Bonyolultságelmélet

Iván Szabolcs

Monday 11th October, 2021, 21:51

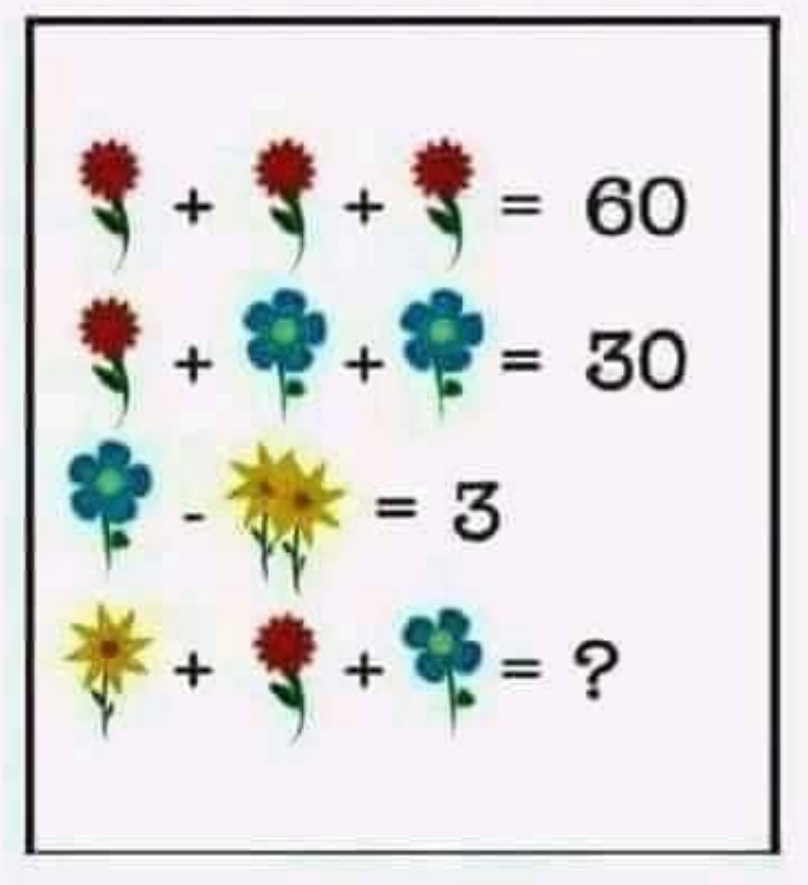
készült Ésik Zoltán korábbi előadásfóliái

és Christos H. Papadimitriou Számítási bonyolultság c. könyve

Az előadás felépítése

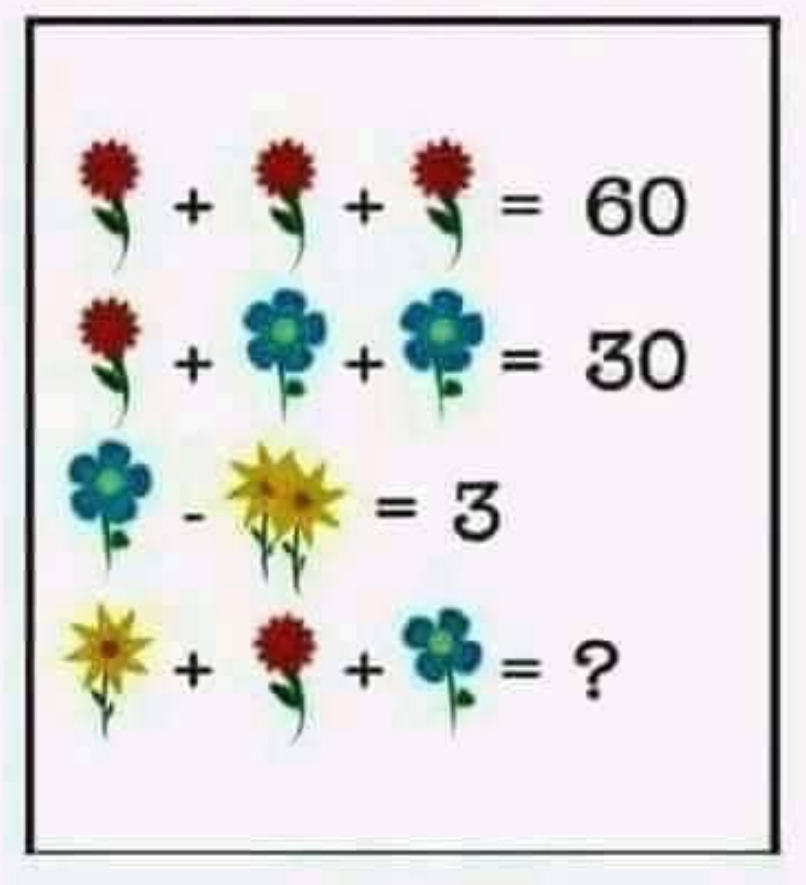
- A kiszámíthatóság- és a bonyolultságelmélet kialakulása
- Turing-gépek. A kiszámítás RAM modellje.
- Problémák egymáshoz viszonyított nehézsége: a (hatékony) visszavezetés
- Eldönthetetlen problémák
- Bonyolultsági osztályok
- Teljesség, nehézség; **NP**-teljes problémák zv!
- Approximáció, pszeudopolinomiális algoritmusok, parametrizált komplexitás
- **PSPACE**-nehéz problémák zv!
- Randomizált algoritmusok talán
- Párhuzamosítás talán

Warm-up



check this

Warm-up



check this

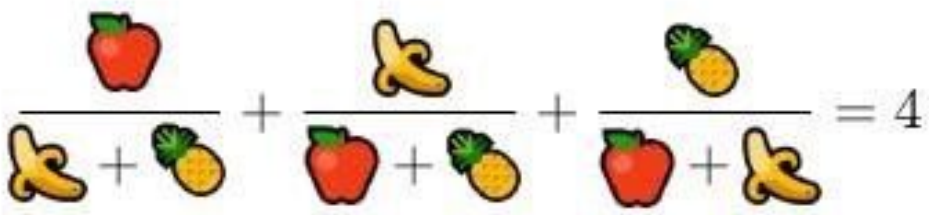
Megoldás

- pirosvirág: 20
- kékvirág: 5
- sárgavirág: 1 (oh my so clever lulz)
- célfüggvény (lol): 25
  ... mert csak négy kék szirom van

köszi a dc pmet erről

Warm-up

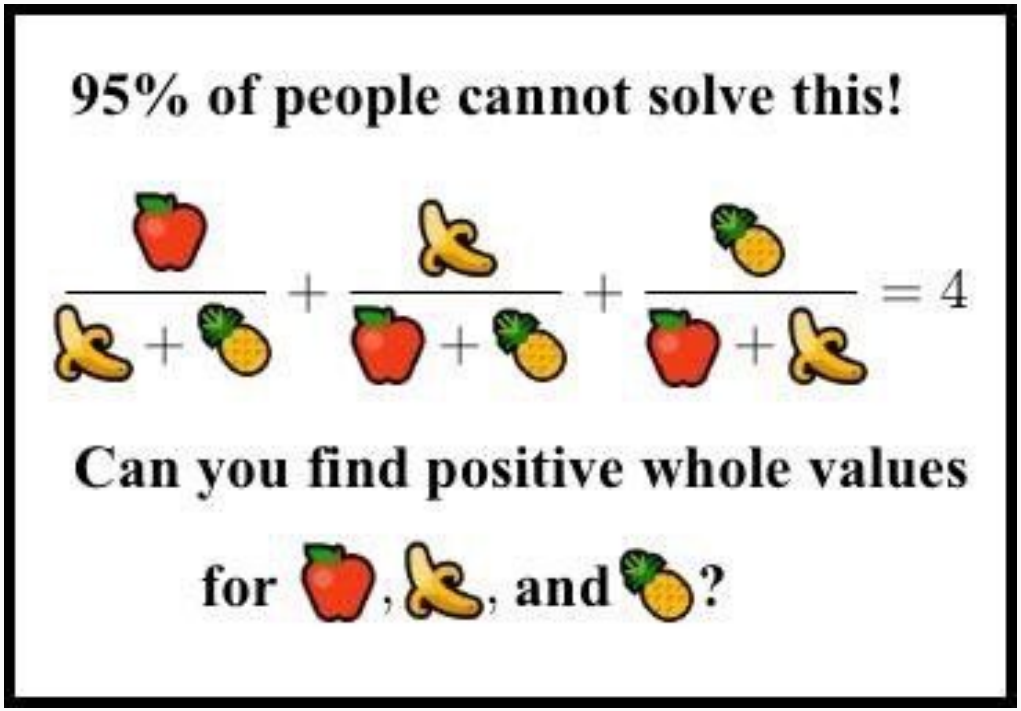
95% of people cannot solve this!



Can you find positive whole values for , , and ?

check this

Warm-up



check this

Legkisebb megoldások

alma: 154476802108746166441951315019919837485664325669565431700026634898253202035277999

banán: 368751317941299998271978115652254748254929799689719709962831374716372246340555 79

ananász: 437361267792869725786125260237139015281653755816161361862143799337842346777 2036

aki akar, megpróbálhat rá programot írni, good luck

A kiszámíthatóság elméletének kialakulása

1900: Hilbert 10. problémája (a 23-ból)

Adott $p(x_1, \ldots, x_n)$ egész együtthatós polinom, létezik-e (egész értékű) zérushelye.

pl: $x_1^3 + x_2^3 + x_3^3 - 3x_1^2x_2 - 3x_1x_2^2 - 3x_1^2x_3 - 3x_1x_3^2 - 3x_2^2x_3 - 3x_2x_3^2 - 5x_1x_2x_3$.

A kiszámíthatóság elméletének kialakulása

1900: Hilbert 10. problémája (a 23-ból)

Adott $p(x_1, \ldots, x_n)$ egész együtthatós polinom, létezik-e (egész értékű) zérushelye.

🍎🍎🍎+🍌🍌🍌+🍍🍍🍍 = 3🍎🍎🍌+3🍎🍌🍌+3🍎🍎🍍+3🍎🍍🍍+3🍌🍌🍍+3🍌🍍🍍+5🍎🍌🍍.

Gyümölccsel minden jobb.

1928: Hilbert

Találjunk olyan algoritmust, amellyel a predikátumkalkulus tetszőleges formulájáról eldönthetjük, hogy tautológia-e.

A kiszámíthatóság elméletének kialakulása

1900: Hilbert 10. problémája (a 23-ból)

Adott $p(x_1, \ldots, x_n)$ egész együtthatós polinom, létezik-e (egész értékű) zérushelye.

pl: $x_1^3 + x_2^3 + x_3^3 - 3x_1^2x_2 - 3x_1x_2^2 - 3x_1^2x_3 - 3x_1x_3^2 - 3x_2^2x_3 - 3x_2x_3^2 - 5x_1x_2x_3$.

1928: Hilbert

Találjunk olyan algoritmust, amellyel a predikátumkalkulus tetszőleges formulájáról eldönthetjük, hogy tautológia-e.

Algoritmusból sokkal kevesebb (megszámlálhatóan végtelen) van, mint eldöntési problémából (kontinuum) – Cantor, 1874 –, emiatt a problémák túlnyomó többsége megoldhatatlan.

A kiszámíthatóság elméletének kialakulása

Mi az, hogy valami kiszámítható?

1931, Gödel: Primitív rekurzív függvények

- $f(n) = 0$; RETURN 0
- $s(n) = n + 1$; az input növelése eggyel
- $\pi_k^i(x_1, \ldots, x_k) = x_i$; „projekció” = tömbelem-kiválasztás
- $f/n,\ g_1/k, \ldots, g_n/k \Rightarrow h(\boldsymbol{x}) = f(g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$; „kompozíció”

  „az outputo(ka)t odaadjuk egy másik kiszámítható függvénynek inputként”
- $f/k,\ g/(k+2) \Rightarrow$ „primitív rekurzió”

$$h(0, \boldsymbol{x}) = f(\boldsymbol{x}),$$

$$h(n+1, \boldsymbol{x}) = g(n, h(n, \boldsymbol{x}), \boldsymbol{x}).$$

Intuitíve világos, hogy ezek tényleg „kiszámítható” függvények, könnyű rájuk mai programozási nyelven is kódot írni, mely biztos, hogy mindig meg is áll

(a primitív rekurzív hívások is, mert n leszállási feltételként működik)

Néhány primitív rekurzív függvény

Az utolsó konstruktor magyarul „rekurzívan hívjuk a függvényt eggyel kisebb paraméterre, eztán az eredményből, a paraméterből és az input további elemeiből kiszámítjuk az aktuális értéket”.

Példák

- $\mathrm{add}(0, x) = x$, $\mathrm{add}(n+1, x) = \mathrm{add}(n, x) + 1$ – összeadás
  (adjuk össze az eggyel kisebb n-t az x-szel – rekurzív hívás – majd az eredményhez adjunk egyet)
- $\mathrm{mult}(0, x) = 0$, $\mathrm{mult}(n+1, x) = \mathrm{add}(\mathrm{mult}(n, x), x)$ – szorzás
- $\mathrm{pow}(0, x) = 1$, $\mathrm{pow}(n+1, x) = \mathrm{mult}(\mathrm{pow}(n, x), x)$ – hatványozás
- faktoriális, különbség, egyenlőség tesztelés, előjel, egészosztás, maradék, stb

Ha csak ezt a fajta rekurziót támogatjuk, az tényleg elég „primitív”.

Primitív rekurzió ≈ for ciklus

```
int h( int n, int x1, int x2, .., int xk )
{
  int ret = f( x1, x2, ..., xk ); // alapeset: 0-ra
  // fontos: előre ismert, hogy hányszor fut le!
  for( int i = 0; i < n; i++ )
  {
    ret = g( i, ret, x1, x2, ..., xk );
  }
  // nem is kellett hozzá rekurzió, és plusz tár se sok
  return ret;
}
```

Primitív rekurzió ≈ for ciklus

Ezt csináltuk az előbb

```
int add( int n, int x )
{
  // projekció
  int ret = x;
  for(int i = 0; i < n; i++)
  {
    // inkrementálás
    ret = ret + 1;
  }
  // összeadtuk
  return ret;
}
```

```
int mult( int n, int x )
{
  // konstans 0
  int ret = 0;
  for(int i = 0; i < n; i++)
  {
    // addjuk hozzá
    ret = add( ret, x );
  }
  // összeszoroztuk
  return ret;
}
```

```
int pow( int n, int x )
{
  // kompozíció:
  // 0, rákövetkezés
  int ret = 0 + 1;
  for(int i = 0; i < n; i++)
  {
   // szorozzunk rá
    ret = mult( ret, x );
  }
  return ret; // done
}
```

láthatjuk: primitív rekurzióval előálló függvényeket tényleg nagyon egyszerűen, sablonosan (és rekurzió-mentesen) implementálhatunk

Példa: Ackermann-függvény

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

pl. $A(1, 1) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) = A(2,A(3,2)) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) = A(2,A(3,2)) = A(2,A(2,A(3,1))) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) = A(2,A(3,2)) = A(2,A(2,A(3,1))) = A(2,A(2,A(2,A(3,0)))) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) = A(2,A(3,2)) = A(2,A(2,A(3,1))) = A(2,A(2,A(2,A(3,0)))) = A(2,A(2,A(2,A(2,1)))) =$

Példa: Ackermann-függvény

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

pl. $A(1,1) = A(0,A(1,0)) = A(0,A(0,1)) = A(0,2) = 3.$

pl. $A(3,3) = A(2,A(3,2)) = A(2,A(2,A(3,1))) = A(2,A(2,A(2,A(3,0)))) =$
$A(2,A(2,A(2,A(2,1)))) = A(2,A(2,A(2,A(1,A(2,0))))) = A(2,A(2,A(2,A(1,A(1,1))))) =$
$A(2,A(2,A(2,A(1,A(0,A(1,0)))))) = A(2,A(2,A(2,A(1,A(0,A(0,1)))))) =$
$A(2,A(2,A(2,A(1,A(0,2))))) = A(2,A(2,A(2,A(1,3)))) = A(2,A(2,A(2,A(0,A(1,2))))) =$
$A(2,A(2,A(2,A(0,A(0,A(1,1)))))) = A(2,A(2,A(2,A(0,A(0,A(0,A(1,0))))))) =$
$A(2,A(2,A(2,A(0,A(0,A(0,A(0,1))))))) = A(2,A(2,A(2,A(0,A(0,A(0,2)))))) =$
$A(2,A(2,A(2,A(0,A(0,3))))) = A(2,A(2,A(2,A(0,4)))) = A(2,A(2,A(2,5))) = \ldots$

A kiszámíthatóság elméletének kialakulása

- $A(5,1)$ már egy sokkal nagyobb szám, mint ahány atom van az ismert univerzumban.
- Az Ackermann-függvény nem primitív rekurzív függvény: gyorsabban növekszik, mint amit el lehet érni a primitív rekurzió konstruktoraival.
- Első ránézésre talán nem primitív rekurzió, mert a harmadik esetben a belső hívás n mentén csökken, míg a többi hívásnál m csökken.
- Ettől még lehet, hogy fel lehetne írni primitív rekurzív alakban! De be van bizonyítva, hogy nem lehet.

A kiszámíthatóság elméletének kialakulása

Mi az, hogy valami kiszámítható?

1934, Gödel: Általános rekurzív függvények

A primitív rekurzív függvények konstrukciói, plusz:

- $f/(k+1) \Rightarrow h(\boldsymbol{x}) = \min_{n \geq 0}\{n : f(n, \boldsymbol{x}) = 0\}$.

az általános (parciális) rekurzív függvények.

- Magyarul „keressük meg azt a legkisebb n-t, amire az eredeti függvényünk 0-t vesz fel, ha megkapja n-t és magát az inputot".
- Ez intuitíve kiszámítható: indítunk egy while ciklust 0-ról és kiszámolgatjuk $f(0, \boldsymbol{x})$-et, $f(1, \boldsymbol{x})$-et stb, amelyikre először 0 lesz, azt az n-t visszaadjuk.
- „Parciális", mert ha egy $\boldsymbol{x}$-re $f(n, \boldsymbol{x})$ sose lesz 0, akkor a függvény nem definiált.

Az Ackermann-függvény is általános rekurzív függvény.

a defje ugyan nem úgy néz ki, de átírható ekvivalens, általános rekurzív alakúra

Általános rekurzió ≈ while ciklus

```
int h(int x1, int x2, ..., int xk)
{
  // minimumkeresünk 0-ról
  int n = 0;
  // ez akár végtelen ciklusba is eshet!
  // nem tudni feltétlen előre, hányszor fut!
  while( f( n, x1, x2, ..., xk ) != 0 )
  {
    // ha erre se 0, növeljük
    n = n + 1;
  }
  // 0 lett az f
  return n;
}
```

A kiszámíthatóság elméletének kialakulása

Mi az, hogy valami kiszámítható?

- 1931, Gödel: Primitív rekurzív függvények — Ackermann: ez még kevés
- 1934, Gödel: Általános rekurzív függvények — Ackermann: hold my beer.. wait
- 1930-as évek eleje, Church, Kleene, Rosser (Princeton): λ-definiálhatóság

1935: AMS New York-i összejövetele

Church tézise: Az általános rekurzív függvények (matematikai fogalom) megfelelnek az algoritmikusan kiszámítható függvény fogalmának (intuitív fogalom).

„Tézis”: egy real-life fogalmat köt össze egy matematikaival, így nem lehet „bebizonyítani”, csak elfogadni

1936: Kleene tétele

Az általános rekurzív függvények megegyeznek a λ-definiálható függvényekkel.

„Tétel”: csak matematikai fogalmakról beszél + be is van bizonyítva, vitathatatlan

A kiszámíthatóság elméletének kialakulása

1936: Turing tétele és tézise

Bebizonyítja, hogy a Turing-gép

- https://www.youtube.com/watch?v=gJQTFhkhwPA
- https://www.youtube.com/watch?v=E3keLeMwfHY
- https://www.youtube.com/watch?v=FTSAiF9AHN4

ekvivalens a λ-definiálhatósággal (tétel), és megfogalmazza azt a tézist, hogy a Turing-gép megfelel az algoritmussal kiszámítható függvényeknek.

(persze Turing tétele és Kleene tétele miatt ez a tézis pont ugyanazt jelenti, mint Church tézise, ezért a neve ma „Church-Turing tézis")

A Church-Turing tézist tapasztalati tények és matematikai eredmények támasztják alá:

- Minden intuitív értelemben megoldható problémáról sikerült kimutatni, hogy azok megoldhatók a matematikai modellekben is.
- A matematikai modellek ekvivalensek.

A kiszámíthatóság elméletének kialakulása

Ezek után. . .

most már, hogy mindenki megegyezett abban, hogy mi az matematikailag, hogy valami „kiszámítható”, neki lehet állni bebizonyítani, ha valami nem az

1936, ’37: Church, Turing

Nem létezik olyan algoritmus, mely a predikátumkalkulus tetszőleges formulájáról eldöntené, hogy tautológia-e.

1971: Matijasevič

Hilbert 10. problémája algoritmikusan megoldhatatlan.

Kiszámíthatóság és bonyolultság

Kiszámíthatóságelmélet

Melyek az algoritmikusan megoldható problémák?

Bonyolultságelmélet

Melyek a gyakorlatilag megoldható problémák? (erőforrásigény!)

Intuitíve az Ackermann-függvény elméletben kiszámítható, de gyakorlatban nem az

1971: Cook

P és **NP** osztályok, **NP**-teljesség, SAT **NP**-teljes

1972: Karp

Rámutat az **NP**-teljes problémák változatosságára.

1973: Levin

Több kombinatorikus probléma „univerzális a kimerítő keresésre”.

További bonyolultsági osztályok és számítási módok

A kurzuson a két központi (és gyakorlati szempontból is legmotiváltabb) osztályunk a $\mathbf{P}$ és az $\mathbf{NP}$ problémaosztály lesznek, de lesz szó másról is

- $\mathbf{L}$, $\mathbf{NL}$, $\mathbf{PSPACE}$, $\mathbf{EXP}$, ...
- Valószínűségi modellek — kriptográfiában pl nagyon hasznos
- Párhuzamos számítás — a GPU programozás is motivált dolog

stb.

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$					

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$	14				

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$	14	16			

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$	14	16	19		

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$	14	16	19	22	

Mit jelent az időigény?

Az alábbi táblázat megadja, hogy adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut még le egy napon belül.

	 C64 1Mflops	 Cray Y-MP 1Gflops	 mai GPU 5TFlops	 Tianhe-2 34PFlops	 Föld bolygó 10EFlops
n	86.4G	8.64×10^{13}	4.32×10^{17}	2.94×10^{21}	8.64×10^{23}
$n \log n$	2.75G	2.1×10^{12}	8.1×10^{15}	4.5×10^{19}	1.17×10^{22}
n^2	300k	9.3M	657M	54G	929G
n^3	4420	44.208	756k	14.3M	95M
2^n	36	46	58	71	79
1.1^n	264	336	426	518	578
$n!$	14	16	19	22	24

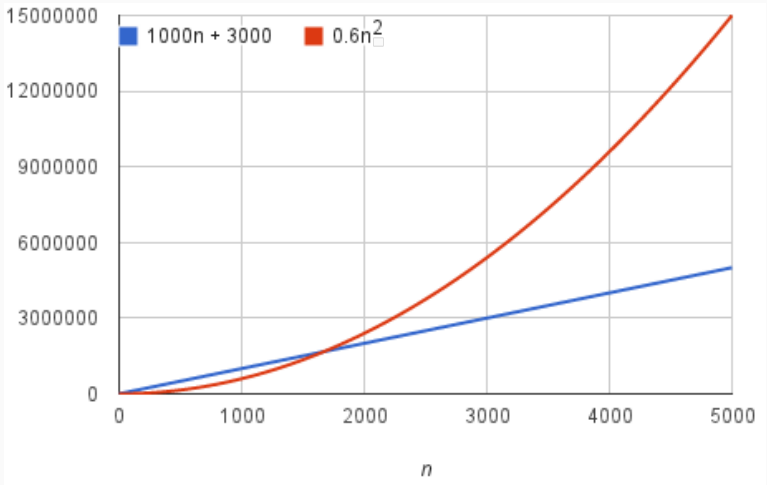
Amíg Moore törvénye (nagyjából) igaz, addig a polinomidejű algoritmusok egy-egy újabb év elteltével konstansszor több adattal tudnak adott időn belül elbánni.

Aszimptotikus jelölések

Legyenek $f, g : \mathbb{N} \to \mathbb{N}$ függvények.

- $f(n) = \mathcal{O}(g(n))$, ha létezik olyan $c > 0$, hogy $f(n) \leq c \cdot g(n)$ majdnem minden n-re.
- $f(n) = \Omega(g(n))$, ha $g(n) = \mathcal{O}(f(n))$.
- $f(n) = \Theta(g(n))$, ha $f(n) = \mathcal{O}(g(n))$ és $g(n) = \mathcal{O}(f(n))$.

Ha $f(n) = \mathcal{O}(g(n))$, de $g(n) \neq \mathcal{O}(f(n))$, annak jele $f(n) = o(g(n))$. (és ω hasonlóan.)



$$1000n + 3000 = \mathcal{O}(0.6n^2)$$

A kép forrása: Khan Academy

Aszimptotikus jelölések

Határértékkel általában könnyebb

Ha $\lim_{n\to\infty} \frac{f(n)}{g(n)} \ldots$

- $\ldots = 0$, akkor $f(n) = o(g(n))$
- $\ldots < \infty$, akkor $f(n) = \mathcal{O}(g(n))$

Példa

Ha $p(n)$ és $q(n)$ polinomok (pozitív főegyütthatóval), akkor

- $p(n) = \mathcal{O}(q(n))$ pontosan akkor, ha p fokszáma legfeljebb akkora, mint q-é;
- $p(n) = \Theta(q(n))$ pontosan akkor, ha fokszámaik megegyeznek.

Legyen $a \in \mathbb{N}$, $a > 1$. Ekkor $p(n) = o(a^n)$.

Nagyobb fokú polinom jobban nő, exponenciális még jobban.

A P osztály, informálisan

Egy probléma a $\mathbf{P}$ osztályba esik, ha megoldható polinom időigényű (vagy $\mathcal{O}(n^k)$ időigényű) algoritmussal.

Ezt még később pontosítjuk: mi is az, hogy „algoritmus időigénye”?

A Cobham – Edmonds tézis

- Egy algoritmus akkor ad gyakorlatilag kielégítő megoldást egy problémára, ha polinom időigényű.
- Egy problémát akkor tekintünk gyakorlatilag megoldhatónak, ha a $\mathbf{P}$ osztályba esik.

Egy tézist persze lehet vitatni is

Néhány ellenvetés

- Elfogadható-e egy $\mathcal{O}(n^{100})$ időigényű algoritmus?
- A konstansok nagysága.
- Kisméretű adatokon egy exponenciális algoritmus is jobban viselkedhet, mint egy polinomiális.
- Legrosszabb eset helyett a várható viselkedés vizsgálata is indokolt lehet.
- A $\mathbf{P}$ osztályba eső egyes problémák hatékonyan párhuzamosíthatók, míg mások (valószínűleg) nem.

Ugyanakkor

- A gyakorlatban előforduló $\mathbf{P}$-beli problémák rendje kicsi.
- A $\mathbf{P}$ osztály elegáns matematikai elmélethez vezet.

A kiszámítás modelljei

De milyen architektúrán polinom?

A kiszámításnak számos (matematikai) modellje létezik:

- Általános rekurzív függvények
- λ-kalkulus
- Turing-gép
- RAM (Random Access Machine)

Ezek mind „ekvivalensek” abból a szempontból, hogy ugyanazokat a függvényeket lehet velük kiszámítani (azaz ugyanazokat a problémákat lehet velük megoldani)

A kurzuson a RAM gépet (kb „pszeudokódot”, de formálisan definiáljuk az elemi utasításokat, hogy tudjunk időigényt számolni) fogjuk használni.

A RAM gép

RAM gép

- rendelkezik végtelen sok regiszterrel
- a regiszterek típusai: Input, Output és Working (munka-)regiszterek, minden típus regiszterei 0-tól indexeltek
- minden regiszter egy nemnegatív egész számot tartalmaz
- címzési módok:
  - k – konstans
  - $I[k]$, $O[k]$, $W[k]$ – direkt
  - $W[W[k]]$, $W[I[k]]$ stb. – indirekt

A RAM gép

RAM gép

- program: utasítások egy véges sorozata
- minden utasítás egy sorszám és egy elemi művelet

  (egy sorszám legfeljebb egy utasításhoz tartozhat – egyelőre)
- elemi műveletek:
  - $X := Y$ – értékadás
    - X itt egy direkt vagy egy indirekt címzés, pl. $O[7]$ vagy $W[I[3]]$
    - Y két címzés összege vagy különbsége ($a - b$ eredménye 0, ha $a < b$)
  - $\mathrm{JZ}\ r, \ell$ – ha $r == 0$, ugrás az ℓ. sorszámú utasításra
  - $\textsc{Halt}$, $\textsc{Accept}$, $\textsc{Reject}$ – megállás
- a programszámláló (PC) pozitív egész, 0-ról indul
  - a gép egy lépésben végrehajtja azt az utasítást, melynek sorszáma a PC tartalma
  - a feltételes ugrást kivéve minden lépés után eggyel nő a PC
  - ha nincs ilyen sorszámú utasítás, a számítás megáll

A RAM gép

- a számítás tetszőleges pillanatában csak véges sok regiszter értéke nem 0

  ez azért jó, mert akkor lehet „igazi” architektúrán is szimulálni

- inicializálás:
  - az input az $I[1], I[2], \ldots, I[m]$ számsorozat, ahol $I[m+1]$ az első 0-t tartalmazó input regiszter
  - a többi regiszter tartalma 0
- az output
  - Accept/Reject, ha a megállás egy Accept/Reject utasítás végrehajtásakor következett be
  - az $O[1], O[2], \ldots, O[k]$ számsorozat, ahol $O[k+1]$ az első 0-t tartalmazó output regiszter, egyébként

Az M Ram gép az $a_1, \ldots, a_m$ inputon számított outputját $M(a_1, \ldots, a_m)$ jelöli; ha M nem áll meg ezen az inputon, annak jele $M(a_1, \ldots, a_m) = \nearrow$.

Feltételek, ugrások

Persze a példáinkban, konstrukcióinkban nem mindent közvetlenül ezekből az elemi műveletekből rakunk össze: a következő pár fólián látunk néhány példát, hogyan lehet RAM gépen strukturáltan programozni

if $L == R$ goto Y

1. $X := L + 1$
2. $X := X - R$
3. if $X == 0$ goto 6 — ha $L < R$, itt ugrunk a 6-ra, azaz „simán megyünk tovább”
4. $X := X - 1$
5. if $X == 0$ goto Y — ha $L > R$, itt nem ugrunk el Y-ra, hanem „simán megyünk tovább” 6-ra

ahol X egy másra nem használt munkaregiszter.

Feltétel nélküli ugrás, $<$, $>$, $\leq$, $\geq$ relációk szerinti feltételek hasonló módon

az ilyen alakú if-goto feltételes ugrás ugyan nem 1, de konstans sok lépés

Stack memória, függvényhívások

Ha egy SP munkaregisztert (pl $W[0]$-t) kinevezünk stack pointernek, és függvényeinket mindig rögzített paraméterszámmal definiáljuk, akkor lokális változókat használó rekurzív (most: „önmagukat hívó") függvényeket is írhatunk a megszokott módon

- $W[\mathrm{SP}] := k$, ahol k az a programsor, ahová a függvényhívás után vissza kell térnünk
- SP egyesével növelése mellett a paramétereket sorban bemásoljuk $W[\mathrm{SP}]$-be
- ugrunk a függvény belépési pontjára
- a függvény a verem tetején megtalálja a paramétereit, a saját lokális változóit ezek tetejére hozza létre
- visszatéréskor a megfelelő értékkel (paraméterszám+lokális változók száma) csökkenti SP-t és a megfelelő, a veremben letárolt címre ugrik vissza

Ha az argumentumokat már kiszámoltuk, akkor a call+return is konstans sok lépés

Heap memória is kell?

A munkaregiszterek kettéosztásával (pl. páratlan sorszámúak a stack, párosak a heap memóriához tartoznak) többféle memóriát is lehet kezelni

- kinevezünk egy regisztert heap counternek
- dinamikus memóriafoglalásnál visszaadjuk a heap counter értékét mint „címet” és megnöveljük a heap countert a foglalni kívánt mérettel
- memória-felszabadítás: nincs rá szükség végtelen sok munkaregiszter van

ebben a modellben így a dinamikus memóriafoglalás is megy konstans időben

Strukturált programozás

while F do (R)

1. if $F == 0$ goto 4
2. R
3. goto 1

a ciklusok időigénye függ attól, hogy hányszor is fut le a ciklus

do (R) while F

1. R
2. if $F == 0$ goto 4
3. goto 1

ha legfeljebb X-szer fut a ciklus és a magja legfeljebb Y lépés, akkor az egész max $X \cdot Y$ lépésben lefut

for $i := 1 \ldots n$ do (R)

1. $i := 1$
2. while $i \leq n$ do (
3. R
4. $i := i + 1$)

kényelmi okokból olhasható változóneveket használunk $W[42]$-like nevek helyett

Legmagasabb bit

function HIGH(n) — pl. $\text{HIGH}(42) = 32$

  if $n == 0$ **then return** 0

  $a := 1$

  while $a + a \leq n$ **do** $a := a + a$

  return a — a lépésszám $\log n$-nel arányos!

