A típusos generálás folyamata

A cél az, hogy feladat szövegeket, és azok megoldását generáljuk le egyszerre, ehhez egy többlépcsős generáló folyamatot használunk:

1. Véletlen generálunk egy fát, amely teljes egészében leírja a feladatot.  
   Ezt a lentebb felírt szabályokkal, mint generatív nyelvtannal megadott CF nyelv véletlenszerű kifejtésével tesszük.
2. A generált fát lefordítjuk véges automatára, tehát csúcspontonként bejárva a fát, alulról felfelé fölépítjük azt a véges automatát, amely a feladat megoldása. Ezzel tudjuk majd ellenőrizni, hogy a feladatra beérkezett megoldás jó-e.
3. A generált fát lefordítjuk magyar nyelvű feladatszövegre. Ezt a szöveget adjuk a diákoknak.

Ez a többlépcsős folyamat megbonyolítja a generálást azonban igen nagy előnyökkel is jár, mert ha kifejtés közben generálnánk a szövegeket és automatákat, akkor ők nem látnák a (még meg nem épült) fát maguk körül.

Olyan ez, mint amikor mielőtt megszólalunk, átgondoljuk, hogy miről is szeretnénk beszélni, és csak akkor kezdünk el beszélni, amikor ez már megvan. Tehát fölépítjük a szemantikát teljesen mielőtt a mondatot elkezdenénk felépíteni, így nem akadunk meg.

Ennek nagy előnye például a magyar mondat generálásánál, hogy a második lépcsőfokon hozzá tudunk adni környezetfüggő viselkedés az eredetileg környezetfüggetlen nyelvtannal generált szerkezethez. Így a toldalékokat is sokkal könnyebb kezelni. Továbbá az is lényeges előny, hogy a generált fának semmivel nem kell bonyolultabbnak lennie, mint ami már egyértelműen definiálja az etalonnak szánt véges automatát és a magyar nyelvű feladatleírást.

Ennek következtén a véletlengenerálás tulajdonságai is a fa generálásánál dőlnek el.

## A típusos generálás főbb tulajdonságai

### Helyesség:

A típusos generálás mindig helyes, mert a fa szerkezet feldolgozása során garantálni tudjuk ezt.

Jelen esetben is az, mert a konstrukciók folyamán rekurzívan garantáljuk, hogy minden amit generálunk, véges automata. Továbbá a magyar nyelvű feladatleírásnál is garantáljuk a mondatok nyelvtani helyességét a szabályokkal.

### Teljesség:

A típusos generálás általában nem teljes, ennek oka, hogy csak azokat a véges kombinációkat tudja, amiket megadtunk neki a fát felépítő szabályoknál. Ez azért nem baj, mert mint ahogy fentebb említettem nem szokott probléma lenni, hogy a véges generálás nem tud mindent generálni, ha tud elegendően sok variációt generálni.

Jelen esetben azonban lehet teljes is a véges automatákat tekintve, hiszen tudjuk, hogy a véges automaták osztálya megegyezik a reguláris nyelvekével, márpedig bármely reguláris nyelvet felírhatunk az abc-beli betűkből az üres nyelvből, és az üres string alkotta nyelvből kiindulva konkatenációval, unióval és Kleene star-al. Az általam felírt generálási szabályok nem teljesek, mert korlátozva van a generálás szélessége és mélysége a feladatok érthetősége miatt, de ennek a föntebb leírt okokból nincs igazán jelentősége.

### Egyediség

A típusos generálás általában nem egyedi és nehéz azzá tenni, viszont lényeges előnyökkel jár ha legalább nagyjából sikerül egyedivé tenni. Hiszen ekkor kevesebb valószínűséggel generál például feladatokat amik ugyanazok csak máshogy megfogalmazva (szinonimák).

Jelen esetben sem egyedi a generálás, de ezzel nincs különösebb probléma.

Foldok

## Általánosítás

Amikor hétköznapi értelemben számokról beszélünk, akkor olyasmik jutnak eszünkbe, hogy 5, vagy 0.34, esetleg Pi. Ha viszont például van egy olyan mondatunk, hogy

1: „Gondoltam egy szóra, melyben az ’a’ betűk száma 3.”

