

- i. Algoritmusok és Adatszerkezetek I
 - 1.1. 1. Részproblémára bontható algoritmusok (mohó, oszd-meg-és-uralkodj, dinamikus programozás), rendező algoritmusok, gráfalgoritmusok (szélességi- és mélységi keresés, minimális feszítőfák, legrövidebb utak)
 - 1.1.1. Mohó algoritmusok
 - 1.1.2. Oszd-meg-és-uralkodj algoritmusok
 - 1.1.3. Dinamikus programozás
 - 1.1.4. Rendező algoritmusok
 - 1.1.5. Gráfalgoritmusok
 - 1.2. 2. Elemi adatszerkezetek, bináris keresőfák, hasító táblázatok, gráfok és fák számítógépes reprezentációja
 - 1.2.1. Listák
 - 1.2.2. Verem
 - 1.2.3. Sor
 - 1.2.4. Prioritási sor
 - 1.2.5. Fák, és számítógépes reprezenzációjuk
 - 1.2.6. Bináris keresőfák
 - 1.2.7. Halmaz
 - 1.2.8. Szótár
 - 1.2.9. Hasító tábla
 - 1.2.10. Gráfok számítógépes reprezentációja
- ii. Bonyolultságelmélet
 - 2.1. 1. Hatékony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák.
 - 2.2. A P osztály
 - 2.2.1. Elérhetőség
 - 2.2.2. Hatékony visszavezetés
 - 2.2.3. Nemdeterminizmus
 - 2.2.4. Az NP osztály
 - 2.2.5. NP-teljes problémák
 - 2.3. 2. A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárígényű visszavezetés. NL-teljes problémák.
 - 2.3.1. A PSPACE osztály
 - 2.3.2. Offline, vagy lyukszagos tárígény
 - 2.3.3. Az NL-osztály
 - 2.3.4. Logtáras visszavezetés
 - 2.3.5. NL-teljes problémák
 - 2.3.6. PSPACE-teljes problémák
- iii. Formális Nyelvek
 - 3.1. 1. Véges automata és változatai, a felismert nyelv definíciója. A reguláris nyelvtanok, a véges automaták, és a reguláris kifejezések ekvivalenciája. Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei.
 - 3.1.1. Véges automata
 - 3.1.2. Ekvivalencia tételek
 - 3.1.3. Pumpáló lemma reguláris nyelvekre
 - 3.1.4. A pumpáló lemma alkalmazása
 - 3.1.5. Következmények
 - 3.2. 2. A környezetfüggetlen nyelvtan, és nyelv definíciója. Derivációk, és derivációs fák kapcsolata. Veremautomaták, és környezetfüggetlen nyelvtanok ekvivalenciája. A Bar-Hillel lemma és alkalmazása.
 - 3.2.1. Környezetfüggetlen nyelvtan
 - 3.2.2. Derivációs fák, kapcsolatuk a derivációkkal
- iv. Közelítő és szimbolikus számítások
 - 4.1. 1. Eliminációs módszerek, mátrixok trianguláris felbontásai. Lineáris egyenletrendszerek megoldása iterációs módszerekkel. Mátrixok sajátértékeinek, és sajátvektorainak numerikus meghatározása.
 - 4.1.1. Eliminációs módszerek
 - 4.1.2. Mátrixok trianguláris felbontásai
 - 4.1.3. Lineáris egyenletrendszerek megoldása iterációs módszerekkel
 - 4.1.4. Mátrixok sajátértékeinek, és sajátvektorainak numerikus meghatározása
 - 4.2. 2. Érintő, szelő, és húr módszer, a konjugált gradiens eljárás. Lagrange interpoláció. Numerikus integrálás.
 - 4.2.1. Érintő módszer
 - 4.2.2. Szelő módszer
 - 4.2.3. Húr módszer
 - 4.2.4. Konjugált gradiens eljárás
 - 4.2.5. Lagrange interpoláció
 - 4.2.6. Numerikus integrálás
- v. Logika és informatikai alkalmazásai

- 5.1. 1. Normálformák az ítéletkalkulusban, Boole-függvények teljes rendszerei. Következtető módszerek: Hilbert-kalkulus és rezolúció, ezek helyessége és teljessége.
 - 5.1.1. Normálformák az ítéletkalkulusban
 - 5.1.2. Boole-függvények teljes rendszerei
 - 5.1.3. Hilbert rendszere
 - 5.1.4. Rezolúció
- 5.2. 2. Normálformák az elsőrendű logikában. Egyesítési algoritmus. Következtető módszerek: Alap rezolúció, és elsőrendű rezolúció, ezek helyessége és teljessége.
 - 5.2.1. * Elsőrendű logika alapfogalmak
 - 5.2.2. Normálformák az elsőrendű logikában
 - 5.2.3. Alap rezolúció
 - 5.2.4. Elsőrendű rezolúció
- vi. Mesterséges Intelligencia 1.
 - 6.1. 1. Keresési feladat: feladatreprézentáció, vak keresés, informált keresés, heurisztikák. Kétszemélyes zéró összegű játékok: minimax, alfa-béta eljárás. Korlátozás kielégítési feladat.
 - 6.1.1. Keresési feladat
 - 6.1.2. Kétszemélyes zéró összegű játékok
 - 6.1.3. Korlátozás kielégítési feladat
 - 6.2. 2. Teljes együttes eloszlás tömör reprezentációja, Bayes hálók. Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere.
 - 6.2.1. * Alapfogalmak
 - 6.2.2. * Függetlenség
 - 6.2.3. * Feltételes függetlenség
 - 6.2.4. Teljes együttes eloszlás tömör reprezentációja
 - 6.2.5. Bayes-hálók
 - 6.2.6. Gépi tanulás
 - 6.2.7. Mesterséges neuronhálók
 - 6.2.8. k-legközelebbi szomszéd módszere
- vii. Operációkutatás
 - 7.1. 1. LP alapfeladat, példa, szimplex algoritmus, az LP geometriája, generálóelem választási szabályok, kétfázisú szimplex módszer, speciális esetek (ciklizáció-degeneráció, nem korlátos feladat, nincs lehetséges megoldás)
 - 7.1.1. Alapfogalmak
 - 7.1.2. LP alapfeladat
 - 7.1.3. Példa
 - 7.1.4. Egy lineáris program felírása
 - 7.1.5. LP feladat megoldása
 - 7.1.6. Szimplex algoritmus
 - 7.1.7. Kétfázisú szimplex algoritmus
 - 7.1.8. LP és konvex geometria
 - 7.2. 2. Primál-duál feladatpár, dualitási komplementaritási tételek, egész értékű feladatok és jellemzőik, a branch and bound módszer, a hátzsák feladat
 - 7.2.1. Primál-duál feladatpár
 - 7.2.2. Komplementaritás
 - 7.2.3. Egész értékű programozás
 - 7.2.4. Korátozás és szétválasztás (branch and bound)
 - 7.2.5. Hátzsák feladat
- viii. Operációs rendszerek
 - 8.1. 1. Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása. Ütemezési stratégiák és algoritmusok kötegelt, interaktív és valós idejű rendszerekben, ütemezési algoritmusok céljai. Kontextus-csere.
 - 8.1.1. Alapfogalmak
 - 8.1.2. Processzusok létrehozása
 - 8.1.3. Processzusok befejezése
 - 8.1.4. Processzusok állapotai
 - 8.1.5. Kontextus-csere
 - 8.1.6. Ütemezés
 - 8.1.7. Ütemezés kötegelt rendszerekben
 - 8.1.8. Ütemezés interaktív rendszerekben
 - 8.1.9. Ütemezés valósidejű rendszerekben
 - 8.2. 2. Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárási mechanizmusok. Konkurens és kooperatív processzusok. Kritikus szekciók és megvalósítási módszerek: kölcsönös kizárási meccsen (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás). Altatás és ébresztés: termelő-fogyasztó probléma, szemaforok, mutex-ek, monitorok, Üzenet, adás, vétel. Írók és olvasók problémája. Sorompók.
 - 8.2.1. Processzusok kommunikációja

- 8.2.2. Versenyhelyzetek
 - 8.2.3. Kritikus szekciók
 - 8.2.4. Kölcsönös kizárási mechanizmusok
 - 8.2.5. Kölcsönös kizárási mechanizmusok tevékeny várakozással
 - 8.2.6. Altatás és ébresztés
 - 8.2.7. Termelő-fogyasztó probléma
 - 8.2.8. Szemaforok
 - 8.2.9. Mutex-ek
 - 8.2.10. Monitorok /nem érhető/
 - 8.2.11. Üzenet
 - 8.2.12. Írók és olvasók problémája
 - 8.2.13. Sorompók
- 8.3. 1. Adatbázis-tervezés: A relációs adatmodell fogalma. Az egyed-kapcsolat diagram és leképezése relációs modellre, kulcsok fajtái. Funkcionális függőség, a normalizálás célja, normálformák.
 - 8.3.1. A relációs adatmodell fogalma
 - 8.3.2. E-K modellből relációs modell
 - 8.3.3. Adatbázis normalizálása
 - 8.3.4. Funkcionális függés
 - 8.3.5. Felbontás (dekompozíció)
 - 8.3.6. Normalizálás
 - 8.4. 2. Az SQL adatbázisnyelv: Az adatdefiníciós nyelv (DDL) és az adatmanipulációs nyelv (DML). Relációsémák definiálása, megszorítások típusai és létrehozásuk. Adatmanipulációs lehetőségek és lekérdezések.
 - 8.4.1. Az SQL nyelv
 - 8.4.2. Relációsémák definiálása (DDL)
 - 8.4.3. Adattábla aktualizálása (DML)
 - 8.4.4. Lekérdezés (DML)
 - 8.4.5. Aktív elemek (megszorítások, triggerek)
- ix. Digitális képfeldolgozás
 - 9.1. 1. Simítás/szűrés képtérben (átlagoló szűrők, Gauss simítás és mediánszűrés); élek detektálása (gradiens-operátorokkal és Marr-Hildreth módszerrel).
 - 9.1.1. Átlagoló szűrés
 - 9.1.2. Medián szűrés
 - 9.1.3. Gauss simítás
 - 9.1.4. Éldetektálás
 - 9.2. 2. Alakreprezentáció, határ- és régió-alapú alakleíró jellemzők, Fourier leírás.
 - 9.2.1. Határvonal alapú tulajdonságok
 - 9.2.2. Régió alapú alakleírás
- x. Programozási nyelvek
 - 10.1. A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai.
 - 10.1.1. Nyelvcsoportok (paradigmák)
 - 10.1.2. Imperatív programozás
 - 10.1.3. Dekleratív programozás
 - 10.1.4. Párhuzamos programozás
- xi. Programozás 1 & 2
 - 11.1. 1. Objektum orientált paradigmá, és annak megvalósítása a JAVA és C++ nyelvekben. Az absztrakt adattípus, az osztály. AZ egységbe zárás, az információ elrejtés, az öröklődés, az újrafelhasználás, és a polimorfizmus. A polimorfizmus feloldásának módszere.
 - 11.2. 2. Objektumok életciklusa, létrehozás, inicializálás, másolás, megszüntetés. Dinamikus, lokális, és statikus objektumok létrehozása. A statikus adattagok és metódusok, valamint szerepük a programozásban. Operáció, és operátor overloading JAVA és C++ nyelvekben. Kivételkezelés.
 - 11.3. 3. JAVA és C++ programok fordítása és futtatása. Parancssori paraméterek, fordítási opciók, nagyobb projektek fordítása. Absztrakt-, interfész-, és generikus osztályok, virtuális eljárások. A virtuális eljárások megvalósítása, szerepe, használata.
 - 11.3.1. Java program fordítása és futtatása
 - 11.3.2. C++ program fordítása és futtatása
 - 11.3.3. Java absztrakt-, interfész-, generikus osztályok
 - 11.3.4. C++ virtuális metódusok, absztrakt osztályok, templatek
- xii. Programozás alapjai
 - 12.1. 1. Algoritmusok vezérlési szerkezetei és megvalósításuk C programozási nyelven. A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés.
 - 12.1.1. Vezérlési módok
 - 12.1.2. Algoritmusok leírása
 - 12.1.3. Folyamatábra
 - 12.1.4. Szekvenciális vezérlés
 - 12.1.5. Szelekciós vezérlés

- 12.1.6. Eljárásvezérlés
- 12.1.7. Ismétléses vezérlés
- 12.2. 2. Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük, és műveleteik. Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord, és az unió típus. Az egyes típusok szerepe, használata.
 - 12.2.1. Egyszerű adattípusok
 - 12.2.2. Típusképzés
 - 12.2.3. Összetett adattípusok
- xiii. Rendszerfejlesztés 1.
 - 13.1. 1. Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei.
 - 13.1.1. A szoftverfejlesztés folyamata
 - 13.1.2. A folyamat modelljei
 - 13.2. 2. Projektmenedzsment. Költségbecslés, szoftvermérés
 - 13.2.1. Projektmenedzsment
 - 13.2.2. Szoftverköltség becslése
 - 13.2.3. Szoftvermérés, metrikák
- xiv. Számítógép architektúra
 - 14.1. 1. Neumann-elvű gép egységei. CPU, adatút, utasítás-végrehajtás, utasítás- és processzorszintű párhuzamosság. Korszerű számítógépek tervezési elvei. Példák RISC (UltraSPARC) és CISC (Pentium 4) architektúráakra, jellemzőik.
 - 14.1.1. Neumann-elvű gép sematikus váza
 - 14.1.2. Adatút
 - 14.1.3. Utasítás végrehajtás
 - 14.1.4. Utasításszintű párhuzamosság
 - 14.1.5. Processzorszintű párhuzamosság
 - 14.1.6. RISC és CISC
 - 14.1.7. Korszerű számítógépek tervezési elvei
 - 14.2. 2. Számítógép perifériák: Mágneses és optikai adattárolás alapelvei, működésük (merevlemez, Audio CD, CD-ROM, CD-R, CD-RW, DVD, Bluray). SCSI, RAID. Nyomtatók, egér, billentyűzet. Telekommunikációs berendezések (modem, ADSL, KábelTV-s internet).
 - 14.2.1. Mágneslemezek
 - 14.2.2. Optikai lemezek
 - 14.2.3. Kimenet/bemenet
 - 14.2.4. Telekommunikációs berendezések

Záróvizsga Tételek 2022

1. Algoritmusok és Adatszerkezetek I

1.1. 1. Részproblémára bontható algoritmusok (mohó, oszd-meg-és-uralkodj, dinamikus programozás), rendező algoritmusok, gráfalgoritmusok (szélességi- és mélységi keresés, minimális feszítőfák, legrövidebb utak)

1.1.1. Mohó algoritmusok

A feladatot pontosan egy részfeladatra bontják, és azt tovább rekurzívan oldják meg. Mindig a legjobbnak tűnő megoldás irányába haladunk tovább.

Nem minden problémára adható mohó megoldás!

De ha létezik, akkor nagyon hatékony!

Mohó választás: Az adott problémát egyetlen részproblémára bontja. Ennek optimális megoldásából következik az eredeti feladat optimális megoldása is.

Mohó algoritmus tervezése

1. Fogalmazzuk meg a **mohó választást**.
2. Bizonyítsuk be, hogy az eredeti problémának minden van olyan **optimális megoldása**, amely **tartalmazza a mohó választást**. Tehát hogy a mohó választás **biztonságos**.
3. Bizonyítsuk be, hogy a mohó választással olyan részprobléma keletkezik, amelynek egy **optimális megoldásához hozzávéve a mohó választást**, az eredeti probléma egy optimális megoldását kapjuk.

Példa: Töredékes háitzsák feladat

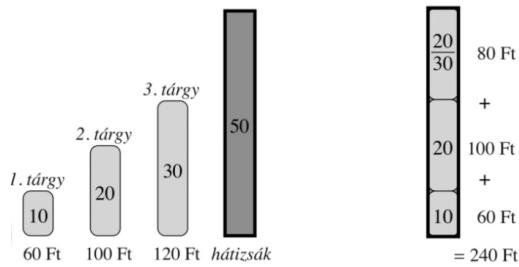
Bemenet: A háitzsák S kapacitása, n tárgy, S_i tárgy súlyok, E_i tárgy értékek

Kimenet: Mi a legnagyobb érték, ami S kapacitásba belefér?

Minden tárgyból 1db van, de az darabolható.

Algoritmus:

- Számoljuk ki minden tárgyra az $\frac{E_i}{S_i}$ arányt
- Tegyük bele a legnagyobb $\frac{E_i}{S_i}$ -vel rendelkező, még rendelkezésre álló tárgyból annyit a zsákba, amennyi belefér



Futás a fenti példán:

- Kiszámoljuk az $\frac{E_i}{S_i}$ értékeket
 - i. Tárgy: 6
 - ii. Tárgy: 5
 - iii. Tárgy: 4
- Végighaladunk a tárgyakon az $\frac{E_i}{S_i}$ arányok szerint
 - Az első tárgy teljes egészében belefér, azt beválasztjuk.
 - A 2. tárgy is teljes egészében belefér, azt is beválasztjuk.
 - A 3. tárgy már nem fér be, beválasztunk annyit, amennyi kitöltheti a szabad helyet. Jelen esetben a tárgy $\frac{2}{3}$ -át.

A probléma nem-törtéletes verziójára ez a mohó algoritmus nem mindig talál optimális megoldást.

1.1.2. Oszd-meg- és-uralkodj algoritmusok

A feladatot több részfeladatra bontjuk, ezek hasonlók az eredeti feladathoz, de méretük kisebb, tehát ugyan azt a feladatot akarjuk egy kisebb bemenetre megoldani.

Rekurzív módon megoldjuk ezeket a részfeladatokat (azaz ezeket is kisebb részfeladatokra bontjuk egészen addig, amíg elemi feladatokig jutunk, amelyekre a megoldás triviális), majd **összevonjuk őket**, hogy az eredeti feladatra megoldást adjanak.

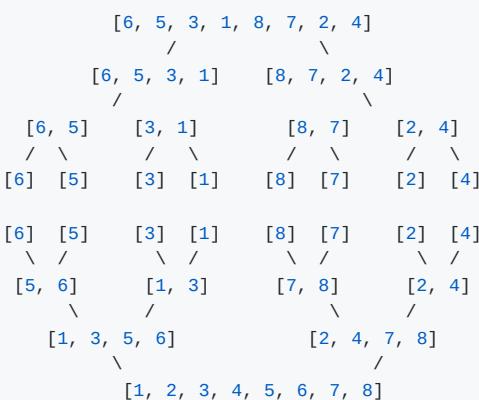
A részfeladatok ne legyenek átfedők. Bár az algoritmus ettől még működhet, de nem hatékony.

Lépések

- Felosztás:** Hogyan osztjuk fel a feladatot több kisebb részfeladatra.
- Uralkodás:** A feladatokat rekurzív módon megoldjuk. Ha a részfeladatok mérete elég kicsi, akkor közvetlenül meg tudjuk oldani a részfeladatot, ilyenkor nem osztjuk tovább rekurzívan.
- Összevonás:** A részfeladatok megoldásait összevonjuk az eredeti feladat megoldásává.

Példa: Összefésülő rendezés

- Felosztás:** Az n elemű rendezendő sorozatot felosztja két $\frac{n}{2}$ elemű részsorozatra.
- Uralkodás:** A két részsorozatra rekurzívan tovább hívjuk az összefésülő rendezés eljárását. Az elemi eset az egy elemű részsorozat, hiszen az már rendezett, ilyenkor csak visszatérünk vele.
- Összevonás:** Összefésüli a két rendezett részsorozatot, ezzel létrehozza az eredeti sorozat rendezett változatát.



Az összefésülés folyamata egyszerű, csak két mutatót vezetünk a két rendezett tömbön, lépkedünk, minden a kisebbet fűzzük egy másik, kezdetben üres tömbhöz.

Példa: Felező csúcskereső algoritmus

Vizsgáljuk meg a középső elemet. Ha csúcs, tértünk vissza vele, ha nem csúcs, akkor az egyik szomszédja nagyobb, vizsgáljuk tovább a bemenet felét ezen szomszéd irányába. Azért megyünk ebbe az irányba, mert erre biztosan van csúcs. Ezt onnan tudjuk, hogy maga ez a nagyobbik szomszéd is egy potenciális csúcs. Ha minden szomszédja nagyobb, akkor mindegy melyik irányba haladunk tovább, egyszerűen azzal, amiről előbb megtudtuk, hogy nagyobb.

1. **Felosztás:** n elemű sorozatot felosztjuk két $\frac{n-1}{2}$ elemű részsorozatra
2. **Uralkodás:** A megfelelő részsorozatban rekurzívan tovább keresünk csúcsot
3. **Összevonás:** Ha csúcsot találtunk, adjuk vissza

```
// Kiindulási tömb:  
[1, 3, 4, 3, 5, 1, 3]  
  
// Középső elemet megkeressük, nem csúcs, így tovább haladunk:  
[1, 3, 4, 3, 5, 1, 3]  
    ^  
// Középső elemet megkeressük, nem csúcs, így tovább haladunk:  
[1, 3, 4][3, 5, 1, 3]  
    ^  
// A középső elem egy csúcs, visszaadjuk  
[1, 3][4][3, 5, 1, 3]  
    ^
```

Ez az algoritmus logaritmikus időigényű. Ezzel szemben az egyszerű megoldás amikor minden elemen végighaladva keresünk csúcsot, lineáris, azaz jelentősen rosszabb.

1.1.3. Dinamikus programozás

Olyan feladatok esetén alkalmazzuk, amikor a **részproblémák nem függetlenek**, azaz vannak közös részproblémák.

Optimalizálási feladatok tipikusan ilyenek.

A megoldott **részproblémák eredményét memorizáljuk** (mondjuk egy táblázatban), így ha azok mégegyszer elő kerülnek, nem kell újra kiszámolni, csak elővenni memoriából az eredményt.

Iteratív megvalósítás

- minden részmegoldást kiszámolunk.
- Alulról-felfelé építkező megközelítés, hiszen előbb a kisebb részproblémákat oldjuk meg, amiknek az eredményét felhasználjuk az egyre nagyobb részproblémák megoldásához.

Rekurzív megvalósítás

- Részmegoldásokat kulcs-érték formájában tároljuk.
- Felülről lefelé építkező megközelítés.
- **Csak akkor használjuk, ha nem kell minden megoldást kiszámolni!**
 - Ha ki kell minden számolni, érdemesebb az iteratív megközelítést választani a függvényhívások overhead-je miatt.

Példa: Pénzváltás feladat

Adott P_i érmékkal (mindből van végtelen sok) hogyan lehet a legkevesebb érmét felhasználva kifizetni F forintot.

```
// Input:  
P1 = 1;  
P2 = 5;  
P3 = 6;  
F = 9;
```

Rekurzív megvalósítással a futás

```
// Egy dimenziós tömbbel dolgozunk, egyes sorokban  
// az egyes hívások állapota látszódik.  
// Első sor a pénzérme indexét jelöli.  
  
0 1 2 3 4 5 6 7 8 9  
0 - - - - - - - - ? // penzvalt(9) = min( penzvalt(3), penzvalt(4), penzvalt(8) ) + 1
```

```

0 - - ? - - - - ? // penzvalt(3) = min( penzvalt(2) ) + 1
0 - ? ? - - - - ? // penzvalt(2) = min( penzvalt(1) ) + 1
0 ? ? ? - - - - ? // penzvalt(1) = min( penzvalt(0) ) + 1
0 1 ? ? - - - - ? // penzvalt(0)-t ismertük már, kiindulástól kezdődően el volt mentve rá a triviális 0 megoldás, így pe
0 1 2 ? - - - - ? // penzvalt(1) visszatér, kiadja penzvalt(2) eredményét
0 1 2 3 - - - - ? // penzvalt(2) visszatér, kiadja penzvalt(3) eredményét
0 1 2 3 - - - - ? // penzvalt(3) visszatér

// penzvalt(9) jelenleg itt tart: min( 3, penzvalt(4), penzvalt(8) ) + 1
0 1 2 3 4 - - - - ? // penzvalt(4) = min( penzvalt(3) ) + 1

// penzvalt(9) jelenleg itt tart: min( 3, 4, penzvalt(8) ) + 1
0 1 2 3 4 - - - ? ? // penzvalt(8) = min( penzvalt(2) = 2, penzvaltas(3) = 3, penzvaltas(7) ) + 1
0 1 2 3 4 - - ? ? // penzvalt(7) = min( penzvalt(1) = 1, penzvaltas(2) = 2, penzvaltas(6) ) + 1
0 1 2 3 4 - ? ? ? // penzvalt(6) -> mivel ilyen érménk van, így ezt nem kell kiszámolni, tudjuk, hogy penzvalt(6) = 1
0 1 2 3 4 - 1 2 ? ? // penzvalt(6) visszatér, kiadja penzvalt(7)-et
0 1 2 3 4 - 1 2 3 ? // penzvalt(7) visszatér, kiadja penzvalt(8)-at
0 1 2 3 4 - 1 2 3 4 // penzvalt(8) visszatér, kiadja penzvalt(9)-et

```

Bár elmondható, hogy egy esetre, az 5-re nem kellett kiszámolnunk az értéket, de ez implementáció függő volt, ha `penzvalt(6)`-ot is ugyan úgy számoltuk volna, mint a többi értéket, akkor minden kiszámoltunk volna, és a rekurzív függvényhívások overhead-je miatt egyértelműen az iteratív megközelítés lenne a jobb.

Iteratív megvalósítással a futás

```

// 0-tól F-ig (9-ig) építünk egy egy dimentziós tömböt
0 1 2 3 4 5 6 7 8 9

0 ? ? ? ? ? ? ? ? ?
0 1 ? ? ? ? ? ? ? ? // penzvalt[1] = min( penzvalt[0] ) + 1
0 1 2 ? ? ? ? ? ? // penzvalt[2] = min( penzvalt[1] ) + 1
0 1 2 3 ? ? ? ? ? // penzvalt[3] = min( penzvalt[2] ) + 1
0 1 2 3 4 ? ? ? ? // penzvalt[4] = min( penzvalt[3] ) + 1
0 1 2 3 4 1 ? ? ? // penzvalt[5] = min( penzvalt[0], penzvalt[4] ) + 1
0 1 2 3 4 1 1 ? ? // penzvalt[6] = min( penzvalt[0], penzvalt[1], penzvalt[5] ) + 1
0 1 2 3 4 1 1 2 ? ? // penzvalt[7] = min( penzvalt[1], penzvalt[2], penzvalt[6] ) + 1
0 1 2 3 4 1 1 2 3 ? // penzvalt[8] = min( penzvalt[2], penzvalt[3], penzvalt[7] ) + 1
0 1 2 3 4 1 1 2 3 4 // penzvalt[9] = min( penzvalt[3], penzvalt[4], penzvalt[8] ) + 1

```

1.1.4. Rendező algoritmusok

Rendezés

- Input:** Egészek egy n hosszú tömbje (egy `<a1, a2, ..., an>` sorozat)
- Output:** n hosszú, rendezett tömb (az input sorozat egy olyan `<a'1, a'2, ..., a'n>` permutációja, ahol $a'1 \leq a'2 \leq \dots \leq a'n$)

Ez egy egyszerű eset, a gyakorlatban:

- Van valamelyen iterálható kollekciót: `Iterálható<Objektum>`
- Van egy függvényünk, ami megondja képt kollekció-elemről, hogy melyik a *nagyobb*: `(a: Objektum, b: Objektum) => -1 | 0 | 1`

Ezek együttesével már megfelelően absztrakt módon tudjuk használni az összehasonlító rendező algoritmusokat bármilyen esetben.

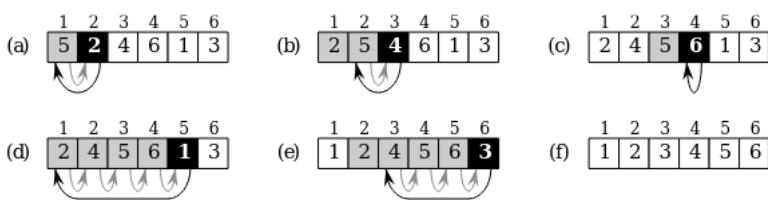
Beszúró rendezés

Helyben rendező módszer.

```

const beszuroRendezes = (A: number[]) => {
  for (let j = 1; j < A.length; j++) {
    const beillesztendo = A[j];
    let i = j - 1;
    for (; i >= 0 && A[i] > beillesztendo; i--) {
      A[i + 1] = A[i];
    }
    A[i + 1] = beillesztendo;
  }
  return A;
};

```



Végig haladunk a tömbön, és minden elemtől visszafelé elindulva megkeressük annak a helyét, és beszúrjuk oda. Amin áthaladtunk, az a részsorozat már rendezett lesz minden.

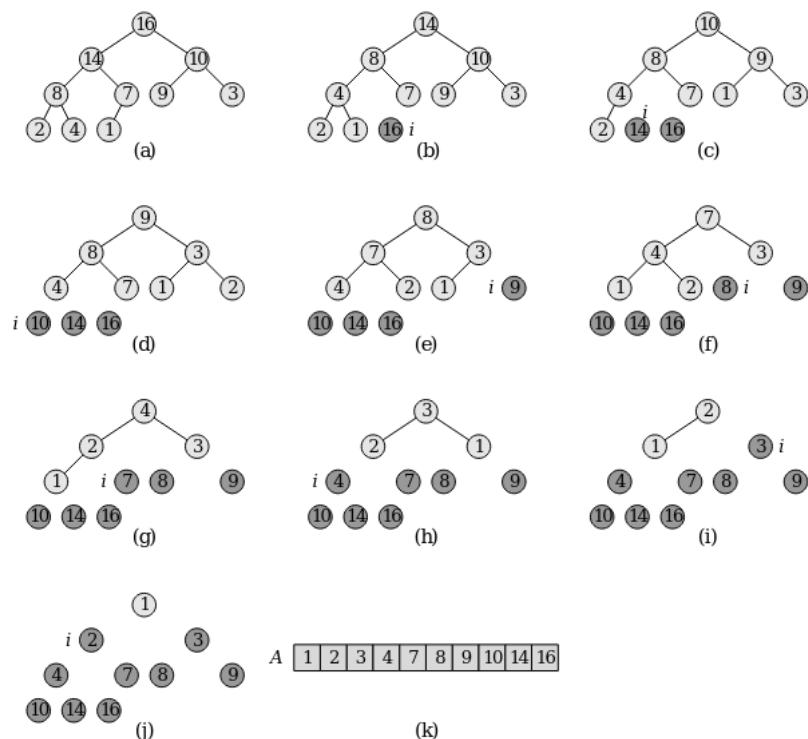
Futásidő	Tárigény (össz ~ inputon kívül)
$O(n^2)$	$O(n) \sim O(1)$

Legrosszabb eset: Teljesen fordítva rendezett tömb az input: [5, 4, 3, 2, 1]. Ekkor minden beillesztendo elemre vissza kell lépkedni a tömb elejéig.

Kupacrendezés

```
const kupacRendezes = (A: number[]) => {
    maximumKupacotEpit(A); // Helyben kupacosítja
    for (let i = A.length - 1, i >= 1; i--) {
        csere(A[1], A[i]);
        kupacMeret[A]--;
        maximumKupacol(A, 1);
    }
    return A;
}
```

Az input tömböt először **maximum-kupaccá** kell alakítani. Ekkor tudjuk, hogy a legnagyobb elem a gyökérben van, így ezt berakhatjuk az éppen vizsgált pozícióra (`csere(A[1], A[i])`). Ez után már csak csökkentenünk kell a kupac méterét, hiszen nem akarjuk mégegyszer a gyökérben az `A[i]`-t. Végezetül helyre kell állítanunk a kupac-tulajdonságot egy `maximumKupacol(A, 1)` hívással. (A 2. paraméter azt mondja meg, melyik csúcsból lefelé szeretnénk helyreállítani, jelen esetben az 1-es, hiszen pont azt a pozíciót rontottuk el, amikor cserélünk. Tehát az egész kupacot helyreállítjuk.)



Futásidő	Tárigény (össz ~ inputon kívül)
$O(n * \log(n))$	$O(n) \sim O(1)$

Gyorsrendezés

Összefűsölő rendezéshez hasonlóan oszd-meg-és-uralkodj algoritmus

- Felosztás:** Az `A[p..r]` tömböt, két (esetleg üres) `A[p..q-1]` és `A[q+1..r]` résztömbre osztjuk, hogy az `A[p..q-1]` minden eleme kisebb, vagy egyenlő `A[q]`-nál, és `A[q]` kisebb vagy egyelő `A[q+1..r]` minden eleménél. A `q` index kiszámítása része ennek a felosztó eljárásnak.
- Uralkodás:** Az `A[p..q-1]` és `A[q+1..r]` résztömböket a gyorsrendezés rekurzív hívásával rendezzük.

- **Összevonás:** Mivel a két résztömböt helyben rendeztük, nincs szükség egyesítésre: az egész $A[p..r]$ tömb rendezett.

```
const feloszt = (A: number[], p: number, r: number) => {
  const x = A[r];
  let i = p - 1;
  for (let j = p; j <= r - 1; j++) {
    if (A[j] <= x) {
      i++;
      [A[i], A[j]] = [A[j], A[i]];
    }
  }
  [A[r], A[i + 1]] = [A[i + 1], A[r]];
  return i + 1;
};
```

```
const _gyorsRendezes = (A: number[], p: number, r: number) => {
  if (p < r) {
    const q = feloszt(A, p, r);
    _gyorsRendezes(A, p, q - 1);
    _gyorsRendezes(A, q + 1, r);
  }
  return A;
};

const gyorsRendezes = (A: number[]) => _gyorsRendezes(A, 0, A.length - 1);
```

Futásidő	Tárigény
$O(n^2)$	$O(n)$

Fontos, hogy az eljárás teljesítménye függ attól, hogy a felosztások mennyire ideálisak. Valószínűségi alapon a vátható rekurziós mélység $O(\log n)$, ami mivel egy hívás futásidője $O(n)$, így az átlagos futásidő $O(n * \log n)$. A gyakorlat azt mutatja, hogy ez az algoritmus jó teljesít.

Lehet úgy implementálni, hogy $O(\log n)$ tárigénye legyen, ez egy helyben rendező, farok-rekurzív ejlárás.

Összehasonlító rendezések teljesítményének alsó korlátja

Minden összehasonlító rendező algoritmus legrosszabb esetben $\Omega(n * \log n)$ összehasonlítást végez.

Ez alapján pl. az összefésűlő, vagy a kupac rendezés **aszimptotikusan optimális**.

Eddigi algoritmusok mind összehasonlító rendezések voltak, a kövezkező már nem az.

Ezt döntési fával lehet bebizonyítani, aminek belső csúcsai meghatároznak két tömbelemet, amiket épp összehasonlítunk, a levelek pedig hogy az oda vezető összehasonlítások milyen sorrendhez vezettek. Nem konkrét inputra írható fel döntési fa, hanem az algoritmushoz. Így ennek a fának a legrosszabb esetben vett magassága lesz az algoritmus futásidéjének felső korlátja.

Leszámoló rendezés

Feltételezzük, hogy az összes bemeneti elem 0 és k közé esik.

Minden lehetséges bemeneti elemhez megszámoljuk, hányszor fordul elő az inputban.

Majd ez alapján azt, hogy hány nála kisebb van.

Ez alapján már tudjuk, hogy az egyes elemeknek hova kell kerülni. Mert ha pl 5 elem van, ami kisebb, vagy egyenlő, mint 2, akkor tudjuk, hogy az 5. pozíció 2-es kell, hogy legyen.

```
const leszamoloRendezes = (A: number[], k: number) => {
  const C = [...new Array(k + 1)].map(() => 0);
  A.forEach((szam) => {
    C[szam]++;
  });
  // Itt a C-ben azon elemek száma van, aminek értéke i
  for (let i = 1; i < C.length; i++) {
    C[i] += C[i - 1];
  }
  // Itt C-ben i indexen azon elemek száma van, amik értéke kisebb, vagy egyenlő, mint i
  const B = [...new Array(A.length)]; // B egy A-val egyező hosszú tömb
  for (let i = A.length - 1; i >= 0; i--) {
    B[C[A[i]] - 1] = A[i];
    C[A[i]]--;
  }
};
```

```

    return B;
}

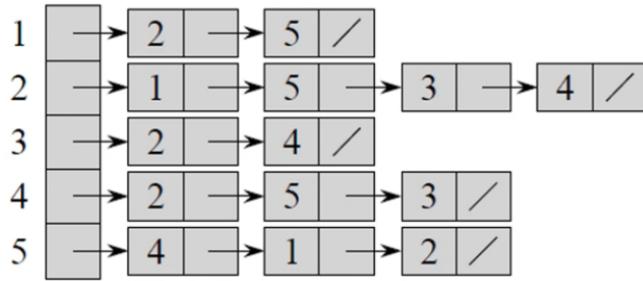
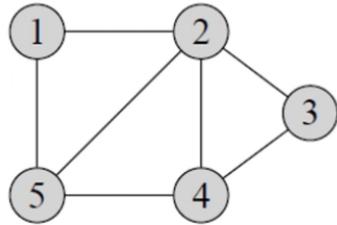
```

Futásidő	Tárigény
$\Theta(k + n)$	$\Theta(2n)$

A gyakorlatban akkor használjuk, ha $k = O(n)$, mert ekkor a futásidő $\Theta(n)$

1.1.5. Gráfalgoritmusok

Gráfok ábrázolása: **éllista** vagy **szomszédsági mátrix**



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

Szélességi keresés

Gráf bejárására szolgál.

A bejárás során kijelöl egy "szélességi fát", ami egy kiindulási csúcsból indulva minden az adott csúcsból elérhető csúcsokat reprezentálja.

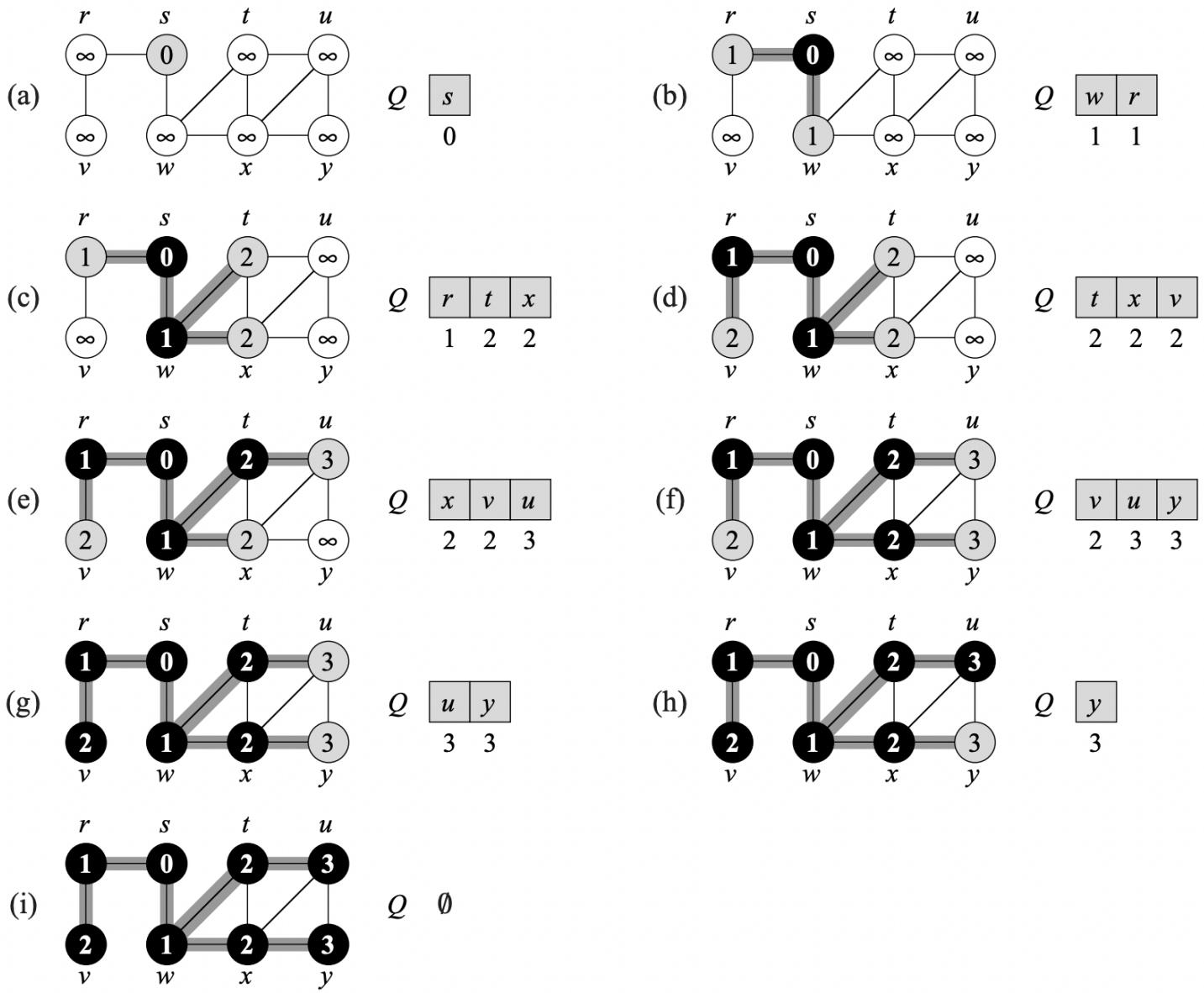
Amilyen távol van a kiindulási csúcstól egy csúcs, az olyan mélységen helyezkedik el ebben a fában.

Irányított, irányítatlan gráfok esetén is alkalmazható.

A csúcsok távolsága alapján halad a bejárás (a kijelölt kezdeti csúcstól), minden k távolságra levő csúcsot elérünk az előtt, hogy egy $k + 1$ távolságra levőt elérnének.

Az algoritmus színezi a csúcsokat, ezek a színek a következőket jelentik:

- fehér**: Kiindulási szín, egy ilyen színű csúcsot még nem értünk el.
- szürke**: Elért csúcs, de még van fehér szomszédja.
- fekete**: Elért csúcs, és már minden szomszédja is elért (vagy szürke vagy fekete).



```
// A G a gráf, s a kiindulási csúcs
szelessegKereses(G, s) {
    for G gráf minden nem s csúcsára {
        szín[csucs] = "fehér"
    }
    szín[s] = "szürke"
    d[s] = 0 // Távolság s-től
    szülő[s] = null
    Q = [] // Üres SOR
    sorba(Q, s)
    while Q nem üres {
        u = sorból(Q)
        for u minden v szomszédjára {
            if (szín[v] === "fehér") {
                szín[v] = "szürke"
                d[v] = d[u] + 1
                szülő[v] = u
                sorba(Q, v) // Tovább feldolgozzuk majd neki a szomszédjait
            }
        }
        szín[u] = "fekete" // Itt már végigmentünk minden szomszédján
    }
}
```

Futásidő

- Minden csúcst egyszer érintünk csak, ez V db csúcs.
- Sorba, és sorból $O(1)$, így a sorműveletek összesen $O(V)$.
- Szomszédsági listákat legfeljebb egyszer vizsgáljuk meg, ezek össz hossza $\Theta(E)$, így összesen $O(E)$ időt fordítunk a szomszédsági listák vizsgálására.
- Az algoritmus elején a kezdeti értékadások ideje $O(V)$.

- Összesített futásidő: $O(E + V)$

Mélységi keresés

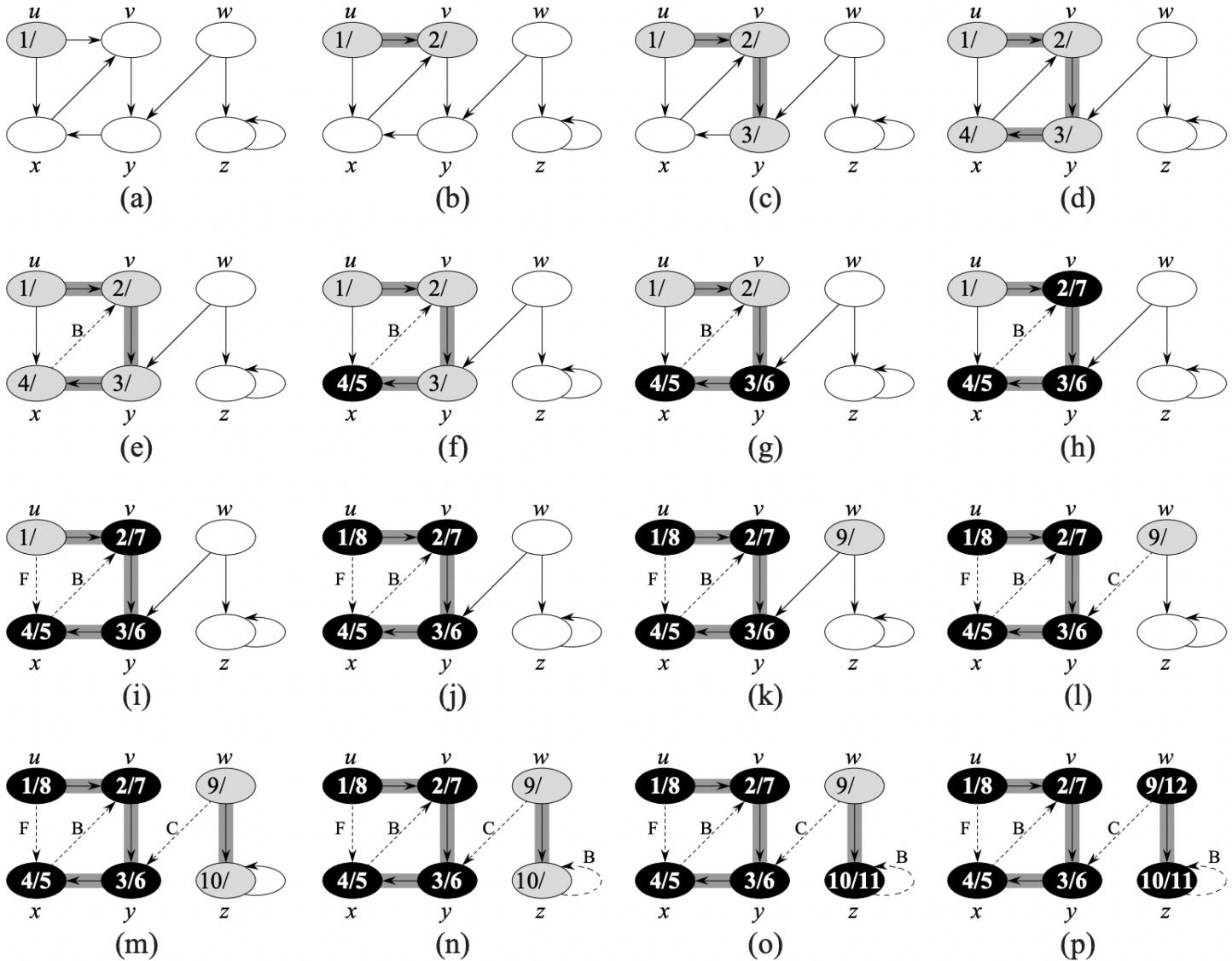
Addig megy a kivezető élek mentén, ameddig tud, majd visszafele indulva minden érintett csúcs kivezető élein addig megy mélyre, amíg lehet.

Ugyan azokat a színekez használja a csúcsok színezésére, mint a szélességi keresés.

Minden csúcshoz feljegyzi, hogy mikor (hány lépés után) érte el, és hagyta el azt.

```
melysegiKereses(G) {
    for G minden u csúcsára {
        szín[u] = "fehér"
        szülő[u] = null
    }
    idő = 0
    for G minden u csúcsára {
        if (szín[u] === "fehér") {
            melysegiBejaras(u)
        }
    }
}

melysegiBejaras(u) {
    szín[u] = "szürke"
    idő++
    d[u] = idő // Ekkor értük el
    for u minden v szomszédjára {
        if (szín[v] === "fehér") {
            szülő[v] = u
            melysegiBejaras(v) // Azonnal már indulunk is el a talált csúcsból
        }
    }
    szín[u] = "fehete"
    idő++
    f[u] = idő // Ekkor hagytuk el
}
```



Futásidő

A `melysegiKereses()` futásideje a `melysegiBejaras()` hívástól eltekintve $\Theta(V)$. A `melysegiBejaras()` hívások össz futásideje $\Theta(E)$, mert ennyi a szomszédsági listák összesített hossza. Így a futásidő $O(E + V)$

A futásidő azért lesz additív mingkét esetben, mert a szomszédsági listák össz hosszára tudjuk mondani, hogy $\Theta(E)$. Lehet, hogy ezt egyszerre nézzük végig, lehet, hogy eloszlata, de összesen ennyi szomszédot vizsgál meg például a `melysegiBejárás()`.

Minimális feszítőfák

Cél: megtalálni éleknek azon **körmentes** részhalmazát, amely élek mentén **minden csúcs összeköthető**, és az élek **összesített súlya legyen a lehető legkisebb**.

Az így kiválasztott élek egy fát alkotnak, ez a **feszítőfa**.

Két mohó algoritmus: **Prim**, **Kruskal**

Kruskal

A gráf csúcsait diszjunkt halmazokba sorolja. Kezdetben minden csúcs 1-1 egy elemű halmaz.

Erre van speciális diszjunkt-halmaz adatszerkezet

Minden iterációban beveszi a legkisebb súlyú élet, aminek végpontjai különböző halmazokban vannak.

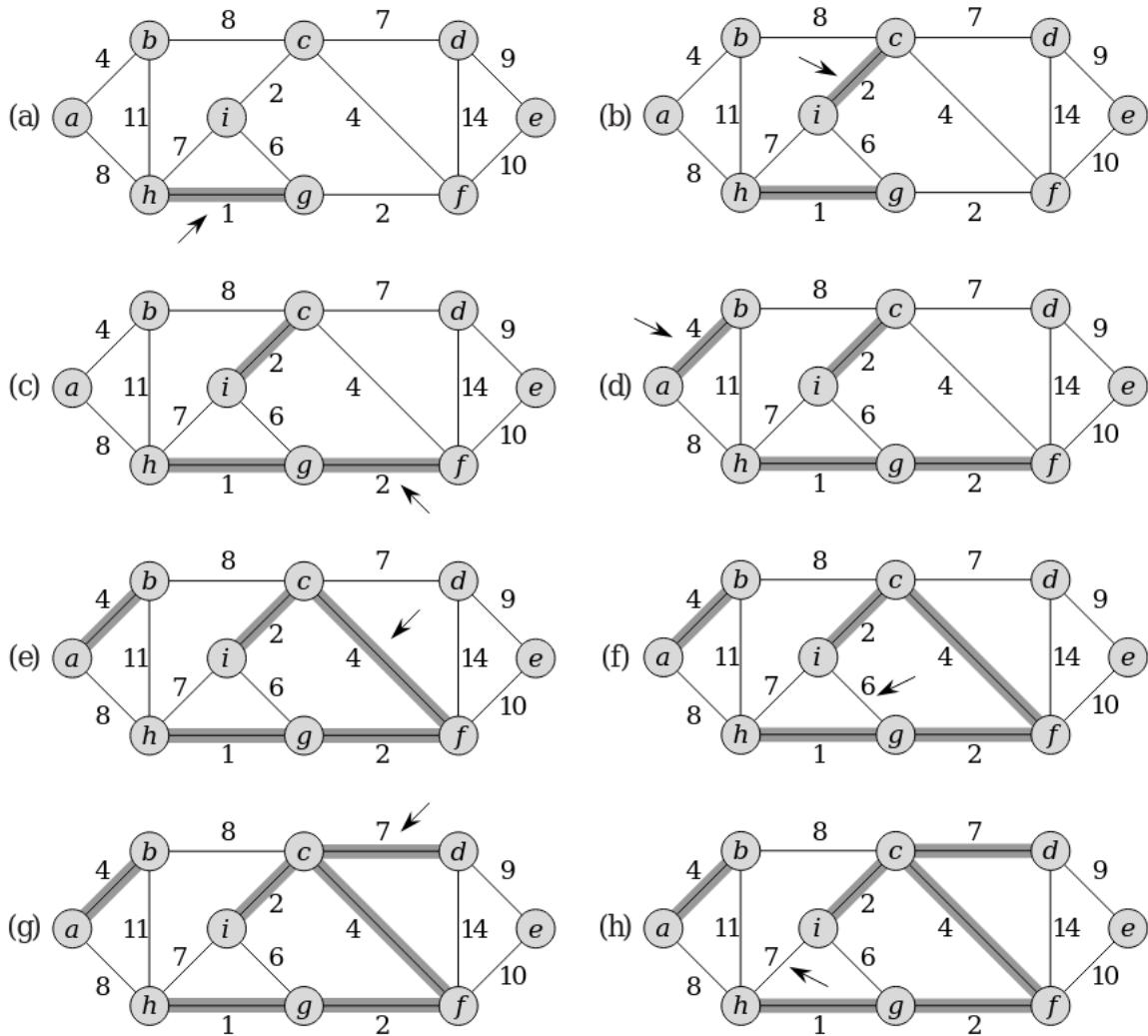
Ez által egy erdőt kezel, mit a végére egy fává alakít. Ez lesz a feszítőfa.

```
kruskal(G, w) { // Az élsúlyokat megadó függvény
    A = 0
    for minden v csúcsra {
        halmaztKeszit(v)
    }
    for minden (u, v) érre, az élsúlyok szerin növekvő sorrendben {
        if halmaztKeres(u) != halmaztKeres(v) {
            A = A unió { (u, v) }
            egyesít(u, v)
        }
    }
}
```

```

    }
}
}
```

`halmaztKeszit`, `halmaztKeres` és `egyesít` a diszjunkt halmazokat kezelő függvények.