Paritás

function ODD(n)

  $a := 0$

  while $a \neq n$ **do** — minden iterációban eggyel kevesebb lesz az 1-es n-ben (bináris rep)

    $n := n - a$ — $\Rightarrow$ max $\log n$-szer fut le

    $a :=$HIGH(n) — a ciklusmag $\log n$-nel arányos lépésszámot igényel

  if $a == 1$ **then return** 1

  return 0

ez így $\log n \cdot \log n = \log^2 n$-nel arányos lépésszám lesz

Felezés

Az előző kettőhöz hasonlóan szintén $\log^2 n$ lépésben megvan: rendre kiszámoljuk a legmagasabb bitet, kivonjuk, és a felét hozzáadogatjuk egy akkumulátorhoz

Szorzás

function $\text{MULT}(a,b)$ kb mint az áltsulis papíron szorzás, gyorsabb, mint b-szer összeadni a-t

$r := 0$

while $b > 0$ **do** ez max $\log b$-szer fut le

if $\text{ODD}(b)$ **then** $r := r + a$ ez $O(\log^2 b)$ lépés

$b := b/2$ ez is $O(\log^2 b)$ lépés

$a := a + a$ ez $O(1)$ lépés

return r

Ciklusmag összesen $O(\log^2 b)$ lépés, $O(\log b)$-szer fut $\Rightarrow$ ez így lefut $O(\log^3 b)$ lépésben

ezzel szemben b-szer összeadni a-t $O(b)$ lépés lenne, ami sokkal több

Aritmetika

Duplázás / shift left

$$R[i] \;:=\; R[i] + R[i]$$

2^k előállítása

Inicializálás 1-gyel, majd k darab shift left – k lépés (!nem $\log k$!)

Az $\{1, \ldots, k\}$ halmaz részhalmazaival végzett műveletek

- k darab regiszterben, mindegyikben 0 vagy 1 a karakterisztikus függvénynek megfelelően
- elem beszúrása, törlése, lekérdezése konstans időben

Ami fontos: pszeudokódban / RAM gépen ha számokkal dolgozunk, melyeknek az értéke legfeljebb N, azokat lehet összeadni, kivonni, szorozni, osztani $\log^k N$ lépésben valamilyen k konstansra (ezt úgy hívják, hogy a lépésszám polilogaritmikus N-ben), de például hatványozni már nem

Problémák eldöntése defek

Eldöntési problémán egy tetszőleges $L \subseteq \mathbb{N}^*$ halmazt értünk.

(azaz számsorozatok egy halmazát. Aki eleme a halmaznak, az „igen" példánya, aki nem, az „nem" példánya a problémának.)

Az M RAM-gép eldönti az L problémát, ha tetszőleges $I = a_1, \ldots, a_m$ inputra

$$M(I) = \begin{cases} \text{ACCEPT} & \text{, ha } I \in L; \\ \text{REJECT} & \text{, egyébként.} \end{cases}$$

azaz: az algoritmus eldönti/megoldja a problémát, ha minden inputon megáll és helyes választ ad: „igen" példányokra ACCEPT, „nem" példányokra REJECT válasszal áll meg.

Az L probléma eldönthető, ha van őt eldöntő RAM-gép.

Eldönthető = algoritmussal megoldható

R jelöli az összes eldönthető probléma osztályát.

Az input mérete és a futásidő: defek

Futásidő egy adott inputon: lépésszám

Az M RAM gép időigénye az $a_1, \ldots, a_m$ inputon a megállásig végrehajtott utasítások száma, ha M megáll $a_1, \ldots, a_m$-en; egyébként a gép nem áll meg és időigénye végtelen

Az input mérete

Az $a_1, \ldots, a_m$ input mérete az a_i számok bináris reprezentációinak hosszának összege (azaz $\sum_{i=1}^{m}(1 + \lfloor \log a_i \rfloor)$).

Általában n jelöli az input hosszát.

A gép időkorlátja

Az M RAM gép időkorlátja az $f(n) : \mathbb{N} \to \mathbb{N}$ függvény, ha tetszőleges n méretű inputon legfeljebb $f(n)$ lépésben megáll.

A P osztály

$\mathbf{P}$ jelöli az összes polinomidőben eldönthető probléma osztályát, azaz melyek eldönthetőek $\mathcal{O}(n^k)$ időkorlátos RAM-géppel, valamely konstans k-ra.

Ha RAM-gép helyett Turing-géppel definiálnánk az $\mathbf{R}$ és a $\mathbf{P}$ osztályokat, akkor is ugyanezeket az osztályokat kapnánk: a bonyolultságelmélet független a választott architektúrától.

(Mindaddig, amíg az architektúra realisztikus.)

(Pl. pontosan azokat a problémákat lehet polinomidőben megoldani Java nyelven, melyeket RAM-gépen is.)

A továbbiakban problémáink inputjainak nem csak számsorozatokat, de gráfokat és egyéb véges matematikai struktúrákat is megengedünk; ez nem gond, mert minden ilyen struktúra kódolható számsorozattal.

Polinom időben eldönthető problémák

Elérhetőség

- Adott: egy $G = (V, E)$ irányított gráf.

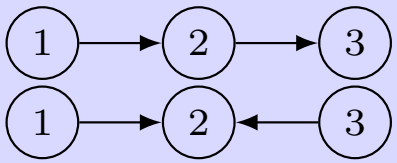
  Feltehetjük, hogy $V = \{1, 2, \ldots, n\}$.

- Kérdés: létezik-e 1-ből n-be vezető irányított út?

Algoritmus

1. $S := \{1\}$.
2. Jelöljük meg az 1 csúcsot.
3. Amíg S nem üres:
   - 3.1 Választunk egy $i \in S$ csúcsot.
   - 3.2 $S := S - \{i\}$.
   - 3.3 $j = 1..n$:
     Ha $(i, j) \in E$ és j nem megjelölt, akkor
     $S := S \cup \{j\}$ és jelöljük meg j-t.

Ha n megjelölt csúcs lett, Accept, különben Reject.



egy „igen” példány

egy „nem” példány

ezek a gráfok az Elérhetőség problémának az igen/nem példányai, nem pedig az algoritmusnak!

Polinom időben eldönthető problémák

Néhány kimaradt részlet

- A bemenet megadása – pl. szomszédsági mátrix
  (Függ a modelltől, de lényeges szerepet nem játszik.)
- S reprezentálása
  - sor: szélességi keresés
  - verem: egyfajta mélységi keresés

Hatékonyság

- A mátrix minden elemét legfeljebb egyszer használjuk fel.
- Ha az egyszerű műveletek (S egy elemének kiválasztása, megjelölése, stb.) konstans időigényűek, akkor $\mathcal{O}(n^2)$.

Függvényproblémák: defek

Függvényprobléma egy tetszőleges $f : \mathbb{N}^* \to \mathbb{N}^*$ leképezés.

„Számsorozatból készít számsorozatot”

de bármilyen objektumból bármilyen objektumot készíthet, miután megegyezünk egy alkalmas kódolásban

Az M RAM-gép az f függvényproblémát számítja ki, ha $M(I) = f(I)$ tetszőleges I inputra.

Az f függvényprobléma rekurzív, ha van őt kiszámító RAM-gép.

ez az „általános rekurzív”ból jön

Mint korábban, a rekurzív függvények osztálya sem változik, ha RAM-gép helyett (például) a Turing-gép lenne a választott architektúra.

Az utazóügynök probléma

TSP (travelling salesman problem) újabban travelling salesperson problem /shrug

- Adott: az $1, 2, \ldots, n$ városok és bármely két i, j város távolsága: $d_{i,j}$.
- Kérdés: találjunk egy, az összes várost pontosan egyszer érintő legrövidebb körutat.
  Eldöntési változata: TSP(E).

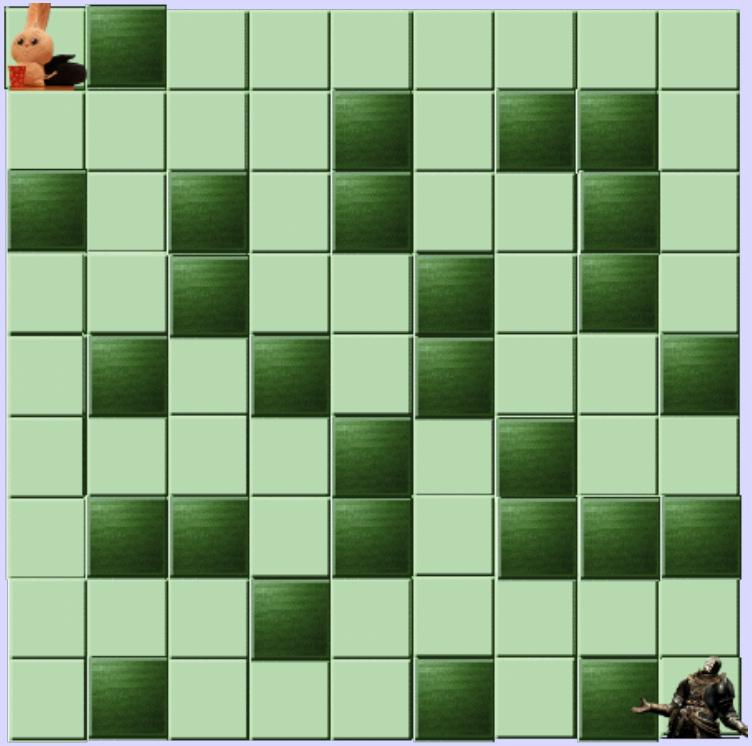
„eldöntési változat”: kapunk egy célszámot is és a kérdés, hogy van-e legfeljebb ekkora költségű körút

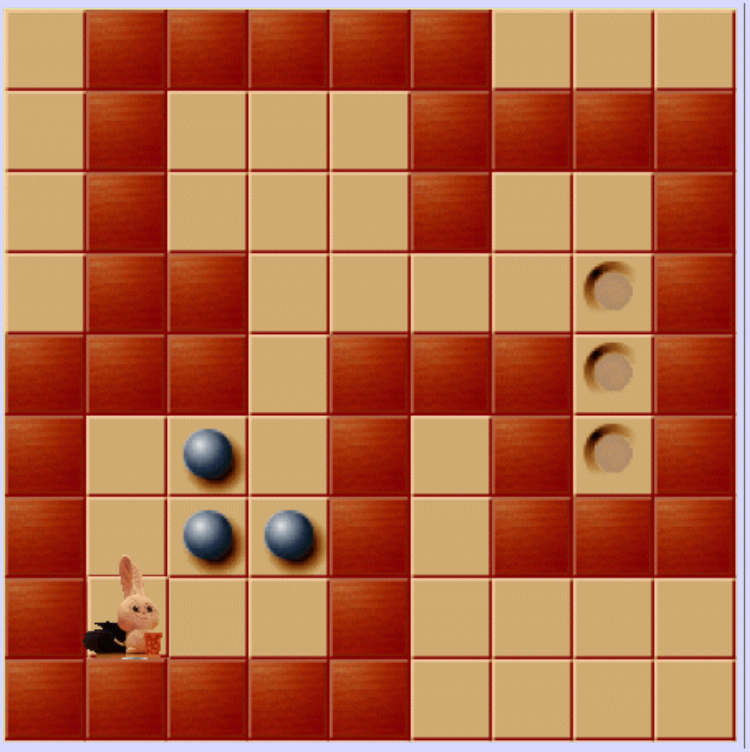
- Triviális módszer: az összes $\frac{(n-1)!}{2}$ lehetséges körút megvizsgálásával.
- Dinamikus programozással: $\mathcal{O}(2^n \cdot n^2)$ időigény

Nem ismert, hogy TSP megoldható-e polinom idejű algoritmussal.

Melyik a nehezebb?

Maze vs Sokoban





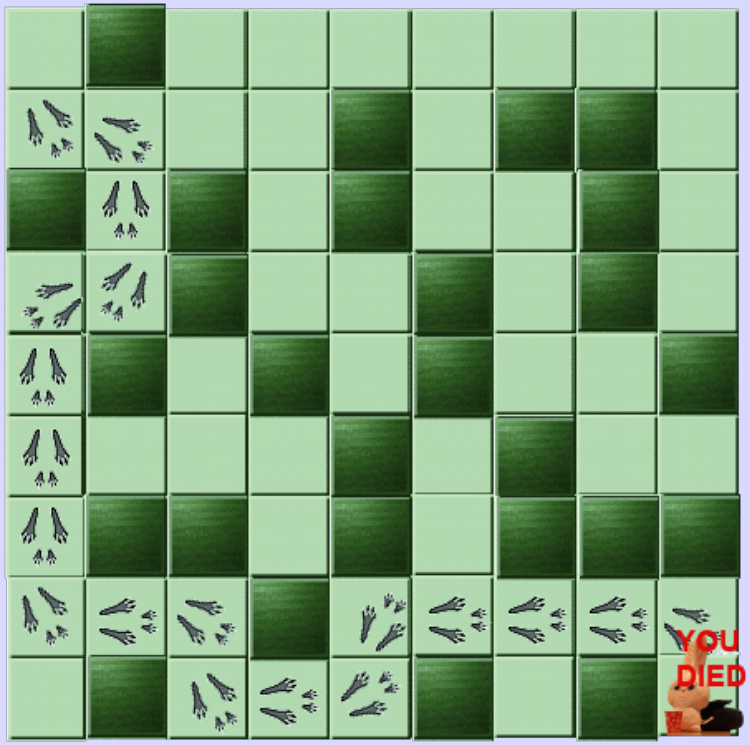
Úgy érzi az ember, hogy a bal oldalira van lineáris időigényű algoritmus, a jobb oldalira meg egy nagy keresési teret kell bejárni.

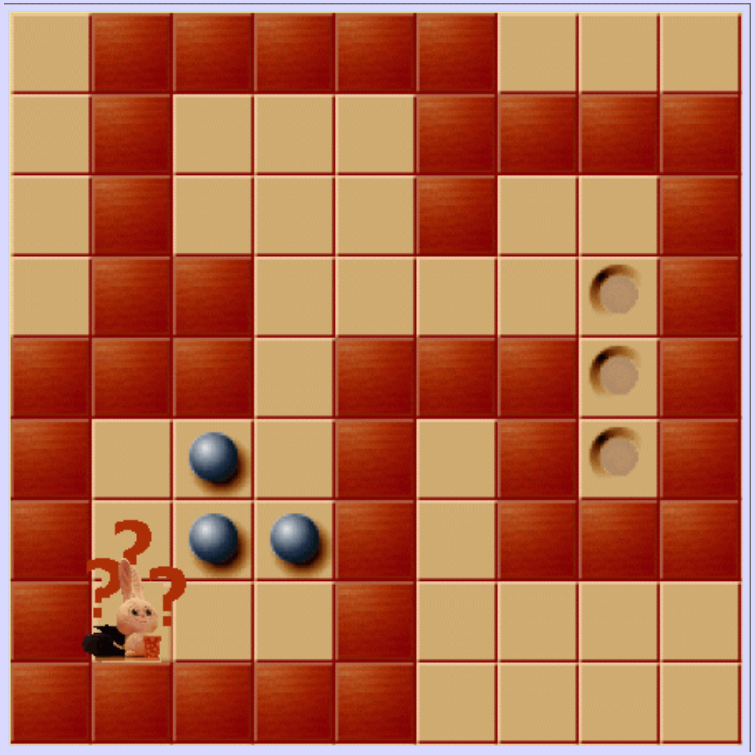
De ezzel két algoritmust hasonlítunk össze, nem két problémát!

(A SOKOBANra se talált még senki polinomidejű algoritmust. De ez nem feltétlen jelenti, hogy nincs is. . .)

Melyik a nehezebb?

Maze vs Sokoban





Úgy érzi az ember, hogy a bal oldalira van lineáris időigényű algoritmus, a jobb oldalira meg egy nagy keresési teret kell bejárni.

De ezzel két algoritmust hasonlítunk össze, nem két problémát!

(A SOKOBANra se talált még senki polinomidejű algoritmust. De ez nem feltétlen jelenti, hogy nincs is. . .)

Turing-visszavezetések

Turing-visszavezetés

Legyenek A és B eldöntési problémák.

B legalább olyan nehéz, mint A, ha létezik olyan hatékony módszer, azaz f rekurzív függvény, mely az A tetszőleges x bemenetéhez hozzárendeli a B egy $f(x)$ bemenetét (példányát) úgy, hogy az A-nak x akkor és csak akkor „igen" példánya, ha B-nek $f(x)$ „igen" példánya. ekkor az inputkonverzió „tartja a választ"

Jelben: $A \leq_R B$, A Turing-visszavezethető (vagy rekurzívan visszavezethető) B-re.

Ekkor:

- ha van egy B-t eldöntő algoritmusunk, akkor A-ra is van:

```
function A(x)
    return B(f(x));
```

- így tehát ha $A \leq_R B$ és A-ról tudjuk, hogy eldönthetetlen, akkor B is az kell legyen.

Hatékony visszavezetés

Az f függvényre további megszorításokat teszünk, hogy a visszavezetés fogalmunk ne csak „kiszámíthatóságra”, hanem „bonyolultságra” is mondjon valamit.

(Hatékony) visszavezetés

Azt mondjuk, hogy az A probléma (polinomidőben) visszavezethető a B problémára, ha létezik olyan polinomidőben kiszámítható f függvény, hogy minden x bemenetre

$$x \in A \Leftrightarrow f(x) \in B.$$

Jelben: $A \leq_{\mathcal{P}} B$.

azaz: van olyan gyors inputkonverzió A-ról B-re, mely tartja a választ.

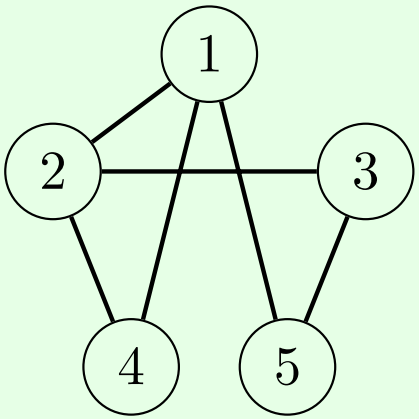
Ez lesz a default visszavezetésünk, ha valahol csak $\leq$ szerepel, az ezt fogja jelenteni.

Hamilton-út $\leq_{\mathcal{P}}$ Tsp(E)

Hamilton-Út

- Input: G irányítatlan gráf.
- Output: van-e G-ben Hamilton-út (minden csúcsot pontosan egyszer érintő út)?

Hamilton-út visszavezethető Tsp(E)-re.



$\Rightarrow$

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

és a célérték $C = 6$.

Tehát: $d_{i,j} = 1$, ha (i, j) él volt G-ben, 2 egyébként; a célérték $N+1$, ahol N a városok száma.
Ekkor ha van Hamilton-út G-ben, annak összköltsége $N - 1$; a két végpontját összekötve (ennek súlya 1 vagy 2) egy legfeljebb $N + 1$ súlyú körutunk van.

Másfelől, ha van egy legfeljebb $N + 1$ súlyú körutunk, abban legalább $N - 1$ él súlya 1 kell legyen; ezek Hamilton-utat adnak G-ben.

Párosítás $\leq_{\mathcal{P}}$ Sat

Párosítás

- Input: $G = (V, E)$ páros gráf.
- Output: van-e G-ben teljes párosítás?

Sat

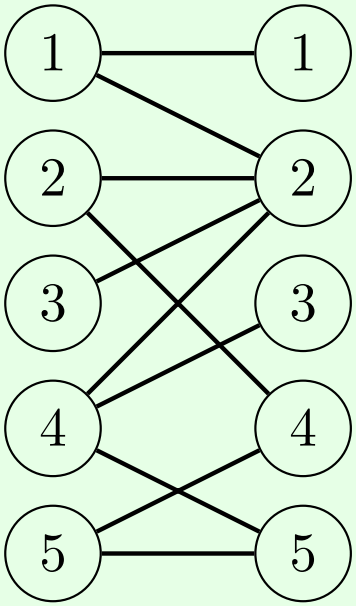
- Input: konjunktív normálformájú (ítéletkalkulus-beli) formula.
- Output: kielégíthető-e?

A visszavezetés

- Minden $(u, v) \in E$ élhez rendelünk egy $x_{u,v}$ logikai változót.
- Intuíció: $x_{u,v}$ akkor 1, ha (u, v) párosításbeli él, 0 egyébként.
- Minden u csúcsra felírjuk, hogy legalább egy rá illeszkedő élt kiválasztunk...
- ...és azt is, hogy legfeljebb egyet.
- Ez minden.

Párosítás $\leq_{\mathcal{P}}$ Sat

Példa

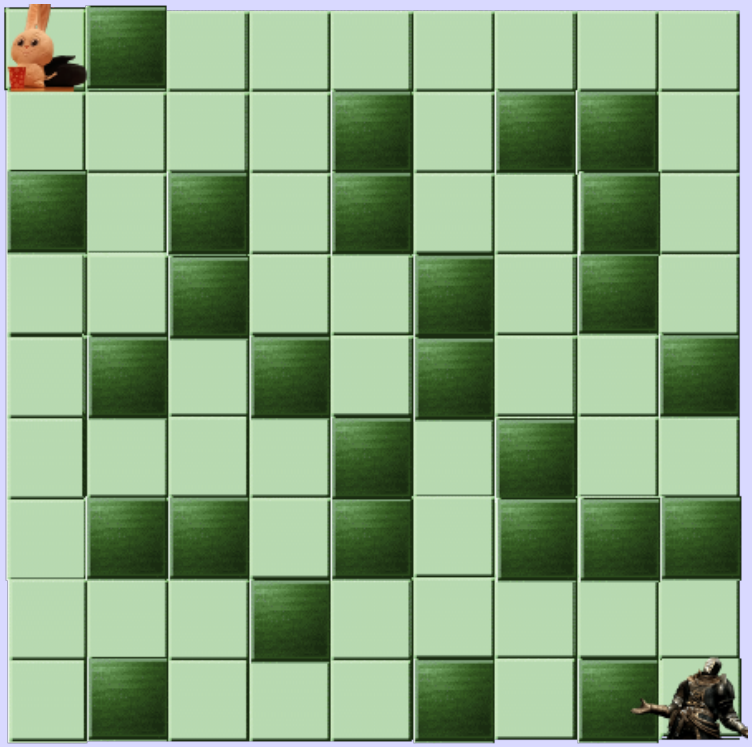


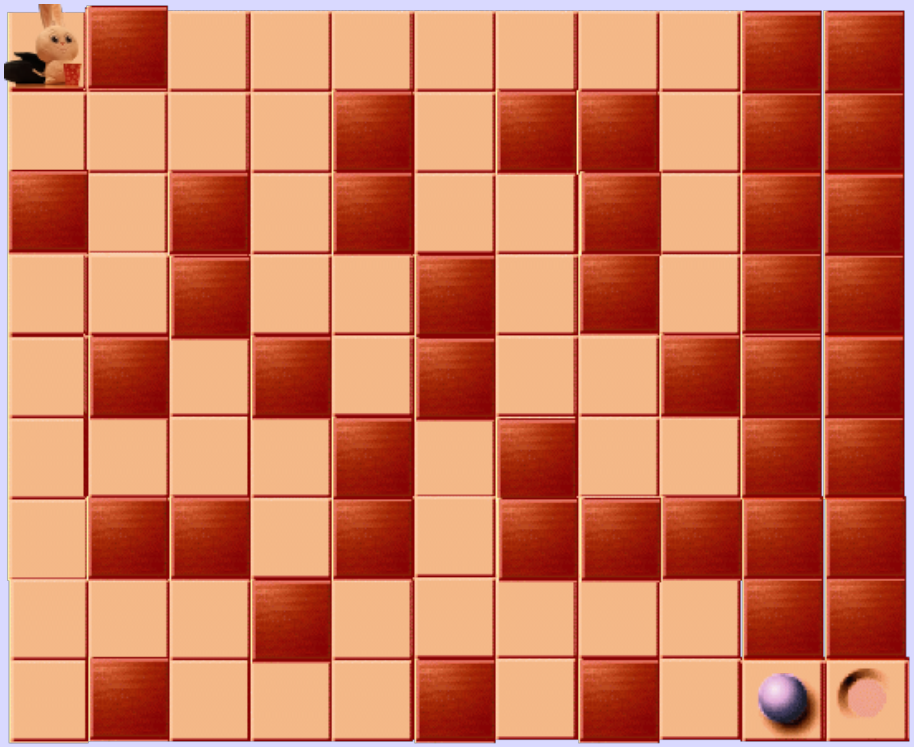
$$(x_{1,1} \vee x_{1,2}) \wedge (x_{2,2} \vee x_{2,4}) \wedge x_{3,2} \wedge (x_{4,2} \vee x_{4,3} \vee x_{4,5})$$
$$\wedge (x_{5,4} \vee x_{5,5}) \wedge x_{1,1} \wedge (x_{1,2} \vee x_{2,2} \vee x_{3,2} \vee x_{4,2})$$
$$\wedge x_{4,3} \wedge (x_{2,4} \vee x_{5,4}) \wedge (x_{4,5} \vee x_{5,5})$$
$$\wedge (\neg x_{1,1} \vee \neg x_{1,2}) \wedge (\neg x_{2,2} \vee \neg x_{2,4}) \wedge (\neg x_{4,2} \vee \neg x_{4,3})$$
$$\wedge (\neg x_{4,2} \vee \neg x_{4,5}) \wedge (\neg x_{4,3} \vee \neg x_{4,5}) \wedge (\neg x_{5,4} \vee \neg x_{5,5})$$
$$\wedge (\neg x_{1,2} \vee \neg x_{2,2}) \wedge (\neg x_{1,2} \vee \neg x_{3,2}) \wedge (\neg x_{1,2} \vee \neg x_{4,2})$$
$$\wedge (\neg x_{2,2} \vee \neg x_{3,2}) \wedge (\neg x_{2,2} \vee \neg x_{4,2}) \wedge (\neg x_{3,2} \vee \neg x_{4,2})$$
$$\wedge (\neg x_{2,4} \vee \neg x_{5,4}) \wedge (\neg x_{4,5} \vee \neg x_{5,5})$$

nice CNF

Visszatérve. . .

Maze $\leq_{\mathcal{P}}$ Sokoban





Jobb alsó sarok mellé rakni az egyetlen ládát, amellé a lyukat.

Btw tudjuk, hogy Sokoban $\leq_{\mathcal{P}}$ Maze nem igaz: nincs olyan inputkonverziós algoritmus, mely egy általános Sokoban példányból gyorsan egy Maze példányt készít választartó módon.

Univerzális RAM-gép

Dacára egyszerű szintaxisának, a RAM-gép is Turing-teljes számítási modell: minden „hagyományos” programozási nyelven megvalósítható algoritmusra létezik RAM program is.

Így például RAM interpretert is írhatunk RAM nyelven...

Az „interpreter”t, olyan programot, mely egy másik program forráskódját (is) kapja inputként és futtatja, „szimulálja” azt, ezen a területen „univerzális program”nak nevezik

Létezik olyan U univerzális RAM-program, amelynek ha adott egy M RAM-program és annak egy x bemenete, úgy viselkedik, mint M az x-en, vagyis $U(M; x) = M(x)$.

Természetesen M számsorozatként kódolva adott U számára (utasítások, tömbcímzések stb. számokkal).

RAM-gép kódolása

Például

- mivel egy utasítás legfeljebb három címzéssel operál (az összeadásnál / kivonásnál a bal oldalon egy, a jobb oldalon két címzés szerepel), így minden utasítást pl. öt regiszterben eltárolhatunk: sorszám, „opcode", címzés, címzés, címzés
- a sorszám természetesen egy pozitív egész szám
- az opcode pl. lehet
  - 1, ha ez egy JZ $W[k_1],\ k_2$ utasítás (ekkor csak két címzést használunk a háromból, a harmadik bármi nemnulla „trash" lehet)
  - 2, ha ez egy JZ $I[k_1],\ k_2$ utasítás
  - 3, ha ez egy JZ $O[k_1],\ k_2$ utasítás
  - 4, ha ez egy JZ $W[W[k_1]],\ k_2$ utasítás
  - ...
  - 13, ha ez egy $W[k_1] := W[k_2] + W[k_3]$ utasítás (ekkor a következő három regiszter rendre a $k_1,\ k_2,\ k_3$ értékeket tárolja)
  - 14, ha ez egy $W[k_1] := W[k_2] - W[k_3]$ utasítás
  - 15, ha ez egy $W[k_1] := W[k_2] + I[k_3]$ utasítás
  - ...
  - 37, ha ez egy $W[k_1] := I[k_2] - I[k_3]$ utasítás
  - ...
  - 59, ha ez egy $W[W[k_1]] := I[k_2] - W[O[k_3]]$ utasítás
- a fontos: hogy ilyen opcode-okból véges sokféle lesz (értékadásnál pl. a bal oldalon $3 + 9 = 12$-féle címzés, a jobb oldalon $1 + 3 + 9 = 13$-féle címzés kétszer és kétféle műveleti jel, ez $13^2 \times 2 = 338$-féle opcode), ezért lehet rá switchelni

RAM-gép kódolása

Például

ekkor mondjuk a

$$
\begin{array}{ll}
1 & W[1] := I[1] - I[2] \\
2 & \text{JZ } W[1],\ 4 \\
3 & \text{ACCEPT} \\
4 & \text{REJECT}
\end{array}
$$

program kódja lehet pl. 1 37 1 1 2 2 1 1 4 8 3 400 1 1 1 4 401 1 1 1

Az interpreter

- tarthatja pl. $W[0]$-ban a programszámlálót, ezt beállítja 1-re az elején
- W további részén az I, W és O tömbök nemnulla elemeit $(\text{index}, \text{érték})$ párokként
- egy lépésben megkeresi a $W[0]$ számú sort az inputban, dekódolja egy hosszú if-elsében az opcode-ot és hogy melyik regisztereket kell olvasni/írni, elvégzi a műveletet, updateli a tárolt asszociatív tömböket

Ha egy rendszer elég összetett, nem tudja saját magát vizsgálni

Relatív eldönthetetlenség

Ha egy akármilyen eszköz

- képes szimulálni ugyanolyan típusú eszközt (akár saját magát)
- képes nemterminálni is, terminálni is
- képes negálni az outputot, másolni az inputot és if/else elágazást

annak súlyos következményei vannak: nem tudja eldönteni az ugyanilyen típusú eszközökről, hogy adott inputon megállnak-e vagy sem

A megállási probléma – első eldönthetetlen problémánk

Megállás

- Input: M program, x input.
- Output: Megáll-e M az x-en futtatva?

Tétel

A Megállás probléma eldönthetetlen.

Bizonyítás

Tegyük fel, hogy Megállás eldönthető egy M_0 programmal.

Legyen D a következő (Ram-program forráskódot váró) program:

$$D(M) = \text{if } M_0(M;M) \text{ then } \nearrow \text{ else halt.}$$

Ekkor:

$$D(D) = \nearrow \Leftrightarrow \; M_0(D;D) = \text{„igen”} \; \Leftrightarrow \; D(D) \neq \nearrow,$$

ami ellentmondás.

Visszavezetés: eldönthetetlenség bizonyítása

Ha már ismerünk egy eldönthetetlen problémát, másokról is meg tudjuk mutatni, hogy eldönthetetlenek.

Hogyan?

Azt használjuk fel, hogy ha

- A eldönthetetlen és
- $A \leq_{\mathcal{R}} B$,

akkor B is eldönthetetlen.

Mindenen Megállás

Az alábbi probléma algoritmikusan eldönthetetlen.

- Adott: M Ram-program.
- Kérdés: M megáll-e minden bemenetén?

Bizonyítás

Adott M és x esetén legyen M_x olyan program, hogy az M_x minden y inputjára:

$$M_x(y) \;=\; M(x).$$

(azaz: M_x tetszőleges y input esetén futtassa M-et x-en.)
Így

$$M(x) \neq \nearrow \Leftrightarrow \; M_x \text{ megáll minden bemenetén.}$$

Ez az $(M; x) \mapsto M_x$ hozzárendelés kiszámítható, tehát rekurzív visszavezetés; és mivel az eldönthetetlen Megállás problémát visszavezettük erre a problémára, ez is eldönthetetlen.

Üres Inputon Megállás

Legyen Üres Inputon Megállás a következő probléma:

- Input: egy M program.
- Output: megáll-e M az üres inputon (azaz mikor minden input regisztert 0-val inicializálunk)?

Eldönthetetlen!

Visszavezetjük rá a Megállás problémát.

Adott M-hez és x-hez legyen M_x az a program, ami tetszőleges y inputra

- ha y az üres input, akkor futtatja M-et x-en;
- egyébként mondjuk megáll.

Ez az M_x az $M; x$ párból elkészíthető és nyilván

$$M \text{ megáll } x\text{-en} \iff M_x \text{ megáll üres inputon.}$$

Tehát Megállás $\leq_{\mathcal{R}}$ Üres Inputon Megállás, így ez utóbbi is eldönthetetlen probléma.

Mit láttunk eddig?

A Megállás, Mindenen Megállás és Üresszón Megállás problémák eldönthetetlensége szerint...

- nincs olyan algoritmus, mely inputként kap egy Java forráskódot és hozzá egy inputot, és megválaszolja, hogy a megadott program megáll-e a megadott inputon;
- nincs olyan algoritmus, mely inputként kap egy forráskódot és megválaszolja, hogy a program eshet-e végtelen ciklusba egyáltalán, vagy akár csak azt, hogy paraméter nélkül elindítva végtelen ciklusba fog-e esni!

A helyzet azonban ennél sokkal rosszabb...

nemsokára meglátjuk, hogy mennyivel

Rekurzív felsorolhatóság

Egy A probléma rekurzívan felsorolható (vagy Turing-felismerhető), ha van olyan M RAM-program, amire

$$M(x) = \begin{cases} \text{ACCEPT} & \text{, ha } x \in A \\ \nearrow & \text{, egyébként.} \end{cases}$$

A rekurzívan felsorolható problémák osztályát $\mathbf{RE}$ jelöli.

„recursively enumerable”

Nyilván $\mathbf{R} \subseteq \mathbf{RE}$. (Hiszen egy RAM programot ACCEPT-től különböző válasz helyett végtelen ciklusba egyszerű küldeni.)

Rekurzív felsorolhatóság

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

Bizonyítás

Az eldönthetetlen MEGÁLLÁS probléma rekurzívan felsorolható, pl. azzal a RAM-programmal, mely futtatja az univerzális RAM-programot az input $M; x$ páron, majd ha az megáll, ACCEPT választ ad.

