Bonyolultságelmélet gyakorlat – 04
Eldönthetetlenség

Recap: eldönthetetlenség

Egy eldöntési probléma **eldönthetetlen**, ha nincs őt eldöntő algoritmus.

Azaz: bármilyen algoritmust is találunk ki A-ra, vagy lesz olyan input, amin rossz választ ad, vagy lesz olyan, amin nem fog megállni, és nem lesz olyan algoritmus, mely A minden inputján megáll és helyes választ ad.

Recap: Megállás

- **Input:** egy M forráskód és annak egy x inputja
- **Output:** M megáll-e, ha x-en futtatjuk?

Például a következő páros:

```
bool M(int x) {
  while (x > 1) {
    if (x % 2 == 0) x := x / 2
    else x := 3 * x + 1
  }
  return true
}
```

és $x = 6$

a megállási problémának (így együtt az M fenti kód és az $x = 6$ input) egy inputja, sőt egy „igen” példánya (mert a ciklus iterálása közben x értéke rendre $6, 3, 10, 5, 16, 8, 4, 2, 1$ lesz és ekkor megáll a futás, tehát ez az M megáll ezen az x-en).

A Megállás (vagy „megállási”) problémáról **tudjuk, hogy eldönthetetlen**.

Recap: eldönthetetlenség bizonyítása

Ha egy A (eldöntési) probléma eldönthetetlenségét akarjuk megmutatni, akkor:

- veszünk egy **már ismerten** eldönthetetlen B problémát (pl. a Megállást)
- ezt a B-t visszavezetjük az A-ra

A visszavezetésnek nem feltétlen kell ezúttal **hatékony**nak lennie, elég, ha (bármennyi időn belül, de) **kiszámítható** (ennek a jele egyébként $B \leq_R A$).

Ha pl. a megállási problémát akarjuk visszavezetni az új A problémánkra, akkor egy olyan **inputkonverziós függvényt**, egy olyan algoritmust kell megadnunk, ami

- inputként kap egy M forráskódot és annak egy x inputját
- outputként elkészíti A-nak egy I inputját
- választartó módon, azaz: ha M megáll x-en, akkor I az A-nak egy „igen” példánya legyen, ha pedig M nem áll meg x-en, akkor I az A-nak egy „nem” példánya legyen.

A gyakorlaton a feladatokban szinte minden esetben a megállási problémát fogjuk visszavezetni az új problémánkra.

ELFOGADÁS

- **Input:** egy N forráskód és egy y inputja.
- **Output:** igaz-e, hogy $N(y) = \text{TRUE}$?

(vagyis a különbség az ELFOGADÁS és a MEGÁLLÁS közt: ha pl. $M(x) = \text{FALSE}$, akkor ez az (M, x) páros a MEGÁLLÁSnak egy „igen” példánya, mert a kód lefut x-en és visszaad hamisat, tehát megáll, de az ELFOGADÁSnak egy „nem” példánya, mert ahhoz igazat kéne visszaadnia, hogy „igen” példánya legyen.)

1. feladat.

Mutassuk meg, hogy az ELFOGADÁS probléma is eldönthetetlen!

MINDENEN MEGÁLLÁS

- **Input:** egy N forráskód.
- **Output:** igaz-e, hogy N minden lehetséges inputon megáll?

2. feladat.

Mutassuk meg, hogy a MINDENEN MEGÁLLÁS is eldönthetetlen!

Megjegyzés.

Egyébként pl. a gyakanyag első oldalán lévő kódról (ha páros, felezd, ha páratlan, szorozd hárommal és adj hozzá egyet, ha eljutsz 1-ig, állj le) **nem ismert**, hogy minden inputon megáll-e vagy sem[1] – az ilyen típusú kérdések nagyon „egyszerű”nek látszó kódok esetén is lehetnek nagyon nehezek. Jellemzően a while ciklusok, vagy ami ezzel kb. ekvivalens, a rekurzív függvényhívások azok, amik kiszámíthatatlanul bonyolulttá teszik a kód „viselkedésével” kapcsolatos automatikus elemzéseket.

[1] ez az ún. Collatz-sejtés

EKVIVALENCIA

- **Input:** két forráskód, M_1 és M_2.
- **Output:** igaz-e, hogy minden lehetséges y input esetén $M_1(y) = M_2(y)$?