, akkor érezhető, hogy a következő mondat:

2: „Gondoltam egy szóra, melyben az ’a’ betűk száma 5.”

nagyon hasonló hozzá. Hasonló a mondat alakja(szintaktika), és jelentése is(szemantika).

De nézzük a következő két szót:

„mivel” és „mível”, ezen szavaknak az alakjuk hasonló, ám teljesen mást jelentenek.

Persze mondhatjuk, hogy a világon lehet egy olyan nyelv, amelyben ez a két szó jelentése is hasonló és ez a téma nagyon összetett, mivel az élő nyelv folyamatosan változik, a szavak jelentése, azok szemantikája, vagy akár szintaktikája változhat egy adott nyelven belül is.

Ezért fontos bevezetni a szemantikai leíró fogalmát:

A szemantikai leírás egy algoritmust ír le, amely felismeri, hogy az inputja eleme-e egy meghatározott halmaznak. Ezt a leírót a megfelelő gépbe beadva a gép eldönti, hogy eleme-e az input a halmaznak.

(Fontos megjegyezni, hogy a megfelelő szemantikai leíró átalakításával egy olyan leírót kapunk, amely alkalmas kigenerálni az adott halmaz elemeit. Ez a Turing gépek és általános nyelvek ekvivalenciáját kimondó tétel következménye.)

Továbbá bevezetjük az értelmezés fogalmát:

Az értelmezés egy algoritmus ami egy mondatról egy szemantikai leíróra fordít.

Tehát egy adott értelmezésben az 1-es mondatot lefordítva kaphatjuk a következő szemantikai leírót:

INPUT: word

int count = 0;

for(char c: word){

if(c == ’a’) count++;

}

if(count == 3) ACCEPT;

else REJECT;

A 2-es mondathoz tartozó leíró pedig:

INPUT: word

int count = 0;

for(char c: word){

if(c == ’a’) count++;

}

if(count == 5) ACCEPT;

else REJECT;

Pongyolán azt mondhatjuk, hogy két mondat jelentése hasonló, ha az adott értelmezés mellett a szemantikai leíróik részben hasonlítanak.

Ennél pontosabban megfogalmazva, ha két mondat jelentése hasonló, akkor létezik olyan leíró, amely megfelelően paraméterezve ekvivalens mindkét -az adott értelmezésben lefordított- leíróval.

Vagy talán még szebben úgy is mondhatjuk, hogy a paraméternek, mint szabad változónak létezik olyan megkötése, amely az 1-es leíróval megegyező működést okoz, és létezik olyan megkötése is, amely a 2-es leíróval megegyező működést okoz.

Itt például:

INPUT: word, N

int count = 0;

for(char c: word){

if(c == ’a’) count++;

}

if(count == N) ACCEPT;

else REJECT;

Ilyenkor ezt a leírót a két mondat általánosított szemantikai leírójának nevezzük.

## Indirekció

A következő mondat:

3: „Gondoltam egy szóra, melyben az ’a’ betűk száma kisebb, mint 3.”

szintén nagyon hasonlít az utóbbi 1-es mondatra:

1: „Gondoltam egy szóra, melyben az ’a’ betűk száma 3.”

Ők is azért hasonlítanak, mert van közös általánosított leírójuk:

INPUT: word, numberPredicate()

int count = 0;

for(char c: word){

if(c == ’a’) count++;

}

if(numberPredicate(count) == true) ACCEPT;

else REJECT;

Ahol a numberPredicate(A) az A számnak egy tulajdonságát ellenőrzi.

Az 1-es esetén ez a tulajdonság: A == 3, míg a 3-asnál A < 3.

Azt, hogy az értelmezés során indirekció történik, a szintaktikában is meg szokták jelölni, pl. így:

3b: „Gondoltam egy szóra, melyben az ’a’ betűk száma A, ahol A kisebb, mint 3.”

Az indirekciók alapvető részét képezik bármely kellően hasznos nyelvnek.

Mivel a fentebbi leírók C nyelven iródtak és az őket futtató gép egy számítógép(C fordító->bináris->processzor utasítások), úgy tűnhet, hogy az indirekciókat megvalósító leírók megvalósítása egy triviális dolog. Ez nincs mindig így. Nézzük a következő példát:

4: „Gondoltam egy szóra, melyben az ’a’ betűk száma kisebb, mint A, ahol A kisebb, mint 3.”

