleges** implementációban, és mindig a specifikus domain dönti el, hogy éppen melyik konkrét adatszerkezet megvalósítása ad „jobb” megoldást. **Bonyolultságelméleti** szempontból ezeket a kérdéseket egy kicsit messzebből nézzük, nekünk **csak** az lesz fontos, hogy van-e **polinomidejű** algoritmus; és azt láthatjuk, hogy akárhogyan is variáljuk a gráf fenti reprezentációit, az n -hez (az input mérete, hány biten ábrázoljuk a gráfot) képest ez a fenti algoritmus mindig polinomidejű lesz: bármelyik adatszerkezetet is választjuk, minden egyes sor $O(N)$ idő alatt fut le, és mivel $N \leq n$, ez $O(n)$ idő; azt is látjuk, hogy a WHILE ciklus minden egyes i -re csak egyszer futhat le, így összesen legfeljebb N -szer választunk egy csúcsot, N -szer vesszük ki a halmazból, a 7–9. sorok ciklusmagja pedig így legfeljebb N^2 -szer fog lefutni, de még ha a legrosszabb implementációt is választjuk, az egész időigény akkor is $O(N^3)$ marad, ami pedig $O(n^3)$, tehát polinom és ez nekünk elég.

¹²emlékszünk: ezt hívják a halmaz **karakterisztikus függvényének**

Az architektúra pontos ismerete nélkül egyébként sem lehet annál pontosabban mondani, mint hogy „polinom”. Ha ezt bármelyik mainstream imperatív programozási nyelven (mint a C, a C++, a JAVA, a PYTHON, a RUBY vagy a PHP), rendes éllistával implementáljuk, mondjuk az X halmazt egy veremmel, az Y halmazt egy bitvektorral ábrázolva, akkor egy $O(|V| + |E|)$, azaz lineáris időigényű algoritmust kapunk, mégpedig a **mélységi keresést**.

Ennyire általában nem fogunk lemenni a technikai szintre, amit ebből az algoritmusból látunk, az nekünk annyit mond, hogy

Állítás

ELÉRHETŐSÉG $\in \mathbf{P}$.

Később, mikor már több bonyolultsági osztályt ismerünk, pontosabban besoroljuk az ELÉRHETŐSÉG problémát.

Egy másik probléma az UTAZÓÜGYNÖK PROBLÉMA, avagy TSP:

Definíció: TSP.

Input: az $1, \dots, N$ városok és bármely két $i \neq j$ város **távolsága**: $d_{i,j}$.

Output: találjunk egy, az összes várost **pontosan egyszer** érintő legrövidebb körutat.

A TSP probléma nem eldöntési, hanem **függvényprobléma**, mivel nem egy bites a kimenete. A reprezentációval most sem foglalkozunk behatóan, az input méretét, n -t, N egy polinomjának vesszük¹³.

A problémára a **triviális** módszer az összes körút felsorolása. Mivel körbe megyünk, tekinthetjük úgy, hogy az 1-es csúcsból indulunk és a maradék $N - 1$ csúcsot kell sorbarakjuk, ezt $(N - 1)!$ -féleképp tehetjük meg (ha a távolságmátrix **szimmetrikus**, akkor elég „csak” a felét megnézni, mert akkor mindegy, hogy egy-egy körön melyik irányba megyünk). A faktoriális elég gyorsan növekvő függvény, $2^{O(n \log n)}$ körüli a növekedési rendje, ez tehát **nem polinom időigényű**.

Dinamikus programozással kicsit javíthatunk az időn: ha minden $X \subseteq \{1, \dots, N\}$ halmazhoz és $i \neq j \in X$ elemeihez letárolunk egy, az i -ből a j -be X minden csúcsán pontosan egyszer áthaladó utat, akkor a kételemű $X = \{i, j\}$ halmazokhoz ezt konstans időben elő tudjuk állítani (egy lépés, $d_{i,j}$ költséggel); általában pedig az (X, i, j) hármas megoldása a $d_{i,k} + (X - \{i\}, k, j)$, $k \in X - \{j, i\}$ értékek minimuma lesz (ahányféleképp lehet, kiválasztjuk az i utáni első csúcsot). Ez részhalmazonként $O(N)$ művelet, részhalmazból és i, j -ből van $N^2 \cdot 2^N$, összesen $N^3 \cdot 2^N$ időigény, ami jobb, mint az $(N - 1)!$. Ha még azt is észrevevessük, hogy a j -t mindig elég csak $j = N$ -re kiszámoljuk, kapunk egy $N^2 \cdot 2^N$ időigényű algoritmust. Ez **szintén nem polinom**. (Hiszen ahogy láttuk, egy exponenciális függvény, mint a 2^N , tetszőleges polinomnál előbb-utóbb nagyobbá válik.)

A mai napig **nem ismert**, hogy **létezik-e** polinomidejű algoritmus a TSP-re. Rengeteg ilyen problémával fogunk találkozni a kurzus során. Ráadásul ezek a problémák mind „egyforma nehezek”¹⁴ lesznek, ami annyit jelent, hogy ha bármelyiket közülük valaki meg tudja oldani polinomidőben, akkor mindegyik megoldható polinomidőben. Mivel azonban ezeknek a (több

¹³bár ha a távolságok óriásiak a városok számához képest, akkor még az $n = N^2 \log D$ is indokolt lehet, ahol D a maximális távolság két város közt. Később látni fogjuk, hogy a problémára akkor sem ismert polinomidejű algoritmus, ha minden távolság 1 vagy 2.

¹⁴NP-teljesek

ezer) problémának egyikére se adott még senki polinomidejű megoldást, elég széles körben elfogadott az a sejtés, hogy ezek a problémák (mint a TSP) nem oldhatók meg polinomidőben, nincsenek \mathbf{P} -ben. De ezt bebizonyítani se sikerült még senkinek; ahogy korábban mondtuk, „alsó korlátokat” bizonyítani nagyon nehéz, még olyan típusúakat is, hogy „erre a problémára szuperpolinomiális¹⁵ algoritmus létezik csak”.

Függvényproblémákkal a kurzus során viszonylag keveset fogunk foglalkozni¹⁶, optimalizálási problémák **eldöntési változatát** viszont gyakran vizsgáljuk majd: a probléma inputján kívül még adunk egy C célértéket is, és azt kérdezzük, hogy az optimum kisebb/nagyobb-e (attól függően, hogy minimalizálási vagy maximalizálási problémáról van szó) C -nél. Az eldöntési változatot általában úgy jelöljük, hogy a probléma neve után írunk egy (E) suffixet. Így például:

Definíció: TSP (E).

Input: az $1, \dots, N$ városok, bármelyik két $i \neq j$ város $d_{i,j}$ távolsága és egy $C \geq 0$ **célérték**.

Output: van-e olyan, minden városon pontosan egyszer áthaladó körút, **melynek össze-költsége legfeljebb C ?**

Ha az eldöntési változat „nehéz”, akkor persze az optimalizálási is nehéz kell legyen: ha az optimalizálási könnyen megoldható lenne, akkor azt oldanánk meg és ezután csak az optimumot kellene összehasonlítani C -vel.

Ha pedig az optimalizálási változat nehéz, akkor **általában** az eldöntési változat is az, hiszen ha az eldöntési változat könnyű lenne, akkor **általában** felelve kereséssel meg tudnánk találni az optimumot: először $C = 1, 2, 4, 8, \dots$ célértékkel oldanánk meg az eldöntést, majd mikor az első IGEN választ kapjuk, már van egy $(2^k, 2^{k+1}]$ alakú intervallumunk, benne a megoldással (hiszen az optimum az a legkisebb C , amire a válasz IGEN), ezután mindig az aktuális intervallumunk felezőpontjára rákérdezve és a válasznak megfelelő fél-intervallummal haladva tovább újabb k lépés után megkapjuk az optimum pontos értékét. Ez a módszer akkor működik, ha az optimalizálandó függvényünk értékkészlete pozitív egész; és ha az optimum értéke legfeljebb n -nek **exponenciális** függvénye¹⁷, akkor az eldöntési változatot **polinom** sokszor hívjuk csak, így ha az eldöntési változatra van hatékony algoritmus, ekkor az optimalizálásra is hatékonyat kapunk.

Ezért általában az eldöntési és az optimalizálási problémák „egyforma nehéznek” számítanak, így mi rendszerint az eldöntési változatokat fogjuk vizsgálni.

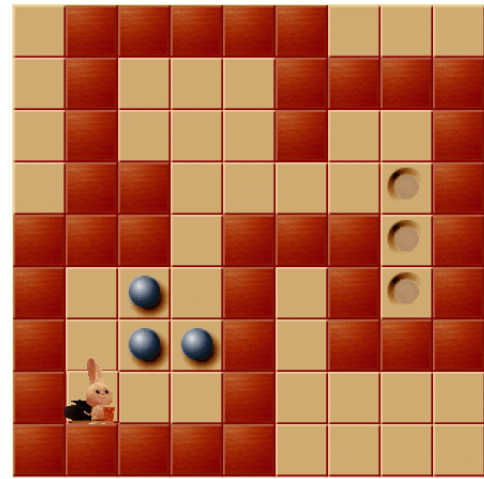
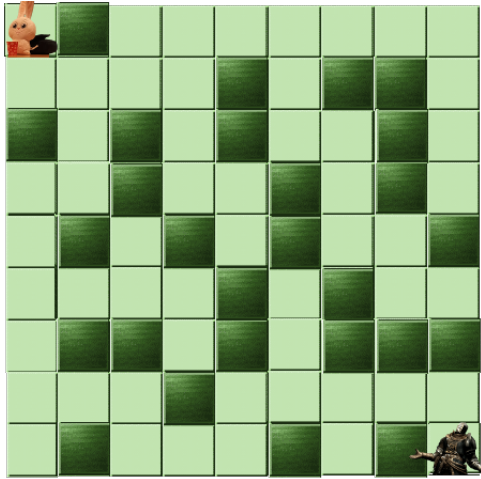
5. Problémák összehasonlítása: visszavezetések

Mielőtt formálisan definiálnánk, mit is értünk azon, hogy egy probléma „nehezebb”, vagy jobban mondva „legalább olyan nehéz”, mint egy másik, nézzünk egy példát.

¹⁵szuperpolinomiális: $\omega(p)$ minden p polinomra, minden polinomnál gyorsabban növekvő. Ilyen a 2^n , a $2^{\sqrt{n}}$ stb.

¹⁶majd mikor approximálunk

¹⁷Ezen a kurzuson „exponenciális” minden $c^{O(n^k)}$, tehát „konstans-a-polinomadikon” alakú függvény. Ezek nagyon nagyok is lehetnek, pl. a faktoriális is ilyen.



MAZE és SOKOBAN

A MAZE problémában a következő a feladat: input egy labirintus (példaként lásd a fenti ábrát), melynek a bal felső sarkából szeretne a nyúl eljutni a jobb alsó sarkában található lovagig (úgy, hogy csak a világos mezőkön tud lépkedni közben). Kérdés, hogy el tud-e jutni a céljához (tehát ez ebben a formában egy **eldöntési probléma**).

A SOKOBAN problémában (ehhez is lásd a fenti ábrát) a nyúl szintén csak a világos mezőkön tud lépkedni, és a golyókat¹⁸ tudja tologatni: ha egy golyó mellé áll vízszintesen vagy függőlegesen, úgy, hogy a golyó másik oldalán világos mező áll, akkor tolhat egyet rajta, ő kerül a golyó helyére, a golyó pedig rágurul a világos mezőre. Kérdés, hogy a nyúl tudja-e tologatni a golyókat úgy, hogy végül minden golyó egy lyukba kerüljön (így ez is egy **eldöntési probléma**).

Az ábrán látható mindkét input a vonatkozó problémának egy **igen** példánya.

A kérdés, hogy melyiket „érezzük” nehezebbnek? Erre egy **szokványos (és rossz) válasz** a következő: a MAZE problémát meg lehet oldani mondjuk (szélességi vagy mélységi) kereséssel: a gráf csúcsai a világos mezők, él köti össze a szomszédosakat, a bal felsőből akarjuk elérni a jobb alsót. Ez tehát így egy ELÉRHETŐSÉG probléma, amire láttunk már polinomidéjű, azaz hatékony algoritmust. A SOKOBAN problémánál viszont a játék egy-egy állásához a golyók és a játékos pozícióját **kell** letárolnunk és be **kell** járnunk egy nagy keresési teret, aminek a mérete akár exponenciális nagy is lehet, ezért azt **nem tudjuk** hatékonyan megoldani, tehát a SOKOBAN nehezebb.

A válasz (pontosabban az indoklás) azért nem jó, mert **nem tudjuk, hogy tényleg be kell-e járjuk azt a keresési teret**. Ahogy a TSP problémára is láttunk egy dinamikus programozási algoritmust, ami az $n!$ méretű keresési teret egészen $n^2 \cdot 2^n$ -re csökkentette, itt se tudjuk kizárni hatékonyabb algoritmus létezését, nincs alapunk azt mondani, hogy „ezt csak így lehet megoldani”. A fenti okoskodással két **algoritmust** hasonlítunk össze (a MAZE-en végrehajtott mélységi keresést és a SOKOBAN keresési terének bejárását), és ebben tényleg a MAZE-re adott **algoritmus** a gyorsabb. De ettől még elképzelhető, hogy valaki egyszer csak talál a SOKOBAN-ra is egy lineáris idejű algoritmust...ez a módszer **nem jó** arra, hogy **problémák** nehézségét hasonlítsuk össze.

Ami viszont jó arra, hogy problémákat hasonlítsunk össze, az a következő intuitív megfogalmazás:

¹⁸A golyók száma és a táblaméret nem rögzített.

Az A probléma legfeljebb olyan nehéz, mint a B , ha a B -re írt algoritmussal meg tudjuk oldani az A -t is egy egyszerű inputkonverzió után.

Ez „jól hangzik”: ha a B problémát megoldó algoritmussal meg tudjuk oldani az A -t, akkor tényleg úgy is érezzük, hogy a B probléma „általánosabb”, „nehezebb”, mint az A .

Bonyolultságelméletben a fenti „inputkonverziót” **visszavezetésnek** hívjuk, és többfélet is definiálunk annak függvényében, hogy mikor számít „egyszerűnek” egy inputkonverzió: a **rekurzív** és a **hatékony** visszavezetést¹⁹.

Definíció: Rekurzív visszavezetés

Az A eldöntési probléma **rekurzívan visszavezethető** (vagy „Turing-visszavezethető”) a B eldöntési problémára, jelben $A \leq_R B$, ha van olyan f **rekurzív** függvény, mely A inputjaiból B inputjait készíti **választartó** módon, azaz minden x inputra $A(x) = B(f(x))$.

A fenti definícióban ha A egy eldöntési probléma és x az A -nak egy inputja, akkor $A(x)$ akkor 1, ha az x egy „igen” példány, és 0, ha x egy „nem” példány. Magyarán, a rekurzív visszavezetésnél nem követelünk meg mást, csak annyit, hogy az inputkonverzió kiszámítható legyen valamilyen algoritmussal, bármekkora is legyen a tár- és időigénye.

Ez a visszavezetés-fogalom **kiszámíthatóság-elméletben** hasznos:

Állítás

Ha $A \leq_R B$ és B eldönthető, akkor A is eldönthető.

Ha pedig $A \leq_R B$ és A eldönthetetlen, akkor B is eldönthetetlen.

Bizonyítás

Az első állítás világos, hiszen ha B -re van algoritmus, mondjuk \mathcal{B} , és f egy rekurzív visszavezetés A -ról B -re^a, akkor az A problémát megoldja az $\mathcal{A}(x) := \mathcal{B}(f(x))$ algoritmus.

A második állítás pedig az első átfogalmazása^b.

^aaz irány fontos! ne keverjük össze, a két probléma közül melyiket vezetjük vissza melyikre.

^blogikában ez a **kontrapozíció**: ha $F \rightarrow G$ igaz, akkor $\neg G \rightarrow \neg F$ is igaz.

Ez a visszavezetés-fogalom viszont **bonyolultságelméletben** nem osztályozza nehézség szerint az eldönthető problémákat²⁰. A (kontinuum sok) eldöntési probléma közül van kettő, melyek intuitíve „a” legkönnyebbek kell legyenek minden értelmes metrika szerint, ezek a **triviális problémák**: az egyik az, melynek minden lehetséges input „igen” példánya, a másik pedig az, melynek minden lehetséges input „nem” példánya. (tehát az egyik az, melyet eldönt a `return true`; kód, a másik pedig, melyet eldönt a `return false`;). Az összes többi eldöntési problémát **nemtriviálisnak** nevezzük. A következő állítás azt mondja, hogy a rekurzív visszavezetéssel nem tudunk „nehézség szerint” különbséget tenni két eldönthető nemtriviális probléma közt:

¹⁹később még lesz logtáras is

²⁰egyébként az **eldönthetetlen** problémákat igen: vannak olyan problémák, amik „eldönthetlenebbek”, mint mások, sőt az eldönthetlenség szintjei egy **végtelen hierarchiát** alkotnak, ezt hívják aritmetikai hierarchiának

Állítás

Ha A eldönthető probléma, B pedig nemtriviális, akkor $A \leq_R B$.

Bizonyítás

Legyen A eldönthető probléma, oldja meg mondjuk az \mathcal{A} algoritmus. Legyen továbbá B egy nemtriviális probléma, mondjuk y_0 egy „nem” és y_1 egy „igen” példánya. Akkor a következő f függvény:

$$f(\mathbf{x}) := \text{if}(\mathcal{A}(x)) \text{ then } y_1 \text{ else } y_0$$

egy rekurzív visszavezetés A -ról B -re.

Valóban, hiszen ha x egy „igen” példány, akkor a fenti $f(\mathbf{x})$ az y_1 -et adja vissza, ami a B -nek „igen” példánya, ha pedig „nem” példány, akkor y_0 -t, ami a B -nek „nem” példánya, tehát f tartja a választ; mivel pedig \mathcal{A} is és a feltételes elágazás is kiszámítható, kompozíciójuk, azaz f is az.

Így tehát az összes eldönthető nemtriviális probléma „egyforma nehéz” eszerint a visszavezetés szerint, ezért tovább kell finomítanunk a definíción, hogy alkalmas legyen eldönthető problémák (mint a MAZE és a SOKOBAN) „összemérésére”. A rekurzív visszavezetés esetében a problémát az okozza, hogy túlságosan sok erőforrást engedünk meg az input konvertálására, annyit, amennyi már ahhoz is elég, hogy helyette megoldjuk az A problémát. A **hatékony visszavezetésben** korlátozzuk az inputkonvertálásra szánható időt:

Definíció: Hatékony visszavezetés

Az A eldöntési probléma **hatékonyan visszavezethető** (vagy „polinomidőben visszavezethető”) a B eldöntési problémára, jelben $A \leq_P B$, ha van olyan f **polinomidőben kiszámítható** függvény, mely A inputjaiból B inputjait készíti **választartó** módon.

A rekurzív visszavezetéshez hasonló összefüggések állnak fenn a hatékony visszavezetésre is:

Állítás

Ha $A \leq_P B$ és B **polinomidőben** eldönthető, akkor A is eldönthető polinomidőben.

Ha pedig $A \leq_P B$ és A -ra nincs polinomidejű algoritmus, akkor B -re sincs.

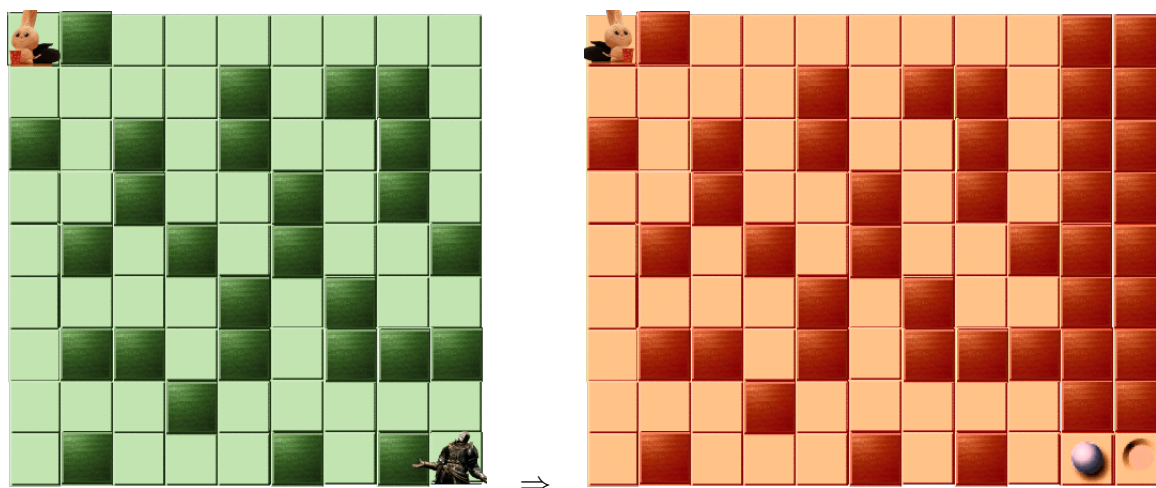
Bizonyítás

Mint az előbb: ha B -t polinomidőben megoldja a \mathcal{B} algoritmus, és f az A -ról B -re egy hatékony visszavezetés, akkor A -t az $\mathcal{A}(x) := \mathcal{B}(f(x))$ algoritmus polinomidőben (mert két polinomidejű algoritmus kompozíciója is polinomidejű) eldönti (a választartás miatt) A -t. A második állítás megint az első kontrapozíciója.

Mivel a bonyolultság mérésére döntően a polinomidejű visszavezetést fogjuk használni, \leq_P **helyett gyakran csak \leq -t fogunk írni** a továbbiakban.

Nézzünk példát:

$\text{MAZE} \leq \text{SOKOBAN}$: egy hatékony visszavezetés kibővíti a táblát, a jobb alsó sarok mellé jobbra helyez egy golyót, amellé jobbra az egyetlen lyukat és leveszi a lovagot a tábláról:



$\text{MAZE} \leq \text{SOKOBAN}$: példa a visszavezetésre.

Világos, hogy a fenti átalakítás általában is tartja a választ: ha a nyúl oda tud érní az eredeti MAZE problémában a lovaghoz, akkor a konvertált példányban is oda tud érní a lovag helyére, majd jobbra löki a golyót és megoldotta a feladványt.

Ha pedig a generált példány IGEN példány, akkor a golyó csak úgy kerülhet a lyukba, ha a nyúl odaér előbb az eredeti jobb alsó sarokba; ugyanezen az útvonalon haladva a MAZE probléma egy megoldását kapjuk.

Az is világos, hogy a konverzió hatékony^a: csak annyit kell tegyünk, hogy le vesszük a lovagot, és kibővítjük a tábla jobb alsó részét két új mezővel, ez (mondjuk) lineáris időben megvan.

^aazaz polinomidejű

Nézzünk még néhány példát. A TSP eldöntési változatát már láttuk. Egy nagyon hasonló probléma a HAMILTON-ÚT:

Definíció: HAMILTON-ÚT.

Input: Egy G gráf^a.

Output: Van-e G -ben minden csúcsot (pontosan) egyszer érintő út?

^aHa nincs nyíl a gráf fölött, az azt jelzi, irányítatlan a gráf; ha \vec{G} -t írok, akkor irányított gráfról van szó.

A következő visszavezetés pedig azt mutatja, hogy a $\text{TSP}(E)$ legalább olyan nehéz, mint a HAMILTON-ÚT:

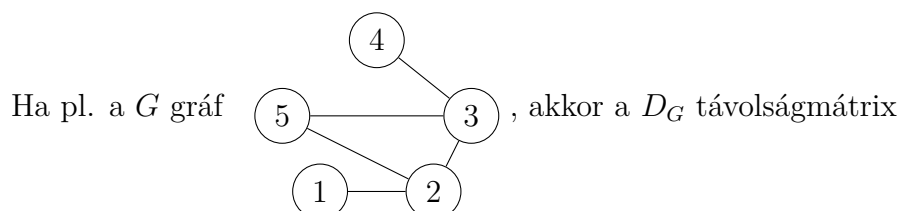
$\text{HAMILTON-ÚT} \leq \text{TSP}(E)$: adnunk kell egy olyan hatékony inputkonverziót, mely a HAMILTON-ÚT egy példányát transzformálja át a $\text{TSP}(E)$ egy példányába, méghozzá választartó módon.

Azaz: egy input G gráfból kell készíteni egy távolságmátrixot városok közt, és egy célszámot, aminél kisebb-egyenlő költségű körutat keresünk a készített térképen.

Ilyenkor persze „jogunkban áll” akár az is, hogy a gráf **élei** legyenek a városok; de ebben a példában nincs ilyen komplikált dolgunk, a generált $TSP(E)$ inputban a városok az eredeti gráf **csúcsai** lesznek. Távolságmátrixot kell építsünk, itt tehát bármelyik két csúcs közé kell betegnünk egy-egy költséget.

Lássuk, mi történik, ha **két csúcs közt volt G -ben él, akkor legyen az ő távolságuk 1; ha nem volt, akkor legyen 2!** Önmagától persze mindenki legyen 0 távolságra.

Lássuk ezt egy példán, mire gondolok:



d	1	2	3	4	5
1	0	1	2	2	2
2	1	0	1	2	1
3	2	1	0	1	1
4	2	2	1	0	2
5	2	1	1	2	0

Mivel a generált $TSP(E)$ példányban minden távolság vagy 1, vagy 2, így az optimális körút költsége valahol N és $2N$ közt lesz, ahol N a csúcsok száma.

Ha van Hamilton-út a gráfban, akkor választva egy Hamilton-utat és azon végigmenve összesen $N-1$ élen haladtunk, eddig az út költsége $N-1$, aztán a Hamilton-út végpontjából visszaugorva a kezdőpontba vagy 1, vagy 2 lesz a távolság értéke (attól függően, hogy volt-e ott él vagy sem), így ilyenkor **a $TSP(E)$ példányban az optimum értéke legfeljebb $N+1$.**

Ha pedig a generált $TSP(E)$ példányban van legfeljebb $N+1$ költségű körút, akkor abban van N él, mindnek a súlya vagy 1, vagy 2. Tehát legfeljebb egy olyan lépést tehet, ahol a távolság 2; ha csupa 1 súlyú lépést tesz, az az eredeti gráfban egy Hamilton-kör (ekkor van Hamilton-út is persze), ha pedig van egy 2 súlyú lépés, azt kihagyva a körútból az eredeti gráfban egy Hamilton-utat kapunk, tehát ilyenkor mindenképp *van Hamilton-út G -ben*.

Azt kaptuk tehát, hogy ha G -ből a fenti módszerrel előállítjuk a D_G távolságmátrixot (ez könnyű átalakítás, csak végig kell menni a G szomszédsági mátrixán és az 1-esekből 1-est, a főátlóbeli értékekből 0-t, a többi 0-ából pedig 2-est csinálni), és célszámnak beállítjuk a $C = N+1$ értéket, akkor

G -ben pontosan akkor van Hamilton-út, ha D_G -ben van legfeljebb C költségű körút

tehát az átalakítás hatékony és tartja a választ, azaz visszavezetés HAMILTON-ÚTról $TSP(E)$ -re.

Általában is ilyen módon fogunk visszavezetéseket készíteni:

1. megadunk egy f átalakítást, ami az A probléma inputjaiból a B probléma inputjait készíti;
2. megmutatjuk, hogy ha A -nak egy igen példányából indulunk, akkor B -nek egy igen példányába érkezünk;
3. megmutatjuk, hogy ha B -nek egy igen példányába érkezünk, akkor A -nak egy igen példányából kellett induljunk.

Sokszor a három közül a harmadik pont lesz a legnehezebb; a tendencia az, hogy az ember amikor visszavezetést próbál gyártani, akkor az első két pont teljesülni szokott, de a harmadik nem mindig. Amikor nem, azt úgy mondom, hogy „hamis találatok” is születnek (mikor nem példányból készül **igen** példány, ekkor persze a konverzió **nem** visszavezetés).

Nézzünk még egy példát. Ehhez előbb megadjuk a két problémát, amiről szó lesz:

Definíció: SAT.

Input: egy konjunktív normálformájú (CNF) formula.

Output: kielégíthető-e?

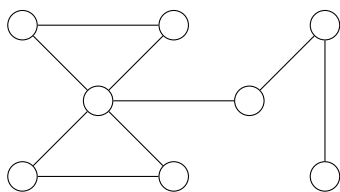
Például, $(p \vee \neg q \vee r) \wedge (p \vee q \vee \neg r) \wedge (\neg p \vee q)$ egy SAT input, ez épp egy **igen** példány, mert mondjuk a $p = q = r = 1$ értékadás kielégíti.

Definíció: PÁROSÍTÁS.

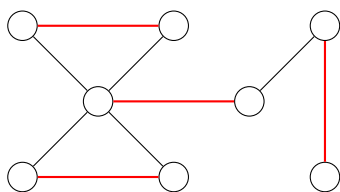
Input: egy G gráf.

Output: van-e G -ben teljes párosítás?

Például, ha az input a következő gráf:



akkor ez egy **igen** példány, egy teljes párosítás benne **pirossal**:



És a mondás az, hogy a SAT a kettő közül a nehezebb (legalábbis: „legalább olyan nehéz”, mint a PÁROSÍTÁS):

PÁROSÍTÁS \leq SAT.

Tehát kell adjunk egy **hatékony** transzformációt, ami a PÁROSÍTÁS inputjából (**egy G gráfból**) készíti a SAT egy inputját (**egy φ_G CNF-et**) úgy, hogy tartsa a választ, vagyis **G -ben pont akkor legyen teljes párosítás, ha φ_G kielégíthető.**

Ha gráfból kell formulát készítsünk, szóba jöhet, hogy a csúcsokból készítsünk változókat, vagy az is, hogy az élekből. Mivel egy teljes párosításba az éleket „válogatjuk be”, azokról döntünk, hogy benne legyenek-e a gráfban vagy sem, ezért itt az tűnik egy logikusabb lépésnek, hogy **minden élhez rendelünk egy logikai változót.**

Az intuíció lehet pl. az, hogy az u élhez rendelt változót akkor állítsuk igazra, ha kiválasztjuk, és akkor hamisra, ha nem választjuk ki.

Ekkor csak le kell írjuk, hogy minden csúcsra pontosan egy kiválasztott él illeszkedik. Ez azért is hangzik jól, mert **univerzálisan kvantálja az input gráfunk csúcsait**, ami

praktice azt jelenti, hogy minden csúcsra felírjuk, hogy erre is meg erre is meg erre is pontosan egy él illeszkedik, és az egészet összeeséljük; ha csúcsonként CNF-et tudunk írni, akkor az egész is egy CNF lesz, mert CNF-ek éselése CNF. Általában is jó ötlet valami **lokális** transzformációt keresni, akkor szinte biztos, hogy hatékony átalakítást fogunk kapni (az nem biztos, hogy rögtön elsőre a választ is tartani fogjuk, de legalább a sebességgel nem lesz gond).

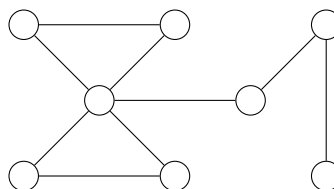
A „pontosan egy” az ugyanaz, mint „legalább egy” és „legfeljebb egy”, ez az „és” szó miatt megint jó hír, mert akkor ezekre is elég CNF-et írjunk.

- Azt, hogy az u csúcsra legalább egy kiválasztott él illeszkedik, azt egyetlen klózzal le lehet írni: $(x_1 \vee x_2 \vee \dots \vee x_k)$, ha u -ra az x_1, \dots, x_k változókkal címkézett élek illeszkednek. Ez a klóz akkor lesz igaz, ha legalább egy változó igaz, azaz ha legalább egy kiválasztott él van ebben a k darabban.
- Azt, hogy az u csúcsra legfeljebb egy kiválasztott él illeszkedik, kicsit hosszadalmasabb leírni: a „legfeljebb egy” azt jelenti, hogy „nem ez a kettő egyszerre” és „nem ez a kettő egyszerre” és... és ciklusban végigmenve minden élpáron, ami u -ra illeszkedik. Röviden $\bigwedge 1 \leq i < j \leq k \neg(x_i \wedge x_j)$, ha az x_1, \dots, x_k változókkal vannak címkézve az u -ra illeszkedő élek. DeMorgan-azonosságot használva ez $(\neg x_i \vee \neg x_j)$ alakú klózekből „nagyon sok”.

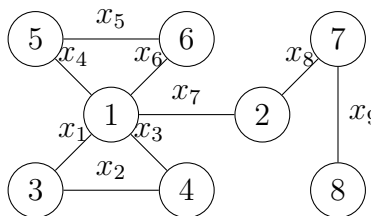
De szerencsére csak négyzetes sok, tehát polinomidőben előállítható: nyilván egy N -csúcsú és M -élű gráfban csak $O(M^2)$ ilyen klózt tudunk létrehozni.

Ha ezt a (nagy) CNF-et létrehozzuk, az pont visszavezetés (ezt most onnan látjuk, hogy elmagyaráztuk, mit jelent a formula és látszik, hogy pontosan akkor lesz igaz, ha a változóértékekadás olyan élhalmaz kiválasztásának felel meg, aminek a gráf minden csúcsára pontosan egy-egy eleme illeszkedik.

Nézzünk erre a fentire egy példát: ha a gráf már megint a



akkor felcímkézve a változókkal ezt kapjuk:



(adtunk a csúcsoknak is nevet közben)

Ha szép sorban megyünk és először az 5. csúcs körül írjuk fel a „pontosan egy változó igaz” feltételt, akkor kapjuk a $(x_4 \vee x_5)$ (legalább egy) és a $(\neg x_4 \vee \neg x_5)$ (legfeljebb egy) klózeket. A 6, 7, 3, 4, 2, és 7. csúcsokkal teljesen hasonló módon kapjuk a $(x_5 \vee x_6) \wedge (\neg x_5 \vee \neg x_6)$ stb. CNF-eket.

A 8. csúcsnál a „legalább egy” az egy x_9 (egység)klózt fog eredményezni, a „legfeljebb egy” pedig nem ad új klózt (mert nincs két különböző változó, amiken amúgy ciklusban mennénk végig). Így ez a csúcs csak egy x_9 klózzal járul majd a végső CNF-hez.

Az 1. csúcs viszont, mivel öt a fokszáma, a többiekénél többet fog behozni: a „legalább egy” klóz egyszerűen $(x_1 \vee x_3 \vee x_4 \vee x_6 \vee x_7)$, viszont a „legfeljebb egy” kitételhez felvesszünk a konstrukció szerint négyzetes sokat: $(\neg x_1 \vee \neg x_3), (\neg x_1 \vee \neg x_4), \dots, (\neg x_6 \vee \neg x_7)$, összesen $\binom{5}{2} = 10$ klóz. Mindösszesen a formula, amit az eredeti gráfból konvertáltunk:

$$\begin{aligned} & (x_4 \vee x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \\ & \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_7 \vee x_8) \wedge (\neg x_7 \vee \neg x_8) \wedge (x_8 \vee x_9) \wedge (\neg x_8 \vee \neg x_9) \\ & \wedge x_9 \wedge (x_1 \vee x_3 \vee x_4 \vee x_6 \vee x_7) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_6) \wedge (\neg x_1 \vee \neg x_7) \\ & \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_6) \wedge (\neg x_3 \vee \neg x_7) \wedge (\neg x_4 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_7) \wedge (\neg x_6 \vee \neg x_7). \end{aligned}$$

Szép hosszú. Mindenesetre amire ez épp jó lehet: ha a PÁROSÍTÁS problémára kell egy solvert írjunk, akkor ezt megtehetjük úgy, hogy ezt a nem túl bonyolult konverziós algoritmust megírjuk, ez outputra tesz nekünk egy CNF-et, letöltünk egy SAT solvert a netről, annak beadjuk ezt a CNF-et és a válaszból megmondjuk, hogy az eredeti gráfban van-e teljes párosítás vagy nincs.

Egy valamirevaló SAT solver pl. erre a fenti formulára azt reagálja, hogy az x_9 egységklóz miatt $x_9 = 1$ kell legyen, ezzel rezolválja a $(\neg x_8 \vee \neg x_9)$ klózt, kapja a $\neg x_8$ klózt, tehát $x_8 = 0$, rezolválja az $(x_7 \vee x_8)$ klózt a $\neg x_8$ -cal, kapja, hogy x_7 , tehát $x_7 = 1$, ezzel az x_7 klózzal rezolválja mind a négy olyan klózt, amiben van $\neg x_7$, kapja a $\neg x_1, \neg x_3, \neg x_4, \neg x_6$ klózeket, aztán ezekkel rezolválva az $x_4 \vee x_5$ -öt és az $x_1 \vee x_2$ -t kapja, hogy x_2 és x_5 pedig 1 kell legyen. Közben eldobálja a kielégített klózeket és ekkor rájön, hogy ennek a formulának az $x_9 = x_7 = x_5 = x_2 = 1, x_1 = x_3 = x_4 = x_6 = x_8 = 0$ egy kielégítő értékadása és ezt visszaadja. Mi pedig ebből megtudjuk, hogy a gráfban van teljes párosítás (mert a formulára a SAT solver azt mondta, hogy kielégíthető). Ha a solver még egy megoldást is visszaad ilyenkor, akkor ha a visszavezetés „szép”, mint ez is, abból még az eredeti problémára is tudunk egy „megoldást” adni, jelen esetben egy kielégítő értékadásból vissza tudunk kapni egy teljes párosítást is.

De a visszavezetést [ezen a kurzuson](#) általában nem erre fogjuk használni, hanem arra, hogy kiindulva abból, hogy A nehéz, azzal együtt, hogy $A \leq B$, kapjuk, hogy B is nehéz. Az első „nehéz” kategóriánk pedig az eldönthetetlen problémáké lesz, amikre egyáltalán nincs eldöntő algoritmus.

6. Eldönthetlenség

Ebben a részben lesz olyan is, hogy egy RAM program egy másik RAM program forráskódját várja inputként, nem akármilyen számsorozatot; ezzel nincs baj, hiszen minden forráskódot felfoghatunk akár mint (akár az angol ábécé feletti) stringet, a program karaktereit azoknak (mondjuk) az ASCII kódjára cserélve máris egy számsorozat reprezentálja a forráskódot. Vagy, ahogy azt assemblyből láttuk, minden utasítás-mnemonicához rendelhetünk egy opcode-ot, és ezzel is reprezentálhatjuk a programot, a konkrét ábrázolás (csakúgy, mint a gráfoknál) teljesen mindegy, a lényeg, hogy van értelme olyan programnak, ami egy programkódot vár inputként.

Nem minden program **dönt el** egy problémát. Emlékszünk: az M program eldönti az A problémát, ha az $x \in A$ inputokra $M(x) = \text{igen}$ (vagy ACCEPT, ízlés dolga), és az $x \notin A$ inputokra $M(x) = \text{nem}$. Bizonyos esetekben ennél csak **gyengébb** programokat tudunk írni, olyanokat, amik nem eldöntenek, csak **felismernek** egy problémát.

Definíció: (Turing-)Felismerés.

Az M program **felismeri** az A problémát, ha az $x \in A$ inputokra $M(x) = \text{igen}$, az $x \notin A$ inputokra pedig $M(x) = \nearrow$.

Tehát eldöntés és felismerés közt az a különbség, hogy a **nem** példányokon eldöntés esetében **nem**-mel kell megállni a programnak, felismerés esetében pedig végtelen ciklusba esünk. A „felismerés” szó helyett használjuk még a „Turing-felismerés” és a „félig eldöntés” kifejezéseket is, ezek szinonimák. Egy probléma (Turing-) **felismerhető** vagy **rekurzívan felsorolható** (recursively enumerable; ezek is szinonimák, vagy „félig eldönthető”), ha van őt felismerő program.

Definíció: RE

A rekurzívan felsorolható problémák osztályát **RE** jelöli.

Van összefüggés a két osztály közt:

Állítás

$$\mathbf{R} \subseteq \mathbf{RE}.$$

Bizonyítás

Legyen A egy eldönthető probléma, melyet eldönt az M program. Cseréljük ki az összes REJECT sort egy végtelen ciklust generáló sorra (pl. `while(1){}`-re), ekkor az így kapott M' program már csak felismeri a problémát, tehát A felismerhető.

Egy másik transzformáció, amit az A problémát eldöntő M programmal tudunk csinálni, az az, hogy az ACCEPT sorok helyett REJECT-et írunk és fordítva. Ekkor az így kapott M' program nyilván az A probléma komplementerét fogja kapni. Ezért:

Állítás

Ha A eldönthető, akkor komplementere is az.

Mielőtt rátérnénk első eldönthetetlen problémánkra, ismerkedjünk meg a **diagonális bizonyítás módszerével**:

Ha veszünk egy $A_{i,j}$ bináris mátrixot, ahol $i, j \in I$ ugyanazzal a(z akár végtelen!) halmazzal vannak indexelve, és komplementáljuk az átlóját – azaz vesszük az $a_i := \neg A_{i,i}$ sorozatot –, akkor egy olyan $(a_i)_{i \in I}$ (esetleg végtelen!) bináris stringet kapunk, ami különbözik a mátrix összes sorától.

Bizonyítás

Az $(a_i)_{i \in I}$ negált átló a j . sortól különbözik a j . pozíción, minden j -re.

Pl. ha a mátrixunk a $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, akkor a negált átló az $(1, 1, 0)$ vektor, ami tényleg nem szerepel a mátrix soraként. (Oszlopaként sem, de ezt most nem használjuk.)

A diagonális bizonyításoknál az „okos” húzás az, amikor definiáljuk, hogy éppen melyik bináris mátrixot akarjuk nézegetni. Első eldönthetetlen problémánk eldönthetetlenségének bizonyításakor is ezt fogjuk alkalmazni. A probléma pedig a SAJÁT FORRÁSKÓDON MEGÁLLÁS:

Definíció: SAJÁT FORRÁSKÓDON MEGÁLLÁS.

Input: egy M program. (A forráskódja, nyilván.)

Output: megáll-e M , ha a saját forráskódját kapja meg inputként?

Állítás

A SAJÁT FORRÁSKÓDON MEGÁLLÁS probléma eldönthetetlen.

Bizonyítás

Készítsük el a következő T (végtelen) bináris mátrixot: a sorait és az oszlopait is a **forráskódokkal** címkézzük! Tehát minden M és N forráskódra $T_{M,N}$ egy bit lesz.

Mégpedig: legyen $T_{M,N} = 1$ akkor, ha M megáll, ha N -t kapja meg mint inputot, és 0, ha $M(N) = \nearrow$.

A diagonális bizonyítás arról szól, hogy ennek a mátrixnak az átlójának a komplementere nem egyezik meg a mátrix egyik sorával sem. Mit jelent ez most?

A mátrix átlója: $T_{M,M}$ akkor 1, ha M megáll M -en és 0, ha nem. Vagyis ez pontosan a SAJÁT FORRÁSKÓDON MEGÁLLÁS probléma! Ennek a komplementere, jelölje ezt D , az a „vektor”, amire $D(M) = 1$, ha M nem áll meg M -en, és $D(M) = 0$, ha M megáll M -en. A diagonális módszer szerint D nem egyezik meg a mátrix egyik sorával sem.

Ha a SAJÁT FORRÁSKÓDON MEGÁLLÁS probléma eldönthető lenne, akkor komplementere is. Mondjuk komplementerét döntse el a H program. Erre a programra $H(M) = 1$, ha M nem áll meg M -en és $H(M) = 0$, ha M megáll M -en. Most ebből a H programból eldöntő program helyett készítsünk egy H' felismerő programot (láttuk, hogy ilyet is lehet); erre a H' programra $H'(M) = 1$, ha M nem áll meg M -en és $H'(M) = \nearrow$, ha M megáll M -en.

Ennek a H' -nek mint programnak ha a $T_{M,N}$ mátrixban megnézzük a sorát, akkor ott eszerint pontosan D -t kellene látnunk! Hiszen $T_{H',M}$ akkor 1, ha M nem áll meg M -en és 0, ha M megáll M -en. De azt már láttuk, hogy D nem lehet T -nek sora, hiszen az átló komplementere, tehát H' nem létezik, ami H komplementere volt, tehát H se létezhet, azaz **a SAJÁT FORRÁSKÓDON MEGÁLLÁS probléma nem eldönthető.**

Kiindulva abból, hogy a SAJÁT FORRÁSKÓDON MEGÁLLÁS probléma eldönthetetlen, a nála általánosabb MEGÁLLÁS probléma eldönthetlenségét is megkapjuk:

Definíció: MEGÁLLÁS.

Input. Egy M program és egy x inputja.

Output. Megáll-e M az x -en?

Állítás

A MEGÁLLÁS probléma eldönthetetlen.

Bizonyítás

Ehhez elég megmutatni, hogy SAJÁT FORRÁSKÓDON MEGÁLLÁS \leq_R MEGÁLLÁS. Tehát: egy M forráskódból kell készítenünk egy M' kódot és egy x inputját úgy, hogy $M(M) = \nearrow \Leftrightarrow M'(x) = \nearrow$.

Ez könnyű: legyen $M' = M$ és $x = M$. Erre nyilván teljesül a feltétel.

Még néhány problémáról megmutatjuk visszavezetéssel, hogy eldönthetetlenek, hogy ráérezzünk a technikára:

Definíció: MINDENEN MEGÁLLÁS.

Input. Egy M program.

Output. Igaz-e, hogy M minden lehetséges bemenetén megáll?

Állítás

A MINDENEN MEGÁLLÁS probléma eldönthetetlen.

Bizonyítás

Visszavezetjük a problémára a MEGÁLLÁS problémát, amiről már tudjuk, hogy eldönthetetlen. Ehhez adnunk kell egy $(M, x) \mapsto M'$ átalakítást, melyre igaz, hogy

$$M \text{ megáll } x\text{-en} \Leftrightarrow M' \text{ minden lehetséges inputon megáll.}$$

Mégpedig úgy, hogy M -ből és x -ből az M' -t „mechanikus” (=rekurzív) módon megkap-hassuk. Ez pl. ilyen kód:

```
bool M'( y )
    M(x)
    return true
```

Hiszen ha M' bármilyen y -t is kap, nem is foglalkozik vele, csak futtatja M -et x -en. Ha M megáll x -en, akkor ez előbb-utóbb véget ér, és M' visszatér TRUE-val (tehát ekkor valóban megáll minden), ha pedig nem, akkor M' végtelen ciklusba fog esni minden y inputon

(tehát ekkor nem igaz az, hogy minden lehetséges inputon megáll). Ezért ez a konverzió tartja a választ is, tehát tényleg visszavezetés.

A fenti probléma (mint a SAJÁT FORRÁSKÓDON MEGÁLLÁS is) egy-egy olyan eldönthetetlen probléma, melynek inputja egy M forráskód, és ennek a kódnak a **viselkedéséről** kell eldönteni valamit (az egyik esetben azt, hogy mindig megáll-e, a másikon azt, hogy a saját kódján mint inputon megáll-e).

Ha egy eldöntési kérdés **triviális**, ami itt azt jelenti, hogy **minden forráskódra ugyanaz a válasz** (tehát vagy minden M -re azt kell mondjuk, hogy **igen**, vagy minden M -re azt kell mondjuk, hogy **nem**), az nyilván eldönthető: az algoritmus meg se nézi a forráskódot, csak outputra teszi a választ.

Rice tétele azt mondja, hogy ennél bonyolultabb dolgokat nem lehet eldönteni:

Állítás: Rice tétele

A rekurzívan felsorolható problémák egyetlen nemtriviális tulajdonsága sem dönthető el.

Bizonyítás

Ha M egy forráskód, jelölje $L(M)$ az M által elfogadott inputok halmazát: $L(M) = \{x : M(x) = \text{igen}\}$. Nyilván $L(M) \in \mathbf{RE}$, hiszen ha egy program nem „igen”-nel áll meg, ahelyett végtelen ciklusba tudjuk küldeni és az így kapott M' program épp $L(M)$ -et fogja felismerni.

Rice tétele formálisan azt mondja, hogy ha van egy \mathcal{L} osztályunk, amire $\emptyset \subsetneq \mathcal{L} \subsetneq \mathbf{RE}$ (azaz: \mathcal{L} -ben rekurzívan felsorolható **problémák** vannak, legalább egy, de nem az összes, ettől nemtriviális; ezek azok a problémák, amik az épp aktuális „tulajdonsággal” rendelkeznek, az $\mathbf{RE} - \mathcal{L}$ -beliek pedig nem), akkor az nem eldönthető, hogy input M kódra teljesül-e $L(M) \in \mathcal{L}$.