A „rosszabb helyzet”: Rice tétele

Rice tétele

A rekurzívan felsorolható problémák egyetlen nemtriviális tulajdonsága sem dönthető el algoritmikusan.

Vagyis: ha $\emptyset \neq \mathcal{C} \subsetneq \mathbf{RE}$ a rekurzívan felsorolható problémák egy (nemtriviális) osztálya, akkor a következő probléma eldönthetetlen: adott egy M program, igaz-e, hogy $L(M) \in \mathcal{C}$?

Másképp mondva: automatikusan semmi nemtriviálisat nem tudunk eldönteni egy program viselkedéséről a forráskódjának ismeretében.

Ezért válnak szükségessé például kommentek, tervezési minták használáta, statikus és dinamikus tesztelés. . .

További eldönthetetlen problémák

Hilbert 10. problémája

- Adott: $p(x_1, \ldots, x_n)$ egész együtthatós polinom.
- Kérdés: Létezik-e egész értékű zérushelye?

Tétel (Matijasevič)

Hilbert 10. problémája algoritmikusan megoldhatatlan.

Post megfelelkezési problémája

- Adott: $u_1, v_1, \ldots, u_n, v_n \in \Sigma^*$.
- Kérdés: Létezik-e olyan $i_1, \ldots, i_k$ $(k > 0,\ 1 \leq i_j \leq n)$ sorozat, hogy

$$u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k} \ ?$$

Tétel (Post)

A fenti probléma algoritmikusan megoldhatatlan.

Post megfelelkezési problémája

Példa

PI: $\binom{0}{100}$, $\binom{01}{00}$, $\binom{110}{11}$-nek van:

$$\binom{110}{11}\binom{01}{00}\binom{110}{11}\binom{0}{100}$$

hiszen ha így rakjuk le a dominókat (note: bármelyiket ér többször is használni), alul is és felül is 110011100 lesz a teljes szó

Post problémájánál is és Hilbert 10. problémájánál is ha van megoldás, azt kitartó kereséssel előbb-utóbb megtalálhatjuk:

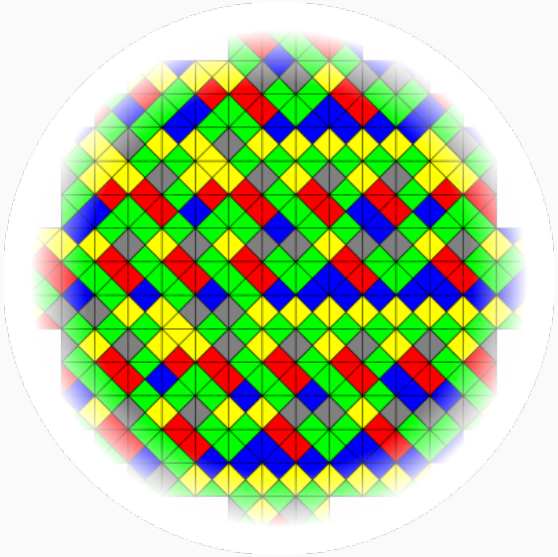
- Post: előbb az összes 1 hosszú dominósorozatot, majd az összes 2 hosszút, stb kipróbáljuk
- Hilbert 10: előbb az összes olyan egész $(x_1, x_2, \ldots, x_n)$ vektort, melyben minden $|x_i| \leq 1$, aztán az összes olyat, melyben minden $|x_i| \leq 2$ stb.

Tehát ez a két probléma rekurzívan felismerhető, de nem dönthető el.

Egy dominó probléma

- Adott: Dominó típusok véges halmaza.
- Kérdés: Kirakható-e ezekkel a dominókkal az egész sík hézagmentesen?

Típus: $\begin{array}{|ccc|}\hline & u_1 & \\ u_2 & & u_3 \\ & u_4 & \\ \hline\end{array}$, az u_i-k színek.



A kép forrása: https://en.wikipedia.org/wiki/Wang_tile

A dominók nem forgathatóak.

A dominó probléma algoritmikusan megoldhatatlan.

Egy megoldatlan probléma

- Adott: m pozitív egész szám.
- Kérdés: Kongruens-e m?

Tehát azt kérdezzük, hogy létezik-e olyan derékszögű háromszög, mely oldalai racionális hosszúságúak és melynek területe m.

1, 2, 3, 4 nem kongruensek. de mindhez sokat kell számolni és ötletelni, hogy miért

5 kongruens (Fibonacci):

$$a = \frac{3}{2},\ b = \frac{20}{3},\ c = \frac{41}{6}$$

Nem ismert, hogy a probléma algoritmikusan eldönthető-e.

Legyen $\mathcal{C}$ problémák egy osztálya. Ekkor

$$\text{co}\mathcal{C} := \{ \overline{L} : L \in \mathcal{C} \}.$$

Példák

- co$\mathbf{R}$: azon problémák, melyek komplementere rekurzív
- co$\mathbf{RE}$: azon problémák, melyek komplementere rekurzívan felsorolható
- co$\mathbf{P}$: azon problémák, melyek komplementere polinomidőben eldönthető

R, RE és coRE

Ha A rekurzív, akkor $\overline{A}$ is az.

Bizonyítás

Az Accept és a Reject sorok cseréjével.

Egy A probléma pontosan akkor rekurzív, ha A és $\overline{A}$ rekurzívan felsorolhatók.

Bizonyítás

- A rekurzív $\Rightarrow$ A rekurzívan felsorolható
  A rekurzív $\Rightarrow$ $\overline{A}$ rekurzív, így $\overline{A}$ rekurzívan felsorolható
- Tegyük fel, hogy A és $\overline{A}$ rekurzívan felsorolhatók. Ekkor A felismerhető egy M, $\overline{A}$ pedig egy $\overline{M}$ programmal. Szimuláljuk M-et és $\overline{M}$-et párhuzamosan!

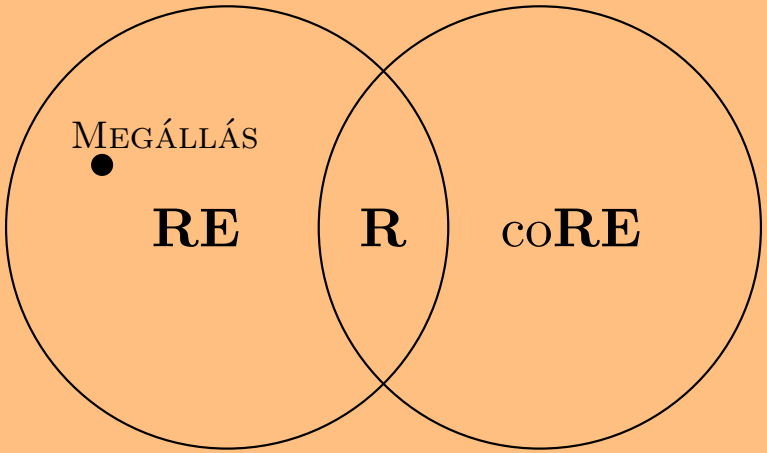
$\mathbf{R}$, $\mathbf{RE}$ és $\mathrm{co}\mathbf{RE}$ viszonya

Az előzőekből és abból, hogy $\mathbf{R} \subsetneq \mathbf{RE}$, ezeket kapjuk:

$$\mathbf{R} = \mathrm{co}\mathbf{R}$$

$$\mathbf{RE} \neq \mathrm{co}\mathbf{RE}$$

$$\mathbf{R} = \mathbf{RE} \cap \mathrm{co}\mathbf{RE}$$



meg persze pl. $\mathbf{P} = \mathrm{co}\mathbf{P}$ is igaz

Nemdeterminizmus

Láttunk rá bizonyítékot, hogy minden „realisztikus” számítási modell szimulálható RAM-gépen, lényegi időigény-romlás nélkül.

Most bevezetünk egy nemrealisztikus számítási modellt.

Egy nemdeterminisztikus RAM-programban

- több különböző programsor is kaphatja ugyanazt a sorszámot
- egy lehetséges futás minden lépésében a PC által kijelölt sorszámú lehetséges utasítások egyike hajtódik végre
- (utána a PC eggyel nő vagy ugrás esetén a megfelelő értékre áll be)

A program elfogadja az inputot, ha van olyan lehetséges futás, mely ACCEPT utasítással terminál, egyébként elutasítja azt

A Next c. mesében láthatjuk a koncepció előnyeit https://www.youtube.com/watch?v=lufECeWtN34

Nemdeterminisztikus generálás

Egy bit

1. $a := 0$
1. $a := 1$
2. $\ldots$

Egy n-bites szám

function ND(n)

1. $r := 0$
2. if $n == 0$ **return** r
3. $r := r + r$
4. $r := r + 1$
4. $r := r + 0$
5. $n := n - 1$
6. goto 2

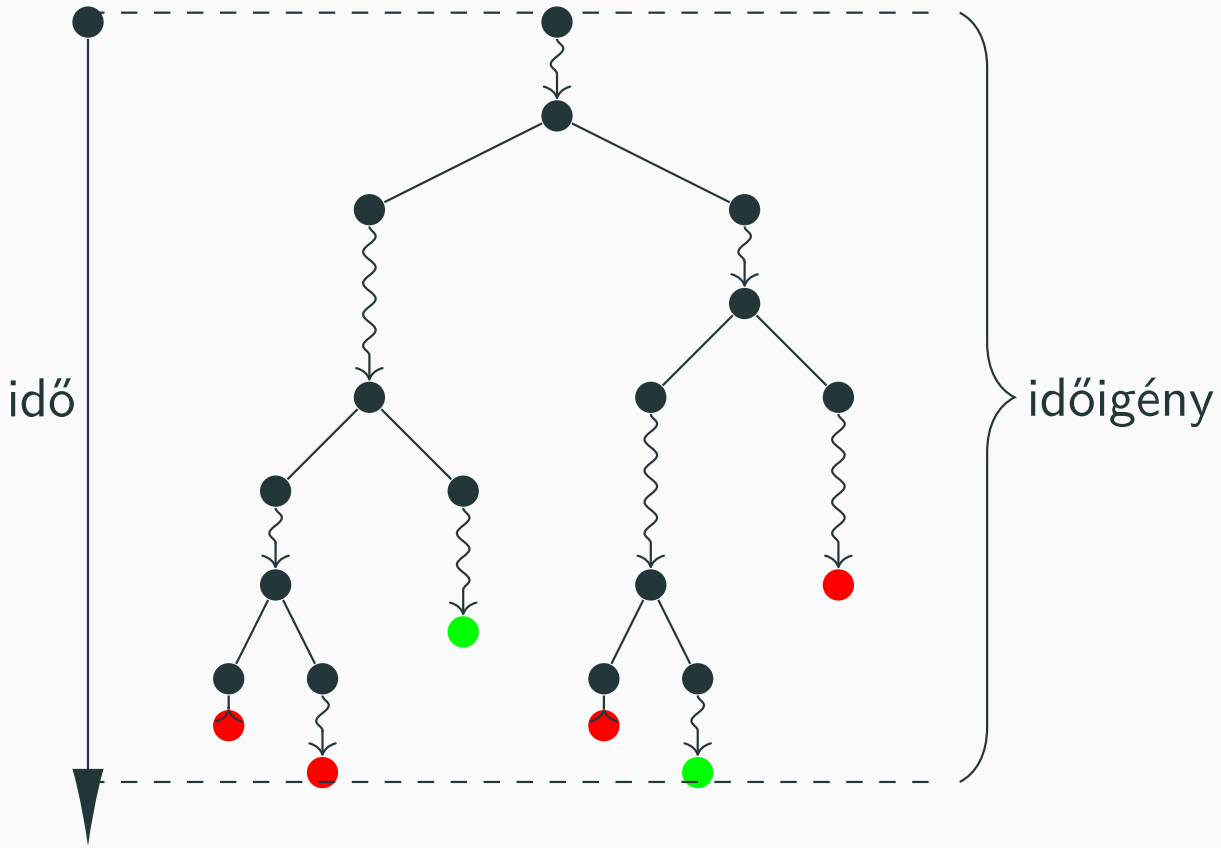
Példa: Hamilton-Út

- Input: $\vec{G} = (V, E)$ (irányított) gráf. Feltehető, hogy $V = \{1, \ldots, n\}$.
- Output: Van-e $\vec{G}$-ben Hamilton-út (azaz minden csúcsot érintő út)?

for $i := 1 \ldots n$ **do**
  $W[i] :=$ND$(1 + \lfloor \log n \rfloor)$ generálunk számokat egy n méretű tömbbe
for $i := 1 \ldots n - 1$ **do**
  if $(W[i], W[i+1]) \notin E$ **then** Reject megnézzük, hogy séta-e
for $i := 1 \ldots n$ **do**
  for $j := 1 \ldots n$ **do**
    if $W[j] == i$ **then** BREAK és hogy minden csúcsot érint-e
  if $j > n$ **then** Reject
Accept ha igen, akkor találtunk egy Hamilton-utat, tehát van benne

Nemdeterminisztikus időigény

Egy nemdeterminisztikus program időigénye $f(n)$, ha bármely n méretű inputon tetszőleges lehetséges futás $f(n)$ lépésben véget ér.



Nemdeterminisztikus időigény

Hamilton-út algoritmus időigénye

Az előző, Hamilton-útra adott algoritmus időigénye például négyzetes:

- nemdeterminisztikusan generál n darab, egyenként $\approx \log n$-bites számot, ez $n \log n$ lépés;
- (determinisztikusan) ellenőrzi, hogy az n szám ebben a sorrendben valóban egy séta-e a gráfban, ez n lépés;
- (determinisztikusan) ellenőrzi, hogy minden csúcs előfordul-e a sétában, ez n^2 lépés.

az input mérete mondjuk n^2 is lehet, ha szomszédsági mátrix, ekkor lineáris az időigény – de mindenképp POLINOM az input bitjeinek számának függvényében

Megjegyzés

Az utolsó ellenőrzés hatékonyabban is elvégezhető, most a pszeudokód egyszerűségét tartottuk szem előtt.

Példa: SAT

SAT **you should know this by now by heart**

- Input: konjunktív normálformájú formula
- Output: kielégíthető-e?

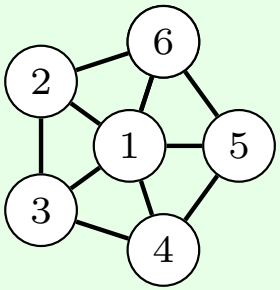
Algoritmus

- minden változónak nemdeterminisztikusan beállítjuk az értékét 1-re vagy 0-ra (lineáris időigény);
- determinisztikusan ellenőrizzük, hogy a kapott kiértékelés kielégítő-e (lineáris időigény);
- ha igen, ACCEPT, egyébként REJECT választ adunk.

Példa: 3 – Színezés

3-Színezés

- Input: $G = (V, E)$ (irányítatlan) gráf.
- Output: kiszínezhető-e G helyesen, vagyis adhatunk-e minden csúcsnak egy-egy színt egy háromelemű színhalmazból, hogy szomszédos csúcsok különböző színt kapjanak?



Ez egy „nem” példány, a csúcsok helyes színezéséhez kell legalább 4 szín.
Figyelem: az inputban nincsenek színezve a csúcsok!

Nemdeterminisztikus algoritmus

- minden csúcsnak nemdet. adunk egy színt az $\{R, G, B\}$ halmazból;
- determinisztikusan ellenőrizzük, hogy a kapott színezés helyes-e;
- ha igen, Accept, egyébként Reject választ adunk.

Az NP osztály

Az **NP** osztályba mindazok a problémák tartoznak, melyek eldönthetők polinom időigényű nemdeterminisztikus programmal.

Azaz, $A \in \mathbf{NP}$, ha van olyan M polinom időkorlátos nemdeterminisztikus program, melyre

- ha $x \in A$, akkor M-nek létezik elfogadó futása x-en, és
- ha $x \notin A$, akkor M-nek minden futása elutasítja x-et.

Hamilton-Út, Sat és $3-$Színezés **NP**-beli problémák.

Mivel minden $f(n)$ időigényű program felfogható $f(n)$ időigényű nemdeterminisztikus programként is, így nyilván $\mathbf{P} \subseteq \mathbf{NP}$.

Polinomidőben verifikálhatóság

Az előző példákban az algoritmusok mind a következő sémára épültek fel:

- nemdeterminisztikusan generáltak valami „bizonyítékot” arra, hogy az input a problémának egy IGEN példánya, majd
- determinisztikusan ellenőrizték, hogy tényleg jó bizonyítékot sikerült-e generálni.

Ez a séma pontosan karakterizálja az **NP** osztályt!

Polinomidőben verifikálhatóság

Egy L probléma pontosan akkor van az **NP** osztályban, ha létezik egy olyan $K \subseteq \mathbb{N}^* \times \mathbb{N}^*$ reláció „inputok” és „tanúk” közt, melyre a következők fennállnak:

- létezik olyan k konstans, melyre ha $(x, y) \in K$, akkor $|y| \leq |x|^k$

  azaz az egyes inputokhoz tartozó tanúk „rövidek”

- K polinom időben eldönthető

  azaz van hatékony algoritmus, mely az (x, y) párra eldönti, hogy y az x-hez tartozó tanú-e

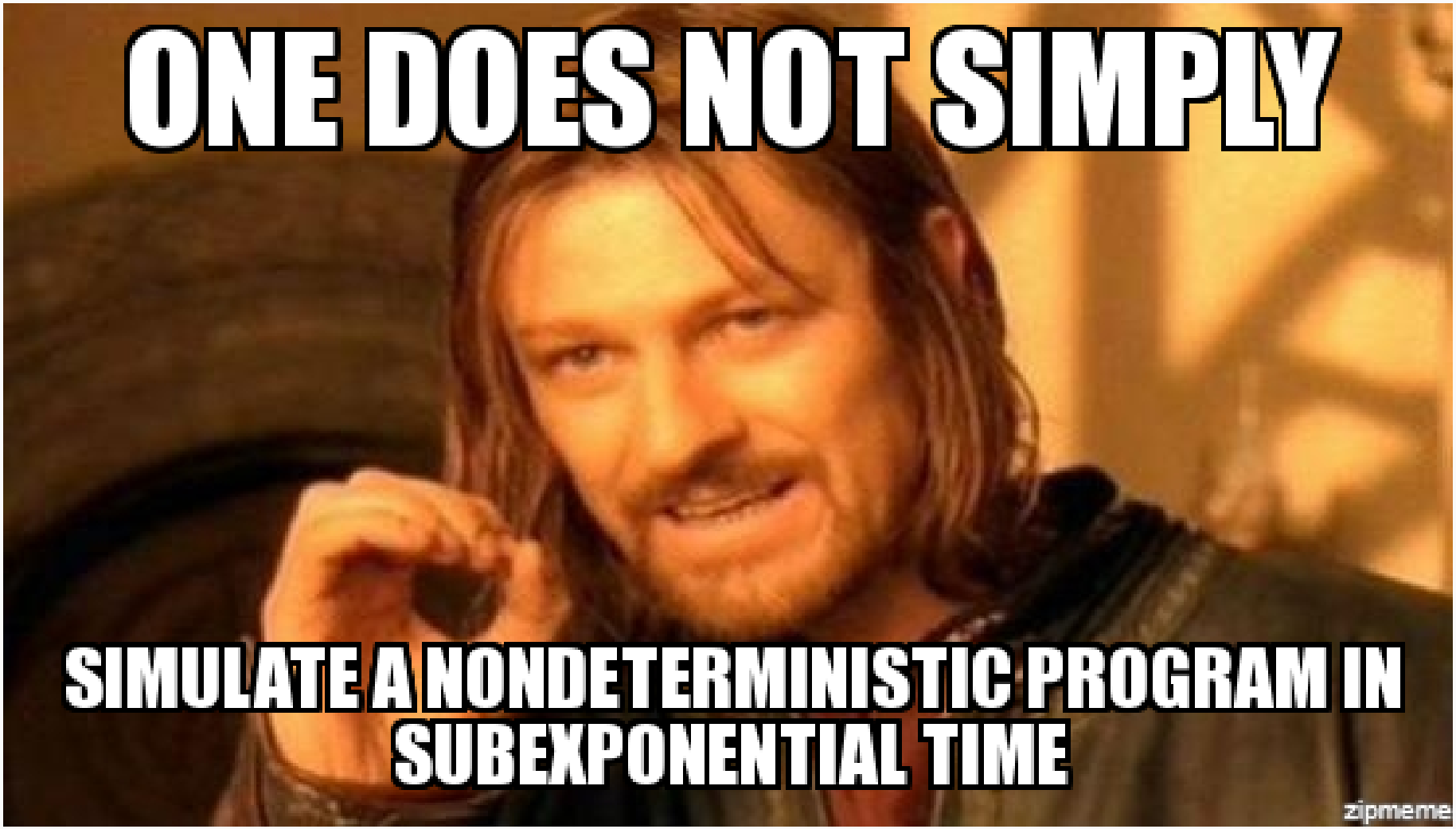
- $x \in L$ pontosan akkor, ha $\exists y : \ (x, y) \in K$

  azok az inputok a probléma IGEN példányai, melyekre van tanú

A HAMILTON-ÚT problémánál egy gráfhoz tartozó tanú pl. maga egy Hamilton-út: lineáris méretű és könnyű ellenőrizni, hogy tényleg Hamilton-út vagy sem, és – nyilván – pontosan az IGEN példányokra van ilyen tanú.

Hatékonyság

Hatékonyak ezek a „polinomidejű” algoritmusok?



that is, realisztikus architektúrán valszeg nem

Nehézség, teljesség

Legyen $\mathcal{C}$ (eldöntési) problémák egy osztálya.

- Az A probléma $\mathcal{C}$-nehéz, ha $\mathcal{C}$ minden eleme visszavezethető rá.
- Ha még $\mathrm{A} \in \mathcal{C}$ is, akkor A $\mathcal{C}$-teljes.

Észrevétel

Ha egy $\mathbf{NP}$-teljes probléma polinomidőben eldönthető, akkor (és csak akkor) $\mathbf{P} = \mathbf{NP}$.

(Hiszen akkor $\mathbf{NP}$ bármelyik problémája eldönthető egy polinomidejű visszavezetés és egy, a fenti problémát eldöntő polinomidejű algoritmus kompozíciójával, tehát polinomidőben.)

átkonvertáljuk a megoldandó probléma inputját ezére a könnyű $\mathbf{NP}$-teljesére, ami megy, mert $\mathbf{NP}$-nehéz, aztán megoldjuk, ami gyorsan megy, mert $\mathbf{P}$-ben van

$\mathcal{C}$-nehéz problémák keresése

Észrevétel

A hatékony visszavezetés tranzitív.

két gyors, választartó inputkonverziót egymás után végrehajtva gyorsan konvertáltunk inputot, továbbra is választartó módon

Ezért hogy megmutassuk egy A probléma $\mathcal{C}$-nehézségét, mindössze vissza kell rá vezetnünk egy másik, ismerten $\mathcal{C}$-nehéz B problémát. Hisz ekkor

- minden $\mathcal{C}$-beli visszavezethető B-re,
- B visszavezethető A-ra,

ezért (tranzitivitás!) minden $\mathcal{C}$-beli is visszavezethető A-ra.

Csakhogy...

...az első $\mathcal{C}$-nehéz problémát hogy kapjuk?

Visszavezetésre való zártság

Azt mondjuk, hogy egy $\mathcal{C}$ bonyolultsági osztály zárt a visszavezetésre, ha valahányszor $A \leq_{\mathcal{P}} B$ és $B \in \mathcal{C}$, mindig teljesül $A \in \mathcal{C}$ is.

azaz: ha egy probléma benne van, akkor a nála könnyebbek is

Az eddig látott bonyolultsági osztályok ($\mathbf{P}$, $\mathbf{NP}$, $\mathbf{R}$, $\mathbf{RE}$) zártak a visszavezetésre.

Ha $\mathcal{C}$ zárt a visszavezetésre és A egy $\mathcal{C}$-teljes probléma, akkor $\mathcal{C}$-ben pontosan az A-ra visszavezethető problémák vannak ($\mathcal{C} = \{B : B \leq_{\mathcal{P}} A\}$). Tehát A reprezentálja az egész osztályt!

Megjegyzés

$\mathbf{P}$ bármely két nemtriviális A, B elemére igaz, hogy $A \leq_{\mathcal{P}} B$.

Első RE-teljes problémánk

MEGÁLLÁS **RE**-teljes.

Bizonyítás

Legyen $\mathrm{A} \in \mathbf{RE}$ egy tetszőleges probléma, amit felismer az M RAM-program. Akkor tetszőleges x inputra

$$x \in A \;\Leftrightarrow\; (M; x) \in \text{MEGÁLLÁS},$$

hiszen ha M az A problémát ismeri fel, akkor pontosan az IGEN példányain áll meg (ACCEPT választ adva), így az $x \mapsto (M; x)$ leképezés egy (hatékony) visszavezetése A-nak a MEGÁLLÁS problémára.

Így ha $\text{MEGÁLLÁS} \leq \mathrm{A}$ valamely A problémára, akkor A is **RE**-nehéz:

- MINDENEN MEGÁLLÁS
- ÜRES INPUTON MEGÁLLÁS
- EKVIVALENCIA
- ...

RE-n kívül

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

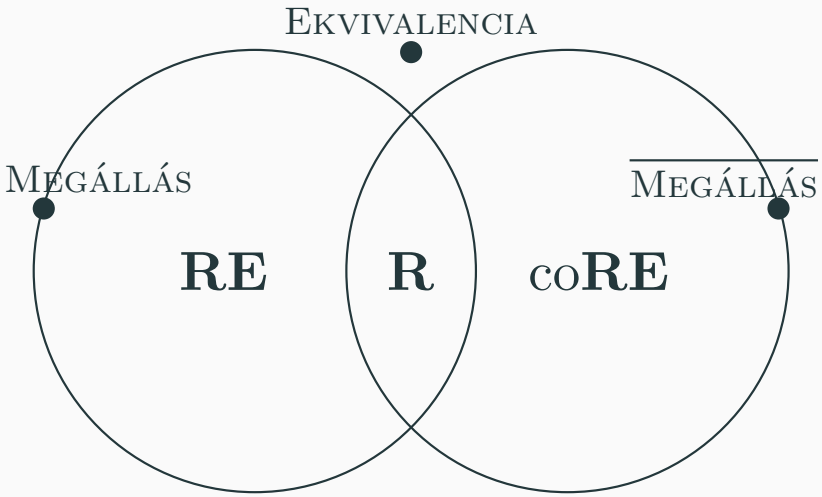
Bizonyítás

Ha A egyszerre $\mathcal{C}$-nehéz és $\mathcal{C}'$-beli probléma, akkor

- minden $\mathcal{C}$-beli probléma visszavezethető A-ra (mert A $\mathcal{C}$-nehéz);
- mivel $\mathrm{A} \in \mathcal{C}'$ és $\mathcal{C}'$ zárt a visszavezetésre, ezért emiatt minden $\mathcal{C}$-beli probléma $\mathcal{C}'$-ben van
- azaz $\mathcal{C} \subseteq \mathcal{C}'$.

RE-n kívül

Láttuk, hogy $\textsc{Megállás} \leq \textsc{Ekvivalencia}$.
Az is igaz, hogy $\overline{\textsc{Megállás}} \leq \textsc{Ekvivalencia}$, tehát Ekvivalencia **RE**-nehéz és co**RE**-nehéz is egyszerre.
Mivel $\mathbf{RE} \neq \text{co}\mathbf{RE}$ (emiatt persze egyikük sem lehet része a másiknak, hisz ha pl. $\mathbf{RE} \subseteq \text{co}\mathbf{RE}$, akkor $\text{co}\mathbf{RE} \subseteq \text{coco}\mathbf{RE} = \mathbf{RE}$, mely esetben egyenlőek), ez azt jelenti, hogy Ekvivalencia nem $\mathbf{RE} \cup \text{co}\mathbf{RE}$-beli probléma!
Azaz sem olyan algoritmus nincs, mely felismeri, ha két program ekvivalens, sem olyan, mely felismeri, ha két program nem ekvivalens.



Rajzainkban általában az osztályok „héjára” tesszük az osztály teljes problémáit.

Logikai hálózatok

Hálózat def

Hálózat: körmentes irányított gráf,
a csúcsok címkézettek: $\wedge$, $\vee$, $\neg$, igaz, hamis, x_i
$\wedge$, $\vee$ címke esetén a csúcs befoka 2,
$\neg$ címke esetén a csúcs befoka 1,
igaz, hamis, x_i címke esetén a csúcs befoka 0.

Általában megköveteljük még, hogy pontosan egy csúcs kifoka legyen 0.

Amennyiben a változók közül az $x_1, \ldots, x_n$ fordulnak elő a hálózatban (legfeljebb), akkor a hálózat kiszámít egy $\{\text{igaz}, \text{hamis}\}^n \to \{\text{igaz}, \text{hamis}\}$ Boole-függvényt.

Hálózat-Kiértékelés

- Adott: változómentes hálózat.
- Kérdés: igaz-e az értéke?

Hálózat-Kielégíthetőség

- Adott: hálózat.
- Kérdés: a változóknak lehet-e úgy értéket adni, hogy a hálózat értéke igaz legyen?

Sat

duh

- Adott: egy konjunktív normálformájú (ítéletkalkulus-beli) formula.
- Kérdés: kielégíthető-e?

Hálózat-Kielégíthetőség $\leq_{\mathcal{P}}$ Sat.

A visszavezetés

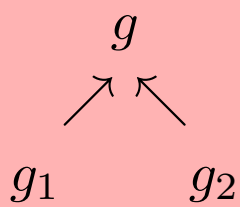
Minden g csúcsnak feleljen meg a g változó.

g címkéje x $\mapsto$ $x \leftrightarrow g$, azaz $(\neg x \vee g) \wedge (x \vee \neg g)$

g címkéje igaz $\mapsto$ g

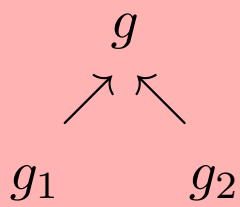
g címkéje hamis $\mapsto$ $\neg g$

g címkéje $\vee$:



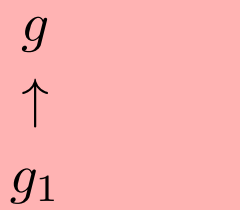
$\mapsto$ $g \leftrightarrow (g_1 \vee g_2)$, azaz $(\neg g \vee g_1 \vee g_2) \wedge (\neg g_1 \vee g) \wedge (\neg g_2 \vee g)$

g címkéje $\wedge$:



$\mapsto$ $g \leftrightarrow (g_1 \wedge g_2)$, azaz $(\neg g \vee g_1) \wedge (\neg g \vee g_2) \wedge (\neg g_1 \vee \neg g_2 \vee g)$

g címkéje $\neg$:



$\mapsto$ $g \leftrightarrow (\neg g_1)$, azaz $(\neg g \vee \neg g_1) \wedge (g_1 \vee g)$

g kimenő kapu $\mapsto$ g

Hálózat-Kielégíthetőség $\leq_{\mathcal{P}}$ Sat

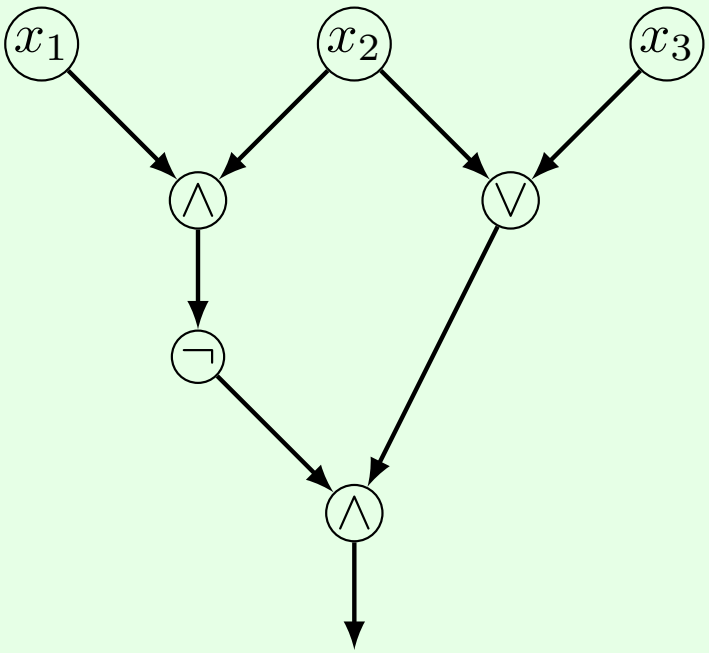
A keresett formula: a fenti formulák konjunkciója.

Adott hálózathoz a formula elkészíthető lineáris időben.

Továbbá a hálózat akkor és csak akkor kielégíthető, ha a formula az.

intuitíve azért, mert az egyes kapukhoz gyártott klózok kikényszerítik, hogy a kapuhoz generált változó értéke épp a kapu outputja legyen, miután beállítjuk az input változókat valamire

Példa



$$(g_1 \leftrightarrow x_1) \ \wedge \ (g_2 \leftrightarrow x_2) \ \wedge \ (g_3 \leftrightarrow x_3)$$
$$\wedge \ (g_4 \leftrightarrow (g_1 \wedge g_2)) \wedge \ (g_5 \leftrightarrow (g_2 \vee g_3)) \wedge$$
$$(g_6 \leftrightarrow (\neg g_4)) \wedge \ (g_7 \leftrightarrow (g_5 \wedge g_6)) \wedge \ g_7.$$

CNF alakban:

$$(\neg g_1 \vee x_1) \wedge (g_1 \vee \neg x_1)$$
$$\wedge \ (\neg g_2 \vee x_2) \wedge (g_2 \vee \neg x_2)$$
$$\wedge (\neg g_3 \vee x_3) \wedge (g_3 \vee \neg x_3)$$
$$\wedge \ (\neg g_4 \vee g_1) \wedge (\neg g_4 \vee g_2) \wedge (g_4 \vee \neg g_1 \vee \neg g_2)$$
$$\wedge \ (\neg g_5 \vee g_2 \vee g_3) \wedge (g_5 \vee \neg g_2) \wedge (g_5 \vee \neg g_3)$$
$$\wedge \ (\neg g_6 \vee \neg g_4) \wedge (g_6 \vee g_4)$$
$$\wedge \ (\neg g_7 \vee g_5) \wedge (\neg g_7 \vee g_6) \wedge (g_7 \vee \neg g_5 \vee \neg g_6)$$
$$\wedge g_7.$$

Itt pl. a formula kielégíthető az $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ és (az ezek miatt kikényszerített) $g_1 = 0$, $g_2 = 1$, $g_3 = 1$, $g_4 = 0$, $g_5 = 1$, $g_6 = 1$ és $g_7 = 1$ értékadás mellett; az eredeti hálózatot pedig kielégíti az $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ értékadás.