Ez egy valid indirekció, de a mondatot még értelmezni is nehéz.

Ha explicitté tesszük az egész kifejezést ez jön ki:

4b: „Gondoltam egy szóra, melyben az ’a’ betűk száma B, ahol B kisebb, mint A és A kisebb, mint 3.”

Tehát 0 <= B < A és A < 3, ahol A és B szabad változók, vagyis bármilyen értéket felvehetnek.

A következő kombinációk lehetségesek (A,B) párokkal felírva:

(1,0),(2,0),(2,1), tehát a B(vagyis az ’a’ betűk száma) a {0,1} halmazból kerülhet ki.

Ezt C kódban sem könnyű kezelni. Ugyanis, ha ezt az indirekciót implicitté tesszük a mondatban ezt kapjuk:

4c:”Gondoltam egy szóra, melyre van olyan szám, ami az ’a’ betűk számánál nagyobb, de 3-nál kisebb.”

Ez pedig egy elsőrendű logikai kifejezés.

C kódban az ilyeneket kezelhetjük úgy, hogy kötött B-re és szabad A-ra rezolúcióval megoldjuk a kielégíthetőséget. Vagy akár ezt a folyamatot be is építhetjük magába a kódba. A prolog is hasonlókat csinál, de ebbe most nem megyek bele, mert csupán érinti a témát.

## Foldok

Az általánosítások és indirekciók kezelése automatáknál sokkal nehezebb, mert az ilyen paramétereket nem tudjuk csak úgy átadni az automatának, hiszen annak egyetlen paramétere a szó amit felismer vagy nem.

Ezeket a paramétereket bele kell építeni az automatába a konstrukció közben, ezt nevezem itt foldnak.

Első példa:

Például ha egy olyan automatát szeretnénk építeni, amely felismeri, hogy szerepel a szóban részszóként ’aba’, akkor ezt az ’aba’-t nem tudjuk csak úgy átadni az automatának. A konstrukció folyamán beépítjük. Ennek legegyszerűbb módja ha egy epsilon átmenetes automatába csinálunk 4 állapotot, és 3 átmenetet köztük sorban ’a’, ’b’,’a’ betűkre. Majd rakunk az elejére és a végére is Sigma csillagot, majd epsilon mentesítünk és determinizálunk.

Másik példa:

Automata, amely ’ab’-t tartalmaz N-szer, ahol N-re igaz valamiféle predikátum(pl páros, =3, >2, stb). Ezt a predikátumot szintén automatával adjuk meg, ahol a szám növekedésére egy ’e’ betűs, nem növekedésére pedig egy ’z’ betűs állapotátmenetet teszünk. Ezután kreálunk egy automatát, amely akkor fogad el, ha a szó ’ab’-re végződik. Majd a két automatát párhuzamosan futtatva hatványhalmaz konstrukciót csinálunk, ahol az N-es automata az ’e’-s átmenetén akkor lép, ha az ’ab’-re végződős automata végállapotba ér, különben a ’z’-s állapotra lép.

Ez logikus is, hiszen ha egy szó ’ab’-t N-szer tartalmaz, az azt jelenti, hogy a szó beolvasása során az ’ab’ N-szer lesz az addig beolvasott szó végén.

Harmadik példa:

Automata, amelynek minden N egymás utáni betűjére igaz valami predikátum. A predikátumot szintén automatával adjuk meg. Itt először csinálunk minden lehetséges N egymás utáni betűre egy állapotot, majd ezeken az N elemű szavakon futtatjuk a predikátumot, mint automatát. Ha elfogadta, akkor célállapot lesz, ha nem akkor nem. Továbbá minden N-nél rövidebb szó automatikusan célállapot, hiszen arra nem vonatkozott a megkötés.

A példákból látható, hogy ez egy nagyon fontos része a generálási folyamatnak. Ezért definiálom a fold fogalmát pontosabban is:

Adott egy probléma, amelyben van legalább egy szabad változó. Adott egy, a szabad változókat megkötő állítás. Foldnak nevezzük azt az algoritmust, ami előállít egy algoritmust, ami megoldja a problémát a változók kötése mellett.

Például:

Probléma: W input szó tartalmazza-e S szót. (S itt egy szabad változó, bármi lehet)

Kötő állítás: S = ’aba’, vagy S a-ra végződik.