Futásidő

Az élek rendezése $O(E * \log E)$.

A halmaz műveletek a kezdeti értékadásokkal együtt $O((V + E) * \alpha * (V))$. Ahol az α egy nagyon lassan növekvő függvény, a diszjunkt-halmaz adatszerkezet sajátossága. Mivel összefüggő gráf esetén $O(|E| \geq |V| + 1)$, így a diszjunkt-halmaz műveletek $O((E) * \alpha * (V))$ idejűek. $\alpha(|V|) = O(\log E)$ miatt $O(E * \log E)$.

Így a teljes futásidő $O(E * \log E)$.

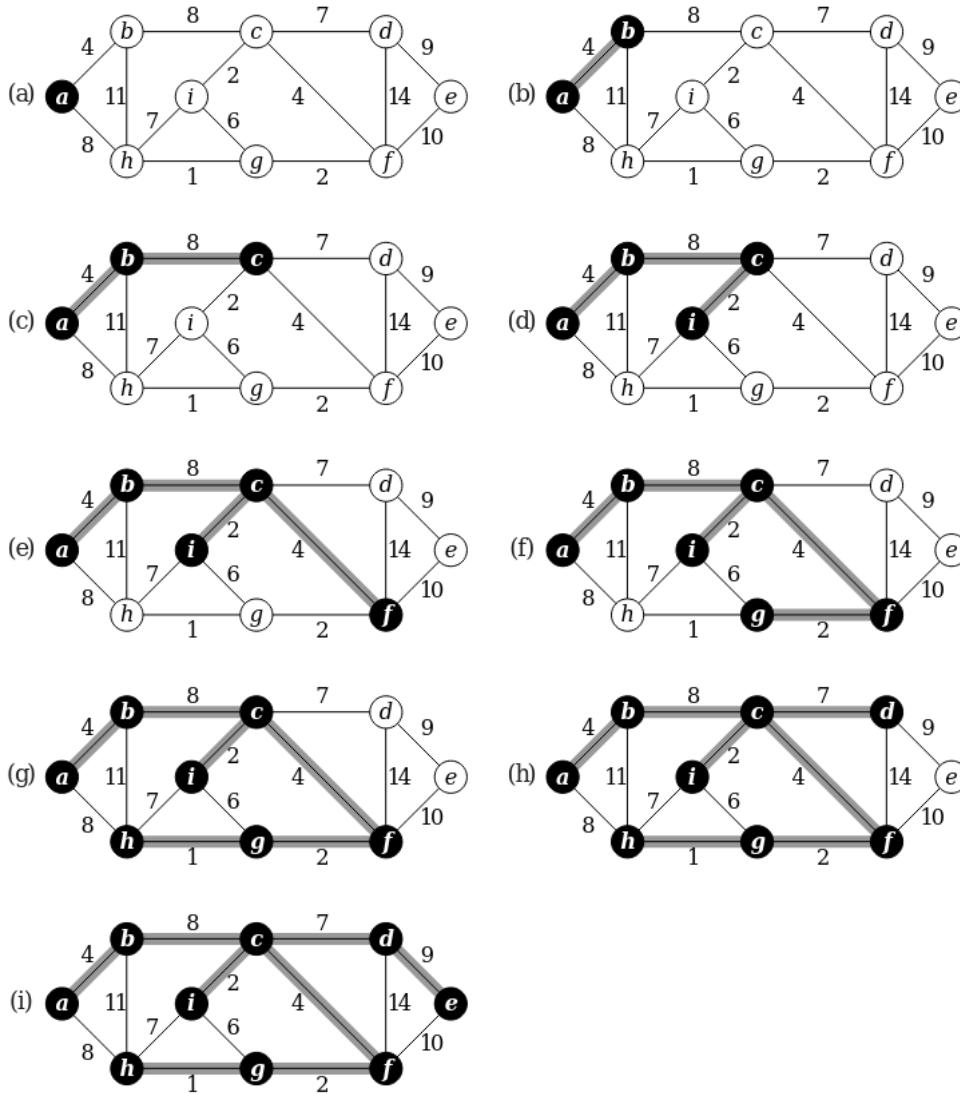
Prim algoritmus

A Kruskallal ellentétben folyamatosan egy darab fát kezel, ezt növeli az iterációkban.

Egy megadott kiindulási csúcsból indulva minden iterációban hozzávesszük azt a csúcst, amit a legkisebb súlyú él köt a meglévő fához.

```

prim(G, w, r) { // Az élsúlyokat megadó függvény
    for minden v csúcsra {
        kulcs[v] = Végtelen
        szülő[v] = null
    }
    kulcs[r] = 0
    Q = G csúcsai // Prioritási sor kulcs[] szerint minimális
    while Q nem üres {
        u = kiveszMin(Q)
        for u minden v szomszédjára {
            if v eleme Q, és w(u, v) < kulcs[v] {
                szülő[v] = u
                kulcs[v] = w(u, v)
            }
        }
    }
}
```



Futásidő

Bináris minimum kupac megvalósítással:

Kezdeti értékkedások: $O(V)$

Egy db kiveszMin művelet: $O(\log V)$. Összesen: $O(V * \log V)$, mivel V -szer fut le a ciklus.

Belső for ciklus $O(E)$ -szer fut, mivel szomszédsági listák hosszainak összege: $O(2|E|)$. (Ez megintcsak additív, nem kell a külső ciklussal felszorozni, mert a szomszédsági listák alapján tudjuk, hogy ennyiszer fog maximum összesen lefutni.) Ezen a cikluson belül a Q -hoz tartozás vizsgálata konstans idejű, ha erre fenntartunk egy jelölő bitet. A kulcsotCsökkent művelet, ami $O(\log V)$ idejű.

Agy tehát az összesített futásidő: $O(V\log V + E\log V) = O(E\log V)$.

Fibonacci-kupaccal gyorsítható az algoritmus, ekkor a kiveszMin $O(\log V)$ -s, kulcsotCsökkent $O(1)$ -es, teljes futásidő: $O(E + V * \log V)$

Legrövidebb utak

Lehetséges problémák:

- Adott csúcsból induló legrövidebb utak problémája:** Egy adott kezdőcsúcsból meg szeretnénk találni minden másik csúcshoz vezető legrövidebb utat.
- Adott csúcsba érkező legrövidebb utak problémája:** minden csúcsból egy adott csúcsba. Ugyan az, mint az előbbi, ha az élek irányát megfordítjuk.
- Adott csúcspár közti legrövidebb út problémája:** Ha az elsőt megoldjuk, ezt is megoldottuk. Nem ismert olyan algoritmus, ami aszimptotikusan gyorsabban megoldaná ezt a feladatot, de az első nem.
- Összes csúcspár közti legrövidebb utak problémája:** Ez persze megoldható lenne az elsővel, ha minden csúcsból elindítjuk, de ennél léteznek gyorsabb megoldások.

Optimális részstruktúra: azt jelenti jelen esetben, hogy két csúcs közti legrövidebb út magában foglalja sokszor másik két csúcs közti legrövidebb utat. Az algoritmusok ezt használják ki.

Negatív súlyú élek: lehetnek, de a gráf nem tartalmazhat negatív összsúlyú kört. Ugyanis ekkor nem definiált a legrövidebb út, hiszen a körön megegyező végig haladva minden kisebb súlyú utat kapunk.

Kör a legrövidebb útban: Negatív összsúlyú tehat nem lehet, mert ekkor maga a feladat nem definiált. **Pozitív összsúlyú sem lehet**, hiszen ekkor jobban járnánk, ha nem járnánk be a kört. **Nulla összsúlyúnak pedig nincsen értelme**, hogy szerepeljen legrövidebb útban, hiszen ekkor ugyan annyi az összsúly a kör megtétele nélkül is. Tehát általánosságban feltételezhetjük, hogy a **legrövidebb út nem tartalmaz kört**.

Két függvény, amit használni fognak az algoritmusok:

```

egyForrasKezdoertek(G, s) { // Kezdőértékek beállítása, ha egy csúsból indul
    for minden v csúcsra {
        d[v] = Végtelen
        szülő[v] = null
    }
    d[s] = 0
}

közelít(u, v, w) { // (u, v) él alapján v távolságának frissítése (ha u-ból jöve kisebb, akkor csökkentjük)
    if d[v] > d[u] + w(u, v) {
        d[v] = d[u] + w(u, v) // A d[v] becslést csökkenti
        szülő[v] = u
    }
}

```

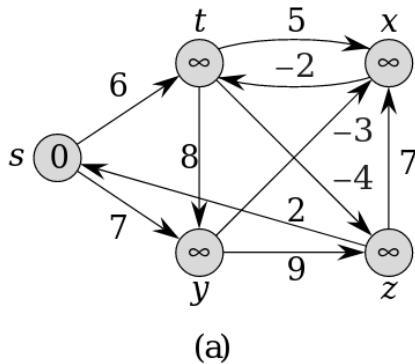
Bellman-Ford algoritmus

Lehetnek negatív élek, ha van negatív összsúlyú él, azt felismeri az algoritmus, jelzi azzal, hogy hamissal tér vissza.

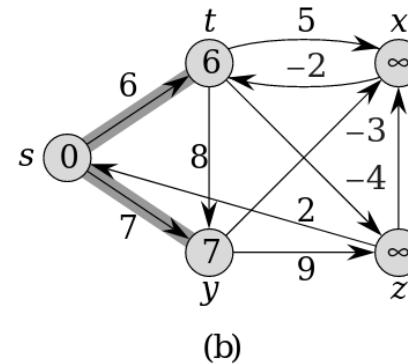
```

bellmanFord(G, w, s) {
    egyForrasKezdoertek(G, s)
    for i = 1 to |V[G]| - 1 {
        for minden (u, v) élre {
            közelít(u, v, w)
        }
    }
    for minden (u, v) élre { // Itt ellenőrzi, hogy volt-e negatív kör
        if d[v] > d[u] + w(u, v) {
            return false
        }
    }
    return true
}

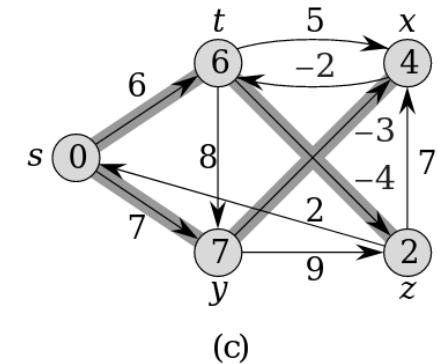
```



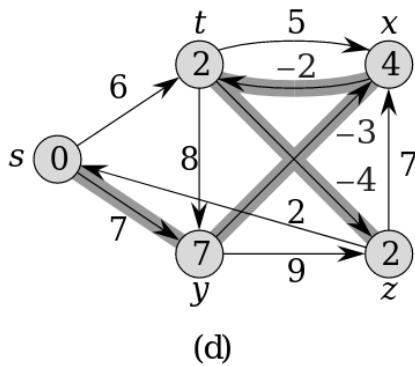
(a)



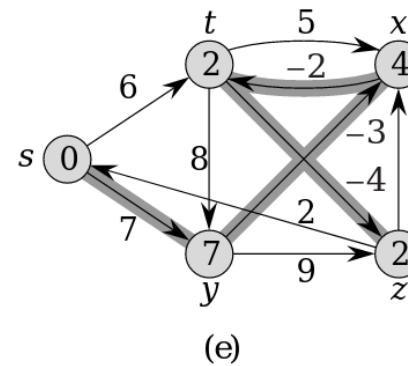
(b)



(c)



(d)



(e)

Futásidő

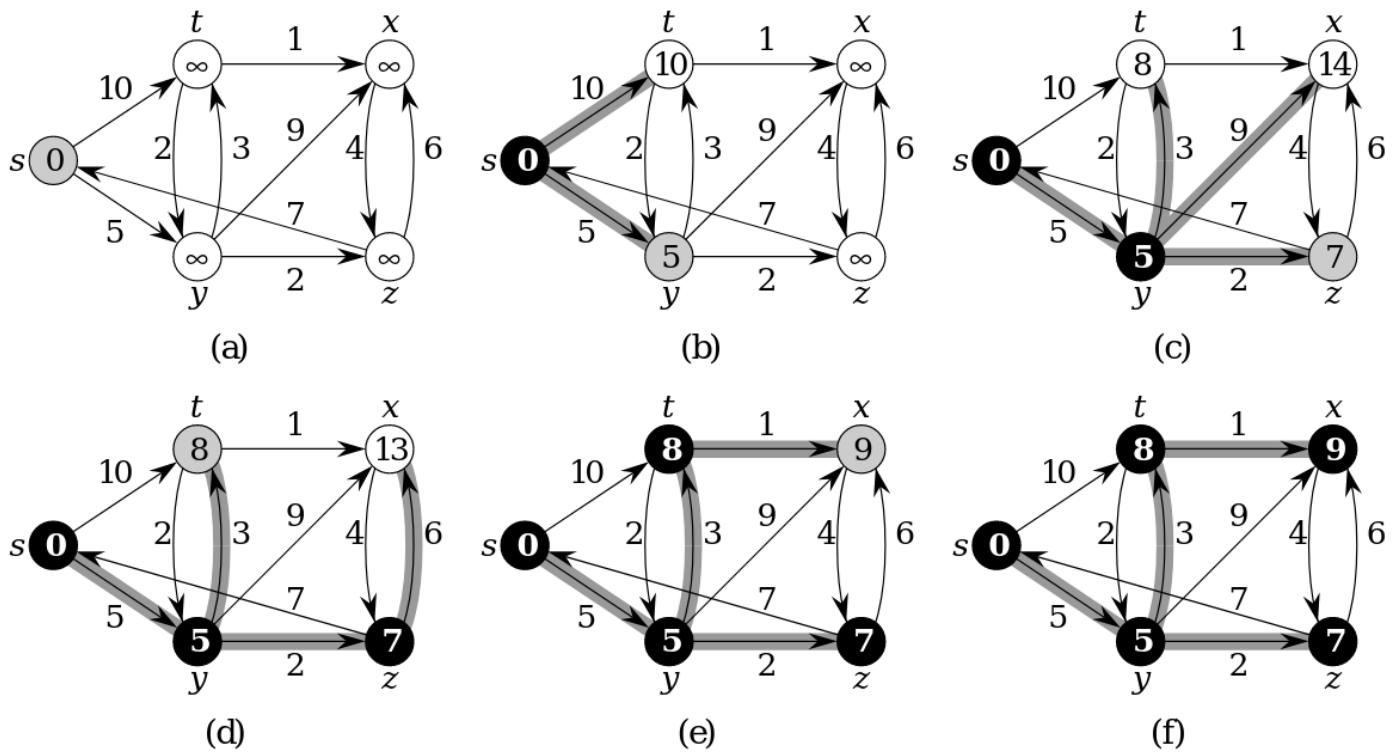
$O(V * E)$ hiszen a kezdőértékek beállítása $\Theta(V)$, az egymásba ágyazott for ciklus $O(V * E)$, a második ciklus pedig $O(E)$.

Nemnegatív élsúlyok esetén működik.

S halmaz: Azon csúcsok kerülnek bele, amikhez már meghatározta a legrövidebb utat a kezdőcsúsból.

```
dijkstra(G, s) {
    egyForrasKezdoertek(G, s)
    S = üresHalmaz
    Q = V[G] // Q minimum prioritási sor
    while Q nem üres {
        u = kiveszMin(Q)
        S = S unió { u }
        for u minden v szomszadjára {
            közelít(u, v, w)
        }
    }
}
```

A Q sorban azok a csúcsok vannak, amik nincsenek S-ben, tehát még nem tudjuk a hozzájuk vezető legrövidebb utat. A sort a d érték szerint azaz az ismert legrövidebb út szerint indexeljük.



Futásidő

Minden csúcs pontosan egyszer kerül át az S halmazba, emiatt amikor szomszédokat vizsgálunk, azt minden csúcsra egyszer tesszük meg, ezen szomszédok vizsgálata összesen $O(E)$ -szer fut le, mert ennyi a szomszédsági listák össz hossza. Így a közelít, és ez által a `kulcsotCsökkent` művelet legfeljebb $O(E)$ -szer hívódik meg.

Az összesített futásidő nagyban függ a **prioritási sor implementációtól**, a legegyszerűbb eset, ha egy **tömbbel implementáljuk**. Ekkor a `beszür` és `kulcsotCsökkent` műveletek $O(1)$ -esek, a `kiveszMin` pedig $O(V)$, mivel az egész tömbön végig kell menni. Így a teljes futásidő $O(V^2 + E)$. **Ritkább gráfok esetén gyorsítható** az algoritmus **bináris kupac** implementációval, és általanosságban gyorsítható fibonacci kupaccal.

Floyd-Warshall algoritmus

Dinamikus programozási algoritmus legrövidebb utak **minden csúcspárra** problémára.

Lehetnek negatív élsúlyok, de negatív összsúlyú körök nem.

Az algoritmus lényege, hogy dinamikus programozással haladunk, egyre több csúcsot használunk fel, és azt figyeljük, hogy a két csúcs között vezető úton jobb eredményt érnénk el, ha az adott iteráció csúcsán keresztül mennénk.

Ez a következő rekurziós képlettel írható fel:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{ha } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{ha } k \geq 1. \end{cases}$$

```
floydWarshall(W) { // W szomszédsági mártix
    n = sorokSzama(W)
    d(0) = W
    for k = 1-től n-ig { // Ezt vizsgáljuk mindenkor minden csúcs
        for i = 1-től n-ig {
            for j = 1-től n-ig {
                d(k)[i, j] = min(
                    d(k - 1)[i, j],
                    d(k - 1)[i, k] + d(k - 1)[k, j]
                )
            }
        }
    }
}
```

A belső értékadás magyarázata: A k . iterációban a legrövidebb út, ami i -ből j -be vezet, az vagy a már megtalált $k - 1$ -edik iterációbeli eredmény, vagy a az előző iterációbeli út i -ből k -ba, plusz k -ból j -be, azaz **felhasználjuk-e a k -t, mint egy köztesen érintett csúcson**.

Futásidő

A három for ciklus határozza meg, mert annak a magja $O(1)$ -es, így a futásidő $\Theta(n^3)$, ahol n a sorok száma.

1.2. 2. Elemi adatszerkezetek, bináris keresőfák, hasító táblázatok, gráfok és fák számítógépes reprezentációja

Az **adatszerkezet** adatok tárolására, és szervezésére szolgáló módszer, amely lehetővé teszi a hatékony hozzáférést és módosítést.

Algoritmushoz válasszuk ki az adatszerkezetet. Előfordulhat, hogy az algoritmus a megfelelő adatszerkezeten alapul.

Absztrakt adatszerkezet: műveletek által definiált adaszerkezet, nem konkrét implementáció.

Adatszerkezetek: Absztrakt adatszerkezetek konkrét megvalósításai. Általában egyes implementációk egyes műveleteket gyorsabban, míg másokat lassabban tudnak végrehajtani. Ez alapján kell az algoritmushoz kiválasztani a megfelelőt.

Absztrakt adatszerkezetek olyanok, mint **interfészek**, az adatszerkezetek pedig azt implementáló **osztályok**.

1.2.1. Listák

Absztrakt adatszerkezet.

Benne az adatok lineárisan követik egymást, egy kulcs többször is előfordulhat benne.

Művelet	Magyarázat
ÉRTÉK(H, i)	i . pozíción (index-en) a kulcs értékének visszaadása
ÉRTÉKAD(H, i, k)	i . pozíción levő értéknek a k érték értéküladása
KERES(H, k)	A k kulcs (érték) megkeresése a listában, indexének visszaadása
BESZÜR(H, k, i)	Az i -edik pozíctó után a k beszúrása
TÖRÖL(H, k)	Első k értékű elem törlése

Közvetlen elérésű lista

Összefüggő memóriaterületet foglalunk le, így minden index közvetlen elérésű.

Művelet	Futásidő
ÉRTÉK(H, i)	$O(1)$
ÉRTÉKAD(H, i, k)	$O(1)$
KERES(H, k)	$O(n)$
BESZÜR(H, k, i)	$O(n)$
TÖRÖL(H, k)	$O(n)$

Beszúrásnál újra kellhet allokálni egyel nagyobb emmóniaterületet.

Jellemzően úgy implementáljuk, hogy definiálunk egy **kapacitást**, és amikor kell, akkor ennyivel allokálunk többet az új memóriaterületen. Illetve jellemzően azt is definiáljuk, hogy mikor kell zsugorítani a területet, azaz hány üresen maradó cella esetén (nem lyukak!) az nem lehet, csak a terület végén levő üres cellák) allokálunk kevesebb területet.

Előnye: $O(1)$ -es indexelés.

Hártánya: Módosító műveletek lassúak, egy nagy memóriablokk kell.

Láncolt lista

Minden kulcs mellett tárolunk egy mutatót a következő, és egy mutatót a megelőző elemre.

Egyszeresen láncolt lista: csak a következőre tárolunk mutatót.

Kétszeresen láncolt lista: következőre, előzőre is tárolunk mutatót.

Ciklikus lista: Utolsó elem rákövetkezője az első elem, első megelőzője az uolsó elem.

Őrszem / fej: Egy NULL elem, ami minden a lista eleje.

Művelet	Futáridő
ÉRTÉK(H, i)	$O(n)$
ÉRTÉKAD(H, i, k)	$O(n)$
KERES(H, k)	$O(n)$
BESZÜR(H, k, i)	$O(1)$
TÖRÖL(H, k)	$O(1)$

Beszúrás, és törlés valójában $O(n)$. Csak akkor $O(1)$, ha már a megfelelő pozícióban vagyunk, azaz már tudjuk, melyik mutatókat kell átírní.

Előnye: Nem egy nagy összefüggő memória blokk kell.

Hártánya: Nem lehet gyorsan indexelni. Tárigény szempontjából rosszabb, minden kulcs mellett tárolunk legalább egy mutatót.

1.2.2. Verem

Lista, amiben csak a legutoljára beszúrt elemet lehet kivenni. (**LIFO**)

Emiatt a speciális művelet végzés miatt gyorsabb, mint a sima lista.

Alkalmazásokra pl.: Függvényhívások veremben, undo-redo, böngésző előzmények.

Verem megvalósítás fix méretű tömbbel

Fenntartunk egy mutatót a verem tetejére, eddig van feltöltve a lefoglalt memóriaterület. (A verem alja a 0. index.)

```
üresVerem(V) {
    return tető[V] == 0
}
```

```
verembe(V, x) {
    tető[V]++ // Tető mutató frissítése, hiszen egyel több elem lesz
    V[tető[V]] = x
}
```

```
veremből(V) {
    if üresVerem(V) {
        throw Error("alulcsordulás")
    } else {
        tető[V]--
        return V[tető[V] + 1] // Ez az index nincs felszabadítva, vagy átírva, egyszerűen a mutató van csökkentve
    }
}
```

Mind a 3 művelet $O(1)$ -es, hiszen csak indexeléseket, értékadásokat tartalmaznak.

Hasonlóan a tömbbel megvalósított listához, itt is érdemes lehet kapacitást meghatározni.

1.2.3. Sor

Mindig a legelőször beszűrt elemet lehet kivenni. (**FIFO**)

Lefoglalunk egy valamekkora egybefüggő memória szegmenst, de nem minden használjuk az egészét. Két mutatót tartunk fent, a `fej` és a `vége` mutatókat, ezek jelölik, hogy éppen mekkora részét használjuk a lefoglalt területnek sorként.

```
sorba(S, x) {
    S[vége[S]] = x // A vége egy üres pozícióra mutat alapból, ezért növeljük utólag.
    if vége[S] == hossz[S] {
        vége[S] = 1 // Ekkor "körbefordult" a sor a lefoglalt memóriaterületen.
    } else {
        vége[S]++
    }
}
```

```
sorból(S) {
    x = S[fej[S]] // A fej mutat a sor "elejére", azaz a legrégebben betett elemre.
    if fej[S] == hossz[S] {
        fej[S] = 1 // Ekkor "körbefordult" a sor a lefoglalt memóriaterületen.
    } else {
        fej[S]++
    }
    return x
}
```

Mind a két művelet $O(1)$ -es, hiszen csak indexeléseket, értékadásokat tartalmaznak.

1.2.4. Prioritási sor

Absztrakt adatszerkezet.

Nem a kulcsok beszúrásának sorrendje határozza meg, mit lehet kivenni, hanem minden minden prioritási sorban minden maximális (minimális) kulcsú elemet tudjuk kivenni.

Művelet	Magyarázat
BESZÜR(H, k)	Új elem beszúrása a prioritási sorba
MAX(H)	Maximális kulcs értékének visszaadása
KIVESZ-MAX(H)	Maximális kulcsú elem kivétele (vagy minimális)

Kupac

Hatókony prioritási sor megvalósítás.

A kupac egy **majdnem teljes bináris fa**, amiben minden csúcs értéke legalább akkora, mint a gyerekei, ezáltal a maximális (minimális) kulcsú elem a gyökérben van.

Majdnem teljes bináris fa alatt azt értjük, hogy a fa legmélyebb szintjén megengedett, hogy balról jobbra haladva egyszer csak már ne álljon fenn a bináris fa tulajdonság.

Tömbös megvalósítás

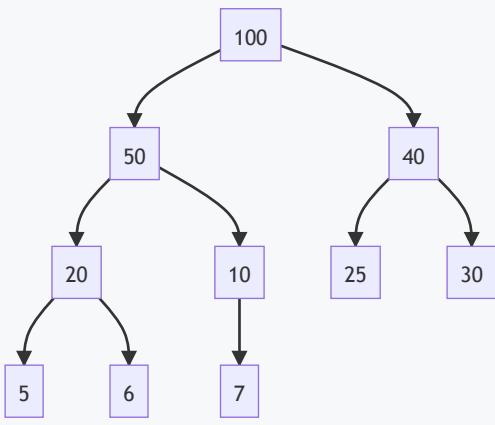
Egybefüggő memóriaterületen van a teljes kupac.

A **szülő**, a **bal gyerek**, és a **jobb gyerek** gyorsan számolható a tömb indexelésével.

```
szülő(i) { // i indexő elem szülője
    return alsoEgeszResz(i / 2)
}
```

```
balGyerek(i) { // i indexű elem bal gyereke
    return 2i
}
```

```
jobbGyerek(i) { // i indexű elem jobb gyereke
    return 2i + 1
}
```



Ennek a kupacnak a tömbös reprezentációja:

```
[100, 50, 40, 20, 10, 25, 30, 5, 6, 7];
```

Kupactulajdonság fenntartása

Garanálnunk kell, hogy az egyes beszúrások, kivételek után a kupacra jellemző tulajdonságok fennmaradnak.

A tulajdonság fenntartására ez a függvény fog felelni:

```

maximumKupacol(A, i) {
    l = balGyerek(i)
    r = jobbGyerek(i)
    if l <= kupacMéret[A] és A[l] > A[i] { // l <= kupacMéret[A] ellenőrzés csak azért kell, hogy az A[l] indexelés biztonságos legnagyobb = l
    } else {
        legnagyobb = i
    }
    if r <= kupacMéret[A] és A[r] > A[i] { // r <= kupacMéret[A] ellenőrzés csak azért kell, hogy az A[r] indexelés biztonságos legnagyobb = r
    }
    if legnagyobb != i {
        csere(A[i], A[legnagyobb])
        maximumKupacol(A, legnagyobb)
    }
}

```

Tehát a vizsgált indexű elemet összehasonlítjuk a gyerekeivel, és ha valamelyik nagyobb, akkor azzal kicséréljük, és rekurzívan meghívjuk rá a `maximumKupacol()`-t, mert lehet, az új szülőjénél/gyerekénél is nagyobb.

`maximumKupacol()` futásideje $O(\log n)$, mert ennyi a majdnem teljes bináris fa mélysége, és legrosszabb esetben az egészben végig kell lépkedni.

Maximum lekérése

A prioritási sor `MAX(H)` függvényének megvalósítása egyszerű, csak vissza kell adnunk a tömb első elemét, ami a kupac gyökere.

```

kupacMaximuma(A) {
    return A[1]
}

```

Maximum kivétele

Ilyenkor az történik, hogy a kupac utolsó elemét áthelyezzük a gyökérbe, és a gyökérből indulva helyreállítjuk a kupac tulajdonságot, "lekupacoljuk" az elemet.

```

kupacbólKiveszMaximum(A) {
    if kupacMéret[A] < 1 {
        throw Error("kupacméter alulcsordulás")
    }
    max = A[1]
    A[1] = A[kupacMéret[A]]
    kupacMéret[A]-- // Méter csökkentése, az érték a memóriában marad, csak nem értelmezzük a kupac részeként.
    maximimKupacol(A, 1) // Mivel beszúrtuk ide az utolsó elemet, helyre kell állítani ("lefelé kupacolni")
}

```

```
    return max  
}
```

Beszúrás

Új elem beszúrása egyszerű, csak szúrjuk be a kupac végére, és onnan kiindulva végezzünk egy helyreállítást, ezzel az új elemet a helyére "felkupacolva".

```
kupacbaBeszur(A, x) {  
    kupacMéter[A]++  
    A[kupacMéret[A]] = x  
    maximumKupacol(A, kupacMéret[A])  
}
```

Futásidők

Művelet	Futásidő
BESZÚR(H, k)	$O(\log n)$
MAX(H)	$\Theta(1)$
KIVESZ-MAX(H)	$O(\log n)$

1.2.5. Fák, és számítógépes reprezentációik

Fa

- Összefüggő, körmentes gráf
- Bármely két csúcsát pontosan egy út köti össze
- Elsőfokú csúcsi: **levél**
- Nem levél csúcsai: **belső csúcs**

Bináris fa

- Gyökeres fa:** Van egy kitűntetett gyökér csúcsa
- Bináris fa:** Gyökeres fa, ahol minden csúcsnak legfeljebb két gyereke van.

Számítógépes reprezentáció

Csúcsokat, és éleket reprezentálunk.

Maga a fa objektumunk egy mutató a gyükérre.

Gyerék éllistás reprezentáció

```
class Node {  
    Object key;  
    Node parent;  
    List<Node> children; // Gyerekek éllistája  
}
```

Első fiú - apa - testvér reprezentáció

```
class Node {  
    Object key;  
    Node parent;  
    Node firstChild;  
    Node sibling;  
}
```

Bináris fa reprezentációja

```
class Node {  
    Object key;  
    Node parent;  
    Node left;  
    Node right;  
}
```

Mindegyik esetben, ha nincs Node, akkor NULL-al jelezhetjük. Pl. a gyökér szülője esetében.

1.2.6. Bináris keresőfák

Absztrakt adatszerkezet a következő műveletekkel:

Művelet	Magyarázat
KERES(T, x)	Megkeresi a fában az x kulcsot, és visszaadja azt a csúcsot
BESZÜR(T, x)	Fába az x kulcs beszúrása
TÖRÖL(T, x)	Fából az x kulcsú csúcs törlése
MIN(T) / MAX(T)	A fa maximális, vagy minimális kulcsú csúcsának visszaadása
KÖVETKEZŐ(T, x) / ELŐZŐ(T, x)	A fában az x kulcsnál egyel nagyobb, vagy egyel kisebb értékű csúcs visszaadása

A T a fa gyökerére mutató mutató.

Cél: minden művelet legalább $O(\log n)$ -es legyen

Bináris keresőfa tulajdonság

Egy x csúcs értéke annak a bal részfájában minden csúcsnál nagyobb vagy egyenlő, jobb részfájában minden csúcsnál kisebb vagy egyenlő.

Keresés

A bináris fa tulajdonságot kihasználva fa keresendő kulcsot hasonlítgatjuk a bal, jobb gyerekhez, és ennek megfelelően lépünk jobbra / balra.

```
fábanKeres(x, k) {
    while x != NULL és k != kulcs[x] {
        if k < kulcs[x] {
            x = bal[x]
        } else {
            x = jobb[x]
        }
    }
    return x
}
```

Minimum / Maximum keresés

A minimum elem a "legbaloldali" elem

```
fábanMinimum(x) {
    while bal[x] != NULL {
        x = bal[x]
    }
    return x
}
```

A maximum elem a "legjobboldali" elem

```
fábanMaximum(x) {
    while jobb[x] != NULL {
        x = jobb[x]
    }
    return x
}
```

Következő / Megelőző

```
fábanKövetkező(x) {
    if jobb[x] != NULL {
        return fábanMinimum(jobb[x])
    }
    y = szülő[x]
    while y != NULL és x == jobb[y] {
        x = y
        y = szülő[y]
    }
}
```

```
    return y  
}
```

Azaz, ha van jobb részfája a fának, amiben keresünk, akkor annak a mimimuma a rákövetkező, ha nincs, akkor pedig addig lépkedünk fel, amíg az aktuális csúcs a szülőjének bal gyereke nem lesz, ugyanis ekkor a szülő a rákövetkező.

Beszűr

```
fábaBeszúr(T, z) {  
    y = null  
    x = gyökér[T]  
    while x != null {  
        y = x  
        if kulcs[z] < kulcs[x] {  
            x = bal[x]  
        } else {  
            x = jobb[x]  
        }  
    }  
    szülő[z] = y  
    if y == null {  
        gyökér[T] = z  
    } else if kulcs[z] < kulcs[y] {  
        bal[y] = z  
    } else {  
        jobb[y] = z  
    }  
}
```

Tehát megkeressük az új elem helyét, az által, hogy jobbra, balra lépkedünk, majd beszűrjuk a megfelelő csúcs alá jobbra, vagy balra.

Töröl

```
fábólTöröl(T, z) {  
    if bal[z] == null vagy jobb[z] == null {  
        y = z  
    } else {  
        y = fábanKövetkező(z)  
    }  
  
    if bal[y] != null {  
        x = bal[y]  
    } else {  
        x = jobb[y]  
    }  
  
    if x != null { // x akkor null, ha y = fábanKövetkező(z)  
        szülő[x] = szülő[y] // "átkötés"  
    }  
  
    if szülő[y] == null {  
        gyökér[T] = x // Ha gyökérbe lett kötve az y, akkor ezt is frissítjük  
    } else if y == bal[szülő[y]] {  
        bal[szülő[y]] = x // "átkötés"  
    } else {  
        jobb[szülő[y]] = x // "átkötés"  
    }  
  
    if y != x {  
        kulcs[z] = kulcs[y]  
    }  
  
    return y  
}
```

Levél törlése

Ha a kitörlendő csúcs egy levél, akkor egyszerűen kitöröljük azt, a szülőkénél a rá mutató mutatót `null`-ra állítjuk.

Egy gyerek belső csúcs

Ebben az esetben a törlendő csúcs helyére bekötjük annak a részfáját (amiből, mivel egy gyereke van, csak egy van).

Két gyerek belső csúcs

Ebben az esetben a csúcs helyére kötjük annak a rákövetkezőjét. Mivel ebben az esetben van biztosan jobb gyereke, így a jobb gyerekének a minimumát fogjuk a helyére rakni (ami mivel egy levél, csak egyszerűen törölhetjük az eredeti helyéről).

Az összes művelet (KERES , MAX / MIN , BESZÚR , TÖRÖL , KÖVETKEZŐ / ELŐZŐ) $O(h)$ -s, azaz a fa magasságával arányos. Ez alap esetben nem feltétlen olyan jó, de kiegyensúlyozott fák esetén jó, hiszen akkor $O(\log n)$ -es.

Pl. AVL-fa, bináris kereső fa kiegyensúlyozott.

1.2.7. Halmaz

Absztrakt adatszerkezet.

Egy elem legfeljebb egyszer szerepelhet benne.

Művelet	Magyarázat
TARTALMAZ(k) (lényegében KERES(k))	Benne van-e egy e k a halmazban?
BESZÚR(K)	Elem behelyezése a halmazba.
TÖRÖL(K)	Elem törlése a halmzból.

Egyéb extra műveletek definiálhatóak, pl.: METSZET , UNIÓ

Közvetlen címzésű táblázat

Egy akkor tömb lefoglalása, mint amekkora a teljes érték univerzum mérete, és ha egy szám eleme a halmaznak, egyszerűen beírjuk ezt a megfelelő indexre.

Jó, mert nagyon gyors megoldás.

Viszont nagy probléma, hogy a tárigény az univerzum méretével arányos, nem pedig a ténylegesen felhasznált elemekkel.

Kis méretű univerzum esetén ajánlatos csak.

1.2.8. Szótár

Absztrakt adatszerkezet.

Egy halmaz elemeihez (kulcsok) egy-egy érték tartozik. Kulcs egyedi, érték ismétlődhet.

dict, asszociatív tömb, map

1.2.9. Hasító tábla

Szótár, és halmaz hatékony megvalósítása.

Cél.: TARTALMAZ , BESZÚR , TÖRÖL műveletek legyenek gyorsak.

Hasító függvény

Kulcsok U univerzumának elemeit (lehetséges kulcsokat) képezi le a hasító táblázat **részére**.

Pl.: $h(k) = k \bmod m$

k a hasító táblázat mérete, azaz a **rések száma**.

Mivel az univerzum, a lehetséges kulcsok száma nagyobb, mint réséké (különben csinálhatnánk tömbös megvalósítást), így elkerülhetetlenné, hogy ürközések legyenek, azaz hogy a hasító függvény két kulcsot ugyan arra a résre képezzen le.

Ezeket az **ütközéseket fel kell oldani**.

Ütközésfeloldás láncolással

A részekben láncolt listák vannak.

Ha olyan helyre akarunk beszúrni, ahol már van elem, akkor a lista elejére szúrjuk be az újat (ez konstans idejű).

Keresés, törlés valamivel romlik, hiszen egy listán is végig kellhet menni.

Kitöltési tényező: $\alpha = \frac{n}{m}$ (**láncok átlagos hossza**)

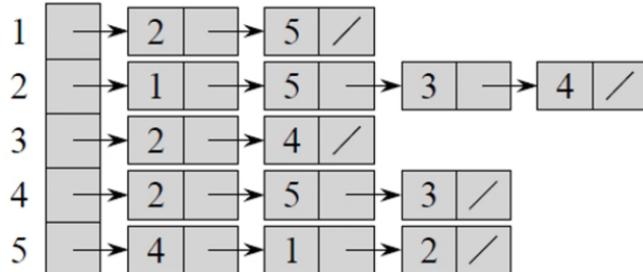
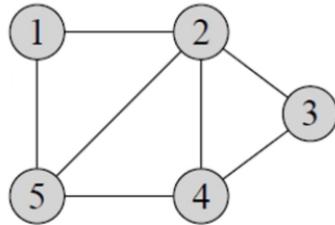
m : részek száma

n : elemek a táblában

Egyszerű egyenletes hasítási feltétel: minden elem egyforma valószínűséggel képződik le bármelyik résre.

Ha egy hasító függvény ezt biztosítja, akkor a keresések (mind sikeres, mind sikertelen) átlagos ideje (nem legrosszabb!) $\Theta(1 + \alpha)$

Ha tudjuk, mennyi elem lesz a táblában, akkor meg tudjuk választani a részek számát úgy, hogy az α egy konstans legyen, ekkor KERES , TÖRÖL , BESZÚR mind $O(1)$.



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

- Csúcsok + élek halmaza
- Szomszédsági mátrix
- Szomszédsági lista

	Létezik (u, v) él?	Összes él listázása	Egy csúcs szomszédainak listázása
Csúcsok + élek halmaza	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
Szomszédsági mátrix	$\Theta(1)$	$\Theta(V ^2)$	$\Theta(V)$
Szomszédsági lista	$\Theta(\text{fokszám})$	$\Theta(E)$	$\Theta(\text{fokszám})$

Érdemes mindenkor előre gondolkodni, hogy milyen reprezentációt választunk, az alapján, hogy milyen gráfokra számítunk, azaz várhatóan milyen az élek és csúcsok eloszlása, azaz mennyire ritka / sűrű a gráf. Ha az élek száma arányos a csúcsok számával, az egy sűrű gráf, ha az élek száma arányos a csúcsok számának négyzetével, az egy ritka gráf.

2. Bonyolultságelmélet

2.1. 1. Hatékony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák.

2.2. A P osztály

P az eldönthető problémák osztálya.

Polinomidőben eldönthető problémák osztálya.

Tehát minden olyan **eldöntési probléma** P-ben van, amire létezik $O(n^k)$ időigényű algoritmus, valamely konstans k -ra.

Ezeket a problémákat tartjuk **hatékonyan megoldhatónak**.

2.2.1. Elérhetőség

P-beli probléma.

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

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

Erre van algoritmus:

- Kiindulásnak veszünk egy $X = 1$ és $Y = 1$ halmazt.
- Mindig kiveszünk egy elemet X -ból, és annak szomszédait betesszük X -be, és Y -ba is.
- Ez által $X \cup Y$ -ban lesznek az 1-ből elérhető csúcsok.

Erre a konkrét implementációtól függően változó lehet, függhet például a gráf reprezentációtól, és a halmaz adatszerkezet megválasztásától. De a lényeg, hogy van-e polinom idejű algoritmus, és mivel általánosságban $O(N^3)$ -el számolhatunk legrosszabb esetnek (előnytelen implementáció esetén is beleférünk), így $O(n^3)$ -ös a futásideje az algoritmusnak (hiszen $N \leq n$, mert biztosan kevesebb a csúcsok száma, mint a gráfot ábrázoló britek).

2.2.2. Hatékony visszavezetés

Rekurzív visszavezetés

A.K.A. Turing-visszavezetés

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

Itt a **rekurzió** azt jelenti, hogy kiszámítható, adható rá algoritmus.

Ebben az esetben ha A eldönthető, akkor B is eldönthető, illetve ha B eldönthetetlen, akkor A is eldönthetetlen.

Lényegében ez azt fejezi ki, hogy " B legalább olyan nehéz, mint A ".

Probléma ezzel a megközelítéssel: Ha A eldönthető probléma, B pedig nemtriviális, akkor $A \leq_R B$.

- Tehát nehézség szempontjából nem mondunk valójában semmit.
- Ennek oka, hogy ebben az esetben az f inputkonvertáló függvényben van lehetőségünk egyszerűen az A probléma megoldására, és ennek megfelelően B egy *igen*, vagy *nem* példányának visszaadására.
- Ez alapján az összes nemtriviális probléma (azaz az olyanok, amik nem minden inputra ugyan azt adják) "ugyan olyan nehéznek" tűnik.
- Probléma oka: **Túl sok erőforrást engedünk meg az inputkonverzióhoz**, annyit, ami elég magának a problémának a megoldására.
- Megoldás: Hatékony visszavezetés.

Hatókony visszavezetés

A.K.A. Polinomidejű visszavezetés

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

Ekkor ha B **polinomidőben** eldönthető, akkor A is eldönthető **polinomidőben**, illetve ha A -ra nincs polinomidejű algoritmus, akkor B -re sincs.

Példa

Egy példa a hatékony visszavezetésre a $PÁROSÍTÁS \leq SAT$

SAT

Input: Egy **CNF** (konjunktív normálformájú formula)

Output: Kielégíthető-e?

Azaz van-e olyan értékkombináció, ami mellett igaz a formula?

PÁROSÍTÁS

Input: Egy G gráf

Output: Van-e G -ben teljes párosítás?

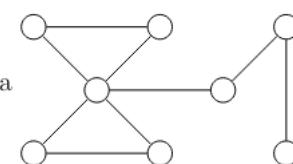
Közös csúccsal nem rendelkező élek halmaza, amik lefednek minden csúcson.

Visszavezetés

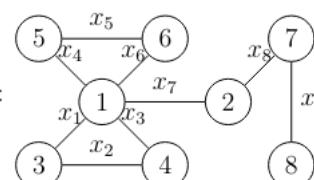
Tehát a cél egy G gráfból egy ϕ_G CNF előállítása választartó módon, polinomidőben úgy, hogy **G -ben pontosan akkor legyen teljes párosítás, ha ϕ_G kielégíthető**.

- minden értelemben rendelünk egy logikai változót.
- Akkor lesz igaz a változó, ha beválasztjuk az élt a teljes párosításba.
- A cél egy olyan CNF előállítása, amiben a következőt formalizáljuk: minden csúcsra felírjuk, hogy pontosan egy él illeszkedik rá, majd ezeket összeességezzük. Ha így egy csúcsra sikerül megfelelő CNF-öt alkotni, akkor azok összeesselése is CNF, hiszen CNF-ek éselése CNF.
- Egy csúcshoz annak formalizálása, hogy pontosan egy él fedik: legalább egy él fedik ÉS legfeljebb egy él fedik.
 - Legalább egy: Egyetlen CNF kell hozzá: $(x_1 \vee x_2 \vee \dots \vee x_k)$.
 - Legfeljebb egy: Négyzetesen sok klóz kell hozzá, minden csúcsprárra megkötjük, hogy "nem ez a kettő egyszerre": $\wedge 1 \leq i < j \leq k \neg(x_i \wedge x_j)$

x_1, \dots, x_k az adott vizsgált csúcsra illeszkedő élek.



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



akkor felcímkézve a változókkal ezt kapjuk:

(adtunk a csúcsoknak is nevet közben)

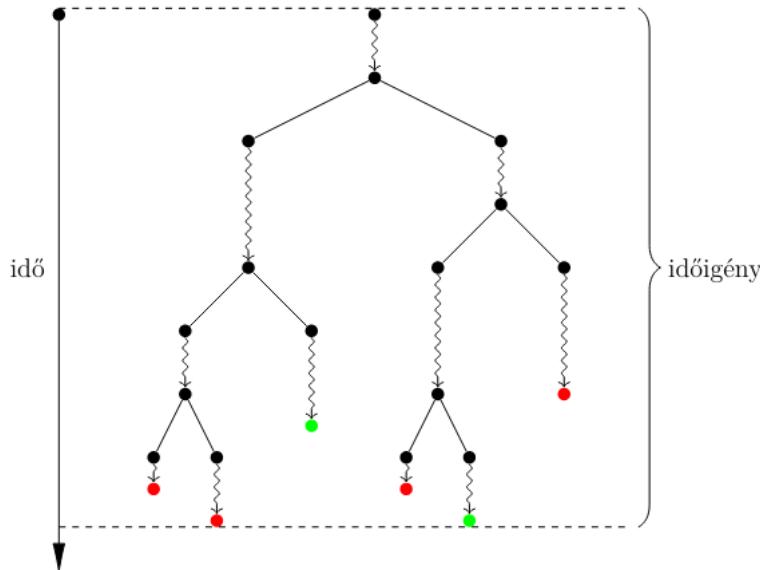
$$\begin{aligned}
& (x_4 \vee x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \\
& \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_7 \vee x_8) \wedge (\neg x_7 \vee \neg x_8) \wedge (x_8 \vee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

2.2.3. Nemdeterminizmus

Nemrealisztikus számítási modell: Nem tudjuk hatékonyan szimulálni.

RAM gépen el lehet képezni a következő utasításképp: `v := nd()`.

Ezzel nemdeterminisztikusan adunk értéket egy bitnek, amit úgy lehet elképzelní, mintha ezen a ponton a számítás elágazna, és az egyik szálon `v = 1`, a másikon `v = 0` értékkel számol. Egy ilyen elágazásnak konstans időben kellene történnie.



A fenti képen egy **számítási fa** van, minden elágazás egy nemdeterminisztikus bitgenerálás.

Időigény: A leghosszabb szál időigénye. Tehát a **számítási fa mélysége**.

Eldöntési algoritmus esetén a végeredmény akkor **true**, ha legalább egy szál **true**, akkor **false**, ha minden szál **false**.

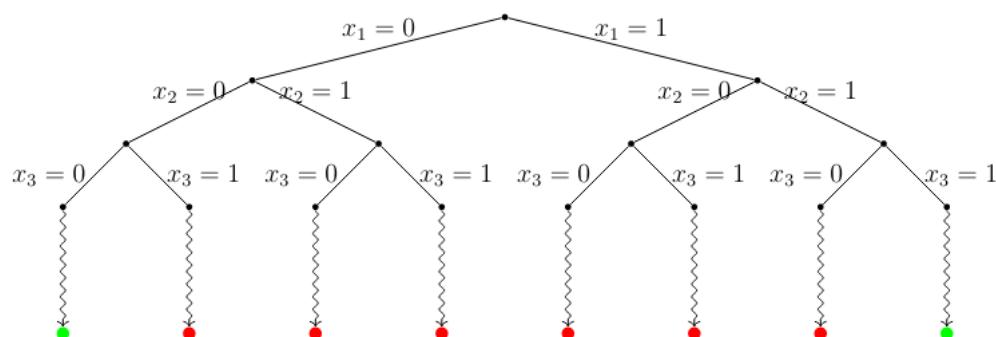
Nemdeterminisztikus algoritmus a SAT-ra

Input formulánkban az x_1, \dots, x_k változók fordulnak elő.

1. Generálunk minden x_i -hez egy nemdeterminisztikus bitet, így kapunk egy értékkedést.
2. Ha a generált értékkedés kielégíti a formulát, adjunk vissza `true`-t, egyébként `false`-t.

Példa input: $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$

Ehhez az inphothoz a számítási fa:



Ennek az algoritmusnak a nemdeterminisztikus időigénye $O(n)$, hiszen n változónak adunk értéket, és a behelyettesítés, ellenőrzés is lineáris időigényű.

2.2.4. Az NP osztály

Nemdeterminisztikus algoritmussal polinomidőben eldönthető problémák osztálya.

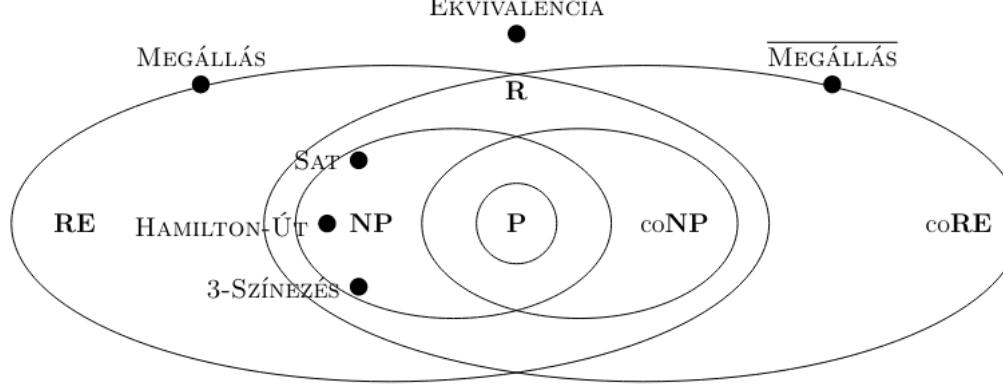
A **SAT** a korábbi példa alapján például **NP-beli**.

$P \subseteq NP$ természetesen igaz, hiszen egy determinisztikusan polinom idejű algoritmus felfogható olyan nemdeterminisztikusnak, ami sosem ágazik el.

$P = coP$ miatt $P \subseteq NP \cap coNP$.

Ennél többet nem tudunk, nem tudjuk, hogy $P = NP$ igaz-e. Széleskörben elfogadott sejtés, hogy nem. Hasonlóan az sem ismert, hogy $NP = coNP$ igaz-e, erről is az az elfogadtott álláspont, hogy nem.

Persze $NP \subseteq R$ is igaz, mert a nemdeterminisztikus számítás szimulálható determinisztikusan, bár ez exponenciálisan lassú.



2.2.5. NP-teljes problémák

C -teljesség definíciója: Ha C problémák egy osztálya, akkor az A probléma

- **C -nehéz**, ha minden C -beli probléma visszavezethető A -ra
- **C -teljes**, ha A még ráadásul C -ben is van

Polinomidőben verifikálhatóság

Az A probléma polinomidőben verifikálható, ha van egy olyan R reláció, **inputok**, és **tanúk** között, melyre:

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

SAT esetében pl. lineáris időben tudjuk ellenőrizni, hogy egy adott értékadás kielégíti-e a CNF-et.

Egy probléma pontosan akkor van NP-ben, ha polinomidőben verifikálható.

SAT

Cook tétele kimondja, hogy a **SAT** egy **NP-teljes** probléma.

Variánsok: FORMSAT, 3SAT is NP-teljes (és minden kSAT $k \geq 3$ -ra), DE 2SAT P-beli, visszavezethető ugyanis az elérhetőségre.

Horn-átnevezhető formulák kielégítése is polinomidőben eldönthető.

Horn-formula, ha minden klózban legfeljebb egy pozitív literál, Horn-átnevezhető, ha bizonyos változók komplementálásával Horn-formulává alakítható.

NP-teljes gráfelméleti problémák

Független csúcshalmaz

Input: Egy G irányítatlan gráf, és egy K szám

Output: Van-e G -ben K darab **független**, azaz páronként nem szomszédos csúcs?

Klikk

Input: Egy G gráf, és egy K szám.

Output: Van-e G -ben K darab páronként szomszédos csúcs?

Hamilton-út

Input: Egy G gráf.

Output: Van-e G -ben Hamilton-út?

Halmazelméleti NP-teljes problémák

Input: Két egyforma méretű halmaz, A , és B , és egy $R \subseteq A \times B$ reláció.

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

A halmaz: lányok, **B** halmaz: fiúk, reláció: ki hajlandó kivel táncolni. Kérdés: Párokba lehet-e osztani mindenkit?

Hármásítás

Input: Két egyforma méretű halmaz, A , B , és C , és egy $R \subseteq A \times B \times C$ reláció.