Feltehetjük, hogy $\emptyset \notin \mathcal{L}$: ha az aktuálisan vizsgált tulajdonságunkkal rendelkezik az olyan M program, ami semmit nem fogad el (ekkor igaz, hogy $L(M) = \emptyset$), akkor nézzük inkább ennek a tulajdonságnak a **komplementerét**. (Ez is nemtriviális lesz, és ebben már nincs benne \emptyset . Tudjuk, hogy ha a komplementer nem eldönthető, akkor az eredeti probléma sem, tehát erről is elég belátnunk, hogy nem eldönthető.)

Mivel \mathcal{L} nemtriviális, van olyan N program, aki rendelkezik ezzel a tulajdonsággal, amire $L(N) \in \mathcal{L}$.

Most már megvan mindenünk, hogy a MEGÁLLÁST visszavezessük az $L(M') \stackrel{?}{\in} \mathcal{L}$ problémára: egy input M programból és x inputjából készítsük a következő kódot:

```
bool M'(y)
    M(x)
    return N(y)
```

Ez az M' kód mivel ekvivalens? Ha M megáll x -en, akkor az a sor csak egy hosszú nop, nem csinál semmit, és M' ugyanazt csinálja, mint N . Tehát ekkor $L(M') \in \mathcal{L}$, ezért ekkor el kéne fogadni az M' -t.

Ha pedig M nem áll meg x -en, akkor M' nem áll meg semmit, tehát ekkor $L(M') = \emptyset \notin \mathcal{L}$, ezért ekkor pedig el kéne utasítani az M' -t.

Tehát (mivel N rögzített, csak a tulajdonságtól függő program) ezt a transzformációt végre is tudjuk hajtani és a választ is tartja, azaz visszavezeti a megállási problémát erre a tetszőlegesre, ami igazolja a Rice tételt.

Az univerzális programozási nyelveket azért hívják így, mert minden rekurzív algoritmust lehet bennük implementálni. Nem meglepő, hogy pl. RAM interpretert is lehet írni RAM nyelven, azaz egy olyan (mondjuk) U programot, ami inputként vár egy M programkódot és annak egy x inputját, és lényegében „futtatja” M -et x -en, azaz $U(M, x) = M(x)$. Ehhez csak azt kell meggondolnunk, hogy ha mondjuk M kódját utasításonként egy regiszterben kapja meg egy számként tárolva, akkor U -nak egy munkaregiszterben elég eltárolnia az interpretált M program aktuális utasításának a sorszámát, azt kiolvasni, dekódolni (ehhez konstans sok regiszter biztos elég, a nagyon kevésre redukált utasításszám miatt), végrehajtani és lépni tovább. A végrehajtáshoz pedig csak annyi pluszt kell elvégezni, hogy ha M az input regiszterekből olvasna, akkor az x rész megfelelőedik regiszteréből olvasson, ha meg a working regisztereket használná, akkor ne legyen átfedés a szimuláláshoz kellő néhány regiszter és a szimulált program által használt regiszterek közt, de mindkét probléma megoldható egy egyszerű offsettel arrébb címzéssel minden egyes direkt/indirekt címzés esetén.

Mindenesetre a lényeg az, hogy **lehetséges** ilyen kódot írni. Ebből a tárgyból ezt az interpretert **univerzális programnak** hívjuk.

Definíció: Univerzális program

Az univerzális program az az U program, mely inputként egy M forráskódot és annak egy x inputját várja és szimulálja M -et x -en, azaz $U(M, x) = M(x)$.

Később (a hierarchia tételeknél) lényeges lesz, hogy a szimuláció során U csak konstansszor annyit lép, mint M és csak konstansszor annyi tárat használ – azaz ha pl. M egy $f(n)$ időkorlátos gép, akkor a szimuláció $O(f(n))$ időben zajlik^a.

^alásd Függelék

Az univerzális programot fel tudjuk használni arra is, hogy belássuk, a végtelen ciklus néha elkerülhetetlen:

Állítás

$$\mathbf{R} \subsetneq \mathbf{RE}.$$

Például a MEGÁLLÁS félig eldönthető, de nem eldönthető probléma.

Bizonyítás

Azt már láttuk, hogy a megállási probléma nem dönthető el. Viszont felismerhető az U interpreter segítségével:

```
bool MegallasFelismer( M, x )
    U(M,x)
    return true
```

Ez a program először futtatja M -et x -en (van különbség az eddigiek és e közt! itt az M és

az x paramétere az input programnak, az előzőekben pedig csak oda kellett másoljuk, az input valami y volt. Itt mivel paraméter, $M(x)$ helyett $U(M, x)$ -et tudunk írni), ami ha végtelen ciklusba esik (vagyis $M(x) = \nearrow$), akkor ez is, és ha M megáll x -en, akkor **true**-t ad vissza. Ez pontosan azt jelenti, hogy felismeri, ha M megáll x -en.

Vannak egyéb eldönthetetlen problémák is, melyek eldönthetetlenségét nem bizonyítjuk, csak azért írunk róluk, hogy lássuk, nem kell, hogy „program” is legyen az inputban.

Állítás: Matijasevič

Hilbert 10. problémája eldönthetetlen.

Definíció: Post megfelelezési problémája, Post Correspondence Problem, PCP

Input: dominó típusok véges halmaza. Egy dominó típus $\begin{pmatrix} u \\ v \end{pmatrix}$ alakú, ahol $u, v \in \{0, 1\}^*$ (bináris stringek).

Output: Lerakhatóak-e egymás mellé ezek a dominók (mindegyik típusból rendelkezésre áll tetszőleges sok) úgy, hogy a felső sorban összeolvasott string ugyanaz legyen, mint az alsó sorban levő?

Példa: ha a dominó típusok $\begin{pmatrix} 01 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 110010 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 1 \\ 1111 \end{pmatrix}$ és $\begin{pmatrix} 11 \\ 01 \end{pmatrix}$, akkor van megoldás: a

$$\begin{pmatrix} 01 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1111 \end{pmatrix} \begin{pmatrix} 110010 \\ 0 \end{pmatrix} \begin{pmatrix} 11 \\ 01 \end{pmatrix} \begin{pmatrix} 11 \\ 01 \end{pmatrix} \begin{pmatrix} 1 \\ 1111 \end{pmatrix}$$

lepakolás esetén alul is, felül is a 01111001011111 szót kapjuk. Tehát ez a PCP-nek egy igen példánya.

Állítás

Post megfelelezési problémája eldönthetetlen.

Ez a probléma is félig eldönthető, hiszen ha van megoldás, akkor előbb kipróbálva az összes egy-dominós, aztán az összes két-dominós, ... lerakást, ha van megoldás, azt előbb-utóbb megtaláljuk.

Egy másik probléma a WANG-CSEMPÉZÉS:

Definíció: WANG-CSEMPÉZÉS

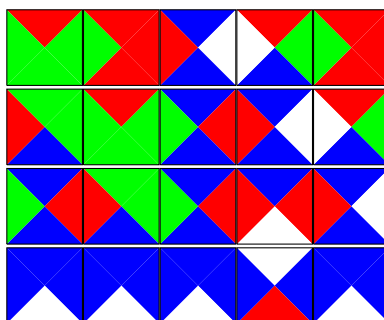
Input: Egységnégyzet alakú csempe-típusoknak egy véges halmaza, minden típusnak mind a négy éle valamilyen színű.

Output: Parkettázható-e az egész sík ilyen csempékkel úgy, hogy szomszédos csempék találkozó élei azonos színűek legyenek? (A csempék nem forgathatóak.)

Példa. Ha a következő csempéink vannak:









akkor ezekkel lefedhető a sík. Egy véges parkettázás például:



Persze hogy ezt ki lehet-e terjeszteni az egész síkra, az ezen a ponton még nem világos...

Láthatjuk, hogy itt még azt is nehéznek tűnik felismerni, hogy a parketta-készlettel lefedhető-e a sík és arra is kevés ötletünk van, hogy vajon miért lehetne kizárni egy parketta-készletet.

Ha valaki elgondolkodik az alsó sor középső elemén, az alá vagy egy , vagy egy  csempe kerülhet (azoknak fehér a teteje). Ha -t teszünk alá, akkor jobbra nem tudjuk folytatni, mert nincs olyan elem, ami bal oldalon kék, fenn pedig piros. Ha -t, akkor tudjuk folytatni jobbra: egyetlen elem lesz, ami balról fehér, fent piros, a . Ha viszont az kerül a negyedik elem alá, akkor a jobb alsó sarokba egy felül fehér, bal oldalt zöld elem kellene, olyan pedig nincs... tehát ezt a részleges parkettázást (vagy bármelyiket, amiben a  csempék így hárman egymás mellett vannak) nem lehet kiterjeszteni az egész síkra...

De le lehet fedni ezzel a készlettel a síkot, akit érdekel, nézze meg itt, hogy hogyan.

A csempekészlet érdekessége, hogy ezzel nem lehet **periodikusan** lefedni a síkot, ez azt jelenti, hogy nem lehet olyan (nagy, de véges) téglalapot kirakni velük, aminek a bal oldalán a színsor ugyanaz, mint a jobb oldalán, és az alján levő színsor ugyanaz, mint a tetején. (Ha lehet ilyen téglalapot kirakni, akkor ilyen téglalapokat egymás mellé téve az egész síkot is le lehet, ez világos. Ezzel a csempekészlettel le lehet ugyan fedni a síkot, de így nem, csak **aperiodikusan**.)

Állítás

A WANG-CSEMPÉZÉS eldönthetetlen probléma.

A rekurzívan felsorolható problémákat okkal hívjuk „félig eldönthető” problémáknak is: ha egy probléma és a komplementere is félig eldönthető, akkor a probléma (teljesen) eldönthető.

Hogy ezt tömörebben megfogalmazzuk, bevezetjük a $\text{co}\mathcal{C}$ jelölést, ahol \mathcal{C} egy problémaosztály:

Definíció: $\text{co}\mathcal{C}$

Ha \mathcal{C} problémák egy osztálya, akkor $\text{co}\mathcal{C} := \{\bar{A} : A \in \mathcal{C}\}$ a \mathcal{C} -beli problémák **komplementereinek** az osztálya.

Például coRE -ben azok a problémák vannak, amiknek **komplementere** rekurzívan felsorolható, coR -ben azok, melyeknek komplementere rekurzív. . .

. . .ja, azok persze rekurzívak is, tehát $\text{coR} \subseteq \text{R}$. Sőt $\text{coR} = \text{R}$, mert:

Állítás

A co operátor monoton: ha $\mathcal{C}_1 \subseteq \mathcal{C}_2$, akkor $\text{co}\mathcal{C}_1 \subseteq \text{co}\mathcal{C}_2$.

Továbbá $\text{coco}\mathcal{C} = \mathcal{C}$.

Állítás

Ha $\mathcal{C} \subseteq \text{co}\mathcal{C}$, akkor $\mathcal{C} = \text{co}\mathcal{C}$.

Bizonyítás

Ha $\mathcal{C} \subseteq \text{co}\mathcal{C}$, akkor a monotonitás miatt co -t alkalmazva mindkét oldalon $\text{co}\mathcal{C} \subseteq \text{coco}\mathcal{C} = \mathcal{C}$, tehát egyenlőek.

A kompaktabb összefüggés R és RE közt pedig:

Állítás

$$\text{R} = \text{RE} \cap \text{coRE}.$$

Bizonyítás

$\text{R} \subseteq \text{RE} \cap \text{coRE}$: ha A rekurzív, akkor rekurzívan felsorolható is, tehát $\text{R} \subseteq \text{RE}$ (ezt tudtuk). Továbbá ha A rekurzív, akkor \bar{A} is rekurzív, tehát \bar{A} rekurzívan felsorolható is, ami pont azt jelenti, hogy $A \in \text{coRE}$, így $\text{R} \subseteq \text{coRE}$. Összerakva a kettőt $\text{R} \subseteq \text{RE} \cap \text{coRE}$.

$\text{RE} \cap \text{coRE} \subseteq \text{R}$: Ha A rekurzívan felsorolható és komplementere is az, akkor van egy M_1 program, ami felismeri A -t és egy M_2 , ami felismeri \bar{A} -t. Csak annyit kell tennünk, hogy egy input x -re párhuzamosan futtatjuk M_1 -et és M_2 -t (azaz egy lépést az egyik program tesz, egyet a másik, megint egyet az egyik, . . .). Ha $x \in A$, akkor mivel M_1 felismeri A -t, előbb-utóbb $M_1(x) = \text{Accept}$ -et kapunk, ekkor adjunk vissza **Accept**-et. Ha pedig $x \notin A$, azaz $x \in \bar{A}$, akkor mivel M_2 felismeri a komplementer problémát, így ekkor előbb-utóbb $M_2(x) = \text{Accept}$ -et kapunk. Ekkor adjunk vissza **Reject**-et. Ez az algoritmus mindig megáll és mindig helyes válasszal, tehát ekkor A is rekurzív.

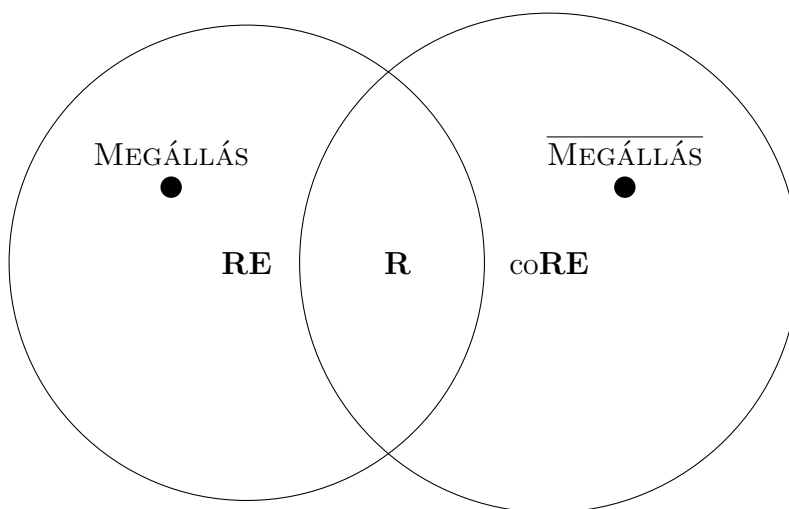
Ebből az is következik, hogy

Állítás

$$\mathbf{RE} \neq \mathbf{coRE},$$

hiszen ha egyenlők lennének, akkor metszetük is **RE** lenne, pedig a metszetükről azt tudjuk, hogy **R**, amiről tudjuk, hogy $\mathbf{R} \neq \mathbf{RE}$.

Tehát a fenti három osztály tartalmazkodási diagramja kb. így néz ki:



Ezt nemsokára kicsit pontosítani fogjuk.

Nehézség, teljesség

Ebben a részben megpróbáljuk megfogalmazni, hogy mi is az: egy bonyolultsági osztály **legnehezebb problémái**. Kiderül, hogy az **RE** osztály egyik legnehezebb problémája épp a **MEGÁLLÁS** lesz.

Definíció: \mathcal{C} -nehéz, \mathcal{C} -teljes problémák

Ha \mathcal{C} problémák egy osztálya, akkor az A probléma...

- **\mathcal{C} -nehéz**, ha minden \mathcal{C} -beli probléma visszavezethető A -ra;
- **\mathcal{C} -teljes**, ha A még ráadásul \mathcal{C} -ben is van.

Rajzban általában azt a konvenciót követjük, hogy a „krumpli”ként rajzolt osztálynak a „széle” fele vannak az egyre nehezebb problémái: ekkor a \mathcal{C} -teljes problémák vannak a \mathcal{C} „héján”, a \mathcal{C} -nehezek pedig a héján vagy \mathcal{C} -n kívül²¹

Egy **RE**-teljes probléma például **RE**-beli, és minden **RE**-beli probléma visszavezethető rá. Láttunk már egyet:

²¹Ez több ponton vérzik, de intuíciónak jó lesz első közelítésben.

Állítás

A MEGÁLLÁS egy **RE**-teljes probléma.

Bizonyítás

Azt már láttuk, hogy $\text{MEGÁLLÁS} \in \mathbf{RE}$.

Legyen $A \in \mathbf{RE}$ egy rekurzívan felsorolható probléma; meg kell mutatnunk, hogy $A \leq \text{MEGÁLLÁS}$. Tehát adnunk kell egy olyan konverziót, ami az A inputjából (legyen ez az input mondjuk x) készíti a megállási probléma egy inputját, mondjuk M -et és y -t úgy, hogy $x \in A$ pontosan akkor legyen igaz, ha M megáll y -on.

A -ról csak annyit tudunk, hogy **RE**-beli, de ez épp elég: azért **RE**-beli, mert van egy olyan M program, ami felismeri őt, amire tehát $M(x) = \text{ACCEPT}$, ha $x \in A$ és $M(x) = \nearrow$, ha $x \notin A$. Tehát

$$x \in A \Leftrightarrow M \text{ megáll } x\text{-en,}$$

azaz az $x \mapsto (M, x)$ egy (lineáris idejű) választartó inputkonverzió A -ról a MEGÁLLÁSra.

Ez például azt is jelenti, hogy $\overline{\text{MEGÁLLÁS}}$ pedig **coRE**-teljes, hiszen:

Állítás

Ha $A \leq B$, akkor $\overline{A} \leq \overline{B}$.

Bizonyítás

Ha f egy visszavezetés A -ról B -re, akkor f visszavezetés \overline{A} -ról \overline{B} -re is, hiszen

$$x \in \overline{A} \Leftrightarrow x \notin A \Leftrightarrow f(x) \notin B \Leftrightarrow f(x) \in \overline{B}.$$

Állítás

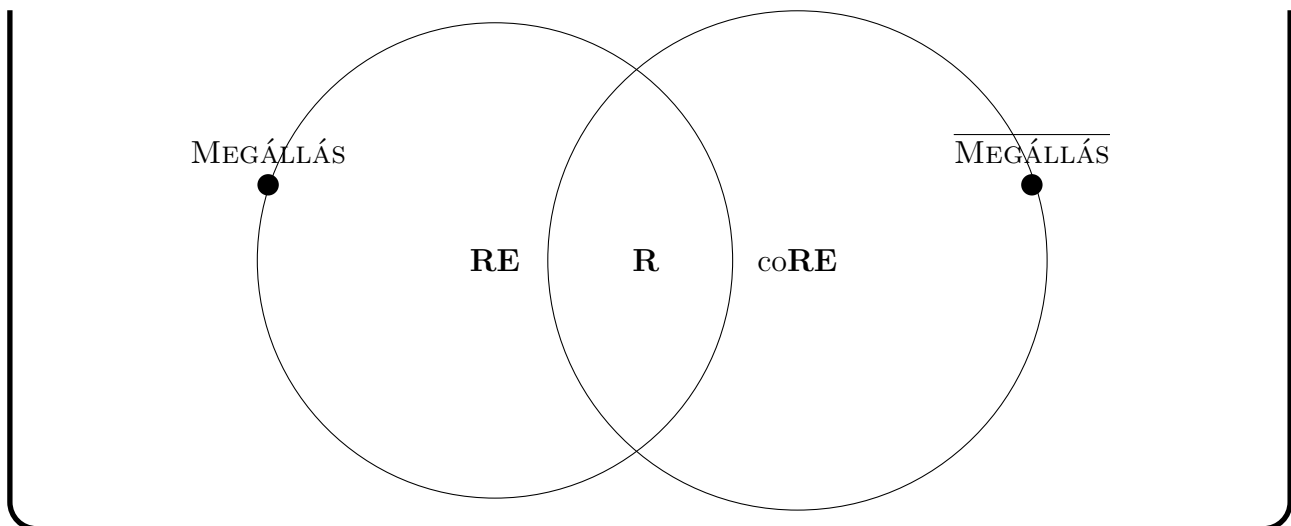
Ha A egy \mathcal{C} -nehéz probléma, akkor \overline{A} egy **coC**-nehéz probléma.

Bizonyítás

Ha A \mathcal{C} -nehéz és $B \in \mathbf{coC}$, akkor $\overline{B} \in \mathcal{C}$ (a **co** definíciója miatt), így $\overline{B} \leq A$ (mert A \mathcal{C} -nehéz), így $B \leq \overline{A}$ (előző állításból, komplementáltuk mindkét oldalt), tehát \overline{A} tényleg **coC**-nehéz.

Ezért pl. a $\overline{\text{MEGÁLLÁS}}$ az **coRE**-nehéz is. Persze mivel $\text{MEGÁLLÁS} \in \mathbf{RE}$, így komplementere **coRE**-ben van, azaz **coRE**-teljes.

Tehát a három osztály inkább így néz ki lerajzolva:



Sőt, olyan problémát is tudunk már mondani, ami ezeken az osztályokon is **kívül** van:

Definíció: EKVIVALENCIA

Input: M_1 és M_2 programok.

Output: **Ekvivalens-e** a két program? (Azaz: minden y -ra $M_1(y) \stackrel{?}{=} M_2(y)$)

Állítás

$\text{MEGÁLLÁS} \leq \text{EKVIVALENCIA}$ és $\overline{\text{MEGÁLLÁS}} \leq \text{EKVIVALENCIA}$.

Bizonyítás

Az első visszavezetéshez jó lesz:

```
bool M1( y )    bool M2( y )
  M(x);          return true
  return true
```

hiszen ha M megáll x -en, akkor mindkét program mindenre **true**-t ad, ekkor ekvivalensek, ha pedig M nem áll meg x -en, akkor $M1$ mindenre végtelen ciklusba esik, $M2$ pedig mindenre **true**-t ad, ekkor nem ekvivalensek.

A második visszavezetéshez pedig jó lesz:

```
bool M1( y )    bool M2( y )
  M(x);          while(1){}
  return true
```

hiszen ha M nem áll meg x -en, akkor mindkét program mindenre végtelen ciklusba esik, ekkor tehát ekvivalensek, ha pedig M megáll x -en, akkor az első program mindenre **true**-t ad vissza, a második pedig mindenre végtelen ciklusba esik, ekkor nem ekvivalensek.

Nemsokára látni fogjuk, hogy ez így azt eredményezi, hogy az EKVIVALENCIA kívül van mindkét osztályon, tehát **nagyon** nehéz probléma.

De előbb meg kell ismerjünk azt a tulajdonságot, ami egy problémaosztályt **bonyolultsági osztállyá** tesz:

Definíció: Visszavezetésre való zártság

A \mathcal{C} problémaosztály **zárt a visszavezetésre**, ha minden $A \in \mathcal{C}$ és $B \leq A$ esetén $B \in \mathcal{C}$ is igaz.

Tehát akkor zárt egy osztály a visszavezetésre, ha minden A eleménél tartalmazza az összes „könnyebbet” is. (Nem „lyukas”, ha már az ábrás hasonlatnál tartunk.)

Például,

Állítás

Az **R**, **RE** és **P** osztályok zártak a visszavezetésre.

Bizonyítás

R: Legyen A rekurzív probléma és $B \leq A$. Konkrétan legyen f egy visszavezetés B -ről A -ra, M pedig az A -t eldöntő algoritmus. Akkor B -t eldönti az $N(x) := \text{return } M(f(x))$ algoritmus, tehát B is rekurzív.

RE: Ha A rekurzívan felsorolható, mondjuk felismeri az M algoritmus és f egy visszavezetés B -ről A -ra, akkor az $N(x) := \text{return } M(f(x))$ algoritmus felismeri, tehát B is rekurzívan felsorolható.

P: Ha A polinomidőben eldönthető, mondjuk az M algoritmussal és f egy visszavezetés B -ről A -ra, akkor az $N(x) := \text{return } M(f(x))$ algoritmus eldönti, ezt tudjuk; ez az algoritmus polinomidejű is lesz, mert az f egy polinomidejű, mondjuk n^k időkorlátos algoritmus kimenetre nem tud túl nagy, csak legfeljebb n^{2k} **méretű** outputot generálni^a. Ha ezen az outputon lefuttatjuk az M algoritmust, ami mondjuk n^ℓ időkorlátos, akkor az legfeljebb $(n^{2k})^\ell = n^{2k\ell}$ lépésben megáll. Tehát összesen a két algoritmus $n^k + n^{2k\ell}$ lépésben megáll, ami polinom, hiszen k, ℓ konstansok.

^aEz azért van, mert a RAM gép t lépésben legfeljebb t -bites számokat tud előállítani, tehát n^k lépésben legfeljebb n^k darab, egyenként legfeljebb n^k -bites számot tud outputra írni, összesen tehát $f(x)$ legfeljebb $|x|^{2k}$ -bites lesz.

Ezzel a legutóbbival egyébként azt is beláttuk, hogy két polinom időkorlátos algoritmus kompozíciója még mindig polinom időkorlátos lesz. Ez jó, mert emiatt:

Állítás

A hatékony visszavezetés **transzitív**: ha $A \leq B$ és $B \leq C$, akkor $A \leq C$.

Ami pedig azért jó, mert ha van egy \mathcal{C} -nehéz problémánk, akkor abból lehet gyártani többet is:

Állítás

Ha A egy \mathcal{C} -nehéz probléma és $A \leq B$, akkor B is \mathcal{C} -nehéz.

Bizonyítás

Tetszőleges $C \in \mathcal{C}$ problémára $C \leq A$ (mert A \mathcal{C} -nehéz), ami $A \leq B$ -vel és a visszavezetés tranzitivitásával együtt adja, hogy $C \leq B$, tehát B is \mathcal{C} -nehéz.

Tehát például az EKVIVALENCIA probléma egyszerre **RE**-nehéz és **coRE**-nehéz is (mert az **RE**-teljes MEGÁLLÁS és a **coRE**-teljes $\overline{\text{MEGÁLLÁS}}$ is visszavezethető volt rá). Ebből ki fog jönni, hogy nem lehet **benne** egyik osztályban sem, amihez előbb belátjuk, hogy:

Állítás

Ha \mathcal{C}' zárt a visszavezetésre, és $A \in \mathcal{C}'$ egy \mathcal{C} -nehéz probléma, akkor $\mathcal{C} \subseteq \mathcal{C}'$.

Bizonyítás

Mivel A \mathcal{C} -nehéz, ezért minden $B \in \mathcal{C}$ -re $B \leq A$. Mivel \mathcal{C}' zárt a visszavezetésre, $B \leq A$ -ból és $A \in \mathcal{C}'$ -ből kapjuk, hogy $B \in \mathcal{C}'$, vagyis minden \mathcal{C} -beli B probléma benne van \mathcal{C}' -ben is, tehát $\mathcal{C} \subseteq \mathcal{C}'$.

Ezt alkalmazni is tudjuk most rögtön:

Állítás

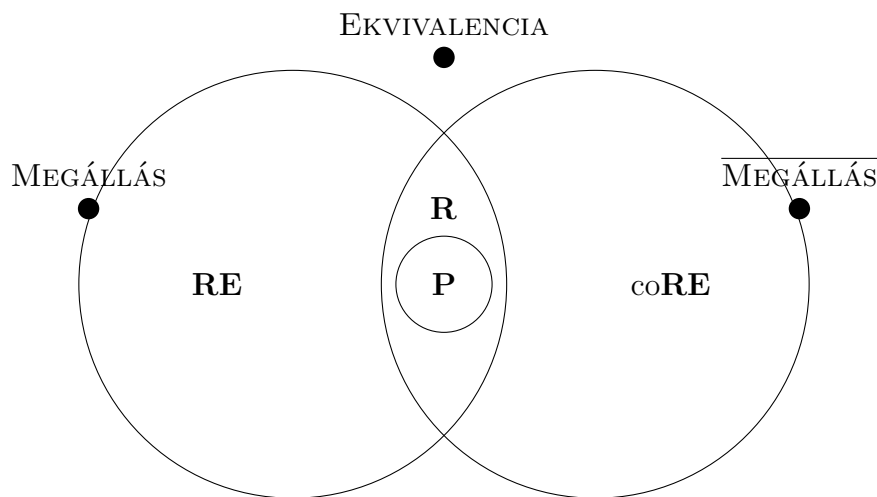
Az EKVIVALENCIA probléma se **RE**-ben, se **coRE**-ben nincs benne.

Bizonyítás

Ha $\text{EKVIVALENCIA} \in \mathbf{RE}$, akkor mivel **coRE**-nehéz és **RE** zárt a visszavezetésre, az előző állításból kapjuk, hogy **coRE** \subseteq **RE, amiről tudjuk, hogy nincs így.**

Ha pedig $\text{EKVIVALENCIA} \in \mathbf{coRE}$ lenne, akkor hasonló módon azt kapnánk, hogy **RE** \subseteq **coRE**, ami szintén tudjuk, hogy nincs így.

Tehát a korábbi ábránkba belerajzolhatjuk az EKVIVALENCIA problémát is, és rákerül a **P** osztály (ami nyilván része **R**-nek, hiszen ami polinomidőben eldönthető, az eldönthető):



Azt ugyan eddig még nem láttuk, hogy $\mathbf{P} \neq \mathbf{R}$, de így lesz.

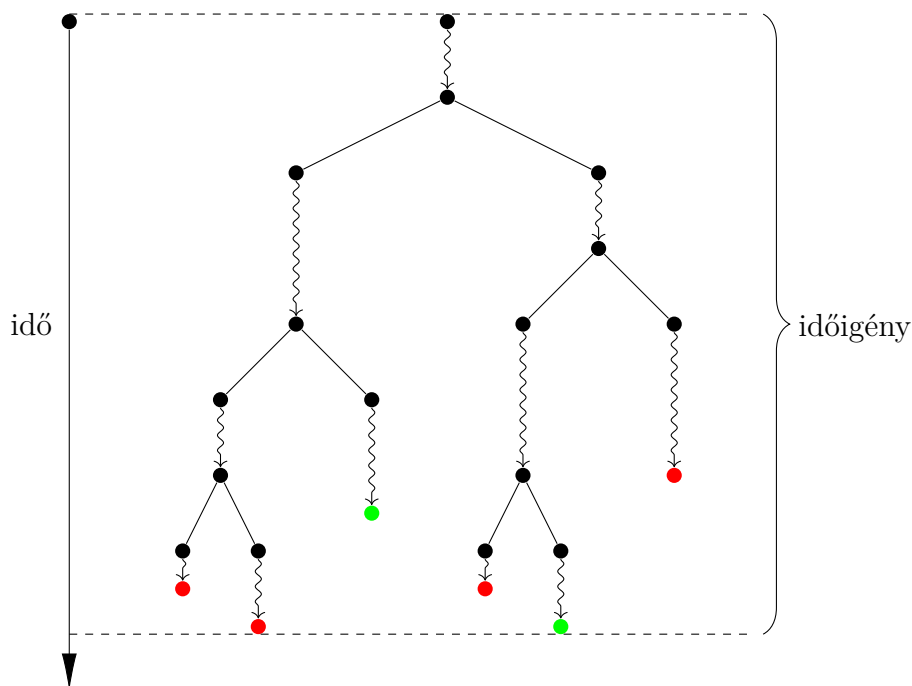
7. Nemdeterminizmus

Ebben a részben megismerkedünk egy **nemrealisztikus** számítási modellel²², a **nemdeterminizmussal**, bevezetjük az **NP** bonyolultsági osztályt és nézünk pár **NP**-teljes problémát.

Ha RAM-gépben (azaz pszeudokódban) gondolkodunk, akkor egy nemdeterminisztikus program szintaxisban annyiban több egy „szokványos”, azaz determinisztikus programnál, hogy lehet benne egy lépésben egy bitnek **nemdeterminisztikusan** értéket adni, mondjuk egy ilyen utasítással:

$$v := \text{nd}()$$

ami annyit tesz, hogy a következő lépésben a v változó értéke vagy 0 lesz, vagy 1, bármelyik lehetséges. Úgy is képzelhetjük, mintha ilyenkor a program „elágazna”, több szálon, az egyik a $v = 0$ értékkel folytatja a számítást, a másikon a $v = 1$ értékkel folytatja. Persze ilyen utasítást többször is végrehajthat a program a futása során. Ha egy ilyen programnak egy adott inputra vizualizáljuk a futását, egy, a lentihez hasonló **számítási fát** kapunk:



Az ilyen fán az idő fentről lefele telik és minden nemdeterminisztikus bit generálásánál „elágazik” a program lefele. Az időigény számításánál a **leghosszabb szál időigényét** nézzük, tehát egy adott inputon a nemdeterminisztikus program időigénye a számítási fa mélysége lesz. Továbbá, a program **időkorlátja** $f(n)$, ha tetszőleges n méretű inputon az időigénye legfeljebb $f(n)$ (azaz: tetszőleges n méretű inputon mindegyik számítási szál hossza legfeljebb $f(n)$).

Ha ezt úgy könnyebb elképzelni, akkor minden egyes elágazásnál tekinthetjük úgy, mintha a program szétesztaná a feladatot két külön számítógépre, mindkettőnek lemásolva az aktuális memóriát (az összes regiszter értékét), és onnan kezdve ezek a szálak „külön életet élnek”, tovább osztódhatnak stb. Mindezt konstans idő alatt. Az időigényt pedig úgy vesszük, mintha megvárnánk az **összes** szálát, hogy termináljon és csak ezután adnánk vissza egy végeredményt.

A végeredményről még nem beszéltünk: ha a nemdeterminisztikus algoritmusunk egy eldöntési algoritmus, azaz minden szálon **true**t vagy **false**t ad vissza, akkor a végeredmény akkor lesz **true**,

²²legalábbis jelen tudásunk szerint nem fogjuk tudni hatékonyan szimulálni

ha legalább egy szálon **true** a végeredmény (a képen a zöld levelek jelzik a **true**, pirosak a **false** outputot), és akkor **false**, ha minden szál **false**-al tér vissza.

Ha a nemdeterminizmust **valószínűségi** algoritmusként szeretnénk felfogni, akkor felfoghatjuk pl. úgy, hogy minden nemdeterminisztikus elágazásnál $\frac{1}{2} - \frac{1}{2}$ eséllyel generál (egymástól függetlenül) egy random bitet az algoritmus, és úgy fut tovább. Ekkor pedig nyilván az a modell, hogy ha nemnulla valószínűséggel fogadja el az inputot, akkor elfogadja azt, ha pedig nullával (mert minden szálon elutasítja), akkor elutasítja.

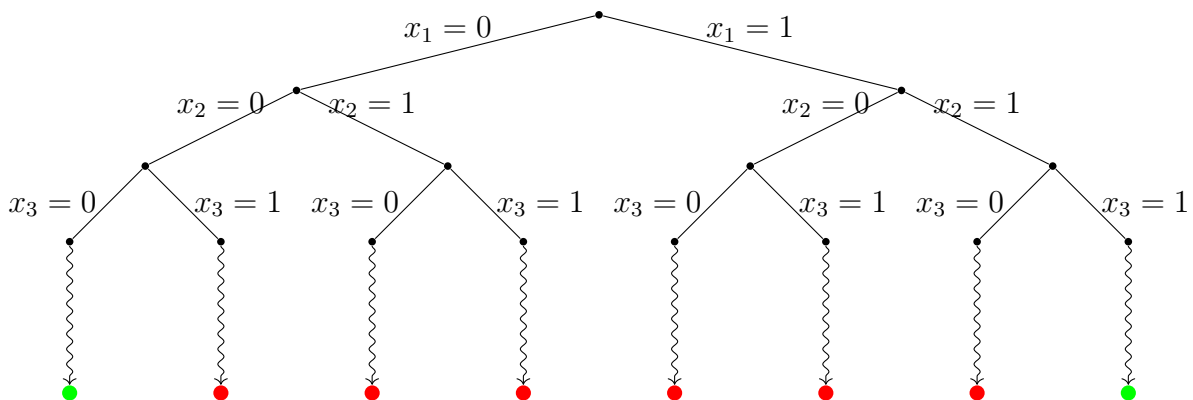
Lássunk pár példát.

Nemdeterminisztikus algoritmus a SATra

Ha az input formulánkban az x_1, \dots, x_n változók fordulnak elő:

1. generáljunk minden x_i -hez egy nemdeterminisztikus bitet, így kapunk egy értékadást
2. ha a generált értékadás kielégíti a formulát, adjunk vissza **true**-et, egyébként **false**-et.

Nézzük ezt egy példán: ha az input $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$, akkor az algoritmus futásához tartozó számítási fa



ahol a generálás utáni hullámos vonal a (determinisztikus) formula-kiértékelő algoritmus.

Ennek az algoritmusnak a nemdeterminisztikus időigénye $O(n)$, hiszen a generálási fázisban legfeljebb n változónak adunk értéket (n az egész formula hossza, felső korlát a benne levő változók számára), és egy rögzített értékadás behelyettesítése egy formulába implementálható lineáris időben, tehát a fa mélysége $O(n)$.

Nemdeterminisztikus algoritmus a HAMILTON-Útra

Ha az input a $G = (V, E)$ gráf, csúcshalmaza $\{1, \dots, N\}$:

1. Hozzunk létre egy N -elemű tömböt: $T[1], \dots, T[N]$.
2. Minden i -re generáljunk nemdeterminisztikusan $T[i]$ -be egy $1 + \lfloor \log N \rfloor$ -bites számot. (A szándék: $T[i]$ -be egy Hamilton-út i . csúcsát szeretnénk generálni, ennyi biten tudunk eltárolni egy 1 és N közti számot.)
3. Ellenőrizzük determinisztikusan, hogy minden $1 \leq i \leq N$ -re van-e olyan $T[j]$, amire

$T[j] == i$. (Ha igen, akkor tényleg sikerült az $\{1, \dots, N\}$ halmaznak egy permutációját, egy sorrendjét belegenerálni a tömbbe.) Ha nem, akkor ez a szál adjon vissza **false**-t.

4. Ellenőrizzük determinisztikusan, hogy minden $1 \leq i < N$ -re $T[i]$ és $T[i + 1]$ szomszédosak-e G -ben. (Ha igen, akkor sikerült egy sétát generálnunk a gráfban, ami, mivel ezen a ponton már tudjuk, hogy minden csúcsot érint, egy Hamilton-út kell legyen.) Ha igen, adjunk vissza **true**-t. Ha nem, **false**-t.

Ennek az algoritmusnak az időigénye ebben a formában: $O(N \log N)$ ideig tart a nemdeterminisztikus bitgenerálás (N darab, egyenként $\log N$ -bites számot generálunk bitenként), aztán mondjuk $O(N^2)$ időben ellenőrizzük, hogy minden szám 1 és N között ki lett-e generálva (ezt persze hatékonyabban is lehet, pl. ha lerendezzük a tömböt előbb, úgy csak $O(N \log N)$, de most csak az a fontos, hogy **polinom** az időigény), a harmadik pontbeli ellenőrzés pedig mondjuk $O(N)$ idő (ez persze attól is függ, hogy milyen gyorsan tudunk éleket lekérdezni a gráfból, ha konstans, mondjuk a szomszédsági mátrix reprezentációval, akkor $O(N)$, ha lineáris, mondjuk az éllistással, akkor $O(N^2)$ idő – de ez is mindenképp polinom lesz).

Definíció: 3-SZÍNEZÉS

Input: egy G gráf.

Output: kiszínezhetőek-e G csúcsai három színnel **helyesen**, azaz úgy, hogy szomszédos csúcsok különböző színt kapjanak?

Nemdeterminisztikus algoritmus a 3 – SZÍNEZÉSre

Ha az input a $G = (V, E)$ gráf, csúcshalmaza $\{1, \dots, N\}$:

1. Hozzunk létre egy N -elemű tömböt: $T[1], \dots, T[N]$.
2. Minden i -re generáljunk nemdeterminisztikusan $T[i]$ -be egy kétbites számot. Ha valahova az 11-et generáltuk, mondjuk egyből, hogy **false**.
3. Ellenőrizzük determinisztikusan, hogy minden (i, j) élre $T[i] \neq T[j]$ igaz-e. Ha mind-egyikre igaz, akkor **true**, egyébként **false**.

Időigény: lineáris.

A központi nemdeterminisztikus (idő)bonyolultsági osztályunk pedig:

Definíció: NP

NP jelöli a **nemdeterminisztikus** algoritmussal **polinomidőben** eldönthető problémák osztályát.

Tehát például a fentiek szerint SAT, HAMILTON-ÚT, 3 – SZÍNEZÉS mind **NP**-beliek.

Hol van **NP** az eddig megismert bonyolultsági osztályok rendszerében? Egyrészt,

$$\mathbf{P} \subseteq \mathbf{NP},$$

hiszen egy (determinisztikus) polinomidejű algoritmust felfoghatunk úgy is, mint egy olyan nemdeterminisztikus algoritmust, ami nem generál egyszer sem nemdeterminisztikusan bitet, tehát végig csak egy szálon fut; ilyenkor persze ez az egy szál fogja adni az időigényét, tehát nemdeterminisztikusan is polinomidejűnek fog számítani.

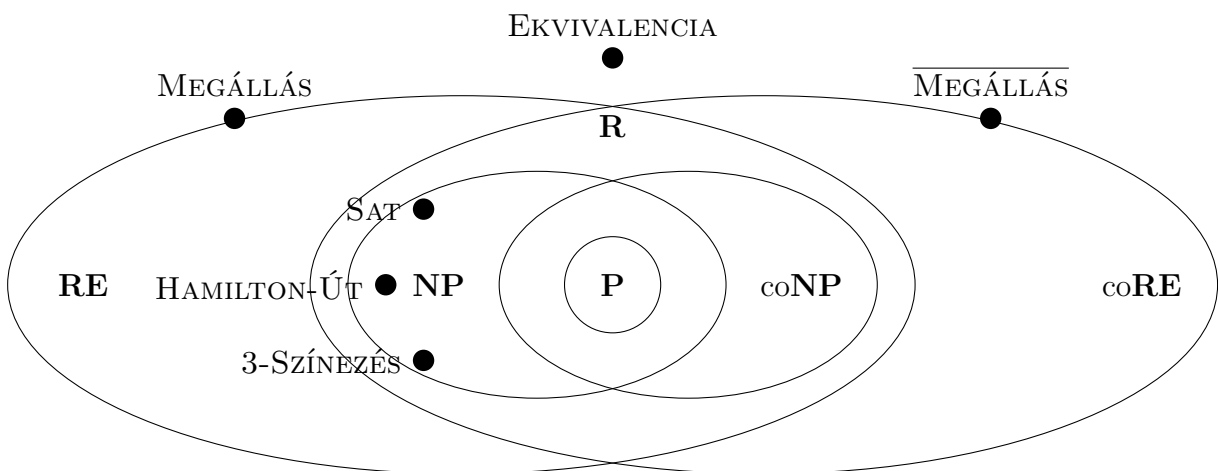
Másrészt $\text{coP} = \mathbf{P}$ miatt (hiszen egy determinisztikus algoritmusban a visszatérési értékeket kicserélve egy ugyanolyan időigényű, a komplementert eldöntő algoritmust kapunk) az is igaz, hogy $\mathbf{P} = \text{coP} \subseteq \text{coNP}$. Tehát,

$$\mathbf{P} \subseteq \mathbf{NP} \cap \text{coNP}$$

is igaz. Ez eddig hasonlít az \mathbf{R} és \mathbf{RE} osztályok kapcsolatára, de a hasonlóság itt véget is ér: **nem tudjuk** és (magánvélemény) én nem is hiszem, hogy \mathbf{P} egyenlő lenne ezzel a metszettel.

Az is igaz, hogy minden \mathbf{NP} -beli probléma eldönthető (determinisztikus) algoritmussal, tehát hogy $\mathbf{NP} \subseteq \mathbf{R}$ (és így $\text{coNP} \subseteq \text{coR} = \mathbf{R}$ is igaz). Hiszen egy számítási fát **bejárni** be tudunk determinisztikusan: ha a programunk időkorlátja mondjuk n^k , akkor futás közben legfeljebb n^k bitet fog generálni nemdeterminisztikusan. Ha előre legeneráljuk mind az n^k darab bitet, az kiválaszt a számítási szálak közül pontosan egyet. Nincs más dolgunk, mint determinisztikusan kipróbálni mind a 2^{n^k} (igen, ez nagyon sok) lehetőséget, és mindegyiket leszimulálni; ez összesen $n^k \cdot 2^{n^k}$ időben bejárja a számítási fát. Ha közben egyetlen szál is **true** értékkel tér vissza, akkor az egész szimuláció is, ha pedig nem, akkor **false** értékkel. Ez tehát egy determinisztikus, exponenciális időigényű algoritmus, ami ugyanazt a problémát dönti el, mint amire az \mathbf{NP} algoritmusunk volt.

Tehát a korábbi ábránkba belerajzolva pár új elemet:



Ezt az ábrát nemsokára pontosítani fogjuk.

A három \mathbf{NP} -beli probléma elhelyezéséről: persze csak azt tudjuk eddig, hogy \mathbf{NP} -beliek. Ettől még benne lehetnének $\mathbf{NP} \cap \text{coNP}$ -ben, sőt akár \mathbf{P} -ben is! Nem tudjuk, hogy benne vannak-e, a mai napig ezt sem bebizonyítani, sem megcáfolni nem sikerült még senkinek. Azt se tudjuk, hogy $\mathbf{NP} = \text{coNP}$ igaz-e vagy sem, és azt sem, hogy $\mathbf{P} = \mathbf{NP}$ igaz-e vagy sem.

A kurzuson fogunk még tanulni több bonyolultsági osztályt, melyekről sokan (én is) inkább azt tartják valószínűnek, hogy tényleg különböznek (pl. szerintem \mathbf{P} tényleg valódi része $\mathbf{NP} \cap \text{coNP}$ -nek, és $\mathbf{NP} \neq \text{coNP}$), az erre utaló jelekről amennyire lehet, próbálok szólni pár szót.

8. Polinomidőben verifikálhatóság

Az előző fejezet nemdeterminisztikus algoritmusai mind arra a sémára épültek, hogy először is generálunk valami „bizonyítékot”, ami bizonyíthatja, hogy a kérdésre a válasz „igen”, majd ezek után determinisztikusan ellenőrizzük, hogy tényleg sikerült-e egy „bizonyítékot” generálni.

Ez nem véletlen egybeesés, az \mathbf{NP} osztályba pontosan azok a problémák esnek, amik megoldhatók ilyen módon. Vagyis:

Definíció: Polinomidőben verifikálhatóság.

Azt mondjuk, hogy az A probléma **polinomidőben verifikálható**, ha van egy olyan R reláció **inputok** és **tanúk** közt, melyre

- ha $R(I, T)$ az I inputra és a T tanúsítványra, akkor $|T| \leq |I|^c$ valamilyen c konstansra (azaz a tanúk „nem túl hosszúak”);
- ha kapunk egy (I, T) párt, arról determinisztikus polinomidőben el tudjuk dönteni, hogy $R(I, T)$ fennáll-e vagy sem (azaz egy tanú könnyen ellenőrizhető);
- pontosan akkor létezik I -hez olyan T , melyre $R(I, T)$ igaz, ha I az A -nak egy „igen” példánya (azaz R tényleg egy jó „tanúsítvány-rendszer” az A problémához).

Például a SAT esetében egy **tanú** egy kielégítő értékadás volt, vagyis az I CNF és a T értékadás közt akkor állt fenn $R(I, T)$, ha T kielégíti I -t. Nyilván egy értékadás $O(n)$ hosszú, ha az I formula n hosszú, pontosan a kielégíthető formuláknak van kielégítő értékadásuk, és azt, hogy egy input I formulát egy input T értékadás kielégít-e, lineáris időben el tudjuk dönteni.

A HAMILTON-ÚT esetében egy I input gráfhoz egy tanú a csúcsok egy sorrendje volt, mely sorrendben bejárva a csúcsokat Hamilton-utat kapunk; ez is nyilván egy polinom (lineáris) méretű, polinom időben ellenőrizhető tanúsítvány-rendszer. A 3-SZÍNEZÉS problémánál pedig egy tanú egy helyes 3-színezés, ami szintén ilyen tulajdonságú.

Az összefüggés pedig:

Állítás

Egy A probléma pontosan akkor van \mathbf{NP} -ben, ha polinomidőben verifikálható.

Bizonyítás

Ha A egy polinomidőben verifikálható probléma, akkor egy nemdeterminisztikus algoritmus egy I input esetén: először generálunk egy $|I|^c$ hosszú T bitsorozatot (ahol c a polinomidőben verifikálhatóságbeli c), majd ellenőrizzük, hogy $R(I, T)$ fennáll-e. Mivel az (I, T) pár hossza $O(n + n^c)$, tehát polinom, ha ezen egy $O(n^k)$ időigényű algoritmust futtatunk $R(I, T)$ ellenőrzésére, a teljes időigény $O(n^{c \cdot k})$, polinom lesz és mivel pontosan az A -beli I -kre létezik ilyen T , ez a polinomidejű nemdeterminisztikus algoritmus eldönti

az A problémát.