Első kiszámítható teljes problémáink

Emlékezzünk: egy $\mathcal{C}$-beli probléma $\mathcal{C}$-teljes, ha minden $\mathcal{C}$-beli probléma visszavezethető rá.

Eddig még csak eldönthetetlen, **RE**-teljes problémákkal találkoztunk: a Megállás ilyen volt.

A Hálózat-Kiértékelés probléma **P**-beli.

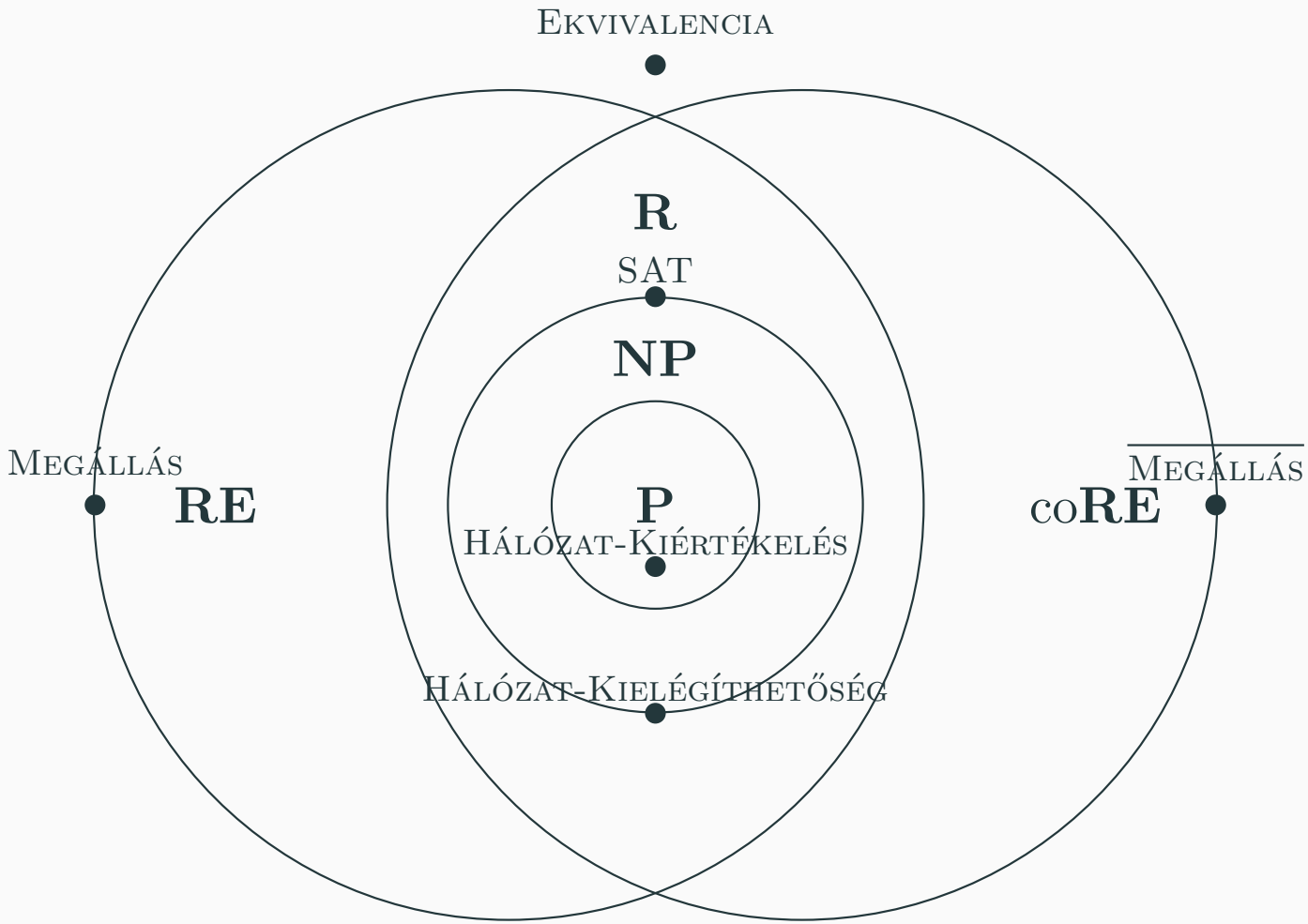
Tétel

A Hálózat-Kielégíthetőség probléma **NP**-teljes.

Következmény (Cook tétele)

A Sat probléma is **NP**-teljes.

Osztályok és problémák eddig



némelyik probléma még nincs a „végleges helyén”

Sat NP-teljességéről

Mivel a Sat probléma **NP**-teljes,

- ha polinomidőben megoldható lenne, akkor $\mathbf{P} = \mathbf{NP}$.

  Ebben persze lehet hinni. De eddig még senki nem oldotta meg polinomidőben.

- Nem ismert rá szubexponenciális algoritmus.
- Ilyenkor bevethetők (a teljesség igénye nélkül):
  - heurisztikák
  - randomizált algoritmusok alkalmazása
  - a követelmények relaxálása... (pl. nem az összes, de minél több klóz egyszerre történő kielégítése)
  - ...majd a relaxált követelmények approximálása
  - vagy olyan speciális esetek keresése, melyre van hatékony algoritmus.

a heurisztikák kivételével mindből fogunk látni példát

A kielégíthetőség változatai

A SAT problémának vizsgálhatjuk speciális eseteit, pl:

- klózonként csak konstans sok literált engedünk meg
- csak speciális (pl. Horn) alakú klózokat engedünk meg

Legyen $k \geq 1$. A kSAT a SAT azon speciális esete, amelyben minden klóz pontosan k (nem feltétlenül különböző) literálból áll.

3SAT **NP**-teljes, és így $k \geq 3$ esetén kSAT **NP**-teljes.

Ötlet: SAT $\leq_{\mathcal{P}}$ 3SAT

Vegyük a következő konstrukciót, mely egy klózhoz egy CNF-et rendel:

$$
\begin{aligned}
\ell &\mapsto \ell \vee \ell \vee \ell \\
\ell_1 \vee \ell_2 &\mapsto \ell_1 \vee \ell_2 \vee \ell_2 \\
\ell_1 \vee \ell_2 \vee \ell_3 &\mapsto \ell_1 \vee \ell_2 \vee \ell_3 \\
\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 &\mapsto (\ell_1 \vee \ell_2 \vee x) \wedge (\neg x \vee \ell_3 \vee \ell_4) \\
&\vdots \quad\quad \vdots \\
\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n &\mapsto (\ell_1 \vee \ell_2 \vee x) \wedge (\neg x \vee \ell_3 \vee y) \wedge \\
&\quad\quad (\neg y \vee \ell_4 \vee z) \wedge \ldots \wedge (\neg u \vee \ell_{n-1} \vee \ell_n)
\end{aligned}
$$

ahol x, y, z stb. teljesen új változók

Az előző oldal konstrukciójának a lényege:

- ha az eredeti ℓ_i literálok változóinak egy értékadása nem elégíti ki a bal oldalon lévő klózt, akkor ugyanezt az értékadást bárhogy is „bővítjük ki”, hogy az új változók is kapjanak értéket, a jobb oldalon lévő CNF hamis lesz;
- ha viszont a bal oldali klóz igaz egy eredeti értékadás mellett, akkor azt ki lehet úgy bővíteni, hogy a jobb oldali CNF is igaz legyen

  a klóz egyik igaz literáljától balra lévő új változóknak az igaz, a jobbra lévőknek a hamis értéket adva

Legyen $\varphi = c_1 \wedge c_2 \wedge \ldots\ldots \wedge c_k$ konjunktív normálformájú formula.
Minden c_i klózhoz legyen c_i' az előzőekben adott kifejezés, ahol minden kifejezésben új változókat használunk.
Ekkor φ akkor és csak akkor kielégíthető, ha $\varphi' = c_1' \wedge c_2' \wedge \ldots \wedge c_k'$ az.

Mivel $\text{SAT} \leq_{\mathcal{P}} \text{3SAT}$ és $\text{3SAT} \in \mathbf{NP}$, ezért SAT $\mathbf{NP}$-teljességéből következik, hogy 3SAT is $\mathbf{NP}$-teljes.

A 2SAT problémára viszont adható hatékony algoritmus.
Legyen φ egy $2 - CNF$ (minden klóz két literálból áll).
Készítünk φ-ből egy $G(\varphi)$ gráfot polinomidőben, majd ezen futtatunk egy polinomidejű algoritmust, amivel meg tudjuk oldani az eredeti problémát.

A $G(\varphi)$ gráf

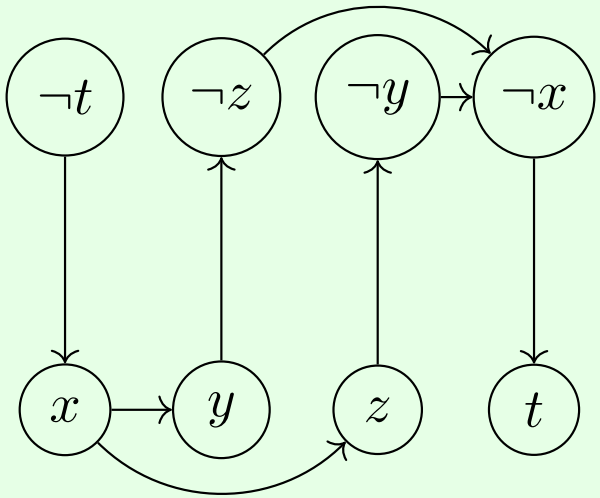
- csúcsok: a φ-ben előforduló változók és negáltjaik.
- élek: (α, β) él $\Leftrightarrow$ $\overline{\alpha} \vee \beta$ vagy $\beta \vee \overline{\alpha}$ a φ tagja (azaz ha $\alpha \to \beta$ vagy $\overline{\beta} \to \overline{\alpha}$ a φ tagja.)

tehát klózonként lesz két él a gráfban, ez megy lineáris időben

Példa

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

Ekkor $G(\varphi)$:



A φ formula akkor és csak akkor kielégíthető, ha nincs olyan x változó, amelyre létezik $G(\varphi)$-ben irányított út x-ből $\neg x$-be és vissza.

Ötlet

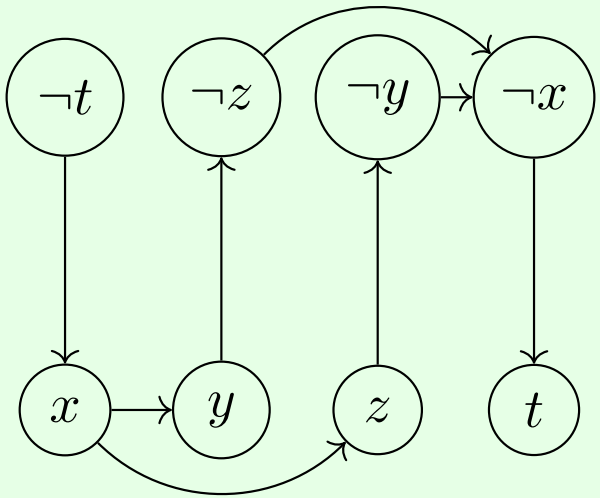
Minden él egy implikációnak felel meg.
Így ha ℓ_1-ből ℓ_2 elérhető, akkor $\varphi \models (\ell_1 \to \ell_2)$.
Ha x-ből is elérhető $\neg x$ és fordítva, akkor $\varphi \models (x \leftrightarrow \neg x)$, így φ kielégíthetetlen.
Ha nincs ilyen változó, akkor ciklusban:

- válasszunk egy olyan ℓ literált, amiből nem érhető el $\overline{\ell}$ és melynek még nem adtunk értéket;
- állítsuk 1-re ℓ-t és minden belőle elérhető literált, komplementereiket pedig nullára;

amíg minden változónak értéket nem adtunk (nem lesz ütközés, ha nincs olyan x, akiből $\neg x$ elérhető és viszont). Az így kapott értékelés kielégíti φ-t.

Példa

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



Válasszuk először x-et. $G(\varphi)$-ben x-ből elérhető $\neg x$.
igaz-ra állítjuk $\neg x$-et és t-t, hamis-ra x-et és $\neg t$-t.
Ezek után válasszuk mondjuk y-t. y-ból nem érhető el $\neg y$, tehát $\alpha := y$.
igaz-ra állítjuk y-t és $\neg z$-t, hamis-ra pedig $\neg y$-t és z-t.
A kapott értékadás kielégíti φ-t.

Mivel az algoritmus (a gráf legyártása, majd abban pl. erősen összefüggő komponensek kiszámítása után ellenőrizni, hogy van-e olyan x, mely $\neg x$-szel azonos komponensben van) polinomidejű, beláttuk a következőt:

2Sat $\in \mathbf{P}$.

Később majd a 2Sat-ot egy még ennél is (valószínűleg) kisebb bonyolultsági osztályba fogjuk tudni helyezni.

Horn-formulák és Horn-átnevezhető formulák

Egy klóz Horn-klóz, ha benne maximum egy pozitív literál szerepel.

Egy CNF Horn-formula, ha benne minden klóz Horn-klóz.

Egy CNF Horn-átnevezhető, ha bizonyos változói komplementálásával Horn-formulává tehető

Vagyis: ha van változók egy olyan X halmaza, hogy minden $x \in X$-re x helyébe $\neg x$-et, $\neg x$ helyébe x-et írva a formulába, az eredmény Horn-formula lesz.

Példa

- $x \wedge (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg y \vee \neg z)$ Horn-formula.
- $(x \vee y) \wedge (\neg x \vee \neg y)$ Horn-átnevezhető (pl. x és $\neg x$ szerepének cserélésével).
- $(\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (x \vee y)$ nem Horn-átnevezhető.

Horn-formulák és Horn-átnevezhető formulák

Horn-átnevezhető formulák kielégíthetősége polinom időben eldönthető.

Ötlet

Az egységrezolúció alkalmazása ezekre a formulákra teljes:

- amíg találunk egy egyetlen literálból álló $\{\ell\}$ klózt:
- ℓ értékét 1-re állítjuk;
- töröljük az összes, ℓ-t tartalmazó klózt;
- a maradék klózokból töröljük az $\overline{\ell}$ literált.

Ha közben megkapjuk az üres klózt, a formula kielégíthetetlen; ellenkező esetben kielégíthető.

Az algoritmus (ésszerű reprezentációval) lineáris időben végrehajtható és Horn-átnevezhető formulákra helyes.

Horn-formulák és Horn-átnevezhető formulák

Megjegyzés

Az is polinom időben eldönthető egy CNF-ről, hogy Horn-átnevezhető-e (ez a definícióból nem ennyire egyértelmű).

„Keverni” nem tudjuk a két hatékonyan eldönthető speciális esetet:

NP-teljes a következő probléma: adott egy olyan CNF, melyben minden klóz

- vagy max. két literálból áll,
- vagy háromelemű negatív klóz,

kielégíthető-e?

Ötlet: $3\textsc{Sat} \leq$ ez

Változónként $(x \vee \hat{x}) \wedge (\neg x \vee \neg \hat{x})$ és ehhez hozzá minden eredeti klózban a pozitív x literálokat $\neg\hat{x}$-re cseréljük. így ami eddig $\neg x$ volt, azt $\hat{x}$ veszi át.

Az előfordulásszám csökkentése

Azt is próbálhatjuk korlátozni, hogy egy változó (vagy egy literál) hányszor fordulhat elő legfeljebb a formulában.

NP-teljes a következő probléma: adott egy 3CNF, melyben minden változó legfeljebb háromszor szerepel, kielégíthető-e?

gyakorlaton

P-ben van a következő probléma: adott egy CNF, melyben minden változó legfeljebb kétszer szerepel, kielégíthető-e?

NP-teljes gráfelméleti problémák

Független Csúcshalmaz

Adott: $G = (V, E)$ irányítatlan gráf, K szám.

Kérdés: Létezik-e K elemű független csúcshalmaz G-ben?

azaz K darab, páronként nem szomszédos csúcs

A Független Csúcshalmaz probléma **NP**-teljes.

Ötlet

Világos, hogy a probléma **NP**-ben van.

Megmutatjuk, hogy 3Sat $\leq_{\mathcal{P}}$ Független Csúcshalmaz.

Legyen φ a 3Sat egy példánya, $\varphi = c_1 \wedge \ldots \wedge c_m$.

Az elkészített G gráf álljon m darab háromszögből, melyek a c_i klózoknak felelnek meg, s melyek csúcsai a c_i-kben lévő literáloknak felelnek meg. Ezen kívül még kössünk össze éllel minden ellentétes literált. Végül legyen $K := m$.

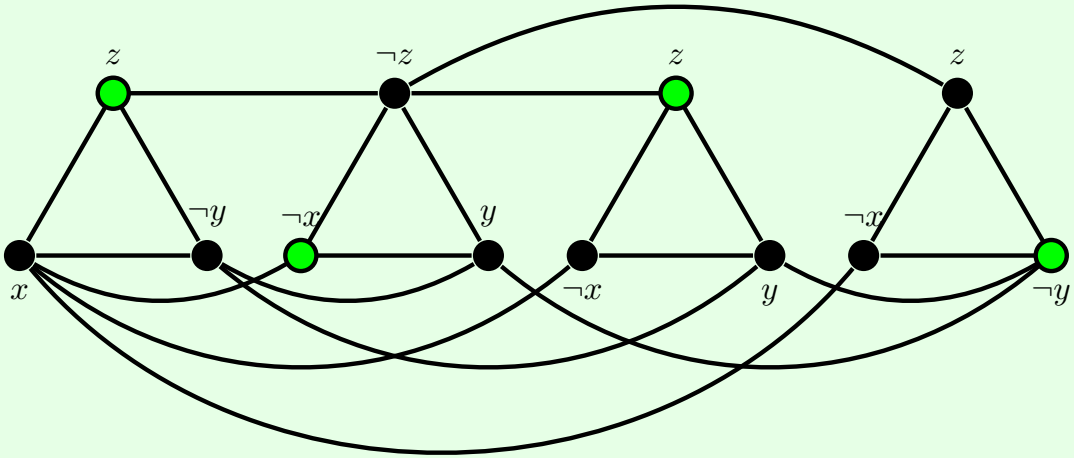
φ kielégíthető $\Leftrightarrow$ G-ben létezik K elemű független csúcshalmaz.

Példa

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

Ekkor a G gráf:

G csúcsai nem színesek, az csak illusztráció!



és $K = 4$.

A zölddel jelölt csúcsok egy négyelemű független csúcshalmazt alkotnak; egy kielégítő kiértékelés pedig: $x = y = 0$, $z = 1$.

NP-teljes gráfelméleti problémák

Visszavezetés konstruálásakor rendszerint egy, az előzőekben már többször látott módszert követünk, hogy belássuk a helyességét:

- az input egy „igen” példányának egy tanújából tudunk az outputhoz egy tanút konstruálni?
- az output ha „igen” példány lett, annak véve egy tanúját tudunk konstruálni egy tanút az inputnak is?

Klikk

Adott: $G = (V, E)$ irányítatlan gráf, K szám.

Kérdés: Létezik-e K elemű klikk (teljes részgráf)?

Klikk **NP**-teljes.

gyakorlaton

NP-teljes gráfelméleti problémák

Csúcslefedés

Adott: $G = (V, E)$ irányítatlan gráf, K szám.

Kérdés: Létezik-e olyan K elemű csúcshalmaz, hogy G minden éle illeszkedik a halmaz egy csúcsához?

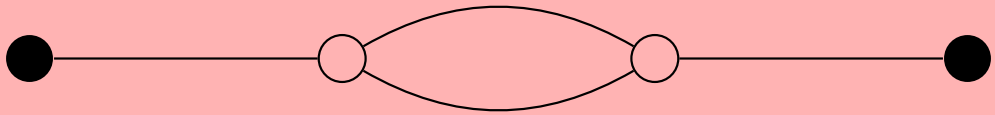
A Csúcslefedés probléma **NP**-teljes.

gyakorlaton

A Hamilton-Út probléma **NP**-teljes.

Ötlet

„Értékválasztó” gadget:

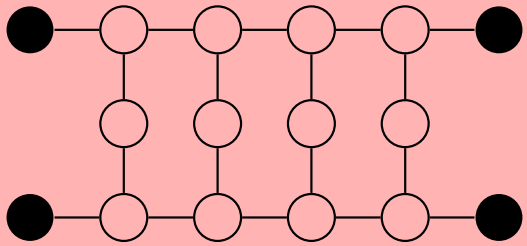


(Balról jobbra ha megy egy Hamilton-út, akkor választanunk kell a „fenti” és a „lenti” útvonal közül egyet.)

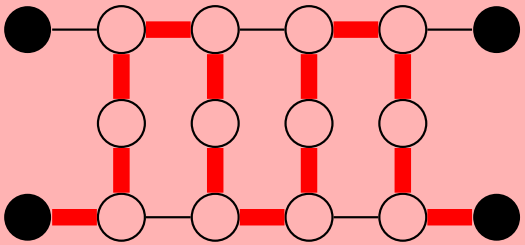
$\Rightarrow$ mint egy értékadás – ilyen gadgetből változónként lesz egy

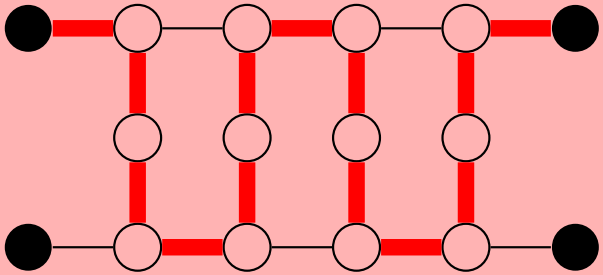
Hamilton-Út NP-teljes

„XOR" gadget:

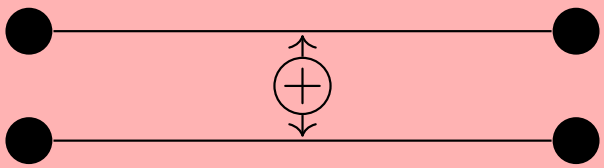


Minden Hamilton-út egy olyan gráfban, ami ezt a gadgetot feszített részgráfként tartalmazza, a következők egyike ezen a gráfon belül:





Hogy átláthatóbb legyen a rajzunk, ezt a gadgetet így rajzoljuk:



Vázlat

A XOR gadgettel egy gráfban olyan Hamilton-kört keresünk, melyben ki tudjuk kötni bizonyos élpárokra, hogy a két él közül a keresett Hamilton-kör pontosan egyen haladjon át.

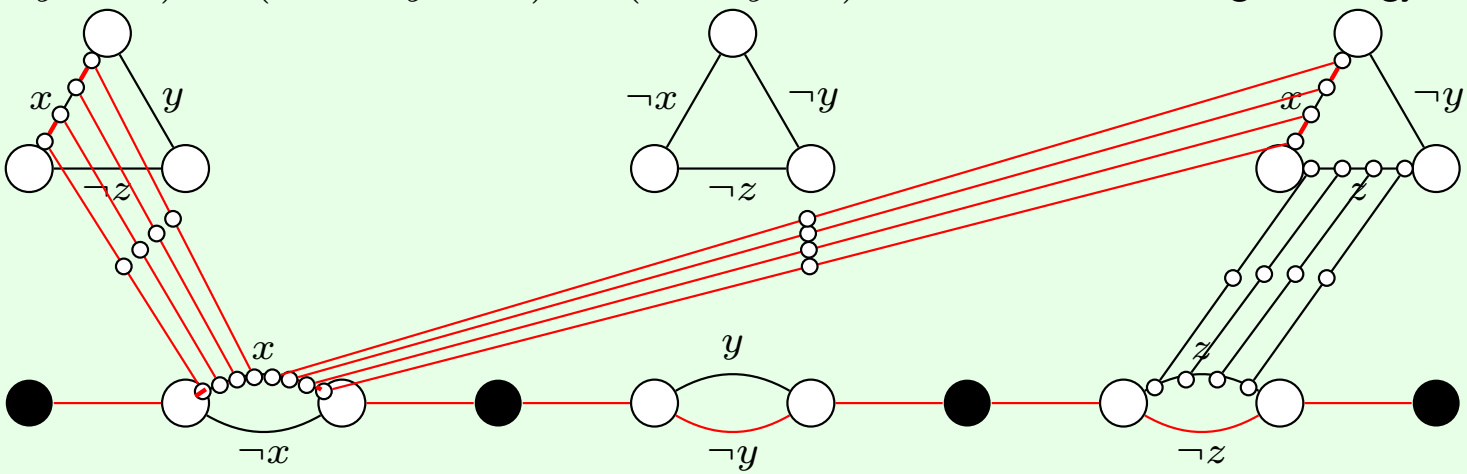
A teljes konstrukció a $3\text{SAT} \leq \text{HAMILTON-ÚT}$ visszavezetéshez:

- minden változóhoz létrehozunk egy értékválasztó gadgetet, az x_i-hez tartozó két párhuzamos élt x_i-vel ill. $\neg x_i$-vel címkézzük;
- minden klózhoz létrehozunk egy háromszöget, az $\ell_1 \vee \ell_2 \vee \ell_3$ klózhoz létrehozott háromszög éleit rendre ℓ_1-gyel, ℓ_2-vel és ℓ_3-mal címkézzük;
- az ellentétes literálokkal címkézett élek között XOR gadgetet hozunk létre;
- az értékválasztó gadgeteket láncba kötjük;
- a háromszögek csúcsaiból, az utolsó értékválasztó csúcsból és még egy plusz csúcsból pedig egyetlen nagy klikket készítünk;
- végül az előző pontbeli plusz csúcsból még egy új csúcsba lépünk.

Ez a konstrukció egy $3\text{SAT} \leq \text{HAMILTON-ÚT}$ visszavezetés lesz.

Hamilton-Út

Pl. az $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$ formulához készített gráfból egy részlet:



- alul a láncba kötött értékválasztó gadgetek; a bal oldali fekete csúcsból kell induljunk
- a felső háromszögek a klózok, minden élük rá van kötve XOR gadgetként egy lenti „hídra” (csak néhány van berajzolva)
- a „piros” élek megfelelnek az $x = 1$, $y = 0$, $z = 0$ értékadásból készített sétának a csúcsokon
- amelyik irányba megyünk a „hidakon”, az oda kötött XOR gadgeteket mindet bejárjuk (két $\neg y$ gadget bejárása nincs a képen, hogy átlátható maradjon)
- ezután be kell járnunk a nem érintett „hidak” párjait (mint pl. a harmadik klóz z élét) is
- amit pont akkor tudunk megtenni, ha minden háromszögnek legalább az egyik élét bejártuk idejövet (mert egy háromszögnek mind a három élét nem tudja egy Hamilton-út bejárni, de egyébként megoldható, hiszen a felső csúcsokat egy nagy klikkbe kötöttük (nincs az ábrán))

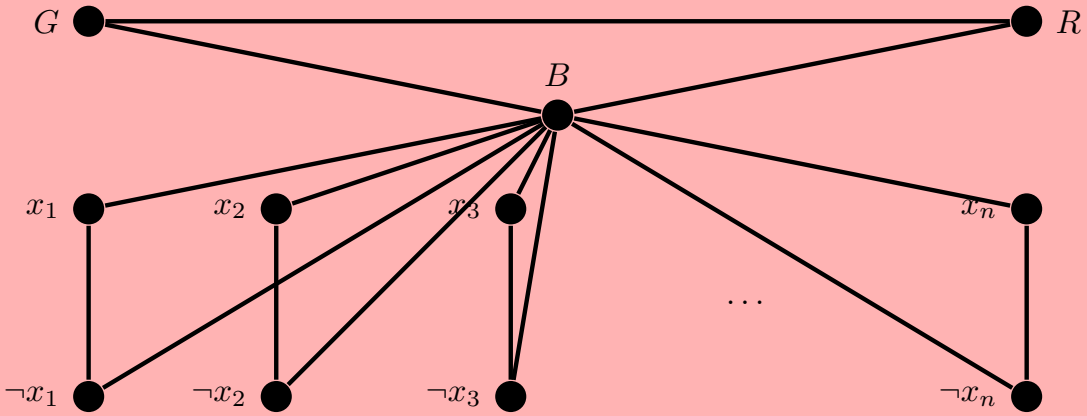
Így, mivel Hamilton-Út $\leq_{\mathcal{P}}$ Tsp(E), így a Tsp(E) probléma **NP**-teljes.

A 3-Színezés probléma **NP**-teljes.

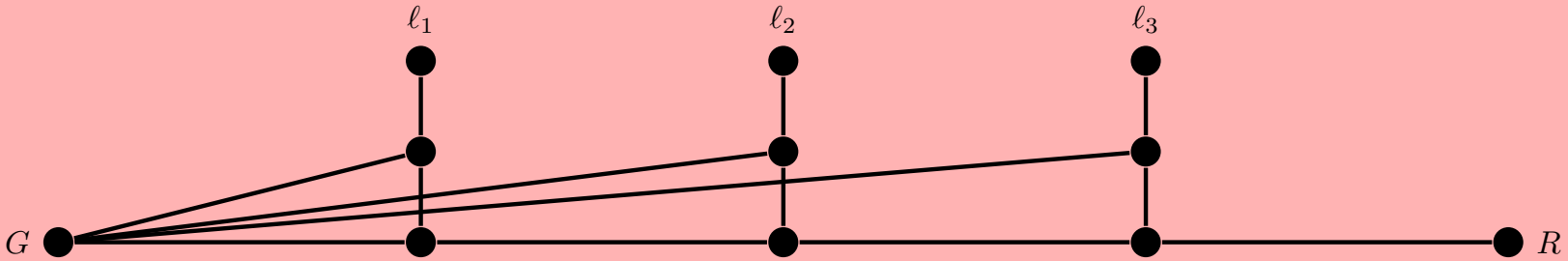
A 2-Színezés probléma **P**-ben van.
A 4-Színezés probléma triviális síkbarajzolható gráfokra.

„Triviális”: itt ez azt jelenti, hogy csak „igen” példánya van: minden síkbarajzolható gráf színezhető helyesen négy színnel, ez az úgynevezett négyszíntétel. Ezt bebizonyítani korántsem „triviális”.

3Sat $\leq$ 3-Színezés: ötlet



így minden literál színe R (=„hamis”) vagy G (=„igaz”) színével kell megegyezzen, komplementere pedig egy másikkal $\Rightarrow$ kiértékelés



Ellenőrizhető: ha mindhárom literál R színét kapja, a plusz hat csúcsot nem lehet helyesen 3-színezni, ha viszont van köztük G színű, akkor igen

NP-teljes problémák halmazokra

Hármasítás

Adott: Három azonos méretű halmaz, F (fiúk), L (lányok), H (házak), és egy $R \subseteq F \times L \times H$ reláció.

Kérdés: Megadható-e az R-beli hármasok egy olyan részhalmaza, melyben minden fiú, lány és ház pontosan egyszer szerepel?

Világos, hogy Hármasítás $\in \mathbf{NP}$.

A Hármasítás probléma $\mathbf{NP}$-teljes.

Párosítás $\in \mathbf{P}$. pl. meg lehet oldani a polinomidejű magyar módszerrel.

NP-teljes problémák halmazokra

Halmazlefedés

Adott: Egy U halmaz részhalmazainak $C = (S_1, \ldots, S_n)$ rendszere és egy K szám.

Kérdés: Kiválasztható-e K darab az S_i-k közül úgy, hogy ezek egyesítése U?

Halmazpakolás

Adott: Egy U halmaz részhalmazainak $C = (S_1, \ldots, S_n)$ rendszere és egy K szám.

Kérdés: Kiválasztható-e az S_i-k közül K darab, páronként diszjunkt halmaz?

Pontos Lefedés Hármasokkal

Adott: Egy $3m$ elemű U halmaz és 3-elemű részhalmazainak $(S_1, \ldots, S_n)$ rendszere.

Kérdés: Kiválasztható-e m darab S_i úgy, hogy ezek egyesítése U?

(A kiválasztott S_i-k nyilván diszjunktak.)

NP-teljes problémák halmazokra

A Halmazlefedés, Halmazpakolás, Pontos Lefedés Hármasokkal problémák **NP**-teljesek.

A Hármasítás a Pontos Lefedés Hármasokkal speciális esete.
A Pontos Lefedés Hármasokkal a Halmazlefedés, valamint a Halmazpakolás speciális esete is.

ezt mondjuk úgy is, hogy pl. a Pontos Lefedés Hármasokkal általánosabb, mint a Hármasítás.

azaz többféle inputra van definiálva, mint az eredeti probléma, úgy, hogy az eredeti problémának is megfelelő inputokra ugyanazt a választ várja el.

nyilván minden probléma visszavezethető minden nála általánosabb problémára: nem is kell konvertálni sehova az inputot, ezért pl. ezek a problémák is mind automatikusan **NP**-nehezek lesznek.

NP-teljes problémák számokra

Egész Értékű Programozás

Adott: Egy n-változós, egész együtthatós lineáris egyenlőtlenség-rendszer.
Kérdés: Létezik-e egész értékű megoldása?