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

Hasonló példa áll, C halmaz házak, ahol táncolnak.

Pontos lefedés hármásokkal

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

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

Halmazfedés

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

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

Halmazpakolás

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

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

Számelméleti NP-teljes problémák

Egész értékű programozás

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

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

Részletösszeg

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

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

Partíció

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

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

Háttérzár

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

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

TODO: erős-, gyenge NP-teljesség kell-e ide?

2.3. 2. A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárigényű visszavezetés. NL-teljes problémák.

2.3.1. A PSPACE osztály

Determinisztikusan (vagy nemdeterminisztikusan), polinomidőben megoldható problémák osztálya.

- $SPACE(f(n))$: Az $O(f(n))$ tárban eldönthető problémák osztálya.
- $NSPACE(f(n))$: Az $O(f(n))$ tárban **nemdeterminisztikusan** eldönthető problémák osztálya.
- $TIME(f(n))$: Az $O(f(n))$ időben eldönthető problémák osztálya.
- $NTIME(f(n))$: Az $O(f(n))$ időben **nemdeterminisztikusan** eldönthető problémák osztálya.

PSPACE-beli problémák még **nehezebbek, mint az NP-beliek**.

Fontos összefüggés $NSPACE$ és $SPACE$ között

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

Ebből következik ez is:

Hiszen a kettes hatványtól függetlenül $f(n)$ ugyan úgy csak egy polinomiális függvény.

Ennek az összefüggésnek az oka, hogy a tár **újra felhasználható**. Emiatt viszonylag kevés tár is elég sok probléma előntésére. Az idő ezzel szemben sokkal problémásabb, nem tudjuk, hogy egy $f(n)$ időigényű nemdeterminisztikus algoritmust lehet-e $2^{O(f(n))}$ -nél gyorsabban szimulálni.

Lineáris tárigény

Az előbb említett előny miatt elég sok probléma eldönthető $O(n)$ tárban.

Pl. **SAT**, **HAMILTON-ÚT**, és a **3-SZÍNEZÉS** mind eldönthető lineáris tárban. Csak lehetséges tanúkat kell generálni, fontos, hogy egyszerre csak egyet, ezt a tárat használjuk fel újra és újra. Ellenőrizzük a tanút, ha nem jó generáljuk a következőt.

2.3.2. Offline, vagy lyukszámos tárigény

Ha az algoritmus az inputot csak olvassa, és az outputot *stream-mód*-ban írja, akkor az input, output regisztereit nem kell beszámolni, csak a working regisztereit.

A cél ezzel az, mert a korábbiak alapján jó lenne, ha lehetne értelme szublineáris tárigénynek. Márpedig ha pl. az inputot már beszámoljuk, akkor az már legalább lineáris.

2.3.3. Az NL-osztály

- $L = \text{SPACE}(\log n)$: Determinisztikusan logaritmikus tárban eldönthető problémák osztálya.
- $NL = \text{NSPACE}(\log n)$: Nemdeterminisztikusan logaritmikus tárban eldönthető problémák osztálya.

Immermann-Szelepcsényi téTEL szerint: $NL = coNL$

Mit nem szabad, hogy legyen esély NL-beli algoritmust készíteni?

- Az **inputot írni**.
- $\Theta(n)$ méretű bináris tömböt felvenni.

Mit szabad?

- Olyan **változót létrehozni**, amibe 0 és n közti számokat írunk, hiszen ezek $\log n$ tárat igényelnek.
- Nem csak n -ig ér számolni, hanem bármilyen **fix fokszámú polinomig**. Pl. ha n^3 -ig számolunk, az is elfér $\log^3 n = 3 * \log n$ biten, tehát $O(\log n)$ a tárkorlátja.
- Az **input valamelyik elemére rámutatni** egy pointerrel, hiszen lényegében ez is egy 0-tól n -ig értékekkel felvező változó.

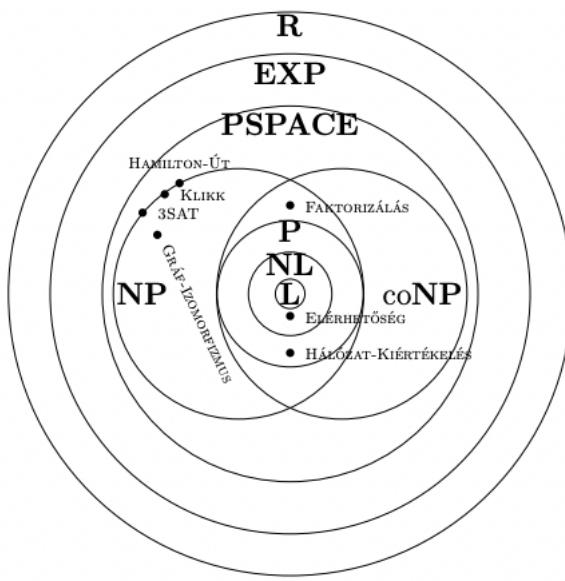
Elérhetőség

Determinisztikusan Savitch tétele szerint az ELÉRHETŐSÉG eldönthető $O(\log^2 n)$ tárban. Ennek oka a rekurzió, hiszen egy példányunk $O(\log n)$ tárás, de ebből egyszerre akár $\log n$ darab is lehet a memoriában.

Nemdeterminisztikusan bele férünk a logtárba. Ekkor "nemdeterminisztikusan bolyongunk" a gráfban, és ha N lépésekben elértünk a csúcsig, akkor `true` amúgy `false`. Tehát minden iterációban átlépünk nemdeterminisztikusan minden szomszédra, ha megtaláltuk a cél csúcsot, `true`, ha nem tudunk már tovább lépni, vagy lefutott minden az N iteráció, akkor `false`.

Ezek alapján tehát:

ELÉRHETŐSÉG $\in \text{NL}$



2.3.4. Logtáras visszavezetés

P -n belül ugye a polinomidejű visszavezetésnek nincs értelme. Hiszen ekkor az inputkonverziót végző függvényben meg tudjuk oldani a problémát, és csak visszaadni egy ismerten `true` vagy `false` inputot.

Definíció

Legyenek A és B eldöntési problémák. Ha f egy olyan függvény, mely

- A inputjaiból B inputjait készíti,
- választartó módon: A "igen" példányiból B "igen" példányait, "nem" példányiból pedig "nem" példányt,
- és logaritmikus tárban kiszámítható,

akkor f egy logtáras visszavezetés A -ról B -re. Ha A és B között létezik ilyen, akkor azt mondjuk, hogy A logtárban visszavezethető B -re, jelben $A \leq_L B$.

f biztosan lyukszagos, hiszen szublineárisnak kell lennie.

Tulajdonságok

A logaritmikus tárígényű algoritmusok polinom időben megállnak, hiszen $O(\log n)$ tárat $2^{O(\log n)}$ féleképp lehet teleírni, minden pillanatban a program K darab konstans utasítás egyikét hajtja éppen végre, így összesen $K * 2^{O(\log n)}$ -féle különbözö konfigurációja lehet, ami polinom.

Ebből következik: Ha $A \leq_L B$, akkor $A \leq_P B$

Azaz a logtáras visszavezetés formailag "gyengébb".

Valójában nem tudjuk, hogy ténylegesen gyengébb-e ez a visszavezetés, de azt tudjuk, hogy akkor lesz gyengébb, ha $L \neq P$.

$L = P$ pontosan akkor teljesül, ha $\leq_L = \leq_P$

Ha f és g logtáras függvények, akkor kompozíciójuk is az. Ez azért jó, mert akkor itt is be lehet vetni azt a trükköt, amit a polinomidejű visszavezetésnél, azaz a C -nehézség bizonyításához elég egy már ismert C -nehéz problémát visszavezetni az adott problémára. Hiszen ekkor tranzitívan minden C -beli probléma visszavezethető lesz az aktuális problémára is.

2.3.5. NL-teljes problémák

Legyen $L \setminus \text{sub}C \subseteq P$ problémák egy osztálya. Azt mondjuk, hogy az A probléma C -nehéz, ha C minden eleme **logtárban** visszavezethető A -ra.

Ha ezen kívül A még ráadásul C -beli is, akkor A egy C -teljes probléma.

Szóval ugyan az, mint P -n kívül, csak logtárban, mivel P -n belül a polinomidejű visszavezetésnek nincs értelme.

P-teljes problémák

- Input egy **változómentes** ítéletkalkulus-beli formula, kiértékelhető-e?
- HÁLÓZAT-KIÉRTÉKELÉS

NL-teljes problémák

- ELÉRHETŐSÉG
- ELÉRHETŐSÉG úgy, hogy az input irányított, **körmentes** gráf

- **ELÉRHETETLENSÉG** (mivel ez az ELÉRHETŐSÉG komplementere, így $coNL$ -teljes, így NL -teljes, hiszen $NL = coNL$ az Immermann-Szelepcsényi téTEL szerint)

- **2SAT** (, és annak a komplementere, megintcsak az Immermann-Szelepcsényi téTEL miatt)

2.3.6. PSPACE-teljes problémák

QSAT

Input: Egy $\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{2m} \phi$ alakú **kvantifikált ítéletlogikai** formula, melynek magja, a ϕ konjunktív normálformájú, **kvantormentes** formula, melyben csak az x_1, \dots, x_{2m} változók fordulnak elő.

Output: Igaz-e ϕ ?

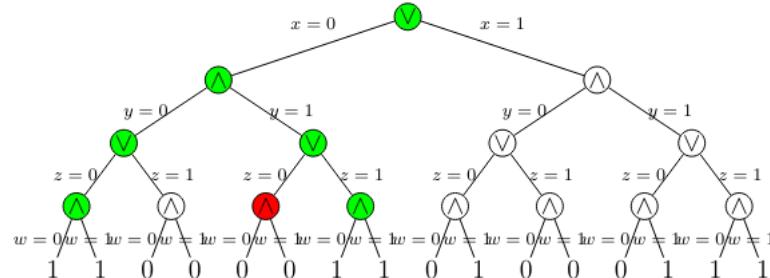
Ez nem első rendű logika, az x_i változók csak igaz / hamis értékeket vehetnek fel.

A QSAT egy **PSPACE-teljes** probléma.

Egy QSAT-ot megoldó rekurzív algoritmus rekurziós fája:

Példa: a felső sorban látható formula rekurziós hívási gráfja lerajzolva:

$$\exists x \forall y \exists z \forall w ((\neg x \vee z \vee w) \wedge (y \vee \neg z) \wedge (x \vee \neg y \vee z))$$



Alul a 0/1-ek a formula magjának a kiértékelései az oda vezető értékadás mellett, a zöld csúcsok értéke 1, a pirosé 0, amelyik fehér maradt, azt (gyorsított kiértékelésnél) ki se kellett értékeljük, pl. mert a vagyolás bal oldala már 1 lett.

Tárigénye $O(n^2)$, mert a rekurziókor lemásoljuk az inputot, ami $O(n)$ méretű, és a mélység $O(n)$

QSAT, mint kétszemélyes, zéró összegű játék

Input: Egy $\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{2m} \phi$ alakú **kvantifikált ítéletlogikai** formula, melynek magja, a ϕ konjunktív normálformájú, **kvantormentes** formula, melyben csak az x_1, \dots, x_{2m} változók fordulnak elő.

Output: Az első játékosnak van-e nyerő stratégiája a következő játékban?

- A játékosok sorban értéket adnak a változóknak, előbb az első játékos x_1 -nek, majd a második x_2 -nek, megint az első stb., végül a második x_{2m} -nek.
- Ha a formula értéke igaz lesz, az első játékos nyert, ha hamis, a második.

FÖLDRAJZI JÁTÉK

Input: Egy $G = (V, E)$ irányított gráf. és egy kijelölt "kezdő" csúcsa.

Output: Az első játékosnak van-e nyerő stratégiája a következő játékban?

- Először az első játékos kezd, lerakja az egyetlen bábuszt a gráf kezdőcsúcsára.
- Ezután a második játékos lép, majd az első, stb., felváltva, mindenkor a bábuszt az aktuális pozíciójából egy olyan csúcsba kell húzzák, ami egy lépésben elérhető, és ahol még nem volt a játék során. Aki először nem tud lépni, vesztett.

További PSPACE-teljes problémák

- Adott egy M determinisztikus RAM program, és egy I inputja. Igaz-e, hogy M elfogadja I -t, méghozzá $O(n)$ tárat használva?
- Adott két reguláris kifejezés. Igaz-e hogy ugyan azokra a szavakra illeszkednek?
- Adott két nemdeterminisztikus automata. Ekvivalensek-e?
- $n \times n$ -es SOKOBAN
- $n \times n$ -es RUSH HOUR

3. Formális Nyelvek

3.1. 1. Véges automata és változatai, a felismert nyelv definíciója. A reguláris nyelvtanok, a véges automaták, és a reguláris kifejezések ekvivalenciája. Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei.

3.1.1. Véges automata

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **determinisztikus automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, az **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**
- $\delta : Q \times \Sigma \rightarrow Q$ egy leképezés, az **átmenetfüggvény**

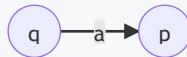
Példa:

- $Q = q_0, q_1, q_2$
- $\Sigma = a, b$
- $F = q_0$
- δ
 - $\delta(q_0, a) = q_1$
 - $\delta(q_1, a) = q_2$
 - $\delta(q_2, a) = q_0$
 - $\delta(q_0, b) = q_0$
 - $\delta(q_1, b) = q_1$
 - $\delta(q_2, b) = q_2$

Automata megadása irányított gráfkként

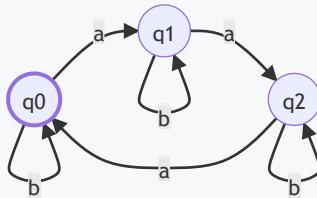
Gráf csúcsai az automata állapotai

Ha $\delta(q, a) = p$, akkor a q csúcsból egy élet irányítunk a p csúcsba, és az élet ellátjuk az a címkelvel



Itt az automata a q állapotból az a input szimbólum hatására átmegy a p állapotba.

A korább példa automata megadása gráffal:



A q_0 állapot jelen példában a végállapot is, amit a vastagított szél jelez.

Automata megadása táblázatként

Első sorban a kezdőállapot, végállapotokat meg kell jelölni (itt most csillag).

A korább példa automata megadása táblázattal:

δ	a	b
* q_0	q_1	q_0
q_1	q_2	q_1

q_2	q_0	q_2
-------	-------	-------

Csillag jelzi, hogy az adott sor állapota végállapot.

Automata átmenetei

M konfigurációinak halmaza: $C = Q \times \Sigma^*$

A $(q, a_1 \dots a_n)$ konfiguráció azt jelenti, hogy M a q állapotban van és az $a_1 \dots a_n$ szót kapja inputként.

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in \Sigma$ -ra, és $\delta(q, a) = q'$.

Azaz amikor az automata átmegy q -ból q' -be, akkor az ehhez "felhasznált" szimbólumot leveszi az input szó elejéről. Pl. itt a hatására ment, és $w = aw'$, így az átmenet után az input szó már csak w' az a nélkül. Mondhatni, hogy az a -t felhasználta az átmenethez.

Átmeneti reláció fajtái

- $(q, w) \vdash_M (q', w')$: Egy lépés
- $(q, w) \vdash_M^n (q', w')$, $n \geq 0$: n lépés
- $(q, w) \vdash_M^+ (q', w')$: Legalább egy lépés
- $(q, w) \vdash_M^* (q', w')$: Valamennyi (esetleg 0) lépés

Az M jelölés egy automatát azonosít, elhagyható, ha éppen csak 1 automatáról beszélünk, mert ilyenkor egyértelmű

$*$, $+$ és * itt is, és mindenhol ebben a tárgyban úgy működik, mint megsokott regexeknél

Felismert nyelv

Az $M = (Q, \Sigma, \delta, q_0, F)$ automata által felismert nyelven az $L(M) = w \in \Sigma^* | (q_0, w) \vdash_M^* (q, \epsilon)$ és $q \in F$ nyelvet értjük.

Azaz q_0 -ból w hatására valamelyik $q \in F$ végállapotba jutunk

ϵ az üres szó

Nemdeterminisztikus automata

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **nemdeterminisztikus automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, az **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ egy leképezés, az **átmenetfüggvény***

Azaz ugyan az, mint a determinisztikus, csak egy input szimbólum hatására egy állapotból többé is átmehet.

A determinisztikus automata ezen általánosítása (hiszen ez egy általánosítás, a determinisztikus automata is lényehében olyan nemdeterminisztikus ami minden állapotnak egy egyelemű halmazába tér át) **nem növeli meg a felismerő kapacitást**, tehát egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus automatával, ha felismerhető determinisztikus automatával.

Ezt "hatvány halmaz módszerrel" lehet bebizonyítani, meg kell nézni, hogy a hatására milyen állapotokba tud kerülni a nemdeterminisztikus automata, és azonkah az uniója lesz egy állapot. Ez a "determinizálás", aminek a során az állapotok száma nagyban megnőhet (akár exponenciálisan).

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in \Sigma$ -ra, és $q' \in \delta(q, a)$.

Felismert nyelv

Az $M = (Q, \Sigma, \delta, q_0, F)$ (nemdeterminisztikus) automata által felismert nyelven az $L(M) = w \in \Sigma^* | (q_0, w) \vdash_M^* (q, \epsilon)$ valamely $q \in F$ -re nyelvet értjük.

Azaz q_0 -ból a w hatására elérhető valamely $q \in F$ végállapot. DE! Nem baj, ha elérhetően nem-végállapotok is.

Teljesen definiált automata

Akkor teljesen definiált egy automat, ha minden szót végig tud olvasni.

Azaz nem tud pl. egy $\delta(q, a) = \emptyset$ átmenet miatt elakadni.

Azaz akkor teljesen definiált, ha minden $q \in Q$ és $a \in \Sigma$ esetén $\delta(q, a)$ **legalább** egy elemű.

Determinisztikus automaták teljesen definiáltak, hiszen pontosan egy állapotba léphetünk tovább.

Nemdeterminisztikus automaták pedig teljesen definíálhatóvá tehetők "csapda" állapot bevezetésével, anélkül, hogy a felismert nyelv megváltozna.

- Felveszünk egy q_c állapotot (ez a "csapda") állapot.
- $\delta(q, a) = \emptyset$ esetén legyen $\delta(q, a) = q_c$
- Legyen $\delta(q_c, a) = q_c$ minden $a \in \Sigma$ -ra.

A 3. pont az, ami miatt ez egy "csapda", nem lehet már ebből az állapotból kijönni.

Nemdeterminisztikus ϵ -automata

Tartalmaz ϵ -átmeneteket.

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **nemdeterminisztikus ϵ -automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, az **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**
- $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow \mathcal{P}(Q)$ egy leképezés, az **átmenetfüggvény**

Azaz ugyan olyan, mint a nemdeterminisztikus, csak lehet olyan átmenete, ami "nem fogyasztja" az inputot. Ez az ϵ -átmenet.

Ez sem bővíti a felismerő kapacitást, egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus ϵ -átmenetes automatával, ha felismerhető nemdeterminisztikus automatával. ϵ automata ϵ -mentesítéssel átalakítható nemdeterminisztikus automatává, ekkor az automata a q állapotból az a hatására azon állapotokba megy át, amelyekre M valamennyi (akár 0) ϵ -átmenettel, majd egy a -átmenettel jut el, továbbá az automata végállapotai azon az állapotok, amiből valamennyi (akár 0) ϵ -átmenettel egy F -beli állapotba jut.

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in (\Sigma \cup \epsilon)$ -ra, és $q' \in \delta(q, a)$.

Ha $a = \epsilon$, akkor éppen $w = w'$

Felismert nyelv

Felismert nyelv definíciója ugyan az, mint a sima nemdeterminisztikus esetben.

3.1.2. Ekvivalencia tételek

Tetszőleges $L \subseteq \Sigma^*$ nyelv esetén a következő három állítás ekvivalens:

1. L reguláris (generálható reguláris nyelvtannal).
2. L felismerhető automatával.
3. L reprezentálható reguláris kifejezéssel.

Ezt külön három párra lehet belátni.

* Reguláris nyelvtan

Egy $G = (N, \Sigma, P, S)$ nyelvtan reguláris (vagy jobblineáris), ja P -ben minden szabály $A \rightarrow xB$ vagy $A \rightarrow x$ alakú.

Egy L nyelvet reguláris nyelvnek hívunk, ha van olyan G reguláris nyelvtan, melyre $L = L(G)$ (azaz öt generálja).

Az összes reguláris nyelvek halmazát REG -el jelöljük.

$REG \subset CF$

Azaz vannak olyan környezetfüggetlen nyelvek, amik nem regulárisak.

* Reguláris kifejezések

Egy Σ ábécé feletti reguláris kifejezések halmaza a $(\Sigma \cup \emptyset, \epsilon, (,), +, *)^*$ halmaz legsűkebb olyan U részhalmaza, amelyre az alábbi feltételek teljesülnek:

1. Az \emptyset szimbólum eleme U -nak
2. Az ϵ szimbólum eleme U -nak
3. minden $a \in \Sigma$ -ra az a szimbólum eleme U -nak
4. Ha $R_1, R_2 \in U$, akkor $(R_1) + (R_2)$, $(R_1)(R_2)$ és $(R_1)^*$ is elemei U -nak.

U -ban tehár maguk a kifejezések vannak.

Az R reguláris kifejezés által meghatározott (reprezentált) nyelvet $|R|$ -el jelöljük, és a következőképp definiáljuk:

- Ha $R = \emptyset$, akkor $|R| = \emptyset$ (üres nyelv)
- Ha $R = \epsilon$, akkor $|R| = \epsilon$
- Ha $R = a$, akkor $|R| = a$
- Ha:
 - $R = (R_1) + (R_2)$, akkor $|R| = |R_1| \cup |R_2|$
 - $R = (R_1)(R_2)$, akkor $|R| = |R_1||R_2|$
 - $R = (R_1)^*$, akkor $|R| = |R_1|^*$

Reprezentálható nyelvek regulárisak

3 → 1 az ekvivalencia tételeben.

Ha $L \subseteq \Sigma^*$ nyelv reprezentálható reguláris kifejezéssel, akkor generálható reguláris nyelvtannal.

Ez R struktúrája szerinti indukcióval belátható.

Reguláris nyelvek felismerhető automatával

1 → 2 az ekvivalencia tételeben.

Ha $L \subseteq \Sigma^*$ nyelv reguláris, akkor felismerhető automatával.

Ennek bizonyítását ez a két lemma képezi, ezekkel fel tudunk írni egy automatát a nyelvtanból:

- minden $G = (N, \Sigma, P, S)$ reguláris nyelvtanhoz megadható vele ekvivalens $G' = (N', \Sigma, P', S)$ reguláris nyelvtan, úgy, hogy P' -ben minden szabály $A \rightarrow B, A \rightarrow aB$, vagy $A \rightarrow \epsilon$ alakú, ahol $A, B \in N$ és $a \in \Sigma$.
 - Ez az átalakítás EZ, csak láncolva új szabályokat kell felvenni, pl. $A \rightarrow bbB$ helyett $A \rightarrow bA_1, A_1 \rightarrow bB$
- minden olyan $G = (N, \Sigma, P, S)$ reguláris nyelvtanhoz, melynek csak $A \rightarrow B, A \rightarrow aB$ vagy $A \rightarrow \epsilon$ alakú szabályai vannak, megadható olyan $M = (Q, \Sigma, \delta, q_0, F)$ nemdeterminisztikus ϵ -automata, amelyre $L(M) = L(G)$.

Ez is EZ, hiszen az $A \rightarrow aB$ jellegű szabályok könnyen felírhatók automatáként, A -ból megy a hatására B -be

Automatával felismerhető nyelvek reprezentálhatók

2 → 3 az ekvivalencia tételeben

Minden, automatával felismerhető nyelv reprezentálható reguláris kifejezéssel.

3.1.3. Pumpáló lemma reguláris nyelvekre

Minden $L \subseteq \Sigma^*$ reguláris nyelv esetén megadható olyan (L -től függő) $k > 0$ egész szám, hogy minden $w \in L$ -re ha $|w| \geq k$, akkor van olyan $w = w_1w_2w_3$ felbontás, melyre $0 < |w_2|$ és $|w_1w_2| \leq k$, és minden $n \geq 0$ -ra, $w_1w_2^n w_3 \in L$

Ha egy L nyelvhez nem adható meg ilyen k , akkor az nem reguláris. Így ezen lemma segítségével bebizonyítható nyelvekről, hogy azok nem regulárisak.

A k szám az L -et felismerő egyik determinisztikus automata (több is felismeri) állapotainak száma.

3.1.4. A pumpáló lemma alkalmazása

A lemma arra használható, hogy nyelvekről belássuk, hogy az nem reguláris.

Példa: Az $L = a^n b^n \mid n \geq 0$ nyelv nem reguláris.

Bizonyítás: Tegyük fel, hogy L reguláris. Akkor megadható olyan k szám, ami teljesíti a pumpáló lemma feltételeit. Vegyük az $a^k b^k \in L$ szót, melynek hossza $2k \geq k$. A pumpáló lemmában szereplő feltételek szerint létezik $a^k b^k = w_1 w_2 w_3$ felbontás, melyre $0 < |w_2|, |w_1 w_2| \leq k$ és minden $n \geq 0$ -ra $w_1 w_2^n w_3 \in L$. Mivel $|w_1 w_2| \leq k$, a középső w_2 szó csak a betűkből áll. Továbbá a $0 < |w_2|$ feltétel miatt a $w_1 w_2^2 w_3, w_1 w_2^3 w_3$, stb szavakban az a -k száma nagyobb, mint a b -k száma, tehát ezen szavak egyike sincs L -ben. Ellentmondás, tehát nem létezik ilyen k szám. Akkor viszont az L nyelv nem reguláris.

Tehát az a baj ezzel a nyelvvel, hogy csak a -kat tudnánk bele pumpálni, de ez kivezet a nyelvből.

3.1.5. Következmények

- Egy automata nem képes számolni, hogy két betű ugyanannyiszor szerepel-e.
- Van olyan környezetfüggetlen nyelv, ami nem reguláris. Azaz $REG \subset CF$. Például ilyen az előző L nyelv.

3.2. 2. A környezetfüggetlen nyelvtan, és nyelv definíciója. Derivációk, és derivációs fák kapcsolata. Veremautomaták, és környezetfüggetlen nyelvtanok ekvivalenciája. A Bar-Hillel lemma és alkalmazása.

3.2.1. Környezetfüggetlen nyelvtan

Egy $G = (N, \Sigma, P, S)$ négyes, ahol:

- N egy ábécé, a nemterminális ábécé
- Σ egy ábécé a terminális ábécé, amire $N \cap \Sigma = \emptyset$
- $S \in N$ a kezdő szimbólum
- P pedig $A \rightarrow \alpha$ alakú ún. átírási szabályok véges halmaza, ahol $A \in N$, és $\alpha \in (N \cup \Sigma)^*$

Környezetfüggetlen nyelvek

Egy L nyelvet környezetfüggetlennek hívunk, ha van olyan G környezetfüggetlen nyelvtan, melyre $L = L(G)$.

Az összes környezetfüggetlen nyelv halmazát CF -vel jelöljük.

Például az $a^n b^n \mid n \geq 0$ nyelv környezetfüggetlen.

Deriváció

Tetszőleges $\gamma, \delta \in (N \cup \Sigma)^*$ esetén $\gamma \Rightarrow_G \delta$, ha van olyan $A \rightarrow \alpha \in P$ szabály és vannak olyan $\alpha', \beta' \in (N \cup \Sigma)^*$ szavak, amelyekre fennállnak, hogy $\gamma = \alpha' A \beta'$, $\delta = \alpha' \alpha \beta'$.

Azaz, ha egy átírással (valamelyik P -beli szabály mentén) átvihető.

Fajtái

- $\gamma \Rightarrow_G \delta$: Egy lépés, közvetlen levezetés, közvetlen deriváció
- $\gamma \Rightarrow_G^n \delta, n \geq 0$: n lépés (0 lépés önmagába viszi)
- $\gamma \Rightarrow_G^+ \delta$: Legalább egy lépés
- $\gamma \Rightarrow_G^* \delta$: Valamennyi (akár 0) lépés

A G alsó indexben elhagyható, ha 1 db nyelvtanról van éppen szó.

Generált (környezetfüggetlen) nyelv

A $G = (N, \Sigma, P, S)$ környezetfüggetlen nyelvtan által generált nyelv:

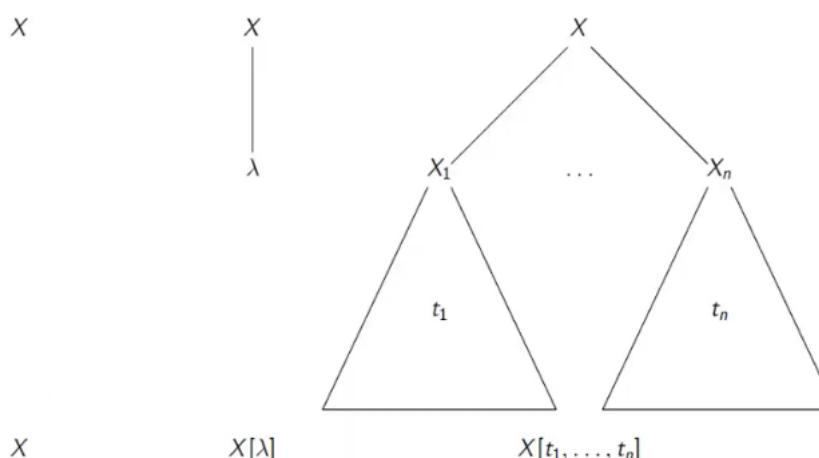
$$L(G) = w \in \Sigma^* \mid S \Rightarrow_G^* w$$

Azaz az összes olyan szó, ami G -ból levezethető.

3.2.2. Derivációs fák, kapcsolatuk a derivációkkal

Az $X \in (N \cup \Sigma)$ gyökerű derivációs fák halmaza a legsűkebb olyan D_X halmaz, amelyre:

- Az a fa, amelynek egyetlen szögpontja (vagyis csak gyökere) az X , eleme D_X -nek.
- Ha $X \rightarrow \epsilon \in P$, akkor az a fa, amelynek gyökere X , a gyökerének egyetlen leszármazottja az ϵ , eleme D_X -nek.
- Ha $X \rightarrow X_1 \dots X_k \in P$, továbbá $t_1 \in D_{X_1}, \dots, t_k \in D_{X_k}$, akkor az a fa, amelynek gyökere X , a gyökeréből k él indul rendre a t_1, \dots, t_k fák gyökeréhez, eleme D_X -nek.



Legyen t egy X gyökerű derivációs fa. Akkor t magasságát $h(t)$ -vel, a határát pedig $fr(t)$ -vel jelöljük és az alábbi módon definiáljuk:

- Ha t az egyetlen X szögpontból álló fa, akkor $h(t) = 0$ és $fr(t) = X$.
- Ha t gyökere X , aminek egyetlen leszármazottja ϵ , akkor $h(t) = 1$, és $fr(t) = \epsilon$.

- Ha t gyökere X , amiből k él indul rendre a t_1, \dots, t_k közvetlen részfák gyökeréhez, akkor $h(t) = 1 + \max h(t_i \mid 1 \leq i \leq k)$ és $fr(t) = fr(t_1) \dots fr(t_k)$.

Azaz $h(t)$ a t -ben levő olyan utak hosszának maximuma, amelyek t gyökeréből annak valamely leveléhez vezetnek.

Azaz $fr(t)$ azon $(N \cup \Sigma)^*$ -beli szó, amelyet t leveleinek balról jobbra (vagy: preorder bejárással) történő leolvasásával kapunk.

Az összefüggés derivációs fák, és derivációk közt

Tetszőleges $X \in (N \cup \Sigma)$ és $\alpha \in (N \cup \Sigma)^*$ esetén $X \Rightarrow^* \alpha$ akkor, és csak akkor, ha van olyan $t \in D_X$ derivációs fa, amelyre $fr(t) = \alpha$.

Az összefüggés következményei

- Tetszőleges $w \in \Sigma^*$ esetén $S \Rightarrow^* w$ akkor és csak akkor, ha van olyan S gyökerű derivációs fa, amelynek határa w .

Ez csak a korábbi téTEL alkalmazása S -re, és egy w -re.

- Tetszőleges $w \in \Sigma^*$ esetén a következő állítások ekvivalensek:

- $w \in L(G)$
- $S \Rightarrow^* w$
- $S \Rightarrow_l^* w$ (ez bal oldali deriváció, minden a legbaloldalibb nemterminális lehet csak helyettesíteni)
- van olyan S gyökerű derivációs fa, amelynek határa w .

Generált nyelv definiálása derivációs fákkal

$$L(G) = \{fr(t) \mid t \in D_S, fr(t) \in \Sigma^*\}$$

4. Közelítő és szimbolikus számítások

Numerikus stabilitás jelentése: A függvény argumentumainak megváltozása mekkora eltérést eredményez a függvényértékben. Ha nagyon akkor numerikusan nem stabilis.

4.1. 1. Eliminációs módszerek, mátrixok trianguláris felbontásai. Lineáris egyenletrendszerek megoldása iterációs módszerekkel. Mátrixok sajátértékeinek, és sajátvektorainak numerikus meghatározása.

4.1.1. Eliminációs módszerek

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \quad a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \quad \dots \quad a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

Tegyük fel, hogy $A \in \mathbb{C}^{n \times n}$, és $b \in \mathbb{C}^n$. Az $Ax = b$ lineáris egyenletrendszernek pontosan akkor van egyetlen megoldása, ha A nem szinguláris (azaz $\det A \neq 0$). Ekkor a megoldás $x = A^{-1}b$. A megoldás i . komponensét megadja a Cramer szabály is:

$$x_i = \frac{\det A^{(i)}}{\det A}$$

$A^{(i)}$ mátrixot úgy kapjuk, hogy az A mátrix i . oszlopát kicseréljük a b vektorral.

Gyakorlatban ez a téTEL nem használatos, mert az inverz számolás nagy műveletigényű lehet, a Cramer szabály pedig numerikusan nem stabilis.

Lineáris egyenletrendszerek megoldási módjai

- Direkt módszerek: Véges sok, meghatározott számú lépésben megtalálják a megoldást.
- Iterációs módszerek: minden iterációs lépésben jobb és jobb közelítést adják a megoldásnak.
 - Magát a megoldást általában nem érik el véges lépésben.

Egyenletrendszerek ekvivalenciája

Két egyenletrendszer akkor tekintünk ekvivalensnek, ha a megoldásai halmaza megegyezik.

Megengedett transzformációk:

- Egy egyenletnek egy nem nulla számmal való beszorzása.
- Egy egyenlet konstansszorosának hozzáadása egy másik egyenlethez.

Egyenletrendszerek megoldása

Ilyen (ekvivalens) átalakításokkal próbálunk **háromszög mátrixot** vagy **diagonális mátrixot** létrehozni. Ez azért jó, mert ilyen alakban az egyenletrendszer könnyen megoldható:

\$\$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & 0 & u_{22} & u_{23} & 0 & 0 & u_{33} \end{bmatrix}$$

$$[x_1 \ x_2 \ x_3]$$

=

$$[b_1 \ b_2 \ b_3]$$

\$\$

Ilyen az esetben a megoldás könnyen kifejezhető:

$$x_3 = \frac{b_3}{u_{33}} \quad x_2 = \frac{b_2 - u_{23}x_3}{u_{22}} \quad x_1 = \frac{b_1 - u_{12}x_2 - u_{13}x_3}{u_{11}}$$

Matlab program

A fentebbi példa módszerének általánosítása felső trianguláris mátrixokra.

```
function x = UTriSol(U, b)
n = length(b);
x = zeros(n, 1);
for j = n : -1 : 2
    x(j) = b(j) / U(j, j);
    b(1:j - 1) = b(1:j - 1) - x(j) * U(1:j - 1, j);
end
x(1) = b(1) / U(1, 1);
```

Műveletigénye $O(\frac{n^2}{2})$

Eliminációs mátrix

A $G_j \in \mathbb{R}^{n \times n}$ **eliminációs mátrix**, ha felírható $G_j = I + g^{(j)} e_j^T$ alakban valamely $1 \leq j \leq n$ -re egy olyan $g^{(j)}$ vektorral, amelynek j -dik komponense, $g_j^{(j)} = 0$

Példa

\$\$ j = 3; G_j =

$$[1 \ 0 \ 2 \ 0 \ 1 \ 3 \ 0 \ 0 \ 1]$$

\$\$

$j = 3$ a mátrix 3. oszlopában látszódik is, csak ott tér el egy egységmátrixtól.

G_j komponensei:

$$g^{(j)} = [2 \ 3 \ 0]; e_j^T = [0 \ 0 \ 1]$$

$j = 3$ miatt a $g^{(j)}$ harmadik sora nulla, illetve az e_j^T csak harmadik koordinátája 1.

Eliminációs mátrix jelentősége

Egy $A \in n \times n$ mátrixot a $G_j = I + g^{(j)} e_j^T$ eliminációs mátrixszal balról szorozva a $B = G_j A$ szorzatmátrix úgy áll elő, hogy A 1, 2, ..., n -dik sorához rendre hozzáadjuk A j -dik sorának $g_1^{(j)}, g_2^{(j)}, \dots, g_n^{(j)}$ -szeresét.

Például a következő mátrixok esetén:

\$\$ A =

$$[1 \ 2 \ 4 \ 6 \ 8 \ 2 \ 9 \ 1 \ 0]$$

$G_j =$

$$[1 \ 0 \ 2 \ 0 \ 1 \ 3 \ 0 \ 0 \ 1]$$

\$\$

Az eredmény:

$$G_j A = [19 \ 4 \ 4 \ 33 \ 11 \ 2 \ 9 \ 1 \ 0]$$

Az A mátrix első sorához valóban kétszer a másodikhoz háromszor a harmadikhoz pedig nullaszor lett hozzáadva az A mátrix harmadik sora.

Könnyen megadható olyan eliminációs mátrix, amivel egyadott oszlop (vagy egy önálló vektor) **egy adott koordináta alatti elemei kinullázhatóak**, például a fentebbi A mátrixhoz ($a_{11} - et$ módszertől 2-re, hogy szemléletesebb legyen a példa):

\$\$

$$\begin{bmatrix} 1 & 0 & 0 & -3 & 1 & 0 & -\frac{9}{2} & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 2 & 4 & 6 & 8 & 2 & 9 & 1 & 0 \end{bmatrix}$$

=

$$\begin{bmatrix} 2 & 2 & 4 & 0 & 2 & -10 & 0 & -8 & -18 \end{bmatrix}$$

\$\$

Az első oszlopban ténylegesen kinullázódott két sor, már csak a második oszlopban kellene az utolsó sort kinullázni, és egy könnyen megoldható egyenletrendszer együtthatómátrixát kapnánk.

4.1.2. Mátrixok trianguláris felbontásai

LU felbontás

Át akarjuk alakítani az $Ax = b$ egyenletrendszeret úgy, hogy a bal oldalon háromszögmátrix szerepeljen.

Ezt valamennyi eliminációs mátrix sorozatával meg tudjuk tenni:

$$M A x = M_{n-1} \dots M_1 A x = M_{n-1} \dots M_1 b = M b$$

Hasonló felbontás megkezdése történt az előző példában.

$$\text{Ekkor } L = M^{-1}, U = M A$$

Könnyű számolni, mert az eliminációs mátrix inverze úgy számolható, hogy a főátlón kívüli elemeket negáljuk.

Egyenletrendszer megoldása LU felbontással

$Ax = b$ helyett az $L U x = b$ egyenletrendszer oldhatjuk meg. Ezt **két lépésben** elvégezve végig háromszögmátrixokkal dolgozhatunk.

Ezzel megkaptuk a **Gauss-elimináció** módszerét:

1. Az A mátrix LU felbontása
2. $L y = b$ megoldása y -ra (az y egy új, mesterséges változó)
3. $U x = y$ megoldása x -re

```
[L, U] = LU(A);
y = LTriSol(L, b);
x = UTriSol(U, y);
```

* LU felbontás Matlabban

```
function [L, U] = LU(A)
[m, n] = size(A);
for k = 1:n-1
    A(k+1:n, k) = A(k+1:n, k) / A(k, k);
    A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n, k) * A(k, k+1:n);
end
L = eye(n, n) + tril(A, -1);
U = triu(A);
```

Főelemkiválasztás

Az LU felbontás csak akkor sikeres, ha az A mátrix nem szinguláris, és minden generáló elem (főátló-beli elemek) nullától különböző (mivel azokkal leosztunk). Ha az utóbbi nem teljesül, még lehet, hogy van felbontás, átrendezéssel, ami ekvivalens feladatot eredményez. Ezt az eljárást főelemkiválasztásnak hívjuk.

Ezeket a sorcseréket egy **permutációs mátrixszal** való beszorzással végezzük.

A P_{ij} permutációs mátrix egy egységmátrix, melyben az i -edik, és j -edik sor fel van cserélve. Dimenziószáma megegyezik a "permutálendő" mátrixéval.

Az A mátrixot ezzel a P_{ij} -vel balról szorozva egy olyan mátrixot kapunk, ami az A mátrix, melyben az i -edik, és j -edik sor fel van cserélve.

Jobbról szorozva az oszlopok cserélődnek.

Cholesky felbontás

Rírka mátrixok esetén hatékonyabb, mint a Gauss-elimináció.

Ha az A mátrix szimmetrikus, és pozitív definit, akkor az LU felbontás $U = L^T$ alakban létezik, tehát $A = LL^T$, ahol L alsó háromszögmátrix, amelynek diagonális elemei pozitív számok. Az ilyen felbontást **Cholesky-felbontásnak** hívjuk.

Matlab implementáció

```
function [x] = LGPD(A, b);
R = chol(A);
y = R' \ b;
x = R \ y;
```

A \backslash operátor transzponál, a \backslash pedig: $R \backslash y := A z$ $R x = y$ egyenletrendszer megoldása

A Cholesky felbontás numerikusan stabilis, műveletigénye $\frac{1}{3}n^3 + O(n^2)$. Feleannyi, mint egy általános mátrix LU felbontásáé.

QR ortogonális felbontás

Egy Q négyzetes mátrix ortogonális, ha $QQ^T = Q^TQ = I$.

Az ortogonális transzformációk megtartják a kettes normát, így numerikusan stabilisak.

Lineáris egyenletrendszer megoldása az $A = QR$ felbontással:

$$Rx = Q^T b$$

Matlabban

```
[Q, R] = qr(A, 0);
x = R \ (Q' * b);
```

Tetszőleges A négyzetes valós reguláris mátrixnak létezik az $A = QR$ felbontása ortogonális és felső háromszögmátrixra.

4.1.3. Lineáris egyenletrendszerek megoldása iterációs módszerekkel

A korábbi megoldási módok **direkt módszerek voltak**, véges lépésekben megtalálták a megoldást. A következők iterációs módszerek, minden iterációban egyre jobb közelítéseket adnak, de általában véges lépésekben nem találják meg a megoldást. Mégis nagyobb méterű, sűrűbb mátrixok esetén előnyös a használatuk.

Jacobi iteráció

Nem minden esetben konvergál a jacobi iteráció! (A megoldás felé)

A módszer:

1. Felírjuk az egyenleteket olyan formában, hogy a bal oldalra rendezünk 1-1 változót.
2. Választunk egy kiindulási x_0 vektort.
3. Elkezdjük az iterációt, minden a megkapott értékeket behelyettesítjük a kifejezett változó jobb oldalába (nulladik iterációban x_0 -t).
4. Ezt addig ismételgethetjük, amíg az eltérés két eredmény között megfelelően kicsi.

Példa

$$-2x_1 + x_2 + 5x_3 = 4 \quad 4x_1 - x_2 + x_3 = 4 \quad 2x_1 - 4x_2 + x_3 = -1$$

Ehhez tartozó **iterációs egyenletek**:

$$x_1 = \frac{-4 + x_2 + 5x_3}{2} \quad x_2 = -4 + 4x_1 + x_3 \quad x_3 = -1 - 2x_1 + 4x_2$$

Ez éppenséggel az $x_1 = (1, 2, 3)^T$ kezdővektorral divergál a megoldástól.

$Bx^{(k)} + c$ az **iterációs egyenleteknek** az általános, tömör felírása.

Jacobi iteráció konvergenciája

Az, hogy a B mátrix **diagonálisan domináns**, elegendő feltétele a Jacobi iteráció konvergenciájának.

Egy mátrix akkor diagonálisan domináns, ha minden sorban a diagonális elem abszolútértékben nagyobb, mint az összes többi sor-beli elem abszolútértékben vett összege.

Iterációs módszerek konvergenciája

Vizsgáljuk meg az $x^{(k+1)} = Bx^{(k)} + c$ iteráció által definiált $x^{(k)}$ sorozat konvergenciáját. Jelöljük az eredeti egyenletrendszerünk megoldásár x^* -al. Az $e_k = x^{(k)} - x^*$ eltérésre a következő állítás érvényes:

Tetszőleges $x^{(0)}$ kezdővektor, esetén a k -adik közelítés eltérése az x^* megoldástól $e_k = B^k e_0$

Következmény: Ha a B mátrix **nilpotens**, akkor $B^j e_0 = 0$, tehát az iterációs eljárás véges sok lépében megtalálja a megoldást.

A nilpotens azt jelenti, hogy van olyan j index, amire $B^j = 0$

Globális konvergencia: Akkor mondjuk, hogy egy iterációs sorozat globálisan konvergens, ha minden indulóvektorral ugyan azt a megoldást kapjuk.

Az $x^{(k+1)} = Bx^{(k)} + c$ iteráció akkor és csak akkor globálisan konvergens, ha $\rho(B) < 1$.

$\rho(B)$ a B mátrix **spektrálrádiusz**-át jelenti, ami a sajátértékeinek abszolút értékben vett maximuma.

Gauss-Seidel iteráció

Annyiban tér el a Jacobi-iterációtól, hogy az iterációs egyenletek jobb oldalán felhasználjuk az adott iterációban már megtalált közelítő értékeket.

Például ha $x_1^{(k+1)}$ már ismert, akkor a továbbiakban $x_1^{(k)}$ helyett $x_1^{(k+1)}$ -et használunk.

Ez valamivel gyorsítja a konverenciát.

Jacobi, Gauss-Seidel matlab kódok kellenek?

4.1.4. Mátrixok sajátértékeinek, és sajátvektorainak numerikus meghatározása

Legyen adott egy A négyzetes mátrix. Adjuk meg a λ számot, és az $x \neq 0$ vektort úgy, hogy $Ax = \lambda x$.

Ekkor λ az A sajátértéke, és x az A sajátvektora.

Baloldali sajátérték, sajátvektor: $y^T A = \lambda y^T$

Mátrix **spektruma**: Sajátértékeinek halmaza, jele: $\lambda(A)$.

Mátrix **spektrálrádiusa**: $\max |\lambda| : \lambda \in \lambda(A)$, jele: $\rho(A)$

Sajátvektor, sajátérték jelentősége

A sajátvektorok irányába eső vektorokat az A mátrix megnyújtja az adott sajátvektorhoz tartozó sajátértéknek megfelelően.

Sajátértékek, sajátvektorok nem egyértelműek

- Egységmátrixnak az 1 n -szeres sajátértéke
- Egy sajátvektorral együtt annak minden nem nulla számmal szorzottja is ugyanahhoz a sajátértékhez tartozó sajátvektora

Sajátértékek, sajátvektorok meghatározása

Megkaphatjuk a $(A - \lambda I)x = 0$ homogén lineáris egyenletrendszerből. Ennek pontosan akkor van nulla vektortól különböző megoldása, ha az $A - \lambda I$ mátrix szinguláris.

Emiatt a sajátértékeket leírja a $\det(A - \lambda I) = 0$ egyenlet. Ennek baloldalán levő n -edfokú polinomot az A mátrix **karakterisztikus polinomjának** nevezzük.

* Sajátértékek korlátai

Tetszőleges A mátrixra és bármely mátrixnormára $\rho(A) \leq \|A\|$.

A mátrix összes sajátértéke benne van a

Missing or unrecognized delimiter for \left

Hatványmódszer

A.K.A. von Mises vektoriterációja

A legnagyobb abszolútértékű sajátérték meghatározására szolgál.

Az algoritmus iterációs képlete:

$$y^k = Ax^k, x^{k+1} = \frac{y^k}{\|y^k\|}$$

Kiindulási vektor:

- $x^0 \neq 0$, és
- x^0 nem merőleges a legnagyobb abszolút értékű sajátértékhez tartozó sajátvektorra.

```

function lambda = hatv(A);
x = [rand(1) rand(1) rand(1)];
for i = 1:100
    y = A * x;
    lambda = y ./ x;
    r = (x' * y) / (x' * x);
    x = y / norm(y);
end

```

A λ komponensenként való osztás.

Az egyes iterációban kapott eredmények:

- r : Rayleigh-féle hánnyadossal kapott eredmény.
- x : Komponensenkénti becsléssel kapott eredmény.

4.2. 2. Érintő, szelő, és húr módszer, a konjugált gradiens eljárás. Lagrange interpoláció. Numberikus integrálás.

4.2.1. Érintő módszer

A.K.A. Newton-módszer

Tegyük fel, hogy az $f(x) = 0$ egyenlet x^* egyszeres, izolált zérushelyét akarjuk meghatározni, és hogy ennek a környezetében $f(x)$ differenciálható.

Válasszunk ki ebből egy x_0 kezdőértéket, majd képezzük az

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

iterációs sorozatot.

A módszer geometriai jelentése

Az aktuális x_k pontban meghatározzuk az $f(x)$ függvény és deriváltja értékét, ezekkel képezzük az adott ponthoz húzott érintőt, és következő iterációs pontnak azt határozzuk meg, amelyben az érintő zérushelye van.

Megoldás garantálása

Ha az $f(x)$ függvény kétszer folytonosan differenciálható az x^* zérushely egy környezetében, akkor van olyan pont, ahonnan indulva a Newton-módszer kvadratikusan konvergens sorozatot ad meg:

$$|x^* - x_{k+1}| \leq C|x^* - x_k|^2$$

valamely pozitív C konstanssal.

4.2.2. Szelő módszer

Legyen x^* az $f(x) = 0$ egyenlet egyszeres gyöke. Válasszunk alkalmas x_0 és x_1 kezdőértékeket, és ezekből kiindulva hajtsuk végre azt az iterációt, amit a következő képlet definiál:

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} = \frac{f(x_k)x_{k-1} - f(x_{k-1})x_k}{f(x_k) - f(x_{k-1})} \quad k = 1, 2, \dots$$

Valójában annyiban tér el a Newton-módszertől, hogy $f'(x_k)$ helyett annak közelítéseként a **numerikus derivált**,

$$\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

szerepel.

Igy tehát ez az eljárás csak egy $f(x)$ függvényt kiszámoló szubrutinra (függvényre) támaszkodik.

A módszer geometriai jelentése

x_{k+1} nem más, mint az $(x_k, f(x_k))$ és az $(x_{k-1}, f(x_{k-1}))$ pontokon átmenő egyenes és az x tengely metszéspontjának x koordinátája.

Tulajdonságok

- Szokás a szelő módszert olyan kezdőértékekkel indítani, amik **közrefogják** a x^* gyököt.
- Ha $f'(x^*) > 0$, és $f''(x^*) > 0$, akkor x^* -nál nagyobb, de ahhoz közeli kezdőértékekkel **szigorúan monoton konvergencia** érhető el.

4.2.3. Húr módszer

A szelő módszer a következő módosításokkal:

- A kezdeti x_0, x_1 pontokban az $f(x)$ függvény **ellentétes előjelű**.
- $f(x_{k+1})$ előjelétől függően a megelőző két pontból **azt választja** a következő iterációs lépéshoz, amelyikkel ez a **tulajdonság fennmarad**.

Például ha x_2 pozitív, és x_0 negatív, x_1 pozitív, akkor a következő iterációban x_2 mellett x_0 -t használja a módszer az x_1 helyett.

4.2.4. Konjugált gradiens eljárás

Optimalitálás elvein alapuló módszer.

Szimmetrikus pozitív definit mátrixú lineáris egyenletrendszer megoldására alkalmas.

Pontos aritmetikával ugyan véges sok lépéshoz megtalálná a megoldást, de a kerekítési hibák miatt mégis iterációs eljárásnak kell tekinteni.

Legyen A egy szimmetrikus, pozitív definit mátrix, akkor a

$$q(x) = \frac{1}{2}x^T Ax - x^T b$$

kvadratikus függvénynek egyetlen x^* minimumpontja van, és erre $Ax^* = b$ teljesül.

Azaz az $Ax = b$ lineáris egyenletrendszer megoldása ekvivalens a $q(x)$ kvadratikus függvény minimumpontjának meghatározásával.

A többdimenziós optimalizálási eljárások rendszerint az $x_{k+1} = x_k + \alpha s_k$ alakban keresik az új közelítő megoldást, ahol s_k egy keresési irány, és α a lépésköz.