Fold: (Probléma, kötő állítás) -> Algoritmus (itt véges automata)

Algoritmus: Eldönti, hogy W-nek része-e ’aba’, vagy a másik példában, hogy W-nek része-e olyan szó ami a-ra végződik.

Az algoritmusokat a generálás során következetesen véges automatákkal adtam meg, ezzel garantálom, hogy akárhány fold után is olyan nyelv keletkezzen, amely felismerhető véges automatával.

Végezetül egy lista a foldokról (aszerint, hogy mi a kötés) nehézsége szint szerint:

Fix szó

Véges szó halmaz

Predikátum(=algoritmussal megadott szó, valószínűleg végtelen halmazból)

Predikátumokból álló logikai kifejezés szabad változó nélkül(=Ítéletlogikai kifejezés)

Logikai kifejezés másik kötött változóra történő hivatkozással(=Első és magasabb rendű kifejezés)

Nyelvi problémák a típusos generálás körül

A típusos generálás során egy kontextusfüggetlen (cf) nyelv segítségével generálunk szavakat a formális nyelvek órán megismert kifejtő módszerrel. Ez így egyszerű, hiszen egy faszerkezetben kifejthetők a szemantikus elemek, viszont ez korlátokat is jelent. Ezen korlátok van, hogy (1) elméleti jellegűek, tehát egy bizonyos feladat fajtát nem tudunk megvalósítani. És van, hogy (2) gyakorlati jellegűek, tehát ugyan a feladatot meg tudjuk valósítani, de nehézkes, problémás.

Általánosan a kontextus független nyelvek korlátja az, hogy nem lehet bennük egy adott elem kifejtésétől függően kifejteni egy másikat.

Mivel a feladatok általánosan a  
 <valamiről> <állítok valamit>   
sémára épülnek ezért egy cf nyelvvel csak úgy tudjuk generálni ha lefixálunk egy típust, például, <valamilyen mérték, ami egy egész szám és jelöljük n-el> <n páros>

## Elméletileg sem lehetséges

A következőt például cf nyelvvel (legalábbis általánosan) nem lehet megcsinálni:

ba^ncb^ma : n - m > 5, n páros

<miről állítunk> : <mit> sémában itt baloldalon definiáljuk, hogy mi is ez az n és m, és ezekre jobboldalán szeretnénk valami állítást megfogalmazni, tehát balról jobbra kell, hogy menjen az az információ, hogy van egy n és egy m egész szám. Ez így, ebben a konkrét esetben működhet, hiszen csinálunk egy típust, ahol fixen n és m változókról állítunk dolgokat, de ez általánosan nem megy:

ba^ncb^mac^k : n-m > 5, n páros, n + k páratlan

Ezt már az előző sémával nem tudjuk megcsinálni, mert nem megy át a k. Általánosan is az a probléma, hogy a <miről állítunk> blokkban keletkeznek a változók amikről állítani szeretnénk valamit és végtelen sok is keletkezhet, erre pedig véges sok típussal nem fogunk tudni felkészülni.

## Lehetséges, de gyakorlatban nehézkes

Sokat elmond a cf nyelvek korlátozottságáról, hogy ha terminális abc elemeitől szeretnénk függővé tenni valamilyen kifejtést, azt nagyon nehezen tehetjük meg. A következő példa ezt jól szemlélteti:

|W|a - |W|b + |W|c páros

Itt arról van szó, hogy van egy aritmetikai kifejezésünk, amely bizonyos abc-beli betűkből alkotott aritmetikai kifejezés eredményéről állít valamit (itt például azt, hogy páros). Cf nyelvben ezt meg lehet csinálni, de minden alkalommal, amikor az abc-nk változik, az összes segédterminálisunkat újra kell írni.

Ezen úgy tudtam segíteni, hogy egy for ciklussal az abc specifikus szabályokat dinamikusan betöltöm a generatív nyelvtanba generálás előtt.

A magyar nyelvű feladatszöveg generálása

Amikor a generált fából a mondatokat összerakjuk, az a legnagyobb nehézség, hogy a mondatok értelmesek legyenek, és ne úgy nézzen ki, mint amit google fordítóból szedtek ki.