Ötlet

A Halmazlefedés felfogható az Egész Értékű Programozás speciális eseteként:

$$Ax \geq 1, \quad \sum_{i=1}^{n} x_i \leq K, \quad 0 \leq x_i \leq 1,$$

ahol A oszlopai a halmazrendszer elemeinek felelnek meg.
Azt is meg lehet mutatni, hogy az Egész Értékű Programozás **NP**-ben van.

Az Egész Értékű Programozás probléma **NP**-teljes.

Példa a visszavezetésre

Pl. ha a Halmazlefedés problémában $U = \{1, 2, 3, 4, 5\}$, a halmazaink $S_1 = \{1, 2, 3\}$, $S_2 = \{1, 3, 5\}$, $S_3 = \{1, 4, 5\}$ és $K = 2$, akkor a generált egyenlőtlenségrendszer:

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2 \\ x_1 + x_2 + x_3 &\geq 1 \\ x_1 &\geq 1 \\ x_1 + x_2 &\geq 1 \\ x_3 &\geq 1 \\ x_2 + x_3 &\geq 1 \end{aligned}$$

és $0 \leq x_1, x_2, x_3 \leq 1$.

Ha egy megoldásban $x_i = 1$, az „azt jelenti", hogy S_i-t kiválasztjuk; ha $x_i = 0$, akkor nem (más opció az utolsó feltétel miatt, és mert a változók egészértékűek, nincs).

Így az első feltétel azt mondja, hogy két halmazt választunk ki legfeljebb.

A többi pedig rendre azt, hogy legalább egy olyan halmazt is kiválasztunk, akiben szerepel az 1-es; legalább egy olyat is, akiben szerepel a 2-es, stb.

NP-teljes problémák számokra

A Részletösszeg probléma

Adott: $a_1, \ldots, a_n$ pozitív egészek és egy $K > 0$ célszám.

Kérdés: Kiválasztható-e az a_i-k közül néhány úgy, hogy összegük K legyen?

A Részletösszeg probléma **NP**-teljes.

Az **NP**-beliség világos, az **NP**-nehézséget a 3Satról való visszavezetéssel igazoljuk.

NP-teljes problémák számokra

3Sat $\leq$ Részletösszeg

- Legyen $\varphi = c_1 \wedge \ldots \wedge c_n$ egy 3CNF, $c_i = \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$.
- A φ-beli változók legyenek $x_1, \ldots, x_m$.
- Ebből elkészítjük a Részletösszeg probléma egy példányát, melyben $n+m$-jegyű (!) számok szerepelnek.
- x_i-ből a t_i és f_i számok készülnek:
  - t_i és f_i első m jegye csupa 0, kivéve az i. jegyet, ami 1-es;
  - t_i utolsó n jegye közül az $m+k$. akkor 1-es, ha c_k-ban szerepel x_i, egyébként 0;
  - f_i utolsó n jegye közül az $m+k$. akkor 1-es, ha c_k-ban szerepel $\neg x_i$, egyébként 0.
- Továbbá, $a_i = b_i = 10^i$ minden $i = 1, \ldots, n$ esetén.
- Az eredmény: a $\{f_i, t_i : 1 \leq i \leq m\} \cup \{a_i, b_i : 1 \leq i \leq n\}$ halmaz és a $K = 11\ldots133\ldots3$ célszám (m darab 1-es, majd n darab 3-as).

Példa

Ha $\varphi = (x_1 \vee \neg x_2 \vee x_3) \ \vee \ (x_1 \vee \neg x_2 \vee x_4) \ \vee \ (\neg x_1 \vee x_2 \vee \neg x_4)$:

$$
\begin{aligned}
t_1 &= 1000110 \\
f_1 &= 1000001 \\
t_2 &= 0100001 \\
f_2 &= 0100110 \\
t_3 &= 0010100 \\
f_3 &= 0010000 \\
t_4 &= 0001010 \\
f_4 &= 0001001 \\
a_1 = b_1 &= 0000100 \\
a_2 = b_2 &= 0000010 \\
a_3 = b_3 &= 0000001 \\
\hline
K &= 1111333
\end{aligned}
$$

- t_1 és f_1 közül pontosan az egyiket kell válasszuk (K első jegye miatt);
- általában t_i és f_i közül is pontosan az egyiket – t_i választása feleljen meg az $x_i = 1$, f_i választása az $x_i = 0$ értékadásnak;
- ezzel az $m + j$. jegyek mindegyike 0, 1, 2 vagy 3 lesz $j = 1, \ldots, n$-re, annak függvényében, hogy c_j-ben 0, 1, 2 vagy 3 literál vált igazzá;
- ha az $m + j$. jegy 3, akkor jó, ha 2, akkor válasszuk ki még mondjuk a_j-t, ha 1, akkor a_j-t és b_j-t is, ha 0, akkor nem tudjuk ezen a jegyen elérni a célszámot

Az Egész Értékű Programozás egy másik speciális esete:

Hátizsák

Adott: N elem mindegyikének w_i súlya és c_i értéke, valamint W és C.

Kérdés: Kiválasztható-e ismétlés nélkül néhány elem úgy, hogy összértékük $\geq C$ és összsúlyuk $\leq W$?

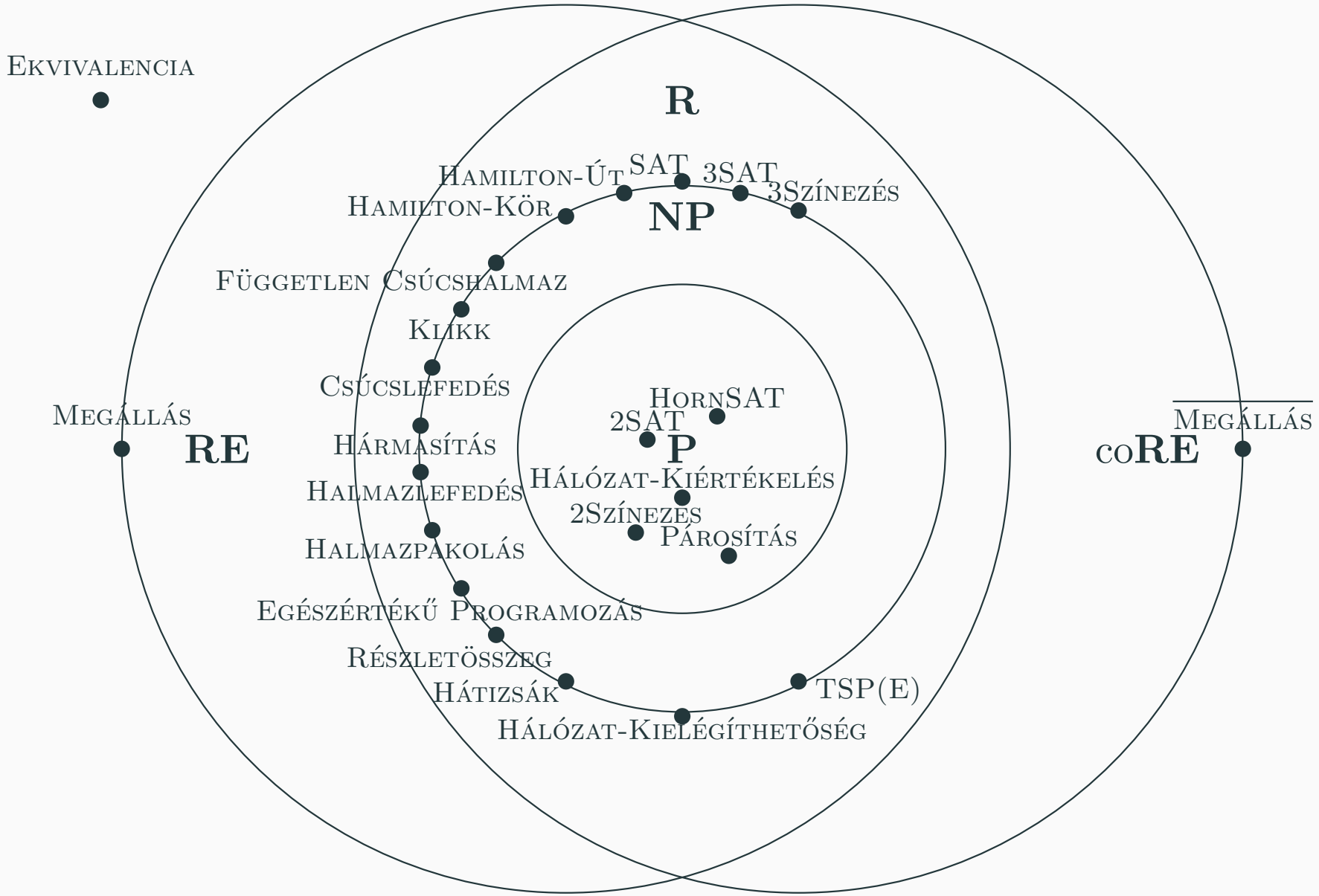
A Hátizsák probléma **NP**-teljes.

Ötlet

A Részletösszeg problémában az a_i értékekből rendre készítsünk egy tárgyat $w_i = c_i = a_i$ súllyal és értékkel, továbbá legyen $C = W = K$, a célszám.

Osztályok és problémák eddig

némelyik probléma még nincs a „végleges helyén”



Néhány további NP-teljes közismert probléma

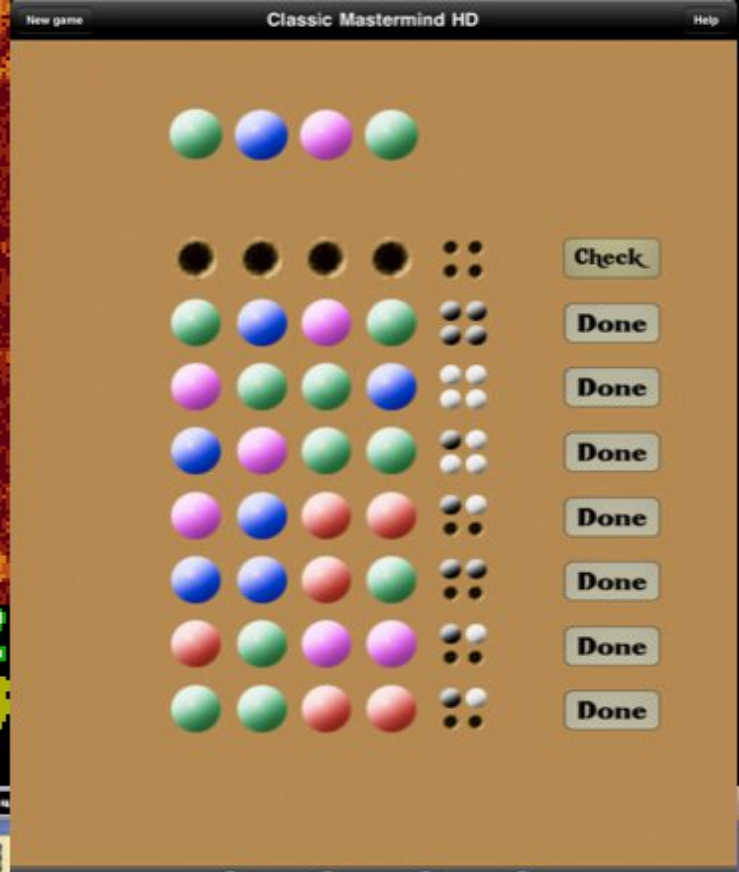
Számos további közismert probléma is **NP**-teljes, mint például:

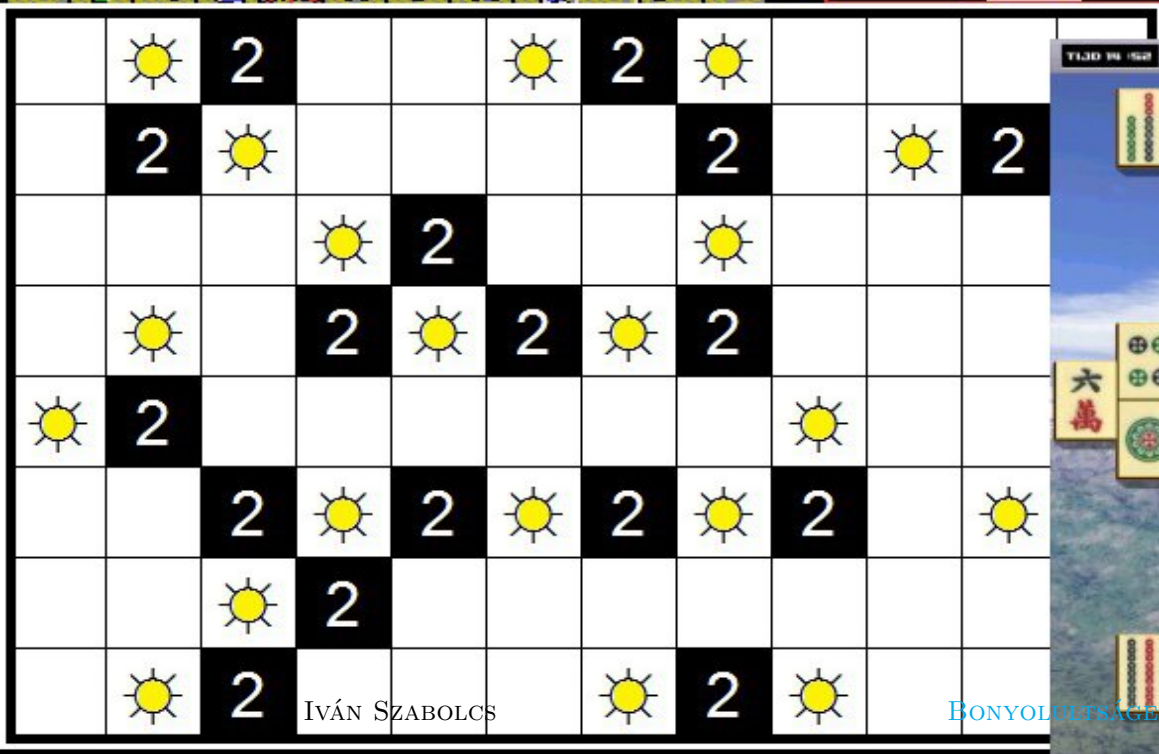
- Adott egy, az Eternity II szabályainak megfelelő csempehalmaz. Kirakható-e a csempékből szabályosan egy négyzet?
- Adott az Aknakereső játék egy részlegesen kitöltött táblája. Be lehet-e fejezni a kitöltést, hogy konzisztens legyen?
- Adott a Super Mario játék egy pályája. Végig lehet-e rajta jutni?
- Adott a Flood-it játék egy pályája. Megoldható-e?
- Adott egy részlegesen kitöltött Sudoku tábla. Konzisztens-e?
- Adott a Tetris játékban elemek egy érkezési sorrendje és egy K szám.
  - El lehet-e tüntetni K sort?
  - Le lehet-e az első K elemet pakolni anélkül, hogy betelne a tábla?
  - Lehet-e K-szor eltüntetni egyszerre négy sort?

Nagyon sok „puzzle”, egyszemélyes játék azért „elég nehéz” ahhoz, hogy addiktív / népszerű legyen, mert **NP**-nehéz. Később majd a kétszemélyeseket is fogjuk tárgyalni.


Eternity II
File Edit Grid ?
2:38 PM
COOLEST GADGETS ON THE WEB.
LABPIXIES
Step 0 / 22
Minesweeper
Game
46
1 32
ptions Help
Score
433
Level
1
Lines
2
High Score
0
Next


OUT 4
IN 0%
TIME 4-52


New game
Classic Mastermind HD
Help
Check
Done
Done
Done
Done
Done
Done
Done



Algoritmus a Hátizsák eldöntésére

A következő rekurzív összefüggéssel számítható $T[i, w]$, ami a legfeljebb az első i tárgy felhasználásával egy w kapacitású hátizsákkal elérhető maximális haszon:

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

Ez ad egy $\mathcal{O}(N \cdot W)$ időigényű algoritmust, ami nem polinom, mert az input mérete $n = N(\log w_i + \log c_i) + \log W$.

dinamikus programozás

Felfoghatjuk úgy is, hogy az algoritmusunk akkor polinomidejű, ha az inputban a súlyokat unárisan reprezentáljuk, vagy ha a súlyok a tárgyak számának egy polinomjával korlátozhatóak.
Ez vezet el az algoritmikus bonyolultságelméletben a pszeudopolinomialitás fogalmához.

Pszeudopolinomiális algoritmusok, erős/gyenge NP-teljesség

Legyen A egy probléma.
Egy A-t eldöntő algoritmus pszeudopolinomiális, ha tetszőleges $(a_1, \dots a_m)$ inputon a futásideje $\mathcal{O}\left(\left(\sum_{1 \leq i \leq m} a_i\right)^k\right)$, valamely konstans k-ra.

Emlékezzünk vissza: az input mérete $\sum_{1 \leq i \leq m} \log a_i$ volt!
tehát nem az input méretéhez képest, hanem az input értékéhez képest kell polinom legyen.

Ha egy **NP**-teljes probléma eldönthető pszeudopolinomiális algoritmussal, úgy gyengén **NP**-teljesnek, ha pedig unáris változata is **NP**-teljes, akkor erősen **NP**-teljesnek nevezzük.

unáris változat: amikor a számok unárisan jönnek be az inputra, pl a 4-et 1111 ábrázolja

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

hiszen unáris számábrázolásnál a polinomiális algoritmus ugyanaz, mint a pszeudopolinomiális: méret=érték

Pszeudopolinomiális algoritmusok, erős NP-teljesség

- A Hátizsák probléma gyengén **NP**-teljes.

  a dinamikus programozós algoritmus pszeudopolinomiális

- A TSP(E) probléma viszont erősen **NP**-teljes.

  hiszen a Hamilton-út $\leq_{\mathcal{P}}$ Tsp(E) visszavezetésnél minden generált szám 1 vagy 2, ezeket unárisan is könnyű kigenerálni ugyanannyi idő alatt

- Pozitív egészek faktorizálására van pszeudopolinomiális algoritmus, de nem ismert rá polinomidejű.

  Pl. a progalapról is ismert „oszd végig a gyökéig mindennel” pszeudopolinomiális

A gyengén **NP**-teljes problémák mindazon példányai gyakorlatilag megoldhatónak tekinthetőek, melyekben az inputon érkező számok értéke korlátozható az input méretének egy polinomfüggvényével.
(Pl. a Hátizsák sok „kisméretű” tárgy esetén.)

Relaxáció

A Hátizsák probléma egészértékű programozási feladatként felírva:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$\sum_{i=1}^{n} c_i x_i \geq C$$

$$0 \leq x_i \leq 1$$

Egy ILP feladat LP-relaxációját kapjuk, ha az egészek helyett a valósak körében oldjuk meg. (Azaz elhagyjuk az „x_i egész" feltételt.)
Mivel $\mathrm{LP} \in \mathbf{P}$, a relaxált feladat hatékonyan megoldható.

nem pont a szimplex algoritmussal, mert az néha exp rosszul viselkedik, de vannak polinomidejű LP solver algoritmusok.

Néhány optimalizálási problémánál a relaxált feladat optimumából egy „elég jó” megoldást lehet előállítani (de korántsem mindnél, és persze kérdés, hogy kinek mi az elég jó).

Töredékes Hátizsák

- Mint a Hátizsák, de a tárgyak törhetők: az i-edik tárgy x_i-ed része, $0 \leq x_i \leq 1$ egy $c_i x_i$ értékű, $w_i x_i$ súlyú tárgy.

  ha a tárgyat félbetöröm, feleződik az értéke. Ez pl. homokra igaz, Mona Lisára kevésbé.

- Erre a feladatba a hatékony mohó algoritmus optimális.

  először a legjobb ár/súly arányút, aztán a következőt stb, amíg fér, utolsót eltörjük.

- Kérdés, hogy mennyire jól „közelíti” az eredeti (függvény)problémát?

  Erre még visszatérünk, de előbb kell pár fogalom.

A következőkben olyan optimalizálási problémákra próbálunk adni hatékony közelítő algoritmusokat, melyeknek eldöntési változata $\mathbf{NP}$-nehéz.

Az f függvény minimalizálási/maximalizálási probléma, ha $f(I) = \arg\min_{s \in S(I)} c(s)$ vagy $\arg\max_{s \in S(I)} c(s)$ alakú, ahol $S(I)$ az I inputhoz tartozó lehetséges megoldások (nemüres) halmaza, c pedig minden megoldáshoz egy valós költséget/értéket rendelő függvény.

azaz: az inputhoz tartozó megoldások közül a legjobb költségűt/hasznút keressük.

Ha f egy optimalizálási probléma, $\alpha > 0$ egy konstans, A pedig egy olyan polinomidejű algoritmus, melyre $A(I) \in S(I)$ minden I inputra úgy, hogy

$$\forall I : \ \max\left\{ \frac{c(A(I))}{c(f(I))}, \frac{c(f(I))}{c(A(I))} \right\} \ \leq \ \alpha,$$

akkor A egy α-approximáló algoritmus f-re.

azaz: „ha az algoritmus által szállított megoldás értéke legfeljebb α-szor rosszabb az optimumnál”

Emlékeztető

Input: $G = (V, E)$ gráf, $K > 0$ egész.

Output: van-e olyan, legfeljebb K méretű X csúcshalmaz G-ben, melyre igaz, hogy G minden élének legalább az egyik végpontja X-beli?

Optimalizálási változat

Input: $G = (V, E)$ gráf.

Output: egy legkisebb lefogó csúcshalmaz G-ben (egy megoldás: egy lefogó csúcshalmaz; költsége: a mérete)

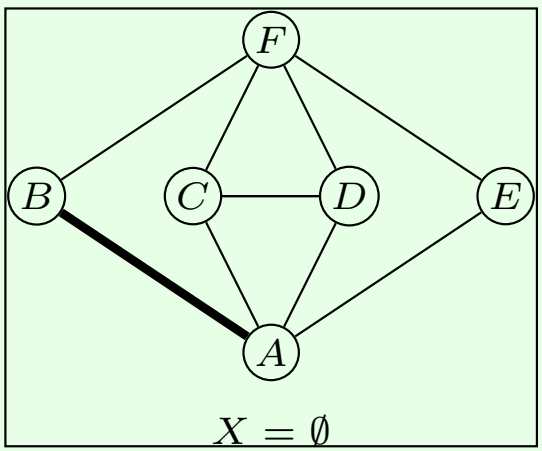
Természetesen ha az optimalizálási változatra lenne egy hatékony módszer, akkor az eldöntésire is. Mivel az eldöntési változat **NP**-teljes, így az optimalizálási változatra sem ismert hatékony algoritmus.

Csúcslefedés

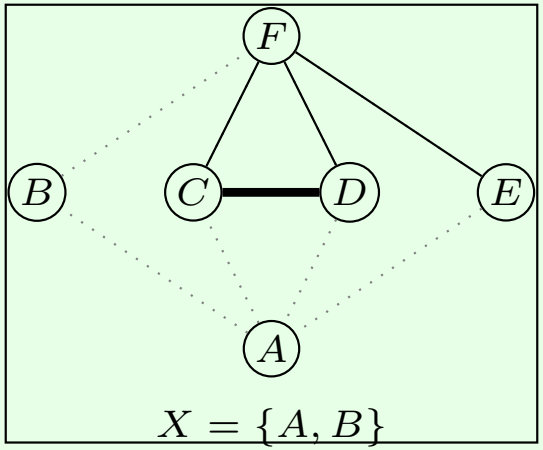
Tekintsük a Csúcslefedés-re a következő egyszerű algoritmust:

- Legyen $X = \emptyset$.
- Amíg van él G-ben:
  - Válasszunk egy tetszőleges e élt.
  - Vegyük be X-be e mindkét végpontját.
  - Vegyük el G-ből az összes, e bármelyik végpontjára illeszkedő élt.
- Adjuk vissza X-et.

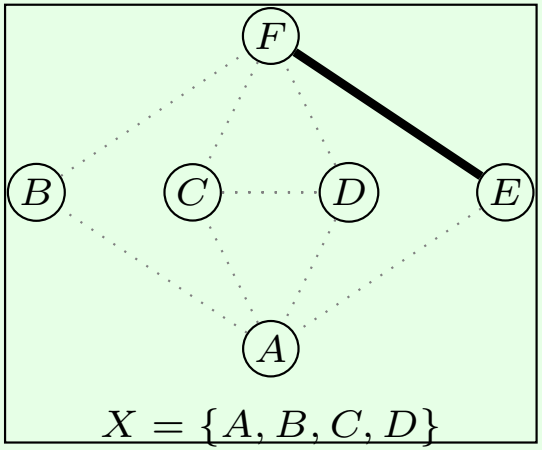
Példa



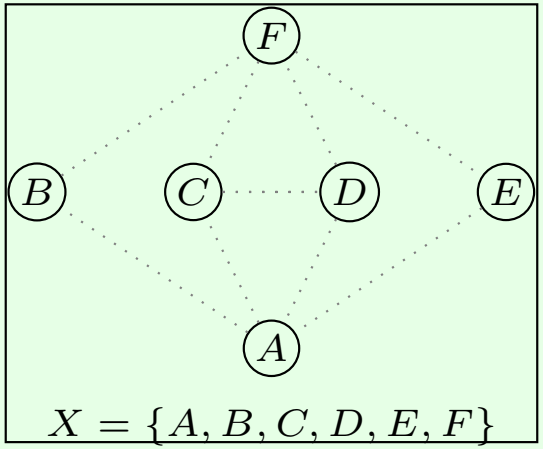
$X = \emptyset$



$X = \{A, B\}$



$X = \{A, B, C, D\}$



$X = \{A, B, C, D, E, F\}$

Így választva az éleket („lexikografikusan”: a legkisebb olyan csúcsot választva, akire még illeszkedik él, azok közül azt, melynek a legkisebb a végpontja) egy 6 méretű csúcshalmazzal fedi le az éleket.
Ha az (A, C), majd az (F, D) élt választaná ki, akkor egy 4 méretűt kapnánk.
Egy optimum pl. az $\{A, C, F\}$ halmaz.

Az előző fólián szereplő algoritmus egy 2-approximáló algoritmus a Csúcslefedésre.

Ötlet

- Az algoritmus nyilván egy lefogó csúcshalmazt ad vissza.
- Ha az X halmazba rendre az e_1, e_2, …, e_k élek végpontjai kerültek bele, akkor $e_1, \ldots, e_k$ egy párosítás G-ben, mindegyiküknek legalább az egyik végpontját le kell fogni, vagyis a minimum legalább k.
- Az algoritmus által visszaadott eredmény pedig $2k$.

A Csúcslefedésre a mai napig nem ismert ennél jobb approximációs algoritmus.

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

Ötlet

Tegyük fel, hogy van egy α-approximáló A algoritmus TSP-re.
Akkor A-t felhasználva meg tudjuk oldani a Hamilton-kört a következőképpen: a $G = (V, E)$ gráfból készítsük $V \times V$-n az alábbi $D(G)$ távolságmátrixot:

$$d_{u,v} = \begin{cases} 1 & \text{ha } (u, v) \in E; \\ |V| \cdot \alpha + 1 & \text{egyébként.} \end{cases}$$

Ekkor ha van Hamilton-kör, az optimum $|V|$;
ha nincs, az optimum legalább $|V| \cdot \alpha + 1$.

Ötlet befejezése

Ekkor a következő algoritmus:

$$\text{Ha } A(D(G)) \leq |V| \cdot \alpha + 1, \text{ fogadjuk el } G\text{-t, egyébként utasítsuk el}$$

eldönti a Hamilton-kör problémát, polinomidőben.

Így várhatóan (hacsaknem $\mathbf{P} = \mathbf{NP}$) nincs approximáló algoritmus sem TSP-re. Kereshetünk viszont olyan speciális esetet, amire van.

Metrikus TSP

A probléma

Input: egy D távolságmátrix, ami teljesíti a háromszögegyenlőtlenséget:

$$\forall i, j, k : \; D_{i,j} + D_{j,k} \; \geq \; D_{i,k}.$$

Output: a minimális összsúlyú körút súlya.

Bonyolultság

Az eldöntési probléma továbbra is **NP**-nehéz.

(Mert ilyen mátrixot állítottunk elő a Hamilton-kör $\mathrm{TSP}(E)$-re való visszavezetésekor: minden súly vagy 1 volt, vagy 2, ez teljesíti a háromszög-egyenlőtlenséget.)

Metrikus TSP: 2-approximáló algoritmus

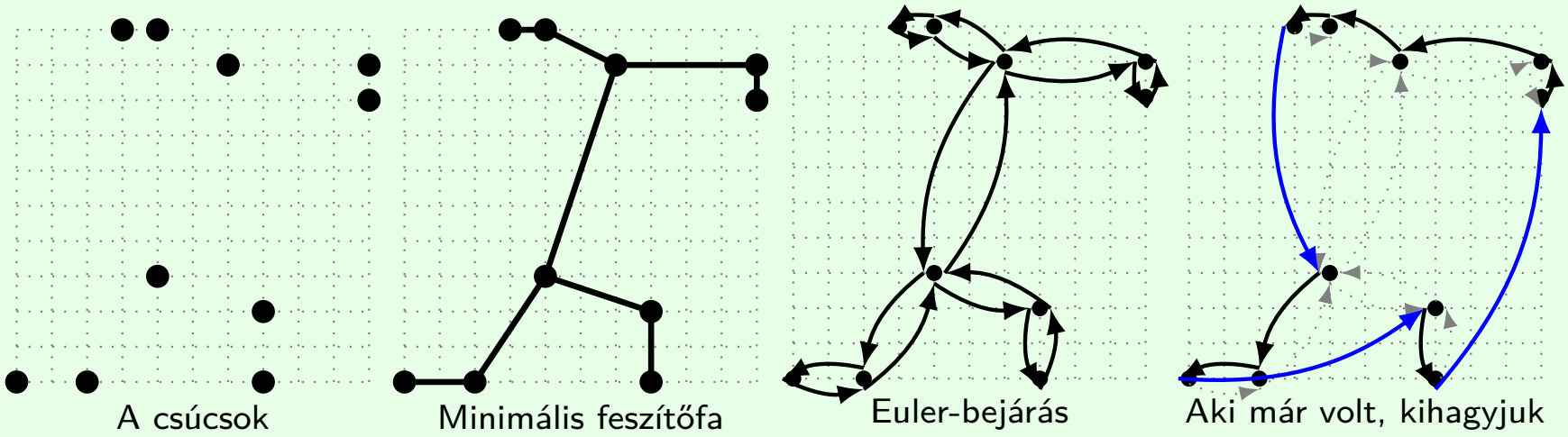
Algoritmus

- Állítsuk elő (Prim vagy Kruskal algoritmusával, például) D egy minimális feszítőfáját.
- Kettőzzük meg a fa éleit.
- A kapott multigráfnak vegyük egy Euler-körvonalát.
- A körvonalból hagyjuk el a korábban már meglátogatott csúcsokat.
- Adjuk vissza az így kapott kör összsúlyát.

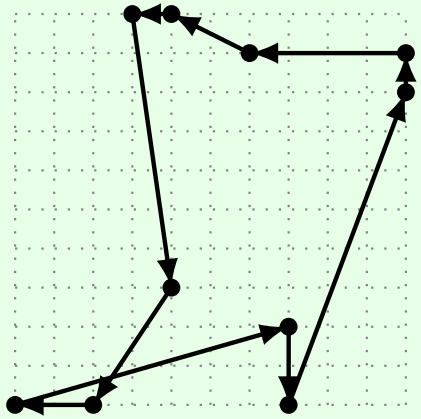
Közelítés

A fenti egy 2-approximáló algoritmus a Metrikus TSP-re.

Példa



Itt a $(10, 8)$ pontból kezdtük el a bejárást, és ennek megfelelően hagytuk el a pontokat.
Az élek hajlítása csak azért van, hogy látsszon a harmadik képen a bejárás; egyenes élekkel a végeredmény:



(nem optimális, pl. azért sem, mert vannak benne egymást keresztező élek, ami javítható lokálisan is.)

Metrikus TSP: 2-approximáló algoritmus

2-approximálás

Az algoritmus 2-approximáló, ez a következőkből áll össze:

- egy tényleges körút költségét adja vissza;
- bármilyen körútból elhagyva egy élt egy feszítőfát kapunk $\Rightarrow$ a minimális feszítőfa összsúlya kisebb, mint a minimális körúté;
- az élek kettőzésével kapott gráfban ill. annak az Euler-körvonalában az élek összsúlya tehát kisebb, mint kétszer a minimális körúté;
- a már látott pontok kihagyásával a háromszög-egyenlőtlenség miatt nem növekszik az összsúly.

Megjegyzés: $\frac{3}{2}$-approximálás

Az élek kettőzése helyett egy „ügyesebb” módszerrel kaphatunk egy $\frac{3}{2}$-approximáló algoritmust is.

Max-2SAT

A probléma

Input: egy $2CNF$, klózonként pontosan két literállal; a literálok változói különböznek.

Output: egyszerre legfeljebb hány klóz elégíthető ki benne?

Tudjuk, hogy a $2\mathrm{SAT}$ probléma $\mathbf{P}$-beli.

Bonyolultság

Ennek az optimalizálási problémának az eldöntési változata (input egy F $2CNF$ és egy K szám, kielégíthető-e F-ben egyszerre K klóz?) $\mathbf{NP}$-teljes.

Először adunk egy randomizált $\frac{4}{3}$-közelítő algoritmust, aztán ezt derandomizálva egy determinisztikusat.

Randomizált algoritmus

Random: várható értékben approximáljon

Egy A randomizált algoritmus α-approximáló, ha várható értéke legfeljebb α-szor rosszabb, mint az optimális, vagyis:

$$\forall x \ \max\left\{\frac{E(A(x))}{f(x)}, \frac{f(x)}{E(A(x))}\right\} \ \leq \ \alpha.$$

Max-2SAT

Ebben az esetben tehát egy olyan algoritmust kell adnunk, mely legalább $\frac{3}{4}X$ klózt kielégít az input formulában, ha legfeljebb X elégíthető ki egyszerre. Ennél többet teszünk: olyan algoritmust fogunk adni, mely (várható értékben) a klózok $\frac{3}{4}$-ét (tehát nem csak az egyszerre kielégíthető klózokét!) igazzá teszi.