Visszafelé, ha A -ra van egy nemdeterminisztikus polinomidejű algoritmusunk, akkor feltételezhetjük, hogy az algoritmusnak mindig pontosan két választási lehetősége van. Így egy n^k hosszú számítási szálát adott I inputon pontosan le tudunk írni egy n^k hosszú bináris stringgel, melynek az i . bitje megadja, hogy az i . lépésben a két választási lehetőség közül melyik irányba folytattuk a számítást. Ekkor egy ilyen tanúsítvány hossza $O(n^k)$ lesz; az R reláció pedig álljon fenn az I input és a T tanú közt pont akkor, ha a T által kijelölt szál az I -hez tartozó számítási fában egy elfogadó számítási szál. Ezt (mivel csak determinisztikusan szimulálni kell a vonatkozó egyetlen szálát) polinomidőben el tudjuk dönteni és persze pontosan azokhoz az inputokhoz fog létezni elfogadó szál, melyeket a nemdeterminisztikus algoritmus elfogad, tehát A „igen” példányaihoz.

Ezért a továbbiakban az „ A probléma **NP**-ben van” típusú állításokat könnyebben be fogjuk tudni látni egy ilyen polinomidőben ellenőrizhető tanúsítvány-rendszer megadásával.

9. Cook tétele

Láttuk, hogy pontosan ugyanazokat a problémákat lehet RAM-gépen polinom időben megoldani, mint amiket egyszalagos Turing-gépen. Cook tételét:

Állítás: Cook tétele.

A SAT **NP**-teljes.

Turing gépen fogjuk bebizonyítani.

Bizonyítás

Legyen M egy $p(n)$ polinom időkorlátos egyszalagos nemdeterminisztikus Turing-gép és $u = a_1 \dots a_n$ egy n hosszú inputja. Le akarunk gyártani egy olyan φ formulát, mely pontosan akkor kielégíthető, ha M elfogadja u -t. Az egyszerűség kedvéért feltesszük, hogy M -nek minden lépésben pontosan két választása van.

Egyrészt, $p(n)$ lépésben az M gép $p(n)$ darab nemdeterminisztikus döntést hoz, ezeket az $x_1, \dots, x_{p(n)}$ logikai változókkal írjuk le: $x_i = 0$ jelenti azt, hogy a gép az i . lépésben az „első” lehetőséget, $x_i = 1$ pedig hogy a „második” lehetőséget választja. Tetszőleges rögzített $\mathbf{x} = x_1, \dots, x_{p(n)}$ sorozat meghatároz egy számítási szálát, ezt akarjuk szimulálni.

Nyilván t lépésben M legfeljebb t messzire tud jutni a szalagján, tehát csak az első $p(n)$ cellát fogja használni. Bevezetünk minden $0 \leq i, j \leq p(n)$ -re, $a \in \Gamma$ -ra (emlékszünk, Γ volt a szalagábécé, ennek része a Σ input ábécé), és $q \in Q$ -ra (ez pedig a Turing-gép egy állapota) néhány logikai változót: $p_{i,j,a}$ -t akkor szeretnénk 1-re állítani, ha a gép \mathbf{x} -nek megfelelő szálán az i . lépésben a j . cellán az a jel áll, egyébként 0-ra. Továbbá, $h_{i,j}$ -t akkor szeretnénk 1-re állítani, ha a gép olvasófeje az i . lépésben a j . cellán áll. Végül, $q_{i,q}$ -t akkor szeretnénk 1-re állítani, ha a gép az i . lépésben a q állapotban van.

A kezdősort kitölteni könnyű: minden $j = 1, \dots, n$ -re $p_{0,j,a_j} = 1$ (mert a j . cellában kezdetben a_j van), $p_{0,0,\triangleright} = 1$ (mert a szalag elején a \triangleright jel áll) és minden $n < j \leq p(n)$ -re $p_{0,j,\sqcup} = 1$ (a szalag input utáni része \sqcup -zel van töltve). Az állapotot és a fejet is: $q_{0,q_0} = 1$, az M kezdőállapotban van kezdetben és $h_{0,0} = 1$, a fej az első cellán van. Ezeket a változókat ésszelve, és a nem említett nulladik sorhoz tartozó változókat (amiket 0-ra kell állítanunk)

negálva hozzáésselve eddig kapunk egy formulát, ami leírja az első sor tartalmát.

Ezek után leírjuk formulával, hogy az $i + 1$. lépésben a j . cella tartalmát honnan kapjuk: ennek a cellának a tartalma csak a fölötte levő cella tartalmától függ, a fej pozíciójától, az i . lépésbeli állapottól és az x_i változó értékétől. Konkrétan, ha a fej pozíciója nem j az i . lépésben, akkor a cella tartalma ugyanaz, mint volt:

$$\neg h(i, j) \rightarrow (p_{i,j,a} \leftrightarrow p_{i+1,j,a})$$

minden a -ra, i -re és j -re (ez összesen polinom sok ilyen formula, pontosabban ennek a CNF-jének a konjunkciója), ha pedig ott a fej, akkor követjük az átmenetet, amit választottunk: ha $h(i, j)$ igaz, akkor $p(i, j, a)$ akkor igaz, ha a $\delta(q, b)$ (ez egy $\{(q_1, b_1, d_1), (q_2, b_2, d_2)\}$ pár, ha $x_i = 0$, akkor az első, ha $x_i = 1$, akkor a második átmenetet követjük, tehát pl. ekkor

$$h_{i,j} \wedge q_{i,q} \wedge x_i \rightarrow (p_{i+1,j,b_1} \wedge q_{i+1,q_1} \wedge h_{i+1,j+d_1}),$$

ahol most a $d_1 \in \{-1, 0, 1\}$ az egyszerűség kedvéért. Tehát „ha itt a fej és q -ban volt a gép és az első alternatívát választja, akkor a fej erre/arra mozog, az állapot q_1 lesz, a cellában levő betű pedig b_1 ”, és hasonló felírunk az $\wedge \neg x_i$ végződésre is. Plusz, még felírjuk a cellára vonatkozólag egy ilyen implikáció jobb oldalára, hogy a minden más $c \neq b_1$ betűre $\neg p_{i+1,j,c}$ (azaz a többi betű nincs ebben a cellában), minden más j' pozícióra $\neg h_{i+1,j'}$ (máshol nincs a fej) és minden más q' állapotra $\neg q_{i+1,q'}$ (másik állapotban nincs a gép).

Ez egy igen hosszú konjunkció, de csak polinom méretű és könnyen (egymásba ágyazott ciklusokkal) kiírható; továbbá, leírja az egész számítási folyamat tartalmát, ha az x_i -ket mindet beállítjuk, akkor minden változó tényleg a számításnak megfelelő módon áll be.

Egy apró részlet: ha a szál hamarabb végezne, akkor úgy vesszük, mintha onnan kezdve ugyanabban az ACCEPT/REJECT állapotban marad a $p(n)$. lépésig.

Ez a szál akkor fogadja el az inputot, ha az utolsó lépésben az állapot ACCEPT, ami egy nagy diszjunkció: $\bigvee_{j=1}^{p(n)} q_{p(n), \text{ACCEPT}}$, ezt még hozzáésselve az eddig legyártott formulánkhöz kapjuk, hogy ez a formula pontosan akkor kielégíthető, ha a $M(u) = \text{ACCEPT}$.

10. A SAT variánsai

Ha nem csak CNF-et, de tetszőleges logikai formulát megengedünk inputként, és azt kérdezzük, kielégíthető-e, kapjuk a FORMSAT problémát:

Definíció: FORMSAT

Input: Egy φ (ítéletkalkulus-beli) formula.

Output: Kielégíthető-e φ ?

Állítás

A FORMSAT is NP-teljes probléma.

Bizonyítás

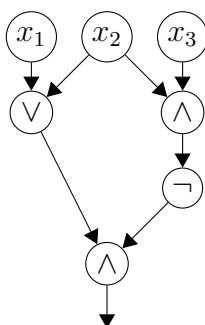
Mivel a FORMSAT probléma **általánosabb**, mint a SAT, legalább olyan nehéz is: formálisan, a $\varphi \mapsto \varphi$ (identikus) leképezés egy jó visszavezetés SAT-ról FORMSAT-ra, hiszen

CNF-ből formulát készít (mivel minden CNF egyben formula is) polinomidőben (lineáris idő elég az átmásoláshoz) és tartja a választ (hiszen ugyanarról a formuláról kérdezzük ugyanazt).

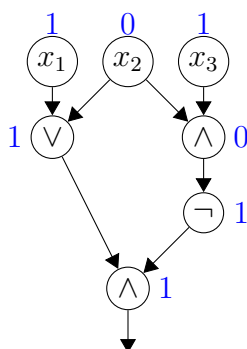
A probléma **NP**-beliségéhez elég azt látnunk, hogy egy φ formulához ugyanúgy jó tanúsítvány annak egy kielégítő értékadása, ezt továbbra is lineáris időben ellenőrizni tudjuk.

Egy **logikai hálózat** a (mondjuk) architektúrákból vagy dimatból megszokott: véges, irányított, körmentes gráf, csúcsait **kapuknak** is nevezzük, minden kapunak van egy címkéje, ami lehet \neg (ekkor a kapu befoka 1 kell legyen), lehet \vee vagy \wedge (ekkor a befoka 2), vagy lehet konstans 0, konstans 1, vagy az x_1, x_2, \dots változók valamelyike. Továbbá egy kapu ki van jelölve benne **output** kapunak.

Például:



Egy hálózatban ha az x_i változóknak adunk egy-egy értéket, akkor a hálózat összes kapujának a logikai kapuk címkéjének megfelelően a gráf topologikus rendezése szerinti sorrendben lesz egy értéke, pl. a fenti hálózatban ha $x_1 = 1$, $x_2 = 0$ és $x_3 = 1$, akkor ez az értékadás a többi kapunak az alábbiak szerint ad értéket:



A hálózat **értéke** egy adott értékadás mellett az output kapu értéke lesz, tehát a fenti példában pl. a hálózat értéke 1; a hálózat pedig *kielégíthető*, ha van olyan értékadás, mely mellett a hálózat értéke 1.

A fenti példa hálózat tehát kielégíthető, mert az $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ értékadás mellett 1 az értéke.

Ezzel kapcsolatban két problémát tudunk mondani:

Definíció: HÁLÓZAT-KIELÉGÍTHETŐSÉG

Adott egy hálózat. Kielégíthető-e?

Definíció: HÁLÓZAT-KIÉRTÉKELES

Adott egy **változómentes** hálózat. Igaz-e az értéke?

Mivel egy input változómentes hálózatot egy egyszerű rekurzív címkézéssel ki tudunk értékelni (minden kapuhoz megjegyezve az értékét, hogy csak egyszer számítsuk ki), így

Bizonyítás

A HÁLÓZAT-KIÉRTÉKELES probléma **P**-ben van.

A HÁLÓZAT-KIELÉGÍTHETŐSÉG probléma pedig emiatt **NP**-beli, hiszen egy megfelelő tanúsítvány-rendszer a változók egy értékadása, mely kielégíti a hálózatot: ez nyilván polinom méretű, pontosan a kielégíthető hálózatokhoz létezik ilyen tanú és mivel a kiértékelés **P**-ben van, így azt hatékonyan tudjuk ellenőrizni, hogy egy állítólagos tanú valóban tanúsítvány-e az adott inputhoz vagy sem.

Ennél több is igaz:

Állítás

A HÁLÓZAT-KIELÉGÍTHETŐSÉG probléma **NP**-teljes.

Bizonyítás

Vegyünk mondjuk a FORMSAT probléma egy φ példányát, egy formulát. Rekurzívan minden egyes F részformulájához rendeljünk egy g_F kaput a következőképp:

- ha $F = x_i$, akkor legyen g_F egy x_i címkéjű kapu;
- ha $F = G \vee H$, akkor g_F legyen egy \vee címkéjű kapu a g_G és g_H ősökkel;
- ha $F = G \wedge H$, akkor g_F legyen egy \wedge címkéjű kapu a g_G és g_H ősökkel;
- ha $F = \neg G$, akkor g_F legyen egy \neg címkéjű kapu a g_G őssel;
- ha $F = G \rightarrow H$, akkor hozzunk létre egy $g \neg$ kaput, g_G őssel, és g_F legyen egy \vee kapu, g és g_H ősökkel;
- végül, ha $F = G \leftrightarrow H$, akkor hozzunk létre egy g_1 és egy g_2 kaput, mindkettőt \neg címkével, g_1 őse g_G , g_2 őse g_H , továbbá egy h_1 és egy h_2 kaput, mindkettőt \vee címkével, h_1 őse g_1 és g_H (ennek értéke $G \rightarrow H$ értéke lesz), h_2 őse g_F és g_2 (ennek pedig $H \rightarrow G$), végül legyen g_F egy \wedge címkéjű kapu a h_1 és h_2 ősökkel.

Vegyünk észre, hogy a generált hálózat lineáris méretű mindenképp (nem azt kapnánk, ha a \leftrightarrow részformulákat előbb CNF-re hoznánk), és persze ekvivalens az eredeti formulával.

Mivel az **NP**-nehéz FORMSAT problémát vissza tudtuk vezetni az **NP**-beli HÁLÓZAT-KIELÉGÍTHETŐSÉGRE, így ez is **NP**-teljes.

Megjegyzés: „Cook tétele” sokszor a fenti formában fut.

Megszoríthatjuk a SAT problémát úgy, hogy csak speciális alakú CNF-eket engedünk meg inputként. Ilyen megszorítás lehet például az is, ha klónként korlátozzuk a literálok lehetséges számát:

Definíció: k SAT

Tetszőleges k konstansra a k SAT a következő probléma:

Input: egy olyan CNF, melyben minden klóz pontosan k darab literálból áll.

Output: kielégíthető-e a formula?

Felmerülhet a kérdés, hogy ez a megszorítás vajon könnyít-e a kielégíthetőség vizsgálatán. A következő állítás azt mondja, hogy nem sokat:

Állítás

A 3SAT is NP-teljes.

Bizonyítás

Visszavezetjük az NP-teljes HÁLÓZAT-KIELÉGÍTHETŐSÉG problémát a 3SAT problémára.

Legyen G egy hálózat. G minden g kapujához létrehozunk egy g logikai változót és minden g kapuhoz felírunk egy C_g CNF-et a következőképpen:

- Ha g változó kapu, akkor nem írunk fel belőle klózat.
- Ha g egy \wedge címkéjű kapu, a g_1 és g_2 ősekkel, akkor felírjuk hozzá a $g \leftrightarrow (g_1 \wedge g_2)$ formulával ekvivalens

$$(\neg g \vee g_1 \vee g_1) \wedge (\neg g \vee g_2 \vee g_2) \wedge (\neg g_1 \vee \neg g_2 \vee g)$$

CNF-et. (A klózokban az ismétlés szerepe csak annyi, hogy minden generált klóz pontosan három literált tartsalmazzon.)

- Ha g egy \vee címkéjű kapu, a g_1 és g_2 ősekkel, akkor felírjuk hozzá a $g \leftrightarrow (g_1 \vee g_2)$ formulával ekvivalens

$$(\neg g \vee g_1 \vee g_2) \wedge (\neg g_1 \vee g \vee g) \wedge (\neg g_2 \vee g \vee g)$$

CNF-et.

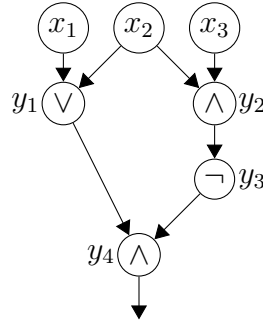
- Ha g egy \neg címkéjű kapu, a g_1 őssel, akkor felírjuk hozzá a $g \leftrightarrow \neg g_1$ formulával ekvivalens

$$(\neg g \vee \neg g_1 \vee \neg g_1) \wedge (g_1 \vee g \vee g)$$

CNF-et.

Az egész hálózathoz pedig azt a CNF-et készítjük, amit a kapukból egyenként generált CNF-ek éselésével kapunk, valamint még hozzávéve a $(g \vee g \vee g)$ klózt is, ahol g az output kapu.

Példa: a



hálózatból generált formula (a belső kapuk mellé odaírtuk a hozzájuk generált változócímkeket is):

$$\begin{aligned}
 & (\neg y_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee y_1 \vee y_1) \wedge (\neg x_2 \vee y_1 \vee y_1) \\
 \wedge & (\neg y_2 \vee x_2 \vee x_2) \wedge (\neg y_2 \vee x_3 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee y_2) \\
 \wedge & (\neg y_3 \vee \neg y_2 \vee \neg y_2) \wedge (y_3 \vee y_2 \vee y_2) \\
 \wedge & (\neg y_4 \vee y_1 \vee y_1) \wedge (\neg y_4 \vee y_3 \vee y_3) \wedge (\neg y_1 \vee \neg y_3 \vee y_4) \\
 \wedge & (y_4 \vee y_4 \vee y_4).
 \end{aligned}$$

Könnyen látható, hogy beállítva az x_1, \dots, x_n változói értékét valamire, a kapukhoz generált CNF-ek mindegyike egyszerre pontosan akkor lesz igaz, ha a belső kapukhoz generált logikai változók éppen a kapu értékét veszik fel ezen a bemeneten; továbbá az utolsó klóz (és így a formula) pedig pontosan akkor lesz igaz, ha az output kapu értéke 1. Tehát ez valóban egy visszavezetés a két probléma közt, és 3SAT **NP**-nehéz; mivel persze az **NP**-beli SAT egy általánosabb probléma, így 3SAT \in **NP** is, tehát ez egy **NP**-teljes probléma.

Persze emiatt tetszőleges $k \geq 3$ -ra már a k SAT is **NP**-teljes lesz, hasonló módon: 3 helyett tetszőleges 3-nál hosszabb fix méretű klózoikat tudunk gyártani literálok ismétlésével.

Ha viszont „elégé” leszorítjuk a klózo méretét, a probléma tényleg könnyebb lesz:

Állítás

A 2SAT probléma **P**-ben van.

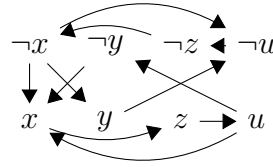
Ennek az állításnak a bizonyításához egy hatékony algoritmust kell gyártsunk, mely eldönti a 2SAT problémát. Ha φ egy $2CNF$, melyben az x_1, \dots, x_n változók fordulnak elő, ebből készítsük el a következő G_φ (irányított) gráfot:

- A G_φ gráf csúcsai az $x_1, \dots, x_n, \neg x_1, \dots, \neg x_n$ literálok.
- Az $\ell_1 \rightarrow \ell_2$ él pontosan akkor van G_φ -ben, ha φ -ben van ezzel mint implikációval ekvivalens klóz (tehát ha $(\overline{\ell_1} \vee \ell_2)$, vagy $(\ell_2 \vee \overline{\ell_1})$ szerepel φ -ben).

Egy példa a konstrukcióra: ha a formula

$$\varphi = (x \vee y) \wedge (\neg x \vee z) \wedge (\neg z \vee u) \wedge (\neg y \vee \neg u) \wedge (x \vee \neg u) \wedge (x \vee x),$$

akkor G_φ a következő gráf:



Állítás

A φ formula pontosan akkor kielégíthetetlen, ha van olyan ℓ literál, hogy ℓ és $\bar{\ell}$ elérhetők egymásból a G_φ gráfban.

Bizonyítás

Tegyük fel, hogy az x csúcsból a G_φ gráfban elérhető az y csúcs. Belátjuk az ehhez kellő lépésszám szerinti indukcióval, hogy ekkor $\varphi \models (x \rightarrow y)$ (tehát hogy φ -ből következik az $x \rightarrow y$ implikáció).

Nyilván mivel $x \rightarrow x$ tautológia, így $\varphi \models x \rightarrow x$ mindig igaz, tehát ha 0 lépésben érhető el x -ből y , akkor készen vagyunk. Legyen x -ből y elérhető $n + 1$ lépésben. Akkor van olyan z csúcs, melyre x -ből elérhető z n lépésben, és melyre (z, y) egy él G_φ -ben.

Az indukciós feltevés szerint ekkor $\varphi \models (x \rightarrow z)$. Másrészt, mivel a (z, y) élt azért vettük fel G_φ -be, mert $(z \rightarrow y)$ mint implikáció szerepel φ -ben, így $\varphi \models (z \rightarrow y)$ is igaz. Tehát $\varphi \models (x \rightarrow z) \wedge (z \rightarrow y)$, így a tranzitivitás miatt $\varphi \models (x \rightarrow y)$ is teljesül.

Ha tehát x -ből $\neg x$ is és $\neg x$ -ből x is elérhető, akkor φ -nek logikai következménye $(x \leftrightarrow \neg x)$, ami egy kielégíthetetlen formula. Tehát ekkor φ is kielégíthetetlen.

Még azt is meg kell mutassuk, hogy ha nincs ilyen literál, akkor a formula kielégíthető. Ezt a klózek száma szerinti indukcióval tesszük.

Ha G_φ -ben nincs klóz, akkor az üres CNF definíció szerint tautológia, tehát kielégíthető.

Legyen most G_φ -ben $n + 1$ klóz és nézzük egy x változóját. Ekkor vagy x -ből nem érhető el $\neg x$, vagy fordítva (különben x olyan literál lenne, melyből a negáltjába és vissza is vezetne út). Legyen ℓ a kettő közül az a csúcs, melyből nem érhető el $\bar{\ell}$. Állítsuk ℓ -t és a belőle G_φ -ben elérhető literálokat 1-re, ellentettjüket pedig 0-ra.

Ez a (részleges) értékadás nem próbálja meg ugyanazt az x változónak egyszerre 0-ra és 1-re is állítani, ugyanis ezt akkor tenné, ha x elérhető lenne ℓ -ből (ezért $x = 1$) és $\neg x$ is elérhető lenne ℓ -ből (ezért $\neg x = 1$, tehát $x = 0$). Csakhogy ez nem lehetséges: vegyük észre, hogy ha egy A csúcsból vezet él egy B csúcsba, akkor \bar{B} -ből is \bar{A} -ba (hiszen mindkét él ugyanazzal az implikációval ekvivalens, $A \rightarrow B \equiv \bar{B} \rightarrow \bar{A}$, így vagy mindkettőt felvesszük, amikor G_φ -t készítjük, vagy egyiket sem). Emiatt ha A -ból vezet **út** B -be, akkor \bar{B} -ből is \bar{A} -ba. Tehát ha $\neg x$ elérhető ℓ -ből, akkor $\bar{\ell}$ is elérhető x -ből. Viszont ekkor ℓ -ből elérhető x , onnan pedig $\bar{\ell}$, pedig abból indultunk ki, hogy ℓ -ből nem érhető el $\bar{\ell}$, ami ellentmondás.

Tehát ez a részleges értékadás nem próbál meg több értéket rendelni egy változóhoz. Így megtehetjük, hogy az érintett változók értékét rögzítjük. Ekkor az összes olyan klóz, melyben szerepel olyan ℓ literál, melynek most értéket adtunk, kielégül: ha a literál értékét

1-re állítottuk, akkor azért, ha pedig 0-ra, akkor ellentettjét, $\bar{\ell}$ -t állítottuk 1-re. Ekkor pedig ha a klóz $\ell \vee \ell'$, úgy felvettük hozzá G_φ -be az $\bar{\ell} \rightarrow \ell'$ élt, így mivel $\bar{\ell}$ -t 1-re állítottuk, ℓ' -t is 1-re állítottuk, és ez kielégíti az $\ell \vee \ell'$ klózt.

Tehát ezzel a parciális értékadással a φ formulánkban kielégítettünk legalább egy klózt (mert x szerepel egy klózban, tehát legalább az a klóz kiesik), eldobva a kielégített klózat pedig egy olyan φ' formulát kapunk, melyben csupa olyan változó szerepel, akiknek még nem adtunk értéket, az ehhez gyártott $G_{\varphi'}$ gráf pedig részgráfja lesz a G_φ gráfnak, tehát ebben sem lesz olyan x változó, akiből a negáltja és vissza elérhető. Az indukciós feltevés szerint ez a φ' kielégíthető, az előző menetben generált változó-értékadással kiegészítve egy az egész φ -t kielégítő értékadást kapunk.

Mivel pedig ezt a „kölcsonösen elérhető” tulajdonságot meg tudjuk hatékonyan oldani (pl. kiszámítjuk lineáris időben G_φ erősen összefüggő komponenseit, majd megnézzük, hogy van-e olyan x változó, hogy x és $\neg x$ ugyanabba a komponensbe kerültek-e), és G_φ elkészíthető φ -ből lineáris időben, így az egész algoritmus lineáris időben eldönti φ kielégíthetőségét.

A példaformulánkban szereplő G_φ -ben pl. ha az x változóból indulunk (x -ből nem érhető el $\neg x$), a belőle elérhető literálok z , u és $\neg y$, ezeket 1-re (vagyis $x = z = u = 1$, $y = 0$) állítva, ellentettjüket 0-ra épp a formula egy kielégítő értékadását kapjuk.

Egy másik lehetséges megszorítása a SAT problémának, ha nem a klózek hosszát, hanem a „formájukat” kötjük meg, egy lehetőség az, hogy ún. Horn-átnevezhető inputot várunk:

Definíció: Horn-formulák, Horn-átnevezhető formulák

Egy konjunktív normálformájú formula

- Horn-formula, ha benne minden klózban legfeljebb egy pozitív literál szerepel;
- Horn-átnevezhető formula, ha bizonyos változói komplementálásával Horn-formulává alakítható.

Például $x \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg y \vee \neg z)$ egy Horn-formula; az $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$ formula Horn-átnevezhető, mert pl. ha az y -okat és a z -ket a komplementerükre cseréljük, akkor a kapott $(x \vee \neg y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee \neg z)$ formula Horn-formula; az $(x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$ formula pedig még csak nem is Horn-átnevezhető.

Állítás

Horn-átnevezhető formulák kielégíthetősége polinom időben eldönthető.

Bizonyítás

Az ún. **egységrezolúció** alkalmazása ezeken a formulákon egy helyes és teljes algoritmus. Az algoritmus a következő:

- Ha van üres klóz a formulában, akkor kielégíthetetlen.
- Ha nincs üres klóz és nincs egységklóz sem, akkor kielégíthető.
- Ha van egy egységklóz (egyetlen literált tartalmazó klóz) a formulában, akkor
 - válasszunk egy $\{\ell\}$ egységklózt

- dobjuk el az összes klózt, mely tartalmazza ℓ -t
- a megmaradó klózokból hagyjuk el $\bar{\ell}$ -t
- menjünk vissza az első lépésre a maradék formulán.

Az algoritmus nyilván polinomidejű, hiszen minden iterációban egy változó eltűnik a formulából.

Amennyiben egy CNF-ben szerepel egy $\{\ell\}$ egységklóz, úgy minden kielégítő értékadásban $\ell = 1$ kell legyen; ekkor tehát ezt a literált beállítjuk 1-re. Ebben az esetben az ℓ -t tartalmazó klózok kielégülnek, a maradékból pedig az $\bar{\ell}$ alakú klózok pontosan akkor elégülnek ki, ha az $\bar{\ell}$ elhagyásával (ennek az értéke 0 lesz) kapott kisebb klóz kielégül. Tehát ha van egységklóz, akkor az algoritmus egy iterációban a φ formulából mindenképp egy olyan φ' formulát készít, mely pontosan akkor lesz kielégíthető, ha φ is az volt. Továbbá vegyük észre, hogy a kapott φ' szintén Horn-átnevezhető: ugyanaz a változóátnevezés, mely φ -t Horn alakra hozza, Horn alakra hozza φ' -t is.

Ha üres klóz is szerepel a formulában, akkor (mivel annak értéke mindenképp 0) a formula kielégíthetetlen, így ez a lépés is helyes választ ad.

Ha pedig sem egységklóz, sem üres klóz nem szerepel a formulában, akkor kielégíthető: ekkor minden klózban legalább két literál szerepel. Egy változó-komplementálás nem változtat a kielégíthetőségen: ha egy T értékadás kielégít egy formulát és abban a változók egy X részhalmazát komplementáljuk, akkor az a T' értékadás, mely T -től pontosan az X -beli változókon különbözik, kielégíti az átalakított formulát. Ha pedig vesszük azt a változó-komplementálást, mely után Horn-formulát kapunk az eredeti φ formulából, abban minden klózban szerepel negatív literál (mert minden klóz legalább kételemű), így a konstans 0 értékadás kielégíti.

„Keverni” viszont nem tudjuk ezt a két hatékonyan eldönthető esetet:

Állítás

A következő probléma NP-teljes:

Input: egy CNF, melynek minden klózában vagy legfeljebb két literál szerepel, vagy csupa negatív literálból álló háromelemű klóz.

Output: kielégíthető-e?

Bizonyítás

Visszavezetjük erre a problémára a 3SATot. Legyen φ egy CNF, benne a változók x_1, \dots, x_n .

Bevezetjük minden x_i változóhoz annak a kalapos változatát, \hat{x}_i -t is, és kikényszerítjük, hogy minden értékadásban x_i és \hat{x}_i ellentétes értéket vegyenek fel. Ezt az $x_i \leftrightarrow \neg \hat{x}_i$ formula CNF-jével: $(x_i \vee \hat{x}_i) \wedge (\neg x_i \vee \neg \hat{x}_i)$ érjük el. A készített φ' formulába tehát minden i -re felvesszük ezt a két bináris klózt, ezek megfelelnek az új probléma input szintaxisának.

Továbbá, az eredeti φ formulában minden pozitívan előforduló x_i változót $\neg \hat{x}_i$ -re cserélünk, így ezekben a klózokban pedig csupa negatív literál szerepel, ami szintén megfelel az új probléma szintaxisának.

Például, a $(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$ formulából a

$$(x \vee \hat{x}) \wedge (\neg x \vee \neg \hat{x}) \wedge (y \vee \hat{y}) \wedge (\neg y \vee \neg \hat{y}) \wedge (z \vee \hat{z}) \wedge (\neg z \vee \neg \hat{z}) \\ \wedge (\neg \hat{x} \vee \neg y \vee \neg \hat{z}) \wedge (\neg x \vee \neg \hat{y} \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg \hat{z})$$

formula lesz.

Világos, hogy ha az eredeti formulát egy T értékadás kielégíti, akkor az elkészített formulát pedig az az értékadás, melyben az eredeti x_i változók értékei a T -beli értékek, a \hat{x}_i változók értékei pedig a $T[x_i]$ értékek negáltjai, kielégíti; továbbá, az is világos, hogy ha a generált φ' formulát kielégíti egy T értékadás, akkor a bináris klózok miatt abban az x_i -k és \hat{x}_i -k mindenképp ellentétes értéket vesznek fel, így a negatív klózok konstrukciója miatt ugyanez az értékadás kielégíti az eredeti φ formulát is.

11. NP-teljes gráfelméleti problémák

Ebben a fejezetben néhány gráfos problémáról mutatjuk meg, hogy **NP**-teljesek.

Az első a FÜGGETLEN CSÚCSHALMAZ lesz:

Definíció: FÜGGETLEN CSÚCSHALMAZ

Input: egy G (irányítatlan) gráf és egy K szám.

Output: van-e G -ben K darab **független** (azaz páronként nem szomszédos) csúcs?

Állítás

A FÜGGETLEN CSÚCSHALMAZ egy **NP**-teljes probléma.

Bizonyítás

Visszavezetjük rá a 3SAT problémát. Vegyük a $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_n$ formulát, minden klózában három literállal.

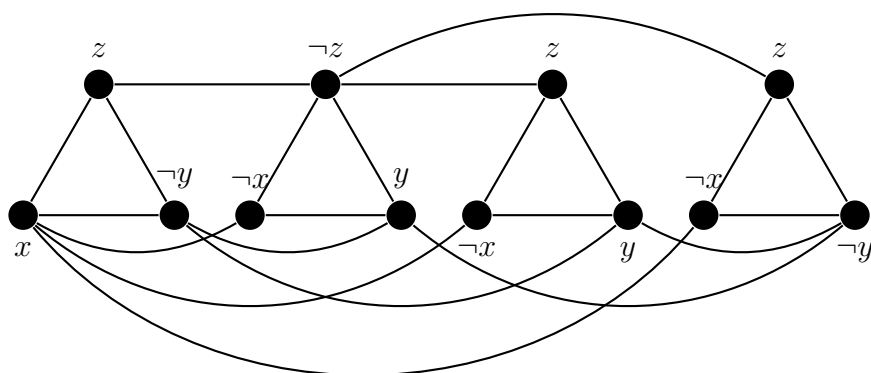
A következő G gráfot hozzuk létre:

- G csúcsai a φ -beli literál**előfordulások** (tehát pontosan $3n$ darab csúcs lesz).
- Az azonos klózban szereplő literál-előfordulásokat összekötjük éllel. (Eddig tehát van n darab háromszögünk.)
- Továbbá, az **ellentétes** literálokat szintén összekötjük. (Tehát ha egy klózban van x , egy másikban $\neg x$, a hozzájuk tartozó literálokat összekötjük.)
- A K célszámunk legyen n , a klózok száma.

Például ha a formula

$$\varphi = (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z),$$

akkor a gráf:



A célszám pedig $K = 4$.

Azt állítjuk, hogy ez egy visszavezetés 3SAT-ról a FÜGGETLEN CSÚCSHALMAZ-ra.

Vegyük észre, hogy mivel egy háromszögön belül nincs két független csúcs, és összesen K háromszöget hozunk létre, így egy K -elemű független csúcshalmazban mindenképp háromszögenként pontosan egy csúcsot kell kiválasztanunk.

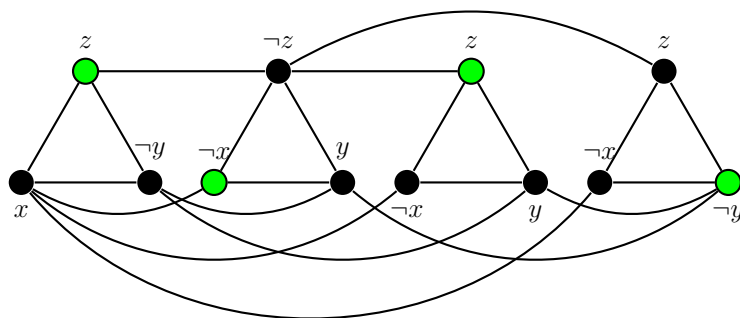
Ha az eredeti φ formula kielégíthető, akkor ez meg is tehető: vegyünk egy T kielégítő értékadást. Ez minden klózban igazra állít legalább egy literált. Válasszunk ki minden klózból egy igaz literált (mint csúcsot a gráfban). Ezek függetlenek lesznek, hiszen:

- egy háromszögből (klózból) csak egy csúcsot választunk ki,
- ha lenne köztük él, akkor az csak úgy lehet, hogy egy klózban x -et, egy másikban $\neg x$ -et választjuk ki – ez pedig nem lehet, hiszen T nem állíthatta mindkettőt igazra.

Tehát ha φ kielégíthető, akkor a generált gráfban létezik K -elemű független csúcshalmaz.

Másrészt, ha a gráfban van K darab független csúcs, akkor minden háromszögből sikerült pontosan egy literált kiválasztanunk. Állítsuk őket igazra – ezt meg tudjuk tenni, csak az okozhatna gondot, ha egyszerre próbálnánk igazra állítani egy változót és a negáltját is, ami viszont egyszerre nem szerepelhet egy független csúcshalmazban, hiszen az ilyen előfordulás-párok össze vannak kötve. Ez a (részleges) értékadás kielégíti a formulát, hiszen minden klózba kerül igaz értékű literál.

Például, a fenti példa formulát kielégíti az $x = 0, y = 0, z = 1$ értékadás és egy (ennek megfelelő) független csúcshalmaz a gráfban:



Egy nagyon hasonló probléma a KLIKK:

Definíció: KLIKK

Input: egy G gráf és egy K szám.

Output: van-e G -ben K darab páronként szomszédos csúcs? (azaz, K -elemű „klikk”?)

Állítás

A KLIKK is egy **NP**-teljes probléma.

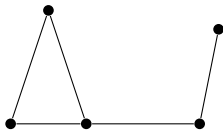
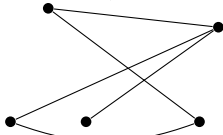
Bizonyítás

Visszavezetjük rá a FÜGGETLEN CSÚCSHALMAZ problémát. Ha \overline{G} jelöli a G gráf **komp-lementer gráfját** (melyben (u, v) pontosan akkor él, ha nem él G -ben), akkor világos, hogy egy tetszőleges X csúcshalmaz pontosan akkor független G -ben, ha klikk \overline{G} -ben.

Tehát a

$$(G, K) \mapsto (\overline{G}, K)$$

(nyilván polinomidejű) inputkonverzió egy jó visszavezetés.

Például, a , $K = 3$ Független Csúcshalmaz példányból a visszavezetés a , $K = 3$ Klikk példányt készíti. (Ebben az esetben egyébként mindkettő „nem” példánya a vonatkozó problémának.)

A következő **NP**-teljes problémánkat már ismerjük:

Állítás

A HAMILTON-ÚT is egy **NP**-teljes probléma.

Bizonyítás

A 3SATot fogjuk visszavezetni a HAMILTON-ÚTra. Ehhez induljunk ki egy φ CNF-ből.

A φ -ből generált gráfot „gadget”-ekből rakjuk össze: minden változóhoz és klózhoz létrehozunk egy-egy alkalmas gráfot (ezek a „gadget”-ek), és ezeket megfelelően kombináljuk össze.

Feltesszük, hogy minden változó előfordul φ -ben pozitívan is és negatívan is. Ezt megtehetjük, hiszen ha φ -ben x pl. csak pozitívan fordul elő, akkor az $x = 1$ értékadás „biztonságos”, ezzel az x -et tartalmazó klózokat eldobhatjuk; ha csak negatívan, akkor pedig az $x = 0$ biztonságos. Ezt az eldobálást iterálva eljutunk vagy egy üres CNF-hez (ami kielégíthető), vagy egy olyanhoz, melyben minden változó előfordul pozitívan is és negatívan is.

Induljunk ki mondjuk a

$$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

formulából.

Először is, a generált G gráfban pontosan két olyan csúcs lesz, melynek fokszáma 1, legyenek ezek s és t . Ezzel azt biztosítjuk, hogy ha van G -ben Hamilton-út, annak a két vége biztosan s és t lesz.

Minden x változóhoz készítünk egy **értékválasztó gadgetet**, mely (egyelőre) így néz ki:



Ha ez a gadget szerepel G -ben, akkor a fehér csúcsok miatt egy Hamilton-út át kell haladjon rajta; áthaladáskor pedig választania kell a „felső” és az „alsó” él közül pontosan egyet. Intuitíve ha a felsőn haladunk, akkor az az $x = 1$, ha az alsón, akkor az az $x = 0$ értékadásnak fog megfelelni.

Ezeket az értékválasztó gadgeteket sorba kötjük az alábbi ábrán látható módon:

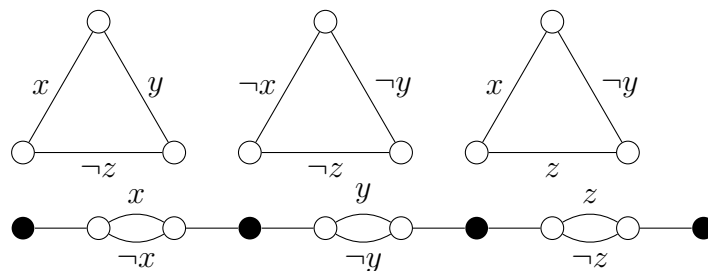


Minden változóhoz létrehozunk egy értékválasztó gadgetet, a példánkban az első az x , a második az y , a harmadik a z változóhoz készített gadget.

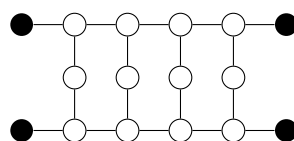
A fekete csúcsok fokszáma G -ben is 2 lesz, így egy Hamilton-út csak úgy haladhat, hogy ebbe a láncba bejön valamelyik oldalon, minden „dupla” élnél választ egy irányt a „fent” és a „lent” közül, és a másik oldalon elhagyja a láncot.

Minden klózhoz is létrehozunk egy **háromszöget** és a háromszög **éleit** címkézzük a klózbeli literál-előfordulással.

A példát tovább folytatva:

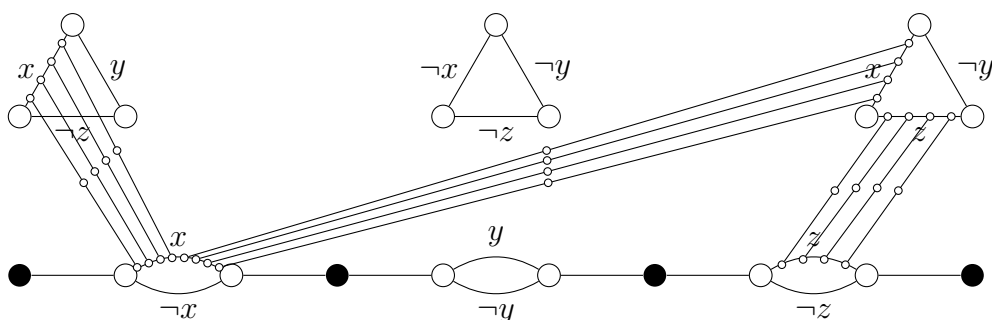


Egy Hamilton-út biztos, hogy nem tud áthaladni egy háromszögnek **mindhárom** élén. (Mert akkor bezáródna egy kis körre.) Tehát úgy szeretnénk elkódolni egy értékadást, hogy azokat az éleket **kerülje el** egy háromszögben egy értékadásnak megfelelő Hamilton-út, amik **igazak** az értékadás mellett. Egyelőre azonban nincs semmi összefüggés a klózek és a változó-értékadások közt. Az összefüggést a következő „XOR-gadgettel” fogjuk megvalósítani^a:



A belső fehér csúcsok fokszáma nem fog változni a későbbiekben. Ezt a gadgetet a kö-

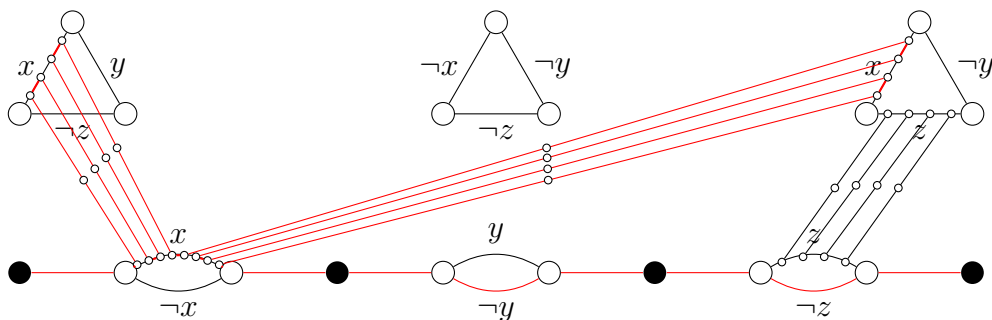
Részlet:



A XOR gadget lényege, hogy ha a gráfban van egy Hamilton-út, az csak a következő kétféleképp tud áthaladni rajta:



Pl. az $x = 1$, $y = 0$ és $z = 0$ értékadáshoz megfelelő út a gráfunk jelenlegi változatában a következő:



2020. december 19.11:49:54

Ez azt is jelenti, hogy ha egy értékadás hamisra állít egy klózban minden literált, akkor **annak a klóznak a XOR-csúcsait már nem fogjuk tudni bejárni**, mert ahhoz a háromszög mindhárom élén végig kellene mennünk, ami bezárná a készülő Hamilton-utunkat egy kis körre.

Már csak azt kell elérnünk, hogy ha viszont egy értékadás igazra állít minden klózban legalább egy literált, akkor be tudjuk fejezni a bejárást. Ezt úgy érjük el, hogy **a háromszögek eredeti sarokcsúcsait** összekötjük közvetlen élekkel is (minden ilyen csúcsot, a példában tehát behúzzuk a kilenc csúcs közé az összes lehetséges 36 élt), és ezt az értékválasztó lánc jobb végével szintén összekötjük.

Ekkor miután végimentünk az értékválasztó láncon, csak annyi dolgunk lesz, hogy sorban végiglátogassuk a klózok háromszögeinek sarokcsúcsait úgy, hogy ha egy háromszögben egy (vagy két) XOR gadget „kimaradt” (mert a nekik megfelelő literál értéke hamis), akkor azon az egy vagy két élen a kimaradt XOR gadgetek csúcsain keresztül vezetjük az utat, a többin pedig a behúzott közvetlen éleket használjuk. Ekkor az összes csúcsot sikerül végiglátogassuk az utunkkal. Végül az utolsó klóz literáljaiból elhúzzunk egy új csúcsba éleket, és abból az új t csúcsba egy élt (és az értékválasztó lánc első csúcsát kinevezzük s -nek), így hogy az út tényleg Hamilton-út lehessen, csak ezen az útvonalon haladhat.

A konstrukció miatt világos, hogy a generált gráfban pontosan akkor lesz Hamilton-út, ha az eredeti formula kielégíthető volt.

^aNote: a Papadimitriou-könyv magyar fordításában ez a gadget hibásan van lerajzolva.

Mivel pedig láttunk már egy $\text{HAMILTON-ÚT} \leq \text{TSP}(E)$ visszavezetést, ezért:

Állítás

A $\text{TSP}(E)$ probléma is **NP**-teljes.

A következő **NP**-teljes problémánkat is ismerjük már:

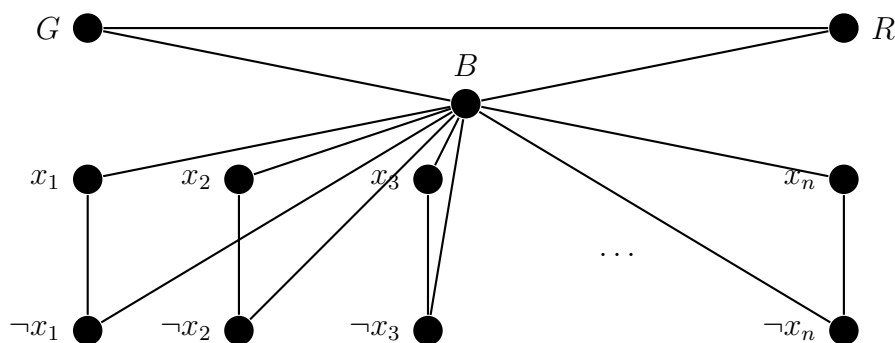
Állítás

A 3 – SZÍNEZÉS probléma is **NP**-teljes.

Bizonyítás

Ismét a 3SAT problémát fogjuk visszavezetni erre.

Legyen egy φ CNF-ünk, melyben az x_1, x_2, \dots, x_n változók fordulnak elő. Ismét gadgetekből pakoljuk össze a 3-színezendő gráfunkat. Az értékválasztó rész ezúttal így néz ki:

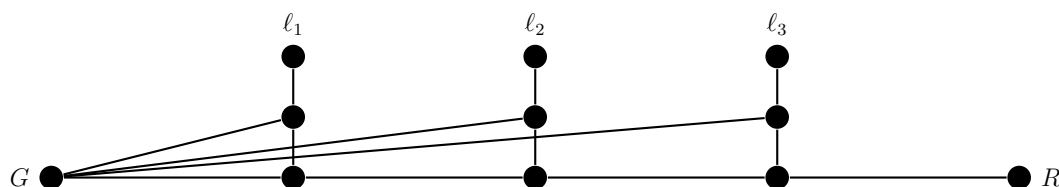


Figyelem: itt az R , G és B **nem** színek, csak a csúcsok **nevei**! Mindenesetre az biztos,

hogy az R , G és B csúcsok a gráf egy helyes 3-színezésében különböző színt fognak kapni. Nevezhetjük ezeket a színeket rendre piros, zöld és kék színeknek.

Ekkor, mivel a literálokkal címkézett csúcsok mindegyike szomszédos B -vel, így ezeket pirosra vagy zöldre kell színezzük; mivel x_i és $\neg x_i$ szomszédosak, így egyiküket pirosra, másikukat zöldre kell színeznünk. Már is összeállt egy értékadás színkódolása: a csúcsok egy színezése egyértelműen megfeleltethető annak az értékadásnak, amely a zöldre színezett literálokat állítja igazra (a pirosakat pedig hamisra).