Itt sok apró probléma van, amelyeket egyesével felsorolok:

## Toldalékok

Ahhoz, hogy értelmes mondatok keletkezzenek, fontos, hogy használjunk toldalékokat, és a megfelelő alakban használjuk azokat.

„A kutya ül a fű.”, vagy „4-al osztható” jellegű kifejezések erősen kétségbe vonnák a feladatok komolyságát.

A megfelelő toldalék választásával nem szokott sok gond lenni. A hangalak egy kicsit problémásabb. Általában a szavak, szótagok magas, mély és vegyes hangrendűségétől függ a toldalékok hasonulása.

Magas hangrendű magánhangzók: e,é,i,í,ö,ő,ü,ű

Mély hangrendű magánhangzók: a,á,o,ó,u,ú

Egy szó magas hangrendű, ha benne csak magas hangrendű magánhangzók vannak, például: „kilenc”, ilyenkor a toldaléka is magas: „kilenccel”

Egy szó mély hangrendű, ha benne csak mély hangrendű magánhangzók vannak, például: „kalóz”, ilyenkor a toldaléka is mély: „kalózzal”

Egy szó vegyes hangrendű, ha benne magas és mély hangrendű magánhangzók is vannak, például: „Dóri”, ilyenkor a toldaléka mély: „Dórival”

Ennél bonyolultabb esetek is vannak a toldalékok hangalakjára, de egyszerűbb esetekben ez működik.

Nekünk viszont nincs szükségünk ilyen bonyolult logikai összefüggések kódolására, hiszen amikre toldalékok kerülnek azok vagy számok (amelyeknek utolsó tizedesjegye 0-9 lehet, tehát véges sok), illetve szavak, melyeknek az utolsó betűjéhez hasonul a toldalék, és ezekből is véges sok van, ezért elegendő egy switch case-el minden lehetőségre kézzel beégetni a toldalékokat.

## Negálás

Magyar nyelvben a negálás nem egyszerűen az, hogy elé rakunk egy „nem”-et az állításnak, hiszen akkor olyan kifejezések keletkeznének, hogy „nem büszke vagyok”.

Például:

„W ab-vel kezdődik” -> „W nem kezdődik ab-vel” (tehát a szórend itt változik)

„W-ben az a-k száma osztható 5-el” -> „W-ben az a-k száma nem osztható 5-el” (a szórend itt nem változik, elég kirakni a „nem”-et)

„Bármely a betű után áll egy b betű” -> „Van olyan a betű, amely után nem áll b betű” (ez egy elsőrendű kifejezés, ezt még nehezebb értelmesen negálni)

Ezeket az esetek CF nyelvvel kezelni elég nehéz, de nem lehetetlen, hiszen minden mondatnak kétféle alakja van, a negált és a negálatlan, tehát véges sok variáció van, ezért generálható cf nyelvvel, csak nem célszerű. Erre a célra a kódban az értelmezés során egy „ál”-kontextfüggő működést raktam be, amely minden negálásnál lefele tol egy jelzést, hogy minden állítás maga tudja eldönteni, hogyan viselkedjen negálás esetén.

## Logikai kifejezések

Amíg egy elsőrendű logikai kifejezésnél természetes a rengeteg egybeágyazott és, nem, vagy, kvantor stb. , addig magyar nyelvben ezeket nem lehet csak úgy zárójelezéssel végtelenül halmozni.

Ha pl. azt akarjuk negálni, hogy „fehér a hó és kék az ég”, akkor azt úgy szokás, hogy „nem igaz az állítás, hogy fehér a hó, és kék az ég”, tehát itt egy indirekciót kell explicit jelölnünk az állításra. Ezt a technikát tetszőleges logikai kapcsolatokra használhatjuk:

„Van egy állításom, hogy piros a hó és zöld az ég, és van egy másik állításom, hogy kék az aszfalt vagy fehér a hold. Ha ezek közül valamelyik igaz, akkor te nyertél.”

Ez azonban kicsit körülményes és nehezen értelmezhető tud lenni.

De vannak kivételek, például ha kizárólag „vagy”, illetve „és” kapcsolatokat halmozunk:

„Dél volt és meleg és tűzött nap és nem volt árnyék, de mégis dolgoztunk tovább.”