(Azaz: minden input esetén vagy mindkettő elfogad, vagy mindkettő elutasít, vagy egyik se áll meg.)

3. feladat.

Mutassuk meg, hogy az EKVIVALENCIA probléma is eldönthetetlen!

ELÉRHETŐ SOR

- **Input:** egy N forráskód és abban egy i sorszám
- **Output:** van-e olyan inputja N-nek, amin meghívva N-t a vezérlés előbb-utóbb az i. sorra fut rá?

Például ha az ELÉRHETŐ SOR probléma inputja a gyakanyag első oldalán lévő forráskód és a 4. sorszám, akkor ez így együtt egy „igen” példánya az ELÉRHETŐ SORnak, mert pl. ha az input $x = 3$, akkor az első ciklusiterációban ráfutunk az else ágra.

4. feladat.

Mutassuk meg,hogy az ELÉRHETŐ SOR is eldönthetetlen probléma!

Megjegyzés.

Biztos mindenki dolgozott már olyan IDEvel, ami aláhúzással jelezte, ha egy sor nem elérhető (azzal a szöveggel, hogy „dead code” vagy „unreachable code”). Ezek az IDEk természetesen nem döntik el ezt a fenti eldönthetetlen problémát: annyit tudnak, hogy nem esnek végtelen ciklusba, de az egyik irányba tévedhetnek: ha aláhúznak valamit, az biztosan dead code, de nem feltétlenül húznak alá minden dead code-ot, csak amiről rájönnek, hogy az.

Sok esetben pl. egy if(true)return; sor utáni utasításról az IDE már nem jön rá, hogy dead code lesz.

Ez egy gyakori megközelítés: ha egy probléma eldönthetetlen, akkor sokszor van értelme annak, hogy az eredeti probléma helyett annak valami „alulról” vagy „felülről” közelítésére adjunk eldöntési algoritmust, ami csak az egyik irányba téved, és minden inputon garantáltan megáll.

RENDEZÉS-E

- **Input:** egy N forráskód, ami egy `int[]` (int tömb) típusú inputot vár és azt is ad vissza.
- **Output:** igaz-e, hogy N egy rendező algoritmus implementációja?

Azaz: igaz-e, hogy bármilyen int tömböt is kap N, garantáltan meg fog állni és mindig az eredeti tömb elemeit adja vissza, növekvőbe rendezett sorrendben?

5. feladat.

Mutassuk meg, hogy a RENDEZÉS-E probléma is eldönthetetlen!

KONSTANS-E

- **Input:** egy N forráskód, ami egy `int` paramétert vár és egy `boolt` ad vissza
- **Output:** igaz-e, hogy N minden inputra ugyanúgy viselkedik?

azaz: igaz-e, hogy N i) vagy mindenre TRUEt ad vissza, ii) vagy mindenre FALSEt ad vissza, iii) vagy semmin nem álll meg?

6. feladat.

Mutassuk meg, hogy KONSTANS-E is eldönthetetlen probléma!

1. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk az ELFOGADÁS probléma inputját, (N, y) szintén forráskód-input párt **választartó** módon, vagyis:

$$M \text{ megáll } x\text{-en} \Leftrightarrow N(y) = \text{TRUE}.$$

Erre pl. a következő (N, y) páros megfelel:

```
bool N(z) {
  M(z)
  return true
}
```

és $y := x$.

Itt az a fontos, hogy a készített N kódot is és az y inputját is el tudjuk készíteni **pusztán az M forráskód szövegéből mint textből és az x inputból**: itt pl. ha megvan az M függvényünk kódja, bármi is az, ezt a fenti négy sort mögé tudjuk írni és egy új N kódunk lesz belőle, ezt meg tudjuk tenni anélkül, hogy bármit is tudnánk arról, hogy M mit is csinál. Hasonlóan, y-t is simán egy másolással állítjuk elő.

Azaz többek között: **ahhoz, hogy elkészítsük N-t és y-t, nem kell tudnunk, hogy M megáll-e x-en vagy sem** és ez fontos: a visszavezetések pont arról szólnak, hogy úgy konvertáljuk az egyik probléma inputjait a másikéra választartó módon, hogy közben nem „oldjuk meg” egyiket sem, csak azt tudjuk biztosan, hogy a konverzió során a válasz nem változik, bármi is legyen az.