Az $\{\ell_1, \ell_2, \ell_3\}$ klózhoz társított gadgetünk pedig a következő:

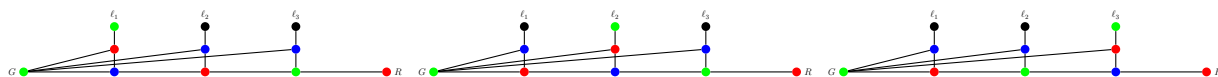


Azaz: felveszünk hat új csúcsot, és ezeket a fenti „mintának” megfelelően kötjük hozzá az $\ell_1, \ell_2, \ell_3, G$ és R csúcsokhoz.

Tudjuk, hogy egy helyes 3-színezésben G színe zöld, R színe piros kell legyen, továbbá ℓ_1, ℓ_2 és ℓ_3 mindegyikének pirosnak vagy zöldnek kell lennie.

Ha mindhárom literál-csúcs piros, akkor ezt a színezést nem lehet helyes 3-színezésre kiterjeszteni: mindegyik literál „alatti” csúcs G -vel (zöld) és a fölötte levő literállal (piros) össze van kötve, így mindegyik kék kell legyen. Csakhogy ekkor a G - R vonalon lévő három csúcsnak balról jobbra rendre pirosnak (mert balra zöld, fölül kék szomszédja van), zöldnek (mert balra piros, fölül kék szomszédja van) kell lennie, így az utolsó csúcsnak van mindhárom színből szomszédja.

Ha viszont van zöld literálcsúcs, akkor megtehetjük azt, hogy az első zöld literál szomszédját pirosra állítjuk, az alatti csúcsot kékre, a többi literál-szomszédot szintén kékre és az alsó sor maradék csúcsait alternálva piros-zölddel színezzük:



Tehát a kapott gráf pontosan akkor lesz 3-színezhető, ha az eredeti formulánk kielégíthető volt.

A színezési problémák variánsairól:

Állítás

- A 2 – SZÍNEZÉS probléma **P**-ben van (hiszen ha valakinek rögzítjük a színét, a szomszédjai „kényszerítve” lesznek, így egy egész komponenst egyszerre kiszínezzük, vagy ütközés áll elő);
- A k – SZÍNEZÉS probléma tetszőleges rögzített $k \geq 3$ -ra **NP**-teljes (ld. gyakorlat);
- Síkbarajzolható gráfok mindig 4-színezhetőek (ez a négyszíntétel);
- Síkbarajzolható gráfok 3-színezhetősége **NP**-teljes.

12. NP-teljes problémák halmazokra és számokra

Tudjuk, hogy a PÁROSÍTÁS probléma, melyet a következő módon is definiálhatunk, P-beli:

Definíció: PÁROSÍTÁS

Input: két egyforma méretű (mondjuk diszjunkt) halmaz, A és B , és egy $R \subseteq A \times B$ reláció.

Output: van-e olyan $M \subseteq R$ részhalmaza a megengedett pároknak, melyben minden $A \cup B$ -beli elem pontosan egyszer van fedve?

A probléma népszerű formája, mikor az A halmazban vannak a lányok, a B halmazban a fiúk, és az $a \in A$ lány és $b \in B$ fiú közt akkor áll fenn az R reláció, ha hajlandóak egymással *táncolni* – a kérdés pedig az, hogy ennek a relációnak megfelelően párokba lehet-e osztani őket.

A HÁRMASÍTÁS probléma ennek a háromdimenziós esete, amikor is ingatlanokkal bővül a feltételrendszer:

Definíció: HÁRMASÍTÁS

Input: három egyforma méretű (mondjuk diszjunkt) halmaz, A , B és C , és egy $R \subseteq A \times B \times C$ reláció.

Output: van-e olyan $M \subseteq R$ részhalmaza a megengedett hármásoknak, melyben minden $A \cup B \cup C$ -beli elem pontosan egyszer van fedve?

Ennek a problémának az előzővel analóg megfogalmazása, mikor az A halmazban a lányok, B -ben a fiúk, C -ben pedig a házak vannak, és egy (a, b, c) hármas akkor van R -ben, ha az a lány hajlandó a b fiúval a c házban *táncolni*. A kérdés pedig, hogy ennek a relációnak megfelelően be lehet-e osztani párokba őket és elhelyezni egy-egy házban.

Az ingatlanok itt is növelik a probléma bonyolultságát:

Állítás

A HÁRMASÍTÁS probléma is NP-teljes.

Bizonyítás

Visszavezetjük a 3SATot (surprise) a HÁRMASÍTÁSra.

Ismét lesz egy értékválasztó gadgetünk és egy klóz gadgetünk.

Induljunk ki a

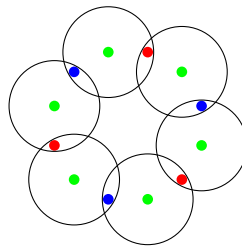
$$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

formulából.

Minden változóhoz készítünk egy értékválasztó gadgetet. Az x változóhoz úgy kapjuk meg az értékválasztót, hogy először is x változó összes előfordulását, ez legyen egy k szám; felvesszünk k lányt, k fiút, „körbeállítjuk őket” és a szomszédok közé felvesszünk egy-egy házat (összesen $2k$ házat). A megengedett hármások: eredetileg szomszédos lányok és fiúk a közjük eső házakba sorolhatóak.

Tehát pl. a fenti formulához az x változó értékválasztó gadgetje: mivel háromszor szerepel

a változó a formulában, három lányt (pirosak), három fiút (kékek) és hat házát veszünk fel:



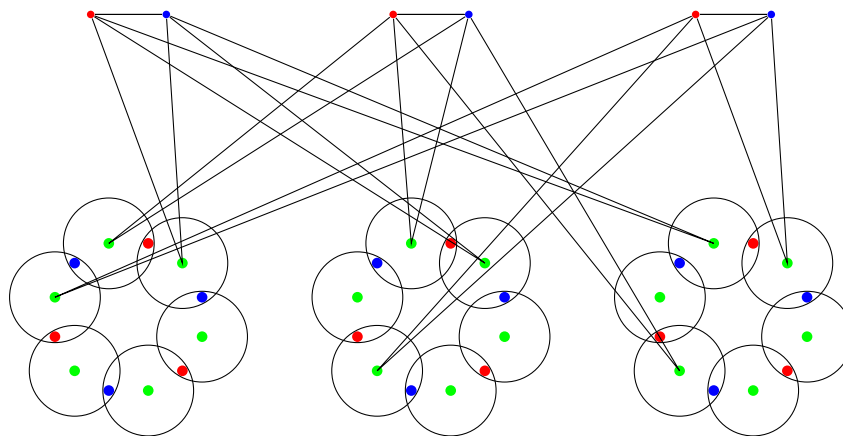
A megengedett hármasok bekarikázva szerepelnek.

Amiért ez értékválasztó gadget lesz: ezeket a lányokat és fiúkat nem fogjuk később szerepeltetni egyetlen további hármasban sem. Tehát őket összesen az alábbi két módon lehet elrendezni:



Ha a $2k$ házát felcímkézzük a képeknek megfelelő módon felváltva az x ill. $\neg x$ literálokkal, azt kapjuk, hogy a bal oldali elrendezésben k darab x címkejű ház marad szabadon (ez felel meg az $x = 1$ értékadásnak), a jobb oldaliban pedig k darab $\neg x$ címkejű ház marad szabadon (ez pedig az $x = 0$ értékadásnak).

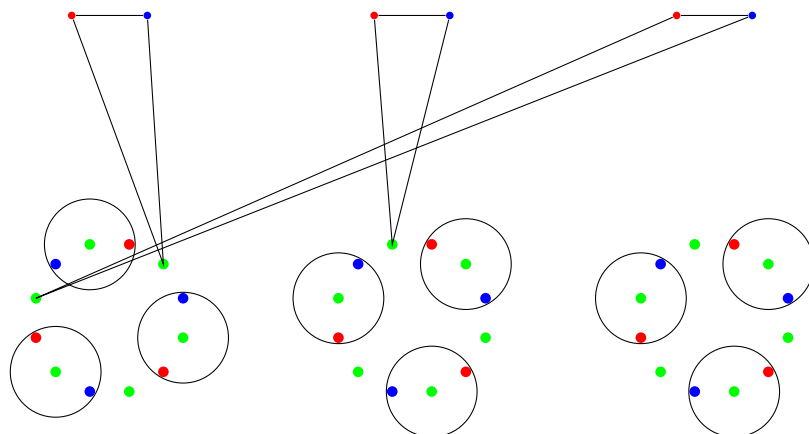
Az értékválasztó gadgetekkel kész vagyunk, nézzük most a klóz gadgeteket: egy $(\ell_1 \vee \ell_2 \vee \ell_3)$ klózhoz felvesszünk egy új lányt és egy új fiút, akik csak egymással hajlandóak táncolni, mégpedig vagy egy ℓ_1 , vagy egy ℓ_2 , vagy egy ℓ_3 címkejű házban. Úgy osztjuk ki a megfelelő házakat, hogy egy házát csak egy klózhoz vesszünk fel. Tehát ebben az esetben:



Egyelőre ott tartunk, hogy az eddig konstruált párokat pontosan akkor lehet elosztani a házakba, ha az input formulának van kielégítő értékadása, mégpedig: ekkor az értékválasztó gadgeteket a megfelelő módon osztjuk ki (szabadon hagyva az igaz értékű literál házait a gadgetban), és minden klózhoz készített párt egy „igaz” értékű házba sorolunk be. Az értékválasztó gadgetek mérete miatt ezt meg tudjuk tenni, mert minden ház legfeljebb egy klózhoz társítottunk.

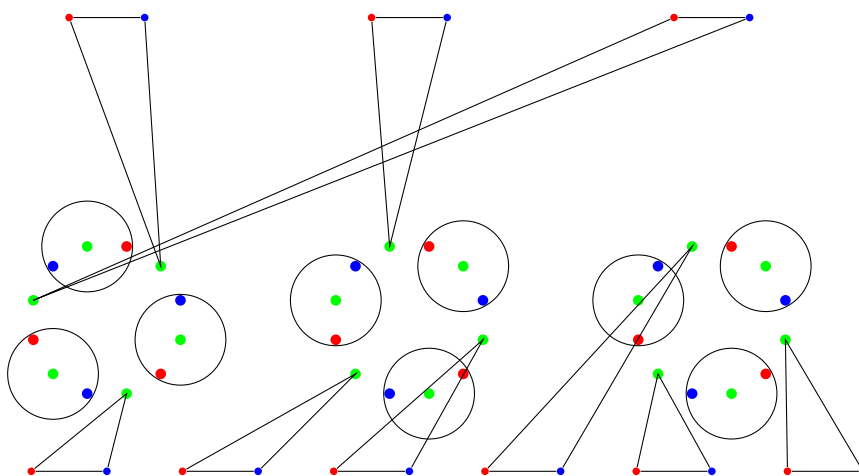
Ebben a példában pl. az $x = 1, y = 0, z = 0$ értékadás kielégítő, pl. az első klózban x -et, a másodikban y -t, a harmadikban megint x -et teszi igazzá (többek közt), az ehhez tartozó

hármásítás:



Már csak annyi hibádzik a konstrukcióban, hogy a lányok-fiúk-házak halmazok nem egyforma méretűek: házból több van és így néhány ház kimarad. Viszont pontosan tudjuk, hogy hány ház marad ki (a konstrukció közben meg tudjuk számolni a generált lány-fiú párok számát, és a házakét is, valami konstans házzal lesz több). Ezt úgy oldjuk meg, hogy ha t darab házzal van több, akkor még t lány-fiú párt felveszünk, akik csak egymással hajlandók *táncolni*, de azt bármelyik házban; ha a többieket el tudjuk rendezni, akkor a kimaradó házakat velük töltjük fel.

Ebben a konkrét példában pl. hat házzal van több, mint párossal, tehát hat párost veszünk fel, akiket minden házzal relációba hozunk. Egy konkrét megoldás ebben az esetben:



A HÁRMASÍTÁSSAL közeli rokon a következő három probléma:

Definíció: PONTOS LEFEDÉS HÁRMASOKKAL

Input: egy U $3m$ -elemű halmaz és **háromelemű** részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere.

Output: Van-e az S_i -k közt m , amiknek uniója U ?

Definíció: HALMAZLEFEDÉS

Input: egy U halmaz, részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere és egy K szám.

Output: Van-e az S_i -k közt K darab, amiknek uniója U ?

Definíció: HALMAZPAKOLÁS

Input: egy U halmaz, részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere és egy K szám.

Output: Van-e az S_i -k közt K darab páronként diszjunkt?

Állítás

A PONTOS LEFEDÉS HÁRMASOKKAL, a HALMAZLEFEDÉS és a HALMAZPAKOLÁS is **NP**-teljesek.

Bizonyítás

- A HÁRMASÍTÁS \leq PONTOS LEFEDÉS HÁRMASOKKAL visszavezetés: legyen $U = A \cup B \cup C$ (tehát a lányok, fiúk és házak mindösszesen) és $\{a, b, c\}$ akkor legyen megengedett részhalmaz, ha $(a, b, c) \in R$ megengedett hármas. Nyilván a kérdés ugyanaz, mint a HÁRMASÍTÁS esetében volt, ez egy általánosabb probléma, mint a HÁRMASÍTÁS.
- A PONTOS LEFEDÉS HÁRMASOKKAL \leq HALMAZLEFEDÉS visszavezetés: készítsük el az (U, S_1, \dots, S_n) , $|U| = 3m$ inputból az $(U, S_1, \dots, S_n, K = m)$ inputot. A kérdés ugyanaz: m darab olyan S_i -t keresünk, melyek uniója U .
- A PONTOS LEFEDÉS HÁRMASOKKAL \leq HALMAZPAKOLÁS visszavezetés: ugyanez jó. Nyilván pontosan akkor tudunk találni m darab 3-elemű halmazt, melyek uniója az U , $3m$ -elemű halmaz, ha ezeknek metszete páronként üres (mert ha nemüres, akkor az uniójuk kisebb lesz; ha pedig üres, akkor uniójuk pont $3m$ -elemű, tehát ekkor ki kell adja az egész U -t).

A következő problémánk már „számokról” szól:

Definíció: EGÉSZ ÉRTÉKŰ PROGRAMOZÁS

Input: egy $Ax \leq b$ egyenlőtlenség-rendszer, A -ban és b -ben egész számok szerepelnek.

Output: van-e **egész** koordinátájú x vektor, mely kielégíti az egyenlőtlenségeket?

Állítás

Az EGÉSZ ÉRTÉKŰ PROGRAMOZÁS egy **NP**-nehéz probléma.

Bizonyítás

Visszavezetjük rá a (surprise) 3SATot. Legyen φ egy CNF, a példában a szokásos

$$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z).$$

Ekkor minden logikai változóhoz rendelünk egy egészértékű változót.

Most az értékválasztó gadget könnyű: minden x változóhoz felveszünk egy $0 \leq x \leq 1$ feltételt. Ez épp azt jelenti, hogy x vagy 0, vagy 1; a szándék az, hogy 0 jelentse a hamis értéket, 1 pedig az igazat.

Ekkor az $1 - x$ (lineáris!) kifejezés pont $\neg x$ értékét fogja visszaadni.

Egy $(\ell_1 \vee \ell_2 \vee \ell_3)$ klózhoz pedig rendeljük az $\ell_1 + \ell_2 + \ell_3 \geq 1$ egyenlőtlenséget. Nyilván egy adott értékhalmaz mellett ez a fenti összeg minden klózra kiadja a klózbeli igaz értékű literálok számát, és akkor lesz igaz a klóz, ha legalább egy ilyen literál van. Tehát a visszavezetés tartja a választ és nyilván polinomidejű.

A példára a következő egyenlőtlenségeket kapjuk:

$$\begin{aligned} 0 &\leq x, y, z \leq 1 \\ x + y + 1 - z &\geq 1 \\ 1 - x + 1 - y + 1 - z &\geq 1 \\ x + 1 - y + z &\geq 1 \end{aligned}$$

Amit ha átrendezünk:

$$\begin{aligned} x &\leq 1 \\ y &\leq 1 \\ z &\leq 1 \\ -x &\leq 0 \\ -y &\leq 0 \\ -z &\leq 0 \\ -x - y + z &\leq 0 \\ x + y + z &\leq 2 \\ -x + y - z &\leq 0 \end{aligned}$$

és ennek pl. az $x = y = 1, z = 0$ ugyanúgy megoldása, ahogy kielégítő értékelése is φ -nek.

Az EGÉSZ ÉRTÉKŰ PROGRAMOZÁSról azt nehéz bebizonyítani, hogy **NP**-beli is (mert elsőre nem világos, hogy ha van megoldás, akkor van-e **polinom méretű** tanú-e), de **NP**-beli is, tehát **NP**-teljes.

A következő problémára aligából tanítunk dinamikus programozásos megoldó algoritmust:

Definíció: RÉSZLETÖSSZEG

Input: pozitív egészek egy a_1, \dots, a_k sorozata és egy K célszám.

Output: van-e ezeknek egy olyan részhalmaza, melynek összege épp K ?

Annak ellenére, hogy tanítunk rá algoritmust:

Állítás

A RÉSZLETÖSSZEG probléma is **NP**-teljes.

Bizonyítás

Visszavezetjük a problémára a 3SATot (duh).

A példaformulánk:

$$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z).$$

Nagy számokat fogunk létrehozni: ha n klóz van és m változó, akkor $n+m$ -jegyű számokat hozunk létre (legyen mondjuk a számrendszerünk, amiben dolgozunk, a négyes számrendszer).

Viszont a visszavezetésben ezeket a számokat csak **kiírnunk** kell, az a hosszukkal arányos, nem az értékükkel, így lehet a visszavezetés polinomidejű.

A létrehozott számok első m jegye a változók „helyiértékei”, a hátsó n jegy pedig a klózok „helyiértékei” lesznek.

Minden x_i , $i = 1, \dots, m$ változóhoz létrehozunk egy t_i (igaz) és egy f_i (hamis) számot, ez a pár most egyszerre lesz értékválasztó és klózkiértékelő gadget a következőképpen:

- t_i -ben és f_i -ben is az első m helyiértéken pontosan az i . helyen van 1-es, a többin 0;
- a hátsó n helyiérték közül a j . akkor 1-es t_i -ben, ha x_i (pozitívan) szerepel a j . klózban;
- a hátsó n helyiérték közül a j . akkor 1-es f_i -ben, ha $\neg x_i$ (negatívan) szerepel a j . klózban;
- egyébként a helyiérték 0.

Tehát erre a példára:

$$t_1 = 100101$$

$$f_1 = 100010$$

$$t_2 = 010100$$

$$f_2 = 010011$$

$$t_3 = 001001$$

$$f_3 = 001110$$

A K célszámot pedig az 111333 számra állítjuk be, tehát általában arra, melynek az első m helyiértékén 1-es, a hátsó n helyiértékén pedig 3-as szerepel.

Vegyük észre, hogy mivel 3SAT-ról beszélünk, a klózok helyiértékein egy oszlopban legfeljebb 3 helyen szerepelhet 1-es; a változók helyiértékein pedig minden oszlopban pontosan 2 helyen szerepel 1-es. Emiatt nincs maradéklépés, ezért vagyunk 4-es számrendszerben.

Ahhoz tehát, hogy az első m helyiértéken csupa 1-est kapjunk, t_1 és f_1 közül is pont az egyiket, t_2 és f_2 közül is, stb. Ez felel meg egy értékadás kódolásának: ha t_i -t választottuk, akkor $x_i = 1$, ha pedig f_i -t, akkor $x_i = 0$.

Ekkor egy klóz helyiértékén az összeg épp a benne szereplő igaz értékű literálok száma lesz. Például az $x = y = 1$, $z = 0$ értékadásnak megfelelő $t_1 + t_2 + f_3 = 111312$ lesz. Tehát: ha az értékadás kielégítő, akkor a hátsó helyiértékek mindegyike 1, 2 vagy 3; ha nem kielégítő, akkor pedig van köztük 0.

Ahhoz, hogy az első esetben lehessen ezt csupa 3-asra felbővíteni, hozzáveszünk minden $i = 1, \dots, n$ klóz-indexhez egy a_i és egy b_i értéket, mindkettő az a szám, melynek a hátsó

n közül az i . helyiértéken 1-es van, most tehát a teljes generált példányunk:

$$\begin{array}{ll} t_1 = 100101 & a_1 = 000100 \\ f_1 = 100010 & b_1 = 000100 \\ t_2 = 010100 & a_2 = 000010 \\ f_2 = 010011 & b_2 = 000010 \\ t_3 = 001001 & a_3 = 000001 \\ f_3 = 001110 & b_3 = 000001 \\ K = 111333. \end{array}$$

Tehát pl. ebben az esetben az $x = y = 1, z = 0$ kielégítő értékadáshoz kapcsolható megoldás a $t_1 + t_2 + f_3 + a_2 + b_2 + a_3 = 111333$ összeg lesz. Általában, ha az i . klózban 1 pozitív literál van, akkor a halmazba bevesszük a_i -t és b_i -t is; ha kettő, akkor (mondjuk) a_i -t, ha pedig 3, akkor egyiket sem, így egy kielégítő értékadásból ki lehet kombinálni pontosan a célösszeget. Ha viszont van olyan klóz-helyiérték, mely 0-ra összegződik, azt nem tudjuk 3-ra feltornászni, mert csak két elem, a_i és b_i áll rendelkezésünkre, melyek megnövelik ezt a helyiértéket, tehát legfeljebb 2-re tudjuk felemelni.

(Annak ellenére, hogy így már a klózik helyiértékein összesen 5 darab 1-es is állhat, továbbra sem lesz maradéktétel egy megoldásban, hiszen hogy 3 legyen az összeg, 4-es számrendszerben ahhoz 7 kellene legyen az összeg, ami nem lesz 5 darab egyesből. Tehát a megoldás úgy állhat csak elő, hogy minden helyiértéken tényleg a pontos összeg jön ki.)

(Persze 4-es számrendszer helyett lehet használni kettést is, értelemeszerű módon átírva a helyiértékeket. A teljes output hossza úgy is csak $O((n+m)^2)$, négyzetes lesz a formula méretében.)

A RÉSZLETÖSSZEG probléma akkor is nehéz marad, ha konkrétan azt kérdezzük, hogy az összeg **fele** előáll-e:

Definíció: PARTÍCIÓ

Input: pozitív egészek egy a_1, \dots, a_k sorozata.

Output: van-e ezeknek egy olyan részhalmaza, melynek összege épp $\frac{\sum_{i=1}^k a_i}{2}$?

Állítás

A PARTÍCIÓ probléma is **NP**-teljes.

Bizonyítás

Visszavezetjük rá a RÉSZLETÖSSZEG problémát.

Legyen adott az a_1, \dots, a_k részletösszeg-input a K célszámmal és legyen $N = \sum_{i=1}^k a_i$ az elemek összege. (Feltehető, hogy $0 \leq K \leq N$.)

Vegyünk fel két új elemet, $b = 2N - K$ és $c = N + K$. Ekkor az (a_1, \dots, a_k, b, c) sorozat összege $N + 2N - K + N + K = 4N$, ennek fele $2N$. Azt állítjuk, hogy ez egy visszavezetés RÉSZLETÖSSZEG-ről PARTÍCIÓ-ra.

Ha a K összeg előáll az eredeti problémában, akkor még ehhez a halmazhoz hozzávéve a $2N - K$ elemet egy $2N$ összegű halmazt kapunk, így „igen” példányból „igen” példányt

készítünk.

Ha pedig a generált példányban előállítható a $2N$ mint összeg, akkor ehhez vagy b -t, vagy c -t fel kell használnjuk (mert a többi elem összege együtt is csak N). Továbbá mindkettőt nem használhatjuk fel, mert b és c összege már $3N$, több, mint a teljes összeg fele. Tehát ha az elemek összegét meg lehet felezni, akkor az egyik félben b , a másik félben c szerepel. Amelyikben b szerepel, abban a $2N$ összeghez az kell, hogy az eredeti a_i elemek egy olyan részhalmazát tegyük b mellé, melynek összege épp K , így ha „igen” példányt készítettünk, akkor „igen” példányból is indultunk.

A következő problémát szintén ismerhetjük algáról:

Definíció: HÁTIZSÁK

Input: i darab tárgy, mindegyiknek egy w_i súlya és egy c_i értéke, egy W összkapacitás és egy C célérték.

Output: Van-e a tárgyaknak olyan részhalmaza, melynek összsúlya legfeljebb W , összértéke pedig legalább C ?

Annak ellenére, hogy erre is tanulunk dinamikus algoritmust, igaz, hogy

Állítás

A HÁTIZSÁK probléma **NP**-teljes.

Bizonyítás

Visszavezetjük rá a RÉSZLETÖSSZEG problémát.

Ha a RÉSZLETÖSSZEG inputja az a_1, \dots, a_n számsorozat és a K célszám, akkor generáljuk belőle a következő HÁTIZSÁK példányt:

- n tárgyunk van;
- az i . tárgy súlya és értéke is $w_i = c_i = a_i$;
- a kapacitás is és a célérték is $W = C = K$.

Ekkor tetszőleges $I \subseteq \{1, \dots, n\}$ részhalmazra a tárgyak összsúlya is, összértéke is $\sum_{i \in I} a_i$. Ha ennek $W = C = K$ egyszerre alsó és felső korlátja is, az csak úgy lehet, ha $\sum_{i \in I} a_i = K$, tehát ennek a hátizsák problémának egy részhalmaz pontosan akkor megoldása, ha az eredeti részletösszeg problémának is az volt.

13. Pszeudopolinomiális algoritmusok, erős és gyenge NP-teljesség

Az előző fejezet végén láthattunk olyan **NP**-teljes problémákat, melyekre ismerünk dinamikus programozásból algoritmust is. Ilyen például a HÁTIZSÁK probléma, melyre a következő két

rekurzió bármelyike megfelelő:

$$C[i, w] := \begin{cases} 0 & \text{ha } i = 0 \\ C[i - 1, w] & \text{ha } w < w_i \\ \max\{C[i - 1, w], c_i + C[i - 1, w - w_i]\} & \text{egyébként} \end{cases}$$

(itt $C[i, w]$ annak a részproblémának a maximális haszna, melyben csak az első i tárgyat használhatjuk és a hátizsákunk mérete w) vagy

$$W[i, c] := \begin{cases} 0 & \text{ha } c \leq 0 \\ \infty & \text{ha } c > 0 \text{ és } i = 0 \\ \min\{W[i - 1, c], w_i + W[i - 1, c - c_i]\} & \text{egyébként} \end{cases}$$

(itt pedig $W[i, c]$ az a minimális hátizsák-méret, mely már elegendő c haszon szerzésére, ha csak az első i tárgyat használjuk fel).

Az első esetben a szükséges időigény $N \cdot W$, a másodikban pedig $N \cdot C$, ha az input a $(w_1, c_1), \dots, (w_N, c_N), W, C$ sorozat. Ez azonban **nem polinom** az input méretéhez képest! Az input mérete ugyanis a bejövő számok **bináris reprezentációjának összes hossza**, azaz

$$n = \sum_{i=1}^N \log w_i + \sum_{i=1}^N \log c_i + \log C + \log W,$$

amihez képest az $N \cdot W$ **exponenciális** is lehet (pl. ha a hátizsák mérete, a célérték, és a tárgyak súlyai és értékei is egy-egy N -bites számmal írhatóak le, akkor az input mérete $\Theta(N^2)$ bitnyi, a fenti két algoritmus pedig $\Theta(N \cdot 2^N)$ időben tölti ki a táblázatot).

Azonban ha a számok **értéke** nem ilyen „extrém nagy”, akkor ezek az algoritmusok joggal mondhatóak mégis hatékonyak: pl. ha a tárgyak mérete (vagy értéke) korlátozható egy polinommal (mondjuk minden tárgy n^3 méretű legfeljebb, ahol n a tárgyak száma), akkor máris van egy polinomidéjű algoritmusunk, mert az algoritmus a bejövő számok **összértékének** függvényében polinomiális volt. Nem minden **NP**-teljes problémánál ez a helyzet, így pl. a $\text{TSP}(E)$ probléma **NP**-teljességére ha visszagondolunk, a $\text{HAMILTON-ÚT} \leq \text{TSP}(E)$ visszavezetésnél a generált TSP példányban minden távolság 1 volt vagy 2 – ott tehát még az se segített, ha minden bejövő „szám” (élsúly) értéke egy konstanssal volt korlátozható.

Ennek a két esetnek a megkülönböztetése, hogy „mennyire” **NP**-teljes egy probléma, gyakorlati szempontból is motivált, és ez vezet el az **erős** és a **gyenge NP**-teljesség fogalmához, a **pszeudopolinomiális algoritmusokon** keresztül.

Definíció: Pszeudopolinomiális algoritmus

Egy A algoritmus **pszeudopolinomiális**, ha van olyan c konstans, melyre az a_1, \dots, a_N inputon A futásideje $O\left(\left(\sum_{i=1}^N a_i\right)^c\right)$.

Tehát ha a futásidő a bejövő számok **mérete** helyett azok **értékének** egy polinomjával korlátozható, akkor pszeudopolinomiális.

Definíció: Gyenge NP-teljesség

Egy **NP**-teljes probléma **gyengén NP-teljes**, ha van őt eldöntő pszeudopolinomiális algoritmus.

Például a HÁTIZSÁK, RÉSZLETÖSSZEG és PARTÍCIÓ **NP**-teljes problémákra van pseudopolinomiális algoritmus (mivel a HÁTIZSÁK a legáltalánosabb közülük, így az arra adott algoritmus jó lesz a másik kettőre is), így ők gyengén **NP**-teljes problémák.

A gyenge **NP**-teljességet ellenpontosítani az „erős” **NP**-teljességgel szeretnénk, de nem mondhatjuk azt, hogy akkor legyen erősen **NP**-teljes egy probléma, ha „nincs” rá pseudopolinomiális algoritmus, mert az **NP**-teljes problémákról azt se tudjuk bizonyítani, hogy nincs rájuk polinomidejű algoritmus. Ehelyett az input **unáris ábrázolásán** keresztül definiáljuk az erős **NP**-teljességet:

Definíció: Erős NP-teljesség

Egy A **NP**-teljes probléma **erősen NP-teljes**, ha unáris számábrázolás mellett is **NP**-teljes marad, vagyis ha minden $B \in \mathbf{NP}$ problémára van olyan f polinomidejű visszavezetés, mely

- A inputjaiból B inputjait készíti **unáris ábrázolásban**
- vagyis A -nak egy I inputjából egy $f(I) = 1^{a_1}, 1^{a_2}, \dots, 1^{a_k}$ sorozatot készít (tehát: a_1 darab 1-es, vessző, a_2 darab 1-es, stb, a_k darab 1-es)
- úgy, hogy tartja a választ, azaz I az A -nak pontosan akkor „igen” példánya, ha (a_1, \dots, a_k) a B -nek „igen” példánya.

Például a SAT egy erősen **NP**-teljes probléma: a Cook-tétel bizonyításában szereplő konstrukciót átalakíthatjuk úgy is, hogy egy x_m változó kiírása helyett x_{1^m} -et írjon ki minden lépésben, máris unáris számábrázolásban készül az output és mivel csak polinom sokféle változót használunk, így ez a kiírás is csak polinomideig tart (bár persze nagyobb fokszámú polinom lesz ez így, mintha a változók indexeit binárisan írnánk ki).

A SAT-ról visszavezetéssel ami 3SATot és a számokat nem is tartalmazó gráfelméleti problémákat: HAMILTON-KÖR, FÜGGETLEN CSÚCSHALMAZ, KLIKK, stb. kaptunk, azok is mind erősen **NP**-teljes problémák. (Technikailag a FÜGGETLEN CSÚCSHALMAZ és társai, melyben egy K célszám is része az inputnak, tartalmaz számot, de ennek a célszámnak az értéke mindig legfeljebb N , a csúcsok száma, így K darab 1-es kiírása még megoldható polinomidőben).

Mivel a $\text{HAMILTON-ÚT} \leq \text{TSP}(E)$ visszavezetésben a generált távolságmátrix minden cellájában vagy 1-es, vagy 2-es szerepelt, így 2-es helyett 11-et írva ugyanez a visszavezetés legfeljebb kétszer addig tart (tehát polinom marad) és így a $\text{TSP}(E)$ is erősen **NP**-teljes.

Viszont pl. a $3\text{SAT} \leq \text{RÉSZLETÖSSZEG}$ visszavezetésénél $\Theta(n)$ -**jegyű** számokat állítottunk elő! Ha ezeket unárisan szeretnénk kiírni a visszavezetés során, akkor az $\Omega(2^n)$ lépésig tartana, tehát úgy a visszavezetés nem lesz polinomidejű. Általában egy probléma erős **NP**-teljességének megmutatásához azt kell tennünk, hogy egy **NP**-teljes problémát visszavezetünk rá úgy, hogy unáris ábrázolásban generáljuk az inputját.

Nem véletlen, hogy pont a RÉSZLETÖSSZEG problémáról nem látjuk így, hogy erősen **NP**-teljes lenne:

Állítás

Ha egy erősen **NP**-teljes problémára van pseudopolinomiális algoritmus, akkor $\mathbf{P} = \mathbf{NP}$.

Bizonyítás

Legyen A egy ilyen probléma. Tetszőleges $B \in \mathbf{NP}$ problémára akkor a következőt tegyük, ha B -nek egy I inputját kapjuk kérdésnek:

- Készítsük el az $f(I) = 1^{a_1}, \dots, 1^{a_k}$ számsorozatot, amit a B problémának az A probléma unáris változatára való visszavezetéssel kapunk. Ezt polinomidőben megkapjuk, mert a visszavezetés hatékony kell legyen. Tehát ha $|I| = n$, akkor $\sum_{i=1}^k a_i = O(n^c)$ valamilyen c konstansra (mert polinomidő alatt csak polinom méretű outputot tudunk generálni).
- Hogy az A problémának „formailag helyes” inputját kapjuk, ebből a számsorozatból készítsük el az (a_1, \dots, a_k) számsorozatot binárisan kódolva. (Ezt $O(|f(I)|) = O(n^c)$ időben szintén megtehetjük, csak számlálót kell növelni minden 1-es olvasásakor.)
- Ezen a „tömörített” inputon futtassuk az A -t megoldó pseudopolinomiális algoritmust. Ennek futásideje $O\left(\left(\sum_{i=1}^k a_i\right)^d\right)$ valamilyen d konstansra, azaz $O(n^{cd})$, szintén polinom.

Ez az algoritmus nyilván megoldja a B problémát és a teljes időkorlátja polinomiális, vagyis $\mathbf{P} = \mathbf{NP}$.

(Persze ekkor nem csak pseudopolinomiális, de polinomiális algoritmus is van minden \mathbf{NP} -beli problémára.)

Ez azt is jelenti, hogy ha $\mathbf{P} \neq \mathbf{NP}$, akkor egy \mathbf{NP} -teljes probléma egyszerre nem lehet „erősen” és „gyengén” is \mathbf{NP} -teljes, ebből a szempontból tehát jók a definíciók.

14. Approximáció

Egy pseudopolinomiális algoritmust felfoghatunk úgy, mint egy módot arra, hogy „kezeljünk” egy \mathbf{NP} -teljes problémát: bizonyos feltételek mellett mégiscsak hatékonyan megoldhatók egyes példányai.

Egy másik lehetséges megközelítés az, ha megengedjük azt, hogy a megoldó algoritmusunk **tévedjen**. Ezt, hogy „mekkora tévedés fér bele”, eldöntési problémára nem igazán lehet jól definiálni: azt az egy output bitet vagy nem szabad eltéveszteni (és akkor csak a teljesen pontos megoldás az elfogadható), vagy el szabad (és akkor meg tkp bármilyen algoritmus „majdnem megoldja” a kérdést). Ezért a **közelítő algoritmusokat** optimalizálási problémákra fogjuk definiálni.

Definíció: Optimalizálási probléma

Egy A problémát **optimalizálási problémának** nevezünk, ha

- minden I inputhoz tartozik **lehetséges megoldások** (solutions) egy $S(I)$ halmaza,
- minden $s \in S(I)$ megoldásnak van egy $c(s)$ **értéke**, egy nemnegatív valós szám,
- az I inputhoz pedig az **elvárt output a legkisebb** (ekkor minimalizálási a probléma)

ma) vagy a **legnagyobb** (ekkor maximalizálási) értékű megoldás, azaz

$$A(I) = \operatorname{argmax}_{s \in S(I)} c(s) \text{ vagy } A(I) = \operatorname{argmin}_{s \in S(I)} c(s).$$

Maximalizálási problémánál az értéket haszonnak, minimalizálásánál pedig költségnek is szokták nevezni.

Az eddig látottak közül pl. optimalizálási problémaként felfogható:

- TSP: megoldás a városok egy tetszőleges sorrendje, költsége az ilyen sorrendű körút össztávolsága;
- KLIKK: megoldás egy olyan csúcshalmaz, melyben a csúcsok páronként szomszédosak, a haszon a csúcshalmaz mérete;
- LEFOGÓ CSÚCSHALMAZ: megoldás egy lefogó csúcshalmaz, költsége a csúcshalmaz mérete;
- HÁTIZSÁK: megoldás a tárgyak egy olyan halmaza, mely belefér a hátizsákba, haszna az összes hasznuk;

és még sok másik. Egy optimalizálási problémából eddig is úgy készítettünk „eldöntési változatot”, hogy kiegészítettük az optimalizálási problémát egy K célszámmal és azt az eldöntendő kérdést tettük fel, hogy igaz-e, hogy az optimum értéke legalább/legfeljebb K . Nyilván ha az eldöntési változat „nehéz”, akkor az optimalizálási változat is az kell legyen (hiszen az eldöntési változatot mindig meg tudjuk úgy oldani, hogy megoldjuk az optimalizálási problémát, majd az optimális értéket összehasonlítjuk a K célszámmal).

Másképpen, ha egy optimalizálási probléma optimumjának *értéke* az input méretének egy **exponenciális** függvényével korlátozható, mondjuk $O(2^{n^k})$ egy k konstansra, akkor bináris kereséssel n^k -szor kérdezve az eldöntési változatot meg tudjuk határozni az optimum értékét (optimum **helyet** persze nem mindig, ahhoz további feltételek kellenek a probléma megoldásának szerkezetéről). Tehát ha az eldöntési változat könnyű, akkor az optimalizálási változat is könnyű, ha csak az értéket akarjuk kiszámítani és ha ez az érték „nem nő túl vadul”.

Ebben a fejezetben optimalizálási problémákra fogunk nézni **approximációs** algoritmusokat, melyek garantáltan polinomidőben futnak minden inputra, de melyek nem feltétlenül adnak optimális értékű megoldást; azonban, azt garantálni tudják, hogy az optimumtól legfeljebb egy konstans szorzóval rosszabbak.

Definíció: Approximációs algoritmus

Legyen A egy optimalizálási probléma és $\alpha > 0$ egy konstans. Azt mondjuk, hogy a B algoritmus az A -ra egy **α -approximációs algoritmus**, ha

- B polinomidejű,
- A -nak minden I inputjára $B(I) \in S(I)$, azaz mindig egy megoldást ad vissza;
- ha egy I inputra $\operatorname{opt}(I)$ jelöli az optimum értékét, akkor ennél $B(I)$ értéke legfeljebb α -szor rosszabb, vagyis:
 - ha A minimalizálási probléma, akkor $c(B(I)) \leq \alpha \cdot \operatorname{opt}(I)$,

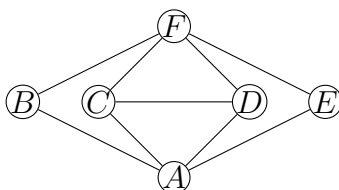
– ha A maximalizálási probléma, akkor $\text{opt}(I) \leq \alpha \cdot c(B(I))$.

Tehát pl. egy minimalizálási problémára egy 2-approximáló algoritmus olyan megoldást ad, mely az optimumnál legfeljebb kétszer drágább; egy 1.01-approximáló pedig egy olyat, mely legfeljebb $1^{23}\%$ -kal drágább. Egy maximalizálási problémára adott 2-approximáló olyat, mely az optimális haszon legalább felét eléri; egy $4/3$ -approximáló pedig legalább az optimum $3/4$ -ét²⁴. A fenti definíciókból nyilván igaz, hogy α -approximáló algoritmus csak $\alpha \geq 1$ értékekre létezik (hiszen az optimumnál jobb értéket nem adhat egy algoritmus sem) és 1-approximáló algoritmus pontosan a polinomidőben megoldható optimalizálási problémákra létezik.

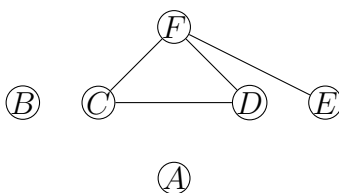
Az első approximáló algoritmusunkat a CSÚCSLEFEDÉS problémára (pontosabban annak optimalizálási változatára) adjuk:

1. Legyen $X = \emptyset$.
2. Ha G -ben már nincs több él, adjuk vissza X -et.
3. Különben válasszuk ki G -nek egy **tetszőleges** (u, v) élt.
4. Tegyük bele X -be u -t is és v -t is.
5. Töröljük ki G -ből azokat az éleket, melyek akár u -ra, akár v -re illeszkednek és menjünk vissza a 2. pontra.

Nézzünk egy példát: ha az input gráfunk a következő:



Akkor az algoritmusunk egy lehetséges futása: első lépésben mondjuk kiválasztjuk az (A, B) élt, ekkor $X = \{A, B\}$ lesz, töröljük az A -ra vagy B -re illeszkedő összes élt:



Ezután kiválasztjuk mondjuk a (C, F) élt, $X = \{A, B, C, F\}$ és a gráfunkból kitöröltünk minden élt, így visszaadjuk az aktuális X halmazt eredményként.

Mivel mindig igaz, hogy az aktuális X halmazunk lefoglalja az eredeti gráfból kitörölt éleket és addig ismétlünk, amíg ki nem törölünk minden élt, így **garantáltan egy lefoglaló csúcshalmazt kapunk**. Az is igaz, hogy az algoritmus **polinomidejű**, hiszen legfeljebb feleannyiszor fut a mag, mint ahány csúcs van a gráfban, ésszerű gráfrepresentációt (pl. éllistát) feltételezve az élek törlése, egy él kiválasztása polinomidőben megy.

²³ $1^{23} = 1$ lol

²⁴Megjegyzés: nem egységes az irodalomban, hogy az α , $\alpha - 1$ vagy az $1/\alpha$ érték jelzi, amit mi α -nak írunk itt.

Az is igaz, hogy ez az algoritmus **2-approximálja** a CSÚCSLEFEDÉST: ehhez azt kell már csak belátnunk, hogy legfeljebb kétszer annyi csúcsot választ ki, mint amennyivel már lefedhetőek az élek.

Tegyük fel, hogy az algoritmus 3. pontjában kiválasztott élek az $(u_1, v_1), \dots, (u_k, v_k)$ élek. Mikor kiválasztjuk (u_i, v_i) -t, akkor sem u_i , sem v_i nem egyezhet meg egyik korábbi u_j -vel vagy v_j -vel sem (hiszen az összes ezekre illeszkedő élt már töröltük a gráfból), tehát ezek az élek egy **párosítást** (nem feltétlenül teljes párosítást, de párosítást) alkotnak az eredeti gráfban.

Másfelől minden lefogó csúcshalmazra igaz, hogy ezeket az éleket is lefogja, tehát minden i -re vagy u_i , vagy v_i benne kell legyen minden lefogó csúcshalmazban. Mivel $2k$ különböző csúcsról beszélünk, így az optimum legalább k méretű, az algoritmus által visszaadott megoldás pedig $2k$ méretű, így ez tényleg egy 2-approximáló algoritmus.

Az persze nem igaz, hogy minden optimalizálási problémára lenne approximáló algoritmus (kivéve, ha $\mathbf{P} = \mathbf{NP}$, mely esetben ugye az egész approximálásnak nincs értelme, ha csak bonyolultságelméleti szempontból nézzük a kérdést és a polinomokat egyforma jónak tekintjük):

Állítás

A TSP probléma nem közelíthető: semmilyen α konstansra nincs α -approximáló algoritmus, hacsaknem $\mathbf{P} = \mathbf{NP}$.

Bizonyítás

Tegyük fel, hogy egy A (polinomidejű) algoritmus α -approximálja a TSP problémát. Ezt az A algoritmust arra fogjuk felhasználni, hogy megoldjuk vele polinomidőben a HAMILTON-KÖR problémát. Mivel ez utóbbi \mathbf{NP} -teljes, így azt kapjuk, hogy ekkor $\mathbf{P} = \mathbf{NP}$.

Hogy a TSP problémát megoldó A algoritmust használhassuk, először is a HAMILTON-KÖR probléma inputjából, azaz egy G gráfból elkészítjük polinomidőben a TSP probléma egy inputját, azaz egy távolságmátrixot.

Az átalakítás hasonló lesz a $\text{HAMILTON-ÚT} \leq \text{TSP}(E)$ visszavezetésben látotthoz: a gráf csúcsai lesznek a városok, és az (u, v) pár közti távolság

1. 0, ha $u = v$;
2. 1, ha (u, v) él G -ben;
3. $\lceil N \cdot \alpha \rceil + 1$ egyébként, ahol N a városok száma.

Jelölje ezt a távolságmátrixot D_G . Vegyük észre a következőket:

- Ha G -ben van Hamilton-kör, akkor D_G -ben van N összköltségű körút. (és ez az optimális).
- Ha G -ben nincs Hamilton-kör, akkor D_G -ben minden körút tartalmaz olyan lépést, amilyen él nincs G -ben, tehát ekkor minden körút költsége több, mint $N \cdot \alpha$.

Mi történik, ha ezt a D_G távolságmátrixot adjuk inputként az A algoritmusnak?

- Ha G -ben van Hamilton-kör, akkor D_G -ben az optimális körút hossza N . Mivel A egy α -approximáló algoritmus, ekkor a visszaadott körút költsége legfeljebb $\alpha \cdot N$ lesz.
- Ha G -ben nincs Hamilton-kör, akkor pedig D_G -ben az optimális körút hossza több, mint $\alpha \cdot N$. Mivel A egy megoldást ad vissza, így a visszaadott körút költsége is több lesz, mint $\alpha \cdot N$.

Tehát: ha az input G gráfból elkészítjük a D_G távolságmátrixot, majd ezt adjuk inputként az A algoritmusnak, végül a visszaadott körút költsége ha legfeljebb $\alpha \cdot N$, akkor „igen”, egyébként „nem” válasszal jövünk vissza, akkor megoldjuk a Hamilton-kör problémát, polinomidőben, ekkor tehát $\mathbf{P} = \mathbf{NP}$.

Tehát magát a TSP problémát approximálni is nehéz. Azonban ilyenkor is kereshetünk olyan **speciális esetet**, ami ugyan \mathbf{NP} -nehéz, de legalább lehet approximálni. Egy ilyen speciális eset a következő:

Definíció: METRIKUS TSP

Input: egy, a **háromszög-egyenlőtlenséget** teljesítő $D_{i,j}$ távolságmátrix: minden i, j, k -ra $D_{i,j} + D_{j,k} \geq D_{i,k}$.

Output: egy minimális költségű körút.

Érdeemes észrevenni, hogy a $\text{HAMILTON-ÚT} \leq \text{TSP}(E)$ visszavezetésben látott algoritmus konkrétan egy METRIKUS TSP inputot készít, mert ha minden távolság 1 vagy 2, akkor a háromszög-egyenlőtlenség teljesül (jó az 1, 1, 1, az 1, 1, 2, az 1, 2, 2 és a 2, 2, 2, távolság-hármas is). Tehát a METRIKUS TSP probléma még mindig \mathbf{NP} -teljes, így van értelme approximáló algoritmust keresni hozzá.

A METRIKUS TSP problémára van 2-approximáló algoritmus.

Mielőtt ismertetnénk az algoritmust, emlékezzünk vissza a **feszítőfa** fogalmára: egy G összefüggő, élsúlyozott gráfnak egy **feszítőfája** egy olyan részgráfja, mely az összes csúcsot tartalmazza (tehát csak éleket hagytunk el G -ből) és fa (tehát még mindig összefüggő, de bármely élét elhagyva már szétesik). Egy **minimális feszítőfa** pedig a gráf feszítőfái közül egy olyan, melynek az éleinek összeválasztása minimális.