Kvadratikus függvényekkel kapcsolatos összefüggések

1. A negatív gradiens a rezudiális vektor: $-\nabla q(x) = b - Ax = r$
2. Adott keresési irány mentén nem kell adaptív módon meghatározni a lépésközöt, mert az optimális α közvetlenül megadható. A keresési irány mentén ott lesz a célfüggvény minimális, ahol a rezudiális vektor merőleges s_k -ra. $\alpha = \frac{r_k^T s_k}{s_k^T A s_k}$

A módszer

Adott x_0 kezdőpontra legyen $s_0 = r_0 = b - Ax_0$, és iteráljuk $k = 1, 2, \dots$ értékekre az alábbi lépésekkel, amíg a megállási feltételek nem teljesülnek:

1. $\alpha_k = \frac{r_k^T r_k}{s_k^T A s_k}$: A **lépéshossz** meghatározása
2. $x_{k+1} = x_k + \alpha_k s_k$: Iterált **közelítő megoldás**
3. $r_{k+1} = r_k - \alpha_k A s_k$: Új **rezudiális vektor**
4. $\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$: Segédváltozó
5. $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$: Új **keresési irány**

Korábbi gradiensmódszerek esetén egyszerűen a negatív gradienst követik minden iterációs lépéshoz, de felismerték hogy ez a meredek falú, enyhén lejtő völgyzserű függvények esetén szükségtelenül sok iterációs lépést eredményez a völgy két oldalán való oda-vissza ugrálás miatt. A kisebb meredekséggel rendelkező irányban viszont lényegesen gyorsabban lehetett volna haladni. A konjugált gradiens módszer a lépésekben megfelelő irányváltottatással kiküszöböli ezt a hibát.

A megállási feltétel szokás szerint az, hogy a felhasználó előírja, hogy az utolsó néhány iterált közelítés eltérése és a lineáris egyenletrendszer két oldala különbsége normája ezekben a pontokban adott kis pozitív értékekhez csak köszít meg rf -et ne alatt maradjanak.

Matlabban

```
function x = kg(A, b, x);
s = b - A * x;
r = s;
for k = 1:20
    a = (r' * r) = (s' * A * s);
    x = x + a * s;
    rr = r - a * A * s;
    s = rr + s * ((rr' * rr) / (r' * r));
    r = rr;
end
```

Az rr valójában r_{k+1} , csak mivel s kiszámolásához r_k -ra is szükség van, így csak az után adjuk értékül r -nek (rr -t).

4.2.5. Lagrange interpoláció

Interpoláció: Az a feladat, amikor adott $(x_i, y_i), i = 1, 2, \dots, m$ pontsorozaton állítunk elő egy függvényt, amely egy adott függvényosztályba tartozik, és minden ponton átmegy.

Azaz x_i helyeken a megfelelő y_i értékeket vegye fel a függvény.

Ha a keresett $f(x)$ függvény polinom, akkor **polinominterpolációról** beszélünk.

Interpoláció másik jelentése: A közelítő függvény segítségével az eredeti $f(x)$ függvény értékét egy olyan \hat{x} pontban becsüljük az interpoláló $p(x)$ polinom $p(\hat{x})$ helyettesítési értékével, amelyre:

$$\hat{x} \in [\min(x_1, x_2, \dots, x_m), \max(x_1, x_2, \dots, x_m)]$$

Ezzel szemben ha

$$\hat{x} \notin [\min(x_1, x_2, \dots, x_m), \max(x_1, x_2, \dots, x_m)]$$

teljesül, akkor **extrapolációról** van szó.

Spline interpoláció: Több alacsony fokszámú polinomból összerakott függvényt keres úgy, hogy az adott ponton való áthaladás megkövetelése mellett az is elvárás, hogy a szomszédos polinomok a csatlakozási pontokban **előírt derivált értékeket** vegyenek föl.

Polinomok fokszáma

Polinom interpoláció esetén a polinom fokszáma, n egyenlő $m - 1$ -el.

Spline alkalmazásakor a fokszám lényegesen kisebb, mint az alappontok száma.

Amennyiben egy olyan polinomot illesztünk, amelynek fokszáma kisebb, mint $m - 1$, akkor **görbeillesztésről** beszélünk.

Görbeillesztéskor a polinom persze nem feltétlen megy át minden alapponton.

Lagrange interpoláció

Lagrange interpolációkor feltesszük, hogy az alappontok különbözök, de ez nem egy túl erős feltétel, hiszen nem is lehet azonos x koordinátán két különböző y értéket érinteni egy függvénykel.

A Lagrange interpoláció az interpoláló polinomokat

$$p_n(x) = \sum_{i=0}^n f(x_i)L_i(x)$$

alakban adja meg, ahol

$$L_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_1) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_n)}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)}$$

Legyenek adottak az x_0, \dots, x_n páronként különböző alappontok. Ekkor az $f(x_i)$, $i = 0, 1, \dots, n$ függvényértékekhez egyértelműen létezik olyan legfeljebb n -edfokú interpoláló polinom, amely megegyezik a Lagrange interpolációs polinommal.

Matlabban

```
function [C, L] = lagran(X, Y)
w = length(X);
n = w - 1;
L = zeros(w, w);
for k = 1:n+1
    V = 1;
    for j = 1:n+1
        if k ~= j
            V = conv(V, poly(X(j))) / (X(k) - X(j));
        end
    end
end
C = Y * L;
```

`conv(u, v)` : Konvolúció, `u` a maszk, amit keresztül tol `v`-n.

`poly(A)` : Karakterisztikus polinom-ot számol ki mátrixból, vagy sajátértékekből.

`~=` : Nem-egyenlő operátor.

4.2.6. Numerikus integrálás

A kvadratúra a numerikus integrálás szinonimája, amikor a

$$\int_a^b f(x) dx = F(b) - F(a)$$

határozott integrál közelítése a feladat. Itt $F(x)$ az $f(x)$ integrálandó függvény primitív függvénye. Ez utóbbi nem minden esetben áll rendelkezésre, sőt sokszor nem is elemi függvény, nem adható meg zárt alakban.

A határozott integrálokat szokás

$$\int_a^b f(x) dx \approx Q_n(f) = \sum_{i=1}^n w_i f(x_i)$$

alakban közelíteni, ahol $Q_n(f)$ -et **kvadratúra-formulának** nevezük.

Általában feltesszük, hogy $x_i \in [a, b]$ teljesül az x_i alappontokra, és ezek páronként különbözök.

A w_i számokat **súlyoknak** hívjuk.

Integrál, és kvadratúra-formula tulajdonságai

$$\$ \int_a^b f(x) + g(x) dx = \int_a^b f(x) dx + \int_a^b g(x) dx$$

\

$$Q_n(f+g) = \sum_{i=1}^n w_i (f(x_i) + g(x_i)) = \sum_{i=1}^n w_i f(x_i) + \sum_{i=1}^n w_i g(x_i) = Q_n(f) + Q_n(g)$$

\

$$\int_a^b \alpha f(x) dx = \alpha \int_a^b f(x) dx$$

\

$$Q_n(\alpha f) = \sum_{i=1}^n w_i \alpha f(x_i) = \alpha \sum_{i=1}^n w_i f(x_i) = \alpha Q_n(f)$$

\

$$\int_a^b f(x) dx = \int_a^{z_1} f(x) dx + \dots + \int_{z_{m-1}}^{z_m} f(x) dx + \int_{z_m}^b f(x) dx \$$$

Kvadratúra-formula képlethibája

$$R_n(f) = \int_a^b f(x) dx - Q_n(f)$$

Ha $R_n(f) = 0$, akkor a **kvadratúra-formula pontos** $f(x)$ -re.

Kvadratúra-formula pontossági rendje az r természetes szám, ha az pontos az $1, x, x^2, \dots, x^r$ hatványfüggvényekre, azaz $R_n(x^k) = 0$ minden $q \leq k \leq r$ -re, de nem pontos x^{r+1} -re.

A Q_n, n alappontos kvadratúra-formula rendje legfeljebb $2n - 1$ lehet.

Interpolációs kvadratúra-formulák

Azt mondjuk, hogy $Q_n(f) = \sum_{i=1}^n w_i f(x_i)$ egy interpolációs kvadratúra-formula, ha az előáll az alappontokra felírt Lagrange polinom integrálásával:

$$\int_a^b f(x) dx \approx \int_a^b p_{n-1}(x) dx = \int_a^b \sum_{i=1}^n f(x_i) L_i(x) dx = \sum_{i=1}^n f(x_i) \int_a^b L_i(x) dx$$

$$\text{ahonnan } w_i = \int_a^b L_i(x) dx.$$

Az alappont az interpolációra, és a kvadratúrára is vonatkozik.

Minden n alapontra épülő Q_n interpolációs kvadratúra-formula rendje legalább $n - 1$.

Ha egy Q_n kvadratúra-formula rendje legalább $n - 1$, akkor az interpolációs kvadratúra-formula.

Véges differenciák

Ekvidisztáns alappontokat adunk meg.

Szomszédos alappontok **távolsága állandó**: $h = x_{i+1} - x_i$.

Az interpolációs alappontok: $x_i = x_0 + ih, i = 0, \dots, n - 1$

Az adott x_k alappontokhoz és $f_k = f(x_k)$ függvény értékekhez tartozó $\Delta^i f_k$ *i-edrendű véges differenciákat* a következő kettős rekurzióval definiáljuk:

$$\Delta^0 f_k = f_k \quad \Delta^i f_k = \Delta^{i-1} f_{k+1} - \Delta^{i-1} f_k$$

Természetes számokra értelmezett binomiális együtthatók általanostásaként vezessük be a:

$$\binom{t}{j} = \frac{t(t-1)\dots(t-j+1)}{j!}$$

jelölést a $t = \frac{x-x_0}{h}$ transzformációhoz.

A véges differenciákkal felírt Lagrange interpolációs polinom:

$$p_{n-1}(x_0 + th) = f_0 + \binom{t}{1} \Delta f_0 + \binom{t}{2} \Delta^2 f_0 + \dots + \binom{t}{n-1} \Delta^{n-1} f_0 = \sum_{i=0}^{n-1} \binom{t}{i} \Delta^i f_0$$

Newton-Cotes formulák

Az interpolációs kvadratúra-formulák egy régi osztálya.

Ekválensztszáns alapontokat használnak.

Azaz a szomszédosak közt ugyanannyi a távolság.

Ha az integrál határai szerepelnek az alapontok között, akkor zárt-, ha a határok nem alapontok, akkor nyitott formuláról beszélünk.

Zárt formulákra összefüggések

$$h = \frac{b-a}{n-1}, a = x_0, b = x_{n-1}, x_i = x_0 + ih \quad 0 \leq i \leq n-1$$

Nyitott formulákra összefüggések

$$h = \frac{b-a}{n+1}, a = x_0 - h, b = x_{n-1} + h, x_i = x_0 + ih \quad 0 \leq i \leq n-1$$

n -edik Newton-Cotes formula

$t = \frac{x-x_0}{h}$ új változó mellett az n -edig Newton-Cotes formula:

$$\int_a^b p_{n-1}(x_0 + th) dx = \int_a^b \sum_{i=0}^{n-1} \binom{t}{i} \Delta^i f_0 dx = \sum_{i=0}^{n-1} \Delta^i f_0 \int_a^b \binom{t}{i} dx$$

A t lényegében az adott változó eltolását fejezi ki az x_0 -tól.

A Δ^i véges differenciál.

Ha a formula zárt:

$$\sum_{i=0}^{n-1} \Delta^i f_0 \int_a^b \binom{t}{i} dx = h \sum_{i=0}^{n-1} \Delta^i f_0 \int_0^{n-1} \binom{t}{i} dt$$

Ha a formula nyitott:

$$\sum_{i=0}^{n-1} \Delta^i f_0 \int_a^b \binom{t}{i} dx = h \sum_{i=0}^{n-1} \Delta^i f_0 \int_{-1}^n \binom{t}{i} dt$$

Első négy zárt Newton-Cotes formula

1. $\int_{x_0}^{x_1} f(x) dx \approx \frac{h}{2}(f_0 + f_1)$: **Trapéz szabály**

2. $\int_{x_0}^{x_2} f(x) dx \approx \frac{h}{3}(f_0 + 4f_1 + f_2)$: **Simpson-szabály**

3. $\int_{x_0}^{x_3} f(x) dx \approx \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3)$: **Simpson $\frac{3}{8}$ -os szabálya**

4. $\int_{x_0}^{x_4} f(x) dx \approx \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$: **Bool-szabály**

Matlabban

```
function f = fxlog(x)
f = x .* log(x);
```

A fentebbi függvény az $x \log(x)$ függvényértéket kiszámoló eljárás, ennek numerikus integrálása a $[2, 4]$ intervallumon:

```
quad(@fxlog, 2, 4);
```

Eredményül 6.7041-et logol az interpreter.

5. Logika és informatikai alkalmazásai

5.1. 1. Normálformák az ítéletkalkulusban, Boole-függvények teljes rendszerei. Következtető módszerek: Hilbert-kalkulus és rezolúció, ezek helyessége és teljessége.

5.1.1. Normálformák az ítéletkalkulusban

* Ítéletkalkulus-beli formulák

- minden változó, és minden logikai konstans formula
- Ha F formula, akkor $(\neg F)$ is formula
- Ha F és G formulák, akkor $(F \wedge G)$, $(F \vee G)$, $(F \leftrightarrow G)$ is formulák
- Más formula nincs

Konjunktív normálforma (CNF)

- **Literál:** CNF legkisebb eleme, lehet egy változó, vagy egy változó negáltja.

- **Negatív literál:** Ha egy változó negáltja alkotja.
- **Pozitív literál:** Ha egy nem negált változó alkotja.

- **Klóz:** Véges sok literál diszjunkciója (vagyolása).

- **Egységlklóz:** 1db változóból álló klóz.
- **Üres klóz:** 0db változóból álló klóz.
 - Értéke minden értékadás mellett hamis.
 - Jele: \square

- **CNF: Klózok konjunkciója** (éselése).

- **Üres CNF:** 0db klózt tartalmaz.
 - Értéke minden értékadás mellett igaz.
 - Jele: \emptyset

Üres klóz az inputban jellemzően nincs, de az algoritmusok generálhatnak.

Minden formula ekvivalens CNF alakra hozható.

1. \rightarrow és \leftrightarrow konnektívák eliminálása. ($A \rightarrow B \equiv \neg A \vee B$ -vel)
2. \neg -k bevitelle változók mellé deMorgan azonosságokkal.
3. \vee jelek bevitelle a \wedge jelek alá disztributivitás alkalmazásával.

Disztributivitás szabályai: $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H) > (F \wedge G) \vee H \equiv (F \vee H) \wedge (G \vee H)$

A "konnektíva" azt jelenti, hogy az operátor formulákat vár (köt össze), nem változókat (az a Boole-függvény).

CNF-ek reprezentálása

Nem stringként, hanem:

- egy klózt a benne szereplő literálok halmazaként,
- egy CNF-et pedig a klózainak halmazaként.

Ezt azért tehetjük meg, mert sem a vagyolás, sem az éselés esetén nem számít a sorrend, illetve az érintett változók multiplicitása sem, pl. $(p \vee p) \wedge (q \vee q)$ ugyan az, mint $q \vee p$ (sorrend fordult, multiplicitás eltűnt).

Diszjunktív normálforma

Ugyan az, mint a CNF, csak nem "vagyolások éselése", hanem "éselések vagyolása".

Negációs normálforma

Ha \neg csak változó előtt szerepel, és \neg -en kívül csak \vee és \wedge szerepel.

Ilyet kapunk ha a CNF-re hozást csak a 2. lépésig csináljuk.

5.1.2. Boole-függvények teljes rendszerei

* Boole függvény (n -változós)

Bitvektort egy bitbe képező függvény: $f : \{0, 1\}^n \rightarrow \{0, 1\}$

f egy n -változós függvény jelölése: f/n

A \neg unáris, egyváltozós Boole-függvény

A többi 4 megadható 4 soros igazságtáblával.

* Indukált Boole-függvény

Ha az F formulában csak a p_1, \dots, p_n változók szerepelnek, akkor F indukál egy n -változós Boole-függvényt, melyet szintén F -el jelölünk:

- $p_i(x_1, \dots, x_n) := x_i$ (ez projekció / tömbelem kiválasztás)
- $(\neg F)(x_1, \dots, x_n) := \neg(F(x_1, \dots, x_n))$
- $(F \vee G)(x_1, \dots, x_n) := F(x_1, \dots, x_n) \vee G(x_1, \dots, x_n)$
- ...

A Boole-függvények átadott bitvektor tulajdonképpen a formula egy értékkadása. A visszaadott bit pedig a formula kiértékelésének eredménye.

* Boole-függvények megszorítása

Legyen f/n a Boole-függvény, $n > 0$. Ha $b \in \{0, 1\}$ igazságérték, úgy $f|_{x_n=b}$ jelöli azt az $(n - 1)$ változós Boole-függvényt, melyet úgy kapunk, hogy f inputjában x_n értékét rögzítjük b -re.

Azaz: $\$f|\{x_n = b\}(x_1, \dots, x_{n-1}) := f(x_1, \dots, x_{n-1}, b)\$$

Például:

- $\vee|_{x_2=1}$ a konstans 1 függvény.
- $\wedge|_{x_2=0}$ a konstans 0 függvény.

Bármenyik koordinátát lehet rögzíteni, nem csak az utolsót.

Teljes rendszerek

Boole-függvények egy H rendszere teljes, vagy adekvált, ha minden n -változós Boole-függvény előáll

- a projekcióból
- és H elemeiből
- alkalmas kompozícióval.

Kompozíció

Ha f/n és $g_1/k, \dots, g_n/k$ Boole-függvények, akkor az $f \circ (g_1, \dots, g_n)$ az a k -változós Boole-függvény, melyre: $(f \circ (g_1, \dots, g_n))(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$

Azaz egy függvényt úgy hívunk meg, hogy az inputjai függvényhívások eredményei.

Shannon expanzió

$\$f(x_1, \dots, x_n) = (x_n \land f|\{x_n = 1\}(x_1, \dots, x_{n-1})) \vee (\neg x_n \land f|\{x_n = 0\}(x_1, \dots, x_{n-1}))\$$

Lényegében ezzel azt írtuk le, hogy az x_n értéke mellett vagy úgy igaz a formula, hogy $x_n = 1$, vagy úgy, hogy $x_n = 0$.

Ennek a következménye: minden Boole-függvény előáll a projekciók, és a \neg, \wedge, \vee alkalmas kompozíciójaként. (Hiszen az előző összefüggésben csak ezeket használjuk fel, és ez ismételhető amíg nem kötöttünk le minden változót.)

Ezt úgy is lehet mondani, hogy a \neg, \vee, \wedge **rendszer teljes**.

Ebből az is következik, hogy minden Boole-függvény indukálható olyan formulával, melyben csak a \neg, \wedge, \vee konnektívák szerepelnek.

További teljes rendszerek

- \neg, \wedge : Mivel $x_1 \wedge x_1 = \neg(\neg x_1 \vee \neg x_2)$
- \rightarrow, \downarrow // Hilbert rendszere
- NAND
- NOR

A NAND-on, és NOR-on kívül nincs másik olyan $f/2$ Boole-függvény, ami egyedül is teljes rendszert alkot.

5.1.3. Hilbert rendszere

Egy input Σ formulahalmaz összes következményét (és csak a következményeket) lehet vele levezetni.

Az ítéletváltozókon kívül ebben a rendszerben csak a \rightarrow konnektívát, és a \downarrow logikai konstanst használhatjuk.

Minden formula ilyen alakra hozható, mert \rightarrow, \downarrow teljes rendszer.

A Hilbert rendszer axiómái

- $(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$
- $F \rightarrow (G \rightarrow F)$
- $((F \rightarrow \downarrow) \rightarrow \downarrow) \rightarrow F$

Ezen a formulák **tautológiák**. Azaz minden értékkadás mellett igazak.

Az axiómák példányai

A 3 axióma egy **példánya**: valamelyik axiómában szereplő F, G, H helyére **tetszőleges formulát** írunk.

Ennek van jelölése is: Ha F egy formula, melyben a p_1, \dots, p_n változók szerepelnek, és F_1, \dots, F_n formulák, akkor $F[p_1/F_1, \dots, p_n/F_n]$ jelöli azt a formulát, melyet úgy kapunk F -ből, hogy benne minden p_i helyére az F_i formulát írjuk.

Leválasztási következtetés, vagy modus ponens

$$F, F \rightarrow G \models G$$

Ha F -et, és $F \rightarrow G$ -t már levezettük, azaz az eredeti formulánknak ők logikai következményei, akkor felvehetjük G -t is, mert ő is logikai következmény.

Levezetés Hilbert rendszerében

Legyen Σ formulák egy halmaza, F pedig egy formula. Azt mondjuk, hogy F levezethető Σ -ból **Hilbert rendszerében**, jelben $\Sigma \vdash F$, ha van olyan F_1, F_2, \dots, F_n formula-sorozat, melynek minden eleme

- Σ -beli vagy
- **axiómapéldány** vagy
- előáll két korábbiból **modus ponenssel**

és melyre $F_n = F$. (Ha Σ üres, akkor $\emptyset \vdash F$ helyett $\vdash F$ -et is írhatunk)

Helyesség, teljesség

Tautológia példányai is tautológiák

Tehát a Hilbert-rendszer **axióma-példányai tautológiák**.

Ez egy általánosabb összefüggés következménye:

Legyenek az F formulában szereplő változók p_1, \dots, p_n , és F_1, \dots, F_n további formulák (melyekben más változók is előfordulhatnak). Legyen \mathcal{A} egy tetszőleges értékkadás. Definiáljuk \mathcal{B} értékkadást a következőképpen: $\mathcal{B}(p_i) := \mathcal{A}(F_i)$ (a p_i értéke \mathcal{B} -ben legyen az az érték, ami F_i értéke \mathcal{A} -ban) Ekkor: $\mathcal{B}(F) = \mathcal{A}(F[p_1/F_1, \dots, p_n/F_n])$

Helyesség

Ha $\Sigma \vdash F$, akkor $\Sigma \models F$.

Azaz, ha egy formulát le lehet vezetni Σ -ból Hilbert rendszerében, akkor az következménye is Σ -nak.

Bizonyítás:

- Legyen F_1, \dots, F_n egy Σ fölötti levezetése F -nek. Teljes indukcióval megmutatjuk, hogy minden i -re $\Sigma \models F_i$
- Ha $F_i \in \Sigma$, akkor $\Sigma \models F_i$
- Ha F_i axiómapéldány, akkor $\emptyset \vdash F_i$ (tautológiák minden elméletben szerepelnek, és az axiómapéldányok a korábbi szabály miatt tautológiák), így a monotonitás miatt $\Sigma \models F_i$ is igaz (nyílván ha az \emptyset -nak következménye, akkor egy bővebb halmaznak, a Σ -nak is).
- Ha pedig $F_i = MP(F_j, F_k)$ a $j, k < i$ indexekre, akkor
 - Az indukciós feltevés szerint $\Sigma \models F_j$ és $\Sigma \models F_k$ (feltételezzük, hogy a korábban felvett formulák már logikai következmények)
 - Tehát $\Sigma \models F_j, F_k$
 - MP def miatt $F_k = F_j \rightarrow F_i : \Sigma \models F_j, F_j \rightarrow F_i$
 - A leválasztási következtetés: $F_j, F_j \rightarrow F_i \models F_i$
 - A tranzitivitás miatt tehát $\Sigma \models F_i$ (tranzitivitást kihasználjuk, mivel $\Sigma \models F_j, F_j \rightarrow F_i \models F_i$)

Így a Hilbert-rendszer egy helyes következtető rendszer.

Teljesség

Dedukciós tétel: Tetszőleges Σ formulahalmazra, és F, G formulákra $\Sigma \vdash (F \rightarrow G) \Leftrightarrow \Sigma \cup F \vdash G$

H-konzisztens halmazok: Egy Σ formulahalmazt H-konzisztensnek nevezünk, ha **nem igaz**, hogy $\Sigma \vdash \downarrow$

Csak simán az, hogy **konzisztens** formulahalmaz, az azt jelenti, hogy **kielégíthető**.

Ekvikalens állítások tetszőleges Σ formulahalmazra:

- Van olyan F formula, melyre $\Sigma \vdash F$ és $\Sigma \vdash (F \rightarrow \downarrow)$ is igaz.
- Σ nem H-konzisztens.
- $\Sigma \vdash F$ minden F formulára.

Maximális H-konzisztens halmazok: Egy Σ formulahalmazt maximális H-konzisztensnek nevezünk, ha

- Σ H-konzisztens, és
- minden $F \notin \Sigma$ -ra $\Sigma \cup F$ már nem H-konzisztens.

Minden Σ H-konzisztens halmazhoz van $\Sigma' \supseteq \Sigma$ maximális H-konzisztens halmaz. "A halmazt fel lehet fújni."

Ha Σ maximális H-konzisztens halmaz, akkor tetszőleges F formulára vagy $F \in \Sigma$, vagy $(F \rightarrow \downarrow) \in \Sigma$, de nem mindkettő.

Azaz minden **formulát, vagy a negáltját** tartalmazzák, de csak az egyiket.

Tetszőleges Σ formulahalmaz pontosan akkor kielégíthető, ha H-konzisztens.

A Hilbert-rendszer helyessége és teljessége:

Ezt kell belátni: $\Sigma \vDash F \Leftrightarrow \Sigma \vdash F$

Most itt egyszerre van belátva mindenktől, de a helyesség fentebbi alapján külön is belátható.

Sorban minden ekvivalenciát tovább fejtünk ekvivalencia mentén:

$$\Sigma \vDash F \Leftrightarrow \Sigma \cup F \rightarrow \downarrow \vDash \downarrow$$

Ennek az alapja egy téTEL: $\Sigma \vDash F$ pontosan akkor igaz, ha $\Sigma \cup \neg F$ kielégíthetetlen. Ez van írva Hilbert-rendszerében.

$$\Leftrightarrow \Sigma \cup F \rightarrow \downarrow \vdash \downarrow$$

Itt a bal oldal azt jelenti, hogy az a halmaz kielégíthetetlen (az összeuniózott). Akkor ez a halmaz nem H-konzisztens, és ekkor levezethető belőle Hilbert-rendszerében az azonosan hamis.

$$\Leftrightarrow \Sigma \vdash (F \rightarrow \downarrow) \rightarrow \downarrow$$

Dedukciós téTEL alkalmazása.

$$\Leftrightarrow \Sigma \vdash F$$

Ennek a legutolsó lépésnek a belátása kicsit nehezebb:

- Egyik irány: $(\Sigma \vdash (F \rightarrow \downarrow) \rightarrow \downarrow) \rightarrow (\Sigma \vdash F)$
 - $((F \rightarrow \downarrow) \rightarrow \downarrow) \rightarrow F$
 - A 3. axióma példányosítása
 - $\Sigma \vdash F$
 - Modus ponens alkalmazása
- Másik irány: $(\Sigma \vdash F) \rightarrow (\Sigma \vdash (F \rightarrow \downarrow) \rightarrow \downarrow)$
 - $((F \rightarrow \downarrow) \rightarrow (F \rightarrow \downarrow)) \rightarrow (((F \rightarrow \downarrow) \rightarrow F) \rightarrow ((F \rightarrow \downarrow) \rightarrow \downarrow))$
 - Az 1. axióma példányosítása
 - $(F \rightarrow \downarrow) \rightarrow (F \rightarrow \downarrow)$
 - Ilyet ér felvenni, hiszen $G \rightarrow G$ alakú, és erre volt példa, hogy az ilyenek az \emptyset -nak is logikai következményei.
 - $((F \rightarrow \downarrow) \rightarrow F) \rightarrow ((F \rightarrow \downarrow) \rightarrow \downarrow)$
 - Előző kettő MP-el
 - $F \rightarrow ((F \rightarrow \downarrow) \rightarrow F)$
 - A 2. axióma példánya
 - $(F \rightarrow \downarrow) \rightarrow F$
 - Előző formula, és feltevés miatt F MP-e
 - $(F \rightarrow \downarrow) \rightarrow \downarrow$

- Előző, és az előtt kettővel levő formulák MP-je

Az ekvivalenciák mentén beláttuk, hogy **Hilbert-rendszer helyes, és teljes**. Azaz tetszőleges Σ halmzból Hilbert rendszerében **pontosan Σ következményei vezethetők le**.

5.1.4. Rezolúció

Rezolúciós következtetés

$$F \vee G, \neg F \vee H \models G \vee H$$

Nyílván, mert ha az F igaz, akkor H igaz kell, hogy legyen, ha F hamis, akkor G igaz kell, hogy legyen.

Emlékeztető: Logikai következmény jelentése: Bármely értékkedás mellett ha a bal oldal igaz (jelen esetben bal oldalon minden igaz, mert egy halmaz áll ott), akkor a jobb is.

Rezolvens

Ha C és D klózok, $p \in C$ és $\neg p \in D$, akkor C és D (p menti) rezolvense a $(C - p) \cup (D - \neg p)$ klóz.

Egy új, harmadik klóz keletkezik.

Rezolúciós algoritmus

Input: Klózok Σ halmaza.

Output: Kielégíthetetlen-e Σ ?

Algoritmus: Listát vezetünk klózokról. Egy klózt felveszünk, ha

- Σ -beli, vagy
- két, a listán már szereplő klóz rezolvense.

Ha az \square üres klóz rákerül a listára, a Σ kielégíthetetlen.

Ha már nem tudunk új klózt felvenni és \square nincs köztük, Σ kielégíthető.

Kielégíthető formulahalmazra nem feltétlen áll meg az algoritmus. Ezért kérdezzük inkább, hogy kielégíthetetlen-e.

Egyszerre több literál mentén nem ér rezolválni!!

Helyesség

Ha az algoritmus "kielégíthetetlen" válasszal áll meg, akkor az input Σ valóban kielégíthetetlen.

Azt látjuk be, hogy minden klóz, ami a listára kerül, az logikai következménye Σ -nak. Ezt indukcióval tesszük: ha a C klóz n . elemként kerül a listára, akkor:

- Ha $C \in \Sigma$, akkor $\Sigma \models C$ minden teljesül.
- Ha C a korábban felvett C_1 és C_2 klózok **rezolvense**, akkor
 - indukciós feltevés szerint $\Sigma \models C_1$ és $\Sigma \models C_2$
 - tehát $\Sigma \models C_1, C_2$ (nyílván, összevagyonl ér őket)
 - a **rezolúciós következtetés** szerint pedig $C_1, C_2 \models C$ (rezolúciós rész eleje) (onnan tudjuk, hogy C a rezolvense C_1 -nek, és C_2 -nek, hogy ez a feltevés ebben a második esetben)
 - így a \models tranzitivitása miatt $\Sigma \models C$.

Így tehát ha $\Sigma \models \square$, akkor Σ valóban kielégíthetetlen, mert kövezetménye a *hamis*. ($Mod(\square) = \emptyset$, nincs őt kielégítő értékkedás)

Teljesség

Ha Σ kielégíthetetlen, akkor az algoritmus minden a "kielégíthetetlen" válasszal áll meg.

Minimális kielégíthetetlen részhalmaz: A Σ kielégíthetetlen klózhalmaznak a $\Sigma' \subseteq \Sigma$ egy **minimális kielégíthetetlen részhalmaza**, ha Σ' is kielégíthetetlen, de Σ' bármelyik valódi részhalmaza már kielégíthető.

Lineáris rezolúció: **Input:** Σ klózhalmaz. **Output:** Kielégíthetetlen-e Σ ? **Algoritmus:** Listát vezetünk klózokról:

- Az első lépésben felvehetjük Σ **bármelyik elemét**, ez lesz a levezetés **bázisa**.
- minden további lépésben felvehetjük az előző lépésben felvett klóznak, és egy vagy már a listán szereplő, vagy Σ -beli klóznak a rezolvensét. Ezt a másik klózt hívjuk ennek a lépésnek az **oldalklózánnak**.

Lineáris rezolúció teljessége:

Ha Σ kielégíthetetlen, és $C \in \Sigma$ benne van a Σ egy **minimális kielégíthetetlen részhalmazában**, akkor Σ -ból levezethető az üres klóz olyan **lineáris rezolúciós** levezetéssel, melynek **bázisa** C .

Bizonyítás:

Az állítást a Σ -beli változók n száma szerinti indukcióval látjuk be.

- Ha $n = 0$, azaz Σ -ban nincs változó, akkor vagy $\Sigma = \square$ (ekkor nincsen benne klóz), vagy $\Sigma = \square$ (ekkor van benne egy klóz, az üres klóz)
 - A kettő közül $\Sigma = \square$ a kielégíthetetlen.
 - Ennek \square az egyetlen eleme, ez egy minimális kielégíthetetlen részhalmazának is eleme.
 - Ha felvesszük bázisként, már le is vezettük az üres klózt.
- Ha $n > 0$, akkor vegyük egy C klózt, mely szerepel Σ egy minimális kielégíthetetlen részhalmazában. Legyen ez a részhalmaz Σ' .
 - Ha $C = \square$, kész vagyunk: vegyük fel bázisnak.
 - Különben legyen $l \in C$ egy C -beli literál.
 - Vegyük észre: minimális kielégíthetetlen részhalmazban nincs pure literál, hiszen ha l pure literál lenne, akkor Σ -nak egy valódi részhalmaza $\Sigma'|_{l=1}$ is kielégíthetetlen lenne. Tehát Σ' -ben \bar{l} is szerepel valahol.
- Vegyük a $\Sigma'|_{l=0}$ klózhalmazokat.
- Mivel Σ' kielégíthetetlen, ezek is azok.
- Bennük csak legfeljebb $n - 1$ változó szerepel (mert l változója kiesik), így alkalmazhatjuk az indukciós feltevést.
- A $\Sigma'|_{l=0}$ klózhalmaznak $C - l$ is eleme, sőt egy minimális kielégíthetetlen részhalmazának is eleme (mert különben $\Sigma' - C$ is kielégíthetetlen lenne).
- Tehát $\Sigma'|_{l=0}$ -ból az indukciós feltevés szerint van \square -nak egy C_1, C_2, \dots, C_m lineáris rezolúciós levezetése, melynek $C_1 = C - l$ a bázisa.
- "Visszaemelve" a $\Sigma|_{l=0}$ cáfolatot Σ' fölötti levezetéssé, az új levezetésben minden klózba bekerül az l literál.
- Ez igaz a bázisra, és minden lépésben az eredeti C_1 és C_2 klózok rezolvense helyett a $C \cup l$ és C_2 vagy $C_2 \cup l$ klózok rezolvensét kapjuk, ami rezolvens, plusz l
- Tehát a konstrukciónak a végén az l egységekben jár a lineáris rezolúciós levezetés.
- Mivel Σ' minimális kielégíthetetlen, kell legyen benne olyan C klóz is, mely \bar{l} -t tartalmazza.
- Akkor $\Sigma'|_{l=1}$ -nek egy minimális kielégíthetetlen részhalmazában szerepel $C - \bar{l}$
- Ebből a klózból indulva az indukciós feltevés szerint van $\Sigma'|_{l=1}$ -nek lineáris rezolúciós cáfolata
- Az előző fázisban kapott l egységeket tudjuk rezolválni ezzel a C klózzal, tehát a $\Sigma'|_{l=1}$ cáfolatát "fel tudjuk emelni" Σ' fölötti levezetéssé.
- A felemelt levezetés végén vagy \square -t, vagy \bar{l} -t kapunk. Utóbbi esetben még egyszer rezolválunk l -lel mint oldalklózzal, és kész vagyunk

5.2. 2. Normálformák az elsőrendű logikában. Egyesítési algoritmus. Következtető módszerek: Alap rezolúció, és elsőrendű rezolúció, ezek helyessége és teljessége.

5.2.1. * Elsőrendű logika alapfogalmak

Függvényjelek, predikátumjelek **aritása / rangja**: Hány változósak

Alapterm: Olyan term, amiben nincs változó

Term:

- Változók
- f/n függvényjel, és t_1, \dots, t_n termek esetén $f(t_1, \dots, t_n)$ is term

* Struktúra

Egy $\mathcal{A} = (A, I, \phi)$ hármas, ahol

- A egy nemüres halmaz, az **univerzum**
- A változók ebből vehetnek fel értékeket
- ϕ a változóknak egy "default" **értékkadása**, minden x változóhoz egy $\phi(x) \in A$ objektumot rendel
- I az **interpretációs függvény**, ez rendel a függvény és predikárumjelekhez szemantikát, "értelemet" az adott struktúrában:
 - ha f/n függvényjel, akkor $I(f)$ egy $A^n \rightarrow A$ függvény
 - Objektum(ok)ból objektumot csinál
 - ha p/n predikátumjel, akkor $I(p)$ egy $A^n \rightarrow \{0, 1\}$ predikátum

- Objektum(ok)ból igazságértéket csinál

Az = bináris predikátumjelet minden struktúrában ténylegesen az egyenlőséggel kell interpretálnunk!

* Term kiértékelése

- Ha $t = x$ változó, akkor $\mathcal{A}(t) := \phi(x)$
- Ha $t = f(t_1, \dots, t_n)$, akkor $\mathcal{A}(t) := I(f)(\mathcal{A}(t_1), \dots, \mathcal{A}(t_n))$

Emlékeztető, a term lehet egy változó, vagy egy függvény, aminek paramétereit termek.

Egy struktúra megadásakor elég csak azon változókat specifikálni, amik ténylegesen használtak (szerepelnek a termekben).

* Formulák kiértékelése

$\mathcal{A}_{[x \mapsto a]}$: Az a struktúra, ami az \mathcal{A} struktúrát úgy változtatja, hogy benne a $\phi(x) := a$

Ha F formula, $\mathcal{A} = (A, I, \phi)$ pedig struktúra, akkor az F értéke \mathcal{A} -ban egy igazságérték, amit $\mathcal{A}(F)$ jelöl, és az F felépítése szerinti indukcióval adunk meg:

- Logikai konstansok: $\mathcal{A}(\uparrow) := 1, \mathcal{A}(\downarrow) := 0$
- Konnektívák: $\mathcal{A}(F \wedge G) := \mathcal{A}(F) \wedge \mathcal{A}(G), \mathcal{A}(F \vee G) := \mathcal{A}(F) \vee \mathcal{A}(G), \mathcal{A}(\neg F) := \neg \mathcal{A}(F), \dots$
- Atomi formulák: $\mathcal{A}(p(t_1, \dots, t_n)) := I(p)(\mathcal{A}(t_1), \dots, \mathcal{A}(t_n))$

Azaz \mathcal{A} -ban először kiértékeljük t_1, \dots, t_n termeket, majd a kapott a_1, \dots, a_n objektumokat befejlettesítjük abba a predikátumba, amit ebben a struktúrában p jelöl.

- Kvantorok:

- $\mathcal{A}(\exists x F)$: 1, ha van olyan $a \in A$, melyre $\mathcal{A}_{[x \mapsto a]}(F) = 1$, különben 0
- $\mathcal{A}(\forall x F)$: 1, ha minden $a \in A$ -ra igaz, hogy $\mathcal{A}_{[x \mapsto a]}(F) = 1$, különben 0

5.2.2. Normálformák az elsőrendű logikában

Zárt Skolem alak

1. **Nyilak eliminálása** ($F \rightarrow G \equiv \neg F \vee G$)

2. **Kiigazítás**: Ne legyen változónév-ütközés

3. **Prenex alakra hozás**: Összes kvantor előre kerül

Idáig volt ekvivalens az átalakítás

1. **Skolem alakra hozás**: Összes kvantor elől, és minden \forall

2. **Lezáras**: Ne maradjon szabad változó-előfordulás

Kiigazítás

- Különböző helyeken levő kvantorok különböző változókat kötnek és
- Nincs olyan változó, mely szabadon is és kötötten is előfordul.

Gyakorlatban annyi ez a lépés, hogy a kötött változókat átnevezzük, jellemzően indexeléssel.

Átnevezni csak kötött változókat ér, szabad változót nem, akkor marad ekvivalens.

Prenex alak

Egy formula **Prenex alakú**, ha $Q_1 x_1 Q_2 x_2 \dots Q_n x_n (F)$ alakú, ahol F **kvantormentes** formula, és minden Q_I egy **kvantor**.

Minden formula ekvivalens Prenex alakra hozható.

Első lépésként **ki kell igazítani a formulát** (előző lépés).

- Ha egy negálást áthúzunk egy kvantoron, megfordul a kvantor: $\neg \exists x F \equiv \forall x \neg F$
- $\exists x F \vee G \equiv \exists x(F \vee G)$
 - Ha x nem szerepel G -ben szabadon, ezért kell előtte kiigazírni!

Skolem alak

Egy formula **Skolem alakú**, ha $F = \forall x_1 \forall x_2 \dots \forall x_n (F^*)$, ahol F^* -ben (a formula magjában) már nincs kvantor.

Skolem alak értelme: $\forall x_1 \dots \forall x_n F^* \models F^*[x_1/t_1, \dots, x_n/t_n]$

Tehát termeket lehet a változók helyére helyettesíteni.

A Skolem-alakra hozás **nem ekvivalens**, **csak s-ekvivalens**: minden F formulához konstruálható eg yolyan F' Skolem alakú formula, ami pontosan akkor kielégíthető, ha F is az. Ennek jele: $F \equiv_S F'$

- Prenex alakra hozzuk a formulát
- Skolem-függvényekkel eltüntetjük a \exists kvantorokat:
 - minden $\exists y$ -lekötött változót a formula magjában cseréljünk le egy $f(x_1, \dots, x_n)$ termre, ahol:
 - f egy teljesen új függvényszimbólum,
 - x_1, \dots, x_n pedig az y előtt szereplő \forall -kötött változók.

Zárt Skolem alak

- minden x szabad előfordulás helyett egy új c_x konstansjelet vezetünk be
- ezt úgy, hogy minden formulában az összes szabad x helyére ugyanazt a c_x -et írjuk

Ez is s-ekvivalens átalakítás

CNF elsőrendű logikában

- **Literál:** Atomi formula (akkor pozitív), vagy negáltja (akkor negatív), pl.: $p(x, c)$, $\neg q(x, f(x), z)$
- **Klóz:** Literálok véges diszjunkciója, pl.: $q(x) \vee \neg q(x, c)$
- **CNF:** Klózok konjukciója, pl.: $(p(x) \vee \neg q(y, c)) \wedge \neg p(x)$

Kvantormentes elsőrendű logikai formulát az ítéletkalkulusban megszokott módon hozhatunk CNF-re.

5.2.3. Alap rezolúció

Alap, mert alaptermek szerepelnek benne

Input: Elsőrendű formulák egy Σ halmaza

Ha Σ kielégíthetetlen, akkor az algoritmus ezt véges sok lépésben levezeti

Ha kielégíthető, akkor vagy ezt vezeti le, vagy végtelen ciklusba esik

Módszer

- Σ elemeit zárt Skolem alakra hozzuk, a kapott formulák magját CNF-re.
 - Jelölje Σ' a kapott klóz halmazt
- Ekkor $E(\Sigma')$ a klózok **alap példányainak halmaza**

Ez annyit takar, hogy a klózban a változók helyére ízlés szerint alaptermeket helyettesítünk, minden ilyennek a halmaza

- Az $E(\Sigma')$ halmazon futtatjuk az ítéletkalkulus-beli rezolúciós algoritmust

Mivel $E(\Sigma')$ általában végtelen, így az algoritmus (mondjuk)

- Egy lépésben legenerálja, és felveszi $E(\Sigma')$ egy elemét
- az eddigi klózokkal rezolvenst képez, amíg csak lehet
- ha közben megkapjuk az üres klózt, Σ kielégíthetetlen
- különben generáljuk a következő elemet.

Ha $\forall x_1 \forall x_2 \dots \forall x_n F$ egy zárt Skolem alak, ahol F kvantormentes, akkor

Herbrand kiterjesztése az

$$E(\forall x_1 \forall x_2 \dots \forall x_n F) := \left\{ F[x_1/t_1, x_2/t_2, \dots, x_n/t_n] : t_i \in T_0 \right\}$$

formulahalmaz. (E mint „extension”)

Ha Σ zárt Skolem normálformák halmaza, akkor legyen

$$E(\Sigma) := \bigcup_{F \in \Sigma} E(F),$$

tehát Σ Herbrand kiterjesztését úgy kapjuk, hogy az összes Σ -beli F formula Herbrand kiterjesztésének vesszük az unióját.

Ha $\Sigma = \{ \forall x p(x), \forall y \neg p(f(y)) \}$, akkor $E(\Sigma)$ -ban van pl.

- $p(c)$ (az első formula Herbrand kiterjesztéséből, c az új konstansjel)
- $p(f(c))$ (szintén az elsőéből)
- $\neg p(f(c))$ (a másodikéból)

és $E(\Sigma)$ kielégíthetetlen (hiszen szerepel benne $p(f(c))$ és $\neg p(f(c))$ is).

Helyesség, és teljesség

- A zárt Skolem alakra hozás s-ekvivalens átalakítás, tehát Σ pontosan akkor kielégíthetetlen, ha Σ' az
- A Herbrand-tétel következménye szerint Σ' pontosan akkor kielégíthetetlen, ha $E(\Sigma')$ az

Mert a Herbrand-kiterjesztés s-ekvivalens transzformáció

- Az ítéletkalkulus kompaktsági tétele szerint $E(\Sigma')$ pontosan akkor kielégíthetetlen, ha van egy véges Σ_0 kielégíthetetlen részhalmaza

Azaz elég véges sokat legyártani

- A rezolúciós algoritmus teljessége szerint ha a Σ_0 véges klózhalmaz kielégíthetetlen, akkor az algoritmus ezt levezeti
- Tehát ha Σ kielégíthetetlen, akkor az algoritmus leáll ezzel a válasszal akkor, amikor egy ilyen Σ_0 halmaznak már legenerálta az összes elemét (és rezolvenseit, közöttük \square -t)

Az alap rezolúcióval az lehet a probléma, hogy nagy a keresési tere azáltal, hogy a változókat alaptermekkel helyettesítgetjük

5.2.4. Elsőrendű rezolúció

Elsőrendű rezolvensképzés

Két elsőrendű logikai klóz, C_1 és C_2 elsőrendű rezolvensét így kapjuk:

- Átnevezzük a klózokban a változókat úgy (legyenek a változónevezések s_1 és s_2), hogy a kapott $C_1 \cdot s_1$ és $C_2 \cdot s_2$ klózok ne tartalmazzanak közös változót.
- Kiválasztunk $C_1 \cdot s_1$ -ből l_1, \dots, l_m és $C_2 \cdot s_2$ -ből l'_1, \dots, l'_n literálokat, mindenből legalább egyet-egyet.
- Futtatjuk az egyesítési algoritmust a $C = l_1, \dots, l_m, l'_1, \dots, l'_n$ klózon.

Emiatt az egyesítési lépés miatt a korábbi literál kiválasztást érdemes úgy csinálni, hogy csak egy féle predikátumjeleket választunk ki, és az egyik klóból csak pozitív előfordulásokat, a másikból csak negatívakat. Így lesz esély arra, hogy egyesíthető legyen.

- Ha C egyesíthető az s legálthatósább egyesítővel, akkor s -et végrehajtjuk a nem kiválasztott literálok halmazán:
$$R := ((C \cdot s_1 - l_1, \dots, l_m) \cup (C_2 \cdot s_2 - l'_1, \dots, l'_n)) \cdot s$$

A kapott R klóz a C_1 és C_2 egy elsőrendű rezolvense.

Példa

$$C_1 = \{p(f(x)), \neg q(z), p(z)\}$$

$$C_2 = \{\neg p(x), r(g(x), a)\}$$

- Átnevezés: (mondjuk) a C_2 -beli x -et y -ra nevezzük át

$$C_1 s_1 = \{p(f(x)), \neg q(z), p(z)\}$$

$$C_2 s_2 = \{\neg p(y), r(g(y), a)\}$$

- Kiválasztás: válasszuk mondjuk $C_1 s_1$ -ből $p(f(x))$ -et és $p(z)$ -t, $C_2 s_2$ -ből pedig $\neg p(y)$ -t

- Egyesítés: a

$$\{p(f(x)), p(z), p(y)\}$$

klóz legáltalánosabb egyesítője $s = [z/f(x)][y/f(x)]$

- Végrehajtás: a nem-kiválasztott literálokon végrehajtjuk s -t:

$$\{\neg q(z), r(g(y), a)\} \cdot [z/f(x)][y/f(x)] = \{\neg q(f(x)), r(g(f(x)), a)\}$$

Ez a két klóz (egyik) rezolvense.

Algoritmus

Input: Elsőrendű klózok egy Σ halmaza. Úgy tekintjük, mintha a Σ -beli klózok változói univerzálisan lennének kvantálva.

Azért tekinthetjük így, mert $\forall x(F \wedge G) \equiv \forall F \wedge \forall G$

Output:

- Ha $\Sigma \models \downarrow$, akkor "kielégíthetetlen"
- Különben "kielégíthető", vagy végtelen ciklus

Listát vezetünk klózokról, egy klózt felveszünk, ha

- Σ -beli, vagy
- két, már a listán szereplő klóz rezolvense.

Ha \square rákerül a listára, akkor Σ kielégíthetetlen.

Különben, ha már nem tudunk több klózt lebezetni, Σ kielégíthető.

$Res(\Sigma)$ jelöli azt a halmazt, amely tartalmazza Σ elemeit, és a belőlük egy rezolvensképzéssel levezethető klózokat.

$Res^*(\Sigma)$ pedig a Σ -ból rezolúcióval levezethető összes klóz halmazát jelöli.

Helyesség

- A helyesség a rezolvensképzés helyességéből következik
- Mivel a klózik univerzálisan kvantáltak (a Skolem alakból), így tetszőleges C klózra, és s helyettesítésre $C \models C \cdot s$
- Tehát a rezolvensképzésnél felírt C_1 -nek $C_1 \cdot s_1 \cdot s$, C_2 -nek pedig $C_2 \cdot s_2 \cdot s$ egy-egy logikai következménye
- Tehát $C_1, C_2 \models C_1 s_1 s, C_2 s_2 s$
- Ennek a két klóznak pedig a rezolvens következménye (az "eredeti" rezolúciós következtetés szerint)

Teljesség

- A teljességi irányhoz felhasználjuk az alap rezolúció teljességét
- Tehát: Ha Σ kielégíthetetlen, akkor az üres klóznak van egy $C'_1, C'_2, \dots, C'_n = \square$ alaprezolúciós levezetése.
- Ebből az alaprezolúciós levezetésből fogunk készíteni egy C_1, C_2, \dots, C_n elsőrendű rezolúciós levezetést.
- A klózokat úgy fogjuk elkészíteni indukcióval n szerint, hogy minden i -re a C_i -nek a C'_i egy (alap) példánya lesz.
 - Ha $C'_i \in E(\Sigma)$, azaz C'_i egy Σ -beli C klóz (alap) példánya, akkor legyen $C_i := C$
 - A másik lehetőség, hogy C'_i a C'_j és C'_k klózok, $j, k < i$, egy rezolvense.

- Ennek az esetnek a belátásához felítjuk az ún. lift lemmát:
- Ha C_1 -nek C'_1 , C_2 -nek pedig C'_2 alap példányai, melyeknek R' rezolvense, akkor van C_1 -nek, és C_2 -nek olyan elsőrendű R rezolvense, melynek R' alap példánya.

Lift lemma bizonyítása kell?

- Mivel a $C'_n = \square$ üres klöz csak önmagának példánya, így $C_n = \square$ kell legyen.

6. Mesterséges Intelligencia 1.

6.1. 1. Keresési feladat: feladatrepräsentáció, vak keresés, informált keresés, heurisztikák. Kétszemélyes zéró összegű játékok: minimax, alfa-béta eljárás. Korlátos kielégítési feladat.

6.1.1. Keresési feladat

A feladatkörnyezetről feltételezzük, hogy *diszkrét, statikus, determinisztikus*, és *teljesen megfigyelhető*.

Feladatrepräsentáció

Következőkkel modellezük a feladatot:

- **Lehetséges állapotok** halmaza
- Egy **kezdőállapot**
- **Lehetséges cselekvések** halmaza
- Egy **állapotátmenet függvény**: minden állapothoz rendel egy (**cselekvés, állapot**) típusú, rendezett párokból álló halmazt.
- Állapotátmenet **költségfüggvénye**, amely minden lehetséges állapot-cselekvés-állapot hármashoz egy $c(x, a, y)$ valós nemnegatív költségértéket rendel
- **Célállapotok** halmaza

Ez egy gúlyozott gráfot definiál, amiben a **csúcsok az állapotok, élek a cselekvések, súlyok a költségek**.

Ez a gráf az **állapottér**

Út: Állapotok cselekvésekkel összekötött sorozata.

Vak (informálatlan) keresés

Cél: Adott kezdőállapotból megtalálni egy minimális költségű utat egy célállapotba.

Ez azért nem egy triviális, legrövidebb út keresési feladat, mert az állapottér nem minden adott teljes egészében, mert nem minden véges.

Megvalósítás: Keresőfával, azaz a kezdőállapotból növesztünk egy fát a szomszédos állapotok hozzávételével amíg célállapotot nem találunk.

Keresőfe nem azonos az állapottérrel! Hiszen az állapottér nem is feltétele fa.

Keresőfa egy csúcsában tárolt információ:

- Szülő
- Állapot
- Cselekvés, ami a szülőből ide vezetett
- Útköltség a kezdőállapottól eddig
- Mélység (kezdőállapoté nulla)

Általános, absztrakt eljárás

```
fakereses() {
    perem = { újcsúcs(kezdőállapot) }
    while perem.nemüres() {
        csúcs = perem.elsökivesz()
        if csúcs.célállapot() {
            return csúcs
        }
        perem.beszür(csúcs.kiterjeszt())
    }
    throw Error
}
```

Ha olyan csúcsot szúrunk be, aminek állapota már szerepel a `perem`-ben, akkor a nagyobb költségűt felesleges benne hagyni.