Max-2SAT

Randomizált algoritmus

Állítsunk minden változót egymástól függetlenül $\frac{1}{2} - \frac{1}{2}$ valószínűséggel igazra vagy hamisra.

Várható érték

Mivel egy klózban két különböző változó van, így annak esélye, hogy a klóz igaz lesz, $\frac{3}{4}$.
A várható érték additivitása miatt így várható értékben a klózok $\frac{3}{4}$-ét kielégítjük.

Derandomizálás

Most az előző dián szereplő véletlen algoritmusban csökkentjük a generálandó véletlen bitek számát: derandomizálunk.

Determinisztikus algoritmus

Legyenek $x_1, x_2, \ldots, x_n$ az input formulában szereplő változók.

- Először értéket adunk x_1-nek, majd x_2-nek, ..., végül x_n-nek, az i-edik menetben x_i-nek.
- Az i-edik menetben $x_1, \ldots, x_{i-1}$ értéke már rögzítve van.
- Számítsuk ki $b = 0, 1$-re, hogy mennyi klóz elégülne ki várható értékben, ha x_i értékét b-re választjuk, $x_{i+1}, \ldots, x_n$ értékét pedig $\frac{1}{2} - \frac{1}{2}$ valószínűséggel, függetlenül választjuk 0-nak ill. 1-nek.
- Amelyik érték nagyobb lett, arra állítjuk ténylegesen x_i értékét, és folytatjuk a ciklust.

Derandomizálás

Vegyük észre, hogy a várható értéket determinisztikusan ki tudjuk számítani.
Annak esélye, hogy egy klóz értéke igaz, a következőképp számítható:

- ha a klózban van olyan literál, melynek értékét 1-re rögzítettük, akkor 1;
- különben $1 - \frac{1}{2^k}$, ahol $0 \leq k \leq 2$ a klózban levő, még nem rögzített értékű változók száma.

A formulában igazra állított klózok értéke ezek összege.
Azt is be lehet könnyen látni, hogy minden menetben a várható érték nem csökkenhet, így az utolsó menet végére a „várható érték” továbbra is legalább a klózok $\frac{3}{4}$-e lesz.

Max-2SAT példa

Példa

$$\begin{aligned} F \quad = \quad & (x_1 \vee x_2) \ \wedge \ (x_1 \vee \neg x_2) \ \wedge \ (\neg x_1 \vee x_2) \ \wedge \ (\neg x_1 \vee \neg x_2) \ \wedge \\ & (\neg x_1 \vee x_3) \ \wedge \ (\neg x_1 \vee \neg x_3) \ \wedge \ (x_2 \vee \neg x_3) \ \wedge \ (x_2 \vee x_4) \ \wedge \\ & (\neg x_2 \vee x_3) \ \wedge \ (\neg x_2 \vee \neg x_3) \ \wedge \ (x_3 \vee x_4) \ \wedge \ (\neg x_3 \vee \neg x_4) \end{aligned}$$

Ha $x_1 = 0$, a várható értékek: $\frac{1}{2}$, $\frac{1}{2}$, 1, 1, 1, 1, a többi $\frac{3}{4}$, összesen 9.5;
ha $x_1 = 1$, a várható értékek 1, 1, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, a többi $\frac{3}{4}$, összesen 8.5;
így legyen $x_1 = 0$.
Ezek után ha $x_1 = 0$ és $x_2 = 0$, a várható értékek 0, 1, 1, 1, 1, 1, $\frac{1}{2}$, $\frac{1}{2}$, 1, 1, $\frac{3}{4}$ és $\frac{3}{4}$, összesen 9.5;
ha pedig $x_1 = 0$ és $x_2 = 1$, a várható értékek 1, 0, 1, 1, 1, 1, 1, 1, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{3}{4}$ és $\frac{3}{4}$, szintén 9.5. Legyen mondjuk $x_2 = 1$.
. . . és így tovább. . .

Könnyebb persze számolni, ha mindent felszorzunk 4-gyel és csak azokat összegezzük, akikben szerepel az aktuális döntési változó és nem 1 még az értéke.

Töredékes Hátizsák

Algoritmus a Töredékes Hátizsákra

Erre a változatra a mohó algoritmus optimális:

- rendezzük csökkenőbe a tárgyakat $\frac{c_i}{w_i}$ szerint;
- eszerint a sorrend szerint vegyünk be annyi tárgyat teljesen, amennyit csak tudunk;
- az első tárgyat, ami nem fér be, pedig „törjük” úgy, hogy a töredék megtöltse a hátizsák maradék kapacitását.

Approximálás?

Adódik a következő ötlet a Hátizsák problémára:

- rendezzük csökkenőbe a tárgyakat $\frac{c_i}{w_i}$ szerint;
- eszerint a sorrend szerint vegyünk be annyi tárgyat, amennyit csak tudunk.

Ez az algoritmus nem α-approximálja a Hátizsák problémát, semmilyen α konstansra.

Példa, amikor nem jól közelít, ha truncoljuk a mohót

Ha pl. két tárgyunk van, $w_1 = 1$, $c_1 = 1$, $w_2 = W$ és $c_2 = W - 1$:

- az első tárgy fajlagosan értékesebb, mint a második
- a töredékes változat optimumhelye $(1, \frac{W-1}{W})$, értéke $1 + \frac{(W-1)^2}{W}$
- a mohó algoritmus lefele kerekítő változata tehát az $(1, 0)$ helyen felvett 1 értéket adja vissza
- az optimumhely $(0, 1)$, értéke $W - 1$.

Ha tehát W elég nagy, $1 \cdot \alpha < W - 1$ lesz, így az algoritmus ezen változata nem α-approximáló.

Kis módosítás

A következő algoritmus viszont 2-approximálja a Hátizsák problémát:

- Tegyük a tárgyakat fajlagos érték szerint csökkenő sorrendbe: $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}$.
- Számítsuk ki a Töredékes Hátizsák optimumhelyét: $(1, 1, \ldots, 1, x, 0, 0, \ldots, 0)$, ahol x a k. tárgy (amelyiket el kellett törnünk, mert már nem fért be).
- Adjuk vissza vagy $\sum_{i=1}^{k-1} c_i$-t, vagy c_k-t, amelyik nagyobb.

Tehát: vagy fajlagosan csökkenő sorrendben beteszünk, amit csak lehet, vagy az így éppen kimaradó tárgyat egyedül rakjuk be, amelyik jobb.

Hátizsák 2-approximálható

Amiért ez legalább az optimum felét adja:

- a töredékes hátizsák optimuma ennek a két értéknek az összegénél kisebb, így a kettő közül a nagyobb legalább a töredékes optimumának fele;
- a töredékes hátizsák optimuma (mivel relaxált változat) legalább akkora, mint a $0/1$ hátizsáké.

Pár kimaradt részlet

Az algoritmus így akkor nem működik, ha minden tárgy befér egyszerre (ekkor az lesz az optimumhely), vagy ha a k. tárgy egyedül se férne be (de az ilyen tárgyakat eleve nem kell felvegyük a listába), de ezek könnyen javíthatók.

Hátizsák $1+\epsilon$-approximálható

Nézzünk egy másik kézenfekvő algoritmust a Hátizsák probléma közelítő megoldására:

Skálázás ϵ-ra

- Legyen $w_1, \ldots, w_n, c_1, \ldots, c_n, W$ a Hátizsák probléma egy inputja és $\epsilon > 0$ egy valós szám.
- Oldjuk meg a $w_1, \ldots, w_n, c'_1, \ldots, c'_n, W$ feladatot dinamikus programozással, ahol $c'_i = \left\lfloor \frac{c_i}{c_{\max}} \cdot \left\lfloor \frac{n}{\epsilon} \right\rfloor \right\rfloor$, $c_{\max} = \max_i c_i$ és adjuk vissza ennek az eredményét.

Tehát mintha a maximális hasznú tárgy haszna $\left\lfloor \frac{n}{\epsilon} \right\rfloor$ lenne, a többi haszna pedig evvel arányosan skálázódna (egészrészt véve).

pl. ha $\epsilon = 0.01$, akkor a legdrágább tárgy új értéke $100n$ lesz, a többi meg ilyen arányban leosztva, lefelé kerekítve. Az eredetileg fele olyan drága tárgy új értéke $50n$ lesz, stb.

Hátizsák $1+\epsilon$-approximálható

A dinamikus algoritmus

A rekurzív összefüggés most:

$$T[i,c] = \begin{cases} 0 & \text{ha } c \leq 0; \\ \infty & \text{ha } 0 = i < c; \\ \min\{T[i-1,c], T[i-1,c-c_i]+w_i\} & \text{egyébként.} \end{cases}$$

„Mekkora zsák kell legalább, hogy az első i tárgyból meglegyen legalább c haszon?"

most a második tengelyen érték van, a korábbi algoritmusnál súly volt

Időigény

Ennek az algoritmusnak az időigénye

$O(\text{poly}(n, n \max c, \log W)) = O(\text{poly}(n \max c \log W))$. (Pszeudopolinomiális.)

Hátizsák $1+\epsilon$-approximálható

de a skálázás után ez már polinomiális lesz!

$O(\text{poly}(n \max c \log W))$

Ha $\max c = \lfloor \frac{n}{\epsilon} \rfloor$, akkor ez polinom időigény az eredeti input méretében.
Persze nem mindig szolgáltat optimális eredményt, az osztás utáni egészrész vételekor vesztett pontosság miatt.
De ha rögzítjük ϵ-t, akkor polinom időigényű és $(1+\epsilon)$-approximáló algoritmus!

az $(1+\epsilon)$-szoros eltérés az optimumtól a kerekítés miatt jöhet be, ezt nem bizonyítjuk be, az elv a fontos

A gyakorlatban is azt szeretjük, ha a megrendelő tud mondani egy hibakorlátot, amivel már elégedett (pl. „ha gyorsan ki tudtok számolni egy olyan megoldást, ami csak garantáltan 1%-kal kerül többe, mint az optimum, az már nekem jó”, itt $\epsilon = 0.01$, van akinek kisebb ϵ kell, van akinek nagyobb is jó). Ennek van neve is:

PTAS, FPTAS

PTAS

Azt mondjuk, hogy az f optimalizálási problémára van polinomidejű approximációs séma, avagy PTAS, ha van olyan algoritmus, melynek f inputja és egy $\epsilon > 0$ szám a bemenete és tetszőleges rögzített ϵ-ra $(1+\epsilon)$-approximáló polinomidejű algoritmus.

FPTAS

Azt mondjuk, hogy az f optimalizálási problémára van teljesen polinomidejű approximációs séma, avagy FPTAS, ha van rá olyan PTAS, mely tetszőleges (x, ϵ) inputra $\mathrm{poly}(|x|, \frac{1}{\epsilon})$ időben fut.

Az előző dián pl. egy FPTAS-t láttunk a Hátizsák problémára.

Ha egy problémára van FPTAS, azzal tetszőleges inputra tetszőleges pontossággal polinomidőben meg tudjuk határozni az optimumot.

FPTAS és pszeudopolinomiális algoritmusok

Ha egy egészértékű optimalizálási probléma

I) optimuma korlátozható az inputméret egy polinomjával

II) és van rá FPTAS,

akkor van rá pszeudopolinomiális algoritmus.

Ötlet

Ha a fenti korlát n^c, akkor $\epsilon = \frac{1}{n^c}$ megfelelő választás lesz.

Következmény

Ha $\mathbf{P} \neq \mathbf{NP}$ és egy, a fenti I) tulajdonsággal rendelkező probléma eldöntési változata erősen $\mathbf{NP}$-teljes, akkor nem lehet rá FPTAS.

A coNP osztály

$$\mathbf{coNP} = \{A : \overline{A} \in \mathbf{NP}\}$$

a co operátor erről szólt: co$\mathcal{C}$-ben a $\mathcal{C}$-beli problémák komplementerei vannak

Példák

ÉRVÉNYESSÉG

Adott: φ Boole-formula

Kérdés: Érvényes-e, azaz azonosan igaz-e φ?

$\overline{\text{HAMILTON-ÚT}}$

Adott: G gráf

Kérdés: Igaz-e, hogy G nem tartalmaz Hamilton-utat?

Az ÉRVÉNYESSÉG és $\overline{\text{HAMILTON-ÚT}}$ problémák co**NP**-ben vannak.

ezeknél nem az „igen” példányokhoz van „tanú”, hanem a „nem” példányokhoz „cáfolat”

NP és coNP

Egy probléma akkor van

- **NP**-ben, ha minden igen példányához létezik polinom méretű, polinom időben ellenőrizhető bizonyíték, és csak az igen példányokhoz létezik ilyen bizonyíték.
- co**NP**-ben, ha minden nem példányhoz létezik polinom méretű, polinom időben ellenőrizhető cáfolat, és csak a nem példányokhoz létezik ilyen cáfolat.

NP $\cap$ co**NP**-ben pedig ezek szerint akkor, ha az „igen” példányokhoz is létezik tanú és a „nem” példányokhoz is létezik cáfolat, melyeket ha valaki megad nekünk, gyorsan tudjuk ellenőrizni, hogy tényleg tanú/cáfolat-e (a faktorizálásra fogunk látni ilyet).

ebből még nem következik, hogy determinisztikusan legyártani is könnyű lenne bármelyiket!

Néhány további eredmény a P és NP osztályokról

Eddig minden $\mathbf{NP}$-beli problémánkról vagy azt mutattuk meg, hogy $\mathbf{P}$-ben van, vagy azt, hogy $\mathbf{NP}$-teljes.
Azonban...

Tétel (Ladner, 1975)

Ha $\mathbf{P} \neq \mathbf{NP}$, akkor létezik olyan $L \in \mathbf{NP} - \mathbf{P}$, mely nem $\mathbf{NP}$-teljes.

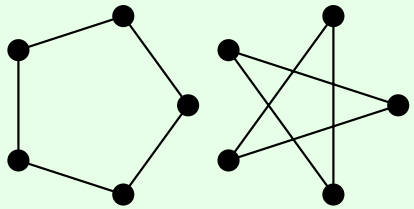
tehát akkor van, aki $\mathbf{NP}$-n „belül", de $\mathbf{P}$-n „kívül" lakik

az ilyen problémákat hívják „$\mathbf{NP}$-köztes" ($\mathbf{NP}$-intermediate, $\mathbf{NPI}$) problémának

P és NPC közt?

Gráf-izomorfizmus

Input: két gráf.
Output: izomorfak-e?



ez a két gráf pl. egy „igen” példány

Hogy **NP**-beli, az világos: csak nemdet meg kell feleltetni egymásnak a csúcsokat és ellenőrizni, hogy ugyanott mennek-e az élek, az „igen” válaszra egy „éltartó bijekció” a tanú

Faktorizálás(E)

Input: $N, K > 0$ egészek.
Output: Van-e N-nek K-nál kisebb prímosztója?

A fenti két **NP**-beli probléma egyikéről sem ismert, hogy **P**-beliek-e, sem az, hogy **NP**-teljesek lennének.

mindkettőről az az „elfogadottabb” sejtés, hogy **NPI**-ben vannak

A Faktorizálás

A Faktorizálásra persze van pszeudopolinomiális algoritmus (végigosztjuk N-t K-ig a számokkal), tehát ha $\mathbf{P} \neq \mathbf{NP}$, akkor nem lehet erősen $\mathbf{NP}$-teljes; másfelől $\mathrm{co}\mathbf{NP}$-beli is, így ha $\mathbf{NP} \neq \mathrm{co}\mathbf{NP}$, nem lehet $\mathbf{NP}$-teljes.

Faktorizálás$(\mathbf{E}) \in \mathbf{NP} \cap \mathrm{co}\mathbf{NP}$

- Nemdeterminisztikusan generálunk legfeljebb $\log N$ darab, egyenként legfeljebb $\log N$-bites számot — megsejtjük a prímtényezős felbontást
- Determinisztikusan ellenőrizzük, hogy prímszámokat generáltunk-e és hogy a szorzatuk épp N-e, ha nem, akkor ez a szál rejectel

  A prímtesztelés $\mathbf{P}$-ben van, azt tudjuk. Legalább egy szál sikeresen kigenerálja a felbontást.
- Faktorizálás: akkor acceptelünk, ha van köztük K-nál kisebb szám.
- $\overline{\text{Faktorizálás}}$: akkor acceptelünk, ha nincs.

itt most tehát a prímtényezős felbontás egyszerre tanú az „igen” és cáfolat a „nem” példányokra, tudjuk, hogy létezik, de még nem ismert rá olyan determinisztikus algoritmus, ami polinomidőben legyártja

A co operátor

A co operátor monoton: ha $\mathcal{C} \subseteq \mathcal{C}'$, akkor $\mathrm{co}\mathcal{C} \subseteq \mathrm{co}\mathcal{C}'$.

Legyen $\mathcal{C} \subseteq \mathcal{C}'$ és $A \in \mathrm{co}\mathcal{C}$. A co defje szerint tehát $\overline{A} \in \mathcal{C}$. Mivel $\mathcal{C} \subseteq \mathcal{C}'$, így $\overline{A} \in \mathcal{C}'$. Megint a co defje szerint $A \in \mathrm{co}\mathcal{C}'$, tehát $\mathrm{co}\mathcal{C} \subseteq \mathrm{co}\mathcal{C}'$.

Következmény

$$\mathbf{P} \subseteq \mathbf{NP} \cap \mathrm{co}\mathbf{NP}.$$

- Tudjuk, hogy $\mathbf{P} \subseteq \mathbf{NP}$.
- Az előzőből akkor $\mathrm{co}\mathbf{P} \subseteq \mathrm{co}\mathbf{NP}$.
- Mivel $\mathbf{P} = \mathrm{co}\mathbf{P}$ (determinisztikus algoritmusoknál elég negálni a választ visszatéréskor), ez azt jelenti, hogy $\mathbf{P} \subseteq \mathrm{co}\mathbf{NP}$.
- Tehát $\mathbf{P} \subseteq \mathbf{NP}$ és $\mathbf{P} \subseteq \mathrm{co}\mathbf{NP}$, ez maga az állítás.

A co operátor

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

Mivel A inputjai ugyanazok, mint $\overline{A}$ inputjai és B inputjai is megegyeznek $\overline{B}$ inputjaival, így f egy (továbbra is polinomidejű, ha eredetileg az volt) inputkonverzió A-ból B-be.
A választ is tartja, hiszen $\overline{A}$ minden I inputjára

$$\overline{A}(I) = \overline{A(I)} = \overline{B(f(I))} = \overline{B}(f(I)).$$

Következmény

Ha A $\mathcal{C}$-nehéz, akkor $\overline{A}$ co$\mathcal{C}$-nehéz.

Az Érvényesség és a $\overline{\text{Hamilton-Út}}$ problémák co**NP**-teljesek.

NP és coNP

Ha $\mathcal{C}$ zárt a visszavezetésre és A $\mathcal{C}$-teljes, akkor $\mathcal{C} = \text{co}\mathcal{C} \Leftrightarrow A \in \text{co}\mathcal{C}$.

$\Rightarrow$ Ha $\mathcal{C} = \text{co}\mathcal{C}$ és A $\mathcal{C}$-teljes, tehát $A \in \mathcal{C}$, akkor nyilván $A \in \text{co}\mathcal{C}$ is.

$\Leftarrow$
- Ha $A \in \text{co}\mathcal{C}$, akkor $\overline{A} \in \mathcal{C}$.
- Ha A $\mathcal{C}$-teljes, akkor $\overline{A}$ $\text{co}\mathcal{C}$-teljes. Tehát tetszőleges $B \in \text{co}\mathcal{C}$-re $B \leq \overline{A}$.
- Mivel $\mathcal{C}$ zárt a visszavezetésre és $\overline{A} \in \mathcal{C}$, emiatt tetszőleges $B \in \text{co}\mathcal{C}$ szintén $\mathcal{C}$-ben van.
- Tehát $\text{co}\mathcal{C} \subseteq \mathcal{C}$. Alkalmazva a monotonitást: $\text{coco}\mathcal{C} \subseteq \text{co}\mathcal{C}$, de persze $\text{coco}\mathcal{C} = \mathcal{C}$, tehát $\mathcal{C} \subseteq \text{co}\mathcal{C}$, így a két osztály egyenlő.

Következmény

$$\mathbf{NP} = \text{co}\mathbf{NP} \Leftrightarrow \textsc{Sat} \in \text{co}\mathbf{NP} \Leftrightarrow \textsc{Érvényesség} \in \mathbf{NP}.$$

tehát pl. ha a kielégíthetetlenségre lenne $\mathbf{NP}$ algoritmus, akkor $\mathbf{NP} = \text{co}\mathbf{NP}$.

(A rezolúcióról tudjuk, hogy nem $\mathbf{NP}$, túl hosszú lehet egy levezetés.)

Determinisztikus algoritmusokkal...

- 3SAT eldönthető $\mathcal{O}(1.3303^n)$ időben (Makino, Tamaki, Yamamoto, 2011)
- 3-SZÍNEZÉS eldönthető $\mathcal{O}(1.3289^n)$ időben (Beigel, Eppstein, 2005)
- FÜGGETLEN CSÚCSHALMAZ eldönthető $\mathcal{O}(1.2108^n)$ időben (Robson, 1986 – Fomin, Grandoni, Kratsch 2009)
- ...

ugyanakkor

- FAKTORIZÁLÁS(E) eldönthető $\mathcal{O}(c^{\sqrt[3]{n \cdot \ln^2 n}})$ időben valamilyen c konstansra (Lenstra et al, 1993. Ez már szubexponenciális: a kitevő $o(n)$)
- GRÁF-IZOMORFIZMUS eldönthető $\mathcal{O}(e^{e^{\sqrt{\log n}}})$ időben (Babai, 2017. Ez is szubexponenciális)

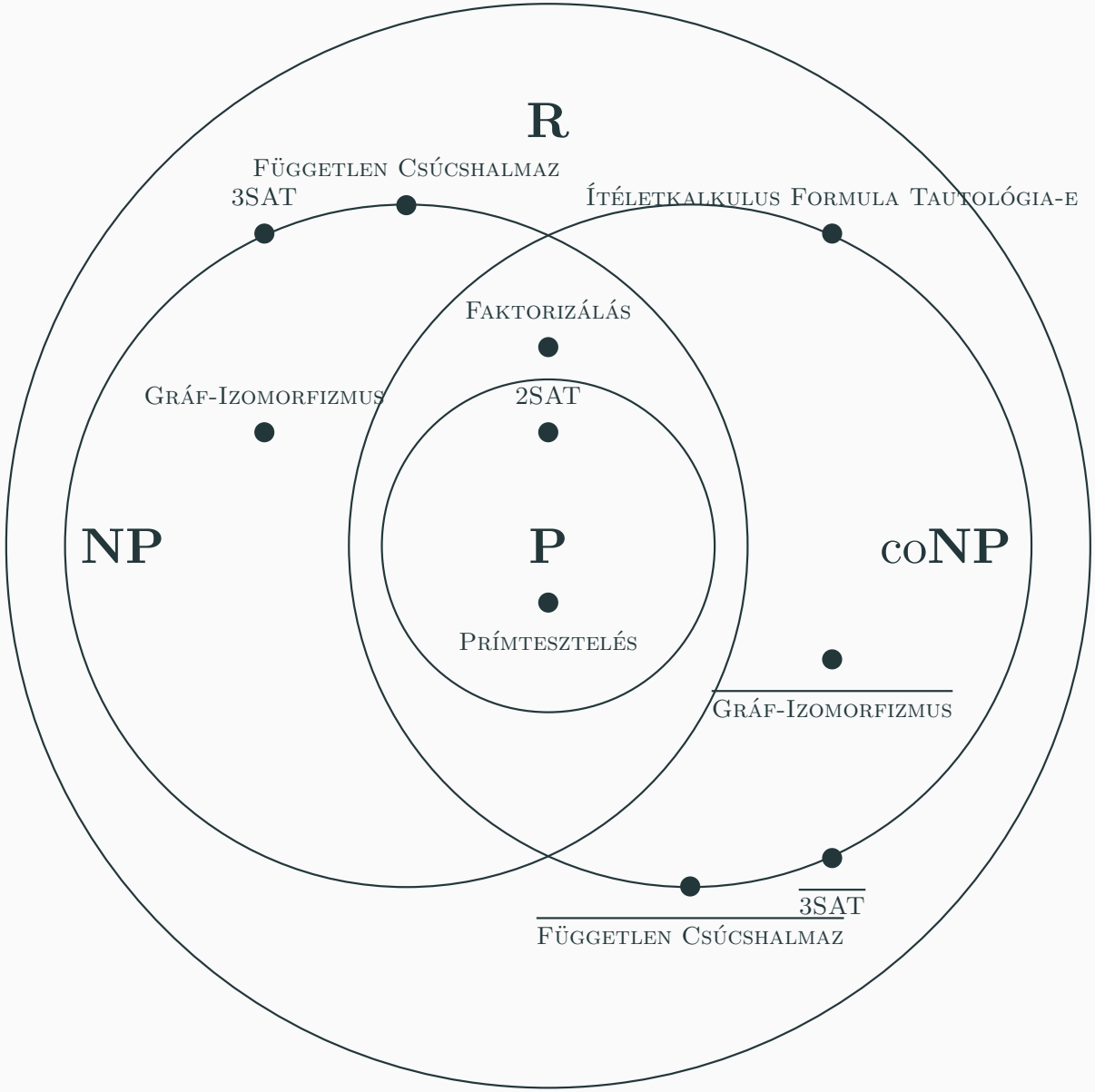
Az Exponenciális Időhipotézis azt a sejtést fogalmazza meg, hogy a 3SAT-ot (és sok más **NP**-teljes problémát) nem lehet szubexponenciális időben megoldani.

Egy osztálykép néhány problémával

a „legelfogadottabb” sejtésekkel

RE

- Hilbert 10. problémája
- Post Megfelelkezési Problémája
- Megállás
- Elsőrendű Formula Tautológia-E



Tárbonyolultság

A futásidő mellett a felhasznált tárterület a másik fontos erőforrás.

Ismét igaz, hogy egy RAM-program esetében ha csak a használt cellák számát vennénk alapul, irreálisan alacsony tárigény-fogalomhoz jutnánk.

Cellánként

Egy memóriacella igénye egy adott inputon: a benne a program futása során tárolt legnagyobb szám bináris alakjának a hossza.

(Azaz $1 + \lfloor \log a \rfloor$, ha ez a maximális érték a, kivéve, ha $a == 0$, mely esetben 1).

PLUSZ a cella címének bináris alakjának a hossza.

Példa

Ha a $W[7]$ regiszterben a program futása során a legnagyobb érték 20, akkor ennek a regiszternek a tárigénye 5 (a $20 = 10100_2$ string hossza) plusz 3 (a $7 = 111_2$ string hossza) $= 8$.

Miért is?

Ha egy (elviekben) végtelen sok regiszterrel rendelkező RAM architektúrát realisztikus architektúrán akarunk szimulálni (pl. Javában), akkor a nem-nulla regiszterek tartalmát tárolnánk egy HashMap-ben, mondjuk.
A HashMap pedig kulcs-érték párok halmaza: el kell tárolnunk a kulcsot is (ami a cella címe) és az értéket is (ami a cella tartalma), ezért definiáljuk így a tárigényt.

Tárbonyolultság

Tárigény egy adott inputon

A teljes program tárigénye egy adott inputon: az összes, a futás során használt cella tárigényének összege.

A két memóriamodell

Amint egy cellát megcímez a program (direkt vagy indirekt címzéssel, írásra vagy olvasásra), használtnak minősül.
Kivéve a következő esetet:

- ha a program az input regisztereket csak olvassa,
- az output regiszterekbe pedig csak ír, ráadásul stream módban: először $O[1]$ kap értéket, majd $O[2]$, $O[3]$ stb.

akkor az input és output regiszterek tárigényét nem kell bevenni a tárigénybe.
(Ez az ún. „offline" vagy „lyukszalagos" eset.)
(Ez megfelel annak a memóriamodellnek, mikor a hívó fél biztosítja az I/O területet: az inputot nem írhatjuk át, az outputot pedig egy output stream formájában kapjuk.)

Tárbonyolultság

A program tárigénye

A program tárigénye az $f(n) : \mathbb{N} \to \mathbb{N}$ függvény, ha bármely, legfeljebb n méretű inputon legfeljebb $f(n)$ tárat használ.

A probléma tárigénye

Egy probléma eldönthető $\mathcal{O}(f(n))$ tárban, ha van őt eldöntő, $\mathcal{O}(f(n))$ tárigényű program. Ezen problémák osztályát $\textsc{Space}(f(n))$ jelöli.

Példa

$\textsc{Elérhetőség}$ eldönthető lineáris tárban:

- a szélességi keresés algoritmusa egy N-csúcsú gráf esetén két, egyenként legfeljebb N méretű csúcshalmazt tárol (a már elért ill. elért és kifejtett csúcsokét), ez pl. N hosszú bitvektor alkalmazásával egy-egy regiszterben $\mathcal{O}(N)$ tárban megoldható.

note: ezért nem lenne jó a „használt regiszterek száma” mint mérőszám, akkor ilyenek jönnének ki, mint pl. „az elérhetőség megoldható konstans tárban”, ami nem passzol a való világbeli tárigény fogalomhoz

Alapvető különbség tár és idő közt

A tár újrafelhasználható!

a lineáris időigény jónak számított, a lineáris tárigény nem annyira:

Lineáris tárban...

- ...eldönthető a HAMILTON-ÚT probléma is: ugyanazt a területet újrahasználva generáljuk a csúcsok összes permutációját
- ...eldönthető a $\text{TSP}(E)$ probléma is, hasonlóképp
- ...eldönthető a 3-SZÍNEZÉS probléma is: ugyanazt a területet újrahasználva generáljuk a csúcsok összes 3-színezését
- ...

Nem ismert, hogy **NP** és $\text{SPACE}(n)$ közt mi az összefüggés.
Csak annyit tudunk biztosan, hogy $\mathbf{NP} \neq \text{SPACE}(n)$.
(mert **NP** zárt a visszavezetésre, $\text{SPACE}(n)$ pedig – látni fogjuk – nem az.)

Nemdeterminisztikus tárbonyolultság

Nemdeterminisztikus program időbonyolultságát a leghosszabb szál hosszaként definiáltuk.

Nemdeterminisztikus program tárbonyolultságát egy adott inputon hasonló módon: a legtöbb tárat használó szál tárigényeként definiáljuk.

A nemdeterminisztikus program tárbonyolultsága szintén akkor $f(n)$, ha tetszőleges, legfeljebb n méretű inputon legfeljebb $f(n)$ tárat használ.

A nemdeterminisztikus programmal $\mathcal{O}(f(n))$ tárban eldönthető problémák osztályát $\mathrm{NSpace}(f(n))$ jelöli.

Nemdeterminisztikus tárbonyolultság

Példa: ELÉRHETŐSÉG $\in$ NL

function ND-ELER($G[1 \ldots N][1 \ldots N]$,$s$,$t$) az input egy gráf és két csúcsa

$u := s$

while $u \neq t$ **do**

$v := \text{ND}(1 \ldots N)$ v értéke nemdet. 1 és N közti egész: egy nemdet választott csúcs

if $G[u][v] == 0$ **then return** FALSE

$u := v$

return TRUE

Tárigény: nemdeterminisztikus logaritmikus: $\mathcal{O}(\log n)$.

Az inputot csak olvassa, output nincs, két munkaváltozót használ, amikben 1 és N közti számok vannak.

Eldönti az ELÉRHETŐSÉG problémát.

(Egy további, N-ig menő ciklusszámlálóval a végtelen ciklus elkerülhető, továbbra is logaritmikus tárban.)

Általában a „logaritmikus tárigény”re úgy lehet érdemes gondolni, mint az olyan algoritmusokra, melyek az inputot csak olvassák és lokális változóként csak konstans sok pointert ill. számlálót használnak; a számlálók csak polinomig számolnak. Mint itt is, két pointerünk van tkp a gráfnak egy-egy csúcsára, u és v, meg esetleg egy lépésszámlálónk, mely maximum N-ig fut.

A nevesített bonyolultsági osztályok

- $\mathbf{R}$, $\mathbf{RE}$ – eldönthető ill. felismerhető problémák
- $\textsc{Time}(f(n))$, $\textsc{NTime}(f(n))$ – det/nemdet. $\mathcal{O}(f(n))$ időben eldönthető problémák
- $\textsc{Space}(f(n))$, $\textsc{NSpace}(f(n))$ – det/nemdet. $\mathcal{O}(f(n))$ tárban eldönthető problémák
- $\mathbf{P} = \textsc{Time}(n^k) = \bigcup_{k \geq 0} \textsc{Time}(n^k)$ – polinomidőben eldönthető problémák
- $\mathbf{NP} = \textsc{NTime}(n^k)$ – nemdet. polinomidőben eldönthető problémák
- $\mathbf{L} = \textsc{Space}(\log n)$ – logaritmikus tárban eldönthető problémák
- $\mathbf{NL} = \textsc{NSpace}(\log n)$ – nemdet. logtárban eldönthetőek
- $\mathbf{PSPACE} = \textsc{Space}(n^k)$ – polinom tárban eldönthetőek
- $\mathbf{NPSPACE} = \textsc{NSpace}(n^k)$ – nemdet. polinom tárban...
- $\mathbf{EXP} = \textsc{Time}(2^{n^k})$... ebből a tárgyból ez az „exponenciális”!

Alapvető összefüggések

Alapvető összefüggések bonyolultsági osztályok közt

i) $\mathrm{TIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$, $\mathrm{SPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$

ii) $\mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f^2(n))$

iii) $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(k^{f(n)})$

Ötletek

Az i) pont világos: egy determinisztikus algoritmust felfoghatunk úgy is, mint egy nemdeterminisztikus algoritmust, mely sosem ágazik el. Tár- és időigénye ugyanannyi marad, akár így nézünk rá, akár úgy.