Algából tanítjuk, hogy egy minimális feszítőfát polinomidőben meg tudunk határozni pl. Prim algoritmusával. Ez a következő:

1. Vegyünk egy tetszőlegesen választott csúcsot, mondjuk 1-et és tegyük bele az X halmazba. Azaz: $X = \{1\}$.
2. Amíg még van a gráfnak olyan csúcsa, mely nincs X -ben, addig
 - (a) Keressünk egy minimális súlyú olyan (u, v) élt, melyre $u \in X$ és $v \notin X$.
 - (b) Vegyük bele az (u, v) élt a feszítőfába és a v csúcsot az X halmazba.

Nyilván az algoritmus (ha a távolságmátrix elemei konstans időben lekérdezhetőek) $O(n^2)$ időben fut (értelmes reprezentációval, pl. egy vektorba szedve a nem X -beli csúcsok X -től való távolságát), tehát egy minimális feszítőfa tényleg polinomidőben elkészíthető.

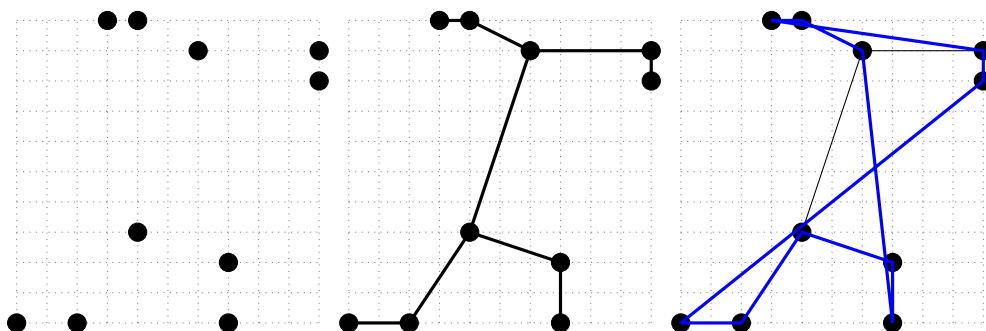
De mi köze is van egy feszítőfának egy körúthoz? Nos, egy körútból elhagyva egy élt egy utat kapunk, mely minden csúcson pontosan egyszer áthalad, ami pedig tehát egy feszítőfa. Vagyis, minden körútnál van kisebb összsúlyú feszítőfa, tehát a minimális feszítőfa összsúlya az legfeljebb annyi, mint a minimális körúté.

Azt tudjuk tehát, hogy a feszítőfa élein közlekedve bejárva a csúcsokat nem járhatunk túl rosszul. Ha ezt tesszük (tehát elindulva a fa egy tetszőleges csúcsából és az éleken mondjuk egy mélységi bejárást végezve), akkor minden élen pontosan kétszer haladunk át, egyszer „oda”, egyszer „vissza”. Közben minden csúcsot **legalább egyszer** érintünk. Az összköltsége ennek a sétának pedig kétszer annyi lesz, mint a feszítőfáé, ami meg kisebb, mint az optimális körúté, tehát

mélységi bejárva a minimális feszítőfát olyan körsétát kapunk, melynek költsége az optimális körúténál legfeljebb kétszer több.

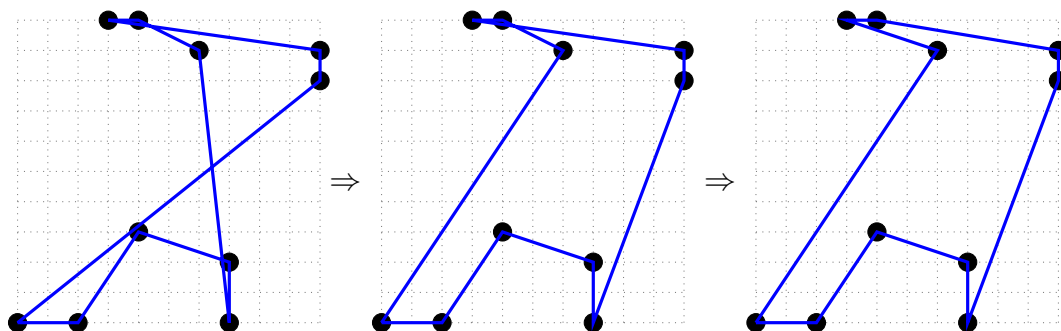
Amit eddig mondtunk, az minden távolságmátrixra működik. A háromszög-egyenlőtlenséget akkor használjuk, amikor ebből a körsétából (melyben egy csúcs többször is szerepelhet) körutat készítünk: elhagyjuk a sétából azokat a csúcsokat, melyeket már láttunk korábban. Könnyen meggondolható, hogy ezzel mindig egy többlépéses útból készítünk egy egylépéseset, ami a háromszög-egyenlőtlenség miatt nem lehet hosszabb, mint az eredeti többlépéses volt, így ezzel az ismétlődés-elhagyással nem válhat hosszabbá a sétánk, és az eredmény egy olyan körút lesz, mely 2-approximálja az optimumot.

Gyakorlaton erre az algoritmusra nézünk részletes példát is; az alábbi ábrán látható az algoritmus egy futása:

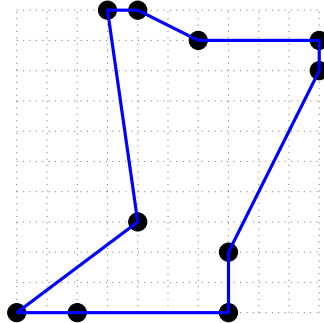


Látható, hogy a **kék** bejárás, melyet az algoritmus szolgáltatott, biztos nem optimális (mert pl. vannak benne keresztező élek – egy keresztezés a háromszög-egyenlőtlenség miatt mindig kiváltható), de már ez az útvonal is legfeljebb kétszer annyi költségű, mint az optimális.

Ha még ezen az útvonalon további optimalizálás címen sorra megszüntetjük a kereszteződéseket: ha az (A, B) és a (C, D) él keresztezi egymást, akkor vagy az (A, C) és (B, D) , vagy az (A, D) és (B, C) élpárokkal kicserélhetjük; mindenképp rövidebb lesz az útvonal és a kettő közül az egyik továbbra is kör lesz (a másik pedig két kör uniója), akkor a következőket kapjuk:



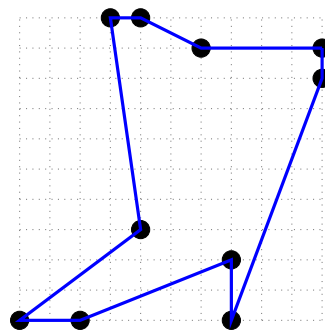
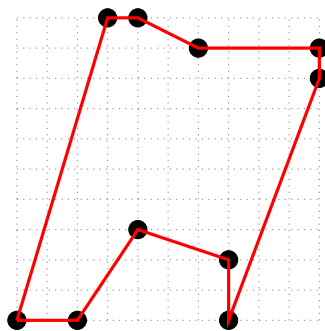
Vizuálisan persze egy ekkora inputra könnyen találunk jobb megoldást, mint pl. ez is:



A 2-approximáló algoritmusnál egy jobb, $3/2$ -approximáló (bonyolultabban implementálható) algoritmus Christofides algoritmusa:

- A gondolatmenet ugyanaz, mint korábban: a páratlan fokú csúcsok közti párosítás összszúlya legfeljebb az optimális körút költségének a fele lesz, így jön ki a $3/2$ -approximálás.

2020. december 19.11:49:54



A következő maximalizálási problémánk a MAX2SAT.

Definíció: MAX2SAT

Input. Egy olyan CNF, melyben minden klóz pontosan két literált tartalmaz, melyek változói (egy klózon belül) különböznek.

Megoldás. Egy kiértékelés.

A megoldás értéke. A kielégített klózok száma.

Tehát egy input formulához olyan értékadást keresünk, ami egyszerre a lehető legtöbb klózt kielégíti.

Dacára annak, hogy a 2SAT hatékonyan eldönthető, ez a maximalizálási probléma mégis **NP**-teljes eldöntési változattal rendelkezik:

Állítás

A MAX2SAT(E) probléma **NP**-teljes.

Bizonyítás

Emlékeztetőül, az eldöntési változatban a CNF-en kívül kapunk egy K célszámot is, és a kérdés, hogy ki lehet-e egyszerre elégíteni K darab klózt.

Az **NP**-beliség világos: generálunk egy értékadást és leellenőrizzük, hogy kielégít-e K darab klózt, easy peasy. Az **NP**-nehézséghez visszavezetjük a problémára a (dobpergés) 3SATot. Klózonként végzünk egy átírást, egy $(\ell_1 \vee \ell_2 \vee \ell_3)$ klózból a következő **tíz** klózt gyártjuk le:

$$\begin{aligned} & \ell_1 \wedge \ell_2 \wedge \ell_3 \wedge x \\ & \wedge (\overline{\ell_1} \vee \overline{\ell_2}) \wedge (\overline{\ell_2} \vee \overline{\ell_3}) \wedge (\overline{\ell_1} \vee \overline{\ell_3}) \\ & \wedge (\ell_1 \vee \neg x) \wedge (\ell_2 \vee \neg x) \wedge (\ell_3 \vee \neg x) \end{aligned}$$

ahol x egy új változó (minden eredeti klózhoz egy-egy új x -et készítünk). Vegyük az ℓ_1, ℓ_2, ℓ_3 változók egy értékadását.

1. Ha $\ell_1 = \ell_2 = \ell_3 = 1$, akkor az $x = 1$ értékadással kiegészítve **hét** klóz lesz igaz a tízből. (Ha $x = 0$, akkor csak hat.)
2. Ha az ℓ -ek közül kettő igaz, egy hamis, akkor az x nélküli hat klózból négy lesz igaz, illetve az utolsó három klózból még kettő, az összesen hat; az $x = 1$ és az $x = 0$ értékadás mellett is **hetet** tudunk kielégíteni összesen.

3. Ha az ℓ -ek közül egy igaz, kettő hamis, akkor a középső sor klózaai igazak lesznek, az három; a felső sorból az egyik ℓ_i igaz, az már négy; az alsó három klózból pedig egy lesz mindenképp igaz, az eddig öt. Ha $x = 1$ -re állítjuk, akkor hat, ha pedig $x = 0$ -ra, akkor **hét** klózt tudunk kielégíteni.
4. Ha pedig mind a három ℓ hamis, akkor az első három klóz hamis, a középső három klóz pedig igaz, ez eddig három. Ha $x = 1$, akkor négy, ha pedig $x = 0$, akkor is **csak hat** klóz lesz egyszerre igaz.

Tehát, ha volt K darab klózunk az eredeti 3SAT inputban, akkor a generált formulában pontosan akkor tudunk $7K$ klózt egyszerre kielégíteni, ha az eredeti formulánk kielégíthető. (Hiszen tetszőleges értékadás mellett ha minden klózba kerül igaz literál, akkor $7 - 7$, összesen $7K$ klóz lehet igaz; ha pedig akár egybe is hamis, abból már csak 6, a többiben pedig legfeljebb 7, összesen kevesebb, mint $7K$ tehető igazzá.)

Még annyi dolgunk van, hogy ezt a fenti konstrukciót átalakítsuk úgy, hogy klózonként pontosan két **különböző** változót tartsa a formulánk. A 3SAT persze úgy is **NP**-teljes marad, ha megköveteljük azt is, hogy egy klózon belül három különböző változó szerepeljen (lásd gyakorlat); az egységklózek mellé pedig egy új y változót bevezetve kapjuk a következő formulát:

$$\begin{aligned}
& (\ell_1 \vee y) \wedge (\ell_2 \vee y) \wedge (\ell_3 \vee y) \wedge (x \vee y) \\
& \wedge (\ell_1 \vee \neg y) \wedge (\ell_2 \vee \neg y) \wedge (\ell_3 \vee \neg y) \wedge (x \vee \neg y) \\
& \wedge (\overline{\ell_1} \vee \overline{\ell_2}) \wedge (\overline{\ell_1} \vee \overline{\ell_2}) \wedge (\overline{\ell_1} \vee \overline{\ell_3}) \\
& \wedge (\ell_1 \vee \neg x) \wedge (\ell_2 \vee \neg x) \wedge (\ell_3 \vee \neg x)
\end{aligned}$$

Ezzel elérjük, hogy a formulánk a MAX2SAT inputjának megfelelő legyen, és világos, hogy y variálásával vagy az első, vagy a második sor klózeit elégítjük ki, a másik sor pedig a korábbi verzió első sorával lesz ekvivalens. Tehát, az így generált 14 klózból egyszerre 11 lehet kielégíthető akkor, ha az ℓ_i -k közt van igaz literál, és csak 10, ha nincs.

Azt kaptuk tehát, hogy már az is **NP**-teljes, hogy az input (klózonként pontosan két különböző változót tartalmazó) formula klózainak $\frac{11}{14}$ -e kielégíthető-e. Mivel $\frac{11}{14} \approx 0.7857$, a következő állítás talán meglepő lehet²⁵:

Egy MAX2SAT input klózainak 3/4-e **mindig** kielégíthető.

Sőt, egy ennyi klózt kielégítő értékadás polinomidőben konstruálható is.

Mármost ha adunk egy olyan hatékony algoritmust, mely kielégíti a klózek 3/4-ét, az garantáltan egy 4/3-approximáló algoritmus lesz, hiszen ha K klóz van az inputban, akkor a maximálisan kielégíthetők száma is legfeljebb K , tehát $\text{opt} \leq K$ és ha legalább $X \geq \frac{3}{4}K$ klózt kielégítünk, akkor így $\text{opt} \leq \frac{4}{3}X$ igaz lesz.

Először is vizsgáljuk meg, hogy egy teljesen random értékadás (melyben minden változó értékét egymástól függetlenül $1/2 - 1/2$ valószínűséggel állítjuk be 0-ra ill. 1-re) **várható értékben**

²⁵hiszen ez valami olyasmit jelent, hogy a klózek 0.75 részét könnyű egyszerre kielégíteni, de a nem sokkal több 0.7857-ed részét már nehéz

hány klózt elégít ki! Mivel a várható érték additív, ehhez csak annyit kell tennünk, hogy minden klózra kiszámítjuk a kielégülés valószínűségét, és ezeket a valószínűségeket összeadjuk.

Egy klózbán két különböző változó van, így annak esélye, hogy mindkét literál hamisra értékelődik ki, az $1/4$ lesz, így egy klóz $3/4$ valószínűséggel lesz igaz; tehát ily módon egy teljesen uniform random értékadással is várható értékben kielégítjük a klózek $3/4$ -ét!

Vannak természetesen randomizált bonyolultsági osztályok is, de ahelyett, hogy velük foglalkoznánk, inkább csökkentjük a generált véletlen bitek számát ebben a fenti „algoritmusban”, amit úgy is mondanak, hogy **derandomizáljuk** az algoritmust. Ebben az esetben elég extrémén tudunk derandomizálni, mert nullára csökkentjük benne a véletlenbitek számát, tehát végül egy determinisztikus algoritmusunk lesz.

A determinisztikus algoritmus váza a következő. Legyenek x_1, x_2, \dots, x_n a formulában szereplő változók.

- Sorban (determinisztikusan) értéket adunk a változóknak, az első iterációban x_1 -nek, a másodikban x_2 -nek, stb.
- Amikor x_i -nek adunk értéket, akkor kiszámoljuk $x_i = 0$ -ra és $x_i = 1$ -re is azt, hogy **ha a hátralevő változóknak random adnánk értéket**, akkor várható értékben hány klóz elégülne ki.
- A két szám közül a nagyobbikat választjuk (egyenlőség esetén pl. az $x_i = 0$ -t.)

Figyeljük meg, hogy közben valójában nem generálunk véletlen biteket! Csak várható értéket kell számolnunk. Az pedig könnyű:

- Ha egy klózbán már van egy rögzített 1 értékű literál, akkor a klóz 1 valószínűséggel kielégül.
- Ha nincs, de még van benne két rögzítettlen értékű literál, akkor $3/4$ valószínűséggel.
- Ha nincs benne rögzített 1-es, de még van benne egy rögzítettlen értékű literál, akkor $1/2$ valószínűséggel.
- Ha pedig minden benne levő literál már beállt 0-ra, akkor 0 valószínűséggel elégül ki.

Ismét a várható érték additivitását használva kapjuk, hogy csak ezeket a számokat kell összeadjuk, és kijön az algoritmus 2. pontjában kért várható érték.

Nézzünk erre egy példát²⁶. Legyen a formulánk a

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \wedge (x_3 \vee x_4).$$

- Először x_1 -et állítjuk be. Az $x_1 = 0$ választás mellett az összsúly $1 + 1/2 + 1/2 + 5 \cdot 3/4$, az $x_1 = 1$ választás mellett pedig $1/2 + 1 + 1 + 5 \cdot 3/4$, ezek közül az $x_1 = 1$ adta a nagyobb értéket.
- Most az x_2 -t állítjuk be. Az $x_2 = 0$ értékadás mellett az összsúly $0 + 1 + 1 + 1/2 + 1/2 + 3 \cdot 3/4$, az $x_2 = 1$ értékadás mellett pedig $1 + 1 + 1 + 1 + 1 + 3 \cdot 3/4$, ismét az $x_2 = 1$ adja a nagyobb értéket.

²⁶A gyakorlaton ezt részletesebben látni fogjuk.

- Az $x_3 = 0$ mellett az összsúly $1 + 1 + 1 + 1 + 1 + 1 + 1/2 + 1/2$, az $x_3 = 1$ mellett pedig az utolsó három tag $1/2 + 1 + 1$, ez a nagyobb, tehát $x_3 = 1$ lesz.
- Végül x_4 -et is 1-re állítjuk. Ebben az esetben minden klóz ki is elégül, de ez persze nem mindig van így.

Amiért ez az algoritmus helyes, annak az az egyszerű oka van, hogy ha $\xi(b_1, \dots, b_n)$ jelöli annak az értékét, ahány klóz kielégül az $x_i = b_i$ értékadás mellett, akkor ha rögzítjük az első $i - 1$ értéket, úgy a többi változóra uniform random értékadás várható értéke

$$\frac{\sum_{b_i, b_{i+1}, \dots, b_n \in \{0,1\}} \xi(b_1, \dots, b_n)}{2^{n-i+1}} = \frac{\sum_{b_{i+1}, \dots, b_n} \xi(b_1, \dots, b_{i-1}, 0, b_{i+1}, \dots, b_n)}{2^{n-i}} + \frac{\sum_{b_{i+1}, \dots, b_n} \xi(b_1, \dots, b_{i-1}, 1, b_{i+1}, \dots, b_n)}{2^{n-i}},$$

vagyis minden iterációban igaz, hogy az **aktuális** várható értékünk épp az **átlaga** az $x_i = 0$, $x_i = 1$ esetekre kiszámolt várható értéknek, így a kettő közül az egyik legalább akkora lesz, mint az aktuális, és azt fogjuk választani. Ezek után az aktuális várható érték pont erre a nagyobb értékre áll be, vagyis minden iterációban igaz, hogy a várható érték monoton növekszik; mivel a klózik 3/4-éről indul, és az utolsó bit beállítása után pedig értéke a ténylegesen kielégített klózik száma lesz, így a végeredményben is legalább a klózik 3/4-e kielégül.

Az approximálás témakörben az utolsó példánk a HÁTIZSÁK probléma lesz.

Láttunk már a HÁTIZSÁK problémára egy pseudopolinomiális algoritmust, és láttuk, hogy **NP**-teljes (tehát láttuk, hogy gyengén **NP**-teljes). Ha felírjuk a w_1, \dots, w_n súlyokkal, c_1, \dots, c_n értékekkel és W kapacitással paraméterezett optimalizálási változatot, mint egészértékű lineáris programozási feladatot, ezt az egyenlőtlenség-rendszert kapjuk:

$$\begin{aligned} \sum w_i x_i &\leq W \\ 0 &\leq x_i \leq 1 \\ \sum c_i x_i &\rightarrow \max \end{aligned}$$

Itt az x_i változók értéke 0 vagy 1 lesz a középső sor feltételei miatt és mert egészértékű feladatról van szó; az 1 jelenti, hogy a tárgyat kiválasztjuk, a 0 pedig, hogy nem. Az első sor feltétele a súlykorlátot, az alsó pedig a bepakolt tárgyak összértékének maximalizálását írja le.

Ha elhagyjuk az „egészértékű” feltételt (amit úgy mondjunk, hogy a probléma **relaxált** változatát vizsgáljuk), akkor kapjuk az ún. TÖREDÉKES HÁTIZSÁK problémát. Szemléletesen ez az a változat, amikor is megengedett a tárgyak egy töredékét elvinni, és az értékük ezzel arányosan skálázódik (tehát pl. ha elvisszük az 1. tárgy felét, akkor annak a darabnak a súlya is, értéke is feleannyi, mint ha az egészet vinnénk).

Algáról tudjuk, hogy a TÖREDÉKES HÁTIZSÁKra a **mohó algoritmus** optimális eredményt ad:

1. Rendezzük a tárgyakat fajlagos érték szerint csökkenő sorrendbe (tehát $\frac{c_i}{w_i}$ szerint csökkenőbe).
2. A legjobb ár-érték arányú tárgytól a legrosszabb felé haladva pakoljuk be a tárgyakat teljesen, az elsőt, amelyik már nem fér be, pedig törjük el, hogy töltsen meg a hátizsákot teljesen.

Persze ha minden tárgy eleve belefér a hátizsákba, akkor tegyük bele mindet és nem kell törni (ahogy akkor sem, ha a valahányadik tárgy épp kitölti a hátizsák maradék részét). Ebben a

két esetben (és csak ezekben) a kapott (optimális) megoldás egészértékű is lesz, és ekkor ez az optima az egészértékű HÁTIZSÁK problémának is.

Felmerül a kérdés, hogy vajon a fenti mohó algoritmusnak a következő **egészértékű mohó** változata approximálja-e a HÁTIZSÁK problémát:

1. Rendezzük a tárgyakat fajlagos érték szerint csökkenő sorrendbe.
2. A legjobb ár-érték arányú tárgytól a legrosszabb felé haladva, végighaladva az összes tárgyon, ha az aktuálisan vizsgált tárgy befér a zsákba, akkor tegyük bele, ha nem, akkor ugorjuk át.

Ez az algoritmus önmagában még nem approximál: ha pl. az egyik tárgyunk a $w_1 = 1$, $c_1 = 2$, a másik pedig a $w_2 = W$, $c_2 = W$, úgy a mohó algoritmus berakja az első tárgyat (mivel annak a fajlagos haszna 2, a másiké pedig csak $\frac{W}{W} = 1$), a másodiknak így nem marad hely és lejelenti a 2 hasznú megoldást. Ha viszont $W \geq 2$, akkor az optimum épp a második tárgy elvitele lenne, W haszonnal; így az arány $\frac{W}{2}$, ami tetszőleges α konstansnál nagyobbá tehető, tehát **az egészértékű mohó algoritmus nem approximál**.

Egy egyszerű módosítása viszont igen:

1. Dobjuk ki azokat a tárgyakat a listából, melyek egyedül se férnek be a zsákba.
2. Futtassuk az egészértékű mohó algoritmust, ez ad egy megoldást.
3. Ha ez a megoldás nem ugrott át tárgyat (mert bepakolt mindent), akkor ezt adjuk vissza.
4. Különben nézzük meg az **első** átugrott tárgyat. Ha ennek az értéke egyedül jobb, mint a mohó által adott megoldásé, akkor ezt adjuk vissza, egyébként a mohó eredményét.

Például az előző példában a mohó algoritmus a második tárgyat átugorja, mert annak nincs hely már a zsákban; ez a módosított változat viszont megnézi, hogy ennek a tárgynak az értéke (W) jobb, mint a mohó által visszaadotté (2), ezért ezt a tárgyat teszi bele a zsákba megoldásként.

Állítás

A fenti módosított változata az egészértékű mohó algoritmusnak 2-approximálja a HÁTIZSÁK problémát.

Bizonyítás

Az első lépésben eldobott tárgyak nem szerepelnek egyetlen megoldásban sem, így őket valid eldobni.

A harmadik lépésben, ha az algoritmus nem ugrott át tárgyat, azaz berakott minden tárgyat a zsákba, nyilván egy optimális megoldásunk van.

Egyébként a (töredékes) mohó algoritmus az első átugrott tárgyat törné el. Jelölje $\mathbf{x} = (1, 1, \dots, 1, x, 0, 0, \dots, 0)$ a töredékes mohó algoritmus optimumhelyét (ebben a vektorban az elemek már fajlagos érték szerinti csökkenő sorrendben szerepelnek), mondjuk kezd k darab 1-essel.

Az egészértékű mohó algoritmus által visszaadott \mathbf{u} megoldás is k darab 1-essel kezdődik, hiszen ezek a tárgyak teljesen beférnek. A negyedik lépésben megvizsgált megoldás pedig a $\mathbf{v} = (0, 0, \dots, 0, 1, 0, 0, \dots, 0)$ vektor, ahol a $k+1$. elem az 1-es. (Azért dobtuk el az első lépésben a nem-beférő tárgyakat, hogy ez tényleg egy megoldás legyen).

Vagyis azt kaptuk, hogy erre a két vektorra $\mathbf{u} + \mathbf{v} \geq \mathbf{x}$. Tehát $\mathbf{c}^T \mathbf{u} + \mathbf{c}^T \mathbf{v} = \mathbf{c}^T (\mathbf{u} + \mathbf{v}) \geq \mathbf{c}^T \mathbf{x}$, így a két érték közül a nagyobbik legalább $\frac{\mathbf{c}^T \mathbf{x}}{2}$. Mivel pedig a számlálóban a relaxált változat optimauma áll (hiszen tudjuk, hogy a töredékes mohó arra optimumot ad vissza), ami legalább annyi, mint az egészértékű (mert a célfüggvény ugyanaz, a lehetséges megoldások tere pedig nagyobb), ezért $\frac{\mathbf{c}^T \mathbf{x}}{2}$ legalább az egészértékű optimum fele, vagyis az algoritmus tényleg 2-approximál.

Gyakorlaton ennek az algoritmusnak egy kifinomultabb (de még mindig 2-approximáló) változatára nézünk példát.

Egy további javítása az algoritmusnak, ha nem csak egy tárgyat, hanem k tárgyat rögzítünk le fixen a zsákba, ahányféleképp csak lehet, és a maradék tárgyakra futtatjuk az egészértékű mohó algoritmust:

1. Legyen k egy konstans paraméter.
2. Ahányféleképp csak lehet, válasszunk ki az n tárgyból legfeljebb k darabot; ha a kiválasztott tárgyak beférnek együtt a zsákba, akkor tegyük bele őket és a maradék tárgyakon a zsákban fennmaradó helyen futtassunk egy egészértékű mohó algoritmust.
3. Az összes lehetséges k -as közül a legnagyobb értéket nyújtót adjuk.

(Gyakorlaton a fenti algoritmust vesszük a $k = 1$ paraméterre, azaz mikor egy tárgyat rögzítünk a zsákba ahányféleképp csak lehet, a többire pedig futtatjuk a mohót.)

A fenti algoritmus-család $O(n^{k+1})$ időigényű (mivel k -asból van $O(n^k)$ és mindre egy lineáris idejű mohót futtatunk, a tárgyakat elég az elején egyszer lerendezni), tehát **rögzített k -ra polinom**, és (bizonyítás nélkül) igaz az is, hogy **$1 + \frac{1}{k}$ -approximál**.

Vagyis, mivel $1 + \frac{1}{k}$ tart 1-hez, így ha egy adott ϵ konstans approximáló hiba alá szeretnénk menni, akkor „csak” annyit kell tennünk, hogy a fenti algoritmust futtatjuk $k = \frac{1}{\epsilon}$ paraméterrel. Tehát ha pl. egy 1.1-approximálás a célunk, azaz $\epsilon = 0.1$ közelítési hibát engedünk meg, akkor $k = 10$ -es paraméterrel futtatjuk az algoritmust (és ekkor egy $O(n^{11})$ időigényű algoritmusunk lesz, ami garantálja nekünk a 0.1-es hibatagot). Ha $\epsilon = 0.01$ hibát szeretnénk, akkor egy $O(n^{101})$ időigényű²⁷ algoritmusunk lesz.

Amit kaptunk, azt a szaknyelvben polinomidejű approximációs sémának, polynomial-time approximation scheme, PTAS nevezik:

Definíció: Polinomidejű approximációs séma, PTAS

Az \mathcal{A} algoritmus(-család) az A optimalizálási problémára **polinomidejű approximációs séma**, ha tetszőleges ϵ paraméterre $1 + \epsilon$ -approximálja A -t, és tetszőleges *előre rögzített* ϵ -ra futásideje polinom.

²⁷yikes

Tehát a fenti módosított változata a mohó algoritmusnak, mikor is k tárgyat rögzítünk, ahányféleképp csak lehet, az egy PTAS a HÁTIZSÁK problémára.

Azért az előző algoritmussal van egy kis gond, mégpedig a polinom fokszáma: a futásidő $O(n^{1+\frac{1}{\epsilon}})$ lesz. Jobban szeretnénk, ha a hiba paramétere **kitevőben** nem jelenne meg (különben egy n^{101} futásidejű algoritmust kapunk, amiért annyira nem rajong a lelkünk, hiába polinom).

Erre is van név:

Definíció: Teljesen polinomidejű approximációs séma, FPTAS

Az \mathcal{A} algoritmus(-család) az A optimalizálási problémára **teljesen polinomidejű approximációs séma**, ha olyan polinomidejű approximációs séma, melyre van olyan k konstans, hogy futásideje $O((n + \frac{1}{\epsilon})^k)$ tetszőleges $\epsilon > 0$ megengedett közelítési hiba és n méretű input esetén.

Az előző család pl. **nem** FPTAS, csak PTAS, hiszen az $\frac{1}{\epsilon}$ fenn van a kitevőben.

Bizonyítás nélkül, a következő algoritmus viszont FPTAS a HÁTIZSÁK problémára:

1. Legyen az eredeti HÁTIZSÁK probléma inputja $w_1, \dots, w_n, c_1, \dots, c_n, W$. Legyen $c = \max c_i$ a legnagyobb érték.
2. Készítsük el a $w_1, \dots, w_n, c'_1, \dots, c'_n, W$ inputját a HÁTIZSÁK problémának, ahol $c'_i = \lfloor \frac{c_i \cdot n}{c \cdot \epsilon} \rfloor$.
3. Ezt a kapott problémát oldjuk meg a HÁTIZSÁKra látott második pseudopolinomiális algoritmussal (amelyik a $W[i, c]$ tömböt tölti ki $O(n \cdot C)$ időben).
4. A kapott optimumhelyet adjuk vissza.

Csak azt nézzük meg, hogy ez a fenti algoritmus valóban egy megoldást ad vissza $O((n \cdot \frac{1}{\epsilon})^k)$ időben.

Egyrészt, a tárgyak súlyát és a hátizsák méretét nem változtatjuk (csak az értékeket), tehát a módosított probléma megoldásai ugyanazok, mint az eredetie. Mivel a pseudopolinomiális algoritmus mindig egy lehetséges megoldást ad vissza (sőt, az optimálisat), így tényleg megoldást kapunk.

Most nézzük meg a c'_i mágikus értékeket, mi történik. Ténylegesen az történik, hogy átskálázzuk az értékeket: a legdrágább, c értékű tárgy új értéke pl. $\lfloor \frac{n}{\epsilon} \rfloor$ lesz, a fele ekkora értékű tárggyé a fele (lefele kerekítve), stb. Tehát a tárgyak „arányaiban” nagyon hasonló értékekkel szerepelnek a módosított feladatban, leszámítva a kerekítési hibát.

Az algoritmus időigénye pedig: ebben a módosított feladatban egy tárgynak a maximális értéke $\frac{n}{\epsilon}$, tehát mivel n tárgy van, így az elméleti maximális haszon is csak $C = \frac{n^2}{\epsilon}$, a teljes $n \cdot C$ méretű táblázat kitöltése pedig így $O(\frac{n^3}{\epsilon})$ idő alatt megtörténik. Tehát, ez egy $O((n + \frac{1}{\epsilon})^4)$ időigényű algoritmus.

Bizonyítás nélkül, a kerekítési hibák lehetséges hatásai miatt lesz ez az algoritmus nem feltétlenül optimális, de $(1 + \epsilon)$ -approximáló. Tehát ez **egy FPTAS** a HÁTIZSÁK problémára.

Van összefüggés FPTAS és pseudopolinomiális algoritmusok közt:

Állítás

Ha egy egészértékű optimalizálási probléma optimum értéke korlátozható az input n méretének egy polinomfüggvényével, és van rá FPTAS, akkor van rá pszeudopolinomiális algoritmus.

Bizonyítás

Ha a probléma optimumértékére a korlát n^c , akkor ha $\epsilon = \frac{1}{n^c}$ hibakorlátot megengedve és futtatva az FPTAS-t garantáltan az optimális megoldást kapjuk (hiszen ha az output egy $x \in \{0, 1, \dots, n^c\}$ egész szám, és kapunk egy megoldást, értékével az $[x \cdot (1 - \frac{1}{n^c}), x \cdot (1 + \frac{1}{n^c})]$ intervallumban, akkor az csak x lehet, hiszen nincs benne másik egész szám). A futásidő pedig $(n + n^c)^k$ valamilyen k konstansra, tehát (mivel c is konstans) polinom.

15. P és NPC közt

Az eddigiek alapján úgy tűnhet, mintha minden NP-beli probléma vagy P-beli, vagy NP-teljes lenne. Ladner tétele (bizonyítás nélkül) szerint ez nincs így:

Állítás: Ladner tétele

Ha $P \neq NP$, akkor van olyan NP – P-beli probléma, mely **nem** NP-teljes.

Ezeket az NP-beli problémákat, melyek nem annyira nehezek, hogy NP-teljesek legyenek, de annyira nehezek, hogy már ne lehessen őket hatékonyan (azaz polinomidőben) megoldani, **NP-köztes**, NP-intermediate problémáknak nevezzük.

Persze ha $P = NP$, akkor nincs NP-köztes probléma; mivel nem tudjuk, hogy $P = NP$ vagy sem, ezért egyelőre azt se tudjuk, hogy vannak-e NP-köztes problémák vagy sem. Mindenesetre **jelöltek** (candidates) vannak:

Definíció: GRÁFIZOMORFIZMUS

Input: két gráf.

Output: Izomorfak-e?

Tehát a kérdés, hogy „ugyanaz”-e a két gráf, csak a csúcsaikat kell átnevezni vagy sem.

Definíció: FAKTORIZÁLÁS

Input: N és K pozitív egészek.

Output: van-e N -nek K -nál kisebb prímosztója?

Így ebben a formában egy eldöntési problémát kapunk; a vonatkozó függvényproblémában az input az N szám, az output pedig prímtényező felbontása. Persze ismét, ha az eldöntési változatra van $f(n)$ időigényű algoritmus, akkor a függvényproblémára is: felezve kereséssel $O(n)$ darab hívással megkapjuk az n -bites szám egy prímtényezőjét, és mivel egy n -bites számnak

legfeljebb n prímtényezője van (mert n prímszám szorzata már legalább 2^n , ami $n+1$ -bites), így egy $f(n) \cdot n^2$ időigényű algoritmust kapunk a prímtényezőzés felbontás előállítására. (Itt persze $n = \log N$ körüli, az N szám bináris reprezentációjához kellő bitek száma).

A fenti két problémáról ugyan megoszlanak a vélemények, de sokak szerint ezek valóban **NP**-köztes problémák. A GRÁFIZOMORFIZMUSra jelenleg ismert és széles körben elfogadott aszimptotikusan leggyorsabb algoritmus $O(2^{\sqrt{n \log n}})$ időigényű, de 2015. novemberben ezt állítólag megjavították $O(2^{\log^c n})$ -re, ahol c egy konstans, ennek a bizonyításnak az ellenőrzése még folyik. A FAKTORIZÁLÁSra a legjobb algoritmus szintén szubexponenciális, $2^{O(\sqrt[3]{n \log^2 n})}$ -es (szubexponenciális, azaz $o((1+\epsilon)^n)$ -es minden $\epsilon > 0$ -ra). Erről azt tudjuk, hogy **NP** \cap **coNP**-beli, hiszen magát a prímtényezőzés felbontást nemdeterminisztikusan ki lehet generálni és ellenőrizni (felhasználva, hogy a prímszám-tesztelés **P**-ben van, ezt 2002 óta tudjuk), és ezzel az „igen” és a „nem” példányokra is van polinomidőben verifikálható tanúsítvány-rendszer. Tehát ha a FAKTORIZÁLÁS egy **NP**-teljes probléma lenne, akkor **NP** = **coNP**, amiben a területtel foglalkozók többsége nem hisz, ezért nem gondolják **NP**-teljesnek a problémát. Polinomidejű algoritmust pedig szintén nem talált még rá senki (recall: ha N az input, akkor $n = O(\log N)$ az input mérete, ehhez képest kell polinom legyen a futásidő).

A szubexponenciális algoritmusok létezése is tekinthető egyfajta érvnek amellett, hogy ezek a problémák **nem NP**-teljesek, mert van egy sokak által elfogadott sejtés, az EXPONENCIÁLIS IDŐHIPOTÉZIS, miszerint:

A 3SATot (és sok más **NP**-teljes problémát) nem lehet szubexponenciális időben determinisztikusan eldönteni.

Persze ez a sejtés erősebb, mint a **P** \neq **NP**, hiszen ha ez igaz, akkor nyilván nem lehet egyenlő a két osztály - tehát ha igaz, azt még nehezebb bebizonyítani.

16. Savitch tétele

A számítási modellek (RAM és Turing-gép) definíciójánál láttuk, hogy nem csak az időigénnyel, hanem a tárigénnyel is foglalkozunk a kurzuson. Most a korábban definiálnál egy egyszerűbb tárigeny-fogalommal fogunk dolgozni: pszeudokódjainkban egy-egy változó által igényelt tárterület a benne valaha tárolt legnagyobb szám bináris reprezentációjának a hossza (kb. a logaritmus) lesz, és ezeket egyszerűen összeadjuk. Mivel minden kódunk konstans sok lokális változót fog deklarálni, ez csak egy konstanssal tér el a RAM esetében definiált „adjuk össze a benne tárolt legnagyobb értéket és a tároló regiszter címének a hosszát” tárigeny-fogalomtól, cserébe intuitívabb, könnyebb számolni vele és ezt használjuk algán is.

Ebben a részben mutatunk egy algoritmust, mely szublineáris tárban dönti el az ELÉRHETŐSÉG problémát. Emlékeztetőül: ahhoz, hogy $o(n)$ tárról beszélni tudjunk egyáltalán, az kell, hogy az inputot read-only módban használjuk, különben az input méretét is hozzá kellene adni a tárigenyhez, és úgy már nem mehetnénk semmiképpen n alá.

Mondjuk az algoritmus számára legyen adott a $G = (V, E)$ gráf. Nézzük a következő algoritmust:

```

1: function F( $x, y, d$ )
2:   if  $d == 0$  then
3:     return  $x == y$  OR  $(x, y) \in E$ 
4:   for  $z \in V$  do
```



```

5:      if  $F(x, z, d - 1)$  AND  $F(z, y, d - 1)$  then
6:          return true
7:      return false

```

Mivel ez egy rekurzív (most abban az értelemben, hogy hívja önmagát) algoritmus, a tárigényét elemezni kicsit összetettebb lesz, mint egy sima összeadás. Először próbáljuk meg megfejtetni, hogy mit csinál, mi a **szemantikája** ennek a kódnak.

Az állítjuk, hogy $F(x, y, d)$ pontosan akkor lesz true, ha **x -ből y elérhető egy legfeljebb 2^d hosszú sétával.**

Ezt d szerinti indukcióval igazoljuk: ha $d = 0$, akkor a legfeljebb $2^d = 2^0 = 1$ hosszú séta azt jelenti, hogy x -ből y legfeljebb egy lépésben elérhető – vagyis vagy $x = y$ (akkor nulla lépésben), vagy (x, y) egy él G -ben. A harmadik sorban épp ezt teszteljük, tehát $d = 0$ -ra igaz az állítás.

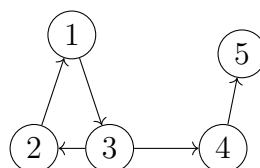
Most nézzük meg $d > 0$ -ra. Ha x -ből y -ba van egy legfeljebb 2^d hosszú séta, akkor annak van egy középső (páratlan hossz esetén bármelyik „középső” jó lesz a kettőből) csúcsa, legyen ez mondjuk z . Nyilván ezen a sétán x -ből z -be legfeljebb feleannyi, tehát 2^{d-1} lépésben, és utána z -ből y -ba szintén legfeljebb 2^{d-1} lépésben eljutunk – tehát erre a z -re indukció szerint az ötödik sor feltétele igaz lesz, így az algoritmus tényleg true-val tér vissza. Másfelől, ha az algoritmus true-val tér vissza, azt csak úgy teheti, hogy az ötödik sor feltétele igazzá válik, ami pont azt jelenti, hogy van olyan z csúcs, mely legfeljebb 2^{d-1} lépésben elérhető x -ből, és ahonnan y szintén ugyanennyi lépésben elérhető, ezt a két sétát összeragasztva kapunk egy legfeljebb 2^d hosszú sétát, tehát ez is stimmel.

Vagyis, ha a gráfnak N csúcsa van, akkor az $F(s, t, \lceil \log N \rceil)$ hívással megkapjuk, hogy az s csúcsból a t csúcs elérhető-e legfeljebb $2^{\lceil \log N \rceil} \geq N$ lépésben, tehát azt, hogy egyáltalán elérhető-e s -ből t . Vagyis így **eldönti az ELÉRHETŐSÉG problémát.**

Most kielemezzük a tárigényt. Először is, az F függvényben használjuk az x, y és z változókat, melyek egy-egy csúcsot vesznek fel értékül, másrészt a d számot, ami egy pozitív egész, legfeljebb $\log N$. Ha a gráf csúcsai az $1, \dots, N$ számok, akkor ezeket tárolni is elég $O(\log N)$ bit, tehát az F függvény **egy példánya** $O(\log N)$ tárigényű. Ezt a mennyiséget **fel kell szoroznunk azzal, ahány példányban F jelen van a memóriában egyszerre.** Ehhez idézzük fel a függvényhívás mikéntjét: ha egy ponton hívunk egy függvényt, akkor a verembe letesszük **a helyet, ahova vissza kell térnünk**, a függvény **argumentumait** és utána beugrunk a hívott függvényblokk első sorára. Visszatéréskor pedig kivesszük a veremből a kapott argumentumokat a saját lokális változóinkkal együtt, amiket közben létrehoztunk, kivesszük a visszatérési címet, odaugrunk és (mondjuk) a verembe visszarakjuk a visszatérési értéket.

A verembe lerakott „a helyet, ahova vissza kell térnünk” most egy A vagy B fogja azonosítani: ha A -t teszünk le, az jelzi, hogy ez most az ötödik sor $F(x, z, d - 1)$ hívása, ha meg B -t, az az $F(z, y, d - 1)$ hívás.

Nézzük az egészet egy példán, legyen a gráfunk



Ha ezen a gráfon mondjuk az $F(1, 5, 2)$ hívást végezzük (tehát 1-ből kérdezzük, hogy el lehet-e jutni 5-be $2^2 = 4$ lépésben legfeljebb), akkor a hívási verembe bekerül $x = 1$, $y = 5$ és $d = 2$, csak azért írjuk persze ki az x, y, d neveket, hogy jobban látsszon, mi történik; a vermet, hogy

szebben elférjen, most balról jobbra írjuk ki, jobb oldalt lesz a teteje:

$$x = 1; y = 5; d = 2$$

A függvény ekkor teszteli, hogy d vajon 0-e, nem, továbbmegy, létrehozza a z lokális változóját a veremben, adja neki a $z = 1$ értéket:

$$x = 1; y = 5; d = 2; z = 1$$

Elérkezik az ötödik sorra, ott az első függvényhívásnál leteszi, hogy A -ba kell visszatérjen, és bemásolja az argumentumokat:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1$$

(az $y = 1$ pl. onnan jön, hogy a híváskor a második formális paraméter, y , a z aktuális értékét 1-et fogja kapni.) Eztán megint beugrik a második sorba, rekurzívhívni magát.

Itt megint teszteli, hogy vajon $d = 0$ -e, hát nem, továbbmegy, létrehozza a saját z lokális változóját a veremben $z = 1$ értékkel:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1; z = 1$$

és a saját ötödik sorában is csinálja az első rekurzív hívást:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1; z = 1; A; x = 1; y = 1; d = 0$$

ugrik a második sorra. Ellenőrzi, hogy vajon $d = 0$ -e, hát bizony, és mivel ezen a szinten $x == y$, így **true**-val jön vissza, takarít és visszaugrik az A pontba, mert az van írva a verembe:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1; z = 1; \text{true}$$

és most a vezérlés az ötödik sor ifjének az AND szálán van. Ez OK, most akkor jön az $F(z, y, d - 1)$ hívás, másolunk, és ez a B pont, ahova visszatérnénk:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1; z = 1; B; x = 1; y = 1; d = 0$$

és bizony mivel $d == 0$ és $x == y$, ez a példány is **true**val tér vissza, takarítás után:

$$x = 1; y = 5; d = 2; z = 1; A; x = 1; y = 1; d = 1; z = 1; \text{true}$$

tehát az ötödik sor éselése végül igaz lett, ezért ez a példány is **true**val tér vissza, takarít maga után:

$$x = 1; y = 5; d = 2; z = 1; \text{true}$$

ezen a szinten pedig szintén egy A pontba tértünk vissza, kiértékeljük az éselés másik ágát:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1;$$

belépve F -be ellenőrizzük, hogy $d = 0$ -e, nem, jön a $z = 1$ a verembe és a rekurzív hívás:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 1; A; x = 1; y = 1; d = 0$$

ez $d = 0$ és $x = y$, tehát **true**:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 1; \text{true}$$

mivel igazat kaptunk az A pontban, továbbmegyünk a B hívással:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 1; B; x = 1; y = 5; d = 0$$

ez viszont hamis, mert $d = 0$ és $x = 1, y = 5$ nem is egyenlőek és él sincs köztük, **false**:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 1; \text{false}$$

tehát az ötödik sor éselése hamis lesz, akkor növeljük z értékét, megy a forciklus:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 2;$$

és erre megint az ötödik sorban lévő A pontot hívjuk:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 2; A; x = 1; y = 2; d = 0$$

beugrunk a kód elejére, mivel $d = 0$ és $(1, 2)$ nem él (mert a gráf irányított), hát **false** ez is:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 2; \text{false}$$

Ez egy A pontba ugrott vissza, a gyorsított kiértékelés szabályai szerint ha már az éselés egyik ága hamis, akkor a másikat ki se értékeljük, tehát az **if** hamis, megy tovább a forciklus, növeljük z -t:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 3;$$

és hívjuk az ötödik sor A pontjáról rekurzívan magunkat:

$$x = 1; y = 5; d = 2; z = 1; B; x = 1; y = 5; d = 1; z = 3; A; x = 1; y = 3; d = 0$$

ez végre megint igaz lesz, mert $d = 0$ és $(1, 3)$ egy él a gráfban, de ezek után meg az $x = 3; y = 5; d = 0$ ág fog hamisat visszaadni stb.

A teljes futás **időigénye** „nagyon sok”, ez érződik: valójában $N^{\log N}$ környékén mozog, tehát nem is polinom. Viszont a **tárigény** a fenti példa alapján jó: ahogy látjuk, **minden d -re egyszerre csak egy példány van jelen a memóriában** (tulajdonképpen ezért jó leszálási feltételnek a d , meg azért, mert csökken), mert amikor a függvény végez, takarít maga után, így a rekurzió **mélysége** számít. Mivel pedig eredetileg $d = \lceil \log N \rceil$ paraméterrel hívjuk a függvényt, így összesen $O(\log N)$ példány van a számítás tetszőleges pillanatában jelen. Egy-egy példány memóriaigénye pedig szintén $O(\log N)$, tehát összesen $\log N$ -szer van szükség $\log N$ memóriára, azaz a teljes program tárigénye $O(\log^2 N)$. Kicsit precízebben, ha az N csúcsú gráf egy csúcsának a tárolásához t bit kell, akkor $O(\log N \cdot t)$.

Ezzel bebizonyítottuk a következőt:

Állítás: Savitch tétele

ELÉRHETŐSÉG eldönthető $O(\log^2 n)$ tárban.