- `csúcs.kiterjeszt()`: Létrehozza a csúcsból elérhető összes állapothoz tartozó keresőfa csúcsot, a mezőket megfelelően inicializálja.

- `perem`: Egy prioritási sor.
- `perem.elsökivesz()`: Ez definiálja a bejárás stratégiát. (Lényegében a prioritási sorban a kulcsok rendezésének módja.)

Szélességi keresés

FIFO perem.

- Teljes, minden véges számú állapot érintésével elérhető állapotot véges időben elér.
- Általában nem optimális, de akkor pl. igen, ha a költség a mélység nem csökkenő függvénye.
- Időigény = Tárigény = $O(b^{d+1})$

Exponenciális komplexitás miatt nem skálázódik nagy d -kre.

b : szomszédok maximális száma.

d : legkisebb mélységű célállapot mélysége a keresőfában.

Mélységi keresés

LIFO perem.

- Teljes, ha a keresési fa véges mérezű. Egyébként nem.
- Nem optimális.
- Időigény: $O(b^m)$, Tárigény: $O(bm)$

Az időigény nagyon rossz, tárigény jó, mert nem exponenciális.

m : keresőfa maximális mélysége.

Iteratívan mélyülő keresés

Mélységi keresések sorozata 1, 2, 3 srb. mélységekre korlátozva, amíg célállapotot nem találunk.

- Teljesség és optimalitás a szélességi kereséssel egyezik meg.
 - Időigény: $O(b^d)$
- Általában jobb, mint a szélességi
- Tárigény: $O(bd)$
- Jobb, mint a mélységi

Meglepő, de igaz, hogy annak ellenére, hogy az első szinteket újra, meg újra bejárjuk, javítunk.

Ez a legjobb informálatlan kereső.

Egyenletes költségű keresés

Költség alapján rendezi a permet, először a legkisebb útiköltségű csúcsot fejtjük ki.

Gráfkeresés

Ha nem fa az állapottér.

Ha a kezdőállapotból több út is vezet egy állapotba, akkor a fakeresés végtelen ciklusba eshet, de legalábbis a hatékonysága drasztikusan csökken.

Cél: Ugyan azon állapotba vezető útak redundáns tárolásának elkerülése.

Zárt halmaz: Ebbe tároljuk a peremből már egyszer kivett csúcsokat.

```
gráfkereses() {
    perem = { újcsúcs(kezdőállapot) }
    zárt = { }
    while perem.nemüres() {
        csúcs = perem.elsökivesz()
        if csúcs.célállapot() {
            return csúcs
        }
        zárt.hozzáad(csúcs)
        perem.beszűr(csúcs.kiterjeszt() - zárt)
    }
    throw Error
}
```

Mi van, ha egy zárt halmazban levő csúcshoz **találnánk jobb megoldást?**

- Egyenletes költségű kereséskor nincs ilyen eset, mert az úgy pont Dijkstra algoritmusa az állapottérre.
- Mélységi keresésnél előfordulhat ilyen, ekkor át kell linkelni a zárt halmazban tárolt csúcsot a jobb út felé.

| Ez csak annyi, hogy frissítjük a szülőt, mélységet, költséget, és cselekvést?

Informált keresés

Az eddigi algoritmusok nem fogtak semmit arról, hogy merre haladnak tovább.

Heurisztika: minden állapotból megbeszülni, hogy mekkora az optimális út költsége az adott állapotból egy célállapotba. Ez alapján tudjuk kiválasztani, merre érdemes haladni.

| Például útvonal-tervezési probléma esetén jó heurisztika lehet a légvonal-beli távolság.

$h(n)$: Optimális költség közelítése n állapotból a legközelebbi célállapotba.

$g(n)$: Tényleges költség a kezdőállapozból n -be.

Mohó legjobb-először

A peremben rendezést h alapján végezzük, a legkisebb értékű csúcsot vesszük ki.

$h(n) = 0$, ha n célállapot feltételezése mellett:

- Teljes, ha a keresési fa véges mélységű
- Nem optimális
- Időigény = Tárigény = $O(b^m)$

| Jó h -val javítható ez a komplexitás

A^*

A peremben a rendezést $f() = h() + g()$ alapján végezzük, a legkisebb értékű vesszük ki.

| A teljes út költségét becsüli.

Teljesség és optimalitás

h elfogadható (megengedhető): Ha nem ad nagyobb értéket, mint a tényleges optimális érték, azaz nem becsül túl.

Fakeresés esetén ha h elfogadható, és a keresési fa véges, akkor az A^* optimális.

h konzisztns (monoton): Ha $h(n) \leq c(n, a, n') + h(n')$ minden n -re, és n minden n' szomszédjára.

Gráfkeresés esetén ha h konzisztns, és az állapottér véges, akkor az A^* optimális.

Hatákonyság

Az A^* optimálisan hatékony, hiszen csak azokat a csúcsokat terjeszti ki, amelyekre $f() < C^*$ (C^* az optimális költség).

| Bár ezeket minden optimális algoritmusnak ki kell terjeszteni.

A tárigény általában exponenciális, de nagyon függ h -től (ha $h = h^*$, akkor konstans).

Időigény szintén erősen függ h -től.

Egyszerűsített memóriakorlátosított A^*

Próbáljuk meg az összes rendelkezésre álló memóriát használni, és kezeljük le, ha elfogy.

Futtassuk az A^* -öt amíg van memória, ha elfogyott:

- Töröljük a legrosszabb levelet a keresőfában, egyezés esetén a régebbit.
- A törölt csúcs szülőjében jegyezzük meg az innen elérhető ismert legjobb költséget (így később vissza lehet ide tértő, ha minden többi útról kiderülne, hogy rosszabb)

Teljes, ha a legkisebb mélységű célállapot mélységényi csúcs belefér a memóriába egyszerre. (Tehát az oda vezető egész út.)

Hasonló költségű utak esetén előfordulhat, hogy ugrál a két út között, így lassan talál megoldást.

Relaxáció

Feltételek elhagyása.

7	2	4
5		6
8	3	1

Például a 8-kirakós játék esetén

- h_1 : Rossz helyen lévő számok, ábrán 8
 - Relaxáció: minden szám egyből a helyére rakható
- h_2 : Manhattan távolság számonként, ábrán 18
 - Relaxáció: minden szám tolható a szomszédba, akkor is ha van ott másik szám
- h_{opt} : Optimális költségek, ábrán 26

Észervétel: $\forall n : h_1(n) \leq h_2(n)$, azaz h_2 dominálja h_1 -et.

A relaxált probléma optimális költsége \leq az eredeti probléma optimális költségénél, mivel az eredeti probléma állapotterre része a relaxálnak (mivel pl az hogy akkor is léphetünk szomszédba, ha foglalt, növeli az állapotteret). Tehát elfogadható heurisztikát kapunk.

Konzisztens is

Relaxálás automatizálása

Ha formális kifejezés adja meg a lehetséges lépéseket a probléma lehetséges állapotairól, akkor automatikusan elhagyhatunk feltételeket, pl. 8-kirakó esetében formálisan:

1. Egy számot csak szomszédos pozícióra lehet mozgatni, és
2. Egy számot csak üres pozícióba lehet mozgatni

Ha elhagyjuk mindkét szabályt, akkor h_1 -et kapjuk.

Ha csak a 2-es szabályt hagyjuk el, kkor pedig h_2 -t.

Több heurisztika kombinálása

$$h(n) = \max(h_1(n), \dots, h_k(n))$$

Ha igaz, hogy minden i -re h_i konzisztens, akkor h is konzisztens.

6.1.2. Kétszemélyes zéró összegű játékok

Kétszemélyes, lépésváltásos, determinisztikus, zéró-összegű játék

- Lehetséges állapotok halmaza
 - Legális játékállások
- Egy kezdőállapot
- Lehetséges cselekvések halmaza
- Állapotátmenet függvény
- Célállapotok
- Hasznosségfüggvény: Célállapothoz hasznosságot rendel

Ez a **játékgráf** (jellemzően nem fa)

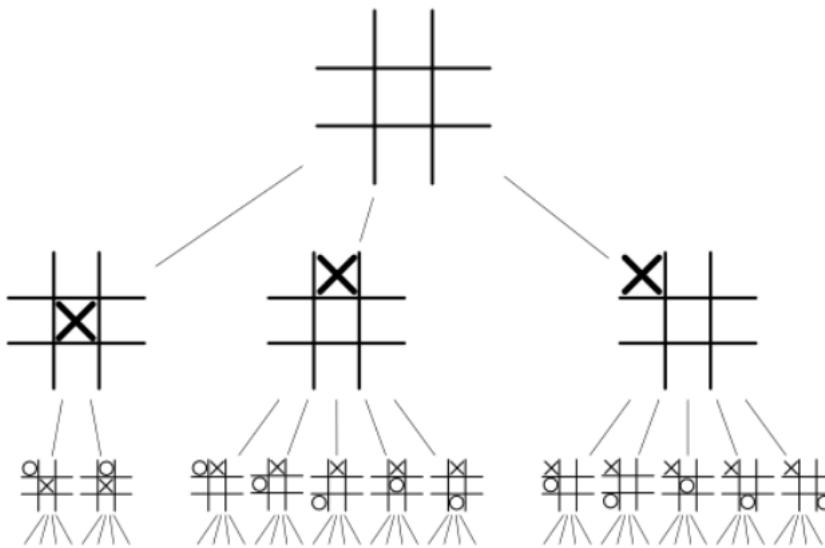
Két **ágens** van, felváltva lépnek.

- **MAX játékos**: Maximalizálni akarja a hasznosság függvényt.
- **MIN játékos**: Minimalizálni akarja a hasznosság függvényt.

Konvenció: MAX kezd

Első célállapot elérésekor a játéknak vége.

Zéró összegű játék: Modellünkben MIN minimalizálja a hasznosságot, lényegében maximalizálja a negatív hasznosságot. Így lényegében a MIN játékos hasznosságfüggvénye a MAX játékosénak -1-szerese, innen az elnevezés, mert a kettő összege nulla.



Minimax algoritmus

Tökéletesen racionális hipotézis: Tfh. mindenki játékos a teljes játékgráfot ismeri, tetszőlegesen komplex számításokat tud elvégezni, és nem hibázik.

Stratégia: minden állapotra meghatározza, hogy melyik lépést kell választani.

Minmax a következőket számolja minden n csúcsra:

$$\text{minimax}(n) = \begin{cases} \text{hasznosság}(n) & \text{ha végállapot} \\ \max_{\{a \text{ } n \text{ szomszédja}\}} \text{minimax}(a) & \text{ha MAX jön} \\ \min_{\{a \text{ } n \text{ szomszédja}\}} \text{minimax}(a) & \text{ha MIN jön} \end{cases}$$

```
maxErtek(n) {
    if végállapot(n) return hasznosság(n)
    max = -végtelen
    for a in n szomszédaí {
        max = max(max, minÉrték(a)) // Itt a MIN játékos lép!
    }
    return max
}
```

```
minErtek(n) {
    if végállapot(n) return hasznosság(n)
    min = végtelen
    for a in n szomszédaí {
        min = min(min, maxÉrték(a)) // Itt a MAX játékos lép!
    }
    return min
}
```

Az eljárás `maxErtek(kezdőállapot)` hívással indul, hiszen a MAX játékos kezd.

Ha van a játékgráfban köt, akkor nem terminál. Ez a gyakorlatban azért nem probléma, mert csak adott mélységgel futtatjuk.

Sok játék esetén a szabályok kizárták a végtelenséggel futó köröket.

A minmax érték az optimális hasznosság, amit az adott állapotból el lehet érni, ha az ellenfél tökéletesen racionális.

Alfa-béta vágás

Ha tudunk, hogy MAX egy adott csúcs rekurzív kiértékelése közben talált olyan stratégiát, amellyel ki tud kényszeríteni pl. 10 értékű hasznosságot az adott csúcsban, akkor a csúcs további kiértékelése közben nem kell vizsgálni olyan állapotokat, amelyekben MIN ki tud kényszeríteni ≤ 10 hasznosságot, hiszen tudunk, hogy MAX sosem fogja ide engedni a játéket.

Új paraméterek:

- **Alfa:** MAX-nak már felfedeztünk egy olyan stratégiát, amely **alfa** hasznosságot biztosít egy olyan állapotból indulva, ami a keresőfában az n állapotból a gyökér felé vezető úton van.
- **Béta:** MIN-nek már felfedeztünk egy olyan stratégiát, amely **béta** hasznosságot biztosít egy olyan állapotból indulva, ami a keresőfában az n állapotból a gyökér felé vezető úton van.

Számítás a `maxÉrték(kezdőállapot, -végtelen, végtelen)` hívással indul.

```
maxErtek(n, alfa, beta) {
    if végállapot(n) return hasznosság(n)
    max = -végtelen
    for a in n szomszéda {
        max = max(max, minÉrték(a, alfa, beta)) // Itt a MIN játékos lép!
        if max >= beta return max // Vágás!
        alfa = max(max, alfa)
    }
    return max
}
```

```
mixErtek(n, alfa, beta) {
    if végállapot(n) return hasznosság(n)
    min = végtelen
    for a in n szomszéda {
        min = min(min, maxÉrték(a, alfa, beta)) // Itt a MIN játékos lép!
        if alfa >= min return min // Vágás!
        beta = min(min, beta)
    }
    return min
}
```

Ha mindenkor legjobb lépést vesszük, akkor $O(b^{m/2})$, amúgy $O(b^m)$.

Gyakorlatban használhatunk rendezési heurisztikákat, amik sokszor közel kerülnek az optimális esethez.

6.1.3. Korlátozás kielégítési feladat

Lehetséges állapotok halmaza: $\mathcal{D} = \mathcal{D}_1 \times \dots \times \mathcal{D}_n$, ahol \mathcal{D}_i az i . változó lehetséges értékei, azaz a feladat állapotai az n db változó lehetséges értékkombinációi.

Célállapotok: A megengedett állapotok, amelyek definíciója a következő: Adottak C_1, \dots, C_m korlátozások, $C_i \subseteq \mathcal{D}$. A megengedett vagy konzisztens állapotok halmaza a $C_1 \cap \dots \cap C_m$. (Ezek minden korlátozást kielégítnek)

Gyakran egy C_i korlátozás egy változóra vagy változópárra fejez ki megszorítást

Kényszergráf: A feladatban szereplő változók és a korlátozások által definiált gráf. Ugye ez változó párokra értendő, de ha esetleg több változót érintenek a korlátozások, az is felírható olyan korlátokkal, amik kettőt érintenek.

Inkrementális kereső algoritmusok

A probléma éllapottárbeli keresésként értelmezve:

- minden változóhoz felveszünk egy új "ismeretlen" értéket. Jelölje ezt "?". a kezdeti állapot: $(?, \dots, ?)$.
- Az állapotátmenet függvény rendeljen hozzá minden állapothoz olyan állapotokat, amelyekben egyelőre kevésbé ? van, és amelyek megengedettek.
- A költség minden állapotátmenetre legyen konstans.

Mélységi keresés jó választás lehet, mert a keresőfa mélysége kicsi.

Informált kereséssel: próbálunk nehezen kielégítható változókat kifejteni előbb:

- Amelyikhez a legkevésesebb megengedett lépés maradt
- Amelyre a legtöbb korlátozás vonatkozik

A választott változó megengedett lépései ből kezdjük azzal, amelyik a legkevésbé korlátozza a következő lehetséges lépések számát.

Optimalizáló algoritmusok, lokális keresők

Egy másik lehetséges megközelítés **optimalizálási problémát definiálni**:

- A **célfüggvényt** definiáljuk pl. úgy, hogy legyen a megsértett korlátozások száma. Ha eleve tartozott célfüggvény a feladathoz, össze kell kombonálni vele.
- Az **operátorokat** definiálhatjuk sokféleképpen, pl. valamely változó értékének megváltoztatásaként.

6.2. 2. Teljes együttes eloszlás tömör reprezentációja, Bayes hálók. Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere.

6.2.1. * Alapfogalmak

Véletlen változók

Van **neve**, és **lehetséges értékei**, azaz **domainje**.

Elemi kijelentés: A értékének egy korlátozását fejezik ki (pl. $A = d$, ahol $d \in D_A$)

Itt A egy véletlen változó, D_A domainnel.

Véletlen változók típusai:

- **Logikai:** A domain ekkor *igaz, hamis*.

◦ Pl. "Fogfájás=igaz", vagy csak simán "fogfájás" (és negáltja \neg fogfájás) egy elemi kijelentés.

- **Diszkrét:** Megszámlálható domain, pl. idő, ahol a domain *nap, eső, felhő, hó*.

◦ Pl. "Idő=nap", vagy csak simán "nap".

- **Folytonos:** Pl.: X véletlen változó, $D \subseteq \mathbb{R}$

◦ Pl. $X \leq 3.2$ Egy elemi kijelentés

Komplex kijelentések: Elemi kijelentésekből képezzük a szokásos logikai operátorokkal.

Elemi esemény: minden véletlen változóhoz értéket rendel: Ha A_1, \dots, A_n véletlen változókat definiálunk a D_1, \dots, D_n dománekkal, akkor az elemi események halmaza a $D_1 \times \dots \times D_n$ halmaz.

Valószínűség: Egy függvény, amely egy kijelentéshez egy valós számot rendel.

Véletlen változók: nagy betűk Kijelentések, és a változók értékei: kis betűk

$P(a)$ az a kijelentés valószínűsége.

$P(A)$: A tetszőleges értékére érvényes az adott állítás / képlet.

A P függvény a **valószínűségi eloszlás**, amely minden lehetséges kijelentéshez valószínűséget rendel.

$$P(A, B) = P(A \wedge B)$$

Feltételes valószínűség

$$P(a|b) = P(a \wedge b)/P(b) \text{ (feltéve, hogy } P(b) > 0\text{)}$$

Jelentése: Az a kijelentés **feltételes valószínűsége**, feltéve, hogy az összes tudásunk b (b is egy kijelentés)

Szorzatszabály: $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$

Teljes együttes eloszlás

Az **összes elemi esemény valószínűségét** megadja.

Ebből kiszámolható **bármely kijelentés valószínűsége**.

Továbbiakhoz feltesszük, hogy az **összes változó diszkrét**.

Példa

Legyenek a véletlen változók **Luk**, **Fogfájás**, **Beakad**, minden logikai típusú. Ekkor a teljes együttes eloszlást egy táblázat tartalmazza:

Luk	Fogfájás	Beakad	$P()$
nem	nem	nem	0.576
nem	nem	igen	0.144
nem	igen	nem	0.064
nem	igen	igen	0.016
igen	nem	nem	0.008
igen	nem	igen	0.072
igen	igen	nem	0.012
igen	igen	igen	0.108

6.2.2. * Függetlenség

Az a és b kijelentések függetlenek akkor, és csak akkor, ha $P(a \wedge b) = P(a)P(b)$.

Az A és B véletlen változók függetlenek akkor, és csak akkor, ha $P(A, B) = P(A)P(B)$, vagy ekvivalensen $P(A|B) = P(A)$, illetve $P(B|A) = P(B)$

Azaz két változó független, ha az egyik sem tartalmaz információt a másikról.

6.2.3. * Feltételes függetlenség

Az a és b **kijelentések** feltételesen függetlenek c feltételével akkor, és csak akkor, ha $P(a \wedge b|c) = P(a|b)P(b|c)$. Ekkor a és b nem feltétlenül független abszolút értelemben.

Tipikus eset, ha a és b közös **oka** c .

A és B **véletlen változók** feltételesen függetlenek C feltevésével akkor, és csak akkor, ha $P(A, B|C) = P(A|C)P(B|C)$, vagy ekvivalensen $P(A|B, C) = P(A|C)$, illetve $P(B|A, C) = P(B|C)$.

Egy lehetséges elérhető tömörítés:

$$P(A, B, C) = P(A, B|C)P(C) = P(A|C)P(B|C)P(C)$$

Első egyenlőség: **Szorzatszabály**

Második egyenlőség: Feltételes függetlenség **feltevése**

$$P(A|C), P(B|C), P(C) \text{ táblázatok méreteinek összege sokkal kisebb lehet, mint az eredeti } P(A, B, C)$$

Ha A feltevése mellett B_1, \dots, B_n kölcsönösen függetlenek, azaz

$$P(B_1, \dots, B_n|A) = \prod_{i=1}^n P(B_i|A)$$

Ez a naiv Bayes modell alakja. Itt $O(n)$ tömörítés érhető el, hiszen

$$P(B_1, \dots, B_n|A) = P(A) \prod_{i=1}^n P(B_i|A)$$

6.2.4. Teljes együttes eloszlás tömör reprezentációja

Feltételes függetlenséggel tudjuk tömöríteni a teljes együttes eloszlás reprezentációját

Láncszabály

Legyen X_1, \dots, X_n a félelten változók egy tetszőleges felsorolása, ekkor a láncszabály:

$$P(X_1, \dots, X_n) = P(X_1|X_2, \dots, X_n)P(X_2, \dots, X_n) = P(X_1|X_2, \dots, X_n)P(X_2|X_3, \dots, X_n)P(X_3|X_4, \dots, X_n)\dots P(X_n) = \prod_{i=1}^n P(X_i|X_{i+1}, \dots, X_n)$$

Minden $P(X_i|X_{i+1}, \dots, X_n)$ tényezőre az X_{i+1}, \dots, X_n változókból vegyük egy $Szülöök(X_i)$ részhalmazt, melyre igaz, hogy $P(X_i|X_{i+1}, \dots, X_n) = P(X_i|Szülöök(X_i))$, és a $Szülöök(X_i)$ halmaz minimális, azaz belőle több elem nem hagyható el, különben a fenti tulajdonság sérül.

Ez a tömörítési lépés, és annál jobb, minél kisebb a $Szülöök(X_i)$ halmaz.

Ekkor persze:

$$P(X_1, \dots, X_n) = \prod_{i=1}^n P(X_i|Szülöök(X_i))$$

Ez a teljes együttes eloszlás tömörített reprezentációja.

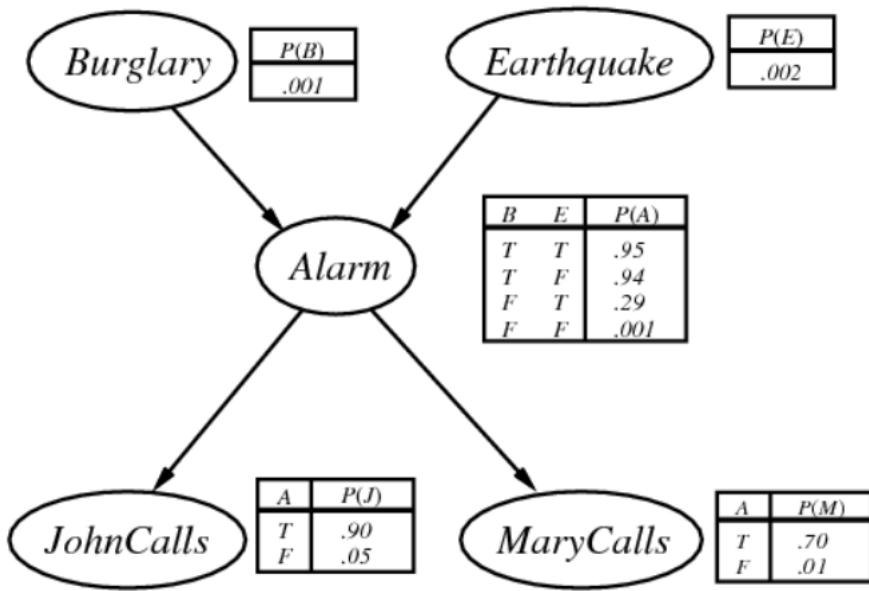
6.2.5. Bayes-hálók

Teljes együttes eloszlás tömörítése alapján egy **gráfot definiál**, ami egy tömör, és intuitív reprezentációt tesz lehetővé.

A teljes együttes eloszlás tömörítésekor bevezetett $Szülöök(X_i)$ jelölés már a gráf struktúrára utal, amelyben a csúcsok az X_i véletlen változók, és az (Y, X_i) irányított él akkor és csak akkor van behúzva, ha $Y \in Szülöök(X_i)$. A gráfban minden X_i változóhoz tartozik egy $P(X_i|Szülöök(X_i))$ eloszlás.

Ez egy irányított, körmentes gráfot alkot.

Példa



Egy konkrét lehetséges világ valószínűsége pl.:

$$P(B \wedge \neg E \wedge A \wedge J \wedge \neg M) = P(B)P(\neg E)P(A|B, \neg E)P(J|A)P(\neg M|A) = 0.001 * (1 - 0.002) * 0.94 * 0.9 * (1 - 0.7)$$

Ha egy Bayes-hálóban X -ből Y csúcsba vezet él, akkor Y és X nem független.

DE nem igaz, hogy ha X és Y nem független, akkor van köztük él.

Egy eloszláshoz **sok különböző Bayes háló** adható.

Bayes-háló konstruálása

Gyakorlatban sokszor egy szakértő definiálja a változókat, és a hatásokat a változók között, és ennek a tudásnak a segítségével kitöljük a változóhoz tartozó feltételes eloszlásokat. Ekkor adott az intuitív reprezentáció, ami definiál egy teljes együttes eloszlást, és ezt felhasználjuk valószínűségi következtetésekre.

Elméleti szempontból bármely adott együttes elosztásra konstruálható Bayes háló a láncszabálytal.

FONTOS! A láncszabály alkalmazásakor használt, rögzített változósorrendtől függ a Bayes háló.

Függetlenség, és Bayes hálók

Függetlenséggel kapcsolatos információk olvasása a hálóból.

Ha Y nem leszármazottja X -nek, akkor: $P(X|Szülök(X), Y) = P(X|Szülök(X))$

Bármely Y változóra igaz: $P(X|Markov-takaró(X), Y) = P(X|Markov-takaró(X))$

Markov-takaró(X): Az a halmaz, amely X szülőinek, X gyerekeinek, és X gyerekei szülőinek az uniója.

6.2.6. Gépi tanulás

Felügyelt gépi tanulás

Egy $f : X \rightarrow Y$ függvényt keresünk, amely illeszkedik adott példákra.

A példák $(x_1, f(x_1)), \dots, (x_n, f(x_n))$ alakban adottak. $x_i \in X$

Pl.: X az emailek halmaza, $Y = \text{spam}, \neg \text{spam}$

Példák kézzel osztályozott emailek.

Egy $h : X \rightarrow Y$ -t keresünk, ami f -et közelíti.

A h függvény **konzisztens** az adatokkal, ha $h(x_i) = f(x_i)$ minden példára.

A h függvényt minden egy H hipotézistérben keressük. Azaz a függvényt minden egy adott alapkabn keressük, pl. adott fokszámú polinom.

A tanulás **realizálható**, ha létezik $h \in H$, amelyre H konzisztens.

Gyakorlatban elég, ha h "közel" van a példákhoz, nem kell pontosan illeszkednie. Ez azért van, mert sokszor a tanuló példák zajt tartalmaznak, és káros ha ezeket megtanuljuk. (**túltanulás**)

Indukció problémája: f jó közelítése olyan, amely a példákon kívül is jól közelíti f -et, azaz jó **általánosít**.

Jó tanulás alapja:

- H hipotézistér gondos megválasztása
 - Tartalmazza f -et
 - Éppen annyira általános, amennyire kell (Occam borotvájának elve)
- Tanuló algoritmus megválasztása
 - H -ból kiválasztja h -t
 - Törekedhet tömörebben leírható h -t választani egy nagyon általános H -ból is.
- Számítási szempontból egyszerű reprezentáció, és hipotézistér
 - Hatékonyiság miatt
 - Túl egyszerű reprezentáció sokszor komplex hipotézistereket eredményez

Reprezentáció

Az X és Y halmazok tetszőleges objektumokat írhatnak le, fontos ezek jó reprezentációja.

Pl.: Szövegek esetén szemantikus beágyazások.

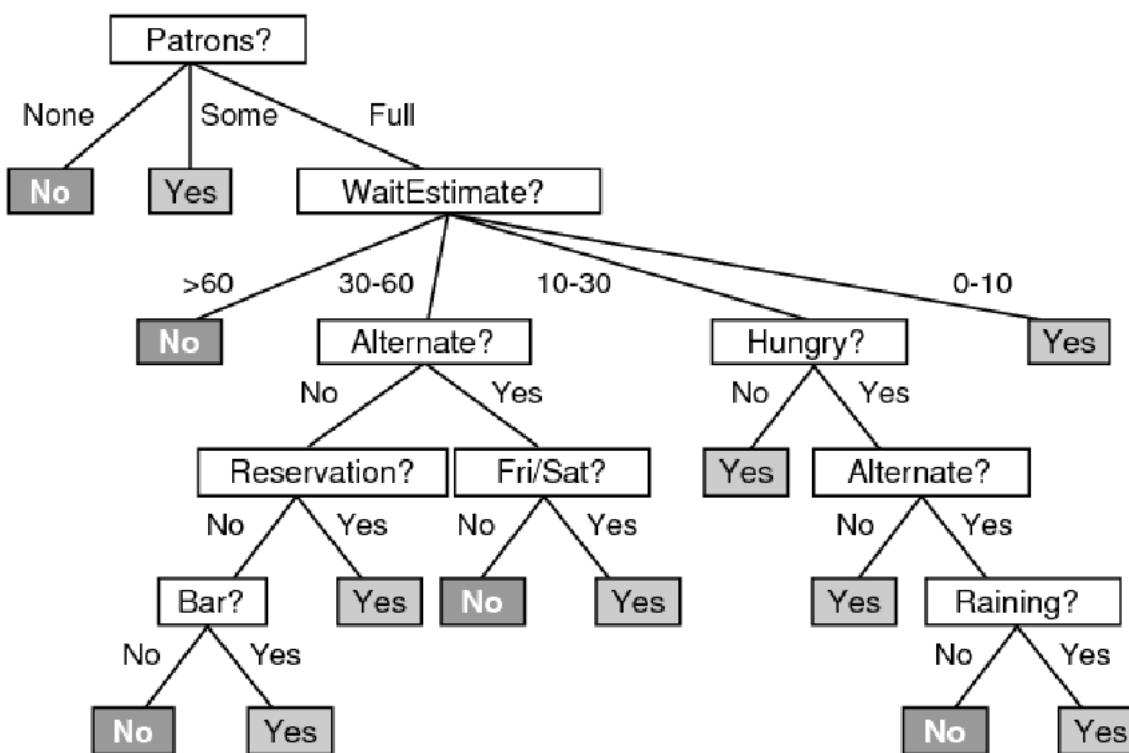
Y halmaz tartalmazhat diszkrét osztálycímkeket, vagy lehet folytonos halmaz is (regresszió).

Döntési fák

$x \in X$ diszkrét változók értékeinek vektora.

$f(x) \in Y$ egy diszkrét változó egy értéke. pl. $Y = \text{igen}, \text{nem}$

Mivel Y véges halmaz, osztályozási feladatról beszélünk, ahol X elemeit kell osztályokba sorolni, és az osztályok Y értékeinek felelnek meg.



Ez a döntési fa pl. azt határozza meg, hogy az adott jellemzők mellett (X) érdemes-e asztalra várni az étteremben (Y).

Ennek az előnye, hogy a döntések megmagyarázhatóak, emberikel értelmezhető a fenti ábra. Míg mesterséges neuron hálók esetében ez nem igaz.

Kifejezőerő

A kifejezőerő az **ítéletkalkulus**.

T.f.h. a címke logikai érték.

- **Ítéletek:** Változó értékkedások.
- **Modell:** Egy $x \in X$ változóvektor egy modell, mert minden ítélet igazságértékét megadja.
- **Formula:** A döntési fából logikai formula gyártható, és fordítva,

- Fából formula: Az "igen" címkekhez vezető utakban összeéseljük az éleket, majd összevagyoljuk az utakat.

- Formulából fa: A formula igazságátláját fel lehet írni fa alakban. Vegyük a változók egy A_1, \dots, A_n felsorolását, az A_1 a gyökérm A_1 értékei az élek, és az i . szinten a fában minden pontban A_i van, amely pontokból az élek A_i értékei. Az A_n változóból kivezető élek már levelek lesznek, értékük az igazságátlában található érték (a levéltől gyökérig vezető út meghatározza, az igazságátlába melyik sora kell).

Ez a faépítés nagy fákat eredményez, a gyakorlatban általában alkalmaznak tömörítési technikákat.

Döntési fa építése

Adottak ilyen felépítésű példák:

```
(  
  (  
    Vendégek=tele,  
    Várakozás=10-30,  
    Éhes=igen,  
    VanMásHely=nem,  
    Esik=igen,  
    Foglalás=nincs,  
    Péntek/Szombat=igen,  
    VanBár=ige  
  ),  
  igaz  
)
```

És ilyenekből minél több, legalább már száz.

Példákat bemagolni könnyű, pl. tekinthetjük a példákat az igazságátlábla ismert sorainak, és az alapján építünk fát a korábbiak szerint. Ismeretlen sorokhoz véletlenszerű értéket írhatunk. Ez konzisztens az adatokkal.

De a magolás nem általánosít!

Ötlet: A gyökérbe tesszük azt a változót, ami a legtöbb információt hordozza, ezáltal a legjobban szeparálja a példákat.

Majd rekurzívan a még nem rögzített változók közül megint választunk.

Speciális esetek, amikor megállítják a rekurziót:

- Ha csak pozitív, vagy negatív példa van, akkor levélhez értünk, megcímkezzük.
- Ha üres halmazról van szó, akkor egy alapértelmezett értéket definiálunk, pl. a szülőben levő többségi döntést.
- Ha pozitív, és negatív példa is van, de nincs több változó: A többségi szavazattal címkezzük. (Ez akkor fordul elő, ha zajos az adat)

A legjobban szeparáló attribútum

Egy p_1, \dots, p_n valószínűségi eloszlás várható (átlagos) információtartalma, más néven **entrópiája**:

$$H(p_1, \dots, p_n) = - \sum_i p_i \log(p_i)$$

ahol $\sum_i p_i = 1$

Ennek minimuma 0, ami a **maximális rendezettségi állapotot jelöli**. Amikor pontosan egy i -re $p_i = 1$, és $p_j = 0, i \neq j$.

A maximum pedig $-\log(1/n) = \log n$, ez a maximális rendezetlenség állapota.

Legyen egy példahalmazban n^+ pozitív, és n^- negatív példa. Ekkor a példahalmaz entrópiája

$$\$ H \left(\frac{n^+}{n^+ + n^-}, \frac{n^-}{n^+ + n^-} \right) = \frac{n^+}{n^+ + n^-} \log \frac{n^+}{n^+ + n^-} + \frac{n^-}{n^+ + n^-} \log \frac{n^-}{n^+ + n^-} \$$$

Információnyereség egy A változóra nézve:

$$H \left(\frac{n^+}{n^+ + n^-}, \frac{n^-}{n^+ + n^-} \right) - \sum_i \frac{n_i^+ + n_i^-}{n^+ + n^-} H \left(\frac{n_i^+}{n_i^+ + n_i^-}, \frac{n_i^-}{n_i^+ + n_i^-} \right)$$

Példahalmaz entrópiójának, és a az A változó lehetséges értékei szerinti felosztás után keletkező részhalmazok átlagos entrópijának a különbsége.

n_i^+ és n_i^- az A változó i . lehetséges értékét tartalmazó pozitív illetve negatív példák száma.

Ez alapján választható egy maximális nyereségű változó.

Zajszűrés az attribútumválasztásban

Problémák:

- Magolás

- **Túllilesztés:** Túl pontosan illesztjük az adatokra a modellt. Akkor, ha túl általános a modellünk, pl. egy lineáris közelítés helyett magas fokszámú polinom.

Zajszűrés: Megnézzük, hogy az **információnyereség statisztikailag szignifikáns-e**.

Pl.: χ^2 (**khí-négyzet**) próbával a χ^2 **metszés** algoritmusával.

naiv Bayes módszer

Bayes szabály

a és b kijelentésekre:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

Ebből következik: $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$

Általában is A és B változókra vagy változóhalmazokra:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Naiv Bayes algoritmus

Statisztikai következtető módszer.

Adatbázis-beli példák alapján példákat osztályoz

Legyen A célváltozó, és B_1, \dots, B_n a nyelvünk szavai.

Pl. A lehet igaz, ha az adott email spam, hamis, ha nem spam.

B_i pedig logikai változó, az i . szó előfordulását jelzi. Igaz, ha az emailben szerepel az adott szó.

A feladat egy adott b_1, \dots, b_n email esetében meghatározni, hogy A mely értékére lesz $P(A|b_1, \dots, b_n)$ feltételes valószínűség maximális.

Ehhez a következő átalakításokat, illetve függetlenségi feltevéseket tesszük:

$$P(A|b_1, \dots, b_n) = \alpha P(A)P(b_1, \dots, b_n|A) \approx \alpha P(A) \prod_{i=1}^n P(b_i|A)$$

Első egyenlőség: a Bayes tétele alkalmazása. $\alpha = 1/P(n_1, \dots, b_n)$.

Második közelítő egyenlőség: naiv Bayes feltevés. A pontatlanságért cserébe (ezért csak közelítő egyenlőség) $P(A)$ és $P(b_i|A)$ könnyen közelíthető az adatbázisban található példák segítségével.

Így gyakorlatban kiszámolható A minden lehetséges értékére, nagysági sorrend meghatározható. A legvalószínűbb értéket választjuk.

Modellillesztés

Adottak a $(x_1, y_1), \dots, (x_n, y_n) \subseteq X \times Y$ példák.

Egy $h^* : X \rightarrow Y$ függvényt keresünk, amely a példákra jól illeszkedik, és jól általánosít.

Optimalizációs megközelítés: Definiáljuk az $l : X \times Y \times H \rightarrow \mathbb{R}$ veszteségsfüggvényt, amely egy $(x, y) \in X \times Y$ példára megadja, hogy az adott $h \in H$ hipotézis "bennyi bajt okoz" az adott példán.

Példa veszteségsfüggvényre: Négyzetes hiba: $l(x, y, g) = (h(x) - y)^2$

Rögzített hibafüggvény esetén az optimalizálási feladat a következő:

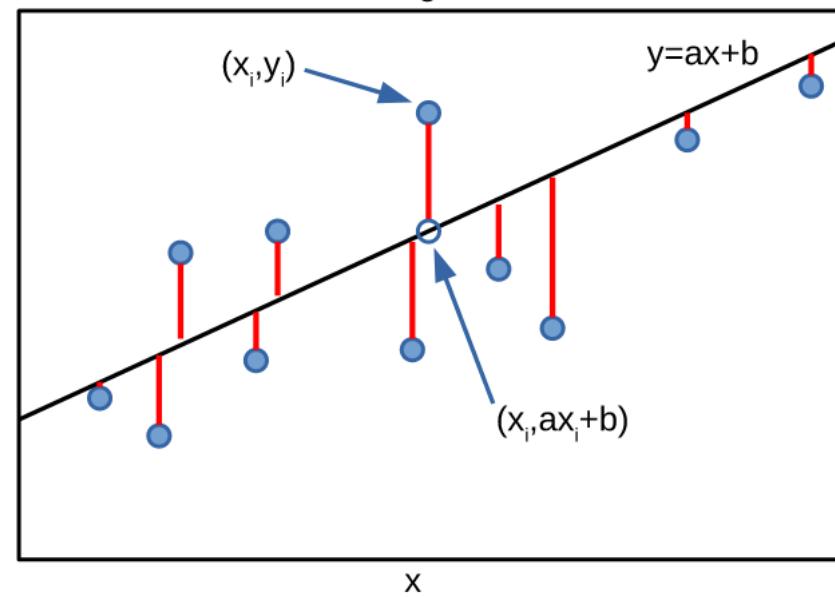
$$h^* = \arg \min_{h \in H} \sum_{i=1}^n l(x_i, y_i, h)$$

Lineáris regresszió

Három fő komponens:

- $H = h_{a,b}(x) = ax + b : a, b \in \mathbb{R}$, azaz lineáris, eltolást (b) is tartalmazó függvények halmaza.
- $l(x, y, h_{a,b}) = (h_{a,b}(x) - y)^2 = (ax + b - y)^2$, azaz a négyzetes hiba.
- Az optimalizáló algoritmus legyen a **gradiens módszer**.

Lineáris regresszió



Gradiens módszerek

Tegyük fel, hogy az $f : \mathbb{R} \rightarrow \mathbb{R}$ differenciálható.

Az $f'(x)$ derivált:

- 0, ha f lokális szélsőérték
- pozitív, ha f növekszik x -nél
- negatív, ha f csökken x -nél

A gradiens módszer alapja, hogy valamennyi tetszőleges x_0 -ból kiindulunk, majd kiszámoljuk az

$$x_{t+1} = x_t - \gamma_t f'(x_t)$$

képlettel az x_t sorozat elemeit, amelyre azt szeretnénk, hogy közelítse f egy lokális minimumát.

Megfelelő γ_t választása esetén ez garantált, azonban a pontos γ_t nem mindenkor ismert.

- | A gyakorlatban sokszor $\gamma_t = \gamma$ konstans értéket használjuk, ami elég kicsi ahhoz, hogy a rendszer konvergálni tudjon, vagy idővel csökkentjük γ_t -t
- | γ_t -t a gépi tanulásban **learning rate**-nek nevezzük.

Ha $f : \mathbb{R}^d \rightarrow \mathbb{R}$ egy d dimenziód differenciálható függvény, a fenti módszer változtatása nélkül alkalmazható, hiszen a $\nabla f(x) \in \mathbb{R}^d$ gradiens vektor a legnagyobb növekedés irányába mutat, így

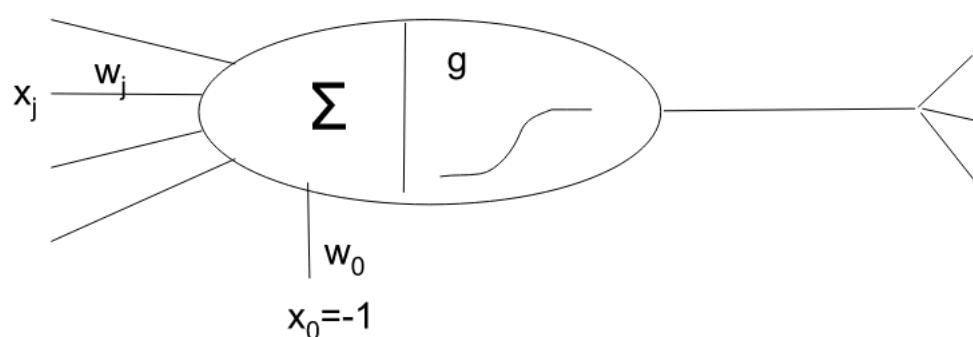
$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

hasonlóan viselkedik.

6.2.7. Mesterséges neuronhálók

Különálló neuron

bemeneti kapcsolatok	bemeneti függvény	aktivációs függvény	kimenet	kimeneti kapcsolatok
----------------------	-------------------	---------------------	---------	----------------------



Az x_j a j . bemeneti érték, a w_i a j . bemenet súlya.

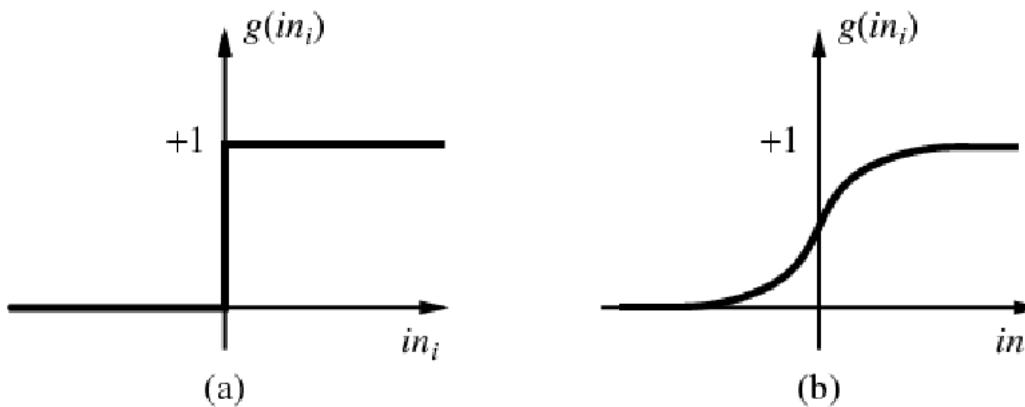
A w_0 az eltolássúly (bias weight), x_0 fix bemenet minden -1.

A neuron először a bemenetekből, és súlyokból kiszámolja a következő összeget:

$$\sum_{j=0}^d x_j w_j$$

Majd az összeg végeredményén alkalmazza az **aktivációs függvényt**.

Aktivációs függvény



Eredeti célja: Ha "jó" input jön, akkor adjon 1-hez közeli értéket, ha "rossz" input jön, akkor 0-hoz közelít.

Manapság már nem követelmény, hogy $[0, 1]$ -ből adjon értéket.

De fontos, hogy nemlineáris, ugyanis akkor magukkal a súlyokkal is kifejezhető lenne.

Pár példa:

- **Küszöbfüggvény**

- $g(x) = 0$ ha $x < 0$
- $g(x) = 1$ ha $x > 0$
- Ezzel az aktivációs függvénnnyel a neuron a klasszikus "**perceptron**"

- **Sigmoid függvény**

- $g(x) = \frac{1}{1+e^{-x}}$

- **Rectifier aktiváció**

- $g(x) = \max(0, x)$
- Ezt használó neuront **ReLU**-nak hívjuk

Neuron kimenete

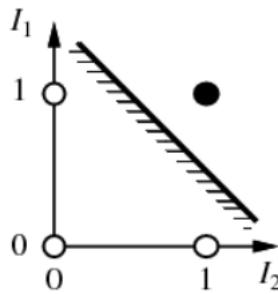
A neuron a teret két részre osztja egy hipersíkkal.

- $w \cdot x > 0$ esetén elfogadja az inputot
- különben nem

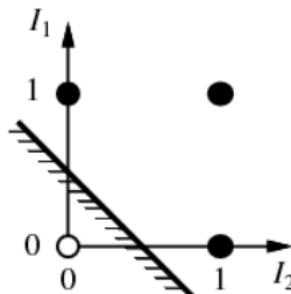
| $w \cdot x$ a w és x vektorok belső szorzata, ugyan az mint a fentebb leírt szumma, ami az aktivációs függvény inputja.

Emiatt csak lineárisan szeparálható függvények reprezentálhatóak hiba nélkül.

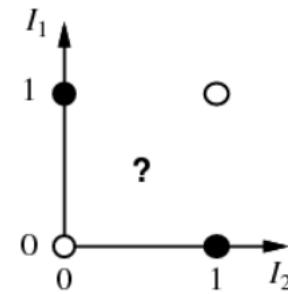
Példa logikai műveletek lineáris szeparálására:



(a) I_1 and I_2



(b) I_1 or I_2



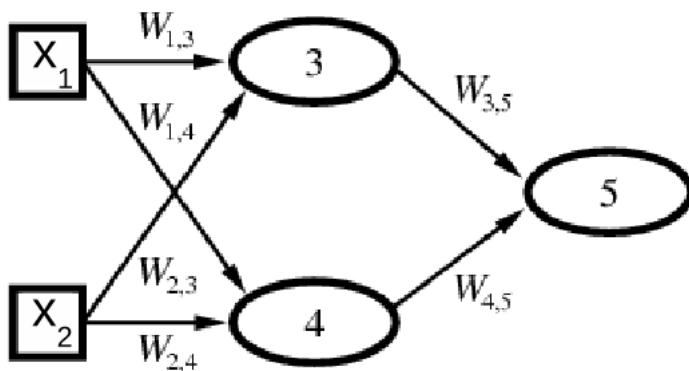
(c) I_1 xor I_2

Látszik, hogy a XOR nem lineárisan szeparálható, mert nem lehet olyan egyenest behúzni ami szeparálja az igaz, és hamis értékeket.

Többrétegű hálók

Neuronokból épített hálózat.

$$h_w(x_1, x_2) = g[w_{3,5}g(w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g(w_{1,4}x_1 + w_{2,4}x_2)]$$



x_1 és x_2 a bemenet.

Itt most nincsenek eltolássúlyok, de amúgy kellenének.

Réteg csoportok:

- **Bemeneti réteg:** Szögletes csúcsok alkotják, ez maga a bemenetek halmaza
- **Kimeneti réteg:** 5-ös számú neuron alkotja most egyedül, de lehetne több is, pl. 1-1 minden lehetséges osztálycímeknek
- **Rejtett rétegek:** minden másik köztes réteg

Egy egy előrecsatolt hálózat (a kapcsolatok az input irányától az output irányába mutatnak), de lehet olyat is építeni, amiben van kör, ezek a rekurrens hálózatok.

Többrétegű hálók algoritmusai

Optimalizáció alapú megközelítést alkalmazunk, ennek komponensei lehetnek pl.:

- $H = h_w : w \in \mathbb{R}^m$
 - Ahol $h_w : \mathbb{R}^d \rightarrow [0, 1]$ egy rögzített struktúrájú, w súlyokkal rendelkező neuronháló
- $l(x, y, h_w) = -\log P(y|x, h_w)$
 - Ez a h_w hipotézis **negatív log likelihoodja**
- Optimalizáló algoritmus legyen gradiens módszer

Visszaterjesztés (**backpropagation**): Valójában a gradiens kiszámolására szolgál. A kimeneten jelentkező veszteség visszaterjesztődik a rejtett neuronokra, ez alapján a súlyok állítása.

Több osztály esetén több kimeneti neuron, veszteségszámlálási függvény: **kereszt entrópia**

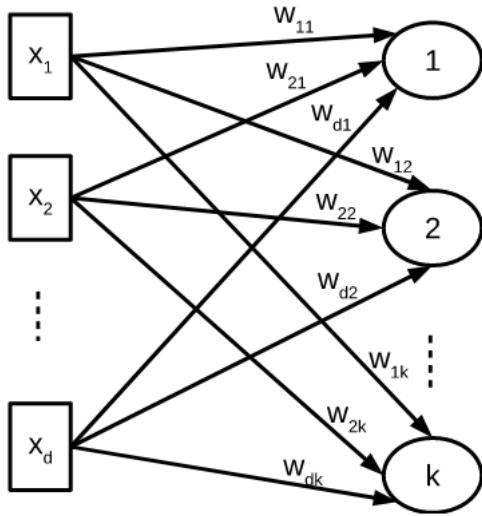
De ennek van bináris változata is (2 osztály)

Modern neuronháló-architektúrák

Teljesen összefüggő réteg

d dimenziós input k dimenziós inputra leképezése:

input teljesen összefüggő réteg



Összes input össze van kötve az összes kimenettel, összesen $d \cdot k$ kapcsolat, minden külön súlyval.

Tehát összesen eltolási súlyokkal együtt $d \cdot k + k$ súly van.

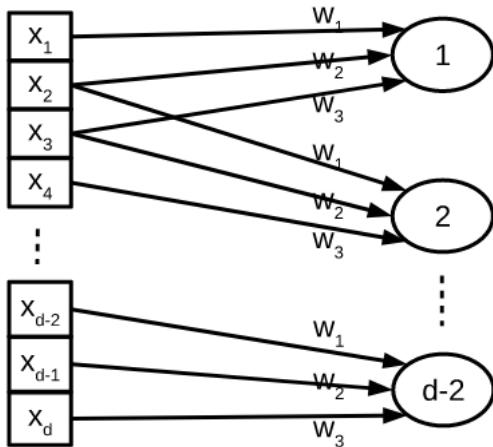
Konvoluciós réteg

Ha az inputban fontos a szomszédsági / sorrendiségi reláció.

Például képek, hangok esetén.

Egy 1D struktúrájú inputtal (A.K.A egy vektorral) működő konvoluciós réteg:

input 1D konvoluciós réteg



lépésköz=1, szűrőmérét=3

- Réteg azonos neuronokból áll, a súlyaik megegyeznek
- minden neuron csak egy rögzített összefüggő sávot lát az inputból

Az ábrán látható réteg például leírható 3db súlyjal. Ami a teljesen összefüggő réteghet képest nagyon kevés.

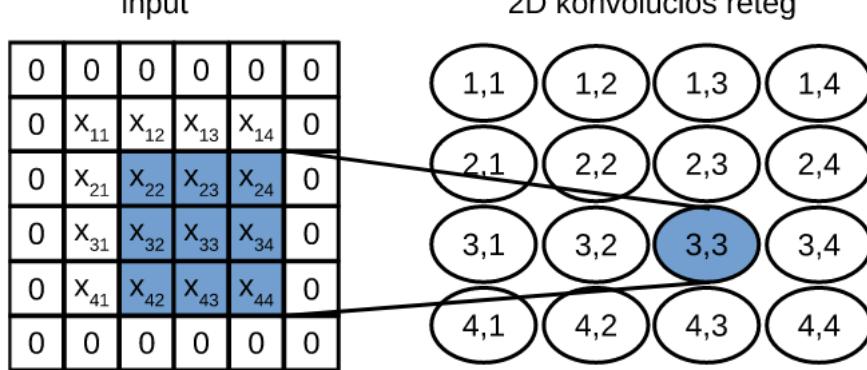
3 szélességű szűrővel végigpásztazzuk az inputot.

Paraméterek: lépésköz (**stride**), illetve a szűrőmérét (F)

Ezek a paraméterek meghatározzák az output dimenzióját, ami $\lceil (d - F + 1) / S \rceil$. Ha $F > 1$, akkor ez mindenkor kevesebb, mint az input dimenzió.