Miután megadjuk a konstrukciót, hogy hogy készüljön el az új input (most N és y), meg kell mutassuk, hogy tényleg tartja a választ. Ez most azért igaz, mert:

- Ha M megáll x-en, akkor N az $y = x$ inputon (azaz az $N(x)$ híváskor) előbb is meghívja M-et x-en, ez valahány lépés után megáll, az eredményt eldobjuk és visszaadjuk, hogy TRUE, tehát ekkor $N(y) = \text{TRUE}$ igaz, „igen” példányból „igen” példány lesz;
- Ha M nem áll meg x-en, akkor N az $y = x$ inputon elindítja M-et x-en és így ő sem fog megállni, azaz $N(y) =\nearrow\neq \text{TRUE}$ lesz, azaz „nem” példányból „nem” példány lesz.

taktika az ilyen feladatok megoldásához

Ezen a gyakon a többi feladatban is érdemes olyan „mintákban” gondolkodni, ha forráskódot készítünk (mint most N), akkor abba kerüljön bele, hogy valamilyen feltétel teljesülése esetén futtatjuk az eredeti M-ünket az eredeti x-en (most azért, mert $y = x$-et adjuk be a z paraméter helyén N-nek), és esetleg aztán **ha** M megáll x-en, utána még csinálunk valamit (most pl. visszaadtunk egy „igaz”at)

2. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk a MINDENEN MEGÁLLÁS probléma inputját, N-t, ami ezúttal csak egy forráskód, **választartó** módon, vagyis:

$$M \text{ megáll } x\text{-en} \Leftrightarrow N \text{ minden lehetséges } y \text{ inputján megáll.}$$

Erre pl. a következő N forráskód megfelel:

```
1 bool N(y) {
2   M(x)
3   return true
4 }
```

(vagyis: bármilyen inputot is kap N, elindítja M-et x-en és **ha** M valahány lépésben megáll x-en, akkor N mindenképp igazat ad vissza.)

Választartó, mert:

- ha M megáll x-en, akkor N is tetszőleges y esetén (miután futtatja M-et x-en) meg fog állni, tehát „igen” példányból „igen” példány készül,
- ha M nem áll meg x-en, akkor N semmin nem fog megállni (mert mindenképp M-et futtatja x-en, bármit is kap), tehát (nagyon) nem igaz, hogy minden inputon megállna, így „nem” példányból „nem” példány készül.

(És persze ez az inputkonverzió kiszámítható is: ha megkapjuk M kódját és egy fix x inputot, akkor gond nélkül ki tudjuk bővíteni egy új N függvénnyel az egészet, ami mást nem is csinál, csak meghívja az M függvényt ezzel a kapott x inputtal, majd returnölünk egy truet.)

3. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk az EKVIVALENCIA probléma inputját, M_1-et, és M_2-t, azaz **két** forráskódot, **választartó** módon, vagyis:

$$M \text{ megáll } x\text{-en} \Leftrightarrow \text{minden } y\text{-ra } M_1(y) = M_2(y).$$

Erre pl. a következő M_1 és M_2 forráskód megfelel:

```
1 bool M1(y) {
2   M(x)
3   return true
4 }
```

```
1 bool M2(y) {
2
3   return true
4 }
```

Az világos, hogy ha megkapjuk M kódját és egy fix x inputot, akkor ezt a két függvényt gond nélkül le tudjuk pluszban implementálni: az egyik simán visszaad mindenre TRUEt, a másik meg még előtte meghívja a kapott M függvényt a szintén megkapott fix konstans x inputtal, azaz ez a konverzió algoritmikusan tényleg kiszámítható.

A választ is tartja, hiszen:

- M_2 minden inputra mindig TRUE-t ad vissza
- ha M megáll x-en, akkor M_1 is (mielőtt ezt megtenné, „fölöslegesen” még meghívja M-et x-en, ami előbb-utóbb lefut, az eredményt pedig eldobja), azaz ilyenkor M_1 és M_2 tényleg ekvivalens, „igen” példányból „igen” példány készül,
- ha M nem áll meg x-en, akkor $M_1(y) =\nearrow$ minden lehetséges y inputra, ami persze egész más, mint $M_2(y) = \text{TRUE}$, tehát ilyenkor nem lesznek ekvivalensek, „nem” példányból „nem” példány készül.

4. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk az ELÉRHETŐ SOR probléma inputját, N-et, és az N kódnak egy i sorszámát **választartó** módon, vagyis:

$$M \text{ megáll } x\text{-en} \iff \text{valamilyen } y\text{-ra } N(y) \text{ kiszámítása közben } N \text{ ráfut az } i. \text{ sorra.}$$

Erre pl. a következő N kód és i sorszám megfelel:

```
1 bool N(y) {
2   M(x)
3   return true
4 }
```

és $i := 3.$ (azaz a „return true” sor)

Ez azért tartja a választ, mert

- ha M megáll x-en, akkor N bármilyen y-t is kap, előbb-utóbb befejezi $M(x)$ kiszámítását, és eztán ráfut a 3. sorra,
- ha viszont M nem áll meg x-en, akkor N bármilyen y-t is kap, a 2. sorbeli függvényhívásból „nem fog kijutni” és sose lép a 3. sorra.

5. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk a RENDEZÉS-E probléma inputját, N-et **választartó** módon, vagyis:

M megáll x-en $\Leftrightarrow$ N mindig az inputjában érkező számokat adja vissza, növekvőbe rendezve.

Erre pl. a következő N kód megfelel:

```
1  int[] N(int[] y) {
2    M(x)
3    for (int i := 0; i < y.length; i++)
4      for (int j := i + 1; j < y.length; j++)
5        if (y[i] > y[j]) {
6          int tmp := y[i]
7          y[i] := y[j]
8          y[j] := tmp
9        }
10   return y
11 }
```

(magyarán: futtassuk M-et x-en, eztán pedig **ha** M megállt x-en, egy buborékrendezést az eredeti tömbön és adjuk vissza az eredményt.)

Ez a konverzió azért tartja a választ, mert

- ha M megáll x-en, akkor bármilyen y tömböt is kap N, először („fölöslegesen") futtatja M-et x-en, ez előbb-utóbb véget ér, az eredményt eldobjuk és ezután ténylegesen rendezzük a tömböt, tehát végeredményben biztos, hogy rendezni fogjuk a tömböt,
- ha M nem áll meg x-en, akkor pedig bármilyen y tömböt is kap N, el se fog jutni a rendezésig, mert „beakad" már az $M(x)$ híváskor, így nemhogy nem egy rendezett tömböt, de semmit se fog visszaadni.

És persze ez az inputkonverzió is elvégezhető automatikusan: ha megkapjuk az M függvény kódját és annak egy fix x inputját, akkor persze gond nélkül tudjuk implementálni ezt az N függvényt. (Kulturáltabb programozási nyelvekben azért van valamiféle beépített SORT függvény/metódus, olyankor persze elég azt meghívnunk egy kézzel implementált buborékrendezés helyett és az is teljesen jó megoldás.)

6. feladat megoldása.

Visszavezetjük rá a MEGÁLLÁS problémát.

Azaz: a megállási probléma inputjából, egy (M, x) forráskód-input párból kell készítsünk a RENDEZÉS-E probléma inputját, N-et **választartó** módon, vagyis:

$$M \text{ megáll } x\text{-en} \iff N \text{ viselkedése független az inputjától.}$$

Erre pl. a következő N kód megfelel:

```
1 bool N(int y) {
2   if (y == 0) return true
3   M(x)
4   return true
5 }
```

Nyilván ezt a függvényt le tudjuk kódolni M és x ismeretében, a választ pedig azért tartja, mert

- ha M megáll x-en, akkor ez a függvény minden inputra TRUEt ad vissza (az $y = 0$-ra gyorsan, a többi lehetséges inputra lassabban, mert előbb még futtatja M-et x-en, de az egyszer megáll és akkor ad vissza N TRUE-t), tehát „igen” példányból „igen” példány készül,
- ha viszont M nem áll meg x-en, akkor pl. $N(0) = \text{TRUE}$ és $N(1) = \nearrow$, amik nem ugyanaz a viselkedés, tehát ekkor „nem” példányból „nem” példány készül.