Látszólag ez nem sok eredmény, egy problémát megoldunk tárhatékonyan. Csakhogy az ELÉRHETŐSÉG azért érdekes probléma, mert **minden nemdeterminisztikus számítás felfogható egy ELÉRHETŐSÉG problémának!** A módszert „elérhetőségi módszer”-ként ismerjük, és a következőképp szól: egy programnak egy konfigurációját (emlékszünk: egy konfiguráció egy snapshot a számítás aktuális „állapotáról”, tehát RAM program esetén kiírjuk pl. egy asszociatív tömbbe a nemnulla értékű regisztereket index-érték párok listájában, és hogy épp hanyas számú sort készülünk végrehajtani) felfogjuk egy gráf csúcsának, és két konfiguráció,

C_1 és C_2 közt akkor megy él, ha C_1 -ből a nemdeterminisztikus programunk C_2 -be tud lépni. Tehát még egyszer: a csúcsok a konfigurációk, az élek a (nemdeterminisztikus) program egy-egy lehetséges lépésének felelnek meg.

Mivel több elfogadó konfiguráció is lehet, így annyit módosítunk a programunk gráfján, hogy ha accept soron áll, akkor onnan egy további lépésben egy új, elfogadó konfigurációba lépünk, jelölje ezt mondjuk t , a kezdeti konfigurációt (amikor minden munkaregiszter tartalma 0, és az első sor jön) pedig s . Ekkor a nemdeterminisztikus programunk pontosan akkor fogadja el az inputot, ha s -ből elérhető t ! Mármint ha a szóban forgó nemdeterminisztikus program mondjuk $f(n)$ tákorlátos, az pont azt jelenti, hogy egy konfigurációt $O(f(n))$ biten le tudunk írni, azaz ennyi bit kell egy csúcs tárolásához. Továbbá, ennyi bitet $2^{O(f(n))}$ -féleképp lehet teleírni, tehát a konfigurációs gráfnak (legfeljebb) ennyi csúcsa van! Ennek a logaritmusa pedig $O(f(n))$. Tehát ha ezen a gráfon futtatjuk a (determinisztikus!) Savitch algoritmust, azt kapjuk, hogy az eredetileg $f(n)$ tárigényű nemdeterminisztikus programunkat tudjuk „szimulálni” ezzel az összesen $O(f^2(n))$ tárigényű determinisztikus algoritmussal, vagyis

Állítás

az $f(n)$ tárban nemdeterminisztikusan eldönthető problémák mind eldönthetők determinisztikusan, $(f(n))^2$ tárban is.

Ha használjuk a következő jelöléseket:

- $\text{SPACE}(f(n))$: az $O(f(n))$ tárban eldönthető problémák osztálya;
- $\text{NSPACE}(f(n))$: az $O(f(n))$ tárban nemdeterminisztikusan eldönthető problémák osztálya;
- $\text{TIME}(f(n))$: az $O(f(n))$ időben eldönthető problémák osztálya;
- $\text{NTIME}(f(n))$: az $O(f(n))$ időben nemdeterminisztikusan eldönthető problémák osztálya;

akkor ugyanez rövidebben:

Állítás

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

Mivel pedig polinom négyzete polinom, így ez azt is jelenti, hogy ami nemdeterminisztikusan polinom tárban eldönthető, az determinisztikusan is, azaz:

Állítás

$$\text{PSPACE} = \text{NPSPACE}.$$

Tárra nézve, mivel a tár újra és újra felhasználható, a nemdeterminizmus kevesebb bónuszt ad; időre nézve sokkal rosszabb a helyzet, ott általában egy $f(n)$ időigényű nemdeterminisztikus algoritmust $2^{O(f(n))}$ -nél gyorsabban nem tudjuk, hogy lehet-e szimulálni vagy sem.

17. A lineáris tárigény

Ha időbonyolultságról van szó, akkor a (determinisztikus) lineáris időben eldöntés egy könnyű problémára utal. Egész más a helyzet a tárigénnyel: a különbség az, hogy real-life környezetben is újra és újra felhasználhatjuk számolásra ugyanazt a tárterületet (míg az idő ha már eltelt, azt nem). Ha meg pl. van egy k -bites memóriánk, azt 2^k -féleképpen lehet teleírni és ezt nem is feltétlenül nehéz: egyszerűen kezelhetjük ezt a számot egy k -bites számlálóként és növelhetjük egyesével nulláról. Persze ha egy determinisztikus számításról beszélünk, ami nem esik végtelen ciklusban, akkor ez egy felső korlát is már a lépésszáma: ha a programunk a futás közben kétszer ugyanabba a konfigurációba kerül (ugyanaz a memóriatartalom is és a végrehajtott utasítás sorszáma is), és determinisztikus, akkor végtelen ciklusba esik. Ezért ha a programunk S soros és $O(f(n))$ tárigényű, akkor $S \cdot 2^{O(f(n))} = S \cdot c^{f(n)} = O(c^{f(n)})$ időigényű is valamilyen c konstansra²⁸

Determinisztikusan a lineáris tár már eszerint jelenthet olyan algoritmust is, mely c^n ideig fut és tényleg sok mindenre elég annyi tár:

- A SATot el tudjuk dönteni úgy, hogy egy N -bites tárterületen értékadásokat tárolunk, egy bináris számsorozatként, ennek megfelelően kiértékeljük a formulát, ha igaz lett, akkor elfogadjuk az inputot, ha hamis, akkor generáljuk (pl. binárisan növelve az aktuális értékadást mint egy N -bites számot) a következő értékadást; ha elfogytak az értékadások és mind hamis lett, akkor elutasítjuk az inputot. Ez kb. ugyanaz, mintha az igazságtáblát írnánk fel, de egyszerre csak egy sorát és mindig újrahasználnánk azt a tárterületet, ahova az aktuális sorunkat írjuk. Erre elég lineáris tár. (Itt most N a változók száma, az input mérete persze $n \geq N$, ezért az N -bites számláló $O(n)$ biten persze elfér.)
- A HAMILTON-ÚT is eldönthető lineáris tárban: kiindulunk a csúcsok egy lehetséges permutációjából, pl az $1, 2, 3, \dots, N$ -ből, és ciklusban a következőt tesszük: megnézzük, hogy az aktuális permutációnk út-e a gráfban (ehhez elég $O(N)$ idő, csak ellenőrizni kell, hogy a permutációban szomszédos csúcsok közt az input gráfban van-e él mindenhol). Ha igen, akkor elfogadjuk az inputot, ha meg nem, akkor generáljuk a következő permutációt²⁹ és folytatjuk a ciklust. Ha eljutottunk az utolsó permutációig és az se jó, akkor elutasítjuk az inputot. Még azt érdemes talán fejben átgondolni (hogy szokjuk a tárigényben való gondolkodást), hogy a permutáció tárolása hány biten is történik, de mivel van N darab számunk, mindegyik 1 és N közti, így a permutáció eltárolása összesen $O(N \cdot \log N)$ tárat igényel, ha pedig a gráf mondjuk szomszédsági mátrixként jön be, akkor annak mérete $n = \Theta(N^2)$, tehát a tárigény tényleg belefér a lineárisba; ha éllistával, akkor szintén, hiszen tudjuk úgy módosítani az algoritmust, hogy csak akkor kezdjen ebbe a permutációgyártásba, ha minden csúcs foka legalább 1 (ha vannak izolált csúcsok a gráfban, akkor persze nincs benne Hamilton-út), ekkor meg a gráfban $\Omega(N)$ él kell legyen, aminek a végpontjai megadása darabonként $\log N$ tárat igényel, így az input teljes mérete $n = \Omega(N \cdot \log N)$, tehát ehhez az n -hez képest lineáris a tárigényünk.
- A 3-SZÍNEZÉS is eldönthető lineáris tárban: egy $O(N)$ méretű tárterületen minden csúcs-hoz generálunk egy színt (tkp mindenkihez a $0, 1, 2$ „számjegyek” valamelyikét rendelve –

²⁸A kettesből úgy lesz c , hogy ha pl. az az $O(f(n))$ a kitevőben pl. $k \cdot f(n)$, akkor $S \cdot 2^{k \cdot f(n)} = S \cdot (2^k)^{f(n)}$ hatványozás azonosság, és az a 2^k jó lesz c konstansnak.

²⁹Ez nem annyira straightforward, mint a bináris növelés, de itt hozzá egy sample code; ez az implementáció nem használ plusz tárterületet, helyben gyártja a rákövetkezőt. Persze egy eggyel kevésbé igényes megoldás az is, hogy mint n -es számrendszerbeli számot kezeljük a sorozatot és úgy növeljük egyesével, mindig ellenőrizve azt is a Hamilton-út tesztelésnél, hogy mindenki pontosan egyszer szerepel-e a sorozatban.

ez meg felfogható egy N -jegyű hármas számrendszerbeli számnak akár), majd ciklusban megnézzük, hogy ez így egy helyes színezés-e, ha igen, elfogadjuk az inputot, ha nem, akkor generáljuk a következő számsort „ternáris növeléssel” akár, és folytatjuk a ciklust; ha kigeneráltuk az összeset és egyik se volt jó, akkor elutasítjuk az inputot. Ez is persze lineáris tár lesz.

A fentiekből az a benyomás alakulhat ki esetleg az olvasóban, hogy minden **NP**-beli probléma eldönthető determinisztikus $O(n)$ tárban, vagy fordítva – nos ezt nem tudjuk, az egyetlen, amit biztosan tudunk, az az, hogy $\text{SPACE}(n) \neq \text{NP}$ ³⁰.

Viszont ha visszaemlékszünk az **NP** osztály egyik karakterizációjára: egy probléma pontosan akkor van **NP**-ben, ha van hozzá olyan „tanúsítványrendszer”, melyre az x inputhoz pontosan akkor létezik egy y „tanú”, ha x egy „igen” példány, ennek a tanúnak x -hez képest polinom méretűnek kell lennie, és az (x, y) párról (az \bar{o} együttes hosszukban) polinomidőben eldönthető kell legyen, hogy az y az x -nek egy tanúja-e vagy sem. A fentebbi három algoritmusban pontosan azt csináltuk, hogy egy tárterületre generáltuk sorban a lehetséges tanúkat, mellette egy másik területen pedig végigellenőriztük, hogy tanú-e. Intuitíve nagyjából világos, hogy ehhez viszont polinom tár garantáltan elég lesz (azon egy keveset még kell töprengeni, hogy a tanú ellenőrzése miért igaz, hogy nem használhat el túl nagy területet, később be is látjuk majd, hogy polinomidő alatt csak polinom tárat használhatunk fel), és ezért $\text{NP} \subseteq \text{PSPACE}$ igaz lesz. Később ennél általánosabban is be fogjuk ezt látni.

18. Az NL osztály

A tárigénynél pontosan azért definiáltuk külön az „offline” vagy „lyukszalagos” esetet, mert az előző fejezet miatt nagyjából érthető okból szeretnénk, ha lenne értelme szublineáris tárigénynek is, de ha az n -bites inputot magát számolásra is használjuk, akkor meg illene be is számolni a tárigénybe. Ezért szól úgy a tárigény definíciója, hogy ha az algoritmusunk az inputját csak olvassa, az outputot pedig „csak írja”, azaz stream módban kiírja a kimenetnek előbb az első, majd a második stb. számját, és azokat nem olvassa vissza, akkor az input és output regisztereket nem kell beleszámolni a tárigénybe, csak a working regisztereket.

A lineárisnál kisebb tárigények közül a **logaritmikussal** fogunk foglalkozni: $\text{L} = \text{SPACE}(\log n)$ és $\text{NL} = \text{NSPACE}(\log n)$ a determinisztikusan ill. nemdeterminisztikusan logaritmusos tárkorláton belül eldönthető problémák osztálya lesz.

Mit is jelenthet egy logaritmusos tárigény? Ahogy a Savitch-tétel bizonyításában is láttuk, az inputot csak olvashatjuk (különben mindenképp legalább lineáris lesz a korlátunk), és nem vehetünk fel pl. egy $\Theta(n)$ méretű bináris tömböt (mint ahogy tettük volna pl. a mélységi vagy szélességi keresésnél, vagy a SAT, 3SZÍNEZÉS eldöntésére az előző fejezet lineáris tárigényű algoritmusai is kiesnek). Amit biztosan tudunk használni, olyan változók, melybe 0 és n közti számokat írunk a futás során végig, hiszen ezeket $\log n$ tárban tudjuk tárolni. Ha nem n -ig, hanem valami fix fokszerű polinomig szeretnénk elszámolni, az is rendben van, hiszen pl ha egy változóba garantáltan 0 és n^3 közti értékek kerülnek, akkor a legnagyobb is elfér $\log n^3 = 3 \cdot \log n$ biten, tehát az $O(\log n)$ tárkorlátba beférünk. Speciálisan ha az inputnak valamelyik elemére rá akarunk „mutatni” egy pointerrel, az is felfogható úgy, mint egy számláló, mely 0 és n közti értéket tárol: megmondja, hogy az input hanyadik bitjéről beszélünk, vagy hanyadik input regiszter tartalmáról, ezek is elférnek $\log n$ bitben.

³⁰Ezt az ún. tárhierarchia tétellel és a QSAT probléma **PSPACE**-teljességével tudjuk belátni majd később: ezekből kijön, hogy $\text{SPACE}(n)$ nem zárt a polinomidejű visszavezetésre, **NP** meg persze az, ezért a két osztály nem lehet egyenlő.

Összességében tehát: ha egy algoritmusban a program futása során csak összesen konstans korlátos darab változót használunk, amik mind vagy pointerek az inputban valahova, vagy egy polinomig értéket tárolni képes számlálók, akkor egy logtáras algoritmussal van dolgunk. A Savitch-tétel bizonyításában szereplő algoritmus azért nem ilyen, mert a rekurzió miatt ugyan egy-egy példányban csak $O(\log n)$ tárat használunk, mert csak a gráf csúcsaira „mutatunk pointerekkel” és egy d intünk is van, mely csak 0 és $\log n$ közti értékeket vehet fel, ebből jó, de ilyen példányból a rekurzió során egyszerre akár $\log n$ is lehet a memóriában, és ezért összesen a felhasznált logtáras változókból nem csak konstans sok lesz, hanem akár $\Omega(\log n)$ darab, ami már túlló a $\log n$ tárkorláton.

Viszont ha **nemdeterminizmust** is használhatunk, akkor már elég a logtár az ELÉRHETŐSÉG eldöntéséhez:

```
boolean ndElérhetőség( $G, s, t$ ) { //input egy  $N$ -csúcsú irányított gráf és két csúcsa
     $u := s$ ; //ebben a csúcsban vagyunk most
     $steps := 0$ ; //ennyi lépést tettünk meg
    while(  $steps < N$  ) {
        if(  $u == t$  ) return true; //ha odaértünk, akkor van út
         $v := \text{ND}(1 \dots N)$ ; //generálunk egy számot  $1..N$  közt: egy csúcsot
        if(  $G[u][v]$  ) { //ha ide átléphetünk, akkor átlépünk
             $u := v$ ;
             $steps := steps + 1$ ;
        } else return false; //ha nem olyan csúcsot generálunk, ahova át tudunk lépni
    }
    return false; //ha  $N$  lépésben nem értünk oda, akkor ez a szál nem talált oda
}
```

Itt ND az a nemdeterminisztikus metódusunk, mellyel generálni tudunk egy számot 1 és N közt, kiírva nemdeterminisztikusan az N letárolásához szükséges $\log N$ darab bitet nemdeterminisztikusan; ha a generált szám nagyobb lett, mint N (pl. ha $N = 10$, akkor 4-bites számokat kell generáljunk, és ha épp az 1100-t generáljuk, ami 12 (attól függően, hogy kis vagy nagy endiánban tároljuk a számokat), ilyenkor ez több lesz, mint N), akkor mondjuk azonnal visszatérünk hamissal.

Ez pl. láthatóan egy olyan kód, mely nem írja a G, s, t értékeket, és mivel eldöntő algoritmus logikai kimenettel, az output regisztereket nem is használja, tehát az inputot nem kell beszámoljuk a tárigénybe; a lokális változók u, v és $steps$, mindegyik 0 és N közti értékeket vehet fel (a v generáláskor esetleg túlmegy N -en, mint ahogy fentebb is 12-t generáltunk, de a $\log N$ tárból nem megy ki ügysem), nincs benne rekurzív függvényhívás, így ez egy logtáras algoritmus. Nemdeterminisztikus, hiszen ott van benne az ND függvény is.

Maga az algoritmus lényegében azt csinálja, hogy egyszerre egy csúcsot tart csak a memóriában, az u -t, ahol éppen van, ezt az s csúcsra inicializálja és egy legfeljebb N lépésen át tartó „nemdeterminisztikus bolyongással” keresi t -t: minden iterációban nemdeterminisztikusan ki-generálja a következő csúcsot, ahova u -ról léphet (ha nem sikerült szomszédot generáljon, akkor azon a szálon feladja a számolást), és átlép oda, ezt addig ismétli, amíg oda nem ér t -hez (ekkor nyilván van út s -ből t -be, hiszen egy $s \rightsquigarrow t$ sétán közlekedett végig), vagy le nem lépett N -t, amikor is kilövi ezt a szálát.

Az világos, hogy ha az algoritmus egy szálon igazzal jön vissza, akkor s -ből t tényleg el is érhető; ha pedig s -ből t elérhető, akkor a legrövidebb oda vezető út kevesebb, mint N hosszú, így lesz egy olyan szál, mely pont ezt az utat generálja le és a nemdeterminisztikus program végeredményben tényleg elfogadja az input gráfot – vagyis ez a kód eldönti az ELÉRHETŐSÉG

problémát, nemdeterminisztikus logtárban, tehát:

Állítás

ELÉRHETŐSÉG \in NL.

19. Az Immerman-Szelepcsényi tétel

Ebben a részben megpróbálunk **komplementálni** nemdeterminisztikus algoritmust, a **tár-igény** lényegi romlása nélkül. Azt már láttuk korábban, hogy míg determinisztikus algoritmusnál elég kicserélni az ACCEPT sorokat REJECT sorokra (és így kapjuk, hogy $\text{TIME}(f(n)) = \text{coTIME}(f(n))$ és $\text{SPACE}(f(n)) = \text{coSPACE}(f(n))$, hiszen ezzel a probléma komplementerét végeredményben ugyanannyi tárban és időben döntjük el, mint eredetileg, $\text{co}\mathcal{C} \subseteq \mathcal{C}$ -ből pedig a co monotonitása miatt következik, hogy ekkor $\mathcal{C} = \text{coco}\mathcal{C} \subseteq \text{co}\mathcal{C}$ is igaz, tehát egyenlőek), de nemdeterminisztikus algoritmusoknál ez nem ilyen egyszerű: ha összesen ezt tesszük egy nemdeterminisztikus algoritmussal, akkor ha egy inputra vegyesen voltak igen/nem szálak, akkor továbbra is vegyesen lesznek (csak most már nem/igen szálak lesznek), és így ezt az inputot az eredeti és a módosított algoritmus is ugyanúgy elfogadja, tehát így általában nem tudunk komplementálni³¹.

Először is, megint csak az elérhetőségi módszert akarjuk alkalmazni: ha megoldjuk azt a kérdést, hogy „igaz-e, hogy az input gráfban az s csúcsból **nem** érhető el a t csúcs”, nemdeterminisztikusan logaritmikus tárban, akkor ez jó lesz: ezt az algoritmust fogjuk futtatni a konfigurációs gráfon és kész is lesz, hiszen ahogy a Savitch-tételnél is láttuk:

- egy $O(f(n))$ tárigényű RAM-programnak legfeljebb $c \cdot 2^{O(f(n))}$ konfigurációja lehet egy konstans c -re, ami persze $2^{O(f(n)) + \log c}$ (itt c a programsorok számából jön, azt is le kell tároljuk a regiszterek tartalmán kívül, hogy melyik sorban vagyunk éppen)
- akkor ha s a kezdő konfiguráció, t pedig az egyetlen elfogadó konfiguráció (korábban már megegyeztünk abban, hogy feltehető, hogy a program elfogadás előtt visszaállítja az összes regisztere értékét 0-ra, majd ugrik az egyetlen ACCEPT sorra, ezért feltehető, hogy a konfigurációs gráfban egyetlen elfogadó konfiguráció van), úgy ebben a gráfban a „igaz-e, hogy nincs út s -ből t -be” kérdés eldönthető lesz nemdeterminisztikus logtárban, azaz $\log(2^{O(f(n)) + \log c}) = O(f(n)) + \log c$ tárban
- ami, mivel $f(n)$ a $\log c$ **konstansnál** ordóban biztos nagyobb-egyenlő, azt jelenti, hogy nemdeterminisztikus $O(f(n))$ tárban el tudjuk dönteni az eredeti probléma komplementerét is.

Szóval most a célunk ez lesz: egy input gráf s és t csúcsáról nemdeterminisztikusan $O(\log n)$ tárban akarjuk eldönteni, hogy igaz-e, hogy s -ből **nem** érhető el t . Ha ez sikerül, akkor abból emiatt a fenti miatt kijön, hogy minden $f(n)$ függvényre $\text{NSPACE}(f(n)) = \text{coNSPACE}(f(n))$ (itt azért az fontos, hogy a NSPACE osztály is, mint ahogy a többi is, eleve „ordóban” definiálja a megengedett erőforrásigényt).

³¹Ha meg nincs olyan input, amire vegyes válaszok vannak, akkor minek is nemdeterminisztikus az egész? ha mondjuk mindig az első lehetőséget választjuk, akkor determinisztikusan jutnánk el ugyanoda. Tehát a „vegyes output” lehetősége, majd ebből az eredmények vagyolásának visszaadása egy fontos komponens ahhoz, hogy legyen „értelme” a nemdeterminizmusnak.

Ehhez egy **nemdeterminisztikus függvénykiszámító** algoritmust fogunk fejleszteni, nevezük mondjuk $\text{ELÉRHETŐKSZÁMA}(G, s)$ -nek, ami inputként megkapja a G irányított gráfot és annak az s csúcsát, nemdeterminisztikus és minden szálon vagy egy **számot** ad vissza, vagy azt, hogy „passz”, mégpedig:

- van minden G, s inputra olyan szál, amelyik egy számot ad vissza (azaz olyan nem lehet, hogy minden szál passzoljon), és
- ha számot ad vissza, az garantáltan a G -ben s -ből elérhető csúcsok **száma** lesz.

Hogy ez mire jó? Nézzük csak ezt az algoritmust:

```
main(G, s, t) {
  count := elérhetőKSzáma(s);
  if( count == „Passz” ) return „Passz”;
  for( u := 1..N ) {
    if( ndElérhető( s, u, n ) ) {
      if( u == t ) return false; //ha elértük t-t, nyilván elérhető  $\Rightarrow$  false
      count := count - 1;
    }
  }
  if( count == 0 ) return true;
  return „Passz”;
}
```

Itt az $\text{NDELÉRHETŐ}(s, t, d)$ egy nemdeterminisztikus eljárás, ami szokásos true/false értékekkel tér vissza, és pontosan akkor **térhet** vissza trueval (legalább egy szálon), ha s -ből u elérhető legfeljebb d lépésben:

```
boolean ndElérhető(s, t, d) { //input két csúcs és egy max lépésszám
  u := s; //ebben a csúcsban vagyunk most
  steps := 0; //ennyi lépést tettünk meg
  while( steps  $\leq$  d ) {
    if( u == t ) return true; //ha odaértünk, akkor van út
    v := Nd(1..N); //generálunk egy számot 1..N közt: egy csúcsot
    if( G[u][v] ) { //ha ide átléphetünk, akkor átlépünk
      u := v;
      steps := steps + 1;
    } else return false; //ha nem olyan csúcsot generálunk, ahova át tudunk lépni
  }
  return false; //ha d lépésben nem értünk oda, akkor ez a szál nem talált oda
}
```

Alapvetően ez ugyanaz a kód, mint amit már láttunk az előző fejezetben, csak plusz egy d lépéskorláttal: először is inicializálja a current változót s -re, majd egy ciklusban ellenőrzi, hogy „odaért-e” már t -re, ha igen, akkor leállhat a keresés és mehet a **true**, hiszen ekkor s -ből biztos elérhető t . Ha még nem ért oda, akkor nemdeterminisztikusan generálunk egy újabb csúcsot a gráfban, majd megnézzük (akár a szomszédsági mátrixszal, akár mésképp), hogy átléphetünk-e a current csúcsból az újonnan generált next csúcsba. Ha igen, mert van él, akkor ezt meg is tesszük; ha nem tudunk átlépni, akkor ezen a szálon **false** jön vissza. Tehát az „élő” szálakon tkp nemdeterminisztikusan s -ből indulva lépkedünk; ha s -ből t elérhető, akkor lesz olyan szál,

ami legfeljebb d lépésben átér s -ből t -be (pl az, amelyik a legrövidebb úton halad s -ből t -be). Ezért az egész lépkedést egy ciklusba tesszük, amit d -ig számol; azokat a szálakat, melyek d lépésen se érték el s -ből t -t, kilőjük **false**-al.

Tehát, az $\text{NDELÉRHETŐ}(s, u, d)$ egy „szokványos” nemdeterminisztikus algoritmus azzal, hogy

- ha s -ből u nem érhető el d lépésben, akkor minden szálon **false** az eredmény;
- ha elérhető d lépésben, akkor van legalább egy szál, ahol **true** az eredmény.

Ez alapján nézzük meg, hogy mit adhat is vissza a $\text{MAIN}(G, s, t)$ függvényünk, ha sikerül úgy megírjuk az $\text{ELÉRHETŐKSZÁMA}(s)$ függvényt, ahogy szeretnénk: vagy mondja meg, hogy mennyi csúcs érhető el s -ből (s -sel együtt), vagy passzoljon, de legalább egy szálon mondja meg a számértéket.

- Ahhoz, hogy **false** jöjjön vissza, az kell, hogy a belső ciklusban $u == t$ igaz legyen. Azaz, hogy az $\text{NDELÉRHETŐ}(s, t, n)$ függvény igazat adjon, mert az u változó végigmegy az összes csúcson, tehát lesz olyan, hogy $u == t$. Mármint ha s -ből t elérhető, olyankor egy teljesen valid forgatókönyv, hogy
 - az $\text{ELÉRHETŐKSZÁMA}(s)$ függvény visszaadja azt, hogy hány csúcs érhető el s -ből,
 - akkor az ezutáni `if` nem aktiválódik, mert egy szám jött vissza, megyünk tovább,
 - végigscanneljük az összes csúcsot, köztük t -t is és amelyekre a belső hívott NDELÉRHETŐ függvény igazat ad, megnézzük, hogy t -e, ha igen, visszaadunk hamist, különben simán csökkentünk egy számlálót
 - és konkrétan amikor ezzel a ciklussal a t -hez érünk, olyankor az NDELÉRHETŐ függvény adhat igazat is egy szálon, és mi meg visszajövünk azzal, hogy **false**.
- Ahhoz mi kell, hogy **true** jöhessen vissza?
 - **true**-val csak akkor jövünk vissza, ha kiszabadulunk az u -s forciklusból és a `count` értéke ekkor 0 lesz.
 - A `count` persze egy számra kelljen beálljon az elején, hiszen ha „passz” lenne, már a ciklusig se jutnánk el.
 - Tehát, amikor nekiálljuk darálni a ciklust, akkor a `count` változóban pontosan az s -ből elérhető csúcsok száma lesz (mert így akarjuk megírni azt az ELÉRHETŐKSZÁMA függvényt, amit még nem írtunk meg, de így szeretnénk, hogy működjön).
 - Amikor megy a ciklus, akkor ha s -ből nem érhető el u , úgy biztos, hogy a NDELÉRHETŐ függvény is hamist ad, ilyenkor tehát nem fogjuk csökkenteni a `count` változó értékét.
 - Azaz, a `count` csak akkor fog csökkenni, ha u elérhető csúcs és ráadásul az NDELÉRHETŐ (nemdeterminisztikus) függvény el is érte. (A $d == n$ paraméter biztos jó lesz, hiszen ha valaki elérhető egy n csúcsú gráfban s -ből, akkor n lépésben is elérhető.)
 - Mivel `count`-ban alaphól az összes elérhető csúcs száma volt, ezért csak akkor tudjuk lehozni 0-ra, ha az $\text{NDELÉRHETŐ}(s, u, n)$ hívásokkor az **összes** olyan u -ra, ami elérhető s -ből, tényleg **true**-t kapunk vissza. Ettől lesz `count` értéke nulla a ciklus végén.

- De ha t is elérhető lenne, akkor már ahogy bemegyünk az ifbe, azonnal hamist adtunk volna vissza és nem jutunk el a return trúig.
- Tehát: pontosan akkor adhatunk vissza true-t a mainból, ha az elérhetők száma függvényünk visszaadja azt, hogy G -ben s -ből hány csúcs érhető el (ez lehetséges lesz), végigpörgetjük az u -ra a ciklust, mindent elérünk akit csak el lehet és nincs köztük a t .
- Azaz: akkor tudunk visszaadni true-t, ha s -ből nem érhető el. (és ekkor van is a fentiek szerint ilyen futás, ami igazt ad vissza.)

Tehát: ha sikerül megírjuk az ELÉRHETŐKSZÁMA függvényt úgy, ahogy ígértük, akkor a MAIN függvényünk pontosan akkor tud true-t visszaadni, ha s -ből t **nem** érhető el.

Tárigényre akarunk optimalizálni, de eddig nincs gond: az NDELÉRHETŐ függvény nem ad új értéket s -nek, sem u -nak, sem d -nek, így őket nem kell beszámoljuk; ha s is és u is csúcsok, d pedig egy 0 és n közti lépésszámláló (úgy hívjuk őket a MAINból, tehát akkor biztos azok), akkor őket le tudjuk írni $\log n$ biten; a steps változó nemnegatív egész, n -ről indul, nőni nem tud, csak csökkenni, tehát valahol 0 és n közt lesz mindig az értéke, ez elfér $\log n$ biten; a next változó egy helyen kap értéket, ott egy 1 és n közti számot (egy csúcsát a G gráfnak, ez elfér $\log n$ biten); a current változó meg vagy az s -t, vagy a nextet kapja meg értékül, amik elférnek $\log n$ biten, tehát current is.

Mivel a függvény nem rekurzív, és nem bántja az inputot, az NDELÉRHETŐ függvény ezek szerint nemdeterminisztikus logtáras, hiszen minden lokális változója az (az eredeti input gráfhoz képest persze).

A MAIN függvény pedig használ egy count változót, amibe elviekben az s -ből elérhető csúcsok száma kerül, az 1 és n közt van (s -ből s mindenképp elérhető, ezért legalább 1 lesz), az elfér $\log n$ biten, még úgy is, hogy plusz egy opció a „passz” (amit lehet akár 0-val reprezentálni). Aztán van egy u változó, ami 1 és n közt vesz fel egész értékeket, ez is elfér $\log n$ biten, más változó meg nincs, tehát a függvény lokális változói $O(\log n)$ tárat igényelnek, belül az NDELÉRHETŐ függvényhívást az előbb számoltuk ki, hogy az is csak $\log n$ tárat igényel, az ELÉRHETŐKSZÁMA függvényt pedig szintén logtáras (nemdet) függvényként akarjuk implementálni, ezért a MAIN függvény egyrészt eldönti az ELÉRHETETLENSÉG problémát, másrészt logtáras lesz, ha az elérhető csúcsok számát tényleg nemdet logtárban eldönteni.

Na akkor próbáljuk meg implementálni az ELÉRHETŐKSZÁMA(s) függvényt. Vagy vissza kéne adja az s -ből G -ben elérhető csúcsok számát, vagy azt, hogy „passz”, legyen olyan szál, amikor a pontos értéket adja vissza, mindezt logtárban.

Próbáljunk meg írni egy ELÉRHETŐKSZÁMA(s) függvényt, ami ezt egy forciklussal oldja meg, ami $d = 0$ -tól megy n -ig, és kiszámolgatja a ciklusmagban, hogy legfeljebb d lépésben hány csúcs érhető el s -ből! Ha meg közben hibára fut egy szálon, mondja azt ez is, hogy „passz”.

```

elérhetőkszáma( s ) { //input a gráf s csúcsa
  d := 0; //alap eset: 0 lépésben hány csúcs érhető el?
  prev := 1; //hát csak 1: maga az s csúcs
  for( d := 1...N ) { //belépéskor a max d-1 lépésben elérhetők száma prev-ben van
    count := 0; //ebbe számoljuk, hogy mennyit sikerült elérni d lépésben
    for( u := 1...N ) { //megpróbáljuk elérni u-t d lépésben
      check := 0; //ellenőrzi, hogy minden v-t elérünk-e lentebb, akit csak lehet
      for( v := 1...N ) { //el lehet-e érni v-t d-1, aztán onnan u-t 1-et lépve?
        if( ndElérhető( s,v,d-1 ) {
          if( G[v][u] or v == u ) { //u elérhető, számoljuk meg és u-ciklus goes brr

```

```

        count := count + 1;
        continue u;
    }
    check := check + 1; //megszámoljuk v-t, mert d-1 lépésben elérhető volt
}
}
if( check ≠ prev ) return „Passz”; //valakit nem találtunk meg d-1 lépésben
}
prev := count; //countba gyűjtöttük, mennyit sikerült d lépésben elérni
}
return prev;
}

```

Az algoritmus kintről nézve mit végez: visz egy d változót, van rá egy forciklus. Egyrészt, d értéke 0 és n közt fog mozogni, tehát $\log n$ biten elfér. Másrészt, a ciklusinvariánsunk az lesz, hogy a ciklusból kijövéskor igaz lesz, hogy a $prev$ változóban pontosan a d lépésben elérhető csúcsok száma lesz. (Mikor pedig visszalépünk a ciklusmagba, akkor növeljük d -t, ezért a ciklus magjában ez a feltétel azt jelenti, hogy $prev$ -ben pontosan a $d - 1$ lépésben elérhető csúcsok száma lesz.)

Ez a kód elején, a ciklusba való első belépéskor igaz: legfeljebb 0 lépésben 1 csúcs érhető el s -ből, maga s . Ha a ciklusinvariánst fenntartjuk, akkor a végén a RETURN $prev$ tényleg jó lesz, mert akkor az a legfeljebb n lépésben elérhető csúcsok számát fogja majd tárolni azon a ponton.

Mi történik a d ciklusban? 0-ra állítunk egy $count$ változót, ebbe fogjuk összeszámolni, hogy hány csúcs érhető el legfeljebb d lépésben, és közben intenzíven használni fogjuk azt a tudásunkat, hogy legfeljebb $d - 1$ lépésben $prev$ darab csúcs érhető el s -ből. Ha ez sikerül, akkor fennmarad a ciklusinvariáns, mert a $prev$ változónak csak egyszer, a ciklus végén adunk értéket, magát $count$ -ot, akibe addigra össze akarjuk számolni a d lépésben elérhető csúcsok számát.

Az összeszámlálás úgy megy, hogy viszünk egy $u = 1 \dots n$ ciklust, és minden u csúcsot megpróbálunk elérni d lépésben. Ha sikerül, növeljük a $count$ változót. (És ismét: ha közben detektáljuk, hogy elszámoltunk valamit, akkor kijövünk „passz”-szal). Ha ezt rendesen csináljuk meg, akkor az u ciklus végére a $count$ változóban tényleg a max d lépésben elérhető csúcsok száma lesz.

Lássuk, hogy akarjuk elérni az u csúcsot: keresünk egy olyan v csúcsot ciklusban, aki elérhető $d - 1$ lépésben, és ahonnan át lehet lépni u -ra (vagy aki maga u , hogy a „legfeljebb” opció biztos kijöjjön). Tehát végigtolunk egy ciklust $v = 1 \dots n$, és ha olyan v csúcsot találunk, akire az NDELRHETŐ függvény azt mondja, hogy elérhető legfeljebb $d - 1$ lépésben, akkor ennek megörülünk, megnöveljük az elért csúcsok $count$ számát és megyünk tovább a következő u -ra, hogy az akkor vajon elérhető-e.

Csak hogy, ha az NDELRHETŐ függvény azt mondja, hogy v elérhető, akkor biztosan igazat mond, de ha azt mondja, hogy nem elérhető, akkor attól még lehet, hogy valójában az, csak pont nem egy „accept” szálon futott, erről szól a nemdeterminizmus. Ezt a tévedést detektálni akarjuk és ha rájövünk, hogy tévedett az NDELRHETŐ függvény, mi meg észrevesszük, akkor „passz”-szal jövünk vissza.

A tévedést úgy detektáljuk, hogy minden egyes u csúcs keresénél amikor scanneljük a v -ket, közben számoljuk, a $check$ nevű változóba, hogy hány v csúcsot sikerült elérni legfeljebb $d - 1$ lépésben: ha v -re azt mondja az elérhető függvény, hogy igen az, akkor növeljük a $check$ változó

értékét. Namost ha lefut a v ciklus, az azt jelenti, hogy nem találtunk olyan v -t, aki elérhető max $d - 1$ lépésben s -ből, és akiből elérhető u legfeljebb egy lépésben. Ha itt ezen a ponton $\text{prev} = \text{check}$, az azt jelenti, hogy pontosan annyi csúcsra mondta az NDELÉRHETŐ függvény, hogy elérhetők s -ből max $d - 1$ lépésben (check), ahányan tényleg azok is (prev), és mivel az NDELÉRHETŐ függvény ha igent mond, akkor nem téved, ez tényleg azt jelenti, hogy nincs is olyan v csúcs, ami max $d - 1$ messze van s -től és ahonnan u max egy messze van, azaz azt, hogy u nem elérhető max d lépésben s -ből és minden rendben, hajthatjuk tovább az u ciklust, ez az u tényleg nem elérhető.

Ha viszont $\text{check} \neq \text{prev}$ (konkrétan kisebb lesz), az azt jelenti, hogy volt olyan v , akire az NDELÉRHETŐ függvény azt jelentette le, hogy nem érhető el s -ből max $d - 1$ lépésben, pedig az volt; ezt a hibát nem próbáljuk meg kijavítani, hanem kijövünk az egész függvényből azzal, hogy „passz”. Sok szál fog emiatt „passz”-t mondani, de lesz egy olyan szál, ahol az NDELÉRHETŐ függvény mind a kb akár n^2 hívásakor a helyes választ fogja adni, és ekkor a ciklus végén a count változóban tényleg azokat számoltuk meg, akik max d lépésben elérhetők s -ből. Odaadjuk prev -nek és hajtjuk tovább a d ciklust: ez a kód így a végén tényleg vagy a korrekt számot adja vissza, hogy hány csúcs érhető el s -ből (max n lépésben, azaz egyáltalán), vagy pedig azt, hogy „passz”, de van olyan futása mindig, ami a számmal jön vissza.

OK, tehát nemdeterminisztikusan (mert az NDELÉRHETŐ függvény az) ki tudjuk számolni, hogy hány csúcs érhető el G -ben s -ből. Továbbá, ebben a függvényben az inputot nem bántjuk, a lokális változók d , aki 0 és n közti értéket vesz fel, prev , check és count , akik csúcsokat számolnak, tehát 0 és n közti értéket vesznek fel, u és v , akik csúcsokat vesznek fel értékül, tehát 1 és n közti számok, más meg nincs. Tehát minden lokális változónk elfér log tárban, és az egyedüli meghívott függvényünk, az NDELÉRHETŐ is logtáras, ezeknek az összege is logtáras, tehát

Állítás: Immerman-Szelepcsényi tétel

Létezik olyan nemdeterminisztikus algoritmus, mely logtárban kiszámolja a G gráf s csúcsából elérhető csúcsok számát.

Itt a „nemdeterminisztikus kiszámítás” pont azt jelenti, amit csináltunk: minden egyes szál vagy visszaadja a korrekt eredményt, vagy kiáll hibára, és minden inputra létezik legalább egy olyan szál, mely a korrekt eredménnyel jön vissza.

Azt meg már láttuk, hogy ezt az algoritmust hogyan tudjuk felhasználni az elérhetetlenség megoldására, tehát:

Állítás

$$\text{ELÉRHETETLENSÉG} \in \text{NSPACE}(\log n).$$

Az elérhetőségi módszerrel pedig a fejezet elején kifejtett módon lehet megoldani tetszőleges nemdeterminisztikus RAM gép konfigurációs gráfján egy elérhetetlenség probléma megoldásával, hogy igaz-e, hogy az adott program az adott inputot nem fogadja el, emiatt pedig a következő igaz:

Állítás

$$\text{Tetszőleges } f(n) \text{ függvényre } \text{NSPACE}(f(n)) = \text{coNSPACE}(f(n)).$$

Azaz ha egy problémát el lehet dönteni valamennyi tárban nemdeterminisztikusan, akkor a komplementerét is (nagyságrendben) ugyanannyi tárral el lehet dönteni nemdeterminisztikusan. Ezt úgy is mondják, hogy **a nemdeterminisztikus tárbonyolultsági osztályok zártak a komplementerképzésre.**

20. Alapvető összefüggések bonyolultsági osztályok közt

Láttuk korábban, hogy $P \subseteq NP$ és vázlatosan láttuk azt is, hogy $NP \subseteq PSPACE$, vagyis tudtunk mondani különböző erőforrással és determinizmus móddal definiált osztályok közt tartalmazásokat. Ebben a fejezetben három ilyen fogunk látni, ennél általánosabban, tetszőleges $f(n)$ idő- és tárkorlátra. Itt is többször fogjuk használni az **elérhetőségi módszert**, amikor is a programot úgy „szimuláljuk”, hogy a konfigurációs gráfján oldunk meg egy elérhetőségi problémát.

Az első állításcsoport elég egyszerű:

Állítás

$$\begin{aligned} \text{TIME}(f(n)) &\subseteq \text{NTIME}(f(n)) \\ \text{SPACE}(f(n)) &\subseteq \text{NSPACE}(f(n)) \end{aligned}$$

Ez persze azért van, mert ha egy problémát $f(n)$ időben/tárban eldönt egy program, akkor ugyanaz a program felfogható úgy is, mint egy nemdeterminisztikus program, ami sose ágaztatja ketté a számítást; a kimenete persze ugyanaz lesz így is. Nyilván pl. emiatt $L \subseteq NL$ és $P \subseteq NP$.

A második összefüggés hasonlít az $NP \subseteq PSPACE$ állításunkra korábbról, de annál általánosabb:

Állítás

$$\text{NTIME}(f(n)) \subseteq \text{SPACE}(f^2(n))$$

Bizonyítás

Tehát a mondás az, hogy ha van egy N nemdeterminisztikus programunk, ami $f(n)$ **időkorlátos**, azt tudjuk szimulálni egy $f^2(n)$ **tárkorlátos**, determinisztikus programmal.

Először is „normalizáljuk” az N programot egy kicsit: tegyük fel róla az általánosság megszorítása nélkül, hogy mindig **pontosan két** választása van megállásig, ezeket jelölje 0 és 1: Ha pl. azért lenne 5 választása, mert épp a 8. utasítást kéne végrehajtása és abból van öt darab a kódban, ezt át tudjuk alakítani: átszámozzuk a 8. utasításokat öt új sorszámot kiosztva, a 8. utasításból egy ugrást készítünk egy még újabb helyre, ott nemdeterminisztikusan generálunk egy három bites számot (mert ennyi elég az 5 lehetőségre), ha 000 lett, akkor az első opcióra ugrunk, ha 001, akkor a másodikra, stb, ha 100, akkor az ötödikre, ha meg túl sok, akkor rejectelünk egyet. Tehát ezzel át tudjuk konvertálni olyanná a kódot, mely már minden lépésben legfeljebb kétfelé ágazik el, és nem lesz sokkal lassabb: egy eredetileg k -felé történő elágazás egy lépés helyett $\log k$ lépésben történik meg, de persze k a programtól függ, az inputtól nem, csak attól, hogy egy sorszám hány példányban van kiosztva, így $\log k$ is konstans, vagyis az így módosított program továbbra is $O(f(n))$ időkorlátos lesz. Ha csak egy lehetősége van, akkor azt felfoghatjuk úgy, hogy kettő, de mindkettő ugyanaz.

Ilyen módon most már ha van egy $f(n)$ hosszú bináris stringünk, az leír **egy** számítási szálát: az első bit mondja meg, hogy az első lépésben a két lehetőség közül melyiket válasszuk, a második bit azt, hogy a második lépésben melyiket az akkori két lehetőség közül, stb. Nagyon hasonlóan a „SAT megoldása lineáris tárban” történethez, itt is azt fogjuk csinálni, hogy beállítjuk (mondjuk) egy amúgy nem használt regiszterbe bináris számként a $00 \dots 0$ számot, mely $f(n)$ darab 0-ból áll, és azt az egy szálát szimuláljuk le, amit ez elkódol; ha ez acceptel a végén, akkor mi is, egyébként meg növeljük eggyel a szálát kódoló számlálót és újra kezdjük a szimulációt. Ha végigpróbáltuk az összes lehetőséget, vagyis kipróbáltuk már az $11 \dots 1$ szálát is, és még az se fogadta el az inputot, akkor pedig elutasítjuk az inputot is. Lényegében ezt csináltuk a három **NP**-teljes probléma lineáris tárban történő megoldásakor is.

Van pár probléma, amit meg kell oldjunk azért ezzel a megközelítéssel:

1. Nem biztos, hogy maga az $f(n)$ függvény olyan könnyedén kiszámítható (lehetséges, hogy $f(n)$ darab 0 kiszámításához több tár kéne, mint $f^2(n)$, vannak ilyen elborult függvények^a), ezért semmi nem garantálja, hogy tudjuk inicializálni a számlálónkat. Ehelyett egy **iteratívan mélyülő mélységi keresést** csinálunk: futtatunk egy k számlálót 1-től fölfelé egy while ciklusban, és minden egyes k értékre:

- kigenerálunk k darab 0-t, ezzel inicializáljuk a szál azonosítóját,
- leszimuláljuk a szálát, amit az azonosító megad, ennyi lépésig, a következők történhetnek:
 - ezen a k lépésen belül találunk egy acceptet, ekkor accepteljük mi is az inputot.
 - ezen a k lépésen belül rejectel a szál, lépünk tovább a következőre, növelve a számlálót.
 - ezen a k lépésen belül nem végez még a szál. Ezt megjegyezzük egy boolean változóba, melybe azt tároljuk el, hogy az aktuális k -ra volt-e befejezetlen szálunk. Ezt k növelésekor mindig beállítjuk hamisra, az ilyen esetekben pedig igazra tesszük.
- ha elértünk a k darab 1-esig, és nem volt accept, akkor két eset lehetséges:
 - volt olyan szál közben, ami nem fejeződött be. Ekkor növelük k -t eggyel és megyünk vissza a „generálunk k darab 0-t” lépésre.
 - nem volt ilyen szál közben. Ez azt jelenti, hogy ezen a k lépésen belül minden szál véget ért rejecttel – ekkor pedig mi is rejecteljük az inputot.

Nyilván ha k elér $f(n)$ -ig, akkor már biztos, hogy minden szál be fog fejeződni k lépésen belül, tehát magát a számlálót tényleg el fogjuk tárolni $f(n)$ biten, és nem kell tudnunk róla, hogy tulajdonképpen mennyi is a pontos értéke, így nem bajos az, ha esetleg amúgy maga az $f(n)$ nem lenne kiszámítható a megadott tárkorláton belül.

2. A szimuláció be kéne férjen $O(f^2(n))$ tárba. Ez alapvetően nem problémás: azt lehet igazolni indukcióval, hogy egy n -bites inputon elindítva egy RAM programot, a t . lépésben a regiszterekben tárolt számok (így persze a címük is, mert mikor megcímzünk egy regisztert, akkor vagy konstanst, vagy egy már regiszterben lévő számot használunk címezni) legfeljebb $n + c + t$ -bitesek lehetnek egy alkalmas c konstansra.

Ez a program elindításakor persze igaz, hiszen a 0. lépésben csak az input regiszterekben vannak nemnulla számok, amik legfeljebb n -bitesek. Egy utasítás pedig ha feltételes ugrás, vagy accept/reject, akkor nem változtat a regiszterek tartalmán; ha pedig értékadás, akkor egy regiszterbe kerülhet egy összeg vagy egy különbség, mégpedig két olyan értéké, ami mindkettő vagy egy regiszterben van benne, vagy konstans. Mármint ha a programban szereplő konstansok mind legfeljebb c -bitesek, akkor a legnagyobb szám, amit eredményként elérhetünk, két $(n + c + t)$ -bites szám összege lesz, ami egy legfeljebb $(n + c + t + 1)$ -bites szám lehet (itt fontos pl, hogy a szorzás nem elemi művelet, mert akkor az megduplázhathatna a használt tárat minden lépésben), és meg is vagyunk az indukcióval. Tehát $f(n)$ lépés után, még ha folyamatosan mindig újabb és újabb regiszterekbe is irkáljuk a számainkat, összesen legfeljebb $f(n)$ darab, egyenként max. $(n + c + f(n))$ -bites számunk lesz, olyan regiszterekben, melyeknek a címe is egyenként max. $(n + c + f(n))$ -bitesek. Mivel időkorlátról van szó, arról abban egyeztünk meg korábban, hogy $f(n) = \Omega(n)$, ezért $n + c + f(n) = O(f(n))$, így van $f(n)$ darab $O(f(n))$ -bites számunk, $O(f(n))$ -bites című regiszterekben, tehát egy regiszter leírásához elég $O(f(n)) + O(f(n)) = O(f(n))$ bit (cím+tartalom), ilyenből van $f(n)$, tehát összesen $O(f(n) \cdot f(n)) = O(f^2(n))$ tárterület elég a komplett szimulációhoz^b.