A generálás során a programban inkább az ilyen kivételes fajtákból generálok, mert ezeknek természetesebb a hangzása.

## Időrendben fordított toldalékolás

Ezt különvettem, mert itt az történik, hogy az a szó még nem hangzott el, ami a toldalékot meghatározza, de a toldalék már ott van egy másik, őt megelőző szón. Például:

„Szó, ami a-val kezdődik.” de:

„Szó, amiben legalább kettő b betű van.”, itt a „benne van” ige már az előtt toldalékot tesz az ami-re, hogy fölolvasásra kerülne.

A feladatmegoldás modellezése

Ebben és a következő fejezetben pár gondolatot írok, amelyeket a feladatok nehézségének beállításához használtam volna, de az időkorlát miatt nem ezeknek az implementálására nem került sor. Itt egy egyszerű modellt mutatok be a problémára.

A feladatmegoldás folyamatát 5 fázisra bontottam:

Ebben a sorrendben: értelmezés, modellezés, redukció, konstrukció, implementáció

Részletesebben:

### Értelmezés

Ebben a részben a feladatot a hallgató meg kell, hogy értse. Ehhez hozzá tartozik, hogy például egy automatás feladatnál már tud példákat mondani, hogy milyen szavakat fog elfogadni, és miket nem. Ez triviálisnak hangzik, de egy hosszabb szöveges feladatnál ez sok időbe telhet és könnyű hibázni is.

### Modellezés

Ebben a részben az értelmezett feladatot olyan alakra kell bontania, olyan fogalmakkal kell párhuzamba hoznia a hallgatónak, amelyekkel kapcsolatosan már ismer matematikai fogalmakat, és amelyek absztrakt módon fogják meg a feladat lényegét. Ilyen például egy való életbeli optimalizálós feladat felírása például egy egyenletrendszerrel, vagy logikai formulákkal, esetleg egy gráffal, stb. Automatás feladatoknál ennek kevesebb a jelentősége, később látni fogunk olyan típust ahol gráftulajdonságokat kell ellenőrizni automatával, ott a gráftulajdonságok szótulajdonságokra való átalakítása például modellezés.

### Redukció

Ebben a részben a már megértett és kezelhető fogalmakra hozott feladatot olyan alakra kell átalakítani, amit a hallgató meg is tud oldani. Már a modellezés menete is személyfüggő lehetett, de ez a fázis már végképp az, hiszen egy problémát többféleképpen meg lehet oldani. A legjobb az, ha minél egyszerűbb módot talál a hallgató. Automatánál redukció az például, hogy felismerjük, hogy az a szó, ami „a-val kezdődik és ba-val kezdődik” nem létezhet és ezzel a feladat nagyjából meg is van oldva. Továbbá redukció az is, amikor olyan ekvivalens alakra hozzuk a feladatot, amit könnyen meg tudunk oldani.

### Konstrukció

Ebben a részben a már kellően redukált és jól megfogalmazott feladatra találnunk kell egy módszert(algoritmust), ami megoldja azt. Konstrukció például két automata metszete. De ez nem ilyen egyszerű, mert a hallgató valószínűleg kevés konstrukciót ismer, illetve amit ismer azt sem biztos hogy jól. Ezért itt eszébe kell, hogy jusson a konstrukció menete, vagy ha nem ismeri, akkor ki kell találnia a konstrukció menetét, ami nehéz és időigényes. Ezt a részt a redukciótól nehéz elválasztani, hiszen a hallgató pont olyan alakra próbálja redukálni a feladatot, amire van általa jól ismert konstrukció, de ez nem mindig sikerül.

### Implementáció

Itt a már ismert/kitalált konstrukció elvégzése a feladat, itt áll elő a megoldás. Ez könnyűnek tűnhet, de ha nagy a feladat, akkor könnyű elszámolni, hibázni. Itt a pontosság, következetesség a fontos.

## Részfeladat megoldásának lehetséges útjai

A föntebbi modellből látható, hogy a feladatmegoldás útja általában koránt sem egyenes, ezért az időigénye instabil lehet, hiba esetén sokszor újra kell kezdeni egy részt ami sok időt ad a feladat megoldásának idejéhez. Ennek modellezésére pár lehetséges út a részfeladat megoldásához:

### Intuíció

A hallgató pontosan tudja, hogyan kell megoldani az adott feladatot, valószínűleg azért, mert már rengeteget gyakorolta azt a részt.