Ha nem akarjuk, hogy csökkenjen az input, lehet **zero padding**-et használni

Példa 2D-s inputtal, zero padding-el:

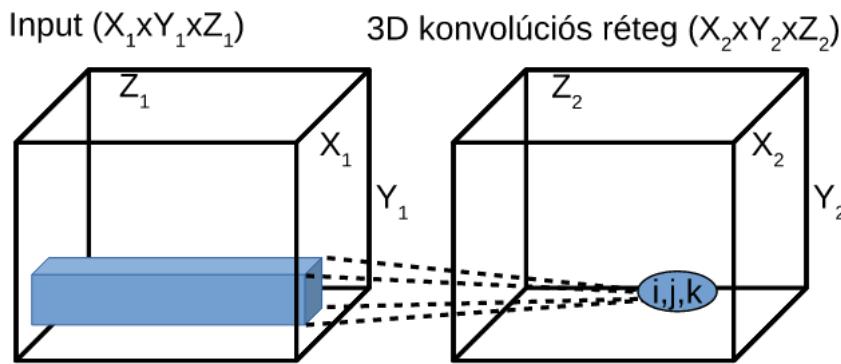


lépésköz=1, szűrőméret=3x3, nulla feltöltés=1

Neuronok is 2D elrendezésben.

Ez a réteg 9 súlyval leírható (3×3 -as szűrő)

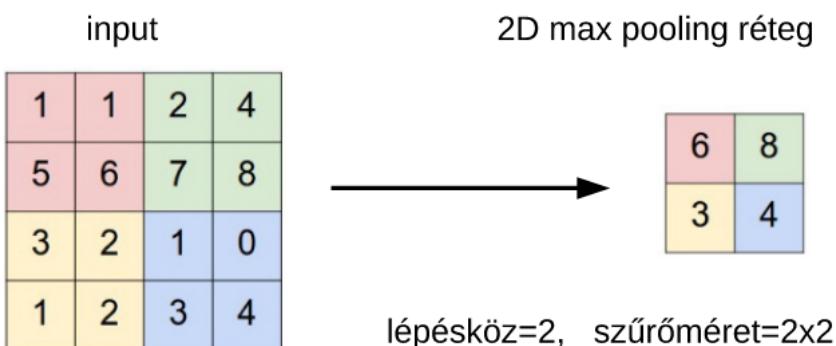
3D input, a 3. dimenzió, a mélység lehet pl. színes képek esetén 3, az RGB színeknek:



Max pooling réteg

Input méretének csökkentésére szolgál.

Például 2D-ben egy 2×2 -es max szűrő 2-es lépésközzel:



6.2.8. k-legközelebbi szomszéd módszere

Példányalapú tanuló algoritmus.

Adottak $(x_1, y_1), \dots, (x_n, y_n)$ példák.

Adott x -re az y -r az x -hez közelíti a példák alapján határozzuk meg.

1. Keressük meg x kdb legközelebbi szomszédját (k pl. 5 és 10 között, lehetőleg páratlan, hogy ne legyen holtverseny)

2. A $h(x)$ értéke ezen szomszédok y -jainak átlaga (esetleg távolsággal súlyozva) ha y folytonos, ha diszkrét akkor pedig pl. többségi szavazat.

Sűrűség közelítése: Ha x_1, \dots, x_n példák adottak (y nélkül), akkor a $P(X)$ sűrűség közelítésére is jó: adott x -re nézzük meg, hogy a k legközelebbi szomszéd mekkora területen oszlik el. $P(x)$ ekkor fordítottan arányos a terüettel.

Távolság függvény: $D(x_1, x_2)$.

- Diszkrét esetben, pl. Hamming-távolság: a különböző jellemzők száma

- Folytonos esetben pl. euklideszi távolság, manhattan távolság, stb.

- Folytonos jellemzőket normalizálni kell, mert egyes jellemző értékek más skálán mozoghatnak (pl. hőmérséklet, magasság).
- Standardizálás: Jellemzőkből kivonjuk az átlagot, és elosztjuk a szórással.

Az algoritmus hibái:

- Érzékeny a távolságfüggvény definíciójára
- Sok példa esetén költséges a k legközelebbi szomszédot megtalálni
- Sok jellemző esetén (sok dimenziós térben) a távolságfüggvény nagyon nem intuitív

7. Operációkutatás

7.1. 1. LP alapfeladat, példa, szimplex algoritmus, az LP geometriája, generálóelem választási szabályok, kétfázisú szimplex módszer, speciális esetek (cikлизáció-degeneráció, nem korlátos feladat, nincs lehetséges megoldás)

7.1.1. Alapfogalmak

- Döntési változók:** $x_1, x_2, \dots, x_i, \dots$
- Változók értelmezési tartománya:** $x_1, x_2 \geq 0$
- Cél:** min / max probléma
- Célfüggvény** (max / min): $2x_1 + 5x_2$
- Korlátozások** (egyenletek, egyenlőtlenségek): $3x_1 + 2x_2 \leq 10$

7.1.2. LP alapfeladat

$$\begin{array}{ll}
 \text{Max} & c_1x_1 + c_2x_2 + \dots + c_nx_n = z \\
 \text{Felt.} & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\
 & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\
 & \vdots \\
 & a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\
 & x_1, \dots, x_n \geq 0
 \end{array}$$

- Lineáris programozási feladat:** Keressük meg adott lineáris \mathbb{R}^n értelmezési tartományú függvény (célfüggvény) szélsőértékét (minimumát, vagy maximumát) értelmezési tartományának adott lineáris korlátokkal (feltételekkel) meghatározott részében.
- Lehetséges megoldás:** Olyan $p = (p_1, \dots, p_n) \in \mathbb{R}^n$ vektor, hogy p_i -t x_i -be helyettesítve ($\forall i = 1, \dots, n$) kielégíti a feladat feltételrendszerét.
- Lehetséges megoldási tartomány:** Az összes lehetséges megoldás (vektor) halmaza.
- Optimális megoldás:** Olyan lehetséges megoldás, ahol a célfüggvény felveszi a minimumát / maximumát.

Általános alak tömöre

$$\sum_{j=1}^n a_{ij}x_j \leq b_i \quad i = 1, 2, \dots, m \quad x_j \geq 0 \quad j = 1, 2, \dots, n \quad \text{--- --- --- --- ---} \quad \max \sum_{j=1}^n c_jx_j = z$$

Standard alakban minden feltétel \leq maximalizálás esetén, \geq minimalizálás esetén. Illetve minden változó nemnegatív.

7.1.3. Példa

Játékgyártó cék kétféle terméket gyárt:

- Katonákat**
 - \$10 anyagköltség
 - \$14 munkadíj
 - 1 órafafaragás
 - 2 órakozás-festés

◦ Eladási ár: \$27

• Vonatokat

- \$9 anyagköltség
- \$10 munkaadó
- 1 óra fafaragás
- 1 óra lakközös-festés
- Eladási ár: \$21

Erőforrások:

- Fafaragó műhely: 80 munkaóra
- Lakközös-festés: 100 munkaóra

Extra megkötés: A cég legfeljebb 40 katonát akar gyártani.

LP feladat

$$\max z = 3x_1 + 2x_2 \quad x_1 + x_2 \leq 80 \quad 2x_1 + x_2 \leq 100 \quad x_1 \leq 40 \quad x_1, x_2 \geq 0$$

Lehetséges megoldás: $x = (20, 20)$

Azaz 20 katona, 20 vonat

Optimális megoldás: $x^* = (20, 60)$

Optimum értéke: $z^* = 180$ (Célfüggvénybe helyettesítve: $3 * 20 + 2 * 60$)

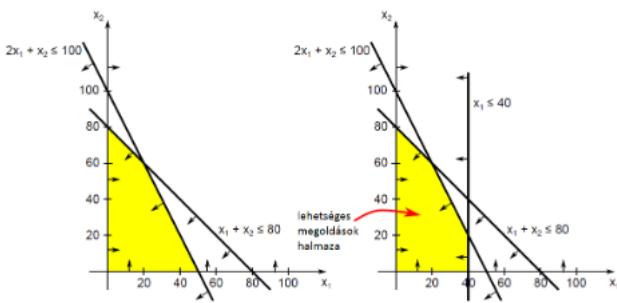
Azaz \$180 profiot érhet el a cég

7.1.4. Egy lineáris program felírása

1. Válasszuk meg a **döntési változókat**
2. Határozzuk meg a **célt**, és a **célfüggvényt** (lineáris függvény)
3. Írjuk fel a **korlátozó feltételeket** (lineáris egyenlőtlenségek)
4. Határozzuk meg a **változók értelmezési tartományát** (előjel feltételek)

7.1.5. LP feladat megoldása

Csúcs (extremális) pont: Két egyenes metszéspontja, ha a korlátozó feltételek által meghatározott lehetséges megoldások halmazát az alábbi módon ábrázoljuk:



Tétel: Ha egy LP feladatnak van optimális megoldása, akkor olyan optimális megoldása is van, ami a lehetséges megoldási tartomány csúcspontja.

Ötlet: Keressük meg a csúcspontokat, értékeljük ki az összes helyen, és vegyük a maximumot.

Ezzel az a probléma, hogy **sok ilyen csúcspont lehet**.

Mesterséges változók

Annak érdekében, hogy az egyenlőtlenségeket egyenlőségekre cserélhessük, mesterséges változókat adunk az egyenlőtlenséges bal oldalaihoz:

$$\text{Max } z = 3x_1 + 2x_2$$

$$\begin{aligned} x_1 + x_2 + x_3 &= 80 \\ 2x_1 + x_2 + x_4 &= 100 \\ x_1 + x_5 &= 40 \\ x_1, \dots, x_5 &\geq 0 \end{aligned}$$

Az az egyenletrendszer, amiben a mesterséges változókat kifejeztük:

A mesterséges változókkal bővített általános feladat:

$$\begin{array}{lclllll} a_{11}x_1 & + & a_{12}x_2 & + \dots & + & a_{1n}x_n & + & \textcolor{blue}{x_{n+1}} & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + \dots & + & a_{2n}x_n & + & \textcolor{blue}{x_{n+2}} & = & b_2 \\ & & & & & & & \vdots & & \\ a_{m1}x_1 & + & a_{m2}x_2 & + \dots & + & a_{mn}x_n & + & \textcolor{blue}{x_{n+m}} & = & b_m \\ \hline c_1x_1 & + & c_2x_2 & + \dots & + & c_nx_n & & & & = z \end{array}$$

Ebből a szótár:

$$\begin{array}{lclllll} \textcolor{blue}{x_{n+1}} & = & b_1 & - & a_{11}x_1 & - & a_{12}x_2 & - \dots & - & a_{1n}x_n \\ \textcolor{blue}{x_{n+2}} & = & b_2 & - & a_{21}x_1 & - & a_{22}x_2 & - \dots & - & a_{2n}x_n \\ & & & & & & & \vdots & & \\ \textcolor{blue}{x_{n+m}} & = & b_m & - & a_{m1}x_1 & - & a_{m2}x_2 & - \dots & - & a_{mn}x_n \\ z & = & & & c_1x_1 & + & c_2x_2 & + \dots & + & c_nx_n \end{array}$$

- Bázisváltozók:** A szótár feltétel egyenleteinek bal oldalán álló változók
- Nembázis változók:** A szótár feltételeinek jobb oldalán álló változók
- Szótár bázismegoldása:** Olyan x vektor, amelyben a nembázis változók értéke nulla, (ezért) a bázisváltozók értékei az öket tartalmazó egyenletek jobb oldali konstansai.
- Lehetséges (feasible) bázismegoldás:** Olyan bázismegoldás, ami egyben lehetséges megoldás is, azaz a szótárra teljesül, hogy $b_i \geq 0$, $i = 1, 2, \dots, m$ a bázismegoldásban.

7.1.6. Szimplex algoritmus

Adott standard alakú LP feladat, hozzá szótár.

T.h.f. $b_1 \geq 0, \dots, b_m \geq 0$, azaz a szótár bázismegoldása lehetséges megoldás.

Ez a feltétel nem része a szimplexnek, negatív jobb oldalú feltételekről később.

Az algoritmus **iteratív optimum keresés**.

Ismételt áttérés más szótárakra a következő feltételek mellett:

- Minden iteráció szótára ekvivalens az előző iterációval
- Minden iteráció szótárának a bázismegoldásán a célfüggvény értéke nagyobb, vagy egyenlő, mint az előző iterációén
- Minden iteráció bázismegoldása lehetséges megoldás

Pivot lépés: Új szótár megadása egy bázis és nembázis változó szerepének felcserélésével.

- Belépőváltozó:** A szimplex algoritmus egy iterációjának belépőváltozója az a nembázis változó, ami a következő szótárra áttérés hatására bázisváltizóná válik
- Kilépőváltozó:** A szimplex algoritmus egy iterációjának kilépőváltozója az a bázisváltozó, ami a következő szótárra áttérés hatására nembázis változóná válik

Szótárak ekvivalenciája: Két szótár ekvivalens, ha az általuk leírt egyenletrendszer összes lehetséges megoldásai és a hozzájuk tartozó célfüggvényértékek rendre megegyeznek

Tétel: A pivot lépés előtti és az utána előálló új szótár ekvivalensek.

- Egyenleteket átrendezni ekvivalens átalakítás
- Egy egyenlethez egy másik konstans-szorosát hozzáadni ekvivalens átalakítás

Hányadosteszt

Így döntjük el, hogy melyik egyenletből fejezzük ki a belépőváltozót.

A legszűkebb korlátot adó egyenletből fejezzük ki a belépő változót.

Azon előfordulások közül, ahol a belépő változó negatív előjelű együtthatóval szerepel, kiválasztjuk azt ahol a $(b_i / |a_{ij}|)$ a legkisebb.

Ez a pozitív együtthatóval szerepel, az nem ad korlátot a változóra!

Optimális bázismegoldás

Honnan tudjuk, hogy az aktuális bázismegoldás optimális?

Tétel: Ha egy szótárban nincs pozitív $c_j = (j = 1, 2, \dots, n + m)$ célfüggvény együttható és negatív $b_i (i = 1, 2, \dots, m)$ konstans a feltételes egyenleteiben, akkor a szótár bázismegoldása optimális megoldás.

Az algoritmus

Input: Egy lehetséges induló szótár

Output:

- $x^* = (x_1^*, \dots, x_n^*)$ optimális bázismegoldás

Azaz a szótár bázismegoldása lehetséges megoldás

- z^* a célfüggvény optimuma

Lépések:

1. A szótárban $c_j \leq 0$ minden $j = 1, 2, \dots, n$ -re?

i. **Igen:** Az aktuális bázismegoldás **optimális**, az algoritmus megáll

ii. **Nem:** Folytatás 2. ponttal

2. Válasszunk a nembázis változók közül belépőváltozónak valamelyik x_k -t, amelyre $c_k > 0$ (pozitív célfüggvény együttható)

3. $-a_{ik} \geq 0$ minden $i = 1, 2, \dots, m$ -re?

i. Igen: Az LP feladat nem korlátos, az algoritmus megáll

ii. Nem: Folytatás 4. ponttal

4. Legyen l valamely index, amelyre $-a_{lk} < 0$ és $\left| \frac{b_l}{a_{lk}} \right|$ minimális

Ez a **hányadosteszt**

5. Hajtsunk végre egy pivot lépést úgy, hogy x_k legyen a belépőváltozó, és az l . feltétel bázisváltozója legyen a kilépő

Generálóelem választási szabályok (Pivot szabályok)

Olyan szabály, ami egyértelművé teszi, hogy a szimplex algoritmusban mely változók legyenek a belépő-, és a kilépőváltozók, ha több változó is teljesíti az alapfeltételeket.

Klasszikus Szimplex algoritmus pivot szabály

- Lehetséges belépő változók közül válasszuk a legnagyobb c_k értékűt, több ilyen esetén azok közül a legkisebb indexűt
- Lehetséges kilépő változók közül (hányadosteszt!) válasszuk a legkisebb l indexűt.

Bland szabály

- Lehetséges belépő változók közül **legkisebb indexűt**, ÉS
- Lehetséges kilépő változók közül **legkisebb indexűt** választjuk

Tétel: Ezzel a szabállyal az algoritmus véget ér, azaz a ciklizáció elkerülhető.

Lexikografikus szabály

Ciklizáció oka a degeneráció, azaz bázisváltozó 0 értékűvé válása a bázismegoldásban.

- **Adjunk** kezdő szótár minden feltételének jobb oldalához egy pozitív ϵ_i **konstanst**
- minden feltételre legyenek ezek **különböző nagyságrendűek** a következőképpen: $0 < \epsilon_1 \ll \epsilon_2 \ll \dots \ll \epsilon_m$
- így a szimplex algoritmus során biztosan **nem ütik ki egymást**
- **Nem lesznek degenerált bázismegoldások**, így nincs ciklizáció
- Az utolsó, optimális szótárból elhagyjuk az ϵ -okat
- Lehetséges belépőváltozók közül válasszuk a **legnagyobb c_k értékűt**, több ilyen esetén azok közül a **legkisebb indexűt**
- Lehetséges kilépőváltozók közül válasszuk azt, amelynek l indexű egyenletére az együtthatókból álló vektor **lexikografikusan a legkisebb**

Tétel: Ezzel a szabállyal az algoritmus véget ér, azaz a ciklizáció elkerülhető.

- Példa: feladat induló szótára (látjuk, hogy degenerált)

$$\begin{array}{rcl} x_3 & = & \frac{1}{2} - x_2 \\ x_4 & = & - 2x_1 + 4x_2 \\ x_5 & = & x_1 - 3x_2 \\ x_6 & = & - x_1 + 6x_2 \\ \hline z & = & 4 + 2x_1 - x_2 \end{array}$$

- Szótár konkrét értékek helyett szimbolikus ε -okkal kiegészítve

$$\begin{array}{rcl} x_3 & = & \frac{1}{2} + \varepsilon_1 - x_2 \\ x_4 & = & + \varepsilon_2 - x_1 + x_2 \\ x_5 & = & + \varepsilon_3 + x_1 - 3x_2 \\ x_6 & = & + \varepsilon_4 - 2x_1 + 4x_2 \\ \hline z & = & 4 + 2x_1 - x_2 \end{array}$$

Speciális esetek

Nem korlátos LP feladat

Ha az LP feladat maximalizálálandó (minimalizálálandó), és célfüggvénye tetszőlegesen nagy (kicsi) értéket felvehet a lehetséges megoldások halmazán, akkor a feladatot nem korlátosnak nevezzük.

Tétel: Ha egy szótárban van olyan pozitív $c_j (j = 1, 2, \dots, n+m)$ célfüggvény együttható, hogy minden $-a_{ij} (i = 1, 2, \dots, m)$ együttható nemnegatív, akkor az LP feladat, amihez a szótár tartozik, nem korlátos.

Ciklizáció

Degenerált iterációs lépés: Olyan szimplex iteráció, amelyben nem változik a bázismegoldás.

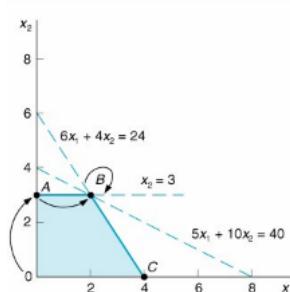
Degenerált bázismegoldás: Olyan bázismegoldás, amelyben egy vagy több bázisváltozó értéke 0.

Ciklizáció: Ha a szimplex algoritmus valamely iterációja után egy korábbi iteráció szótárát kapjuk meg, akkor azt ciklizációknak nevezzük.

Tétel: Ha a szimplex algoritmus nem áll meg, akkor ciklizál.

A ciklizáció **elkerülhető** megfelelő pivot szabály alkalmazásával.

Ciklizáció szemléletesen: **több** korlátozó feltételhez tartozó egyenes **megy át ugyan azon a ponton**:



7.1.7. Kétfázisú szimplex algoritmus

Ha egy szótárban minden $b_i \geq 0 (i = 1, 2, \dots, m)$, akkor mehet a szimplex algoritmus.

Ellenkező esetben vezessünk be egy új mesterséges változót (x_0) és tekintsük a következő segédfeladatot:

$$\begin{array}{ll} \text{Max } w = & -x_0 \\ & 2x_1 - x_2 + x_3 - x_0 \leq 4 \\ & 2x_1 - 3x_2 + x_3 - x_0 \leq -5 \\ & -x_1 + x_2 - 2x_3 - x_0 \leq -1 \\ & x_1, x_2, x_3, x_0 \geq 0 \end{array}$$

x_4, x_5, x_6 mesterséges változók bevezetésével:

$$\begin{array}{ll} \text{Max } w = & -x_0 \\ & 2x_1 - x_2 + x_3 - x_0 + x_4 = 4 \\ & 2x_1 - 3x_2 + x_3 - x_0 + x_5 = -5 \\ & -x_1 + x_2 - 2x_3 - x_0 + x_6 = -1 \\ & x_i \geq 0 \end{array}$$

Vegyük a legnegatívvabb jobboldalú egyenletet (2-es), és fejezzük ki x_0 -t ebből, a többiből a mesterséges változókat.

Tétel: A standard feladatnak akkor, és csak akkor létezik lehetséges megoldása, ha $w = 0$ a hozzá felírt segédfeladat optimuma.

Ha a segédfeladatot megoldjuk a szimplex algoritmussal és annak optimuma $w = 0$, akkor a megoldás legutolsó szótárából könnyen felírhatunk egy olyan szótárat, amely:

- Az eredeti feladat szótára
- Bázismegoldása lehetséges megoldás is egyben

A szótár felírásának lépései:

- Ha $x_0 = 0$ szerepel a feltételek között, akkor elhagyjuk
- Ha x_0 bázisváltozó, akkor az egyenleténak jobb oldalán levő nem 0 együtthatójú változók valamelyikét belépőváltozónak, x_0 -t kilépőváltozónak tekintve végrehajtunk egy pivot lépést
- Elhagyjuk x_0 megmaradt előfordulásait
- A célfüggvény egyenletét lecseréljük az eredeti célfüggvényre, amit átírunk az aktuális bázisváltozónak megfelelően

Az így megkapott lehetséges szótárból kiindulva indítjuk a szimplex algoritmust.

7.1.8. LP és konvex geometria

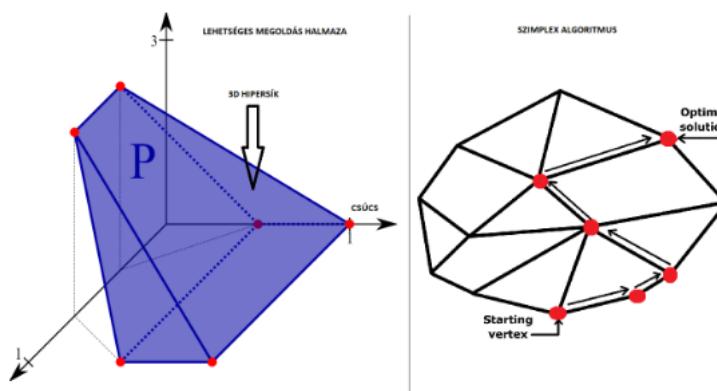
Az olyan fogalmaknak, mint bázismegoldás, lineáris feltétel, lehetséges megoldások halmaza, megfeleltethető egy-egy geometriai objektum.

R^n : n dimenziós lineáris tér a valós számok felett, elemei az n elemű valós vektorok.

E^n : n dimenziós euklideszi tér, olyan lineáris tér, amelyben értelmezett egy belső szorzat, és egy távolság függvény a következő módon:

- $\langle x, y \rangle = x^T y = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$
- $\|x\| = \sqrt{\langle x, y \rangle}$ norma
- $d(x, y) = \|x - y\|_2 = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$

Korlátozások által kijelölt csúcsok a térben:



A **lineáris feltételek** zárt féltereket (\leq) és síkokat ($=$) határoznak meg

A **lehetséges megoldások halmaza** ezen félterek és síkok metszete

Poliéder: Zárt, véges sok csúcsponttal rendelkező ponthalmaz

A lehetséges megoldások halmaza egy konvex poliéder.

Bázismegoldások = poliéder csúcsok.

Ciklizáció: az adott csúcsban ragadunk.

Oka, hogy a csúcspont leírására használt síkokat cserélgetjük.

7.2. 2. Primál-duál feladatpár, dualitási komplementaritási tételek, egész értékű feladatok és jellemzőik, a branch and bound módszer, a hátizsák feladat

7.2.1. Primál-duál feladatpár

Játékgyáras példához:

Legyen egy egység fa piaci ára y_1 , egy egység festék ára y_2

A gyártó opciói:

- Eladhatja az erőforrásokat piaci áron
- Vehet további fát, festéket
- Gyárt a rendelkezésre álló erőforrásokból, és eladja a játékokat

A készletet $80y_1 + 100y_2$ -ért lehetne eladni

Két eset:

- Ha egy katona alapanyagára kisebb, mint az eladási, azaz $y_1 + 2y_2 < 3$
 - Ekkor a gyártó nem adja el az erőforrásokat
- Ha az alapanyagára nagyobb, azaz $y_1 + 2y_2 \geq 3$
 - Ekkor a gyártó eladhatja az erőforrásait

Duális feladat a játékgyár példára

Ezen példa esetén az alapanyag-felvásárló szemszögéből megoldott felatát:

A primál feladat:

$$\begin{array}{rcl}
 x_1 & + & x_2 \leq 80 \\
 2x_1 & + & x_2 \leq 100 \\
 & & x_1, x_2 \geq 0 \\
 \hline
 \text{Max } z = 3x_1 & + & 2x_2
 \end{array}$$

A duál feladat:

$$\begin{array}{rcl}
 y_1 & + & 2y_2 \geq 3 \\
 y_1 & + & y_2 \geq 2 \\
 & & y_1, y_2 \geq 0 \\
 \hline
 \text{Min } w = 80y_1 & + & 100y_2
 \end{array}$$

Ez az eredeti feladat **duálisa**.

Maga az eredeti feladat a **primál** feladat.

Duális feladat általánosan

A primál feladat:

$$\begin{array}{rcl}
 \mathbf{Ax} & \leq & \mathbf{b} \\
 \mathbf{x} & \geq & \mathbf{0} \\
 \text{Max } \mathbf{c}^T \mathbf{x} & = & z
 \end{array}$$

A duál feladat:

$$\begin{array}{rcl}
 \mathbf{A}^T \mathbf{y} & \geq & \mathbf{c} \\
 \mathbf{y} & \geq & \mathbf{0} \\
 \text{Min } \mathbf{b}^T \mathbf{y} & = & w
 \end{array}$$

Állítás: Duál feladat duálisa az eredeti primál feladat.

A duális feladat megoldásában y_i^* az eredeti (primál) feladat *i*. erőforrásához tartozó piaci ár, amit marginális árnak vagy árnyék árnak nevezünk

- Ez az erőforrás éérke az LP megoldójának szemszögéből

- Az i erőforrás mennyiségeinek egy egységnyi növelésével éppen y_i^* -gal nő a nyereség (azaz a célfüggvény értéke)
- y_i^* -nál többet már nem érdemes fizetni az i erőforrásért, de kevesebbet persze igen

Dualitási tételek

Gyenge dualitás tétele

Ha $x = [x_1, \dots, x_n]^T$ lehetséges megoldása a primál feladatnak és $y = [y_1, \dots, y_m]^T$ lehetséges megoldása a duál feladatnak, akkor:

$$c^T x \leq b^T y$$

Azaz a duális feladat bármely lehetséges megoldása feéső korlátot ad a primál bármely lehetséges megoldására (így az optimális megoldásra is)

Erős dualitás tétele

Ha van a primál feladathak optimális megoldása, $x^* = (x_1^*, \dots, x_n^*)$, akkor a duál feladatnak is létezik optimális megoldása, $y^* = (y_1^*, \dots, y_m^*)$, és a célfüggvényértékük megegyezik, azaz $c^T x^* = b^T y^*$

Lemma

Legyen $x^* = [x_1^*, \dots, x_n^*]^T$ lehetséges megoldása a primál feladatnak, és $y^* = [y_1^*, \dots, y_m^*]^T$ lehetséges megoldása a duál feladatnak.

Ha $c^T x^* = b^T y^*$, akkor x^* a primál feladat optimális megoldása, y^* pedig a duál feladat optimális megoldása.

Ez a gyende dualitási tételeből ered

Segédtételek

- Ha a primál feladat célfüggvénye nem korlátos, akkor a duál feladatnak nincs lehetséges megoldása
- Ha a duál feladat célfüggvénye nem korlátos, akkor a primál feladatnak nincs lehetséges megoldása

Összefüggések primál és duál között

Primál

Duál	Nincs lehetséges megoldása	Van optimális megoldása	Nem korlátos
Nincs lehetséges megoldása	✓	-	✓
Van optimális megoldása	-	✓	-
Nem korlátos	✓	-	-

Dualitásból adódó lehetőségek

- Szimplex iterációszáma közelítőleg a sorok számával arányos, így sok feltétel, kevés változó esetén érdemes áttérni a duálusra
- Ha primál esetben 2 fázisra van szükség, míg duális esetén csak egyre, akkor is érdemes áttérni
- Ha menet közben kell új feltételeket hozzávenni az LP-hez, akkor a duál feladattal dolgozva az új feltétel csak egy új, nembázis változóként jelenik meg, hozzávesszük az aktuális szótárhoz, és folytatjuk a feladatmegoldást

7.2.2. Komplementaritás

A primál:	A duál:	A primál kieg.:	A duál kieg.:
$\begin{array}{l} Ax \leq b \\ x \geq 0 \\ c^T x \rightarrow \max \end{array}$	$\begin{array}{l} A^T y \geq c \\ y \geq 0 \\ b^T y \rightarrow \min \end{array}$	$\begin{array}{l} Ax + I_{xs} = b \\ x, xs \geq 0 \\ c^T x \rightarrow \max \end{array}$	$\begin{array}{l} A^T y - I_{ye} = c \\ y, ye \geq 0 \\ b^T y \rightarrow \min \end{array}$

Azt mondjuk, hogy $x = (x_1, \dots, x_n)$ és $y = (y_1, \dots, y_n)$ komplementárisak, ha

$$y^T(b - Ax) = 0 \text{ és } x^T(A^T y - c) = 0$$

vagy a kiegészített feladatra nézve:

$$y^T x s = 0 \text{ és } x^T y e = 0$$

Vagyis

- Ha $y_i > 0$, akkor i -edik egyenletbe helyettesítve $=$ -et kapunk, azaz $x_{n+i} = 0$
 - "a feltétel éles"
- Ha az i -edik feltétel nem éles, azaz $x_{n+i} > 0$, akkor $y_i = 0$

Hiszen a $y^T x s = 0$ és a $x^T y e = 0$ egyenletek esetén ez a két lehetőség van, hogy egyenlőség legyen

Komplementaritási tételek

Tegyük fel, hogy x a primál, y a duál feladat lehetséges megoldása. Az x és y akkor, és csak akkor optimálisak, ha komplementárisak is.

Ezért ha x a primál optimális megoldása, akkor igazak:

- Ha y a duál optimális megoldása, akkor x és y komplementárisak
- Létezik olyan lehetséges y megoldása a duálisnak, hogy x és y komplementáris. Ekkor y optimális is.

Komplementáris lazáság

- Adott x , egy javasolt primál megoldás, ellenőrizzük, hogy lehetséges-e
- Nézzük meg, mely y_i változóknak kell 0-nka lennie
- Nézzük meg, mely duál feltételnek kell élesnek lennie, így egy egyenletrendszert kapunk
- Oldjuk meg ezt a rendszert
- Ellenőrizzük, hogy a kapott megoldás lehetséges megoldása-e a duálisnak

Ha minden lépés sikeres volt, akkor az adott x optimális különben nem

7.2.3. Egész értékű programozás

Olyan LP feladat, amiben valamennyi változó egész értékű

- **IP:** Tiszta egészértékű programozási feladat, ekkor minden változó egész értékű
- **MIP:** Vegyes egészértékű programozási feladat, ekkor csak néhány változóra követelünk egészértékűséget
- **0-1 IP:** minden változó bináris, értéke csak 0 vagy 1 lehet

Lehetséges megoldások halmaza



- LP: Szürke háromszög
- MIP: Sárga szakaszok
- IP: Fehete pontok

LP-lazítás

Egy egészértékű programozási feladat LP-lazítása az az LP, amelyet úgy kapunk az IP-ből, hogy a változókra tett minden egészértékűségi vagy bináris megkötést eltörölünk

Állítások az LP-lazításról:

- Bármelyik IP lehetséges megoldáshalmaza része az LP-lazítása lehetséges megoldástartományának
- Maximalizálásnál az LP-lazítás optimum értéke \geq az IP optimum értékénél
- Ha az LP-lazítás lehetséges megoldáshalmazának minden **csúcspontja** egész, akkor van egész optimális megoldása, ami az IP megoldása is egyben
- Az LP-lazítás optimális megoldása bármilyen messze lehet az IP megoldásától

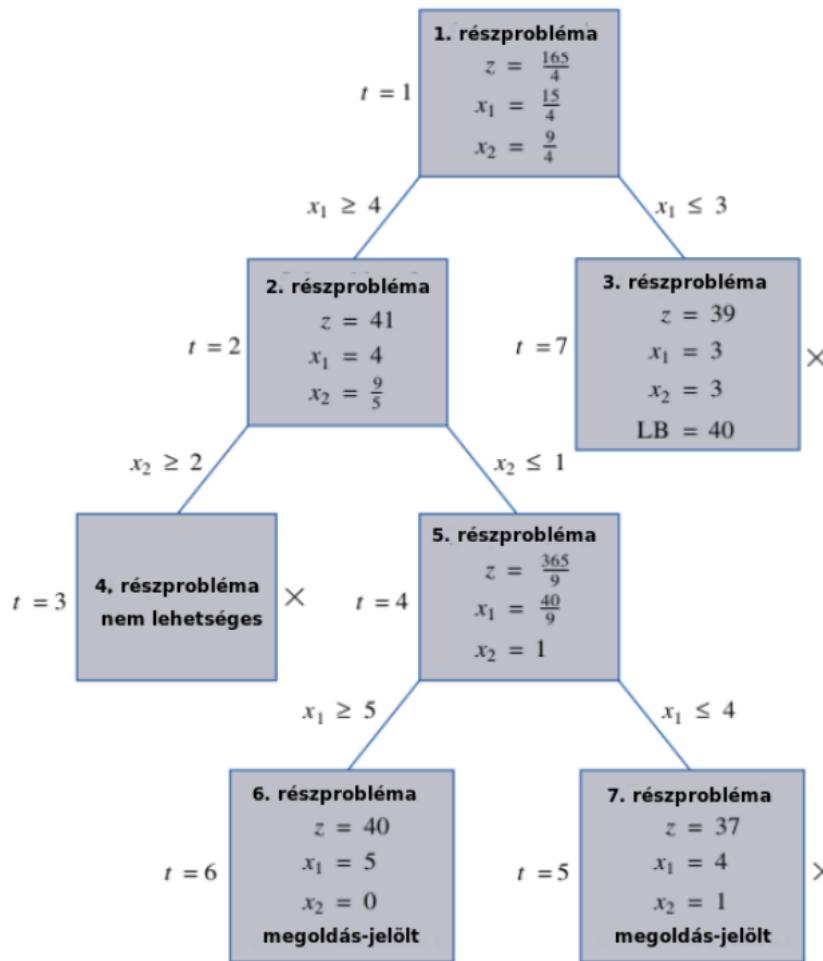
7.2.4. Korátozás és szétválasztás (branch and bound)

Akkor alkalmazzuk, ha egy IP feladat megoldásakor LP-lazításon dolgozunk, és az optimum nem egész

Ehhez egy x_i nem egész változó szerint két részfeladatra bontjuk a feladatot. Ha x_i értéke x_i^* akkor $x_i \leq \lfloor x_i^* \rfloor$, illetve $x_i \geq \lceil x_i^* \rceil$ feltételeket vesszük hozzá egy-egy új feladatunkhoz.

Ezeket a részproblémákat egy fába rendezzük.

- Gyökér az LP-lazítás
- Leszármazottai a részproblémák
- A hozzávetett feltételt az élen adjuk meg
- A csúcsokban az LP-k optimális megoldásait jegyezzük



A 3. részprobléma fáját ki sem kell fejteni, mert már találtunk jobb megoldást a 39-nél, a 6. részproblémában a 40-öt (az is az optimum).

Egy csúcs **felderített** (lezárt), ha:

- Nincs lehetséges megoldása
- Megoldása egészértékű
- Felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

Egy részfeladatot **kizárunk**, ha:

- Nincs lehetséges megoldása
- Felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

7.2.5. Hátzsák feladat

Egy olyan IP-t, amiben **csak egy feltétel** van, hátzsák feladatnak nevezünk.

Példa

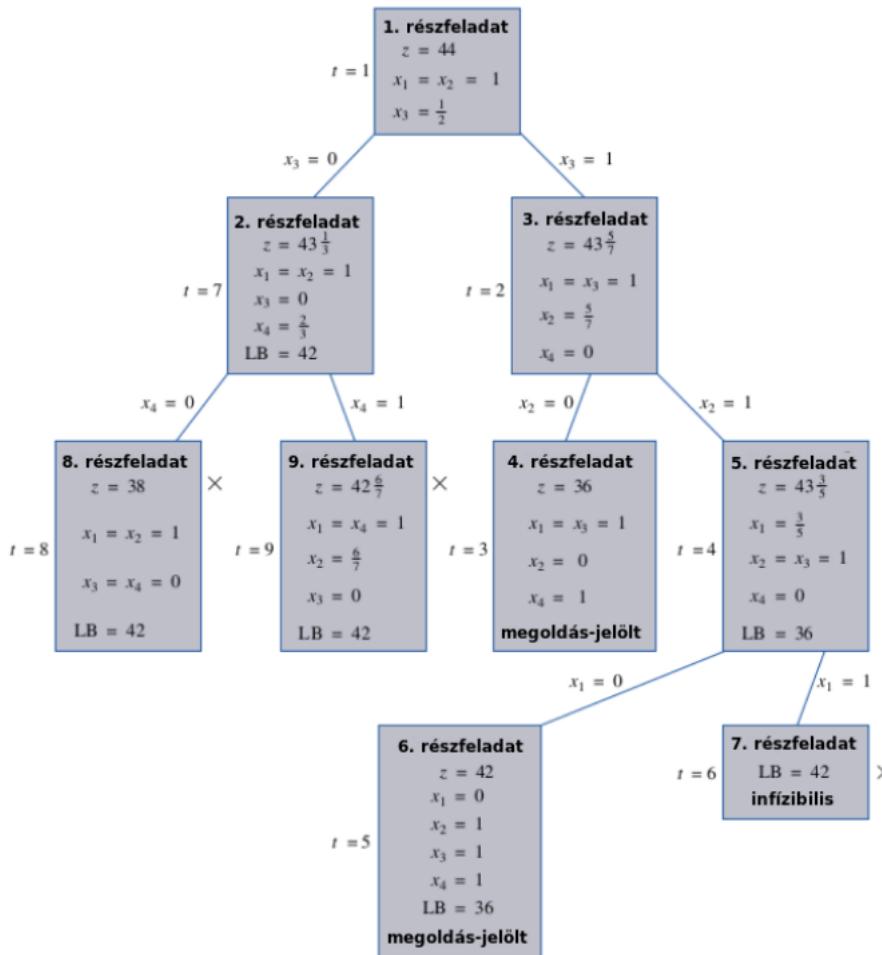
Tárgy	Súly	Haszon	Relatív hasznosság	Sorrend
Tablet	5	16	3.2	1.
Laptop	7	22	3.1	2.
Okostelő	4	12	3	3.
Elemlámpa	3	8	2.7	4.

Matematikai modell:

Legyen $x_i = 1$, ha az i . tárgyat viszi, $x_i = 0$, ha marad. Ekkor a feladat:

$$\max z = 16x_1 + 22x_2 + 12x_3 + 8x_4 \text{ f. h. } 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \quad x_i \in \{0, 1\}$$

LP-lazítás könnyen kiszámolható: Relatív hasznosság szerint sorrendbe rakjuk a tárgyakat, ami belefér azt egészében bele rakjuk, ami nem, annak csak tört részét.



Például itt a lazításban az x_1 és x_2 tárgyak (tablet, laptop) teljesen befértek, míg a telefonnak csak a fele, így amentén ágazunk el, hogy mi lenne, ha $x_3 = 0$ és ha $x_3 = 1$

Legrosszabb esetben 2^n részfeladatot kell megoldani, vagyis a hártsák probléma NP-nehéz.

Egészérkű feladatoknál még rosszabb, 2^{M^n} , ahol M a lehetséges egészek száma egy változóra.

8. Operációs rendszerek

8.1. 1. Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása.
Ütemezési stratégiák és algoritmusok kötegelt, interaktív és valós idejű rendszerekben, ütemezési algoritmusok céljai. Kontextus-csere.

8.1.1. Alapfogalmak

- Processzus:** Egy végrehajtás alatt álló program. minden processzushoz tarozik egy saját címtartomány. Beleértve az utasításszámláló, a regiszterek és a változók aktuális értékét is.
- Szálak (thread):** Processzusok egymással összefüggő erőforrások egy csoportosítása. A processzus címtartománya tartalmazza a program kódját, adatait és más erőforrásait.

8.1.2. Processzusok létrehozása

Négy fő esemény, amely okozhatja egy processzus létrehozását:

- **A rendszer inicializálása**

A felhasználóval tartják a kapcsolatot: előtérben futnak. Nincsenek bizonyos felhasználóhoz rendelve: háttérben futnak, démonok (daemon).

- **A processzus által meghívott processzust létrehozó rendszerhívás végrehajtása**

Kooperatív folyamatok, egymással együttműködő de amúg független processzusok.

- **A felhasználó egy processzus létrehozását kéri**

Interaktív rendszerekben.

- **Kötegelt feladat kezdeményezése**

Amennyiben rendelkezésra áll erőforrás.

8.1.3. Processzusok befejezése

Processzusok befejeződnek, rendszerint a következő körülmények között:

- **Szabályos kilépés (önkéntes)**

A fordítóprogram végzett a feladatával, majd végrehajt egy rendszerhívást, amellyel közli az operációs rendszer felé, hogy elkészült (MINIX 3-ban az `exit` hívás), vagy például képernyőorientált programok esetén (pl.: szövegszerkeztő) rendelkezik olyan billentyű kombinációval amellyel a felhasználó közölheti a processzussal, hogy mentse a munkafájlt és fejezze be a futását.

- **Kilépés hiba miatt (önkéntes)**

Esetlegesen egy hibás programsor miatt. Példa egy illegális utasítás végrehajtása, nem létező memória címre hivatkozás, nullával való osztás.

- **Kilépés végzetes hiba miatt (önkéntelen)**

Végzetes hiba lehet, ha a felhasználó a `cc foo.cc` parancsot kiadva szeretné fordítani a `foo.cc` programot, de nem létezik ilyen nevű fájl. A fordítóprogram egyszerűen kilép.

- **Egy másik processzus megsemmisíti (önkéntelen)**

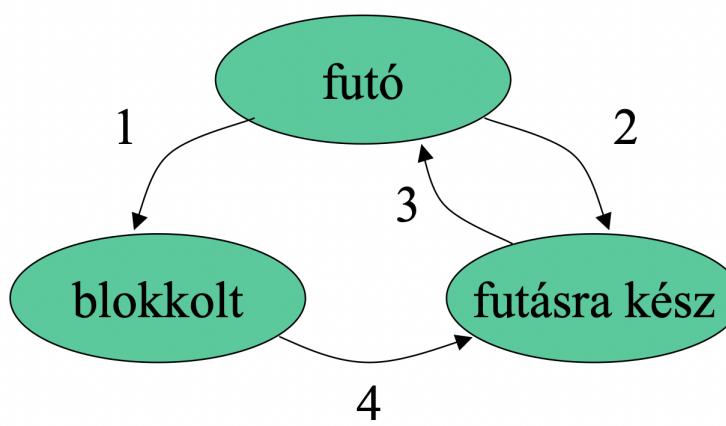
Olyan rendszerhívás végrehajtása, amely közli az operációs rendszerrel, hogy semmisítsen meg egy másik processzust (MINIX 3-ban `kill` hívás). A megsemmisítőnek természetesen rendelkeznie kell a megfelelő jogosultságokkal.

8.1.4. Processzusok állapotai

- **Futó:** Végrehajtás alatt áll, a CPU-t használja.

- **Blokkolt:** Logikailag nem lehet folytatni. Bizonyos külső esemény bekövetkezéséig nem képes futni.

- **Futásra kész:** Elivileg készen áll, futásra képes. Ideiglenesen leállították, hogy egy másik processzus futhasson.



1. A processzus eseményre várva blokkolt
2. Az ütemező másik processzust szemelt ki
3. Az ütemező ezt a processzust szemelte ki
4. Az esemény bekövetkezett

Minden processzus aktuális állapotáról információt kell tárolni, amelyet az operációs rendszer táblázatban tárol el.

- Utasításszámláló
- Veremmutató
- Lefoglalt memória
- Megnyitott fájlok állapota
- Egyéb

8.1.5. Kontextus-csere

Több egyidejűleg létező processzus és egy CPU esetén a CPU váltakozva hajtja végre a processzusokat. A CPU átvált P1 processzusról a P2 processzusra, P1 állapotát a CPU regisztereiből el kell menteni az erre fenntartott memóriaterületre, P2 korábban elmentett állapotát helyre kell állítani a CPU regisztereiben.

8.1.6. Ütemezés

Amikor több processzus képes futni, viszont csak egy processzor áll rendelkezésre, akkor az operációs rendszerek el kell döntenie, hogy mely fusson először. Az operációs rendszer azon részt, amelyik ezt a döntést meghozza, ütemezőnek (scheduler) nevezik, az erre a célra használt algoritmus pedig az ütemezési algoritmus.

8.1.7. Ütemezés kötegelt rendszerekben

Nincsenek felhasználók, emiatt nem megszakítható ütemezési algoritmusok, vagy minden processzus számára hosszú időintervallumokat engedélyező, megszakítható algoritmusok használata gyakran elfogadható. Így csökken a processzusváltások száma és nő a teljesítmény.

- **Sorrendi ütemezés**

Nem megszakítható, olyan sorrendben osztja a CPU-t, ahogy a processzusok azt kérik. Addig fut amíg nem blokkolódik. A sor elején lépő processzus kapja a CPU-t, amikor egy blokkolt útra futáskész a sor végére kerül. könnyen megérthető, páratlan.

- **Legrövidebb feladatot először**

Nem megszakítható, feltételezi, hogy ismerjük a futásidőket. Akkor kell használni, ha több egyformán fontos felelősség van. Csak akkor optimális ha minden feladat egyszerre rendelkezésre áll.

- **Legrövidebb maradék idejű következzen**

Megszakítható, mindig azt a processzust választja az ütemező, amelynek legkevesebb a befejeződésig még a hátralévő ideje, új processzus esetén ha kevesebb idő igényel az új processzus, akkor lecseréljük az új processzusra. Az új, rövid feladatok jó kiszolgálásban részesülnek.

- **Háromszintű ütemezés:**

- **Bebocsátó ütemező:**

Megfelelő keveréket állít elő a CPU és I/O igényes processzusokból, a rövid feladatokat előbb beengedi, de a hosszabbakra

- **Memóriaütemező:**

Sok processzus esetén azok nem férnek el a memóriában, ki kell őket tenni merevlemezre. Körültekintőnek kell lennie. Döntési szempontok:

- *Mennyi idő telt el a processzus lemezre vitele óta?*
- *Mennyi CPU időt használt fel processzus nemrégiben?*
- *Melyen nagy a processzus?*
- *Mennyire fontos?*

- **CPU-ütemező:**

valójában kiválasztja, hogy a futásrakész processzusok közül melyik fusson következőnek.

8.1.8. Ütemezés interaktív rendszerekben

Az időnkénti megszakítás nélkülözhetetlen, nehogy valamely processzus kisajátítsa a CPU-t, ezzel megakadályozva a többit a futásban. Tehát megszakítások ütemezésre van szükség.

- **Round robin ütemezés**

Processzusoknak időintervallum van osztva amely alatt engedélyezett a futásuk. Az időintervallum végén futó processzusok átadják a CPU-t és az idő előtt befejezett vagy blokkolt processzusok is. Listát vezet a futtatható processzusokról és az időszelet felhasználása után a lista végére kerül a processzus.

- **Prioritás ütemezés**

Minden processzushoz prioritást rendelünk, a legmagasabb prioritású, futáskész kapja meg a CPU-t. Annak megeközése érdekében, hogy a magas prioritásúak végig ideig fussanak, minden óraütemben csökkenti a futó processzus prioritását. Amikor a második legfontosabb lesz akkor kontextus csere. Maximális időszeletet rendelünk a processzusokhoz, és amikor lejár az idő, akkor a második processzus a prioritási sorban kapja a CPU-t.

- **Statikus:** Adott rendszerben adott felhasználókra automatikusan adott prioritású processzusok jönnek létre (Unix rendszerben a `nice` utasítással csökkenthetjük a prioritást)
- **Dinamikus:** Erősen I/O műveleteket végző processzusok esetén, azonnal meg kellene kapniuk a CPU-t, lehetővé tenni számára a következő I/O művelet megkezdését, majd másik proszesszus fog futni.

Érdemes a processzusokat prioritási osztályokba sorolni, és osztályokon belül a Round robin ütemezést alkalmazni.

- **Többszörös sorok**

Lassú a kontextus csere végrehajtása. Prioritási osztályok felállítása, úgy hogy a legmagasabb osztályban egy időszakosan futnak, a következőben kettőig, a következő osztályban négy időszakosan és így tovább. Ha elhasználja az időszakot akkor egy osztályal lejebb kerül, így egyre kevesebb gyakorisággal fog futni.

- **Legrövidebb processzus következzen**

Interaktív processzusok általában a következő séma követik:

1. Várakozás utasításra
2. Utasítás végrehajtása
3. GOTO 1.

Kötegelt rendszerben ez minimális válaszidőt ad, viszont párhuzamos processzusoknál nehéz meghatározni, hogy melyik a legrövidebb. Becslés végrehajtása múltbeli viselkedés alapján. Becslés aktualizálása súlyozott átlag számolásával:

$$aT_0 + (1 - a)T_1$$

Ahol T a becsült idő és az a megválasztásával megválaszthatjuk, hogy a processzus gyorsan elfelejtse-e a régi futásokat, vagy sokáig emlékezzen rájuk. Az $a = 1/2$ választással a következő egymás utáni becsléseket kapjuk:

$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/8 + T_3/2,$$

Öregedéssel számolva: vesszük a mért érték és az előző becslés súlyozott átlagát.

- **Garantált ütemezés**

Igéret tétele a felhasználónak a teljesítménnyel kapcsolatban ezt be is tartjuk. (Példa: n felhasználó esetén a CPU $1/n$ -ed részét kapod). A betartáshoz nyomon kell követni hogy a CPU mennyi időt kapott a létrehozása óta, ezután kiszámítja mindenekhez a neki járó mennyiséget.

- **Sorsjáték ütemezés**

Alapötlet, hogy minden processzusnak sorsjegyet adunk a különböző erőforrásokhoz, mint például CPU idő. A fontosabb processzusok többsége sorsjegyet kapnak, hogy növeljék a nyerési esélyeiket. Kooperatív processzusok átadhatják egymásnak a sorsjegyeiket. Nagyon jó a válaszidő.

- **Arányos ütemezés**

Ebben a modellben minden felhasználó kap valamekkora hányadot a CPU-ból. Két felhasználó esetén ez 50%-50% attól függetlenül, hogy hány processzust futtatnak.

8.1.9. Ütemezés valósidejű rendszerekben

Nem minden van szükség megszakításos ütemezésre, mivel a processzusok eleve nem futnak sokáig. Csak a szóban forgó alkalmazás érdekeit szem előtt tartó programok futnak.

Jellemzően egy vagy több külső fizikai eszköz ingert küld a számítógép felé, amire annak megfelelően reagálnia kell egy adott időn belül.

- **Szigorú valósidejű rendszerek:** Abszolút határidők vannak, kötelező betartani.
- **Toleráns valósidejű rendszerek:** Egy-egy határidő elmulasztása nem kívánatos, de azért tolerálható.

A valós idejű viselkedés eléréséhez a programot több processzusra osztjuk, ezek viselkedése ismert/megjósolható, és rövid életűek. Az ütemező feladata, hogy a processzusokat úgy ütemezze, hogy a határidők be legyenek tartva.

Eseményeket két csoportba sorolhatjuk:

- **Periodikusak:** Rendszeres intervallumként fordulnak elő.
- **Aperiodikusak:** Megjósolhatatlan az előfordulásuk.

A valós idejű ütemezési algoritmusok dinamikusak vagy statikusak lehetnek.

8.2. 2. Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizáras. Konkurens és kooperatív processzusok. Kritikus szekciók és megvalósítási módszereik: kölcsönös kizáras tevékeny várakozással (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás). Altatás és ébresztés: termelő-fogyasztó probléma, szemaforok, mutex-ek, monitorok, Üzenet, adás, vétel. Írók és olvasók problémája. Sorompók.