3. Újra kell tudnunk indítani a szimulációt. Ez megoldható, ha N offline, hiszen akkor az inputot nem bántjuk, ezért ekkor csak a munkaregiszttereket kell kinullázzuk, mielőtt újraindítjuk a szimulációt. Ha N nem offline, akkor pedig azzá tehető könnyen: első körben lemásoljuk az inputot új munkaregisztterekbe, és innentől őket kezeljük inputként. Ekkor persze az időlimit megnő a másoláshoz kellő $O(n)$ lépéssel, de ismét, mivel $f(n) = \Omega(n)$, ezért $n + f(n) = O(f(n))$, ez sem okoz gondot.

Más probléma nem merül fel ezzel a megközelítéssel, így tehát tényleg tudunk determinisztikusan $f^2(n)$ tárban szimulálni nemdeterminisztikus, $f(n)$ időkorlátos programt.

^aKésőbb a „nem ilyen elborult függvényeket” fogjuk majd „megengedett bonyolultsági függvény”nek nevezni.

^bEbből jön ki a négyzet a képletben. Ha Turing-gépünk lenne az architektúránk, ott ez a tétel úgy szólna, hogy $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n))$, mert Turing-gépen egy lépésben nem lehet ilyen mértékben növelni a „realisztikus” tárterületet.

Emiatt pl. most már formálisan is tudjuk, hogy minden **NP**-beli probléma, mondjuk n^k időkorlátos, szimulálható determinisztikusan $(n^k)^2 = n^{2k}$, azaz szintén polinom, tárban, vagyis **NP** \subseteq **PSPACE**.

Lássuk, hogy **nemdeterminisztikus tárat** ha szeretnénk **determinisztikus időben** tárolni, akkor ott milyen romlással számolhatunk: exponenciálissal

Állítás

$$\text{NSPACE}(f(n)) \subseteq \text{TIME}(k^{f(n)}).$$

Bizonyítás

Még mindig, egy ilyen képletben az n mindig az input méretét jelenti, k pedig egy tetszőleges konstans, tehát a $\text{TIME}(k^{f(n)})$ jelölés az $\bigcup_{k \geq 0} \text{TIME}(k^{f(n)})$ osztályt jelenti, benne van a $2^{f(n)}$ is, a $3^{f(n)}$ is, stb.

Itt megint az **elérhetőségi módszer**et használjuk. A nemdeterminisztikus gép, ha $f(n)$ tárkorlátos, akkor van egy $O(f(n))$ méretű munkaterülete, melyet $2^{O(f(n))}$ -féleképp írhat tele, továbbá van konstans sok, mondjuk K darab utasítás, melyekből áll a programja, tehát ha egy „snapshotot” készítünk róla működés közben, elmentjük a regiszterek tartalmát és hogy hol áll épp a vezérlés a programban, vagyis egy **konfigurációját**, az $K \cdot 2^{O(f(n))} = O(k^{f(n)})$ -féle lehet. Ha tehát legyártjuk a **konfigurációs gráfot**, melynek csúcsai a konfigurációk, és $C_1 \rightarrow C_2$ él akkor van benne, ha C_1 -ből C_2 -be egy lépésben átjuthat a gép, akkor a feladatunk azt eldönteni, hogy ebben a gráfban elérhető-e a kezdő konfigurációból egy elfogadó konfiguráció. Erre ha akár egy szélességi vagy mélységi keresést használunk, az a gráf méretével arányos időben fut le: tudjuk, hogy a csúcsok száma $O(k^{f(n)})$, azt is tudjuk, hogy minden csúcsból csak max konstans sok él fut ki (egy konfigurációnak még az is feltehető, hogy csak max két rákövetkezője van, mint ahogy az előző bizonyításban is volt), tehát az élek száma is $O(k^{f(n)})$, így ezen a gráfon egy szélességi keresés futtatása valóban végrehajtható $O(k^{f(n)})$ időben és készen is vagyunk.

Persze legyárthatjuk előre az egész konfigurációs gráfot és aztán futtathatunk rajta egy keresőalgoritmust, de eggyel szebb megoldás on-demand generálni az érintett csúcsokat: mikor a keresésben egy x csúcs (konfiguráció) szomszédait generáljuk le és szűrjük be a szürke listába, ha még nem láttuk, elég csak akkor legyártani az x konfiguráció két rákövetkező konfigurációját. Ezzel így egy csúcs kifejtése $O(f(n))$ időben megvan (max két ekkora rákövetkező konfigurációt kell legyártanunk), annak ellenőrzése, hogy láttuk-e már a csúcsot, szintén megvan $O(f(n))$ időben (radix tree használva a halmazhoz), hozzárakni a konténer adatszerkezetéhez pedig $O(f(n))$ idő szintén, amíg odamásoljuk a konfigurációt. Összességében tehát így a keresőalgoritmus egy csúcsot $O(f(n))$ idő alatt kezel le, tehát az összes időnköltségünk így $O(f(n) \cdot k^{f(n)})$, ami persze szintén $O(k^{f(n)})$ esetleg egy nagyobb k konstansra, hiszen $f(n) = O(k^{f(n)})$ és így az előző időigény $O((k^{f(n)})^2) = O((k^2)^{f(n)})^a$.

^ahatványozás-azonosság

Eszerint az állítás szerint pl. $\mathbf{NL} \subseteq \mathbf{P}$, hiszen $\mathbf{NL} = \text{NSPACE}(\log n) \subseteq \text{TIME}(k^{\log n})$, a $k^{\log n}$ pedig egyenlő $n^{\log k}$ -val (ez pl onnan is látszik, hogy ha mindkét kifejezés logaritmusát vesszük, és logaritmus azonossággal lehozzuk szorzóba a kitevőt, mindkettőből $\log k \cdot \log n$ lesz), és mivel k a képletben egy konstans, így $\log k$ is az, tehát $n^{\log k}$ egy polinom. Ugyanígy, $\mathbf{PSPACE} = \text{SPACE}(n^k) \subseteq \text{NSPACE}(n^k) \subseteq \text{TIME}(k_1^{n^k}) = \text{TIME}(2^{O(n^k)}) = \mathbf{EXP}$.

Azaz beláttuk, hogy az eddig megismert és nevesített osztályaink láncba rendezhetőek:

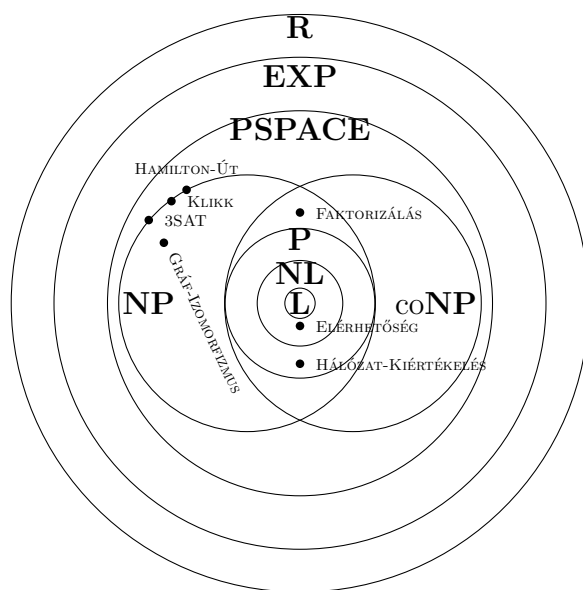
Állítás

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{R} \subsetneq \mathbf{RE}.$$

Meg persze az Immerman-Szelepcsényi tétel szerint tudjuk, hogy $\mathbf{NL} = \text{coNL}$, azt alaptól tudjuk, hogy \mathbf{L} , \mathbf{P} , \mathbf{PSPACE} és \mathbf{EXP} is zártak a komplementálásra, \mathbf{NP} -ről nem tudjuk (és sokan nem is hisszük), hogy így van-e, de a co monotonitása miatt coNP is \mathbf{P} és \mathbf{PSPACE}

közt van, Savitch tételéből azt tudjuk, hogy $\mathbf{PSPACE} = \mathbf{NPSPACE}$, ő azért nincs rajta a képen.

Tehát egyelőre kb. ennyit tudunk az eldönthető problémák nevesebb osztályairól és a bennük lévő némelyik problémákról:



21. A hierarchia tételek

A hierarchia tételek kb. arról szólnak, hogy ha ugyanabból a típusú erőforrásból sokkal több áll rendelkezésre, akkor olyan problémát is meg tudunk oldani, mint amit kevesebb erőforrással már nem.

A legjobb típusú ilyen tétel persze valami olyasmi lenne, hogy pl. „ha $g(n) = o(f(n))$, akkor $\text{TIME}(g(n)) \subsetneq \text{TIME}(f(n))$, ez mondaná azt, hogy ha nagyságrendben bármennyivel is több időnk van, akkor azzal máris több mindent tudunk megoldani.

Van hasonló tétel, de – mivel a bizonyításban diagonális módszert használunk és van benne egy olyan rész, ami kiszámítja $f(n)$ -t – kell bele egy olyan rész, mely azért felel, hogy ki tudjuk számítani és közben ne csússzunk ki a „nagyobb” függvény időigényéből. Ezeket a „normális” függvényeket fogjuk **megengedett bonyolultsági függvénynek** nevezni:

Definíció: Megengedett bonyolultsági függvény

Az $f(n) : \mathbb{N} \rightarrow \mathbb{N}$ függvény megengedett bonyolultsági függvény, ha van olyan algoritmus, mely ki tudja számítani az $1^n \mapsto 1^{f(n)}$ függvényt (tehát inputként kap egy n darab 1-esből álló stringet és outputra pontosan $f(n)$ darab 1-est kell kiírjon), $O(n + f(n))$ időben és $O(f(n))$ tárban.

Az $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f^2(n))$ tartalmazás bizonyításakor is azért kerültük meg a „gyártunk le $f(n)$ darab nullát számlálónak” részt és csináltunk helyette iteratívan mélyülő mélységi keresést, mert így ez a tartalmazkodás minden $f(n)$ -re igaz, ha a bizonyításban léptünk volna egy illet, akkor csak azokra a függvényekre jött volna ki, hogy igaz, amiket $f^2(n)$ tárban ki lehet számítani.

Gyakorlati szempontból nem nagy megszorítás az, hogy „megengedett” függvényeket használjunk:

- az $f(n) = n$ függvény megengedett, hiszen csak ki kell másoljuk az inputról az outputra, amit ott találunk, ez megy $O(n)$ időben és logaritmikus tárban (a pointerrel ahogy végigmegyünk az 1-eseken, ahhoz kell csak egy logtáras pointert használnunk), tehát beférünk az $O(n + n)$ időbe és $O(n)$ tárba;
- az $f(n) = c$ konstans függvények megengedettek, hiszen csak ki kell írunk c darab 1-est, ez megy konstans időben és konstans tárban;
- az $f(n) = 1 + \lfloor \log n \rfloor$ függvényt (tehát az n szám bináris alakjának a hosszát) ki tudjuk számítani úgy, hogy először az első munkaregiszterbe megszámoljuk, hány egyes van az inputon, ez $O(n)$ idő (minden egyesre egy $O(1)$ inkrementálás) és $O(\log n)$ tárigényű (ez elég az n szám tárolására), majd ezek után egy másik regisztert elkezdünk 1-ről duplázni, minden duplázás után kiírva egy 1-est az outputra, egész amíg el nem érjük az n -t, ez további $O(\log n)$ tár és idő. (Az inputot csak olvassuk, az outputra csak stream módban írunk). Így összesen $O(\log n) + O(\log n) = O(\log n)$ tárat használunk, és $O(n + \log n)$ időt (azaz $O(n)$ -t), ez is rendben van.
- ha $f(n)$ és $g(n)$ megengedett bonyolultsági függvények, akkor $f(n) \cdot g(n)$ is az: első menetben ahelyett, hogy $f(n)$ darab 1-est íránk ki az outputra, növelgetünk egy munkaregisztert $f(n)$ -szer, ezt meg tudjuk tenni $O(n + f(n))$ időben, $O(f(n))$ tárban, egy másik munkaregiszterbe kiszámoljuk binárisan ugyanígy $g(n)$ -t is, $O(n + g(n))$ időben és $O(g(n))$ tárban (note: ehhez a két részeredményhez kell pluszban $O(\log f(n))$ és $O(\log g(n))$ plusz tár, de persze $O(f(n)) + O(\log f(n)) = O(f(n))$, tehát ez még befér a nagyságrendbe. Ezután összeszorozzuk a két regiszter tartalmát, erre már láttunk RAM algoritmust, mely $O(\log^3 f(n))$ idő alatt elvégzi a szorzást, majd az eredményt egyesével csökkentve minden csökkentés után kiírunk outputra egy 1-est, amíg nullára nem érünk, ez újabb $O(f(n) \cdot g(n))$ idő lesz. A felhasznált idő $O(n + f(n)) + O(n + g(n)) + O(\log^3 f(n)) + O(f(n) \cdot g(n))$, ami $O(n + f(n) \cdot g(n))$, ez rendben, a tár pedig $O(f(n)) + O(\log f(n)) + O(g(n)) + O(\log g(n)) = O(f(n) \cdot g(n))$, ez is rendben.
- Tehát például minden konstans, az identikus függvény, két függvény szorzata, összege is megengedett, így minden polinom is az, meg még számos másik függvény, ami az ember eszébe juthat.

Most, hogy jobban ismerjük a megengedett függvényeket, kimondhatjuk a hierarchia tételeket:

Állítás: Tárhierarchia tétel

Ha $f(n)$ és $g(n)$ megengedett bonyolultsági függvények és $f(n) = o(g(n))$, $g(n) = \Omega(\log n)$, akkor $\text{SPACE}(f(n)) \subsetneq \text{SPACE}(g(n))$.

Például: ha $f(n) = \log n$ és $g(n) = n$, akkor rájuk teljesül a tétel feltétele, tehát $\text{SPACE}(\log n) \subsetneq \text{SPACE}(n)$, vagyis lineáris tár ténylegesen több mindenre elég, mint logaritmikus. Persze akkor polinom tár is több mindenre elég, mint logaritmikus és azt kapjuk, hogy $\mathbf{L} \subsetneq \mathbf{PSPACE}$.

Ennél tudunk jobbat is: a Savitch-tétel következménye szerint $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$. Ha $f(n) = \log n$ -t írunk ebbe, akkor kapjuk, hogy $\mathbf{NL} = \text{NSPACE}(\log n) \subseteq \text{SPACE}(\log^2 n)$. A tárhierarchia tételbe írva $f(n) = \log^2 n$ -t és $g(n) = n$ -t, ezekre is állnak a feltételek ($\log n$ -t láttuk, hogy megengedett, szorozni meg lehet őket), tehát $\text{SPACE}(\log^2 n) \subsetneq \text{SPACE}(n) \subseteq \mathbf{PSPACE}$ is igaz, ezek miatt $\mathbf{NL} \subsetneq \mathbf{PSPACE}$.

Az időre vonatkozó hierarchia tétel:

Állítás: Időhierarchia tétel

Ha $f(n) \geq n$ és $g(n)$ megengedett bonyolultsági függvények, és $f(n) = o(g(n))$, akkor $\text{TIME}(f(n)) \subsetneq \text{TIME}(g(n))$.

Itt pedig például azt kapjuk, hogy ha $f(n) = 2^n$ -t és (mondjuk) $g(n) = 3^n$ (mindkét függvény megengedett bonyolultsági függvény), akkor a feltétel azt kéri tőlünk, hogy $2^n = o(3^n)$ legyen, ez stimmel is. Tehát az időhierarchia tétel szerint $\text{TIME}(2^n) \subsetneq \text{TIME}(3^n)$. Azt is tudjuk, hogy $\mathbf{P} \subseteq \text{TIME}(2^n)$, hiszen minden n^k polinomra $n^k = O(2^n)$. Azt is tudjuk, hogy $\text{TIME}(3^n) \subseteq \mathbf{EXP}$. Ezekből pedig azt kapjuk, hogy $\mathbf{P} \subsetneq \mathbf{EXP}$. (sőt már azt is, hogy $\mathbf{P} \subsetneq \text{TIME}(c^n)$ tetszőleges $c > 1$ alapra, ha $g(n) = c^n$ -nel és $f(n) = c_2^n$ -nel indítjuk, egy tetszőleges $1 < c_2 < c$ konstansra.)

A kettő közül az időhierarchia tételt be is bizonyítjuk vázlatosan:

Bizonyítás

Legyen $f(n) \geq n$ egy megengedett bonyolultsági függvény. Adnunk kell egy olyan problémát, mely eldönthető $O(f(n))$ időben, de nem dönthető el $o(f(n))$ időben. Ez a probléma a következő lesz, nevezzük MEGÁLLÁS $f(n)$ LÉPÉSBEN-nek:

- **Input:** Egy M offline RAM program.
- **Output:** Igaz-e, hogy M ha a saját forráskódját kapja meg inputként, azon $f(|M|)$ lépésen belül megáll?

Itt persze M (mondjuk) abban a kódolásban van megadva, melyben az univerzális RAM gép várja a forráskódokat, $|M|$ pedig ebben a kódolásban az M leírásához szükséges bitek száma.

Először is, a probléma $\text{TIME}(f(n))$ -ben van^a

- Először is kiszámoljuk $f(n)$ -t (a „megengedett bonyolultsági függvény” defje szerint pl. úgy, hogy $f(n)$ darab munkaregiszterbe 1-eseket írunk, majd összeszámoljuk őket egy regiszterbe. Ez lesz az óránk, ami ketyeg vissza. Mivel $f(n)$ megengedett, így kiszámítható $O(f(n))$ időben.
- Ezután elkezdjük szimulálni M -et a saját forráskódján. Mivel M offline, nem fogja átírni az inputot, így nem probléma, hogy a forráskód is és az input is ugyanazon a területen foglal helyet: mikor M -ben egy utasítás egy $I[t]$ értéket próbálna elérni, akkor a tényleges $I[t]$ -t adjuk vissza, mikor pedig a szimulációhoz szűjtjük be az utasítás opcode-ját és argumentumait, olyankor a t -edik utasítás opcode-ját $I[4t]$ -ben, az argumentumokat az utána lévő legfeljebb három regiszterben keressük.
- Futtatjuk M -et a saját forráskódján (egy szimulált lépést konstans sok lépésben kiváltva), közben számolunk vissza az óránkon (csökkentve az óra regiszter tartalmát minden lépésben eggyel). Így egy lépést még mindig konstans sok lépésben szimulálunk. Ha M megállna, ACCEPT, ha pedig az óra jár le, akkor REJECT választ adunk.

Ez a szimuláció ezek szerint lemegy összesen $O(f(n))$ időben, tehát a probléma tényleg $\text{TIME}(f(n))$ -ben van.

Tegyük most fel, hogy a probléma eldönthető $g(n) = o(f(n))$ időben egy C (mint „cudálatos”) programmal. Azaz,

$$C(M) = \begin{cases} \text{ACCEPT} & \text{ha } M \text{ megáll } M\text{-en mint inputon } f(|M|) \text{ lépésben} \\ \text{REJECT} & \text{különbén (azaz ha nem áll meg a saját kódján, ha azt kapja} \\ & \text{inputnak, } f(|M|) \text{ lépésben belül)} \end{cases}$$

és C minden lehetséges M inputon $g(|M|)$ lépésben belül megáll.

Ennek a C programnak a forráskódjából készítünk egy D (mint „diagonális”, megállási probléma PTSD) programot, ami szintén egy forráskódot vár és a következőképp üzemel:

$$D(M) = \begin{cases} \nearrow & \text{ha } M \text{ megáll } M\text{-en mint inputon } f(|M|) \text{ lépésben} \\ \text{ACCEPT} & \text{különbén (azaz ha nem áll meg a saját kódján, ha azt kapja} \\ & \text{inputnak, } f(|M|) \text{ lépésben belül)} \end{cases}$$

Ezt könnyen el tudjuk érni: C forráskódjában ahol ACCEPT utasítás van, ott ehelyett csak feltétel nélkül ugranunk kell ugyanerre a sorra vissza, ahol meg REJECT, azt átírjuk ACCEPTre. Az így módosított D program ugyanannyi lépésben fut minden input M -re, mint az eredeti C , tehát $g(|M|)$ lépésben belül megáll. Még egy keveset módosítunk D forráskódján: a végére beszúrunk felesleges (persze elérhetetlen) kódrészleteket (pl egy csomó REJECT-et, akármit) úgy, hogy D hossza, $|D|$ már annyi legyen, hogy $g(|D|) < f(|D|)$ teljesüljön. Mivel $g(n) = o(f(n))$, létezik olyan n_0 küszöbszám, hogy minden $n > n_0$ -ra $g(n) < f(n)$ igaz legyen, annyi (fölösleges) REJECTet kell csak írunk D kódjának a végére, hogy elérje ezt a küszöbszámot.

Na most tegyük fel a kérdést, amire remélhetőleg mindenki várt:

$$D(D) = ?$$

Mivel D bármilyen forráskódot megkaphat inputként, akár a sajátját is.

Tudjuk, hogy $D(D)$ azt vizsgálja, hogy (az if feltételében M helyére mindenhol D -t kell írni, mert ez az input most) igaz-e, hogy D megáll D -n mint inputon $f(|D|)$ lépésben. Azt tudjuk, hogy D ugyanannyit lép D -n mint inputon, mint amennyit C lépne D -n mint inputon, C -ről pedig azt mondtuk, hogy $g(n)$ időkorlátos, tehát C , és így D is, legfeljebb $g(|D|)$ lépésben megáll D -n mint inputon. De D kódjába azért raktunk bele egy csomó fölösleges elérhetetlen rejectet, hogy $g(|D|) < f(|D|)$ teljesüljön. Tehát D megáll D -n mint inputon $f(|D|)$ lépésben belül. Azaz, D definíciója szerint ekkor végtelen ciklusba kell küldjük, ami meg azt jelenti, hogy D végtelen ciklusba is esik és meg is áll $f(|D|)$ lépésben belül a saját forráskódját megkapva inputként. Ez persze egyszerre nem lehetséges, tehát ez ellentmondás – vagyis az egyetlen „tegyük fel” rész feltevése kell hibás legyen és ezek szerint nem létezik olyan C program, mely $g(n) = o(f(n))$ időben eldöntené ezt a problémát.

Tehát $\text{TIME}(g(n)) \subsetneq \text{TIME}(f(n))$, ha $g(n) = o(f(n))$ és $f(n) \geq n$ tényleg fennáll.

“Ha RAM-gép helyett Turing-géppel tolnánk az anyagot, itt lenne egy különbség, mert a tömbcímezés hiánya miatt az univerzális Turing-gép nem tudja olyan gyorsan „összeszedni” az aktuális utasítást meg a szalagtartalmat, mint a RAM-gép; ezért Turing-géppel definiált időbonyolultsági osztályok esetén csak valami $\text{TIME}(g(n)) \subsetneq \text{TIME}(f^3(n))$ -like állítás jön ki „könnyen”, amit lehet kicsit optimalizálni, de afaik Turing-gépre a determinisztikus időhierarchia tétel kimondásánál kicsit „nagyobb gap” van $f(n)$ és $g(n)$ közt megkövetelve, mint a RAM gép esetében lévő o .

22. A logtáras visszavezetés

Korábban láttuk, hogy a polinomidejű visszavezethetőség „túl erős”, ha pl. a \mathbf{P} -beli problémákat szeretnénk egymáshoz viszonyítani, hiszen ha $A \in \mathbf{P}$ és B egy nemtriviális probléma, legyen mondjuk b_1 a B -nek egy „igen”, b_2 pedig a B -nek egy „nem” példánya (mindkettő van, mert B nemtriviális), akkor a következő algoritmus:

```
kamuVisszavezetés(x) {  
    if( A(x) ) return b1  
    else return b2  
}
```

ahol $A(x)$ az A -t polinom időben eldöntő algoritmus, formailag egy teljesen valid polinomidejű visszavezetés A -ról B -re: A -nak az „igen” példányaiból B -nek „igen” példányát (well konkrétan b_1 -et), a „nem” példányaiból pedig a b_2 „nem” példányt készíti – ez történik, ha a visszavezetésre, az inputkonverzióra annyi erőforrást engedünk, amennyivel a bal oldali problémát már akár meg is lehetne oldani.

Ahhoz, hogy a \mathbf{P} -n belüli problémákat is tudjuk osztályozni (mivel tudjuk, hogy $\mathbf{NL} \subseteq \mathbf{P}$, így például valid kérdés lehet, hogy mik a \mathbf{P} -nek azok a problémái, melyek **nincsenek** \mathbf{NL} -ben, ha a két osztály különbözik), egy „gyengébb” visszavezetés-fogalom kell. Amit ezen a kurzushoz nézni fogunk, az a **logaritmikus tárigényű**, avagy logtáras visszavezetés lesz:

Definíció: Logtáras visszavezetés

Legyenek A és B eldöntési problémák. Ha f egy olyan függvény, mely

- A inputjaiból B inputjait készíti,
- választató módon: A „igen” példányaiból B „igen” példányait, „nem” példányból pedig „nem” példányt,
- és logaritmikus tárban kiszámítható,

akkor f egy logtáras visszavezetés A -ról B -re. Ha A és B közt létezik ilyen, akkor azt mondjuk, hogy A logtárban visszavezethető B -re, jelben $A \leq_L B$.

Persze ekkor a fenti f mindenképp lyukszalagos algoritmus kell legyen, hiszen ha az inputot be kéne számítani a tárigénybe, az máris adna egy $\Omega(n)$ tárigényt, az nem lesz logaritmikus.

Persze mivel f egy determinisztikus algoritmus által kiszámított függvény, ezért nem érinthet a számítás közben kétszer ugyanolyan konfigurációt. Mivel az $O(\log n)$ tárat $2^{O(\log n)}$ -féleképp lehet teleírni, és minden pillanatban a program a K konstans darab utasítás egyikét hajtja éppen végre, így összesen $K \cdot 2^{O(\log n)}$ -féle különböző konfigurációja lehet, ami polinom, hiszen $2^{c \cdot \log n} = (2^{\log n})^c = n^c$. Emiatt a logaritmikus tárigényű algoritmusok szükségképpen polinom időben megállnak, és ezért a logtáras visszavezetés formailag tényleg „gyengébb” a polinomidejűnél:

Állítás

Ha $A \leq_L B$, akkor $A \leq_P B$.

Nagyon gyakran használtuk a \leq_P tranzitivitását, ami könnyen kijött: először végrehajtottuk az első visszavezetést, az eredményen meg végrehajtottuk a másodikat. Az első menet polinomidőben fut, létrehoz egy polinom méretű outputot, aztán ahhoz képest egy polinomidejű, tehát (mivel a polinomok zártak a kompozícióra: ha egy polinomban n helyére egy polinomot helyettesítünk mindenhová, akkor az eredmény egy polinom lesz) összességében az összetett algoritmus is polinomidejű volt. Ezért volt elég egy \mathcal{C} -nehézség bizonyításához a legtöbb esetben egyetlen már ismert \mathcal{C} -nehéz problémát visszavezetnünk az aktuális problémánkra, mert a tranzitivitás miatt ebből következik, hogy az aktuális problémánk is \mathcal{C} -nehéz.

A logtáras visszavezetés esetében ennyire nem egyszerű a dolgunk: a logtáras visszavezetésről tudjuk, hogy mivel logtáras, így polinomidőben megáll – de ettől még tud generálni polinom méretű outputot! Ezért azt nem tehetjük meg, hogy letároljuk a köztes eredményt munkaregiszterekben, hiszen az már túllőhet a logaritmikus tárkorláton.

Egy ügyesebb (de lassabb) konstrukcióval viszont kijön:

Állítás

Ha f és g logtáras függvények, akkor kompozíciójuk is az.

Bizonyítás

Legyen A egy program, ami logtárban kiszámítja f -et, B pedig g -t, szintén logtárban. Módosítsuk A -t úgy, hogy az output regiszterei helyett **két** új working regisztert használ, legyenek ők mondjuk R_1 és R_2 :

- amikor eredetileg végrehajtana egy $O[x] := y$ alakú értékadást, azaz írna az outputra (amit mivel logtáras, stream módban kell tegyen), hajtsa ehelyett végre az $R_1 := x$, $R_2 := y$ értékadásokat!

Mivel A eredetileg logtáras volt, x és y is bele kell férjen a logtárba, ezért az eredeti programunk plusz ez a két regiszter az output helyett még mindig csak logtáras.

Most változtassuk meg B kódját is: egyrészt használjon A -tól teljesen különböző munkaterületet (ez pl megoldható úgy is, hogy megint megváltoztatjuk A kódját úgy, hogy minden regisztere helyett kétszer akkora indexűt használjon, így A fogja használni a „páros” indexűeket, B kódjában pedig minden index helyett a kétszer akkora plusz egy indexet használjuk, így B fogja használni a „páratlan” indexűeket. Mivel mindkét algoritmus max $O(\log n)$ munkaregisztert használhat, különben túlmennének a tárkorláton, így összesen a címekhez szükséges tárterület csak $O(\log n)$ -nel nő meg, és a két számítás nem fog beza-
varni egymásnak).

Másrészt, amikor B az $I[k]$ inputregisztere értékét olvasná ki, akkor ugye valójában az $f(x)$ outputjának a k . regiszterére lenne szüksége. Ilyenkor történjen a következő:

- ha $R_2 > k$, akkor A törölje az összes használt regiszterét és kezdje elölről a számítást;
- amíg $R_2 < k$, futtassuk A -t, B közben ne lépjen semmit, ő most várja az A outputjának a k . regiszterjének az értékét;
- és amint $R_2 = k$, vagyis A kiírja az eredeti $O[k]$ -t R_1 -be, vegye át B a vezérlést és tegye R_1 -gyel, amit $I[k]$ -val tett volna.

Vagyis mindig csak egy output regiszter tartalmát jegyezzük meg, azzal együtt, hogy ő hanyadik; alapvetően B -t futtatjuk, A csak arra az időre kapja meg a vezérlést, ameddig előállítja B számára azt az output regisztert, amire annak épp akkor inputként szüksége van.

Így összesen marad az $O(\log n)$ tárkorlát és az összetett programunk épp a két függvény kompozícióját fogja kiszámítani.

Azt nem tudjuk, hogy \leq_L tényleg gyengébb-e, mint \leq_P , mindenesetre az biztos, hogy pontosan akkor gyengébb, ha $\mathbf{L} \neq \mathbf{P}$ (amiben azért a legtöbb elméleti számítástudós hisz is), hiszen

- ha $\leq_L = \leq_P$, akkor pl. vegyünk azt a problémát, amiben az input n darab 1-es, akkor kéne elfogadni, ha n páros, legyen ez a probléma A (valójában mindegy, hogy mi is ez az A , csak legyen egy nemtriviális \mathbf{L} -beli probléma: ez az, hiszen csak egy pointerrel végig kell futnunk az inputon és közben váltogatni egy bitet, hogy páros vagy páratlan volt eddig). Tudjuk, hogy \mathbf{P} bármelyik két nemtriviális problémája visszavezethető egymásra polinomidőben, ezért ha $\leq_L = \leq_P$, akkor logtárban is; így bármelyik \mathbf{P} -beli probléma logtárban visszavezethető erre az A problémára, ami logtárban eldönthető. Összetéve a két algoritmust úgy, ahogy a \leq_L tranzitivitásának bizonyításában láttuk, kapunk végeredményben egy logtáras algoritmust, mely eldönti a kiindulási tetszőleges \mathbf{P} -beli problémánkat – vagyis ekkor \mathbf{P} minden problémája eldönthető logtárban, azaz $\mathbf{P} = \mathbf{L}$.
- ha pedig $\mathbf{P} = \mathbf{L}$, akkor az azt jelenti, hogy minden polinomidőben eldönthető **eldöntési** probléma eldönthető **logtárban** is. Ezt kéne felhasználni ahhoz, hogy egy polinomidőben **kiszámítható függvényt** logtárban is kiszámítsunk. Legyen mondjuk ez a függvényünk az f ; azt már tudjuk, hogy $f(x)$, az output hossza szintén polinom lehet csak (onnan, hogy már láttuk korábban: minden RAM programhoz van olyan c konstans, melyre igaz, hogy tetszőleges n -bites input feldolgozásakor a t . lépés után minden regiszterben csak $\max(n + c + t)$ -bites számok lehetnek, az output regiszterekben is, t ebben az esetben polinom lesz és csak polinom sok van belőlük). Legyen ez a polinom felső korlát az output méretére $p(n)$. Vegyük akkor a következő eldöntési problémát: input (x, k) , ahol x n -bites és $k \leq p(n)$, igaz-e, hogy $f(x)$ k -adik bitje 1-es? Ez így egy polinomidőben eldönthető eldöntési (!) probléma, hiszen csak futtatnunk kell f -et és megnézni az output k . bitjét. Viszont ha $\mathbf{P} = \mathbf{L}$, akkor erre van logtáras eldöntő algoritmus. Nincs más dolgunk tehát, mint $k = 1 \dots p(n)$ -ig (ezt a k -t egy munkaregiszterbe el tudjuk rakni, mérete $\log p(n)$ lesz, ami $O(\log n)$, belefér a tárkorlátba) futtatni egy ciklust, mindre eldönteni logtárban, hogy $f(x)$ -nek a k -adik bitje 1-es-e, ha az, akkor 1-est, ha nem az, akkor 0-t teszünk outputra és kész is van egy logtáras algoritmus f kiszámítására. (Az output regisztereket is meg tudjuk címezni, mert az is belefér a logtárba.)

(Technikailag mivel nem bitstreamet, hanem szám-streamet teszünk outputra, valójában inkább az „ $f(x)$ k -adik output regiszterének ℓ -edik bitje létezik-e/nulla-e/egyes-e” problémát kellene formailag használni, a konstrukció ugyanaz.)

Tehát megkaptuk, hogy:

Állítás

$\mathbf{L} = \mathbf{P}$ pontosan akkor teljesül, ha $\leq_L = \leq_P$.

Annyit ezen a ponton érdemes megjegyezni, hogy a kurzuson ami polinomidejű visszavezetést eddig csak néztünk, azok titokban mind logtárasak is voltak. Vannak (pl. a Papadimitriou

könyv is ilyen) olyan szerzők, akik az **NP**-teljes, **PSPACE**-teljes stb. fogalmakat eleve nem a polinomidejű, hanem a logtáras visszavezetéssel definiálják – az egy nyitott kérdés szintén, hogy vajon ugyanazok a problémák **NP**-teljesek-e a polinomidejű visszavezetésre nézve, mint amelyek a logtárasra.

23. P-teljes és NL-teljes problémák

Mivel **P**-n (és így **NL**-en) belül nincs értelme a polinomidejű visszavezetést használni problémák bonyolultságának összehasonlításra, ezért ezeken az osztályokon belül másképp definiáljuk a rájuk vonatkozó nehézségi/teljességi fogalmakat:

Definíció

Legyen $L \subsetneq C \subseteq P$ problémák egy osztálya. Azt mondjuk, hogy az A probléma **C-nehéz**, ha C minden eleme **logtárban** visszavezethető A -ra.

Ha ezen kívül A még ráadásul C -beli is, akkor A egy **C-teljes** probléma.

Tehát a különbség az, hogy **NP**-nél még a polinomidejű visszavezetés volt, amire vadásztunk, **P**-n belül lefelé a logtáras lesz, amivel problémák bonyolultságát mérjük. (Azért van ott, hogy **L**-nél C legyen bővebb, mert **L**-en belül megint csak nincs értelme a logtáras visszavezetésnek pont azért, amiért a polinomidejűnek nincs **P**-n belül.)

Ha megnézzük a Cook tételének bizonyítását, abban egy tetszőleges **NP**-beli problémát polinomidőben eldöntő nemdeterminisztikus Turing-géphez konstruáltunk meg egy formulát, melynek az egyes változói azt kódolták, hogy a nemdeterminisztikus program a változóhoz tartozó lépésben a két választási lehetősége közül melyiket választja. Magának a gépnek a működését leíró részei a formulának változómentesek voltak. Továbbá, ez a visszavezetés is logtáras volt.

Ebből következik, hogy ha ezt a visszavezetést egy **determinisztikus** polinomidejű Turing-gépre alkalmazzuk azzal, hogy az összes $x_1, \dots, x_{p(n)}$ változó helyett (mondjuk) konstans 0-t írunk be (a nemdeterminisztikus esetben úgy kezeltük eleve a két lehetőséget, mikor csak egy volt, hogy vigyenek akkor ugyanoda – így mindegy, hogy a valójában determinisztikus lépésekhez tartozó x_i értéke 0 vagy 1, ugyanaz lesz a formula értéke, hát akkor hagyjuk ki a változót és legyen mondjuk 0 mind), akkor a kapott változómentes formula pontosan akkor lesz igaz, ha a gép elfogadja az inputját.

Tehát:

Állítás

A következő problémák **P**-teljesek:

- input egy változómentes ítéletkalkulus-beli formula, igaz-e az értéke?
- HÁLÓZAT-KIÉRTÉKEELÉS

A hálózatkértékelés (adott egy változómentes hálózat, igaz-e az értéke) azért lesz **P**-teljes, mert formulából könnyen tudunk (szintén logtárban) őt kiszámító hálózatot konstruálni, és a \leq_L tranzitivitása miatt elég az ismert **P**-teljes formulakiértékelést visszavezetni a HÁLÓZAT-KIÉRTÉKEELÉSre és kijön, hogy utóbbi is **P**-nehéz (és mivel kiértékelni persze ki tudunk egy bejárással és az értékek kapukra írásával egy hálózatot, **P**-teljes is lesz).

Mivel azt tudjuk, hogy $\mathbf{NL} \subseteq \mathbf{P}$, és hogy $\text{ELÉRHETŐSÉG} \in \mathbf{NL}$, ez így azt mondja, hogy ha $\mathbf{NL} \neq \mathbf{P}$, akkor az elérhetőség egy könnyebb probléma, mint a hálózatkértékelés.

Az elérhetőség probléma pontos bonyolultságát most tudjuk meghatározni:

Állítás

Az ELÉRHETŐSÉG probléma \mathbf{NL} -teljes.

Bizonyítás

Azt már tudjuk, hogy \mathbf{NL} -beli, így csak azt látjuk be, hogy \mathbf{NL} -nehéz. Legyen tehát $A \in \mathbf{NL}$ egy probléma, és M egy olyan algoritmus, mely logtárban eldönti az A problémát.

Ismét az elérhetőségi módszert használjuk: nincs más dolgunk, mint elkészíteni adott x inputhoz az M algoritmus x -en való futásának a konfigurációs gráfját (ehhez csak az $O(\log n)$ -bites konfigurációkat kell kiírunkunk munkaregiszterekbe, mindig újra felhasználva ugyanazt a tárterületet, kettesével, majd ellenőrizni, hogy az aktuális an kiírt (C_1, C_2) konfiguráció-pár közt van-e átmenet M konfigurációs gráfjában – ez mindössze annak ellenőrzéséből áll, hogy a C_1 -beli utasításnak megfelelően változott-e a regiszterek tartalma (pointerekkel végig tudunk menni a regiszter indexeken, csak egy változhat, azt szintén tudjuk pointerekkel ellenőrizni, hogy az az egy megfelelően változott-e, majd azt, hogy a sorszám is megfelelően változott-e), és ha igen, akkor kitesszük outputra a (C_1, C_2) élt. Ha több elfogadó konfiguráció is van, akkor még ezután kiteszünk még egy plusz csúcsot, ebbe viszünk éleket az elfogadó konfigurációkból és ezt nevezzük ki t elérendő csúcsnak, a kezdő konfiguráció pedig az s kiindulási csúcs lesz.

Az így kapott gráfban persze pont akkor lesz $s \rightsquigarrow t$ út, ha M -nek van olyan számítási sorozata, mely elfogadja x -et, így ez egy (logtáras) visszavezetés A -ról az ELÉRHETŐSÉG -re.

Ennél egy kicsit többet is be tudunk ugyanezzel a módszerrel bizonyítani, hasonlóan ahhoz, ahogy a SAT \mathbf{NP} -teljességénél is meg tudtuk szigorítani az inputot $3CNF$ -ekre:

Állítás

Az ELÉRHETŐSÉG probléma akkor is \mathbf{NL} -teljes, ha irányított **körmentes** gráfokat engedünk csak meg inputként.

Bizonyítás

Az előző konstrukciót egészítsük ki a következővel: az M algoritmus mivel logtáras, létezik egy olyan $p(n)$ polinom, amennyi különböző konfigurációja lehet egy n hosszú inputon. Módosítsuk M -et annyival, hogy inicializáljunk egy új változót a $p(n)$ értékkel (ez logtárban kiszámítható), és minden lépés után csökkentsük ennek a változónak az értékét eggyel! Ha leér nullára az értéke, akkor utasítsuk el az inputot. (Ez hasonlít ahhoz a számlálóhoz, amit az elérhetőséghez adott algoritmusunkban használtunk.)

Ez az új algoritmus is ugyanúgy eldönti az A problémát (hiszen ha van elfogadó számítás, akkor olyan is van, amiben nem tesz fölöslegesen köröket úgy, hogy egy konfigurációt kétszer is lát, tehát akkor van egy legfeljebb $p(n)$ hosszú elfogadó számítás is), viszont mivel a változó értéke mindig csökken, így a konfigurációs gráf, melyet generálunk, garantáltan körmentes lesz, nem érhetünk vissza ugyanoda, hiszen akkor az óráknak is vissza kéne

állni, mint minden változónak, a korábban már látott értékre. Tehát tetszőleges **NL**-beli probléma (logtárban) visszavezethető egy körmentes gráfban vett elérhetőségi problémára.

Tudjuk, hogy ha A \mathcal{C} -teljes, akkor \overline{A} komplementere $\text{co}\mathcal{C}$ -teljes; ezt ugyan csak a polinomidejű visszavezetésre nézve mondtuk ki, de igaz a logtárasra is, hiszen arra is igaz, hogy ha f egy visszavezetés A -ról B -re, akkor ugyanaz az f visszavezetés \overline{A} -ról \overline{B} -re is. Tehát az elérhetőségi probléma komplementere, ELÉRHETETLENSÉG ezek szerint coNL -teljes. Mivel pedig az Immerman-Szelepcsényi tétel szerint $\mathbf{NL} = \text{coNL}$, kapjuk a következőt:

Állítás

Az ELÉRHETETLENSÉG is egy **NL**-teljes probléma. (Körmentes gráfra is.)

Néhány problémáról ismert, hogy irányítatlan gráfokra más (lehet) a bonyolultságuk, mint irányítottakra; a Hamilton-út pl. nem ilyen, az akár irányított gráfról beszélünk, akár irányítatlanról, **NP**-teljes marad. Az elérhetőség viszont nem ilyen: az ELÉRHETŐSÉG IRÁNYÍTATLAN GRÁFOKRA már **L**-beli probléma.

A 2SAT problémára korábban adtunk hatékony algoritmust: elkészítettünk hozzá egy segédgráfot, melynek csúcsai a literálok voltak, és akkor vettünk fel egy $\ell_1 \rightarrow \ell_2$ élt, ha $\{\overline{\ell_1}, \ell_2\}$ egy klóz volt az inputban. Ez szintén megkonstruálható logaritmikus tárban, hiszen csak mint az előző konstrukcióban, fel kell írjuk a literálpárokat (logtárba elfér, csak a két változó indexét és polaritását kell eltárolni, az $O(\log n)$ tárban megy), majd egy pointerrel végigfutni az input klózáin és ellenőrizni, hogy a klóz megfelel-e az épp felírt két literálnak. Ha igen, kiírjuk ezt az élt az output éllistába. Majd, a formula pontosan akkor volt kielégíthetetlen, ha volt benne olyan x változó, melyre ebben a segédgráfban volt $x \rightsquigarrow \neg x \rightsquigarrow x$ kör, ami tartalmazta x -et is és a komplementerét is.

Ezt nemdeterminisztikus logtárban el lehet dönteni az elérhetőséghez hasonló nemdeterminisztikus bolyongással: először nemdeterminisztikusan kiválasztjuk az egyik x változót, majd (egyszerre csak egy csúcsot tartva a memóriában, azt, amelyiken épp vagyunk) nemdeterminisztikusan elkezdünk lépegetni, mindig az aktuális csúcs egy szomszédjára. Ha rátalálunk így $\neg x$ -re, akkor ezután folytatjuk a bolyongást, de most x -et keressük. Ha megint meglett, akkor elfogadjuk az inputot. Ha pedig nem szomszédra próbálunk lépni, vagy lejár egy N -ig számoló óra, akkor elutasítjuk.

Ez az algoritmus nemdeterminisztikus logtárban dönti el a 2SAT komplementerét (pontosabban: az az algoritmus, amelyik nem azt csinálja, hogy először kigenerálja az egész segédgráfot, és aztán azon bolyong, mert akkor a köztes eredmény túl nagy lesz; hanem az, amelyiket a logtáras konstrukciók komponálásánál néztük, hogy meg lehet azt is csinálni logtárban), tehát ez egy **NL**-beli probléma; mivel pedig $\mathbf{NL} = \text{coNL}$ (nagyon megy ebben a fejezetben az Immerman-Szelepcsényizés), ezért az is igaz, hogy 2SAT is **NL**-beli.

Ennél több is igaz:

Állítás

A 2SAT és komplementere **NL**-teljes problémák.

Bizonyítás

Az Immerman-Szelepcsényi tétel miatt elég csak az egyikre, mondjuk a 2SAT-ra visszavezetni egy ismert **NL**-teljes problémát, mondjuk az ELÉRHETETLENSÉGET.

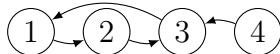
Azaz, egy irányított gráfból kell készítsünk egy 2CNF formulát, ami akkor lesz pontosan kielégíthető, ha a gráfban az s csúsból nem érhető el a t , mindezt logtárban.

Vegyünk fel minden v csúcshoz egy x_v logikai változót; ha (u, v) él a gráfban, akkor vegyünk fel a $(\neg x_u \vee x_v)$ klózt, végül vegyünk fel az $(s \vee s)$ és a $(\neg t \vee \neg t)$ klózokat! Azt állítjuk, hogy ez egy visszavezetés lesz a két probléma közt. (Mivel ezt is meg lehet csinálni úgy, hogy végigmegyünk egy pointerrel az összes élen és mindre kiírunk egy egyszerűen elkészíthető klózt outputra, a konstrukció logtáras.)

Ha s -ből nem érhető el t , akkor a generált formula kielégíthető lesz: vegyünk pl. azt az értékadást, mely x_u -t 1-re állítja, ha u elérhető s -ből és 0-ra, ha nem az! Ekkor az $(s \vee s)$ klóz igaz lesz, a $(\neg t \vee \neg t)$ klóz szintén igaz lesz, egy $(\neg x_u \vee x_v)$ klóz pedig hogy hamis legyen, ahhoz $x_u = 1$ és $x_v = 0$ kellene teljesüljön, de ez azt jelentené, hogy u elérhető s -ből (mert $x_u = 1$), v meg nem (mert $x_v = 0$), pedig van (u, v) él a gráfban (ezért vettük fel ezt a klózt), ami nyilván nem lehetséges, tehát a többi klóz is mind igaz lesz, tehát a formula kielégíthető.

Ha pedig s -ből elérhető t , akkor vegyünk egy $s \rightarrow u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_t \rightarrow t$ utat a gráfban és tegyük fel, hogy egy \mathcal{A} értékadás kielégíti a formulát. Tehát ekkor, mivel $(s \vee s)$ egy klóz, ezért $\mathcal{A}(x_s) = 1$. Az $s \rightarrow u_1$ él miatt $(\neg x_s \vee \neg x_{u_1})$ is egy klóz, és $x_s = 1$, ez csak úgy lehet, hogy $\mathcal{A}(x_{u_1}) = 1$ is igaz. Ugyanígy igaz kell legyen, haladva tovább az éleken, x_{u_2}, x_{u_3}, \dots , végül x_t is, de ez ellentmond a $(\neg x_t \vee \neg x_t)$ klóznak. A formula tehát ekkor tényleg kielégíthetetlen.