Ennek időigénye minimális és stabil. A hiba esélye minimális.

### Keresés

A hallgató ismeri a részfeladat megoldásának menetét, de nem emlékszik pontosan, ki kell próbálnia pár variációt mire rájön melyik is volt a jó.

Ennek időigénye közepes, és relatíve stabil. A hiba esélye csekély, mert ha rosszul kezdi el, érezni fogja, hogy ez nem az, mint ahogy korábban, például az órán megoldotta.

### Levezetés

A hallgató nem ismeri a részfeladat megoldásának menetét, sosem csinált ilyet, de birtokában van olyan ismereteknek, amelyekből a megoldás helyes menete levezethető.

Ennek időigénye már magas, és igen instabil, a hiba esélye nagy. Nagyban függ attól, hogy a hallgató például kipihent-e, és mekkora gyakorlata van az adott területen. Ha rossz irányba indul el az lehetetlenné is teheti az időben történő megoldást.

### Találgatás

Ha levezetni sem tudja a megoldást, akkor nincs más, mint hogy találgat. Kellően összetett feladatnál ez szinte lehetetlen is lehet.

Az időigénye a legmagasabb, leginstabilabb és szinte biztos a hiba.

De fontos megemlíteni, hogy a találgatás nehézségét nagyban meghatározza az, hogy mekkora a keresési tér. Ha a keresési tér kicsi, akkor a találgatás is időben stabil.

Itt nagyon lényeges az is, hogy milyen visszacsatolása van a hallgatónak a találgatás során. Belső visszacsatolás esetén a hallgató látja azt, hogy a megoldása nem jó, mert például a példa inputokból nem a megfelelő példa outputot csinálja. Külső visszacsatolás esetén többször is beküldheti a feladatot és láthatja, hogy az jó-e.

Hosszabb távon ha a keresési tér kicsi, és van visszacsatolás, akkor a hallgató látja, hogy melyik feladatra mi a jó megoldás és már ebből is képes következtetéseket levonni, amelyek birtokában egy későbbi feladatot már lehet, hogy le is tud vezetni.

# A feladatmegoldás céljai

A fentebbi modellek látván felmerül a kérdés, milyen feladatot kapjon a hallgató, mi a célja a feladatnak, miben fejleszti a hallgatót, mit kér pontosan számon.

Egy kiegyensúlyozott feladatban lehetséges, hogy mind az 5 fázis nehéz, de attól függően, hogy mi a cél, az egyes fázisokat lehet nehezebbé, könnyebbé tenni:

A lehetőségek fázisokra bontva:

### Értelmezés

Ha a szövegértést szeretnénk hangsúlyozni, akkor fontos, hogy a feladat kellően összetett módon, hosszan legyen megfogalmazva, ám a feladat lényegében ne legyen annyira összetett. Tehát valamilyen egyszerű dolgot kell minél bonyolultabban tálalni.

Az értelmezés során az intuíción van a hangsúly, mert ha hallgató nem érti a feladatot, és nem kap hozzá megfelelő magyarázatot és példákat, akkor a keresés, levezetés, találgatás értetlen, nem fogja tudni megoldani. Az időigény ezért stabil, vagy tudja és akkor kevés, vagy nem és akkor végtelen.

### Modellezés

A modellezést nem könnyű nehezebbé tenni, mert a keresési tér nagysága miatt itt nem nagyon lehet levezetni, találgatni. Ha az órán gyakorolt modelleket nem ismeri, nem valószínű, hogy megtalálja a megfelelőt. Az időigény itt is stabil, vagy tudja és kevés, vagy nem és akkor végtelen.

### Redukció

A redukciót az értelmezéshez hasonlóan úgy lehet nehézzé tenni, ha a matematikai fogalmakkal már felírt modell összetett, de azon sokat lehet egyszerűsíteni, vagy vannak rá olyan módszerek, amellyel könnyen megoldható. Az időigény itt már instabil, mert lehetőség nyílik és általában szükség is van keresésre, levezetésre, ami attól függően, hogy mennyi zsákutcába fut a hallgató sok is lehet és kevés is.