Alapvető összefüggések

$\mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f^2(n))$

Tehát egy $f(n)$ időkorlátos N nemdeterminisztikus programot lehet szimulálni $f^2(n)$ tárkorlátos determinisztikus programmal.

Feltehetjük, hogy N-nek mindig pontosan két választása van megállásig, a 0. és az 1.

A determinisztikus algoritmus:

- Futtatunk egy k számlálót 1-től fölfelé (ennyi lépésig szimuláljuk a nemdeterminisztikus programot).
- k minden értékére leszimuláljuk az összes k hosszú választási sorozatot (ez 2^k darab):
  - ha van köztük elfogadó, elfogadjuk az inputot; találtunk acceptet
  - ha mind elutasító, elvetjük az inputot; bejártuk az egész fát
  - különben növeljük k-t és folytatjuk a ciklust. k még kisebb, mint $f(n)$, go deeper

(Kimaradt részlet: t utasítás végrehajtása alatt egy RAM program $\mathcal{O}(t^2)$ memóriát használ – ez pl. ismét annak a következménye, hogy a szorzás nem elemi művelet, az összeadás nem növelheti „túlzott ütemben” a változók értékét. Maga az $f(n)$ függvény nem biztos, hogy egyáltalán kiszámítható, ezért nem számoljuk ki előre, hanem iteratívan mélyülő mélységikeresünk.)

$\mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f^2(n))$, tovább

- A szimulálást a tárterület újrafelhasználásával tesszük (feltehetjük, hogy N offline, így az inputot a szimuláció nem rontja el).

  Ha N nem offline, akkor átírjuk: először másolja át munkaterületre „magának” az inputot és aztán azzal dolgozzon. A kimenetet is hasonlóan munkaterületen állíthatjuk elő, majd átmásoljuk. Ettől az időigény nagyságrendben nem romlik el.

- A k. menetben egy k hosszú bitvektorban tartjuk nyilván az aktuális választási sorozatot, ezt növeljük mondjuk binárisan 0-ról indítva.
- Mivel $f(n)$ lépésben N is csak $f^2(n)$ tárat használ (cellánként max $f(n)$ tárat, lépésenként max egy új cellát használ fel), a szimulációhoz elég $f^2(n)$ tár; mivel $k \leq f(n)$, a bitvektornak is elég $f(n)$ tár.

Alapvető összefüggések

$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(k^{f(n)})$

Tehát egy $f(n)$ tárkorlátos N nemdeterminisztikus programot akarunk determinisztikusan szimulálni.

Feltehetjük, hogy

- N offline;
- elfogadás előtt közvetlenül kinullázza munkaregisztereit.

Nevezzük konfigurációnak a RAM program teljes leírását a számítás egy pillanatában, miután az n hosszú $(a_1, \ldots, a_n)$ inputon elindítjuk:

- a programszámláló értéke: k_1 konstans sok lehetőség
- az összes nemnulla munkaregiszter aktuális tartalma $(i, W[i])$ párok listájának formájában: $\mathcal{O}(f(n))$ bit hosszú lista, $2^{\mathcal{O}(f(n))}$-féle lehetőség

Összesen $k_1 \cdot 2^{\mathcal{O}(f(n))} = k^{f(n)}$ konfiguráció.

gyakorlatilag listában tároljuk a (kulcs,érték) párokat a felhasznált regiszterekre

Alapvető összefüggések

$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(k^{f(n)})$, tovább

Az adott inputhoz tartozó konfigurációs gráf: a konfigurációk a csúcsok, C_1-ből akkor vezet él C_2-be, ha N átléphet C_1-ből C_2-be.
A determinisztikus algoritmus: elkészítjük az inputhoz tartozó konfigurációs gráfot és azon lineáris időben megoldunk egy Elérhetőség problémát a kezdőkonfigurációból a (kinullázás miatt egyértelmű) elfogadó konfigurációba.
(A gráfot nem kell előre elkészítenünk és letárolnunk, on-the-fly is számíthatjuk.)

Az eddigiekből $\mathrm{NTIME}(f(n)) \subseteq \mathrm{TIME}(k^{f^2(n)})$ kijön, de az erősebb $\mathrm{NTIME}(f(n)) \subseteq \mathrm{TIME}(k^{f(n)})$ is igaz.

A „nevesített” osztályok sorrendje

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} (\subsetneq \mathbf{R} \subsetneq \mathbf{RE})$$

$\mathbf{L} \subseteq \mathbf{NL}$ (i)

$\mathbf{NL} = \mathrm{NSPACE}(\log n) \subseteq \mathrm{TIME}(k^{\log n}) \subseteq \mathbf{P}$ (iii)

itt használjuk, hogy $k^{\log n} = n^{\log k}$ hatványozás-azonosság, így már látszik, hogy polinom

$\mathbf{P} \subseteq \mathbf{NP}$ (i)

$\mathbf{NP} = \mathrm{NTIME}(n^k) \subseteq \mathrm{SPACE}(n^k) = \mathbf{PSPACE}$ (ii)

$\mathbf{PSPACE} = \mathrm{SPACE}(n^k) \subseteq \mathrm{TIME}(2^{n^k}) = \mathbf{EXP}$ (iii)

A központi kérdés, hogy $\mathbf{P} = \mathbf{NP}$ vagy sem, de a fentiek közül egyiket se tudjuk, hogy egyenlőség-e!

A hierarchia tételek

Megengedett függvények

Az $f : \mathbb{N} \to \mathbb{N}$ monoton növő függvény megengedett, ha létezik olyan program, ami az $1^n \mapsto 1^{f(n)}$ függvényt számítja ki $\mathcal{O}(f(n))$ tárban és $\mathcal{O}(n + f(n))$ időben.

Azaz n darab egyesből $f(n)$ darab egyest készít, közben nem használ indokolatlanul sok tárat, sem időt. A polinomok, exp, log függvények stb. amik „realisztikusan” előjönnek tár/időigényként, mind ilyenek

Időhierarchia tétel

Ha $f(n) > n$ megengedett függvény és $g(n) = o(f(n))$, akkor

$$\mathrm{TIME}(g(n)) \subsetneq \mathrm{TIME}(f(n)).$$

Emiatt pl. $\mathbf{P} \neq \mathbf{EXP}$.

„sokkal több idő több mindenre elég”

Tárhierarchia tétel

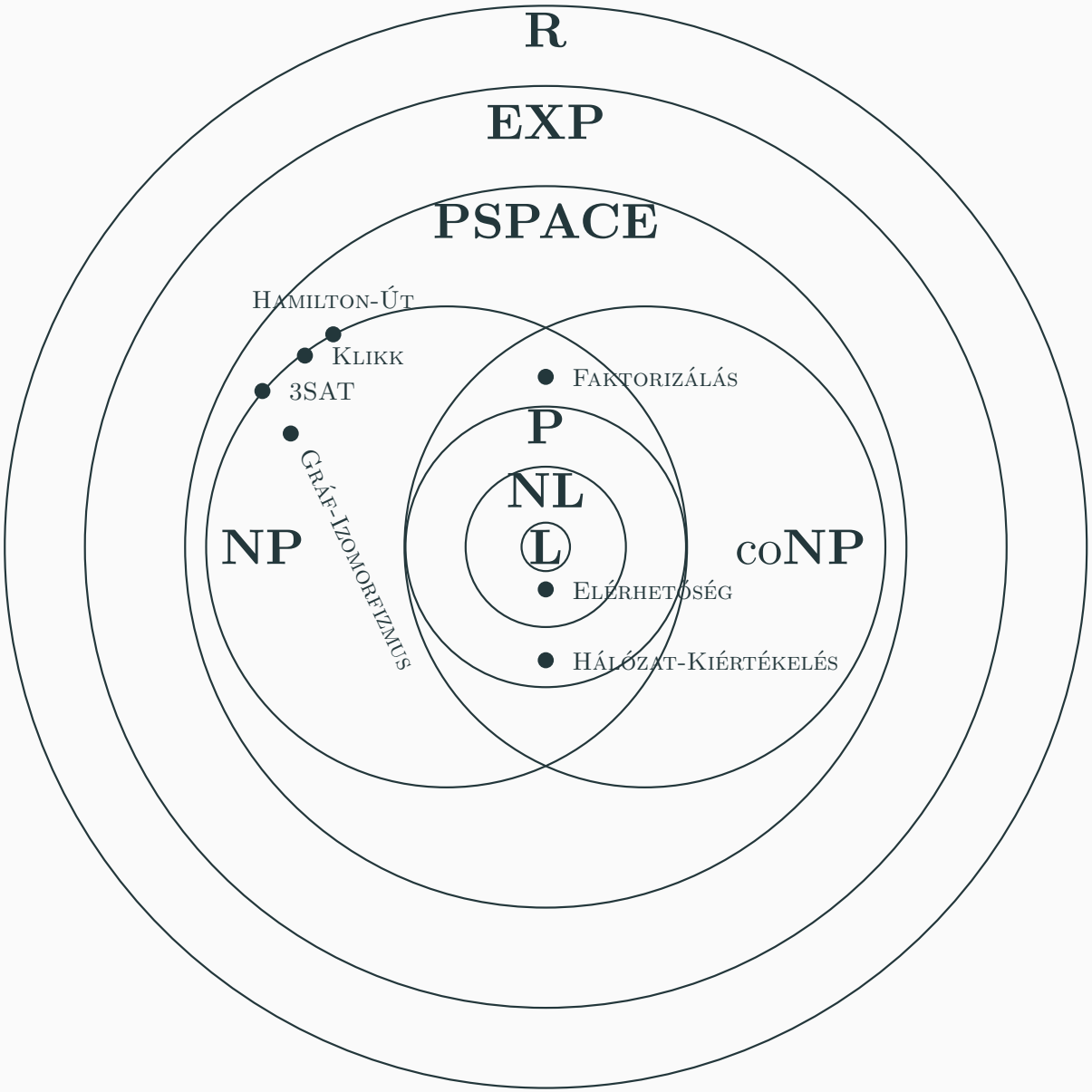
Ha $f(n) > \log n$ megengedett függvény és $g(n) = o(f(n))$, akkor

$$\mathrm{SPACE}(g(n)) \subsetneq \mathrm{SPACE}(f(n)).$$

Emiatt pl. $\mathbf{L} \neq \mathbf{PSPACE}$.

„sokkal több tár több mindenre elég”

Osztálykép



- coNL, NPSPACE, coNPSPACE okkal nincsenek a képen, ezt nemsokára látni fogjuk, hogy miért
- Elérhetőség és Hálózat-Kiértékelés helye pontosodni fog nemsokára
- Gráf-Izomorfizmus és Faktorizálás helyét ennél pontosabban nem tudjuk
- a hierarchia tételek miatt pl. $\mathbf{P} \neq \mathbf{EXP}$ és $\mathbf{L} \neq \mathbf{PSPACE}$
- hamarosan lesznek problémáink az NP és R közti részen is

Savitch tétele

A most következő részben tárhatékonyan próbálunk megoldani problémákat.

Egy algoritmus

Elemezzük az alábbi algoritmust!

Input: irányított gráf, mondjuk $G[1..N][1..N]$ szomszédsági mátrixával

Output: ?

function $\textsc{f}(x, y, d)$ ez egy függvénydeklaráció

  if $d == 0$ **then return** $\Big((x == y) \text{ OR } G[x][y]\Big)$;

  for $z = 1 \dots N$ **do**

    if $\Big(\textsc{f}(x, z, d-1) \text{ AND } \textsc{f}(z, y, d-1)\Big)$ **then return** TRUE

  return FALSE

return $\textsc{f}(1, N, \lceil \log N \rceil)$; ez a main call

Savitch tétele

A függvény

Az $f(x, y, d)$ hívás pontosan akkor ad TRUE-t, ha a G gráfban van x-ből y-ba legfeljebb 2^d hosszú út.

- $d == 0$: legfeljebb egy hosszú út, vagyis $x == y$ vagy van él x-ből y-ba;
- $d > 0$: ha van x-ből y-ba egy legfeljebb 2^d hosszú út, akkor ennek felezőpontjába, z-be van x-ből egy legfeljebb 2^{d-1} hosszú út és z-ből y-ba is, ez akkor teljesül, ha $f(x, z, d-1)$ és $f(z, y, d-1)$ is TRUE-val tér vissza.

A belépési pont

Az $f(1, N, \lceil \log N \rceil)$ hívás tehát akkor ad TRUE-t, ha van legfeljebb $2^{\lceil \log N \rceil} \geq N$ hosszú út 1-ből N-be, vagyis egy a kód egy (determinisztikus) algoritmus az ELÉRHETŐSÉG problémára.

Savitch tétele

Tárigény

- Az f függvény egy példányának tárigénye $\mathcal{O}(\log n)$, hisz csak az x, y, d, z változókat deklarálja, mindegyik $0 \dots N$ közti értéket vesz fel;
- a hívási verem mélysége $\lceil \log N \rceil = \mathcal{O}(\log n)$, legfeljebb ennyi példánya van egyszerre f-nek a memóriában

  mert d értéke mindig csökken eggyel a rekurzív hívásoknál

tehát a teljes tárigény $\mathcal{O}(\log^2 n)$.

Ezzel beláttuk Savitch tételét:

Savitch tétele

$$\textsc{Elérhetőség} \in \mathrm{SPACE}(\log^2 n).$$

Savitch tétele

Következmény

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f^2(n)).$$

Ötlet

Az Elérhetőség problémát a nemdeterminisztikus program konfigurációs gráfján oldjuk meg.

- Egy $O(f(n))$ tárkorlátos (nemdeterminisztikus, offline) programnak egy n méretű inputon elindítva $k^{f(n)}$ konfigurációja lehet.
- A konfigurációs gráf tehát ennyi csúcsú: ezen futtatva a Savitch algoritmust a tárigény:

$$\log^2(k^{f(n)}) = (O(f(n)) \cdot \log k)^2 = O((f(n))^2).$$

Savitch tétele

Következmény

$$\mathbf{PSPACE} = \mathbf{NPSPACE}.$$

ezért nem volt ott az előző osztályképen **NPSPACE** külön

Hiszen polinom négyzete is polinom, így pl. egy n^3 tárigényű nemdeterminisztikus algoritmust $O((n^3)^2) = O(n^6)$ tárigényű determinisztikussal tudunk szimulálni, ami még mindig polinom tár.

Következmény

$$\mathbf{NL} \neq \mathbf{PSPACE}.$$

Hiszen $\mathbf{NL} \subseteq \mathrm{SPACE}(\log^2 n) \subsetneq \mathrm{SPACE}(n)$ Savitch tétele és a tárhierarchia tétel szerint.

Az Immerman-Szelepcsényi tétel

Most alulról felfelé haladva felépítünk egy összetett nemdeterminisztikus algoritmust.

A cél: nemdeterminisztikus algoritmus eredményét szeretnénk komplementálni. Ez nem megy simán ACCEPT/REJECT cserével, mert a „vegyes válasz" továbbra is az marad, így továbbra is elfogadnánk, amit el kéne utasítanunk.

Egy nemdeterminisztikus algoritmus

Input: gráf, $G[1 \dots N][1 \dots N]$ szomszédsági mátrixával.

function $\text{Út}(s, t, d)$ pont mint az első **NL** algoritmusunk, csak megkapja d-t is

```
i := s;
while d ≥ 0 do
    if i == t then return ACCEPT
    j := nd(N);
    if NOT G[i][j] then return REJECT
    i := j;
    d := d − 1;
```

- A függvény visszaadhat ACCEPT-et, ha s-ből t d lépésen belül elérhető.
- Egyébként mindenképp REJECT-et ad vissza.
- Tárigény: $\mathcal{O}(\log n)$.

Az Immerman-Szelepcsényi tétel

Egy nemdeterminisztikus algoritmust hívó algoritmus

Input: gráf, $G[1 \dots N][1 \dots N]$ szomszédsági mátrixával.

function T-IN-SK$(s, t, prev, k)$
  $check := 0$;
  for $u := 1 \dots N$ **do**
    if ÚT$(s, u, k-1)$ **then**
      $check := check + 1$;
      if $G[u][t]$ **then return** ACCEPT
  if $check == prev$ **then return** REJECT
  THROW ERROR;

- Ha $prev$ az s-ből legfeljebb $k-1$ lépésben elérhető csúcsok száma, akkor:
- ha s-ből t legfeljebb k lépésben elérhető, akkor az eredmény ACCEPT vagy ERROR, lehet ACCEPT;
- ha nem, akkor az eredmény REJECT vagy ERROR, lehet REJECT.
- Logaritmikus tárban.

először érdemes a check változó nélkül megérteni a kódot, mintha ott se lenne
a CHECK == PREV akkor lesz igaz, ha az ÚT hívás mindenkit tényleg megtalált, aki elérhető $k-1$ lépésben
de ha minden ilyet megtalált és egyikből se lehet t-be jutni, akkor tényleg nem lehet k lépésben t-be jutni

Az Immerman-Szelepcsényi tétel

Ez lesz az algoritmus, amit szerettünk volna:

Egy nd algoritmust hívó algoritmust hívó algoritmus

Input: gráf, $G[1 \dots N][1 \dots N]$ szomszédsági mátrixával és s csúcsa.

$prev := 1;$
for $k := 1 \dots N$ **do**
  $current := 0;$
  for $t := 1 \dots N$ **do**
    if T-IN-SK$(s, t, prev, k)$ **then**
      $current := current + 1;$
  $prev := current;$
return $prev;$

- Az algoritmus visszaadja az s-ből elérhető csúcsok számát, vagy ERROR-t dob; van, mikor a számot adja vissza.
  ERROR-t még a T-IN-SK függvény dobhatott, ez azt tovább dobja mint egy kivételt
- Mindezt logaritmikus tárban.

Az Immerman-Szelepcsényi tétel

Beláttuk az Immerman-Szelepcsényi tételt:

Az Immerman-Szelepcsényi tétel

Van olyan nemdeterminisztikus algoritmus, mely logaritmikus tárban kiszámítja az input gráf megadott csúcsából elérhető csúcsok számát.

Nemdeterminisztikus függvénykiszámítás jelentése: minden szálon vagy a helyes eredményt adja vissza, vagy REJECT-et; továbbá van olyan szál, mikor a helyes eredményt.

a FAKTORIZÁLÁS probléma megoldásánál is valami ilyesmit csináltunk: nemdeterminisztikusan generáltuk a prímtényezős felbontást, és ha nem sikerült, rejecteltünk.

Az Immerman-Szelepcsényi tétel

A tétel megint akkor lesz igazán hasznos, ha a konfigurációs gráfra alkalmazzuk.

Következmény

$$\mathrm{NSPACE}(f(n)) = \mathrm{coNSPACE}(f(n)).$$

Ötlet

A konfigurációs gráfban először nd kiszámítjuk az elérhető konfigurációk számát, aztán ciklusban mindet nemdeterminisztikusan megpróbáljuk elérni, számoljuk, hogy mennyit sikerült. Ha annyit értünk el, amennyi csak elérhető és egyik sem elfogadó, akkor Accept, különben Reject.

Következmény

$$\mathbf{NL} = \mathrm{co}\mathbf{NL}.$$

na $\mathrm{co}\mathbf{NL}$ meg ezért nem szerepelt az osztályképen

A logaritmikus tárban való visszavezetés

A polinomidejű visszavezetésre nézve $\mathbf{P}$ minden nemtriviális eleme „$\mathbf{P}$-teljes”.

mert ami erőforrást az input konvertálására használhatnánk, az már arra is elég, hogy megoldjuk a problémát

Bevezetünk egy (formailag) „gyengébb” visszavezetést, melyet a $\mathbf{P}$ osztályon belül alkalmazunk problémák egymáshoz viszonyított nehézségének definiálására.

mert hát az azért mégsincs rendben, hogy őket mind egyforma nehéznek vesszük

Az f függvény az A eldöntési problémának a B eldöntési problémára való logaritmikus tárigényű („logtáras”) visszavezetése, ha

- f kiszámítható logaritmikus tárban és
- tetszőleges I inputra $A(I) = B(f(I))$.

logtárban kiszámítható, választartó inputkonverzió

Ha létezik A-nak B-re való logaritmikus tárigényű visszavezetése, annak jele $\mathrm{A} \leq_{\mathcal{L}} \mathrm{B}$: A logtárban visszavezethető B-re.

A logtáras visszavezetés

Logaritmikus tárigényű (tehát mindenképp offline) algoritmusnak legfeljebb $2^{\mathcal{O}(\log n)}$, azaz polinom sok konfigurációja lehet adott input esetén; mivel a visszavezetés nem eshet végtelen ciklusba, így polinomidőben meg is kell álljon.

Vagyis ha $\mathrm{A} \leq_{\mathcal{L}} \mathrm{B}$, akkor $\mathrm{A} \leq_{\mathcal{P}} \mathrm{B}$ is.

Nem ismert, hogy $\leq_{\mathcal{L}}$ és $\leq_{\mathcal{P}}$ egybeesnek-e...
...ha igen, akkor $\mathbf{L} = \mathbf{P}$, hiszen akkor

- $\mathbf{P}$ minden eleme visszavezethető $\mathbf{P}$ minden nemtriviális elemére polinomidőben,
- akkor logtárban is a feltevés szerint,
- van $\mathbf{L}$-ben is nemtriviális probléma,
- így ez az $\mathbf{L}$-beli probléma $\mathbf{P}$-teljes lesz a logtáras visszavezetésre,
- ami miatt $\mathbf{L} = \mathbf{P}$ igaz lesz

Nehézség, teljesség

Legyen $\mathcal{C}$ problémák egy osztálya. Az A probléma $\mathcal{C}$-nehéz a logtáras visszavezetésre nézve, ha minden $\mathcal{C}$-beli probléma logtárban visszavezethető A-ra.
Ha még $\mathrm{A} \in \mathcal{C}$ is, akkor A egy $\mathcal{C}$-teljes probléma a logtáras visszavezetésre nézve.

Nem ismert, hogy a logtáras visszavezetésre nézve $\mathbf{NP}$-teljes problémák ugyanazok-e, mint a polinomidejű visszavezetésre nézve $\mathbf{NP}$-teljesek.
A Sat egy $\mathbf{NP}$-teljes probléma a logtáras visszavezetésre nézve is; az összes hatékony visszavezetés, melyet eddig láttunk, egyben logtáras is volt, implementálhatóak voltak konstans sok pointer / polinomig számláló intek használatával.

A logtáras visszavezetés tranzitivitása

A logtárban való visszavezethetőség tranzitív: ha f az A-nak B-re, g pedig a B-nek C-re való logtáras visszavezetése, akkor az $f \circ g$ összetett függvény az A-nak C-re való logtáras visszavezetése.

Ez miért is nem nyilvánvaló?

A gond

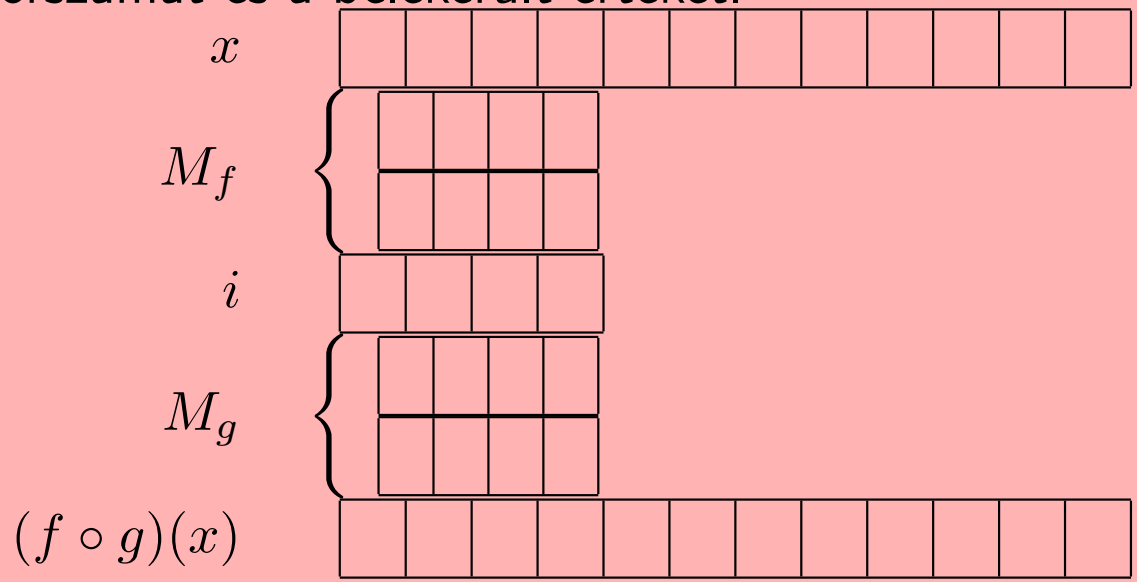
Legyen M_f az f-et, M_g pedig a g-t kiszámító program.
Az nyilván igaz, hogy $I \in A \Leftrightarrow f(I) \in B \Leftrightarrow g(f(I)) \in C$.
Csakhogy ha $M_f(I)$-t munkaregiszterekbe írjuk, sokkal hosszabb lehet, mint $\log(|I|)$!

azaz, az első visszavezetés outputját nem tudjuk eltárolni $O(\log n)$ tárban.

A visszavezetés tranzitivitása

A trükk

Nem tároljuk el M_f teljes outputját, csak a legutolsóként írt output regiszter sorszámát és a belekerült értéket.



M_g-nek szüksége van egy nagyobb sorszámú regiszter tartalmára: folytatjuk M_f szimulálását.

M_g-nek szüksége van egy kisebb sorszámú regiszter tartalmára: elölről kezdjük M_f szimulálását.

Itt pl. fontos, hogy az offline programok az outputra stream módban írnak.

P-teljesség, NL-teljesség

Korábban láttuk, hogy $\mathbf{P}$ (így $\mathbf{NL}$) bármely két nemtriviális problémája hatékonyan visszavezethető egymásra, így a polinomidejű visszavezetésre nézve ezen osztályokon belül a „teljesség” fogalma értelmetlenné válik.

Azt mondjuk, hogy egy probléma $\mathbf{NL}$/$\mathbf{P}$-teljes/nehéz, ha a logtáras visszavezetésre nézve az.

A Hálózat-Kiértékelés probléma $\mathbf{P}$-teljes.

jegyzetben ez is benne, ha kíváncsi vagy rá

Mivel $\mathbf{NL} \subseteq \mathbf{P}$ és Elérhetőség $\in \mathbf{NL}$, ezért ez azt is mondja, hogy az Elérhetőség könnyebb, mint a Hálózat-Kiértékelés (ha $\mathbf{NL} \neq \mathbf{P}$).

Az Elérhetőség probléma NL-teljessége

Az Elérhetőség probléma **NL**-teljes.

Ötlet

Azt már láttuk, hogy Elérhetőség $\in \mathbf{NL}$.

Az **NL**-nehézséghez mint korábban, az elérhetőségi módszert alkalmazzuk: egy nemdeterminisztikus, $\mathcal{O}(\log n)$ tárkorlátos gép konfigurációs gráfjában megoldva az elérhetőségi problémát a kezdőkonfigurációtól az elfogadóig, készen vagyunk.

(hiszen egy ilyennek $k^{O(\log n)}$ csúcsa van, annak a logaritmusa $O(\log n)$ lesz.

Az Elérhetetlenség probléma **NL**-teljes.

Hiszen az Immerman-Szelepcsényi tétel szerint $\mathbf{NL} = \mathrm{co}\mathbf{NL}$.

Logaritmikus tár

A 2SAT probléma **NL**-teljes.

Ötlet

Azt már tudjuk, hogy $2\text{SAT} \in \mathbf{NL}$: vegyük a 2SATot megoldó algoritmust.

- A $G(\varphi)$ gráf elkészíthető logaritmikus tárban.
- Olyan x változót, melyből $\neg x$ elérhető és viszont, nemdeterminisztikusan logtárban tudunk keresni, ha van.
- A két algoritmust a tranzitivitásban megismert módszerrel összekomponálva egy nemdet logtáras algoritmust kapunk.

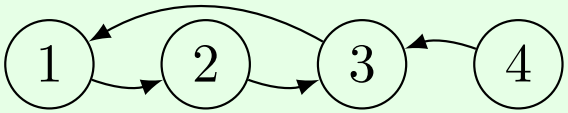
Az **NL**-nehézséghez az $\text{ELÉRHETETLENSÉG} \leq_{\mathcal{L}} 2\text{SAT}$ visszavezetést mutatjuk meg.

Logaritmikus tár

Tekintsük az ELÉRHETETLENSÉG egy példányát.
Minden csúcshoz rendeljünk egy x változót.
A 2SAT azon φ példányát készítjük el, mely az összes $\neg x \vee y$ klózokból áll, ahol (x, y) él, továbbá az s és $\neg t$ egységklózokból, ahol s a kezdőcsúcs és t a cél.
Így φ kielégíthető $\Leftrightarrow$ s-ből nem érhető el t.

Példa

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:

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

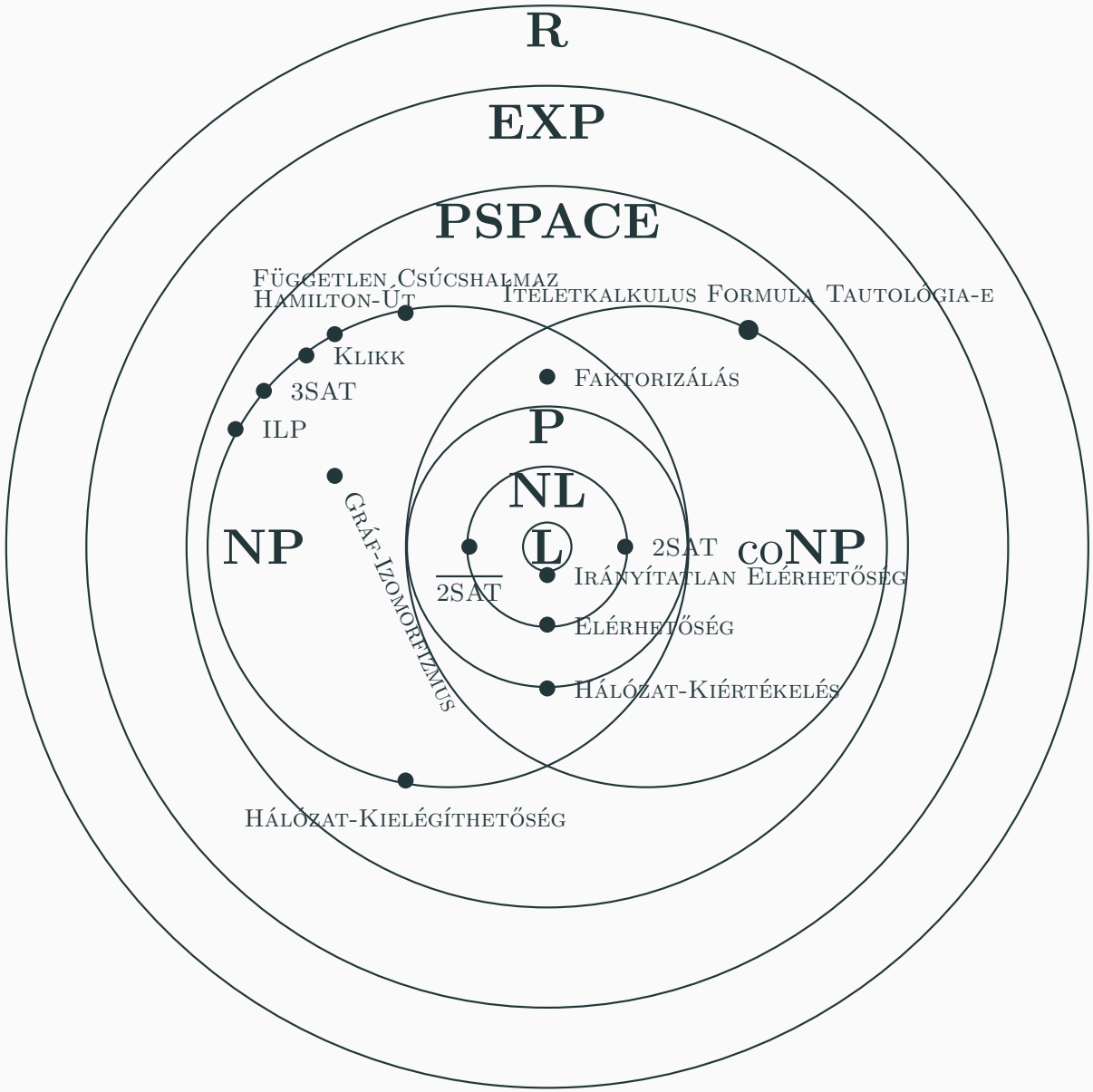
Ami kielégíthető az $x_1 = x_2 = x_3 = 1$, $x_4 = 0$ értékadással.
(Ha s-ből t nem érhető el, akkor egy kielégítő értékadás: az s-ből elérhető csúcsokat állítjuk igazra – azokat muszáj is –, a többit hamisra – t-t muszáj is.)

Emiatt a 2SAT komplementere is **NL**-teljes.

Osztálykép

RE

- Hilbert 10. problémája
- Post Megfelelkezési Problémája
- Megállás
- Elsőrendű Formula Tautológia-E



A polinomiális tár

QSAT **"quantified SAT", kvantoros szat**

Adott: egy $\exists x_1 \forall x_2 \ldots Q_m x_m \varphi$ alakú kvantifikált ítéletlogikai formula úgy, hogy a magja, φ konjunktív normálalakú kvantormentes formula, melyben legfeljebb csak az $x_1, \ldots, x_m$ változók fordulnak elő.

Kérdés: Igaz-e az adott formula?