Példa a visszavezetésre. Legyen az ELÉRHETETLENSÉG inputja az alábbi gráf:



ez egy „igen” példány, hiszen 1-ből 4 elérhetetlen.

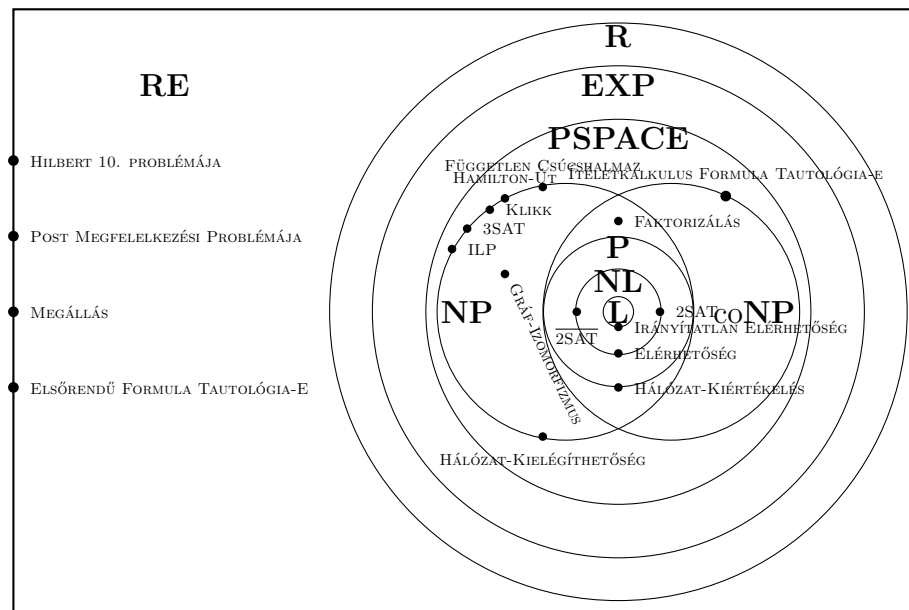
Akkor a visszavezetés a következő 2CNF-et készíti el:

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_4 \vee x_3) \wedge (x_1 \vee x_1) \wedge (\neg x_4 \vee \neg x_4).$$

Ami kielégíthető az $x_1 = x_2 = x_3 = 1, x_4 = 0$ értékadással.

(Ha s -ből t nem érhető el, akkor egy kielégítő értékadás: az s -ből elérhető csúcsokat állítjuk igazra – azokat muszáj is –, a többi hamisra – t -t muszáj is.)

A nevesített bonyolultsági osztályokról, ill. az ő teljes problémáikról kb. ezeket tudjuk eddig mostanra:



Nem láttunk még viszont **PSPACE**-re jellemző, mondjuk **PSPACE**-teljes problémákat. A következő fejezetben ezt a hiányt fogjuk orvosolni szívünkben.

24. PSPACE

Láttuk már korábban, hogy $\text{NP} \cup \text{coNP} \subseteq \text{PSPACE} = \text{NPSPACE}$, tehát ha eddig az **NP**-beli problémákról úgy tartottuk, hogy azok nehezek, akkor a **PSPACE**-nehéz problémák pedig nagyon nehezek. Azt persze még nem bizonyította be senki, hogy akár csak $\text{P} \neq \text{PSPACE}$ igaz lenne, de valahogy „nehezebbnek is érződnek” a **PSPACE** problémái, mint mondjuk az **NP**-beliek.

Láttuk már, hogy SAT, HAMILTON-ÚT, 3 – SZÍNEZÉS mind **PSPACE**-beliek, sőt $\text{SPACE}(n)$ -beliek. Persze a tárhierarchia tétel szerint $\text{SPACE}(n) \subsetneq \text{SPACE}(n^2) \subsetneq \text{SPACE}(n^3) \subsetneq \dots \subseteq \text{PSPACE}$, ahogy növeljük a kitevő értékét, egyre több és több probléma válik a megadott korláton belül eldönthetővé.

Egy újabb $\text{SPACE}(n)$ -beli probléma a kvantifikált SAT, azaz QSAT:

Definíció: QSAT

- Input: egy $\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{2m} \varphi$ alakú kvantifikált **ítéletlogikai** formula, melynek magja, φ konjunktív normálformájú kvantormentes formula, melyben csak az x_1, \dots, x_{2m} változók fordulnak elő.
- Kérdés: igaz-e φ ?

Fontos: ez **nem elsőrendű logika**! Itt az x_i változók csak az igaz/hamis értékeket vehetik fel.

Példa: ha az input $\exists x \forall y ((x \vee y) \wedge (\neg x \vee \neg y))$, akkor

- az $x = 0$ értékadás mellett ez $\forall y ((0 \vee y) \wedge (\neg 0 \vee \neg y)) \equiv \forall y (y \wedge (1 \vee \neg y)) \equiv \forall y (y)$ lesz, ez a formula hamis, hiszen $y = 0$ -ra (y) hamis lesz.

- az $x = 1$ értékadás mellett ez $\forall y((1 \vee y) \wedge (\neg 1 \vee \neg y)) \equiv \forall y((0 \vee \neg y)) \equiv \forall y(\neg y)$ lesz, ez a formula hamis, hiszen $y = 1$ -ra (y) hamis lesz.

Mivel pedig ezek szerint sem az $x = 0$, sem az $x = 1$ nem teszi igazzá a formula maradék részét, így a formula hamis.

(Fontos, hogy kívülről befelé haladjunk.)

Amit fentebb csináltunk, a szintaktikai értékadás: $F[x/0]$ jelenti azt, hogy F -ben az összes (szabad) x helyébe 0-t írunk. Ezzel a jelöléssel tudunk írni egy rekurzív kiértékelőt:

```
boolean eval(F) {
  switch( F ) {
    case 0: return false;           //ha a formula hamis, akkor hamis
    case 1: return true;            //guess what, ha igaz, akkor igaz
    case  $G \vee H$ : return eval(G) || eval(H);           //vagyolunk
    case  $G \wedge H$ : return eval(G) && eval(H);           //éselünk
    case  $\neg G$ : return !eval(G);           //negálunk
    case  $G \rightarrow H$ : return eval(H) || !eval(G)           //nyíl
    case  $G \leftrightarrow H$ : return eval(G) == eval(H);           //ekvivalencia
    case  $\exists x G$ : return eval(G[x/0]) || eval(G[x/1]);           //elég ha egyik jó, "létezik"
    case  $\forall x G$ : return eval(G[x/0]) && eval(G[x/1]);           //both jó kell legyen, "minden"
  }
}
```

Ez nem csak az elvárt input formátumnak megfelelően oldja meg a feladatot, hanem minden kvantifikált, zárt Boolean formulát ki tud értékelni.

Mi ennek az algoritmusnak a **tárigénye**? Alapvetően „lokális változó” nincs is benne, de mikor pl. a kvantoros esetben hívjuk a függvényt $G[x/0]$ -ra ill. $G[x/1]$ -re, akkor ezeket a „behelyettesített” formulákat azért le kell gyártuk, majd átadni őket rekurzívan, és egy ilyen formulának a hossza $O(n)$, ha az eredetié n volt (leesik egy kvantor, és a változó amit kötött átmegy 0-ba vagy 1-be, az még lineáris hossz marad in general). Viszont rekurzív függvény: mivel csak saját magát hívja, nem másik függvényt rekurzívan, így a tárigényt úgy kapjuk meg (a Savitch-tételben már látott módon), hogy megszorozzuk a lokális változók tárigényét ($O(n)$) a maximális rekurziós mélységgel. Mivel minden rekurzív hívásnál a formulának egy részformuláját vagy annak egy helyettesített változatát adjuk át, mindig rövidül a formula, így maximum $O(n)$ lehet a rekurziós mélység, tehát a tárigénye ennek az algoritmusnak $O(n) \cdot O(n)$, azaz $O(n^2)$.

Ennél tudunk eggyel jobbat is azért:

```
boolean *values = new boolean[2m]; //ide tároljuk az aktuális értékadásunkat
boolean eval(F) {
  switch( F ) {
    case 0: return false;           //ha a formula hamis, akkor hamis
    case 1: return true;            //guess what, ha igaz, akkor igaz
    case  $x_i$ : return values[i];           //változó értékét az értékadásból olvassuk ki
    case  $G \vee H$ : return eval(G) || eval(H);           //vagyolunk
    case  $G \wedge H$ : return eval(G) && eval(H);           //éselünk
    case  $\neg G$ : return !eval(G);           //negálunk
    case  $G \rightarrow H$ : return eval(H) || !eval(G)           //nyíl
    case  $G \leftrightarrow H$ : return eval(G) == eval(H);           //ekvivalencia
  }
}
```

```

case  $\exists x_i G$ :
    values[i] := 0;                                //először 0-val próbáljuk ki
    if( eval(G) ) return true;                       //ha az 1, akkor van amire igaz
    values[i] := 1;                                //ha nem jött be a 0, nézzük az 1-et
    return eval(G);
case  $\forall x_i G$ :
    values[i] := 0;                                //először 0-val próbáljuk ki
    if( !eval(G) ) return false;                     //ha az 0, akkor nem mindenre igaz
    values[i] := 1;                                //ha bejött a 0, nézzük az 1-et is
    return eval(G);
}

```

Ezzel már csak egy globális boolean tömböt kell nyilvántartanunk, az inputot most már nem írjuk át és mindig az aktuális formulánknak egy részformuláját értékeljük ki, amire elég egy pointert átadjunk. Ez így a RAM gép memória modelljének a defje alapján $O(n + n \cdot \log n) = O(n \log n)$ tárban megoldja a problémát (globális memória + a max n rekurziós mélység szorozva a $\log n$ méretű egy darab átadott pointerrel).

Az algoritmus tárigénye tovább javítható úgy, hogy a formulát egy darab pointerrel járjuk be rekurzió helyett és magából az értékadásból számítjuk ki azt is, hogy kell-e még az $x_i = 1$ értékkel is újjáírjuk ugyanazt a szálát, vagy már visszaadhatunk egy végeredményt; ennek a kódjában már bele kellene mennünk abba, hogy hogyan van reprezentálva a formula és hogy kapjuk meg a részformulára mutató pointerből az őt közvetlen részformulaként tartalmazó formula őseit. Ezzel már kapnánk egy $O(n)$ tárkorlátos algoritmust.

Mindenesetre az, hogy lineáris, $n \log n$ -es vagy négyzetes, az nem számít a következőben:

Állítás

QSAT \in PSPACE.

A következő tétel megmutatja, hogy miért pont a QSAT az, amivel random elkezdtünk most foglalkozni:

Állítás

A QSAT egy **PSPACE**-teljes probléma.

Bizonyítás

Azt már láttuk, hogy a QSAT egy **PSPACE**-beli probléma. Tehát, azt kell még megmutatnunk, hogy QSAT **PSPACE**-nehéz is. Legyen M egy polinom, mondjuk n^k tárkorlátos algoritmus, mely eldönt egy problémát.

Meg kell adnunk egy konverziót, mely az M problémának egy I inputjából elkészít polinom időben egy F kvantifikált boolean formulát, ami pontosan akkor igaz, ha $M(I)$ is igaz.

Ismét, mint már többször, az **elérhetőségi módszert** alkalmazzuk. Ha M egy n^k tárkorlátos algoritmus, akkor neki egy konfigurációja leírható n^k bittel (most vegyük ide az aktuális programsort is, legyen annak leírására is elég az n^k korlát), amit leírhatunk n^k darab logikai változóval.

Ebben a bizonyításban így fogunk elkódolni konfigurációkat, és egy félkövér **x** pl. egy $(x_1, x_2, \dots, x_{n^k})$ vektort fog jelenteni: n^k darab változót, melyek értékei balról jobbra ol-

vasva kiadják az M programnak egy konfigurációját.

A bizonyítás in spirit nagyon emlékeztet a Savitch-tételére: minden $d \geq 0$ -ra felírunk egy $\varphi_d(\mathbf{x}, \mathbf{y})$ (kvantifikált) logikai formulát, melyben az $x_1, \dots, x_{n^k}, y_1, \dots, y_{n^k}$ változók lesznek szabadok, és ami pontosan akkor igaz, ha **az \mathbf{x} konfigurációból az M algoritmus legfeljebb 2^d lépésben elérheti az \mathbf{y} konfigurációt.**

Ahhoz, hogy φ_0 -t felírjuk, megint csak szükségünk lenne arra, hogy egy konfigurációt *pontosan* hogyan is tudunk tárolni egy n^k hosszú bináris vektorban; ha ez megvan, utána már csak a „ha egy regiszter értéke nem egyenlő \mathbf{x} -ben és \mathbf{y} -ban, akkor vagy (ebben a sorban voltunk és ennek a regiszternek az értéke plusz ennek az értéke ennyi), vagy (ebben a sorban voltunk és ennek a regiszternek az értéke mínusz ennek a regiszternek az értéke ennyi) -like implikációt kell „csak” felírunk: a technikai részletektől most eltekintünk, az előadásjegyzethez egyszer remélhetőleg elkészülő függelékben majd szerepelni fog a pontos konstrukció.

Mindenesetre induljunk ki abból, hogy a $\varphi_0(\mathbf{x}, \mathbf{y})$ formula, mely azt fejezi ki, hogy \mathbf{x} -ből \mathbf{y} legfeljebb $2^0 = 1$ lépésben elérhető, tehát amivel leírjuk, hogy „vagy $\mathbf{x} = \mathbf{y}$, vagy \mathbf{x} -ből 1 lépésben \mathbf{y} -ba jutunk”, azt $O(n^k)$ hosszú formulával le tudjuk írni. (Ez hihetőnek is hangzik: az egyenlőség teszteléséhez kell pl. egy n^k hosszú éselés, hogy $x_i = y_i$, az egy lépésben elérhetőséghez meg olyan, fenti típusú implikációkból n^k darab éselése, amiknek a jobb oldalán a program méretétől függő (azaz, egy M -től függő konstans) hosszú vagyolás van.)

Namost mivel n^k bittel le tudunk írni kompletten egy konfigurációt és a szimulált M program (mivel eldönt valamilyen problémát) nem eshet végtelen ciklusba, ezért a futása során minden konfigurációt max egyszer érinthet, ez azt jelenti, hogy legfeljebb 2^{n^k} lépésben (ennyiféle konfiguráció van max) megáll.

Továbbra is feltehetjük, hogy pontosan egy darab, $\mathbf{x}_{\text{accept}}$ elfogadó konfigurációnk van: akár úgy, hogy elfogadás előtt törölünk minden regisztert, ahogy szoktuk, akár úgy, hogy mesterségesen bedrótozunk egy új konfigurációt, amibe accept esetén lépünk (ez utóbbi megoldás a φ_0 -t, az előbbi meg a programot bonyolítja, de egyiket se indokolatlan mértékben). Jelölje mondjuk \mathbf{x}_0 a kezdő konfigurációt: ezt I -ből ki tudjuk számítani, neki megfelelően kell legyenek beállítva az input regiszterek, a többi meg nullázva kell legyen.

Tehát ha sikerül felépítenünk a φ_d -ket úgy, ahogy szeretnénk, akkor $\varphi_{n^k}(\mathbf{x}_0, \mathbf{x}_{\text{accept}})$ lesz az a formula, ami pontosan akkor igaz, ha a kezdőkonfigurációból az elfogadó elérhető, azaz ha M elfogadja I -t.

Ha visszaemlékszünk a Savitch-tételre, ott is azt használtuk ki, hogy x -ből y pontosan akkor érhető el legfeljebb 2^{d+1} lépésben, ha létezik olyan z , amire igaz, hogy x -ből z -be is eljutunk legfeljebb 2^d lépésben, és z -ből y -ba is legfeljebb 2^d lépésben. Ez így első körben adna egy ötletet φ_{d+1} felírására.

$$\varphi_{d+1}(\mathbf{x}, \mathbf{y}) := \exists \mathbf{z} (\varphi_d(\mathbf{x}, \mathbf{z}) \wedge \varphi_d(\mathbf{z}, \mathbf{y}))$$

Ez így abból a szempontból korrekt is, hogy pont akkor lesz igaz (arra kell figyelni persze, hogy \mathbf{z} tényleg új változókból álljon, ne ütközzön az x -ekkel meg az y -okkal – alapvetően minden változó x_i alakú, csak áttekinthetőségi okokból használunk most itt x -eket, y -okat meg z -ket), ha \mathbf{x} -ből \mathbf{y} elérhető legfeljebb 2^{d+1} lépésben.

A probléma ezzel az, hogy egy **formula hossza** nem ugyanaz, mint **egy algoritmus tárligénye**! Ha ezt egy kóddal értékeltetnénk ki, melynek d is argumentuma (ahogy tettük a

Savitch tétel bizonyításában), akkor mikor a második φ_d -t elkezdenénk kiértékelni, addigra már felszabadulna az első φ_d által lefoglalt memóriaterület. De formula kiírásakor ez nem így megy! ott a teljes hossz lesz a formulának a hossza, és ezzel a képlettel mindig, mikor növeljük d -t, egy **kétszer akkora** formulát kapunk, mert kétszer írjuk le a korábbit, átnevezve a változókat (ami nem változtat a hosszán számottevőt). Így φ_{n^k} , amit végeredményben le szeretnénk gyártani, hossza n^k duplázás után már 2^{n^k} -szer hosszabb lenne, mint φ_0 , azaz exponenciális hosszú, amit képtelenség kiírjunk polinomidőben. Tehát ez az inputkonverzió ugyan tartja a választ, de nem polinomidejű, ezt kell még megjavítsuk.

A hossz duplázódása azért volt, mert az előző iterációban kiírt φ_d -t kétszer is leírtuk. A következő módszerrel elég egyszer leírni:

$$\varphi_{d+1}(\mathbf{x}, \mathbf{y}) := \exists \mathbf{z} \forall \mathbf{z}' \forall \mathbf{z}'' \left(((\mathbf{x} = \mathbf{z}' \wedge \mathbf{z} = \mathbf{z}'') \vee (\mathbf{z} = \mathbf{z}' \wedge \mathbf{y} = \mathbf{z}'')) \rightarrow \varphi_d(\mathbf{z}', \mathbf{z}'') \right)$$

Jó oké szemre ez úgy néz ki, mintha hosszabb lenne, mint az előző volt, de ha φ_d alaphoz hosszú, akkor ez már nem így van! Ha kiszámoljuk ennek a formulának a hosszát: kezd három darab n^k hosszú kvantoros kötéssel, aztán van még 8 konfiguráció, amik közt egyenlőséget tesztel, és el meg vagyol (pl. az $\mathbf{x} = \mathbf{z}$ rész leírható formulával: $(x_1 \leftrightarrow z_1) \wedge (x_2 \leftrightarrow z_2) \wedge \dots \wedge (x_{n^k} \leftrightarrow z_{n^k})$, a többi is analóg módon), akkor ez eddig $3 + 8 = 11$ -szer n^k méret, plusz még φ_d mérete, amit ezúttal csak egyszer írunk le! Ez azt jelenti, hogy amikor növeljük d -t, növelésenként csak $11n^k$ -val lesz hosszabb, ami azt jelenti, hogy mire φ_{n^k} -hoz érünk, addigra $n^k \cdot 11n^k$ -val, azaz $11n^k$ -val lesz hosszabb a formula, mint az eredeti φ_0 volt, amiről azt mondtuk, hogy $O(n^k)$ hossz elég lesz.

Tehát, ha ez a φ_d konstrukció tartja a választ, akkor legalábbis a mérete az jó; nincs ugyan CNF-ben, de ez megoldható a korábbi HÁLÓZAT-KIELEGÍTHETŐSÉG \leq_P SAT visszavezetéssel pl., hogy abban legyen, a kvantorok pedig nincsenek negáción belül az egész így gyártott formulában, tehát egyszerűen a formula elejére kiírhatjuk őket (kezddhetünk összesen $3n^k$ darab konfiguráció, azaz $3n^{2k}$ kvantált logikai változó kiírásával pl).

Ez a „ránézésre hosszabb” formula pedig ekvivalens az előzővel, hiszen mikor jön ki hamisra az implikáció? akkor, ha a bal oldala igaz, a jobb oldala hamis. A bal oldal egy vagyolás; ha a bal oldala igaz, $(\mathbf{x} = \mathbf{z}' \wedge \mathbf{z} = \mathbf{z}'')$, akkor beírva az implikációba \mathbf{z}' és \mathbf{z}'' helyére kapjuk, hogy $\varphi_d(\mathbf{x}, \mathbf{z})$ hamis; ha a jobb oldala igaz, akkor ugyanígy azt kapjuk, hogy $\varphi_d(\mathbf{z}, \mathbf{y})$ hamis. Tehát a formula akkor lesz igaz, ha létezik olyan \mathbf{z} , melyre $\varphi_d(\mathbf{x}, \mathbf{z})$ is igaz és $\varphi_d(\mathbf{z}, \mathbf{y})$ is igaz, ez meg pont ugyanaz, mint amit a Savitch-tétel alapján is felírtunk első próbálkozásként, tehát a konstrukció helyes.

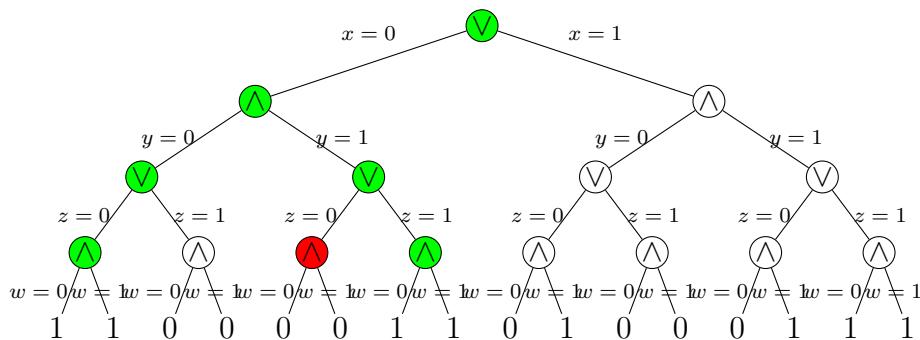
A szemfüles olvasó észrevehette, hogy a QSATban nem engedünk meg két egymás melletti egyforma kvantort, de ez a két kvantor közé betett másik kvantorral lekötött „dummy” változóval könnyen kezelhető és ekvivalens formulát ad (hiszen az új változó nem fogja befolyásolni az értékadást, melyben nem is szerepel).

Mielőtt néznénk egy „emberközelibb” PSPACE-teljes problémát (megint az van, hogy az első PSPACE-teljes problémát megtalálni ilyen munkás történet, a többi tudjuk úgy gyártani, hogy ebből az egyből vagy egy addigra már másik ismertből visszavezetéssel kapjuk meg őket), egy játékosabb szemüvegen keresztül nézzük meg a QSAT problémát.

Észrevehetjük, hogy az eredeti szintaxisnak megfelelő QSAT problémán ha futtatjuk a rekurzív solverünket, akkor ha felírjuk a rekurzió hívási gráfját, akkor olyan fát kapunk, ami kísértetiesen hasonlít egy (mestintből ismert) minimax fára:

Példa: a felső sorban látható formula rekurziós hívási gráfja lerajzolva:

$$\exists x \forall y \exists z \forall w ((\neg x \vee z \vee w) \wedge (y \vee \neg z) \wedge (x \vee \neg y \vee z))$$



Alul a 0/1-ek a formula magjának a kiértékelései az oda vezető értékadás mellett, a zöld csúcsok értéke 1, a pirosé 0, amelyek fehér maradt, azt (gyorsított kiértékelésnél) ki se kellett értékeljük, pl. mert a vagyolás bal oldala már 1 lett.

Mesterséges intelligencia tanulmányainkból már tudjuk, hogy egy ilyen fa kiértékelése ugyanaz, mint egy kétszemélyes, egymás elleni, zéró összegű játékban szeretnénk meghatározni, kinek is van nyerő stratégiája (és mi a játék értéke): a játék értéke végeredményben 0 lesz vagy 1, ilyen módon a \vee csúspontok kiértékelésekor pont az értékek maximumát, a \wedge kiértékelésekor pedig a minimumát kapjuk. A minimalizáló játékos értelemszerűen a kettő közül a 0-ra játszik, a maximalizáló pedig az 1-re – azaz a minimalizáló, aki a \wedge , vagyis a \forall -kötött változókhoz tartozó csúcsokban választ ágat, hamissá akarja tenni a formula magját, a maximalizáló, aki a \vee , vagyis a \exists -kötött változókhoz tartozó csúcsokban választ ágat, igazgá akarja tenni a formula magját. Tehát a QSAT probléma felfogható a következő módon:

Definíció: QSAT mint kétszemélyes játék

- **Input:** egy $\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{2m} \varphi$ alakú kvantifikált **ítéletlogikai** formula, melynek magja, φ konjunktív normálformájú kvantormentes formula, melyben csak az x_1, \dots, x_{2m} változók fordulnak elő.
- **Kérdés:** az első játékosnak van-e nyerő stratégiája a következő játékban?
 - A játékosok sorban értéket adnak a változóknak, előbb az első játékos x_1 -nek, majd a második x_2 -nek, megint az első x_3 -nak, stb., végül a második x_{2m} -nek.
 - Ha a formula értéke igaz lesz, az első játékos nyert, ha hamis, akkor a második.

Mivel ez pontosan ugyanaz a probléma a fentiek szerint, mint a QSAT volt, ez így ebben a formában is **PSPACE**-teljes.

Mindenesetre az, hogy itt épp egy formula az input és kvantorok vannak az elején, nem sokat számít abban, hogy **PSPACE**-ben legyen a formula: ha van egy „játékunk”, amiben is játék **állapotok** vannak, egy kiindulási állapotból, két játékos játszik egymás ellen, az egyik minimalizálni, a másik maximalizálni akarja a játék értékét, továbbá egy állapotban egyértelműen meghatározható az, hogy ez egy végállapot-e, ha az, akkor mi az értéke, ha meg nem az, akkor szép sorban megkonstruálhatók a rákövetkező lépések (ahogy pl. a QSAT-nál meg tudtuk konstruálni előbb $G[x/0]$ -t, aztán $G[x/1]$ -et), akkor egy algoritmus a játék minimax fájának kiértékelésére:


```

int eval(C) {
  if( C.isFinal() ) return C.getValue(); //ha vége a játéknak, akkor ez az értéke
  (int,int) => int op =
    switch( C.whoMoves() ) { //lássuk ki jön
      case MIN: (x,y) => min(x,y); //ha min jön, op a minimumot képző függvény lesz
      case MAX: (x,y) => max(x,y); //ha max jön, akkor a maximumot
    }
  //lekérjük sorban a rákövetkező állapotokat, kiértékeljük őket és eltároljuk,
  //majd visszaadjuk a minimumukat/maximumukat
  //egyszerre csak az épp vizsgált állapot kell a memóriában legyen, és az eddigi
  //min/max érték
  return C.getNextConfigs().map(eval).reduce(op);
}

```

Ugyanúgy, ahogy a QSAT-nál is, ha kiszámoljuk a tárigényt, akkor ha a következők mind igazak:

- a játék egy állapota elfér polinom térben;
- polinom térben el lehet dönteni, hogy melyik játékos következik;
- ha vége a játéknak, akkor polinom térben ki lehet számítani az értékét;
- ha nincs, akkor polinom tér felhasználásával lehet generálni az állapot rákövetkezőit egyenként;
- a játék polinom sok lépésben véget ér;

akkor ez a (teljes információs, zéró összegű, polinom lépéskorlátos, kétszemélyes) játék **PSPACE**-ben van.

Ennek a fordítottja is igaz: minden **PSPACE**-beli probléma felfogható ilyen játékként! Persze az, hogy mi is egy „játék”, elég tág fogalom – valójában ehhez elég egy tetszőleges **PSPACE**-beli A problémát mint egy olyan játékot definiálni, melyben első lépésként végrehajtunk egy $A \leq_P$ QSAT visszavezetést (ilyen van, mert QSAT **PSPACE**-teljes) és a kapott formulán a játékosok játszanak egy QSAT játékot – ha az első játékos nyer, akkor (feltéve, hogy optimálisan játszanak) az eredeti input A -nak egy „igen” példánya, ha B , akkor egy „nem” példánya volt.

Ennél egy direkter megközelítést fogunk látni az Alternálás fejezetben.

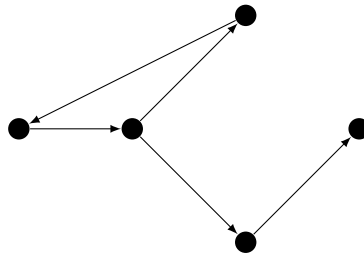
Ebben a fejezetben még nézünk akkor egy „emberközelibb” **PSPACE**-teljes játékot:

Definíció: FÖLDRAJZI JÁTÉK

- **Input:** egy $G = (V, E)$ irányított gráf és egy kijelölt „kezdő” csúcsa.
- **Output:** az első játékosnak van-e nyerő stratégiája a következő játékban?
 - Először az első játékos kezd, lerakja az egyetlen bábuját a gráf kezdőcsúcsára.
 - Ezután a második játékos lép, majd az első, stb, felváltva, mindketten a bábút az aktuális pozíciójából egy olyan csúcsba kell húzzák, ami egy lépésben elérhető, és ahol még nem voltak a játék során. Aki először nem tud lépni, veszített, a másik nyert.

Nézzünk erre egy példát:

Ha a gráf, amin a két játékos játszik, a következő:



és a bal oldali a kitüntetett csúcsa, akkor

- először az első játékos tesz a bal oldali csúcsra
- majd a második választási lehetőség híján jobbra húzza a bábút
- most az első választhat, hogy felfelé vagy lefelé húzzon. Ha felfelé húz, akkor a második játékos nem tud lépni (hiszen nem léphet vissza a kezdőcsúcsba, mert ott már jártak) és veszít; ha lefelé, akkor a második játékos még tud lépni jobbra egyet és ő nyer, mert innen az első nem tud.
- tehát a kettő közül az első játékos azt választja, hogy jobbra felfelé lép a felső csúcsba és így meg is nyeri a játékot, ez a példa a földrajzi játéknak egy „igen” példánya, mert az első játékosnak van nyerő stratégiája.

Ahogy már spoilereztem is:

Állítás

A FÖLDRAJZI JÁTÉK egy **PSPACE**-teljes probléma.

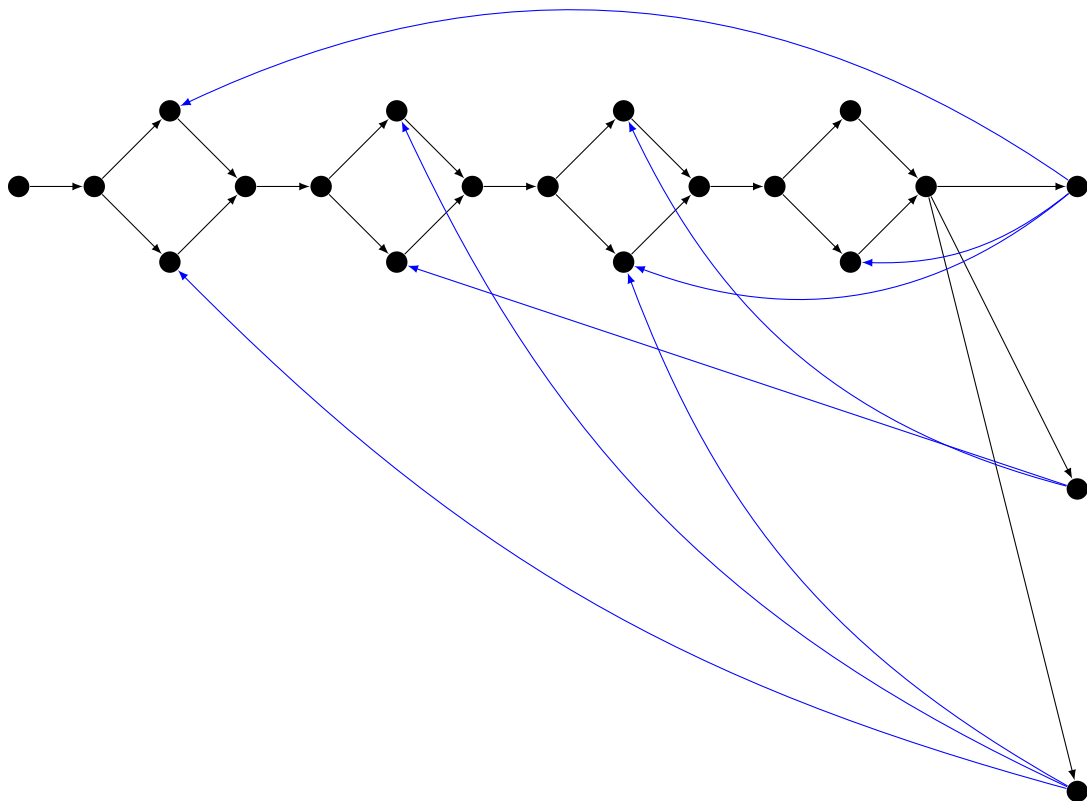
Bizonyítás

A játék egy állásában elég azt tárolnunk, hogy melyik csúcsokon jártunk már és épp hol van a bábu; ebből már kiszámítható, hogy ki jön, de ha nem tudjuk megszámolni a már érintett csúcsokat, vagy megkülönböztetni a páros számokat a páratlanoktól, akkor éppen egy boolean változóban is tárolhatjuk. Ez nyilván elfér polinom térben. Azt is el lehet dönteni, hogy vége-e a játéknak (nincs már él kifelé nem érintett csúcsba), és a rákövetkezőket is meg tudjuk konstruálni (csak egyesével hozzá kell venni a bábus csúcs még nem érintett szomszédait az érintett halmazhoz és odatolni a bábút), mindezt polinom térben, és persze legfeljebb annyi lépés lesz, ahány csúcs, ami polinom (lineáris).

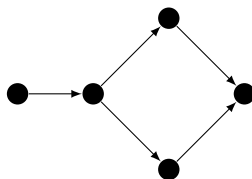
Tehát a játék **PSPACE**-ben van.

A **PSPACE**-nehézséghez megadunk egy (logtáras, tehát polinomidejű) visszavezetést QSAT-ról a FÖLDRAJZI JÁTÉKra.

A konstrukció a $\exists x \forall y \exists z \forall w ((\neg x \vee z \vee w) \wedge (y \vee \neg z) \wedge (x \vee \neg y \vee z))$ formulából ezt a gráfot fogja készíteni (az élek nem színesek, csak azért kék a jobb oldalról visszatérő élek mindegyike, hogy átláthatóbb legyen), melynek a bal szélső csúcsa a kijelölt kezdőcsúcs:



Lássuk részleteiben a konstrukciót: először is minden változóhoz elkészítjük az alábbi gráfot, és ezeket a gráfokat láncba kötjük:



Az előbb azért volt négy ilyen rész egymás után a példában, mert négy változó van benne. Ez a szám páros lesz, mert a QSAT-ban úgy követeltük meg, hogy az utolsóként kötött változó az egy páros indexű, univerzálisan kvantált változó legyen. Minden x változóhoz tartozó részgráfban az elágazás utáni egyik csúcsot x -szel, a másikat $\neg x$ -szel címkézzük. (Legyen a példában mondjuk úgy, hogy a négy „alsó” csúcs rendre x , y , z és w , a négy „felső” csúcs pedig rendre $\neg x$, $\neg y$, $\neg z$ és $\neg w$.)

Ezután az utolsó (az ábrán jobb oldali) változós gadgetnek generálunk annyi szomszédot, ahány klóz van a formula magjában (mivel a példában három klóz volt, ezért ennyi csúcs került jobb oldalra). Végül, a klóz-csúcsokból visszairányítunk éleket a változó-gadgetekbe: mindegyik klózból azokba a literálokba húzunk vissza éleket, amely literálok szerepelnek bennük.

Például a három klóz-csúcsból húztunk egy (az ábrán kék) élt az első változóhoz tartozó gadget felső csúcsába, azaz $\neg x$ -be, a harmadik gadget alsó csúcsába, azaz z -be és a negyedik alsó csúcsába, azaz w -be, mert a klóz $(\neg x \vee z \vee w)$ volt.

Azt állítjuk, hogy ezen a gráfon pontosan annak van nyerő stratégiája a földrajzi játékban, akinek nyerő stratégiája van az eredeti formula QSAT játékában.

Amíg a földrajzi játékot még balról jobbra játsszák a játékosok, addig nem sok lehetőségük

van:

- először az első játékos leteszi bal oldalra, mert az a kezdőcsúcs, a bábút
- a második játékos jobbra húzza a bábút, az az egy lépése van
- most az első játékos dönthet: jobbra fel vagy jobbra le húz. Ezzel ő tkp kiválaszt egyet x_1 és $\neg x_1$ közül. Fogjuk fel ezt egy értékadásnak úgy, hogy ha az x_1 literállal címkézett csúcsba húzza, akkor legyen az x_1 változó értéke **hamis**, ha pedig a $\neg x_1$ -gyel címkézettbe, akkor legyen az x_1 változó értéke **igaz**!
- eztán a kettes játékos nem tehet mást, behúzza középre, átjutottak az első gadgeten, de szerepet cseréltek: most az első játékos az, aki csak jobbra tud húzni.
- eztán a második játékos dönt: vagy jobbra fel, vagy jobbra le húz. Ha x_2 felé megy, akkor azzal azt jelzi, hogy legyen x_2 értéke hamis; ha $\neg x_2$ felé, akkor azzal azt jelzi, hogy legyen x_2 értéke igaz!

Tehát: ahogy a két játékos játszik és amíg balról jobbra halad valamilyen stratégia mentén, akkor döntési helyzetben pontosan akkor kerülnek, mikor a nekik megfelelő paritású gadgetben az elágazáshoz érnek, és ez a döntésük egyértelműen megfeleltethető egy értékadás-választásnak, amit a QSAT játékon játszanának.

(Azaz: eddig teljesen ugyanazt a játékfát rajzolnánk fel ennek a játéknak, mint amit a QSAT-nak rajzolnánk az eredeti formulán, ha kihagyjuk azokat a csúcsokat, ahol a soron következő játékosnak csak egy választása van.)

Az utolsó gadgetnél a paritás miatt a második játékos választ oldalt, az első játékos behúzza középre, és most van egy kis eltérés a két játék játékfája közt: a következő lépésben a második játékos behúzza egy klózba a bábút (ezt meg tudja tenni, ott még nem jártak, bármelyik klózba behúzhatja), és ezután

- az első játékos lehet, hogy máris veszít, ha a klózban lévő literálok mindegyikét érintették már idefele jövet – ekkor nem tud lépni.
- ha viszont van olyan literál, akit nem érintettek idefele jövet, akkor az első játékos abba be tudja húzni a bábút és a második játékos veszít, mert nem tudja onnan középre behúzni a bábút, hiszen ott már jártak.

Mit is jelent az, hogy az első játékos nem tudja kihúzni a klózból a bábút? Azt, hogy minden literálon, ami a klózban szerepel, jártak idefele jövet. Ami azt jelenti, hogy az értékadás, amit generáltak közben, ezeket a literálokat mindet **hamisra** állította. Tehát a klózban minden literál hamis, így a klóz hamis. Vagyis a második játékos akkor nyer, ha be tudja húzni a bábút egy olyan klózba, ami hamis, ami pontosan akkor történik, ha a formula magja, a CNF hamis. Ami pedig pontosan azt jelenti, hogy a második játékosnak pontosan akkor van nyerő stratégiája a QSAT játékon, ha a belőle generált FÖLDRAJZI JÁTÉKON is (bármelyik játékban ha van nyerő stratégiája, azt át lehet konvertálni a másik játék egy nyerő stratégiájára a fentiek szerint), tehát ez a visszavezetés tartja a választ és – ismét azért, mert a gráf legenerálásához elég csak korlátos sok pointer meg számláló – logtáras is.

Bizonyítás nélkül felsorolunk pár további **PSPACE**-teljes problémát:

Állítás

A következő problémák mind **PSPACE**-teljesek:

- Adott egy M determinisztikus RAM program és egy I inputja. Igaz-e, hogy M elfogadja I -t, méghozzá $O(n)$ tárat használva?
- Adott két reguláris kifejezés (klasszikus: csak betűk, unió, egymás után írás, csillag). Igaz-e, hogy ugyanazokra a szavakra illeszkednek?
- Adott két nemdeterminisztikus automata. Ekvivalensek-e?
- Adott egy SOKOBAN feladvány ($n \times n$ -es). Megoldható-e?
- Adott egy RUSH HOUR feladvány ($n \times n$ -es). Megoldható-e?

25. Pár szó PSPACE-től R-ig

Definiáltuk már korábban az $\mathbf{EXP} = \text{TIME}(2^{n^k})$ osztály és ennek a nemdeterminisztikus variánsát: $\mathbf{NEXP} = \text{NTIME}(2^{n^k})$. Tudjuk, hogy $\mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$, és az \mathbf{EXP} vs. \mathbf{NEXP} vs. coNEXP osztályok helyzete hasonlít a \mathbf{P} vs. \mathbf{NP} vs. coNP osztályok helyzetéhez, van is köztük összefüggés:

Állítás

Ha $\mathbf{P} = \mathbf{NP}$, akkor $\mathbf{EXP} = \mathbf{NEXP}$.

Az ember azt gondolná, hogy a 2^{n^k} már egy óriási időigény (elvégre ennek a faktoriális időigény is valahol az alján van, hiszen $n! = o(2^{n^2})$), de ennek is vannak olyan teljes problémái, melyek közismertek: ilyen pl. az (általánosított: $n \times n$ -es táblán játszott) **sakk** (mesterséges lépéskorlát nélkül), vagy a (japán szabályokkal játszott) **gó**. Ezek bár kétszemélyes játékok, az állás polinom tárban tárolható, a rákövetkezők is megkonstruálhatók ennyi tárban, a **PSPACE**-beliségük a lépésszámon „csúszik meg”: a bábukat exponenciális sokféleképp lehet mozgatni a táblán, ezért nem alkalmazható rájuk „automatikusan” a **PSPACE**-beliséget megmutató tétel. Egészen hasonló okokból nincs „automatikusan” **NP**-ben a Sokoban vagy a Rush Hour: van ugyan tanúsítvány-rendszer a megoldhatóságra, maga a lépéssorozat, ami betolja a helyére az összes objektumot, vagy ami kiviszi a piros autót a tábláról, ami ha megvan, még ráadásul könnyen ellenőrizhető is, hogy valóban egy tanú-e – de a hossza a legrövidebb megoldásnak lehet exponenciális, ezért ez a módszer nem mutatja meg az **NP**-beliséget.

Ha már reguláris kifejezések: a klasszikus reguláris kifejezéseket szintaktikailag feedelhetjük úgy, hogy megengedünk újabb operátorokat, pl. a négyzetreemelést (spec ezt a legtöbb regex motor tényleg támogatja is, RR helyett csak $R\{2\}$ -t kell leírni, ettől ugye rövidebb lehet a regex, ha maga R hosszú), vagy a komplementerképzést (ezt rendszerint nem támogatják, de írhatunk \overline{R} -t).

Már a négyzetreemelés magában is még nehezebbé teszi a regex ekvivalencia tesztelést:

Állítás

A következő probléma **EXSPACE**-teljes: adott két reguláris kifejezés, melyekben szerepelhet négyzetreemelés is, ekvivalensek-e?

Az **EXSPACE** persze a $\text{SPACE}(2^{n^k})$ osztály, amiről az alapvető összefüggések részről tudjuk, hogy tartalmazza pl. az egész **NEXP**-et is.

Felmerülhet az olvasóban, hogy mi történik, ha az exponenciális „tornyot” nőni engedjük, vajon az az időigény minden eldönthető probléma eldöntésére elég lesz-e? Nos, ezt a bonyolultsági osztályt is definiálták már:

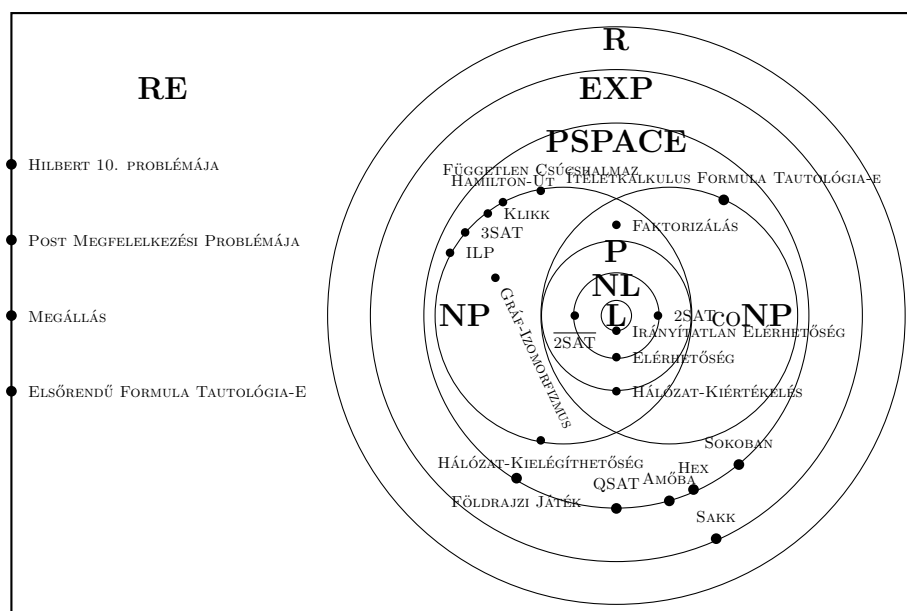
Definíció

Az „elemi” problémák osztálya: $\text{TIME}(2^n) \cup \text{TIME}(2^{2^n}) \cup \text{TIME}(2^{2^{2^n}}) \cup \dots$

Nos, nem, még ennyi idő sem garantáltan elég arra, hogy mindent megoldjunk, amire létezik algoritmus, pár talán meglepő tény:

- A következő probléma teljes az elemi problémák osztályában: adott két reguláris kifejezés, melyben komplementálás is szerepelhet, ekvivalensek-e?
- A 0/1 értéket kiszámító primitív rekurzív függvények osztálya valódi módon tartalmazza az elemi problémákat (és az intróban már említettük, hogy még azt is valódi módon tartalmazza az eldönthető problémák osztálya.)

Zárjuk le a félévet egy relatív friss osztályképpel:



Alternálás, AP és AL

Na ez nem volt 2020-ban.

Párhuzamosítás, NC

Esv2bzeo0aml2nt0.

Kriptográfia, egyirányú függvények, UP

Rm frz ibyg 2020-on.

Valószínűségi algoritmusok, RP, ZPP, BPP

Valószínűleg ez sem volt 2020-ban.

Kvantumszámítás, BQP

Guess what, ez sem volt 2020-ban, ahogy mindenki megfigyelhette.

Orákulumos számítások, P^{NP} , PH

A néni a sütivel, aki a házban lakik, ahol lassú a lift, azt mondta, hogy ez sem volt 2020-ban.

Függelék



meg kéne csinálni az univerzális RAM gépet

kóddal végig fullon



okthxbai

Tartalom

1. A tárgyról	1
2. Kiszámíthatóság- és bonyolultságelmélet	2
3. Architektúrák: RAM-gép és Turing-gép	9
3.1. Turing-gép	9
3.2. Eldöntés, felismerés, kiszámítás	11
3.3. Egyéb Turing-gép variánsok	13
3.4. RAM-gép	13
3.5. A RAM gép és a strukturált programozás	15
3.6. Turing-gép szimulálása RAM géppel	17
3.7. RAM gép szimulálása Turing-géppel	18
4. Első bonyolultsági osztályaink	19
5. Problémák összehasonlítása: visszavezetések	22
6. Eldönthetetlenség	30
7. Nemdeterminizmus	44
8. Polinomidőben verifikálhatóság	48
9. Cook tétele	49
10. A SAT variánsai	50
11. NP-teljes gráfelméleti problémák	58
12. NP-teljes problémák halmazokra és számokra	65
13. Pszeudopolinomiális algoritmusok, erős és gyenge NP-teljesség	72
14. Approximáció	75
15. P és NPC közt	89
16. Savitch tétele	90
17. A lineáris tárigény	95
18. Az NL osztály	96
19. Az Immerman-Szelepcsényi tétel	98
20. Alapvető összefüggések bonyolultsági osztályok közt	104
21. A hierarchia tételek	108
22. A logtáras visszavezetés	112
23. P-teljes és NL-teljes problémák	115
24. PSPACE	119
25. Pár szó PSPACE-től R-ig	129