### Konstrukció

A konstrukciók nehézsége összeforr a redukciókéval. Ha nehézzé akarjuk tenni a feladatot, akkor nem engedjük, hogy azt egyszerű konstrukcióval is meg lehessen oldani és odafigyelünk, hogy a feladat ne legyen egy egyszerű, speciális eset. Tehát itt megszabhatjuk közvetlen, hogy mennyire legyen nehéz és időigényes a konstrukció, azzal, hogy milyen konstrukciókra engedjük redukálni a feladatot.

Itt nagyon lényeges a konstrukciók algoritmikus összetettsége, mert ez fogja meghatározni, hogy mennyire lehet azt levezetni, vagy esetleg még találgatni is.

### Implementáció

Ennek nehézsége nagyban függ a végeredmény nagyságától, a konstrukciók elvégzésének hosszától. Ha sok számolás kell, akkor nehezebb az implementáció és könnyebb hibázni.

Ettől függ az is, hogy mennyire lehet találgatni.

A feladatok nehézségének szabályozása

Bármilyen kisebb célra is fókuszálunk a feladatok kapcsán, minden esetben fontos, hogy a hallgató a feladatok által a célok felé haladjon, azokat egyre jobban megközelítse. A hallgató nem kaphat túl könnyű feladatot, de túl nehezet sem. Tehát döntenünk kell, hogy egy adott hallgatónak mikor, milyen feladatokat adunk, hogy fejlesszük. Ha ezt egy programmal szeretnénk megtenni egy e-learning rendszerben, akkor több lehetőségünk is van, íme pár:

## Statikus fejlődési fa

Ennél a modellnél minden feladattípushoz meghatározzuk, hogy mik az előfeltételei. Először csak olyan feladatok közül kap feladatot a hallgató amelyek könnyűek és nincs előfeltételük. Erre egy fát építünk fel, amelyben felírjuk a típusokat, és hogy a típusnak mik a feltételei.

Ez nagyon egyszerű és hatékony is, hiszen így a feladatok nehézségét a hallgató tudásához tudjuk egyeztetni. Ilyet terveztem az e-learning rendszerhez, de végül idő hiányában ez nem kerül implementálásra.

### Statisztikai módszerek

Az előző módszernél azt az állítást, hogy a hallgató meg tud oldani egy feladatot, feltéve, hogy azok előfeltételeit meg tudja oldani a fa készítőjének személyes tapasztalataira alapoztuk. De ezen előfeltételek statisztikai módszerekkel is felépíthetjük. Ha megfelelő mennyiségű adatunk van arról, hogy milyen feladatokat mely hallgató mennyi idő alatt oldotta meg, akkor ebből akár ebből tetszőleges statisztikai módszerekkel (akár neurális hálókkal) meg tudjuk becsülni, hogy melyek azok a feladatok, amelyek egy hallgatónak pont megfelelő nehézségűek, de akár egy fejlődési fát is fölépíthetünk az adatokból.

Ez egy érdekes módszer és valószínűleg gyakorlatban ez lenne a leghatékonyabb.

### Feladatmegoldási modellel történő finomítás

A statisztikai módszerekkel kontrasztban, elkezdhetjük a megoldási folyamatot apróbb részekre bontani, ezek egyrészt a már fentebb ismertetett általános modellek, másrészt magukat a feladatokat is atomjaikra tudjuk bontani.

Ilyen például, hogy a hallgató képes-e felismerni, hogy két automatát egyszerre működtetve az állapotaikat egy automataként is modellezheti, vagy hogy egy bizonyos betű számolása során a releváns információk véges halmazból tevődnek ki, és ezek megfeleltethetőek állapotoknak.

Ez a módszer nagyon bonyolult, és bár elméletileg ez lenne a legerősebb, a kivitelezés összetettsége miatt valószínűleg a statisztikai módszerek jobb eredményt adnak.

A feladatok sokszínűségének növeléséről

### Természetes fogalomtér

String/betű, számok, állítások

### Új típusok

### Kézi

### Modellváltós típusok

### Automatikus

### Emergens típusok

A feladatgeneráló rendszerek előnyei a hagyományos módszerekkel szemben

Minden speciális eset lefedése

Személyre szabott progresszió

Könnyebb javítás, digitalizált rendszer

Programozástechnikai nehézségek és tapasztalatok