- A kvantorokról feltehetjük, hogy alternálnak: $\exists$ és $\forall$ felváltva jönnek.

  ha pl két $\exists$ jönne egymás után, betehetünk közéjük egy $\forall y$ új változót, a formula értéke nem változik

- Azt is feltehetjük, hogy az első kvantor $\exists$.

  hasonlóan, ha $\forall x_1$-gyel kezdődik a formula, betehetünk elé egy $\exists x_0$ új változót

- Fontos: itt az x_i változók logikai változók, értékük igaz/hamis lehet! (nem pedig elsőrendű logika)

QSAT példa

Példa

$$\exists x\ \forall y(\ (x \vee y)\ \wedge\ (\neg x \vee \neg y)\ )$$

Ez a példány egy „nem” példány, mert a formula hamis: $x = 0$ esetén $y = 0$, $x = 1$ esetén pedig $y = 1$ választása mellett hamis lesz a formula magja.

Példa

$$\exists x\ \forall y\ \exists z\ \forall w\big((\neg x \vee z \vee w)\ \wedge\ (y \vee \neg z)\ \wedge\ (x \vee \neg y \vee z)\big)$$

?

Hogyan értékelünk ki egy ilyen formulát?

A hosszabb példa

Mivel igazságértékekről van szó, a „létezik”-kel kvantált x_i változóknál pl. egy lehetőség kipróbálni az $x_i = 0$ és az $x_i = 1$ értékadásokat is, behelyettesíteni a konstansokat, egyszerűsíteni a formulát és hívni rekurzívan, ha valamelyik a kettő közül igaz lesz, akkor a formula értéke igaz, különben hamis.
A „bármely” kvantorral ellátott változóknál pedig akkor lesz igaz, ha mindkettő igaz lesz.

$$\exists x\ \forall y\ \exists z\ \forall w\Big((\neg x \vee z \vee w)\ \wedge\ (y \vee \neg z)\ \wedge\ (x \vee \neg y \vee z)\Big)$$

$x = 0$-ra az egyszerűsített formula $\forall y\ \exists z\ \forall w\Big((y \vee \neg z)\ \wedge\ (\neg y \vee z)\Big)$.

$x = 1$-re pedig $\forall y\ \exists z\ \forall w\Big((z \vee w)\ \wedge\ (y \vee \neg z)\Big)$.
Ha a két formula közül bármelyik igaz, akkor az eredeti is az.
Az első formula $\forall y\ \exists z\ \forall w\Big((y \vee \neg z)\ \wedge\ (\neg y \vee z)\Big)$.

Ez $y = 0$ választással $\exists z\ \forall w\Big(\neg z\Big)$ lesz (ez a formula igaz: $z = 0$ jó lesz),

$y = 1$ választással pedig $\exists z \forall w\Big(z\Big)$, ami szintén igaz, $z = 1$ választással.

Tehát a $\forall y\ \exists z\ \forall w\Big((y \vee \neg z)\ \wedge\ (\neg y \vee z)\Big)$ formula mindkét lehetséges y választás mellett igaz, így ő is igaz; ezért az eredeti formulánk is igaz, hiszen az egzisztenciálisan kvantált x-nek van legalább egy választása (az $x = 0$), ami igazzá teszi a formulát.

QSAT $\in$ PSPACE

A QSAT probléma **PSPACE**-ben van.

Algoritmus **hasonlít a DPLL-re, csak nem mindig vagyolunk a csúcspontokban**

Input: $F = Q_1 x_1 \dots Q_m x_m \varphi$ alakú zárt formula.

Output: a formula értéke.

function $\textsc{QSat}(F)$
  if $F == \exists x F'$ **then**
    return $\textsc{QSat}(F'[x/0])$ or $\textsc{QSat}(F'[x/1])$
  else if $F == \forall x F'$ **then**
    return $\textsc{QSat}(F'[x/0])$ and $\textsc{QSat}(F'[x/1])$
  else
    return $\textsc{eval}(F)$

Ez az algoritmus $\mathcal{O}(n^2)$ tárban kiszámítja a formula értékét (ahol $\textsc{Eval}$ a változómentes formulát polinom időben és lineáris tárban kiértékelő függvény).

QSAT

A probléma akkor is **PSPACE**-ben van (az algoritmus könnyen módosítható), ha

- φ tetszőleges kvantormentes Boole-formula.

  a végén bármilyen formulát ki tudunk értékelni lineáris időben és tárban, nem baj ha nem CNF

- a kvantorok nem feltétlenül váltakoznak.

  az algoritmus nem is foglalkozott vele, hogy váltakoznak-e

- φ-ben az $x_1, \ldots, x_m$ változókon kívül egyéb változók is előfordulnak, és azt kérdezzük, kielégíthető-e az egész formula.

  ehhez csak $\exists$ kvantorral kell kötni a formula legelején a szabad változókat

- a formula nem feltétlenül prenex alakú.

  több if kell annak függvényében, hogy a külső jel milyen konnektíva

- azt kérdezzük, hogy a formula hamis-e.

  a végső választ kell csak negálni, ez egy determinisztikus algoritmus

QSAT

Amiért a kötöttebb alakú inputtal definiáljuk a QSATot, az azért van, mert már ez a speciális forma is pont olyan nehéz, mint az általános alak:

A QSAT probléma **PSPACE**-teljes.

és innentől más **PSPACE**-nehézséghez elég a QSAT spéci formuláit konvertálni az aktuális probléma inputjába, ezért jó a speciális alak, ugyanúgy, ahogy a SAT esetében a 3SAT volt

Ennek a bizonyításához megint az elérhetőségi módszert fogjuk használni, kicsit másképp:

- veszünk egy polinom tárkorlátos programot
- annak pedig a konfigurációs gráfját egy n méretű I inputon
- és felírunk egy formulát, ami pont akkor lesz igaz, ha ebben a gráfban a kezdőkonfigurációból van út az elfgadó konfigurációba
- a nehézséget az fogja okozni, hogy ez az egész konstrukció beleférjen a polinomidőbe

QSAT PSPACE-teljes

Ötlet

- Egy tetszőleges n^k tárkorlátos M programhoz és x inputjához konstruálunk egy φ formulát, mely pontosan akkor igaz, ha M elfogadja x-et.
- Egy konfigurációt n^k darab bináris változóval kódolunk.

  ennyi elég hozzá, mert erről szól a tárigény
- Logikai változók n^k hosszú vektorát az $\boldsymbol{x}$, $\boldsymbol{y}$ stb. fogják jelölni, pl. $\boldsymbol{x} = (x_1, x_2, \ldots, x_{n^k})$. Felfoghatjuk úgy, hogy $\boldsymbol{x}$ értéke egy konfiguráció lesz.
- Minden $t \geq 0$ egészre készítünk egy $\varphi_t(\boldsymbol{x}, \boldsymbol{y})$ formulát.

  ez a jelölés azt takarta, hogy φ_t-ben a szabad változók $\boldsymbol{x}$ és $\boldsymbol{y}$ lesznek, amit felfoghatunk úgy is, hogy $2 \cdot n^k$ darab szabad logikai változója lesz, vagy úgy is, hogy két szabad konfiguráció-változója lesz.

QSAT PSPACE-teljes

Ötlet

- A $\varphi_t(\boldsymbol{x}, \boldsymbol{y})$ formula akkor lesz igaz, ha az $\boldsymbol{x}$ konfigurációból el lehet jutni az $\boldsymbol{y}$ konfigurációba legfeljebb 2^t lépésben.

  Savitch-tétel PTSD

- Ha $\boldsymbol{x}_0$ jelöli a kezdő- és $\boldsymbol{y}_{\text{ACCEPT}}$ az elfogadó konfigurációt, akkor ezek szerint a formula, melyet a visszavezetéssel le akarunk gyártani, a $\varphi_{n^k}(\boldsymbol{x}_0, \boldsymbol{y}_{\text{ACCEPT}})$.

  hiszen n^k hosszú bitvektorból csak 2^{n^k} van, így a gráf ennyi csúcsú, tehát ennyi lépésben eljutunk a kezdőcsúcsból az elfogadóba, ha egyáltalán el lehet.

- A $\varphi_0(\boldsymbol{x}, \boldsymbol{y})$ formula pl. akkor kell igaz legyen, ha az $\boldsymbol{x}$ konfigurációból vagy egy lépésben eljutunk az $\boldsymbol{y}$ konfigurációba, vagy $\boldsymbol{x} = \boldsymbol{y}$.
  Ennek megadásától itt most eltekintünk, de felírható egy polinom hosszú kvantifikált formulával.

  mert $t = 0$-nál a lépéskorlát $2^0 = 1$

  jó hosszú vagyolás, annak függvényében, hogy hanyadik utasítást hajtjuk végre $\boldsymbol{x}$-ben, mindre felírva, hogy melyik regiszter (max egy) tartalma hogyan kéne változzon

QSAT PSPACE-teljes

$\varphi_{t+1}(\boldsymbol{x}, \boldsymbol{y})$

A Savitch-tételben látottal analóg módon akkor jutunk el $\boldsymbol{x}$-ből $\boldsymbol{y}$-ba max 2^{t+1} lépésben, ha van olyan $\boldsymbol{z}$ konfiguráció, ami „félúton" van, mindkettőtől legfeljebb 2^t lépésre, amit épp φ_t-vel írtunk fel, így a kézenfekvőnek tűnő megoldás:

$$\exists \boldsymbol{z}\big(\varphi_t(\boldsymbol{x}, \boldsymbol{z}) \wedge \varphi_t(\boldsymbol{z}, \boldsymbol{y})\big),$$

de ez túl hosszú!

Ez így egy exponenciális hosszú formula, hiszen φ_{t+1}-hez felírjuk φ_t-t lényegében kétszer, így t minden növelésekor duplázódik a hossza $\Rightarrow$ a nekünk kellő $t = n^k$. formula hossza már 2^{n^k} lenne, ez nem fér bele a visszavezetés időkorlátjába, hogy legyártsuk.

QSAT PSPACE-teljes

$\varphi_{t+1}(\boldsymbol{x}, \boldsymbol{y})$

Ehelyett:

$$\exists \boldsymbol{z} \forall \boldsymbol{x}' \forall \boldsymbol{y}' \Big(\big((\boldsymbol{x}' = \boldsymbol{x} \wedge \boldsymbol{y}' = \boldsymbol{z}) \vee (\boldsymbol{x}' = \boldsymbol{z} \wedge \boldsymbol{y}' = \boldsymbol{y}) \big) \rightarrow \varphi_t(\boldsymbol{x}', \boldsymbol{y}') \Big).$$

ahol $\boldsymbol{a} = \boldsymbol{b}$ az $(a_1 \leftrightarrow b_1) \wedge \ldots \wedge (a_{n^k} \leftrightarrow b_{n^k})$ formula rövidítése.

Hiszen ez a formula mit mond: van olyan $\boldsymbol{z}$ konfiguráció (ez továbbra is a félúton lévő lesz), hogy bárhogy is választunk két konfigurációt, $\boldsymbol{x}'$-t és $\boldsymbol{y}'$-t,

- ha $\boldsymbol{x}' = \boldsymbol{x}$ és $\boldsymbol{y}' = \boldsymbol{z}$, akkor $\varphi_t(\boldsymbol{x}', \boldsymbol{y}')$ igaz – azaz: $\varphi_t(\boldsymbol{x}, \boldsymbol{z})$ igaz;
- ha pedig $\boldsymbol{x}' = \boldsymbol{z}$ és $\boldsymbol{y}' = \boldsymbol{y}$, akkor is $\varphi_t(\boldsymbol{x}', \boldsymbol{y}')$ igaz – azaz: $\varphi_t(\boldsymbol{z}, \boldsymbol{y})$ igaz.

Ez pont ugyanazt állítja, mint az előző fólián a formula! Viszont nem írjuk le benne kétszer φ_t-t, hanem van $3 \cdot n^k$ kvantálás az elején (konfigurációnként n^k), aztán az implikáció bal oldalán egy kb. $8 \cdot n^k$ hosszú feltétel (az egyenlőségek), és a jobb oldalán φ_t, de csak egyszer.

Tehát φ_{t+1} csak $11 \cdot n^k$-val hosszabb, mint φ_t és ezért φ_{n^k} csak $n^k \cdot 11 \cdot n^k = 11 \cdot n^{2k}$-val lesz hosszabb, mint az alapból is polinom hosszú φ_0!

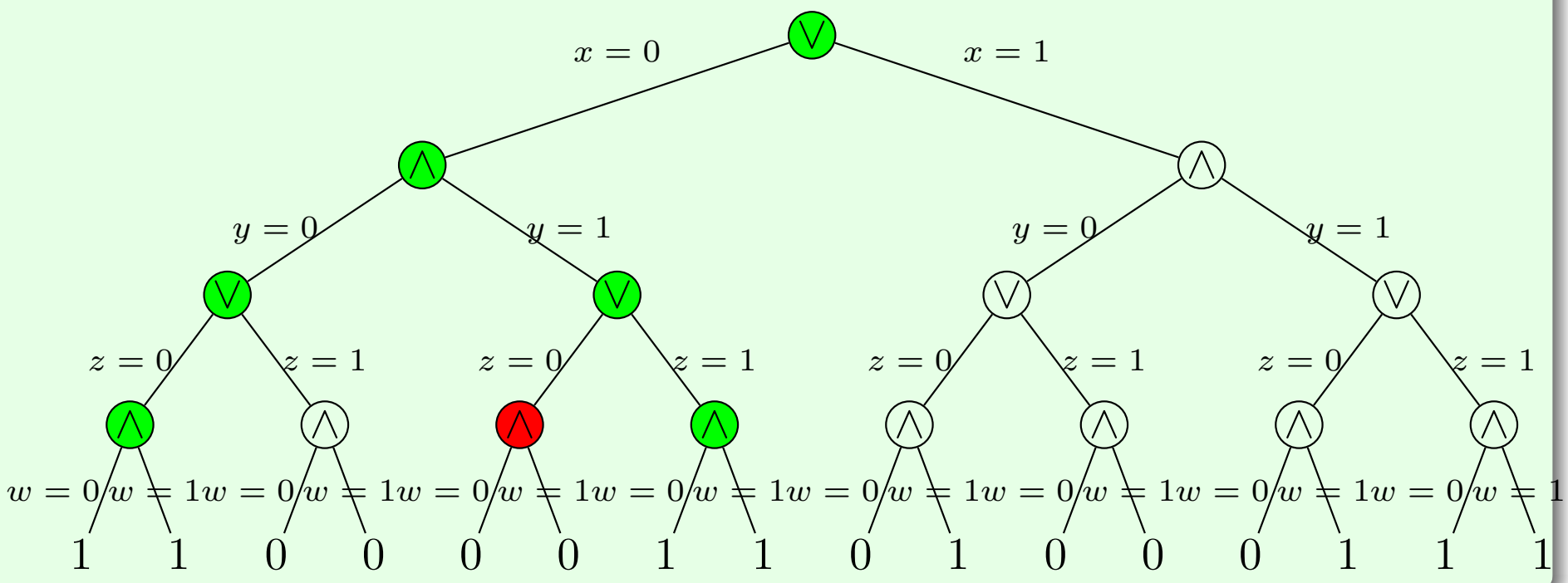
Ez a konverzió már megvalósítható polinomidőben.

A formula magja CNF-re is könnyen hozható pl. a Hálózat-Kielégíthetőség $\leq$ Sat visszavezetés módszerével.

QSAT faként

A QSAT-ot kiértékelő algoritmus rekurzív hívásait faként is tudjuk reprezentálni:

$$\exists x\ \forall y\ \exists z\ \forall w\big((\neg x \vee z \vee w)\ \wedge\ (y \vee \neg z)\ \wedge\ (x \vee \neg y \vee z)\big)$$



a példa kiértékelésekor csak a színes (zöld igaz lett, piros hamis) csúcsokat értékeltük ki, a többit nem kellett

QSAT faként

Az előző fólia fája egy, mestintből ismert minimax játékfa volt:

- két játékos játszik egymás ellen
- az egyik a $\vee$ csúcsok, azaz az $\exists$ változók értékét mondja meg,
- a másik a $\wedge$ csúcsok, azaz az $\forall$ változók értékét mondja meg,
- olyan sorrendben adják meg a változóik értékét, ahogy azok a formulában deklarálva voltak,
- a levelekben a játék értéke attól függően 0 vagy 1, hogy a legyártott értékadás mellett a formula magja hamis vagy igaz,
- mivel a biteken a vagyolás a maximumot, az éselés a minimumot adja vissza, így ez pontosan egy minimax játék
- az $\exists$ játékos célja (aki az első) igazzá tenni a formulát, a $\forall$ játékos célja pedig hamissá tenni.

A QSAT-ot ezért felfoghatjuk mint kétszemélyes játékot, és az „igaz-e a formula”-nak megfelelő kérdés azzá válik, hogy „igaz-e, hogy az első játékosnak van nyerő stratégiája?”

Egy másik kétszemélyes játék, óvodából ismerjük

Földrajzi Játék

Input: $G = (V, E)$ irányított gráf, az egyik csúcsa kezdőcsúcsnak kijelölve.
Output: Az I. játékos nyer-e az alábbi játékban?

- Az I. játékos kezd, rátesz egy bábut a kezdőcsúcsra.
- Minden lépésben minden játékosnak egy olyan csúcsba kell áttolnia a bábut, melyen még nem jártak és melyre egy élen keresztül át tud lépni.
- Egy játékos veszít, ha nem tud lépni.

Ez a játék annak az ismert játéknak a matematikai általánosítása, mikor az egyik óvodás megnevezi Szegedet, majd rendre egy olyan még nem szerepelt várost kell mondani, melynek kezdőbetűje az előzőleg megnevezett város utolsó betűje (pl. Debrecen, majd jöhet Nagybajom, stb)

Földrajzi Játék

A Földrajzi Játék probléma **PSPACE**-ben van.

Ötlet

- A lépések számára van egy polinom felső korlát. most pl. a G csúcsainak száma
- Létezik olyan polinom tárigényű algoritmus, mely adott állásra legyártja az állás rákövetkezőit, vagy ha ilyen nincs, kiszámítja, hogy melyik játékos nyert.

Minden ilyen kétszemélyes játék **PSPACE**-ben van:

- Polinom tárban elkészítjük az adott játék fáját.
- Mélységgel arányos tárral kiértékeljük a játékfát.

Hasonlóan a QSAT-nál látott (gyorsított) rekurzív kiértékelésre, az egész fát nem gyártjuk le előre, hanem on-demand számítjuk ki a csúcsait és töröljük a memóriából, amiket már elhagyunk, ezért lehet mélységgel arányos tárban kiszámítani.

Földrajzi Játék

A Földrajzi Játék probléma **PSPACE**-teljes.

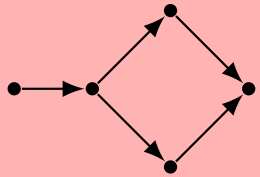
Ötlet

Visszavezetjük rá a QSAT-ot. Tehát: egy $\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_m \varphi$ alakú formulából kell készítenünk egy gráfot egy kezdőcsúccsal úgy, hogy

- ha a formula igaz, vagyis ha a $\exists$ játékosnak van nyerő stratégiája a QSAT játékban, akkor a készített Földrajzi Játék példányban az első játékosnak legyen;
- ha pedig a $\forall$ játékosnak van nyerő stratégiája, akkor a második nyerje a földrajzit.

A konstrukció

- Minden változóhoz elkészítjük az alábbi gadgetet és ezeket láncba kötjük:

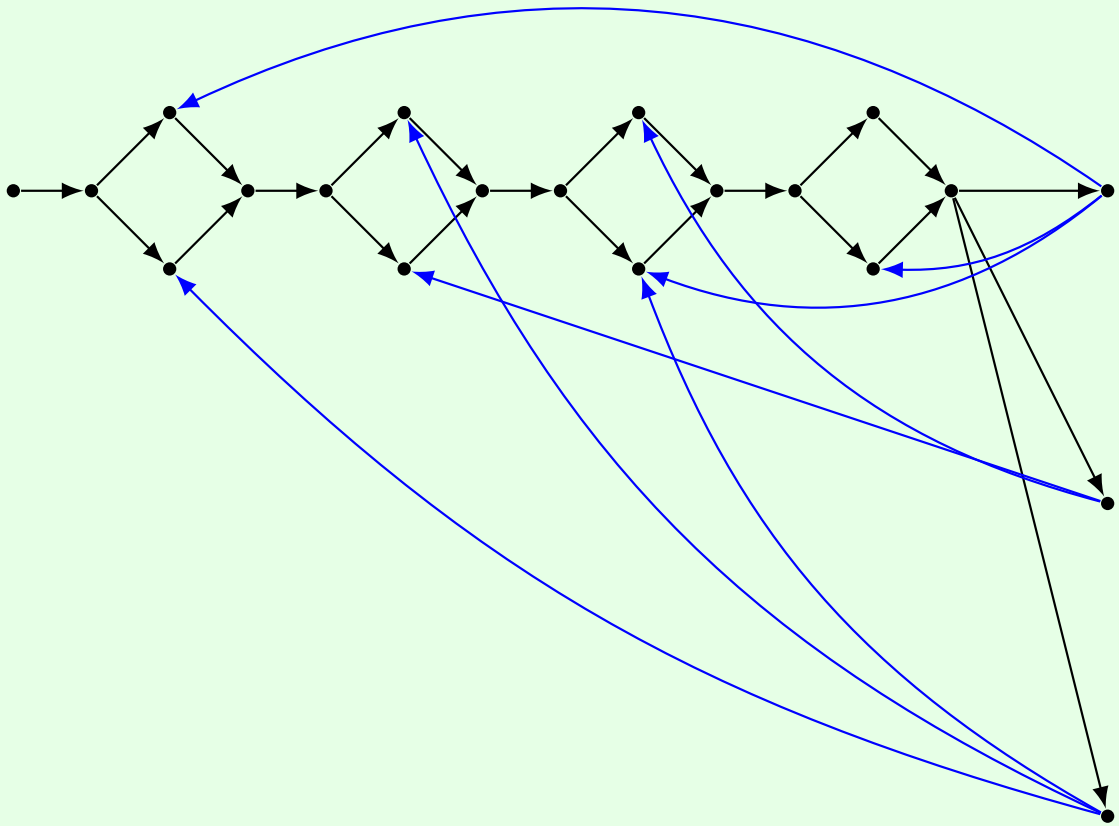


- Amelyik játékos az elágazáshoz ér, az választhat, hogy le vagy fel húzza a bábut; ez megfelel a kérdéses változó 0-ra vagy 1-re állításának.
- Ezt követően a másik játékosnak nincs más választása, mint behúzni a bábut a gadget jobb szélére, mely a következő változó-gadget (ha van) bal oldala lesz; ott az előbb választó játékos van arra kényszerítve, hogy jobbra húzzon és most a másik játékos választ oldalt, azaz értéket.
- Felveszünk továbbá klózonként még egy új csúcsot és az utolsó gadget jobb oldalából éleket húzunk ezekbe.
- Továbbá, a klóz-csúcsokból visszafelé húzunk éleket az értékválasztó gadgetekbe: ha x szerepel benne, akkor az x gadget alsó csúcsába, ha $\neg x$, akkor a felsőbe.

Földrajzi Játék

Nézzük meg ezt a szokásos példánkon.

$$\exists x\ \forall y\ \exists z\ \forall w\big((\neg x \vee z \vee w)\ \wedge\ (y \vee \neg z)\ \wedge\ (x \vee \neg y \vee z)\big)$$



az élek nem színesek, csak átláthatóság végett kékek a klózokból visszaindulók

Földrajzi Játék

- A játék során a játékosok először végighaladnak az értékválasztó gadgeteken, ugyanúgy váltakozva választva fel- illetve lefelé irányt, mint ahogy az eredeti QSAT játékban felváltva adnának értéket a változóknak
- Feltevés szerint páros sok kvantor van (ha nem, beszúrunk egyet) és $\forall$ az utolsó, így utoljára $\forall$ választ
- Ezután $\exists$ behúzza az utolsó gadget jobb szélére a bábut
- Most $\forall$ választ egy klózt. Ha a QSAT játékben volt nyerő stratégiája és aszerint játszott itt is, akkor most a formula hamis, tehát az egyik klóz hamis. Ekkor ide húzza a bábut.
- Ha $\forall$ egy hamis klózba húzza a bábut, akkor $\exists$ vesztett: a klózból kifelé haladó élek összes végpontján jártunk már (pl. ha $\neg x$ szerepel a klózban, és hamis lett, akkor $x = 1$, azaz felfelé haladt a bábu az x gadgetben, és a klózból is föntre vezet az él).
- Ha viszont $\forall$ egy igaz értékű klózba húzza a bábut, akkor $\exists$ nyer: ez a klóz tartalmaz egy igaz literált, az annak megfelelő élen $\exists$ vissza tud szökni a gadgetba, de onnan $\forall$ már nem tudja tovább húzni, mert a gadget végpontjában már biztosan jártak.

Tehát: a QSAT játékban pontosan akkor van nyerő stratégiája az $\exists$ játékosnak, ha ebben a generált Földrajzi Játék példányban az első játékosnak, így ez egy korrekt visszavezetés és Földrajzi Játék tényleg **PSPACE**-teljes.

További PSPACE-teljes problémák

Helyben Elfogadás

Adott: M determinisztikus program és x bemenő szó.

Kérdés: Elfogadja-e M az x-et legfeljebb $|x|$ tárban?

Reguláris Kifejezések Ekvivalenciája

Adott: Két reguláris kifejezés.

Kérdés: Ugyanazt a nyelvet jelölik-e?

Véges Automaták Ekvivalenciája

Adott: M_1 és M_2 véges nemdeterminisztikus automaták.

Kérdés: M_1 és M_2 ekvivalensek-e?

A fenti három probléma mind **PSPACE**-teljes.

PSPACE és a kétszemélyes játékok

Láttuk, hogy két **PSPACE**-teljes probléma, a QSAT is és a Földrajzi Játék is felfogható volt mint kétszemélyes játék. Ez nem véletlen:

Egy probléma pontosan akkor van **PSPACE**-ben, ha definiálható hozzá egy kompetitív kétszemélyes játék az alábbi tulajdonságokkal:

- Mindkét játékos teljesen ismeri a játék állását (konfigurációját);

  „teljes információs", nincsenek „rejtett lapok"

- A játék polinom sok lépésben garantáltan véget ér;
- A játék egy-egy állása elfér polinom tárban;
- Minden állásban egyértelmű, hogy melyik játékos következik és polinom tárban meg lehet konstruálni az egyes rákövetkező állásokat;
- Az első játékosnak van nyerő stratégiája az „igen" példányokon, a másodiknak a „nem" példányokon.

ilyet definiáltunk a QSAT-hoz is, de ezek szerint van ilyen játék pl arra is, hogy két reguláris kifejezés ekvivalens-e

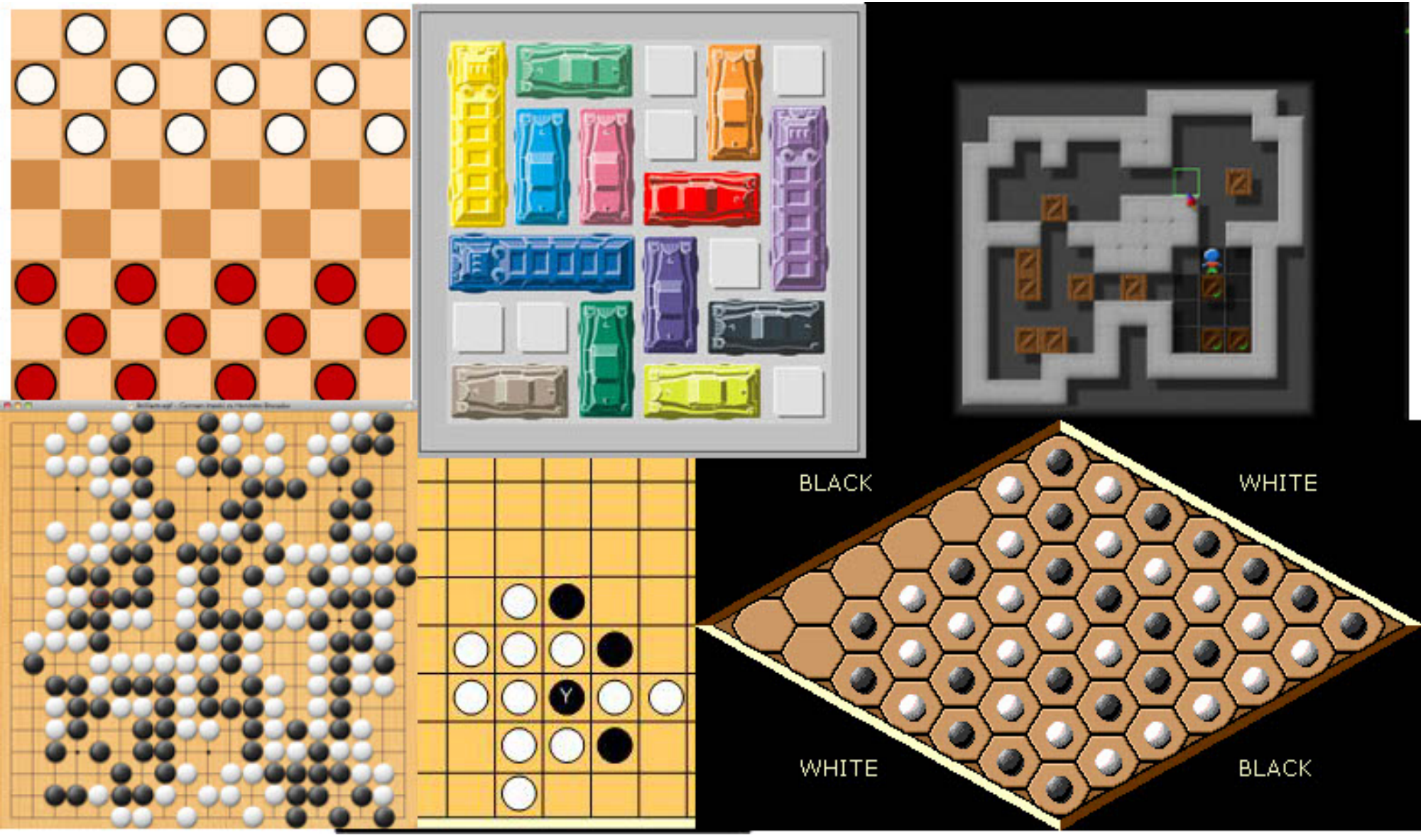
További közismert PSPACE-teljes problémák

Nagyon sok „addiktív” vagy „népszerű” kétszemélyes játékot épp azért nehéz „jól” játszani, mert **PSPACE**-teljesek, mint pl. az alábbiak is:

- Gó: input egy gó állás, melyik játékosnak van nyerő stratégiája?
- Hex: input egy hex állás, melyik játékosnak van nyerő stratégiája?
- Reversi: input egy reversi állás...
- Dáma: input egy dámaállás...
- Amőba: egy amőbaállás...
- Sokoban: megoldható-e az input Sokoban feladvány? (egyszemélyes!)

  és mivel **NL** $\neq$ **PSPACE**, ezért nem lehet visszavezetni az Elérhetőségre, most már tudjuk
- Rush Hour: megoldható-e az input Rush Hour feladvány?
- Életjáték: kipusztul-e minden cella egy adott kezdőállásból elindítva valahány lépésben? („nulla személyes”!)

PSPACE-teljes játékok



$\mathbf{EXP} = \mathrm{TIME}(2^{n^k})$
$\mathbf{NEXP} = \mathrm{NTIME}(2^{n^k})$

Világos, hogy $\mathbf{EXP} \subseteq \mathbf{NEXP}$ és tudjuk, hogy $\mathbf{PSPACE} \subseteq \mathbf{EXP}$.
Nem ismert, hogy az $\mathbf{EXP}$ és $\mathbf{NEXP}$ osztályok megegyeznek-e.

a $\mathbf{P}$ vs. $\mathbf{NP}$ vs. $\mathrm{co}\mathbf{NP}$ kérdések állása nagyon hasonlít az $\mathbf{EXP}$ vs. $\mathbf{NEXP}$ vs. $\mathrm{co}\mathbf{NEXP}$ kérdések állásához, és nem is függetlenek egymástól:

Ha $\mathbf{P} = \mathbf{NP}$, akkor $\mathbf{EXP} = \mathbf{NEXP}$.

Nehéz közismert problémák...

EXP-teljes az...

- ...(általánosított) sakk;
- ...(japán) gó;
- ...(irányított gráfon játszott) rendőrök-rabló játék.

Az időhierarchia tétel miatt tudjuk, hogy az $\mathbf{EXP}$-nehéz problémák biztosan nem oldhatók meg polinomidőben.

Ezek a játékok bár kétszemélyesek, teljes információsak, világos, hogy ki következik, az is, hogy mit léphet, a polinom lépésszámkorlátra vonatkozó feltétel sérül ahhoz, hogy „automatikusan" $\mathbf{PSPACE}$-be kerülhessenek (és hacsaknem $\mathbf{PSPACE} = \mathbf{EXP}$, ami kb. az $\mathbf{NL} = \mathbf{P}$ kérdéssel rokon, ezek tehát nincsenek is $\mathbf{PSPACE}$-ben).

$$\mathbf{EXPSPACE} = \mathrm{SPACE}(2^{n^k})$$

Világos, hogy $\mathbf{NEXP} \subseteq \mathbf{EXPSPACE}$.

Az alábbi probléma **EXPSPACE**-teljes: adott két reguláris kifejezés, melyekben négyzetreemelés is szerepelhet, ekvivalensek-e a kifejezések?

$$\mathbf{ELEMI} = \mathrm{TIME}(2^{n^k}) \cup \mathrm{TIME}(2^{2^{n^k}}) \cup \mathrm{TIME}(2^{2^{2^{n^k}}}) \cup \ldots$$

és van amire egyszerűen nem elég fix magas „torony” se, de eldönthető

Az alábbi eldönthető probléma nem elemi: adott két reguláris kifejezés, melyekben komplementerképzés is szerepelhet, ekvivalensek-e a kifejezések?

Osztálykép

- Hilbert 10. problémája
- Post Megfelelkezési Problémája
- Megállás
- Elsőrendű Formula Tautológia-E