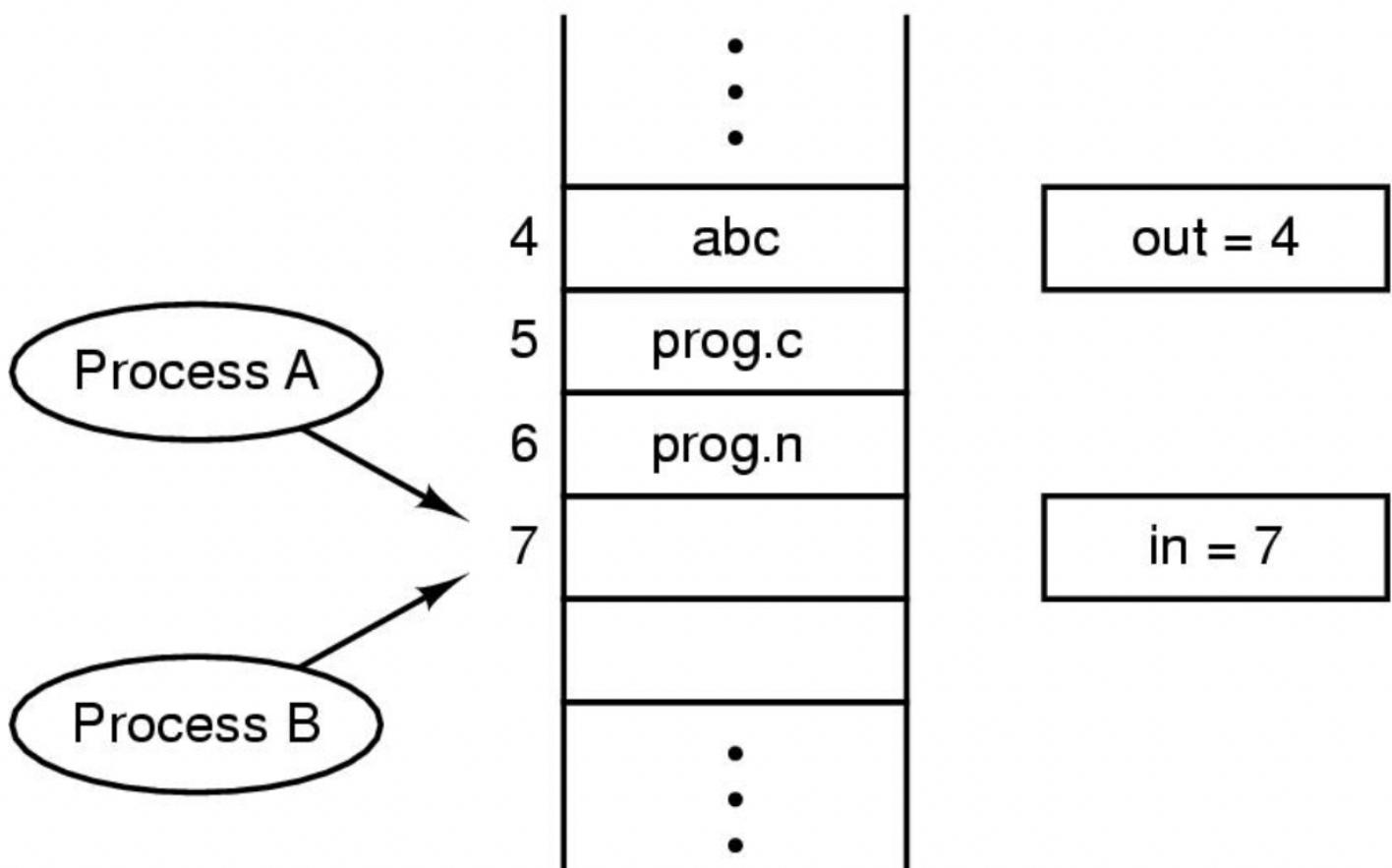
8.2.1. Processzusok kommunikációja

A processzusoknak szükségük van a kommunikációra, és előnyös ha ez nem magiszakításokkal történik. (InterProcess Communication [IPC])

8.2.2. Versenyhelyzetek

Kooperatív processzusok közös tárolóterületen dolgoznak, ahol mindenki írhat és olvashat is. Általános probléma a háttérnyomtatás, ahol egy kliens betesz a nyomtatandó fájl nevét egy háttérkatalógusba majd a nyomtató démon folyamatosan ellenőrzi, hogy kell-e nyomtatni. Ha kell akkor kinyomtatja majd törli a nevét a katalógusból.

Spooler directory



Két megosztott változó van, az `in` és az `out`. Az `in` a katalógus következő szabad helyére mutat és az `out` a következő nyomtatandó állományra. Legrosszabb esetben A olvassa az `in` értékét és eltárolja egy lokális változóban, majd egy kontextus csere történik. B eltárolja az állományt és az `in -t` frissíti az új értekkel majd A folytatja a futását és felülírja a korábbi rekesz tartalmát (kitörli a B által beírt állomány nevét) és frissíti az `in` értékét.

Ez egy versenyhelyzet, mivel megosztott adatok esetén a végeredmény attól függ, hogy ki mikor fut.

8.2.3. Kritikus szekciók

Ahol a program versenyhelyzetbe kerül. Cél a versenyhelyzet elkerülése, meg kell tiltani, hogy egy időben egynél több processzus hozzáférjen a megosztott adatokhoz. Ha két processzus soha nincs egyszerre a kritikus szekciójában a versenyhelyzet elkerülhető.

Szükséges feltételek a kooperatív processzusok megfelelő együttműködéséhez:

1. Ne legyen két processzus egyszerre a saját kritikus szekciójában
2. Semmilyen előfeltétel ne legyen a sebességekről vagy a CPU-k számáról
3. Egyetlen, a kritikus szekcióján kívül futó processzus sem blokkolhat más processzusokat
4. Egyetlen processzusnak se kelljen örökké várni arra, hogy belépjen a kritikus szekciójába

8.2.4. Kölcsönös kizárási mechanizmus

Egy módszer amely biztosítja, hogy ha egy processzus használ valamely megosztott változót vagy fájlt, akkor a többi processzus tartózkodjon ettől a tevékenységtől.

8.2.5. Kölcsönös kizárási meccanizmus tevékeny várakozással

- **Megszakítások tiltása** minden processzus leltija az összes megszakítást a kritikus szekcióba lépés után, majd újra engedélyezi mielőtt elhagyja azt. Ha nem fordulhat elő óramegszakítás akkor a CPU nem fog másik processzura váltani.

Gyakran hasznos technika az operációs rendszeren belül de nem megfelelő a felhasználói processzusok számára mint általános kölcsönös kizárási mechanizmus.(Mivel mi van akkor, ha nem engedélyezi azokat újból, vagy több processzoros rendszernél csak az adott CPU-ravonatkozik a megszakítás tiltása)

- **Változók zárolása:** Egy szoftvermegoldás, megosztott változó, kezdeti értéke 0. Mielőtt egy processzus belépne a saját kritikus szekciójába először megvizsgálja ezt a változót, ha 0 akkor belép és 1-re állítja. Ha már 1 akkor a processzus addig vár, amíg az 0 nem lesz. Így a 0 azt jelenti, hogy egyetlen processzus sincs a saját kritikus szekciójában, az 1 meg hogy valamely processzus a saját kritikus szekciójában van.

Sajnos ez a módszer ugyan azt a végzetes hibát rejt magában mint a háttérnyomtatás. (Mi van ha az egyik processzus elolvassa a zárolásváltozót és épp akkor történik kontextus csere mielőt betudná állítani azt 1 értékre)

- **Szigorú váltogatás:** Folyamatosan tesztel, hogy lássa mikor léphet be a kritikus szekciójába. Azt amikor folyamatosan tesztelünk egy változót egy bizonyos érték megjelenéséig, **tevékeny várakozás**-nak nevezzük. Ez általában nem megfelelő megoldás mivel pazarolja a CPU-t.

Bár ez az algoritmus elkerül minden versenyt, mégsem tekinthető komoly jelöltnek a probléma megoldására mivel megséríti a 3.feltételt miszerint: A processzus blokkolja másik processzus kritikus szekcióból való lépését a kritikus szekción kívül.

- **Peterson megoldása:** Két C eljárás, kritikus szekcióból lépés előtt **enter_region** hívás, kritikus szekció elhagyása után **leave_region** hívása.

Ez egy udvarias megoldás mivel maga elő engedi a másik processzust.

• **TSL utasítás:**

Hardveres segítséget igényel. Általában többprocesszoros gépeknél alkalmazandó. TSL RX,LOCK utasítás van. (Test and Set Lock) Beolvassa a LOCK memória szót az RX regiszterbe, nem nulla értéket ír a memória címre A memória elérés más CPU-knak tiltva van a művelet befejezéséig

A TSL utasítás alkalmazásához egy **LOCK** megosztottváltozót használunk, ezzel összehangolva a megosztott memória elérést. Amikor a **LOCK** 0 akkor bármelyik processzus beállíthatja azt 1-re a TSL utasítás használatával és ezután írhatja, olvashatja a megosztott memóriát. Ha megtette a processzus visszaállítja a **LOCK** értékét 0-ra.

Párban kell használni az eljárásokat, mivel ha valaki csal, akkor a kölcsönös kizárást meghiúsul.

8.2.6. Altatás és ébresztés

Processzusok közötti kommunikációs primitív amelyek a CPU idő pazarlása helyett blokkolnak, amikor nem megengedett, hogy kritikus szekcióból lépjenek. **sleep** és **wakeup** párok. A **sleep** egy rendszerhívás amely a hívót blokkolja (felfüggeszti) mindaddig amíg egy másik processzus fel nem ébreszti. A **wakeup** hívásnak egy paramétere van, maga a processzus amit felkell ébreszteni.

8.2.7. Termelő-fogyasztó probléma

Termelő mely adatokat helyez el a tárolóban és a fogyasztó amely azokat kiveszi, szükség van egy **count** változóra amit a termelő megvizsgál, hogy maximális-e az értéke, (azaz a tároló tele van-e) ha igen akkor a termelő elalszik, ha nem akkor beletesz egy elemet a tárolóba és növeli a **count** értékét. A fogyasztó szintű ellenőrzi **count** értékét és ha 0 elalszik, ellenkező esetben pedig kiveszi az elemet a tárolóból. Probléma a még nem alvó processzusnak küldött ébresztőjel elveszik. Megoldás: ébresztőt váró bit hozzáadása.

8.2.8. Szemaforok

Egész változókban kell számolni az ébresztéseket a későbbi felhasználás érdekében. A szemafor értéke lehet 0 ha nincs elmentett ébresztés, lehet pozitív ha egy vagy több ébresztés függőben van. Ez a két művelet a **down** és az **up**. A **down** művelet megvizsgálja hogy a szemafor értéke nagyobb-e mint 0. Ha igen csökkenti az értékét (azaz elhasznál egy ébresztést) és azonnal folytatja. Ha az értéke 0 akkor a processzust elalatta mielőtt a **down** befejeződne. Az **up** rendszerhívás a megadott szemafor értékét növeli. Ha egy vagy több processzus aludna már ezen a szemaforon akkor egyet kiválasztva a rendszer megengedi annak hogy befejezze a **down** műveletét.

8.2.9. Mutex-ek

A szemafor egy egyszerűsített változata. A mutexek csak bizonyos erőforrások vagy kódrészek kölcsönös kizáráásának kezelésére alkalmasak. Két állapota lehet: **zárolt** és **nem zárolt**. Például ha egy processzus hozzá szeretne jutni a kritikus szekciójához akkor meghívja a **mutex_lock** eljárást. Ha a mutex nem zárolt akkor az eljárás sikeres és beléphet a kritikus szekcióba. Másik esetben pedig a hívó blokkolódik amíg a kritikus szekcióból lévő processzus nem hívja a **mutex_unlock** eljárást.

8.2.10. Monitorok /nem érhető/

Magasabb szintű szinkronizációs primitívek használatának javaslata amelyeket monitoroknak neveztek el.

- Eljárások, változók és adatszerkezetek együttese
- Speciális modulba/csomagba vannak összegyűjtve
- A processzusok bármikor hívhatják a monitorban lévő eljárásokat, nem érhetik el a belső adatszerkezeteit
- Objektumorientált nyelvekben is használt szabály
- minden időpillanatban csak egy processzus lehet aktív egy monitorban
- Könnyű módszer a kölcsönös kizárást eléréséhez
- Nem elegendő, szükség van olyan módszerre, amellyel egy processzust blokkolhatunk, ha nem tud tovább haladni
- Megoldás állapot változók bevezetése (nem számlálók)

Szükség van egy szabályra, hogy mi történjen a `signal` után?

Az újonnan felébresztett processzust hagyjuk futni, de a másikat felfüggesztjük

8.2.11. Üzenet

Két primitívet használ, hasonlítanak a szemaforokra, de nem monitorok `send(cél)` és `receive(forrás)` (rendszerhívások)

Tervezési kérdések az üzenetküldési rendszereknél:

Hálózatba kötött gépeken futó processzusok kommunikációja Az üzenet elveszhet a hálózaton

A küldő és fogadó ugyanazon a gépen található Lassabb, mint a többi eddigi eszköz

Levelesláda adatszerkezet

Addott számú üzenet tárolására alkalmas Ha tele van a levelesláda küldő felfüggesztődik

Ideiglenes tárolás elhagyása

Ha `receive` előtt `send`-et hajtunk végre Küldő blokkolódik, amíg a `receive` végrehajtódik

Ha a `send` előtt `receive` hajtódik végre Fogadó blokkolódik, amíg a `send` blokkolódik

Randevúnak nevezzük ezt a stratégiát, könnyebb megvalósítani, de kevésbé rugalmas

8.2.12. Írók és olvasók problémája

Adatbázis elérést modellező probléma. Például egy légitársaság helyfoglaló rendszere. A modellben léteznek olvasó processzusok amelyek olvassák az adatbázist akár egyidejűleg is és író processzusok amelyek ha írják az adatbázist, akkor más processzusoknak nem szabad elérniük azt. A kérdés az, hogy hogyan programozzuk az írókat és olvasókat?

Egyik megoldás: Az első olvasó aki hozzáfér az adatbázishoz végrehajt egy `down`-t a `db` szemaforon és a következő olvasók csak az `rc` számlálót növelik. Ha egy olvasó kilép, akkor csökkenti az `rc` számlálót, majd az utolsó kilépő végrehajt egy `up`-ot a `db` szemaforon lehetővé téve az írónak, (ha van ilyen) hogy belépjen. Ezzel a probléma hogy mivel nem baj, ha több olvasó lép be, ezért ha sorozatosan érkeznek olvasók, azok bebocsátást nyernek, viszont egy író felfüggesztődik. Abban az esetben, ha mondjuk 2mp-ként érkezik egy olvasó és ezek munkája 5mp-ig tart, akkor az író soha sem lesz beengedve.

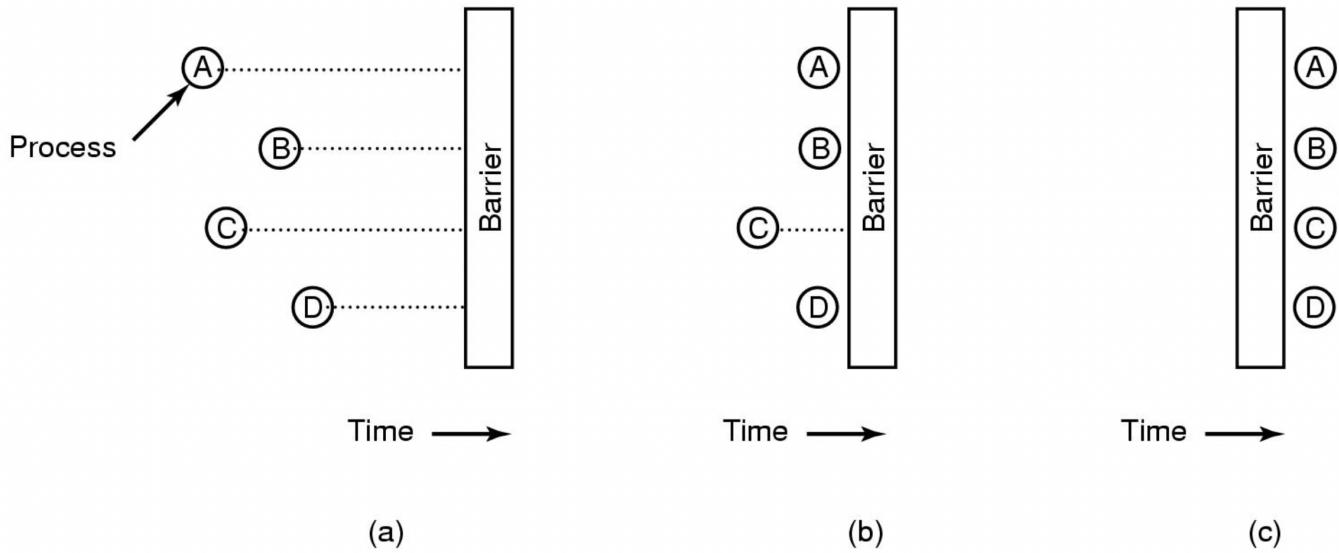
Masik megoldás: Ha érkezik egy olvasó, viszont egy író már vár, akkor az olvasó felfüggesztődik. Így az írónak csak azokat az olvasókat kell megvárnia hogy végezzenek amelyek előtte érkeztek. Ennek a megoldásnak a hátránya, hogy kevesebb párhuzamosságot enged meg, ezáltal a hatékonyság csökken.

Courtois és társai rámutattak egy megoldásra amely az írónak ad elsőbbséget. (*Lásd: Courtois et al., 1971*)

8.2.13. Sorompók

Könyvtári eljárás, fázisokra osztjuk az alkalmazást **Szabály:** Egyetlen processzus sem lehet tovább a következő fázisra, amíg az összes processzus nem áll készen

Sorompó elhelyezése mindegyik fázis végére és amikor egy processzus a sorompóhoz ér, akkor addig blokkolódik ameddig az összes processzus el nem éri a sorompót, majd a sorompó az utolsó processzus beérkezése után elengedi a azokat



Sorompó használata:

- Processzusok közelítenek egy sorompóhoz.
- Egy processzus kivételével minden egyik processzus blokkolódik a sorompónál.
- Amikor az utolsó processzus a sorompóhoz ér, minden egyiket tovább engedi

Adatbázisok

8.3. 1. Adatbázis-tervezés: A relációs adatmodell fogalma. Az egyed-kapcsolat diagram és leképezése relációs modellre, kulcsok fajtái. Funkcionális függőség, a normalizálás célja, normálformák.

Emlékeztető

Alapfogalmak

Adatbázis (DB = database): adott formátum és rendszer szerint tárolt adatok együttese

Adatbázisséma: az adatbázis struktúrájának leírása

Adatbázis-kezelő rendszer (DBMS = Database Management System): az adatbázist kezelő szoftver

- Főbb feladatai:
 - adatstruktúrák definiálása (adatbázisséma)
 - adatok aktualizálása (adatfelvitel, törlés, módosítás) és lekérdezése
 - nagy mennyiségű adat hosszú idejű, biztonságos tárolása
 - több felhasználó egyidejű kiszolgálása, jogosultságok szabályozása
 - több feladat egyidejű végrehajtása, tranzakciók kezelése

Rekord (= feljegyzés): az adatbázis alapvető adategysége

Egyed-kapcsolat modell fogalmai

Egyed (entitás): a valós világ egy objektuma, melyről az adatbázisban információt szeretnénk tárolni

Tulajdonság (attribútum): az egyed egy jellemzője

Összetett attribútum: maga is attribútumokkal rendelkezik (általában egy struktúra, aminek adattagai külön-külön elemi típusú értékekre képződnek le)

- Jelölés: a struktúrát alkotó adattagokat újabb ellipszisekkel kötjük az összetett attribútumhoz

Többértékű attribútum: halmaz vagy lista adattípusra képződik le (előbbinél nem számít a sorrend, utóbbinál viszont igen)

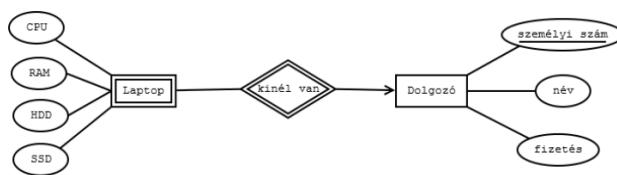
- Jelölés: kettős ellipszis

Gyenge entitás: az attribútumai nem határozzák meg egyértelműen

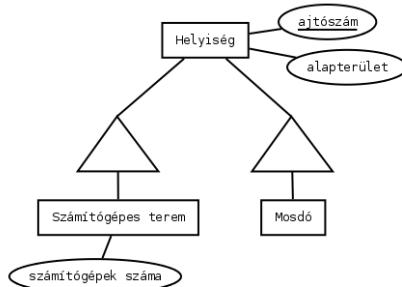
- Jele: kettős téglalap

Meghatározó kapcsolat: gyenge entitást határoz meg

- Jele: kettős rombusz



Specializáló kapcsolat: olyan kapcsolat, amely hierarchiát jelöl az egyedek között. Például egy helyiség lehet számítógépes terem vagy mosdó is.



Kulcs: olyan (minimális) attribútumhalmaz, amely már egyértelműen meghatározza az egyedet

8.3.1. A relációs adatmodell fogalma

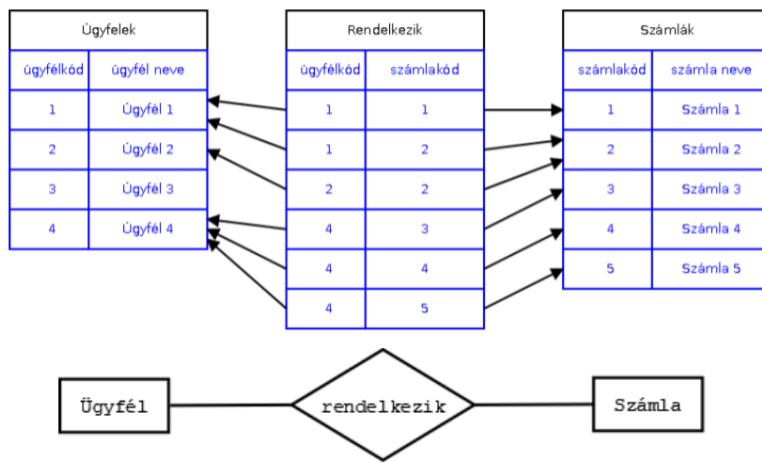
- az adatokat és a köztük lévő kapcsolatokat is kétdimenziós táblákban tárolja; az azonos sorban álló egyedek alkotnak egy relációt
- az erre a modellre épülő adatbáziskezelőket RDBMS-nek (Relational DBMS) nevezzük
- lekérdező nyelvük az SQL (Structured Query Language)
- napjainkban is széles körben használt modell

Adattábla: sorokból (rekordokból) és oszlopokból áll. Egy sor annyi mezőből áll, ahány oszlopa van a táblának.

Attribútum: egy tulajdonság, névvel és értéktartománnal (a Z attribútum értéktartománya $\text{dom}(Z)$)

Értéktartomány megadása: típus, hossz, (korlátozó feltételek)

- A relációs modellben az értéktartomány csak atomi értékekkel állhat!



Relációséma: egy névvel ellátott attribútumhalmaz

- Jelölések:
 - $R(A_1, \dots, A_n)$
 - $R(A)$, ahol $A = A_1, \dots, A_n$
 - $R. A_1$ az R séma A_1 attribútuma
- Példa:
 - Ügyfél (ügyfélkód, ügyfél neve)
 - $\text{dom}(\text{ügyfélkód})$: 10 jegyű egész számok halmaza
 - $\text{dom}(\text{ügyfél neve})$: legfeljebb 200 karakter hosszú sztringek halmaza

Reláció (adattábla) az $R(A_1, \dots, A_n)$ relációs séma felett: $T \subseteq \text{dom}(A_1) \times \dots \times \text{dom}(A_n)$.

T elemei (a_1, \dots, a_n) alakúak, ahol $a_i \in \text{dom}(A_i)$ ($i = 1, \dots, n$).

Az Ügyfél(ügyfélkód, ügyfél neve) relációs séma feletti relációs séma feletti adattábla: $T \subseteq \text{dom}(\text{ügyfélkód}) \times \text{dom}(\text{ügyfélneve})$:

Ügyfelek	
ügyfélkód	ügyfél neve
1	Ügyfél 1
2	Ügyfél 2
3	Ügyfél 3
4	Ügyfél 4

Miért hívják relációnak az adattáblát? Magyarázat: a matematikai relációfogalom

- \mathbb{Z} : természetes számok halmaza
- $\mathbb{Z} \times \mathbb{Z}$: az összes (a, b) párok halmaza

Relációjel: pl. $<$

- A "kisebb" reláció definíciója:
 - $K \subseteq \mathbb{Z} \times \mathbb{Z}$, ahol K azon (a, b) párok halmaza, amelyekre $a < b$
 - Példa: $(2, 3) \in K$, de $(5, 2) \notin K$
- Általánosítás: $K \subseteq A \times B \times C$, ahol K azon (a, b, c) hármasok halmaza, amelyekre valamilyen feltétel teljesül

Megjegyzések

- Az adattábla sorok halmaza, ezért a relációs modellben a tábla minden sora különböző, és a soroknak nincs kitüntetett sorrendje.
- Elvileg a tábla oszlopainak sincs kitüntetett sorrendje
- Az RDBMS-ek (Relational Database Management System) általában megengednek azonos sorokat is, és a soroknak ill. oszlopoknak szükségképpen van egy tárolási sorrendje.

Elnevezések

- Relációséma: a tábla felépítési sémája.
- Reláció vagy adattábla vagy tábla: az adatokat tartalmazza.
- Sor, oszlop.
- Rekord: a tábla egy sora.
- Mező: a séma egy attribútuma, vagy egy bejegyzés a táblában.
- NULL: definiálatlan mezőérték.
- Relációs adatbázis: táblák együttese.

Relációsémák és táblák

- Ügyfél (ügyfélkód, ügyfélnev)
- Számla (számlaszám, számla neve)
- Rendelkezik (ügyfélkód, számlaszám)

A sémák között a közös attribútumok biztosítják a kapcsolatot.

Kulcsok

- **Szuperkulcs (superkey)**: olyan attribútumhalmaz, amely egyértelműen azonosítja a tábla sorait.
- Pontosabban: Egy $R(A)$ relációsémában $K (\subseteq A)$ szuperkulcs, ha bármely R feletti T tábla bármely két sora K -n különbözik.
- **Kulcs (key)**: $K (\subseteq A)$ kulcs, ha minimális szuperkulcs, vagyis egyetlen valódi részhalmaza sem szuperkulcs.
 - Például az Ügyfél (ügyfélkód, ügyfél neve) sémában
 - {ügyfélkód} szuperkulcs és kulcs is
 - {ügyfélkód, ügyfél neve} szuperkulcs de nem kulcs
 - {ügyfél neve} nem szuperkulcs (és nem kulcs)
- **Egyszerű kulcs**: ha egyetlen attribútumból áll.
- **Összetett kulcs**: ha több attribútumból áll.

- Példák:
 - Ügyfél (ügyfélkód, ügyfél neve) -> Egyszerű kulcs: {ügyfélkód}
 - Rendelkezik (ügyfélkód, számlaszám) -> Összetett kulcs: {ügyfélkód, számlaszám}
- Megjegyzések
 - A teljes A attribútumhalmaz minden szuperkulcs.
 - A kulcs valójában egy feltétel előírása a relációsémára.
 - A kulcs a séma tulajdonsága, nem a tábláé.
 - Egy sémában több kulcs lehet.

• **Elsődleges kulcs (primary key):** Ha csak egy kulcs van, az lesz az elsődleges kulcs. Ha több kulcs van, egyet önkényesen kiválasztunk. Jele: aláhúzás.

- Például:

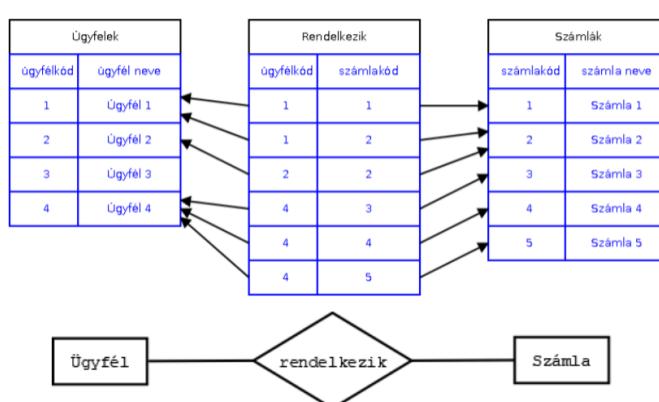
- Felhasználó (felhasználónév, név, e-mail, jelszó, belépés) Kulcsok: {felhasználónév}, {e-mail} Elsődleges kulcs: {felhasználónév}

• Fontos különbség:

- Relációs modell: a tábla minden sora különböző, ezért minden sornak van kulcs.
- Konkrét RDBMS: ha azonos sorokat is megengedünk, akkor nincs kulcs!
 - Példa (itt megengedhető hogy ne legyen kulcs):
 - Vásárlás (dátum, terméknév, mennyiség) 2011.09.04. |banán| 4.0 2011.09.05. |alma| 3.0 2011.09.05. |szilva| 1.5 2011.09.05. |alma| 3.0

• **Külső kulcs (idegen kulcs, foreign key):**

- Egy relációséma attribútumainak valamely részhalmaza külső kulcs, ha egy másik séma elsődleges kulcsára hivatkozik.
- Jelölés: dőlt betű, vagy a hivatkozott kulcsra mutató nyíl
- A külső kulcs valójában egy feltétel előírása a relációsémára.
- Hivatkozási integritás: A külső kulcs értéke a hivatkozott táblában előforduló kulcsérték vagy NULL lehet
- Példa:
 - Ügyfél (ügyfélkód, ügyfélneve) Számla (számlaszám, számla neve) Rendelkezik (ügyfélkód, számlaszám)



Relációs adatbázis séma: relációsémák + kulcs feltételek (elsődleges kulcsok, külső kulcsok)

- Példa: egészségügyi adatbázis
 - Beteg (betegeId, betegnév, lakcím)
 - Orvos (orvosId, orvosnév, osztály, kórház)
 - Kezelés (kezelésId, megnevezés, kategória)
 - Ellátás (betegeId, orvosId, kezelésId, dátum, költség)

Indexelés

- Index: kiegészítő adatstruktúra. Célja:
 - rendezés,
 - keresések gyorsítása (pl. külső kulcs).

- Indexkulcs: valamely $L \subseteq A$ attribútumhalmaz.
 - Megegyezhet a tényleges kulccsal, de más is lehet.
- „Indextábla”: Index (indexkulcs, rekordsorszám)
- Egy táblához több index is létrehozható, de a sok index lassabb műköéshez vezet

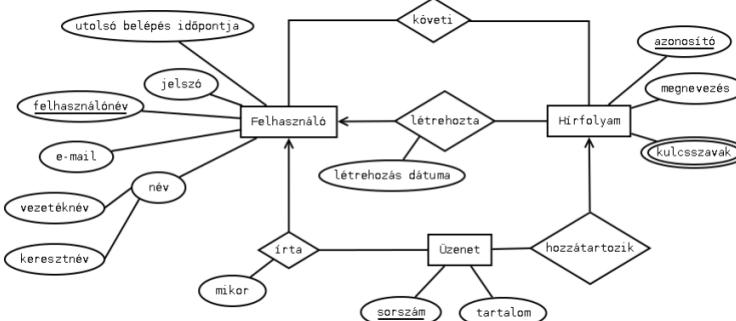
Felhasználó					
fizikai azon.	felhaszn. név	név	email	jelszó	utolsó belépés
1	pbalazs	Balázs Péter	pbalazs@...	e(RpL9IU2	2018-10-06 11:10:00
2	pkardos	Kardos Péter	pkardos@...	87fiHh9O	2018-10-03 9:45:00
3	gnemeth	Németh Gábor	gnemeth@...	2XgfSStw	2018-10-15 17:00:00
4	bodnaar	Bodnár Péter	bodnaar@...	JkFrrS7s	NULL

Név szerinti index		Belépés szerinti index	
név	fizikai azon.	utolsó belépés	fizikai azon.
Balázs Péter	1	2018-10-03 9:45:00	2
Bodnár Péter	4	2018-10-06 11:10:00	1
Kardos Péter	2	2018-10-15 17:00:00	3
Németh Gábor	3	NULL	4

8.3.2. E-K modellből relációs modell

Egyedek leképezése

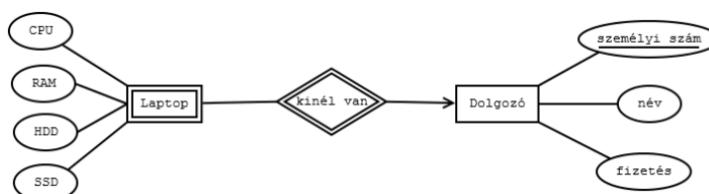
- Szabály: a relációséma neve az egyed neve, attribútumai az egyed attribútumai, elsődleges kulcsa az egyed kulcs-attribútuma(i).
- Megfeleltetés: egyedtípus -> relációséma, egyedpéldány -> tábla egy sora, egyedhalmaz -> teljes tábla
- Attribútumok értéktartománya meghatározandó
- Kulcs-feltétel ellenőrzendő! Van-e több kulcs?



FELHASZNÁLÓ(felhasználónév, jelszó, email, név, utolsó belépés időpontja) ÜZENET(sorszám, tartalom) HÍRFOLYAM(azonosító, megnevezés, kulcsszavak)

Gyenge egyedek leképezése

- Szabály: a gyenge egyed relációsémáját bővíteni kell a meghatározó kapcsolat(ok)ban szereplő egyed(ek) kulcsával.

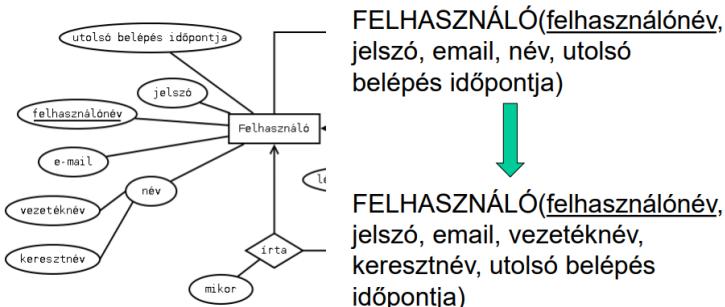


- Tfh. egy dolgozónak nincs két ugyanolyan konfigurációjú laptopja!

◦ LAPTOP(személyi szám, CPU, RAM, HDD, SSD)

Összetett attribútumok leképezése

- Szabály: az összetett attribútumot a komponenseivel helyettesítjük (egy lépésben megtehető).



Többértékű attribútumok leképezése

- megoldás: hosszú string



Hátrány: Lassú keresés

- megoldás: sorok többszörözése

HÍRFOLYAM		
azonosító	megnevezés	kulcsszavak
1	Adatbázis kérdések	adatbázis
1	Adatbázis kérdések	SQL
1	Adatbázis kérdések	oktatás
2	PHP hírek	PHP
2	PHP hírek	programozás
3	Ki a legjobb tanár	vélemény
3	Ki a legjobb tanár	oktatás
4	Milyen gépet vegyek	hardver

Hátrány: egyedi azonosítás elvesztése + redundancia

- megoldás: új tábla felvétele

HÍRFOLYAM_KULCSSZAVAK	
azonosító	kulcsszavak
1	adatbázis
1	SQL
1	oktatás
2	PHP
2	programozás
3	vélemény
3	oktatás
4	hardver

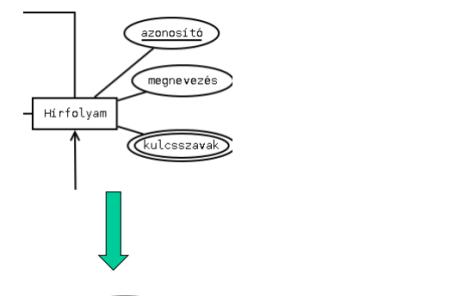
Ha a kulcsszavak sorrendje is számít, akkor az új tábla ezzel bővíthető!

- megoldás: Az ismétlődő kulcsszavak elkerülése (kapcsoló tábla felvétele)

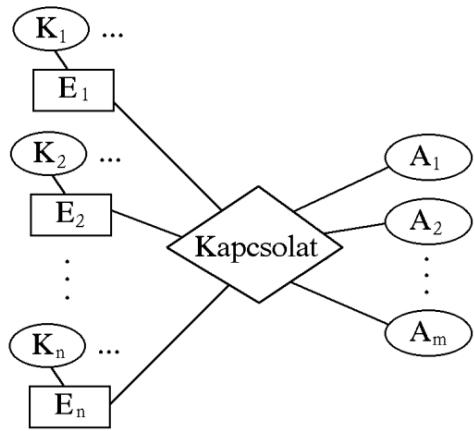
HÍRFOLYAM	
azonosító	megnevezés
1	Adatbázis kérdések
2	PHP hírek
3	Ki a legjobb tanár
4	Milyen gépet vegyek

kulcsszó azonosító	kulesszó
1	adatbázis
2	SQL
3	oktatás
4	PHP
5	programozás
6	vélemény
7	hardver

hírfolyam azonosító	kulcsszó azonosító
1	1
1	2
1	3
2	4
2	5
3	6
3	3
4	7



Kapcsolatok leképezése



1. Új séma felvétele: $Kapcsolat(K_1, \dots, K_n, A_1, \dots, A_m)$

2. Konszolidáció: Ha az új séma kulcsa megegyezik valamelyik E_i kulcsával, akkor azzal összevonható

FELHASZNÁLÓ(felhasználónév, jelszó, email, vezetéknév,
keresztnév, utolsó belépés időpontja)

ÜZENET(sorszám, tartalom)

HÍRFOLYAM(azonosító, megnevezés, kulcsszavak)

+

ÍRTA(felhasználónév, sorszám, mikor)

KÖVETI(felhasználónév, azonosító)

LÉTREHOZTA(felhasználónév, azonosító)

HOZZÁTARTOZIK(sorszám, azonosító)

=

FELHASZNÁLÓ(felhasználónév, jelszó, email, vezetéknév,
keresztnév, utolsó belépés időpontja)

ÜZENET(sorszám, tartalom, felhasználónév, mikor, azonosító)

HÍRFOLYAM(azonosító, megnevezés, kulcsszavak,

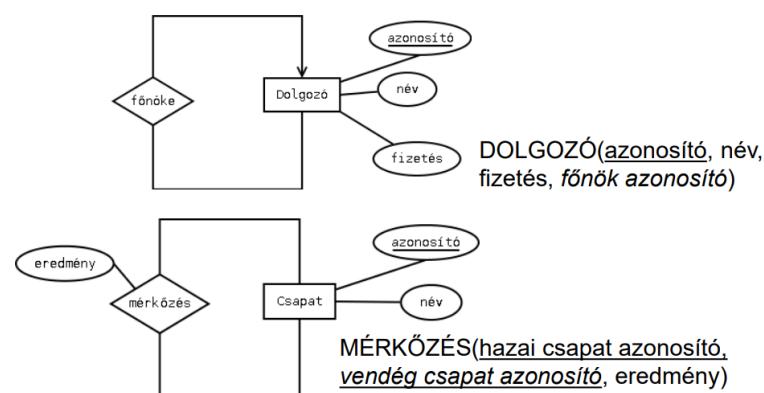
felhasználónév)

KÖVETI(felhasználónév, azonosító)

Kapcsolatok leképezési szabálya, összefoglalás

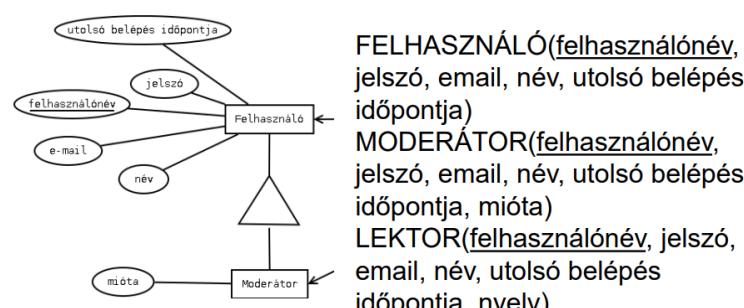
- 1:1 kapcsolat esetén a kapcsolat sémája bármelyik egyed sémájába beolvasztható.
- 1:N kapcsolat esetén a kapcsolat sémája az N oldali egyed sémájába beolvasztható.
- Végezhető egy lépésben.
- N:M vagy többágú kapcsolat esetén a kapcsolat sémája egyik egyed sémájába sem olvasztható be.

Önmagával kapcsolódó entitás



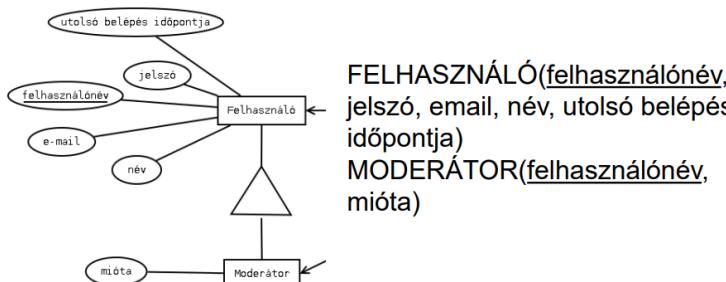
Specializáló kapcsolatok leképezése

1. megoldás: főtípus és minden altípus külön sémában (minden egyedpéldány csak egy táblában szerepel)



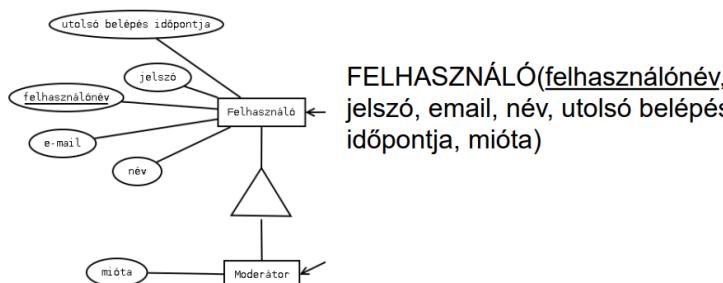
Hátrány: kereséskor több táblát is vizsgálni kell, kombinált altípusokhoz új tábla kell

2. megoldás: minden altípus külön sémában a fő típus kulcsattribútumaival (egy egyedpéldány több táblában szerepelhet)



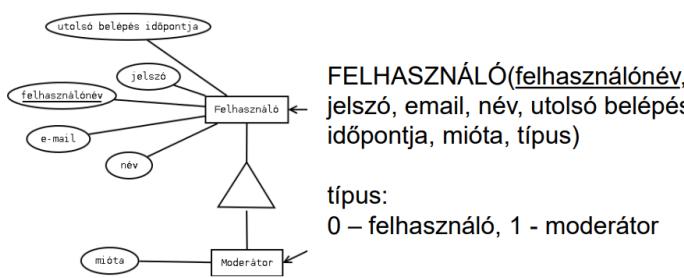
Hátrány: kereséskor több táblát is vizsgálni kell

3. megoldás: egy közös tábla



Hátrány: NULL értékek (tárpazari + tényleges jelentésük elvész)

4. megoldás: egy közös tábla típusjelzéssel



Hátrány: NULL értékek (tárpazarló)

8.3.3. Adatbázis normalizálása

Cél: Redundancia kiszűrése az adatbázisból, aktualizálási anomáliák elkerülése érdekében.

1. Redundancia felderítése a relációsémák vizsgálatával.
2. Redundancia megszüntetése a sémák felbontásával (normalizálás).

8.3.4. Funkcionális függés

- Adott $R(A_1, \dots, A_n)$ relációséma, $P, Q \subseteq A_1, \dots, A_n$
- P -től funkcionálisan függ Q , ha bármely R feletti T tábla esetén valahányszor két sor megegyezik P -n, akkor megegyezik Q -n is
- Jelölés: $P \rightarrow Q$
- Példa:
 - Adott a következő tábla:

Employee number	Employee Name	Salary	City
1	Dana	50000	San Francisco
2	Francis	38000	London
3	Andrew	25000	Tokyo

- Ha tudjuk az Employee number oszlop értékét, akkor megtudhatjuk a hozzá tartozó Employee Name, Salary és City értékeit is. Ez alapján mondhatjuk hogy Employee Name, Salary és City funkcionálisan függenek az Employee number oszloptól. Jelölése:

$$\{Employee number\} \rightarrow \{Employee Name, Salary, City\}$$

Elnévezések:

- $P \rightarrow Q$ triviális, ha $Q \subseteq P$ (vagyis ha Q elemei egyben P elemei is)
- Ellenkező esetben $P \rightarrow Q$ nemtriviális.

- $P \setminus rarr Q$ teljesen nemtriviális, ha $Q \cap P = \emptyset$ (vagyis ha nincsenek közös elemeik)

FÓRUM_KÖVETÉSE

felhasználónév	email	név	hírfolyam azonosító	megnevezés
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1	Adatbázis kérdések
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2	PHP hírek
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4	Milyen gépet vegyek
pkardos	pkardos@inf.u-szeged.hu	Károly Péter	2	PHP hírek
pkardos	pkardos@inf.u-szeged.hu	Károly Péter	3	Ki a legjobb tanár
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1	Adatbázis kérdések
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2	PHP hírek
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3	Ki a legjobb tanár
bodnara	bodnara@inf.u-szeged.hu	Bodnár Péter	4	Milyen gépet vegyek

Teljesen nemtriviális függések (pl.)

{felhasználónév} → {email}, {felhasználónév} → {név}

{felhasználónév} → {email, név}

{email} → {felhasználónév}

{email} → {név}

{hírfolyam azonosító} → {megnevezés}

Nemtriviális függés (pl.)

{felhasználónév, email} → {email, név}

Triviális függés (pl.)

{felhasználónév, email} → {email}

Példa:

- SZÁMLA (cikkszám, megnevezés, egységár, mennyiség, összeg)

◦ összeg = egységár * mennyiség

◦ Függőségek:

- {cikkszám} \rarr {megnevezés, egységár, mennyiség}
- {egységár, mennyiség} \rarr {összeg}

- DOLGOZÓ (adószám, név, beosztás, fizetés)

◦ Itt {beosztás} \rarr {fizetés} függés nem áll fenn!

Megjegyzések:

- A funkcionális függés a séma tulajdonsága (nem a tábláé).

| séma: az adat struktúráját írja le (gyakorlatilag egy tervrajz)

| tábla: az adatok, oszlopok és sorok szerint rendezetten, a séma szerint meghatározva

- „Funkcionális” jelentése: ha $P \setminus rarr Q$, akkor létezik egy $dom(P) \setminus rarr dom(Q)$ függvény. Például: {egységár, mennyiség} \rarr {összeg} kiszámítható, de {felhasználónév} \rarr {email} nem számítható.

- Állítás: $K(\subseteq A)$ akkor és csak akkor szuperkulcs, ha $K \setminus rarr A$

- Relációséma új definíciója: $R = (A, F)$, ahol $A = A_1, \dots, A_n$ attribútumhalmaz, és $F = f_1, \dots, f_m$ az A -n definiált funkcionális függőségek halmaza ($f_i : P_i \setminus rarr Q_i, i = 1, \dots, m$).

- Adattábla (reláció) R felett: $T \subseteq dom(A_1) \times \dots \times dom(A_n)$, amely eleget tesz az F -beli függőségeknek.

Példa:

FÓRUM KÖVETÉSE (A, F), ahol

$A = \{felhasználónév, email, név, hírfolyam azonosító, megnevezés\}$

$F = \{f_1, f_2\}$

$f_1 : \{felhasználónév\} \rightarrow \{email, név\}$

$f_2 : \{hírfolyam azonosító\} \rightarrow \{megnevezés\}$

Származtatható függőségek:

$f_3 : \{felhasználónév\} \rightarrow \{email\}$

$f_4 : \{hírfolyam azonosító, felhasználónév\} \rightarrow \{név\}$

... stb.

Egyszerű szabályok

- Szétvágási szabály:

ha $X \rightarrow \{B_1, \dots, B_k\}$ akkor

$X \rightarrow B_1, \dots, X \rightarrow B_k$ ($B_i \in A$ attribútum, $i = 1, \dots, k$).

◦ Példa: {felhasználónév} → {email, név} ezért {felhasználónév} → {email} és {felhasználónév} → {név}

- Egyesítési szabály:

ha $X \rightarrow B_1, \dots, X \rightarrow B_k$, akkor

$$X \rightarrow \{B_1, \dots, B_k\}$$

- **Vigyázat!** Ha $B_1, \dots, B_k \setminus rarr X$, ebből nem következik, hogy $B_1 \setminus rarr X, \dots, B_k \setminus rarr X$!

- Példa: Fuvar (gkvez, rendszám, indul, érkezik)

▪ {rendszám, indul} $\setminus rarr$ {gkvez}, de ebből nem következik, hogy rendszám $\setminus rarr$ gkvez és indul $\setminus rarr$ gkvez

Armstrong-axiómák

1. Reflexivitás: Ha $Y \subseteq X$, akkor $X \setminus rarr Y$
2. Bővítés: Ha $X \setminus rarr Y$, akkor $X \cup Z \setminus rarr Y \cup Z$
3. Tranzitivitás: Ha $X \setminus rarr Y$ és $Y \setminus rarr Z$, akkor $X \setminus rarr Z$

Állítás: Az Armstrong-axiómák segítségével egy adott függési halmazból következő bármely függés formálisan levezethető. (Levezetésen az axiómák véges sokszori alkalmazását értjük a formális logika szabályai szerint.)

Kulcsok meghatározása

- $K (\subseteq A)$ akkor és csak akkor szuperkulcs, ha $K \setminus rarr A$
 - A függések alapján meg lehet-e határozni?

Attribútumhalmaz lezártja

- Legyen $R(A, F)$ relációséma, és $X \subset A$
- Az X attribútumhalmaz lezártja (X^+) az összes X -től függő attribútum

Példa X^+ meghatározására

$R = (Z, F)$, ahol $Z = \{A, B, C, D, E\}$, és az F-beli függések:

$$\{C\} \rightarrow \{A\}, \quad \{B\} \rightarrow \{C, D\}, \quad \{D, E\} \rightarrow \{C\}$$

$\{B\}^+$ meghatározása:

$$\begin{array}{ll} X^{(0)} = \{B\} & \text{függőség: } \{B\} \rightarrow \{C, D\} \\ X^{(1)} = \{B\} \cup \{C, D\} = \{B, C, D\} & \text{függőség: } \{C\} \rightarrow \{A\} \\ X^{(2)} = \{B, C, D\} \cup \{A\} = \{A, B, C, D\} & \text{függőség: nincs megfelelő} \end{array}$$

$$\text{Tehát } \{B\}^+ = \{A, B, C, D\}$$

Állítás: Legyen $R(A, F)$ relációséma. Egy $K (\subseteq A)$ attribútumhalmaz akkor és csak akkor szuperkulcs, ha $K^+ = A$.

- Kulcs meghatározása: Először legyen $K = A$, ez minden szuperkulcs. K -ból sorra elhagyunk attribútumokat, és minden ellenőrzéssel, hogy $K^+ = A$ teljesül-e.
- Példa kulcs meghatározására:

$$\begin{array}{l} R = (Z, F), \text{ ahol } Z = \{A, B, C, D, E\}, \text{ és az F-beli függések:} \\ \{C\} \rightarrow \{A\}, \quad \{B\} \rightarrow \{C, D\}, \quad \{D, E\} \rightarrow \{C\} \end{array}$$

Láttuk, hogy $\{B\}^+ = \{A, B, C, D\}$, vagyis $\{B\} \rightarrow \{A, B, C, D\}$,
ezért $\{B, E\} \rightarrow \{A, B, C, D, E\}$, vagyis $\{B, E\}$ szuperkulcs.

Mivel sem B, sem E nem hagyható el, ezért **{B, E} kules**.

Több kules nincs, mert Z-ból B-t elhagyva

$$\{A, C, D, E\}^+ = \{A, C, D, E\},$$

Z-ból E-t elhagyva

$$\{A, B, C, D\}^+ = \{A, B, C, D\}$$

tehát B és E bármely kulcsban kell hogy szerepeljen.

Függéshalmaz lezártjának meghatározása

- Függéshalmaz lezártja: az összes F-ből levezethető függés halmaza. Jelölése F^+ .
- Algoritmus F+ meghatározására:
 - i. Vegyük az A attribútumhalmaz összes részhalmazát.
 - ii. minden X részhalmazhoz állítsuk elő X^+ -t.
 - iii. Valamennyi $Y \subseteq X^+$ -ra az $X \subseteq Y$ függést felvesszük F^+ -ba.

8.3.5. Felbontás (dekompozíció)

- Ha egy reláció nem megfelelő normálformában van, akkor a reláció dekompozíciójára van szükség

- Veszteségmentes felbontás (másnéven hűséges felbontás)
 - ha reláció dekompozíciójánál nem vesztettünk információt

- garantálja hogy dekompozíció után a relációk természetes összecsatolása ugyanazt a relációt fogja eredményezni (másképp: $T = T_1 \bowtie T_2$, ahol T az eredeti tábla, T_1 és T_2 a dekompozíció eredményei)

Hűséges felbontás

FÓRUM_KÖVETÉSE				
felhasználónév	email	név	hírfolyam azonosító	megnevezés
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1	Adatházis kérdések
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2	PHP hírek
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4	Milyen gépet vegyek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	2	PHP hírek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	3	Ki a legjobb tanár
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1	Adatházis kérdések
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2	PHP hírek
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3	Ki a legjobb tanár
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter	4	Milyen gépet vegyek

$X = \{\text{felhasználónév, email, név, hírfolyam azonosító}\}$

$Y = \{\text{hírfolyam azonosító, megnevezés}\}$

T_1			
felhasználónév	email	név	hírfolyam azonosító
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	2
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	3
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter	4

T_2	
hírfolyam azonosító	megnevezés
1	Adatházis kérdések
2	PHP hírek
3	Ki a legjobb tanár
4	Milyen gépet vegyek

Nem hűséges felbontás

FÓRUM_KÖVETÉSE				
felhasználónév	email	név	hírfolyam azonosító	megnevezés
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1	Adatházis kérdések
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2	PHP hírek
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4	Milyen gépet vegyek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	2	PHP hírek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	3	Ki a legjobb tanár
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1	Adatházis kérdések
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2	PHP hírek
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3	Ki a legjobb tanár
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter	4	Milyen gépet vegyek

$X = \{\text{felhasználónév, megnevezés}\}$

$Y = \{\text{email, név, hírfolyam azonosító, megnevezés}\}$

T_1			
felhasználónév	megnevezés		
pbalazs	Adatházis kérdések		
pbalazs	PHP hírek		
pbalazs	Milyen gépet vegyek		
pkardos	PHP hírek		
pkardos	Ki a legjobb tanár		
gnemeth	Adatházis kérdések		
gnemeth	PHP hírek		
gnemeth	Ki a legjobb tanár		
bodnaar	Milyen gépet vegyek		

T_2			
email	név	hírfolyam azonosító	megnevezés
pbalazs@inf.u-szeged.hu	Balázs Péter	1	Adatházis kérdések
pbalazs@inf.u-szeged.hu	Balázs Péter	2	PHP hírek
pbalazs@inf.u-szeged.hu	Balázs Péter	4	Milyen gépet vegyek
pkardos@inf.u-szeged.hu	Kardos Péter	2	PHP hírek
pkardos@inf.u-szeged.hu	Kardos Péter	3	Ki a legjobb tanár
gnemeth@inf.u-szeged.hu	Németh Gábor	1	Adatházis kérdések
gnemeth@inf.u-szeged.hu	Németh Gábor	2	PHP hírek
gnemeth@inf.u-szeged.hu	Németh Gábor	3	Ki a legjobb tanár
bodnaar@inf.u-szeged.hu	Bodnár Péter	4	Milyen gépet vegyek

- Függőségörző felbontás

Függőségörző felbontás

$R = (A, F)$ felbontása X, Y szerint $R_1 = (X, F_1)$ és $R_2 = (Y, F_2)$, ahol F_1 úgy választandó meg, hogy F_1^+ az F^+ azon részhalmazával legyen egyenlő, amely csak X -beli attribútumokat tartalmaz, F_2 hasonlóan.

A felbontás **függőségörző**, ha $F_1 \cup F_2$ az eredeti F bázisát adják.

Egy hűséges felbontás nem feltétlenül függőségörző.

Példa: minden dolgozó a hozzá legközelebb lakó osztályvezetőhöz:

Dolgozó (név, adószám, cím, osztálykód, osztálynév, vezAdószám)

{cím} → {vezAdószám} függés lép fel. A

Dolg (név, adószám, cím, osztálykód)

Oszt (osztálykód, osztálynév, vezAdószám)

felbontás hűséges, de a fenti függőség elvész.

- Heath tétele:

- $R(B, C, D)$, ahol B, C és D diszjunkt attribútumhalmazok

- Ha $C \setminus rarr D$, akkor az $R_1(B, C), R_2(C, D)$ felbontás hűséges.

1. normálforma (1NF)

- Egy relációséma 1NF-ben van, ha az attribútumok értéktartománya csak egyszerű (atomi) adatokból áll (nem tartalmaz például listát vagy struktúrát)
 - Ennek teljesülését már a relációséma definíciójánál feltételeztük.
 - Az 1NF-re hozást az E-K modell \rarr relációs modell leképezésnél megvalósítottuk.
- Definíciók
 - Adott $R = (A, F)$, $X, Y \subseteq A$, és $X \setminus rarr Y$
 - X -től teljesen függ Y , ha bármely attribútumot elhagyva a függőség már nem teljesül
 - Egy attribútum elsődleges attribútum ha szerepel a relációséma valamely kulcsában, ellenkező esetben másodlagos attribútum

2. normálforma (2NF)

- Egy $R = (A, F)$ relációséma 2NF-ben van, ha minden másodlagos attribútum teljesen függ bármely kulcstól.
- Következmények:
 - Ha minden kulcs egy attribútumból áll, akkor a séma 2NF-ben van. Például:
 - FELHASZNÁLÓ(felhasználónév, jelszó, email, vezetéknév, keresztnév, utolsó belépés időpontja)
 - Kulcsok: {felhasználónév} {email}
 - egyelemű kulcsok, ebből következik hogy ha elemet hagynánk el belőlük, akkor üreshalmazt kapnánk, ami már nem azonosítja be egyértelműen a másodlagos (vagyis nem kulcs) attribútumokat
 - Ha a sémban nincs másodlagos attribútum, akkor 2NF-ben van. Például:
 - FUVAR(gkvez, rendszám, indul, érkezik)
 - Kulcsok: {gkvez, indul}, {gkvez, érkezik}, {rendszám, indul}, {rendszám, érkezik}
- A séma nincs 2NF-ben, ha egy kulcs részhalmazától függ (egy vagy több másodlagos attribútum)
- 2NF-re hozás: a sémát felbontjuk Heath tétele szerint, a normálformát sértő függőség mentén

FÓRUM_KÖVETÉSE

felhasználónév	email	név	hírfolyam azonosító	megnevezés
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1	Adatbázis kérdések
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2	PHP hírek
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4	Milyen gépet vegyek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	2	PHP hírek
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	3	Ki a legjobb tanár
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1	Adatbázis kérdések
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2	PHP hírek
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3	Ki a legjobb tanár
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter	4	Milyen gépet vegyek

Kulcsok: {felhasználónév, hírfolyam azonosító}, {email, hírfolyam azonosító}

Kulcs részhalmazától való függés:

- (1) {hírfolyam azonosító} → {megnevezés}
- (2) {felhasználónév} → {email, név}
- (3) {email} → {felhasználónév, név}

Felbontás az (1) függés mentén (Heath-tétel alkalmazása):

C = {hírfolyam azonosító}

D = {megnevezés}

B = {felhasználónév, email, név}

FÓRUM_KÖVETÉSE

felhasználónév	email	név	hírfolyam azonosító
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	1
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	2
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter	4
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	2
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter	3
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	1
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	2
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor	3
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter	4

HÍRFOLYAM

hírfolyam azonosító	megnevezés
1	Adatbázis kérdések
2	PHP hírek
3	Ki a legjobb tanár
4	Milyen gépet vegyek

FELHASZNÁLÓ

felhasználónév	email	név
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter

KÖVETI

felhasználónév	hírfolyam azonosító
pbalazs	1
pbalazs	2
pbalazs	4
pkardos	2
pkardos	3
gnemeth	1
gnemeth	2
gnemeth	3
bodnaar	4

HÍRFOLYAM

hírfolyam azonosító	megnevezés
1	Adatbázis kérdések
2	PHP hírek
3	Ki a legjobb tanár
4	Milyen gépet vegyek

3. normálforma (3NF)

- $X, Z \subseteq A$ és $X \not\rightarrow \text{rarr} Z$
 - X -től tranzitívan függ Z , ha van olyan $Y (\subseteq A)$, amelyre $X \not\rightarrow \text{rarr} Y$ és $Y \not\rightarrow \text{rarr} Z$, de X nem függ Y -tól, és az $Y \not\rightarrow \text{rarr} Z$ függés teljesen nemtriviális (vagyis $Y \cap Z$ üres). Ellenkező esetben X -től közvetlenül függ Z .
- Egy $R = (A, F)$ relációséma 3NF-ben van, ha minden másodlagos attribútuma közvetlenül függ bármely kulcstól.
 - Következmény: Ha a sémában nincs másodlagos attribútum, akkor 3NF-ben van.

FELHASZNÁLÓ

felhasználónév	email	név
pbalazs	pbalazs@inf.u-szeged.hu	Balázs Péter
pkardos	pkardos@inf.u-szeged.hu	Kardos Péter
gnemeth	gnemeth@inf.u-szeged.hu	Németh Gábor
bodnaar	bodnaar@inf.u-szeged.hu	Bodnár Péter

Tranzitív függés?

{felhasználónév} → {email} → {név}

{email} → {név} teljesen nemtriviális

VISZONT: {email} → {felhasználónév} !

Nem tranzitív függés, mivel a felhasználónév függ az emailtől.

- A séma nincs 3NF-ben, ha egy vagy több másodlagos attribútum tranzitívan függ valamely kulcstól
- 3NF-re hozás: ha a $K \not\rightarrow \text{rarr} Y \not\rightarrow \text{rarr} Z$ tranzitív függés fennáll, akkor a sémát felbonthatjuk Heath tétele szerint az $Y \not\rightarrow \text{rarr} Z$ függés mentén.
- Példa:
 - DOLGOZÓ(adószám, TAJ szám, dolgozó neve, projektkód, projekt neve)
 - Egy dolgozó csak egy projekten dolgozhat. Ha többen dolgoznak ugyanazon a projekten, akkor a projekt neve ismétlődik.
 - Kulcsok: {adószám}, {TAJ szám}

- A séma 2NF-ben van (csak egyszerű kulcs van benne).
- Tranzitív függés: {adószám} \rarr {projektkód} \rarr {projekt neve}
- Felbontás Heath-tétele alapján:
 - C = {projektkód}
 - D = {projekt neve}
 - B = {adószám, TAJ szám, dolgozó neve}

- Felbontás után:
 - DOLGOZÓ(adószám, TAJ szám, dolgozó neve, projektkód)
 - PROJEKT(projektkód, projekt neve)

- **Állítás:** Ha egy relációséma 3NF-ben van, akkor 2NF-ben is van.

Boyce-Codd normálforma (BCNF)

- Egy relációséma BCNF-ben van, ha bármely nemtriviális $L \rightarrow Z$ függés esetén L szuperkulcs.
- Vagyis: A sémban csak kulcstól való függés van, ezen kívül nincs „kóból függés”.
- A séma nincs BCNF-ben, ha van benne olyan nemtriviális függés, amelynek bal oldalán nem szuperkulcs áll.
- BCNF-re hozás: a sémát felbontjuk Heath tétele szerint, a normálformát sértő függőség mentén.
- Példa:
 - DOLGOZÓ(adószám, TAJ szám, projektkód)
 - Egy dolgozó több projekten dolgozhat. Ha valaki több projekten dolgozik, akkor a TAJ szám ismétlődik.
 - Kulcsok: {adószám, projektkód}, {TAJ szám, projektkód}
 - A séma 3NF-ben van (nincs másodlagos attribútum).
 - BCNF-et sértő függés: {adószám} \rarr {TAJ szám} (adószám nem szuperkulcs)
 - Felbontás Heath-tétele alapján:
 - C = {adószám}
 - D = {TAJ szám}
 - B = {projektkód}
 - Felbontás után:
 - DOLGOZÓ(adószám, TAJ szám)
 - PROJEKT(adószám, projektkód)
- **Állítás:** Ha egy relációséma BCNF-ben van, akkor 3NF-ben is van.

4. normálforma (4NF)

- Példa: Rendelhet (nagyker, kisker, áru), BCNF-ben van.

Nagyker	Kisker	Áru
N1	K1	A1
N1	K1	A2
N1	K1	A3
N1	K2	A1
N1	K2	A2
N1	K2	A3
N2	K2	A1
N2	K2	A4
N2	K3	A1
N2	K3	A4

Tulajdonság: Ha (N_i, K_j) és (N_i, A_k) , akkor (N_i, K_j, A_k) .

- $K, L \subseteq A$, és legyen $M = A \setminus (K \cup L)$
 - K -tól többéértékűen függ L ($K \rightarrow rarr \rightarrow rarr L$), ha bármely R feletti T tábla esetén ha két sor megegyezik K -n, akkor a kombinációjuk is szerepel a táblában

◦ Példa:

- nagyker $\rightarrow\rightarrow$ kiskér
- viszont kiskér $\rightarrow\rightarrow$ nagyker már nem igaz, mert például (N2, K2) és (K2, A2), de (N2, K2, A2) már nem teljesül!

A $K \rightarrow\rightarrow L$ függés **nemtriviális**, ha $K \cap L = \emptyset$ és $K \cup L \neq A$.

Állítás. Ha $K \rightarrow L$, akkor $K \rightarrow\rightarrow L$.

Bizonyítás: $t = t_j$ választással nyilvánvaló.

Állítás. Ha $K \rightarrow\rightarrow L$, akkor $K \rightarrow\rightarrow M$.

Bizonyítás: a szimmetriából nyilvánvaló.

Példa: Ha nagyker $\rightarrow\rightarrow$ kiskér, akkor nagyker $\rightarrow\rightarrow$ áru is teljesül.

Fagin tétele

Egy relációséma 4NF-ben van, ha minden nemtriviális $K \rightarrow\rightarrow L$ függés esetén K szuperkulcs.

$R(B, C, D)$, ahol B, C és D diszjunkt attribútumhalmazok.

R felbontása az $R_1(B \cup C), R_2(C \cup D)$ sémákra akkor és csak akkor hűséges, ha $C \rightarrow\rightarrow D$ fennáll.

A séma nincs 4NF-ben, ha van benne olyan nemtriviális többértékű függés, amelynek bal oldalán nem szuperkulcs áll.

Ekkor a tábla redundanciát tartalmazhat. Ha ugyanis $K \rightarrow\rightarrow L$ és K nem szuperkulcs, akkor a táblában több olyan sor lehet, amely K -n megegyezik, és ezekben a sorokban az L és M -értékek redundánsan szerepelnek.

4NF-re hozás. Dekompozíció a $K \rightarrow\rightarrow L$ függőség szerint:

az $R(A)$ sémát felbontjuk az $R_1(K \cup L)$ és $R_2(K \cup M)$ sémákra. Ez hűséges dekompozíció Fagin tétele szerint.

Állítás. Ha egy $R = (A, F)$ séma 4NF-ben van, akkor BCNF-ben is van.

Példa: A Rendelhet (nagyker, kiskér, áru) séma felbontása a nagyker $\rightarrow\rightarrow$ kiskér függés szerint:

Szállít (nagyker, kiskér), Kínál (nagyker, áru)

- A Rendelhet tábla 4NF felbontása:

Az eredeti Rendelhet tábla:

Nagyker	Kiskér	Áru
N1	K1	A1
N1	K1	A2
N1	K1	A3
N1	K2	A1
N1	K2	A2
N1	K2	A3
N2	K2	A1
N2	K2	A4
N2	K3	A1
N2	K3	A4

A Szállít tábla:

Nagyker	Kiskér
N1	K1
N1	K2
N2	K2
N2	K3

A Kínál tábla:

Nagyker	Áru
N1	A1
N1	A2
N1	A3
N2	A1
N2	A3
N2	A4

- **Állítás:** Ha egy relációséma 4NF-ben van, akkor hűséges felbontással nem lehet redundanciát megszüntetni.

8.4. 2. Az SQL adatbázisnyelv: Az adatdefiníciós nyelv (DDL) és az adatmanipulációs nyelv (DML). Relációsémák definiálása, megszorítások típusai és létrehozásuk. Adatmanipulációs lehetőségek és lekérdezések.

8.4.1. Az SQL nyelv

SQL = Structured Query Language (= strukturált lekérdező nyelv). A relációs adatbáziskezelés szabványos nyelve. Nem algoritmikus nyelv, de algoritmikus nyelvekbe beépíthető (beágyazott SQL).

Általános jellemzők

- Az SQL utasításait két fő csoportba szokták sorolni:

- DDL (= Data Definition Language): adatstruktúra definiáló utasítások.
- DML (= Data Manipulation Language): adatokon műveletet végző utasítások

Szintaxis

- Kisbetű és nagybetű a nyelv alapszavaiban egyenértékű
- Utasítások sorfolytonosan írhatók, lezárás pontosvesszővel
- Változó nincs, csak tábla- és oszlopnevekre lehet hivatkozni. Kifejezésben hivatkozás egy tábla adott oszlopára: tábla.oszlop (ha a tábla egyértelmű, akkor elhagyható)
- Alias név: név AS másodnév (egyes implementációkban AS elhagyható).
- Szövegkonstans: 'szöveg'
- Dátum: DATE '1968-05-12'. Egyes rendszerek az SQL szabványtól eltérő konvenciót alkalmaznak, például 13-NOV-94 (Oracle)
- Idő: TIME '15:31:02.5' (óra, perc, másodperc)
- Stringek konkatenációja: + vagy ||
- Relációjelek: =, <=, >=, !=, <>
- Logikai műveletek: AND, OR, NOT. Az SQL rendszerek "háromértékű logikát" használnak, vagyis a TRUE és FALSE mellett a NULL (definiálatlan) érték is felléphet. Ha egy kifejezés valamelyik eleme NULL, akkor a kifejezés értéke is NULL lesz.
 - Az SQL-szabvány szerint egy logikai kifejezés értéke ISMERETLEN (UNKNOWN), ha benne NULL érték szerepel.
- Az utasítások szintaxisának leírásánál az elhagyható részleteket szögletes zárójellel jelöljük.

8.4.2. Relációsémák definiálása (DDL)

- Relációséma létrehozására a CREATE TABLE utasítás szolgál, amely egyben egy üres táblát is létrehoz a sémához. Az attribútumok definiálása mellett a kulcsok és külső kulcsok megadására is lehetőséget nyújt:
 - CREATE TABLE táblanév (oszlopnév adattípus [feltétel],, oszlopnév adattípus [feltétel] [, táblaFeltételek]);
- Az adattípusok (rendszerenként eltérők lehetnek):
 - CHAR(*n*): *n* hosszúságú karaktersorozat
 - VARCHAR(*n*): legfeljebb *n* hosszúságú karaktersorozat
 - INTEGER: egész szám (röviden INT)
 - REAL: valós (lebegőpontos) szám, másnéven FLOAT
 - DECIMAL(*n*[*d*]): *n* jegyű decimális szám, ebből *d* tizedesjegy
 - DATE: dátum (év, hónap, nap)
 - TIME: idő (óra, perc, másodperc)
- Az adattípushoz "DEFAULT érték" megadásával alapértelmezett érték definiálható. Ha ilyet nem adunk meg, az alapértelmezett érték NULL
- Feltételek (egy adott oszlopra vonatkoznak):
 - PRIMARY KEY: elsődleges kulcs (csak egy lehet)
 - UNIQUE: kulcs (több is lehet)
 - REFERENCES tábla(oszlop) [ON-feltételek]: külső kulcs
- Táblafeltételek (az egész táblára vonatkoznak):
 - PRIMARY KEY (oszloplista): elsődleges kulcs
 - UNIQUE (oszloplista): kulcs
 - FOREIGN KEY (oszloplista) REFERENCES tábla(oszloplista) [ON-feltételek]: külső kulcs
- Ha a (külső) kulcs több oszlopból áll, akkor csak táblafeltétel formájában adható meg.
- A PRIMARY KEY (elsődleges kulcs) és UNIQUE (kulcs) közötti különbségek:
 - Egy sémában csak egy elsődleges kulcs, de tötszöleges számú további kulcs lehet
 - Külső kulcs általában a másik tábla elsődleges kulcsára hivatkozik.
 - Egyes DBMS-ek az elsődleges kulcshoz automatikusan indexet hoznak létre.

- A CREATE TABLE utasítással tulajdonképpen egy R = (A, F) relációséma adunk meg, ahol F megadására szolgálnak a kulcsfeltételek. Ha a relációséma BCNF-ben van, akkor ezzel az összes függés megadható, hiszen ekkor csak szuperkulcstól lehet nemtriviális függés
- Példa (az alábbi relációséma SQL-ben való létrehozása):
 - Osztály (osztálykód, osztálynév, vezAdószám)
 - Dolgozó (adószám, név, lakcím, osztálykód)

```
CREATE TABLE Osztály
  ( osztálykód     CHAR(3)   PRIMARY KEY,
    osztálynév     CHAR(20),
    vezAdószám    DECIMAL(10)
  );
CREATE TABLE Dolgozó
  ( adószám      DECIMAL(10)  PRIMARY KEY,
    név          CHAR(30),
    lakcím       CHAR(40)   DEFAULT 'ismeretlen',
    osztálykód   CHAR(3)    REFERENCES Osztály(osztálykód)
  );
```
- A tábla módosításakor a definált kulcsfeltételek automatikusan ellenőrzésre kerülnek. PRIMARY KEY és UNIQUE esetén ez azt jelenti, hogy a rendszer nem enged olyan módosítást illetve új sor felvételét, amely egy már meglévő kulccsal ütközne.
- REFERENCES (külső kulcs hivatkozás) esetén ON-feltételek megadásával szabályozhatjuk a rendszer viselkedését (jelölje T1 a hivatkozó és T2 a hivatkozott táblát):
 - **Alapértelmezés** (ha nincs ON-feltétel): T1-ben nem megengedett olyan beszúrás és módosítás, amely T2-ben nem létező kulcs értékre hivatkozna, továbbá T2-ben nem megengedett olyan kulcs módosítása vagy sor törlése, amelyre T1 hivatkozik.
 - **ON UPDATE CASCADE**: ha T2 egy sorában változik a kulcs értéke, akkor a rá való T1-beli hivatkozások is megfelelően módosulnak (módosítás továbbgyűrűzése).
 - **ON DELETE CASCADE**: Ha T2-ben törlünk egy sort, akkor T1-ben is törlődnek a rá hivatkozó sorok (törles továbbgyűrűzése).
 - **ON UPDATE SET NULL**: ha T2 egy sorában változik a kulcs értéke, akkor T1-ben a rá való külső kulcs hivatkozások értéke NULL lesz.
 - **ON DELETE SET NULL**: ha T2-ben törlünk egy sort, akkor T1-ben a rá való külső kulcs hivatkozások értéke NULL lesz.
- A kulcsfeltételek ellenőrzése csak indexekkel oldható meg hatékonyan.
- Relációséma törlése:
 - **DROP TABLE táblanév;**
 - Hatására a séma és a hozzá tartozó adattábla törlődik.
- Relációséma módosítása:
 - **ALTER TABLE táblanév [ADD (újelem, ..., újelem)] [MODIFY (módosítás, ..., módosítás)] [DROP (oszlop, ..., oszlop)];**
 - Az ALTER TABLE utasítás szintaxisa és szemantikája rendszerenként eltérő, például oszlopok törlését nem minden rendszer engedi meg.

Indexek létrehozása

- Az indexek kezelése nem része az SQL2 szabványnak, de valamilyen formában minden RDBMS támogatja
- Index létrehozása:
 - **CREATE [UNIQUE] INDEX indexnév ON tábla(oszloplista);**
 - A megadott tábla felsorolt oszlopaira, mint indexkulcsra generál indexet.
 - Ha UNIQUE szerepel, akkor a tábla nem tartalmazhat két azonos indexkulcsú rekordot
- Index törlése:
 - **DROP INDEX indexnév;**
- Példák:
 - CREATE INDEX DolgInd1 ON Dolgozó(név); CREATE INDEX DolgInd2 ON Dolgozó(osztálykód,név);
 - Az első példa egyszerű indexkulcsot tartalmaz, amely a dolgozók név szerinti keresését, illetve rendezését támogatja. A második példában szereplő összetett indexkulcs az osztálykód szerinti, osztályon belül pedig név szerinti keresést/rendezést segíti, mivel a rendszerek általában az osztálykód és név attribútumok konkatenációjával képezik az indexkulcsot. Ez a megoldás viszont a pusztán név szerinti keresést nem támogatja.

8.4.3. Adattábla aktualizálása (DML)

- A táblába új sor felvétele:

- **INSERT INTO táblanév [(oszloplista)] VALUES (értéklista);**
 - Ha oszloplista nem szerepel, akkor valamennyi oszlop értéket kap a CREATE TABLE-ben megadott sorrendben. Egyébként csak az oszloplistában megadott mezők kapnak értéket, a többi mező értéke NULL lesz.
 - Példa:
 - `INSERT INTO Dolgozó (név, adószám) VALUES ('Tóth Aladár', 1111);`
 - A táblába adatokat tölthetünk át másik táblából is, ha a VALUES(értéklista) helyére egy lekérdezést írunk
- Sor(ok) módosítása:
 - **UPDATE táblanév SET oszlop = kifejezés, ..., oszlop = kifejezés [WHERE feltétel];**
 - Az értékadás minden olyan soron végrehajtódik, amely eleget tesz a WHERE feltételnek. Ha WHERE feltétel nem szerepel, akkor az értékadás az összes sorra megtörténik.
 - Példa:
 - `UPDATE Dolgozó SET lakcím = 'Szeged, Rózsa u. 5.' WHERE név = 'Kovács József';`
- Sor(ok) törlése:
 - **DELETE FROM táblanév [WHERE feltétel];**
 - Hatására azok a sorok törlődnek, amelyek eleget tesznek a WHERE feltételnek. Ha a WHERE feltételt elhagyjuk, akkor az összes sor törlődik (de a séma megmarad).
 - Példa:
 - `DELETE FROM Dolgozó WHERE név = 'Kovács József';`

8.4.4. Lekérdezés (DML)

- Lekérdezésre a SELECT utasítás szolgál, amely egy vagy több adattáblából egy eredménytáblát állít elő.
- Az eredménytábla a képernyőn listázásra kerül, vagy más módon használható fel. (Egyetlen SELECT akár egy komplex felhasználói programot helyettesíthet!)
- A "SELECT DISTINCT A_1, \dots, A_n FROM T_1, \dots, T_m WHERE feltétel" utasítás egyenértékű a következő relációs algebrai kifejezéssel:
 - $E = \pi_{A_1, \dots, A_n}(\sigma_{feltétel}(T_1 \times \dots \times T_m))$
 - Vagyis, a felsorolt táblák Descartes-szorzatából szelektáljuk a feltételnek eleget tévő sorokat, majd ezekből projekcióval választjuk ki az E eredménytábla oszlopait.
 - A DISTINCT opciót akkor kell kiírni, ha az eredménytáblában az azonos sorokból csak egyet kívánunk megtartani.
- Ha oszloplista helyére * karaktert írunk, ez valamennyi oszlop felsorolásával egyenértékű. A SELECT legegyszerűbb változatával adattábla listázását érhetjük el:
 - `SELECT * FROM T;`

A relációs algebra műveleteinek megvalósítása:

- Projekció
 - `SELECT [DISTINCT] A_1, \dots, A_n FROM T;`
 - Pl.: `SELECT DISTINCT szerző, cím FROM Könyv;`
- Szelekció
 - `SELECT * FROM T WHERE feltétel;`
 - Pl.: `SELECT * FROM Könyv WHERE kivétel < 2013.01.01;`
- Descartes-szorzat: $T_1 \times T_2$
 - `SELECT * FROM T1, T2;`
- Természetes összekapcsolás
 - Állítsuk elő például az Áru (cikkszám, megnevezés) és Vásárlás (cikkszám, mennyiségi) táblák természetes összekapcsolását:
 - `SELECT Áru.cikkszám, megnevezés, mennyiségi FROM Áru, Vásárlás WHERE Áru.cikkszám = Vásárlás.cikkszám;`
 - A fentivel egyenértékű, szintén gyakran használt szintaxis:
 - `SELECT Áru.cikkszám, megnevezés, mennyiségi FROM Áru INNER JOIN Vásárlás ON Áru.cikkszám = Vásárlás.cikkszám;`

- Megjegyzés: A fenti példákban a SELECT után nem elegendő csak „cikkszám”-ot írni, annak ellenére, hogy esetünkben „Áru.cikkszám = Vásárlás.cikkszám”, tehát mindegy, melyik cikkszámot választja a rendszer. Általában, ha egy lekérdezésben több azonos oszlopnév szerepel, az SQL rendszerek megkövetelik a táblanév megadását

- Külső összekapcsolás

- Az SQL szabvány szerint a LEFT, RIGHT vagy FULL OUTER JOIN kulcsszavakkal adható meg külső összekapcsolás
- Például:
 - SELECT Áru.cikkszám, megnevezés, mennyiség FROM Áru LEFT OUTER JOIN Vásárlás ON Áru.cikkszám = Vásárlás.cikkszám;

- Théta join

- SELECT * FROM T1,T2 WHERE feltétel;

- Unió

- (SELECT _ FROM T1) UNION (SELECT _ FROM T2);
- A két SELECT eredménytáblája kompatibilis kell hogy legyen

- Metszet

- (SELECT _ FROM T1) INTERSECT (SELECT _ FROM T2);
- A két SELECT eredménytáblája kompatibilis kell hogy legyen

- Különbség

- (SELECT _ FROM T1) EXCEPT (SELECT _ FROM T2);
- A két SELECT eredménytáblája kompatibilis kell, hogy legyen. Egyes rendszereknél EXCEPT helyett MINUS használatos.

Alias nevek

- A SELECT után megadott oszloplista valójában nem csak oszlopneveket, hanem tetszőleges kifejezéseket is tartalmazhat, és az eredménytábla oszlopainak elnevezésére alias neveket adhatunk meg
- Például:
 - SELECT név AS áru, egységár*mennyiség AS érték FROM Raktár;

Függvények

- Például:

- Abszolút érték
 - ABS(n)
- Konverzió kisbetűsre
 - LOWER(char)
- stb...

Összesítő függvények

- Egy oszlop értékeiből egyetlen értéket hoznak létre (például átlag). Általános alakjuk:
 - függvénynév ([DISTINCT] oszlopnév)
- Ha DISTINCT szerepel, akkor az oszlopan szereplő azonos értékeket csak egyszer kell figyelembe venni. A számításnál a NULL értékek figyelmen kívül maradnak. Az egyes függvények:
 - AVG: átlagérték.
 - SUM: összeg.
 - MAX: maximális érték.
 - MIN: minimális érték.
 - COUNT: elemek száma. Ennél a függvénynél oszlopnév helyére * is írható, amely valamennyi oszlopot együtt jelenti.
- Példa:
 - SELECT AVG(fizetés) FROM Dolgozó;
 - Az eredménytábla egyetlen elemből áll, amely az átlagfizetést adja.

Csoportosítás (GROUP BY, HAVING)

- Ha a tábla sorait csoportonként szeretnénk összesíteni, akkor a SELECT utasítás a **GROUP BY oszloplista** alparancsal bővítendő.
 - Egy csoportba azok a sorok tartoznak, melyeknél oszloplista értéke azonos.
 - Az eredménytáblában egy csoportból egy rekord lesz
 - Példa:
 - SELECT osztkód, AVG(fizetés) FROM Dolgozó GROUP BY osztkód;
 - A Dolgozó táblából osztályonként az átlagfizetést számoljuk
- **Csoportosítási szabály:** A SELECT után összesítő függvényen kívül csak olyan oszlopnév tüntethető fel, amely a GROUP BY-ban is szerepel.
 - Hibás például az alábbi lekérdezés, amely azt szeretné megtudni, hogy az egyes osztályokon kinek a legnagyobb a fizetése:
 - SELECT osztkód, név, MAX(fizetés) AS maxfiz FROM Dolgozó GROUP BY osztkód;
 - A hiba oka: név nem szerepelhet a SELECT után, mert a GROUP BY után sem szerepel.
- A GROUP BY által képezett csoportok közül válogathatunk a **HAVING feltétel** alparancs segítségével: csak a feltételnek eleget tevő csoportok kerülnek összesítésre az eredménytáblába.
 - Példa. Azon osztályok listája, ahol az átlagfizetés > 180 000 Ft:
 - SELECT osztkód, AVG(fizetés) FROM Dolgozó GROUP BY osztkód HAVING AVG(fizetés) > 180000;
- Az eredménytábla rendezése:
 - Bár a relációs modell nem definiálja a rekordok sorrendjét, a gyakorlatban rendszerint valamilyen rendezettségen kívánjuk látni az eredményt
 - Erre szolgál az **ORDER BY oszlopnév [DESC], ..., oszlopnév [DESC]** alparancs, amely a SELECT utasítás végére helyezhető, és az eredménytáblának a megadott oszlopok szerinti rendezését írja elő
 - Az oszlopnév után írt ASC (ascending) növekvő, DESC (descending) csökkenő sorrendben való rendezést jelent. (Alapértelmezés szerint a rendezés növekvő sorrendben történik, ezért ASC kiírása fölösleges)
 - Példa:
 - SELECT osztkód, név, fizetés FROM Dolgozó ORDER BY osztkód, fizetés DESC;

A SELECT utasítás általános alakja

- A SELECT utasítás az alábbi alparancsokból állhat az alábbi sorrendben (a szögletes zárójelben szereplő részek elhagyhatók):
 - SELECT [DISTINCT] oszloplista projekció FROM táblanévlista Descartes-szorzat [WHERE feltétel] szelekció [GROUP BY oszloplista] csoportonként összevonás [HAVING feltétel] csoport-szelekció [ORDER BY oszloplista]; rendezés

Alkérdések

- Ha egy SELECT utasítás WHERE vagy HAVING feltételében olyan logikai kifejezés szerepel, amely SELECT utasítást tartalmaz, ezt alkérdésnek vagy belső SELECT-nek is nevezik. Általában, valamely SQL utasítás belsejében szereplő SELECT utasítást alkérdésnek nevezzük.
- Példa. Az alábbi utasítás azon dolgozók listáját adja, amelyek fizetése kisebb, mint az átlagfizetés:
 - SELECT név, fizetés FROM Dolgozó WHERE fizetés < (SELECT AVG(fizetés) FROM dolgozó);
- Nem csak a logikai kifejezés tartalmazhat alkérdést, hanem az INSERT utasítás is:
 - **INSERT INTO táblanév [(oszloplista)] SELECT ... ;**
 - A SELECT annyi oszlopot kell hogy kiválasszon, amennyit oszloplista tartalmaz. A többi oszlop NULL értéket vesz fel.
 - Példa. Tegyük fel, hogy a Raktár (cikkszám, név, egységár, mennyiség) táblából egy Készlet (áru, érték) táblát szeretnénk létrehozni, amely az árfűréség megnevezését és az aktuálisan tárolt mennyiség értékét tartalmazza. Ez a következőképp lehetséges:
 - CREATE TABLE Készlet (áru CHAR(20), érték INTEGER); INSERT INTO Készlet SELECT név, egységár*mennyiség FROM Raktár;

Nézettáblák (virtuális táblák)

- Egy adatbázisban általában kétféle adatra van szükségünk:
 - alapadatok: tartalmukat aktualizáló műveletekkel módosítjuk
 - származtatott adatok: az alapadatokból generálhatók.
- Származtatott adattáblát például INSERT ... SELECT segítségével is létrehozhatunk (lásd az előző pontot), ekkor viszont az nem követi automatikusan az alapadatok módosulását, ha pedig minden aktualizáló műveletnél újragenerálnánk, az rendkívül lassú lenne. A problémát a nézettábla oldja meg.
- A nézettábla (virtuális tábla, view) nem tárol adatokat. Tulajdonképpen egy transzformációs formula, amelyet úgy képzelhetünk el, mint ha ennek segítségével a tárolt táblák adatait látnánk egy speciális szűrőn, „optikán” keresztül.

- Nézettáblák alkalmazási lehetőségei:
 - Származtatott adattáblák létrehozása, amelyek a törzsadatok módosításakor automatikusan módosulnak (pl. összegzőtáblák).
 - Bizonyos adatok elrejtése egyes felhasználók elől (adatbiztonság vagy egyszerűsítés céljából)
- Nézettábla létrehozása:
 - CREATE VIEW táblanév [(oszloplista)] AS alkérdez;
- A SELECT utasítás eredménytáblája alkotja a nézettáblát. "Oszloplista" megadásával a nézettábla oszlopainak új nevet adhatunk. A CREATE VIEW végrehajtásakor a rendszer csak letárolja a nézettábla definíóját, és majd csak a rá való hivatkozáskor generálja a szükséges adatokat. Ebből adódóan a nézettábla tartalma minden aktuális.
- A nézettáblák általában ugyanúgy használhatók, mint a tárolt adattáblák, vagyis ahol egy SQL parancsban táblánév adható meg, ott rendszerint nézettábla neve is szerepelhet.
- Példa. Származtatott adatok kezelése. A Raktár (cikkszám, név, egységár, mennyiségi) táblából létrehozott nézettábla:
 - CREATE VIEW Készlet (áru, érték) AS SELECT név, egységár*mennyiségi FROM Raktár;
- Ha a nézettábla tartalmát módosítjuk, akkor a módosítás a megfelelő tárolt táblákon hajtódik végre – és természetesen megjelenik a nézettáblában is

8.4.5. Aktív elemek (megszorítások, triggerek)

- Aktív elem: olyan programrész, amely bizonyos szituációban automatikusan végrehajtódik. Ennek speciális esete a megszorítás, ami bizonyos feltételek ellenőrzését jelenti bizonyos helyzetekben.

Atribútumok megszorítása

- A CREATE TABLE-ben valamely attribútum deklarációja után adhatók meg.
- Kulcs feltételek: a CREATE TABLE utasításban adhatók meg a PRIMARY KEY, UNIQUE, REFERENCES kulcsszavakkal. Aktualizálási műveleteknél a megfelelő feltétel automatikus ellenőrzését váltják ki.
- További megszorítások:
 - NOT NULL
 - Adott attribútum értéke nem lehet NULL. Hatására a rendszer megakadályoz minden olyan műveletet, amely az adott attribútum NULL értékét eredményezné. Adatbevitelnél például ez azt jelenti, hogy az attribútum értékét kötelező megadni
 - CHECK (feltétel)
 - Az adott attribútum módosítását a rendszer csak akkor engedi meg, ha a feltétel teljesül.
 - Példa: A dolgozók nemét is nyilvántartjuk (F=férfi, N=nő):
 - CREATE TABLE Dolgozó (adószám DECIMAL(10) PRIMARY KEY, név CHAR(30) NOT NULL, nem CHAR(1) CHECK (nem IN ('F', 'N')), lakcím CHAR(40), osztkód CHAR(3) REFERENCES Osztály(osztkód));
- Értéktartomány definiálása:
 - CREATE DOMAIN név típus [DEFAULT érték] [CHECK (feltétel)];
 - Értéktartomány módosítása ALTER DOMAIN, törlése DROP DOMAIN utasítással történik.
 - Példa. A nemekhez tartozó konstansértékek definiálása:
 - CREATE DOMAIN NemÉrték CHAR(1) CHECK (VALUE IN ('F', 'N')); CREATE TABLE Dolgozó (adószám DECIMAL(10) PRIMARY KEY, név CHAR(30), nem NemÉrték, lakcím CHAR(40));

Táblára vonatkozó megszorítások

- A CREATE TABLE végére, a táblaFeltételeknél helyezendők el.
- Kulcs feltételek: PRIMARY KEY, UNIQUE, FOREIGN KEY kulcsszavakkal. Ha a CHECK feltétel egyszerre több attribútumot érint, akkor szintén a táblaFeltételeknél helyezendő el.
- Példa. Biztonsági ellenőrzésként megköveteljük, hogy a könyvek kölcsönzésénél a kivétel dátuma előzze meg a visszahozási határidőt:
 - CREATE TABLE Könyv (könyvszám DECIMAL(6) PRIMARY KEY, szerző CHAR(30), cím CHAR(30), kivétel DATE, vissza DATE, CHECK (kivétel < vissza));

Általános megszorítások

- Több táblára (általában, a teljes adatbázissémára) vonatkozhatnak. Megadásuk:
 - CREATE ASSERTION név CHECK (feltétel);
 - A feltételben szereplő táblák bármelyikének módosításakor a feltétel ellenőrzésre kerül.

- Példa: A Dolgozó(adószám, név, fizetés, osztálykód) és Osztály(osztálykód, osztálynév, vezAdószám) táblák esetén megköveteljük, hogy a vezetők fizetése legalább 100 000 Ft legyen:

◦ CREATE ASSERTION VezetőFizetés CHECK (NOT EXISTS (SELECT * FROM Dolgozó, Osztály WHERE Dolgozó.adószám = Osztály.vezAdószám AND fizetés < 100000));

- Az önálló megszorítás törlése:

◦ DROP ASSERTION név;

Triggerek

- A trigger egy aktualizálási művelet esetén végrehajtandó programrészletet definiál.

- Megadása:

◦ CREATE TRIGGER név { BEFORE | AFTER | INSTEAD OF } { DELETE | INSERT | UPDATE [OF oszlopok] } ON tábla [REFERENCING [OLD AS régi] [NEW AS új] [FOR EACH ROW] [WHEN (feltétel)] programblokk;

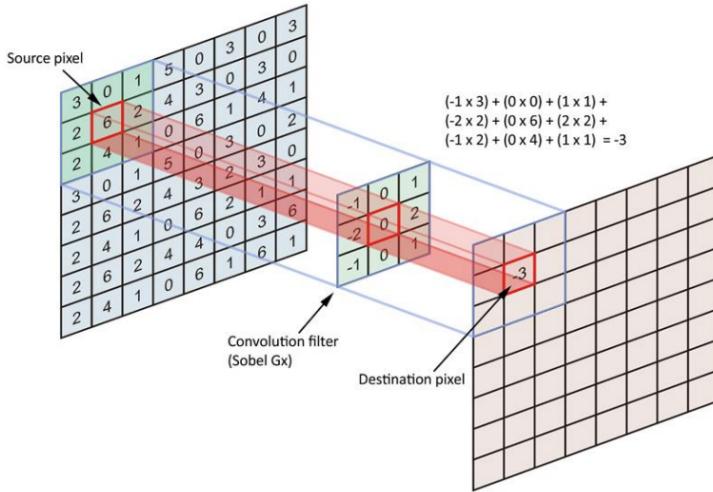
▪ Jelölés: a fenti szintaxis leírásban { x | y } azt jelenti, hogy x és y egyike választható.

▪ név: a trigger neve. **BEFORE, AFTER, INSTEAD OF**: az aktualizálási művelet előtt, után, vagy helyette lép működésbe a trigger. **DELETE, INSERT, UPDATE OF**: az aktualizálási művelet neve. ON tábla: ezen tábla aktualizálásakor lép működésbe a trigger. **REFERENCING**: lehetővé teszi, hogy a tábla aktuális sorának aktualizálás előtti és utáni állapotára névvel hivatkozzunk. **FOR EACH ROW**: ha megadjuk, akkor a trigger a tábla minden egyes sorára lefut, amelyet az aktualizálási művelet érint (sor szintű trigger). Ha nem adjuk meg, akkor egy aktualizálási művelet esetén csak egyszer fut le a trigger (utasítás szintű trigger). **WHEN feltétel**: a megadott feltétel teljesülése esetén hajtódik végre a trigger. **programblokk**: egy vagy több SQL utasításból álló, vagy valamely programozási nyelven írt blokk.

9. Digitális képfeldolgozás

9.1. 1. Simítás/szűrés képtérben (átlagoló szűrők, Gauss simítás és mediánszűrés); élek detektálása (gradiens-operátorokkal és Marr-Hildreth módszerrel).

Konvolúció



Lényege, hogy van egy kernel (a képen "Convolution filter", egy mátrix), amit végigléptetünk egy nála nagyobb mátrixon. minden egyes pozícióban a kernelben lévő számokat összeszorozzuk az "alattuk" lévő számokkal, a szorzatokat szummázzuk, utána ezt az eredményt egy harmadik (vagy a forrás mátrixal egyenlő méretű, vagy nem, attól függ mekkora méretű képet akarunk visszakapni) célmátrixba írjuk. Így a forrásában való végighaladás után kitöltődik az egész célmátrix.

(maszk=kernel)

9.1.1. Átlagoló szűrés

- **Zaj**: a képpont-intenzitások nemkívánatos változása
- Az átlagoló szűrés egy olyan technika, amit képek simítására, másszóval zajtalanítására használnak



eredeti/zajmentes kép

Gauss-zajjal terhelt kép
(az intenzitásváltozás normál eloszlást követ)

só-bors zajjal terhelt kép
(fehér és fekete pontok random előfordulása)

- Lényege, hogy minden egyes pixelt a környezete (ebbe beleszámít a helyettesíteni kívánt pixel is) átlagával helyettesítünk
- Ezt egy olyan konvolúciós szűréssel éri el, ahol a kernel (vagy konvolúciós maszk) egy olyan mátrix, ahol az elemek összege mindig 1
 - Példák konvolúciós maszkokra:

$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$

0	0	$\frac{1}{13}$	0	0
0	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	0
$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$
0	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	0
0	0	$\frac{1}{13}$	0	0

0	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	0
$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$
$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$
$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$
0	$\frac{1}{21}$	$\frac{1}{21}$	$\frac{1}{21}$	0

- Pálda átlag szűrésre:

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0

0	0	0	0	0	0	0	0	0	0
0	0	10	20	30	30	30	20	10	0
0	0	20	40	60	60	60	40	20	0
0	0	30	60	90	90	90	60	30	0
0	0	30	50	80	80	80	60	30	0
0	0	30	50	80	80	80	60	30	0
0	0	20	30	50	50	60	40	20	0
0	10	20	30	30	30	30	20	10	0
0	10	10	10	0	0	0	0	0	0
0	10	10	10	0	0	0	0	0	0

$$f \quad h = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$g=f^*h$$

- Az átlag-szűrő hatása és tulajdonságai

- a képpontok felveszik a környezetük átlagát
- a szűrt kép intenzitásértékei a kiindulási kép intenzitástartományában maradnak
- lineáris operátor (mivel a is konvolúció az)
- haszna: csökkenti a zajt
- kára: gyengíti az éleket, homályossá teszi a képet

- Szűrés a környezet súlyozott átlagával

- Átlagolás: a környezetbe eső valamennyi pont intenzitása egyforma súlyval esik a latba.

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

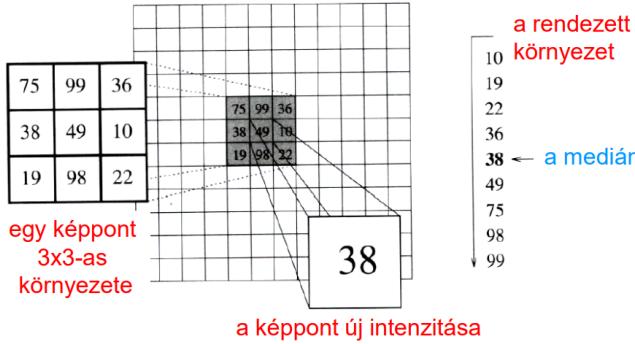
- Súlyozott átlag: a környezet intenzitásaihoz (általában a távolsággal arányosan csökkenő) súlyokat rendelünk

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

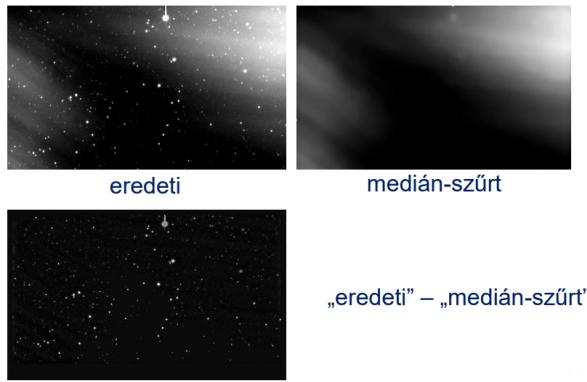
9.1.2. Medián szűrés

- Az [a_1, a_2, ..., a_{2n+1}] (páratlan elemszámú) szám-tömb mediánja a nagyság szerint rendezett tömb középső, (n+1)-dik eleme

- Az átlagoló szűréshez hasonlóan zajszűrésre használatos, viszont jobban megőrzi a fontosabb részleteket
- Ennél a szűrőnél is egy meghatározott méretű környezet van figyelembe véve, de itt nem a szomszédos pixelek átlagával, hanem a mediánjával helyettesíti az egyes pixeleket
- Illusztrálva:



- Alkalmazása:

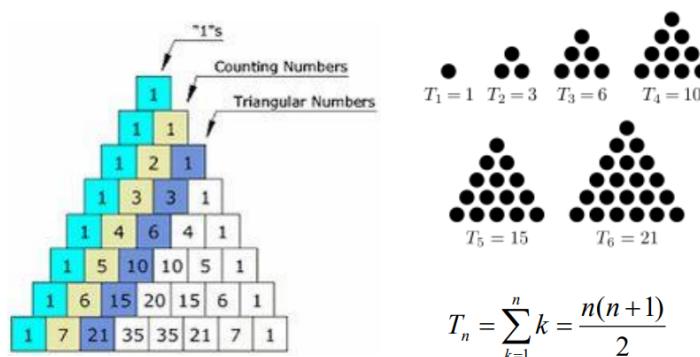


- A medián-szűrés hatása

- Megszünteti az egyedi (és a „kis” kiterjedésű) kiugrásokat
- „Jobban” megőrzi az éleket, mint az átlagolás
- „Nagy” kiterjedésű zajfoltoknál jel-elnyomó.

9.1.3. Gauss simítás

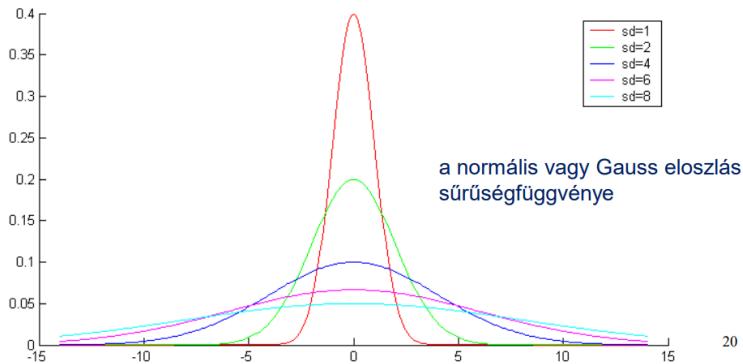
Pascal háromszög



A háromszögszámok azok a számok, amelyek előállnak az első valahány egymást követő természetes szám összegeként.

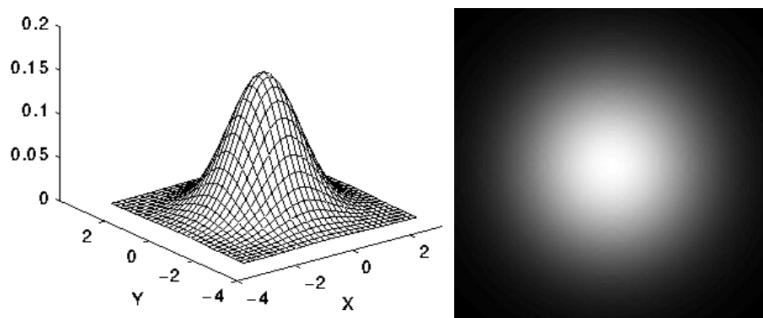
- Szintén zajszűrésre használatos
- A Gauss simítás alkalmazása egy képre nem más, mint konvolválni a képet a Gauss függvénnnyel
 - A maszk egy ("harang alakú") Gauss görbét fog reprezentálni
- 1 dimenziós Gauss függvény
 - σ a szórást jelöli

$$G_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{(-x^2/2\sigma^2)} \quad (\sigma > 0)$$



- 2 dimenziós Gauss függvény

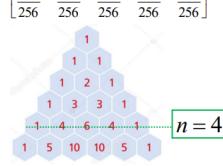
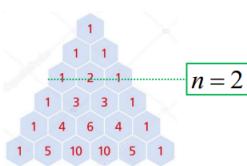
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



- Diszkrét közelítése a Pascal háromszög segítségével (attól függ hogy melyik szintjéből kell kiindulnunk, hogy mekkora maszkot akarunk)

$$\begin{bmatrix} \frac{1}{4} \\ \frac{2}{4} \\ \frac{1}{4} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \end{bmatrix} = \begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{16} \\ \frac{4}{16} \\ \frac{6}{16} \\ \frac{4}{16} \\ \frac{1}{16} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{16} & \frac{4}{16} & \frac{6}{16} & \frac{4}{16} & \frac{1}{16} \end{bmatrix} = \begin{bmatrix} \frac{1}{256} & \frac{4}{256} & \frac{16}{256} & \frac{4}{256} & \frac{1}{256} \\ \frac{4}{256} & \frac{16}{256} & \frac{24}{256} & \frac{16}{256} & \frac{4}{256} \\ \frac{16}{256} & \frac{24}{256} & \frac{36}{256} & \frac{24}{256} & \frac{16}{256} \\ \frac{4}{256} & \frac{16}{256} & \frac{24}{256} & \frac{16}{256} & \frac{4}{256} \\ \frac{1}{256} & \frac{4}{256} & \frac{16}{256} & \frac{4}{256} & \frac{1}{256} \end{bmatrix} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

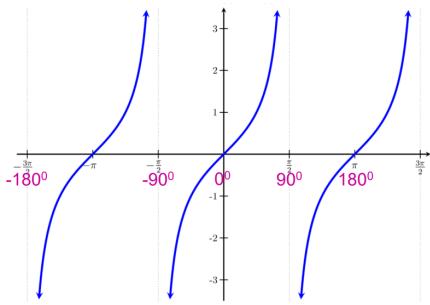


- Példa Gauss szűrésre 3x3-as maszkkkal:



9.1.4. Éldetektálás

Tangens függvény



Első rendű derivált

Geometriai jelentése: az érintő iránytangense.

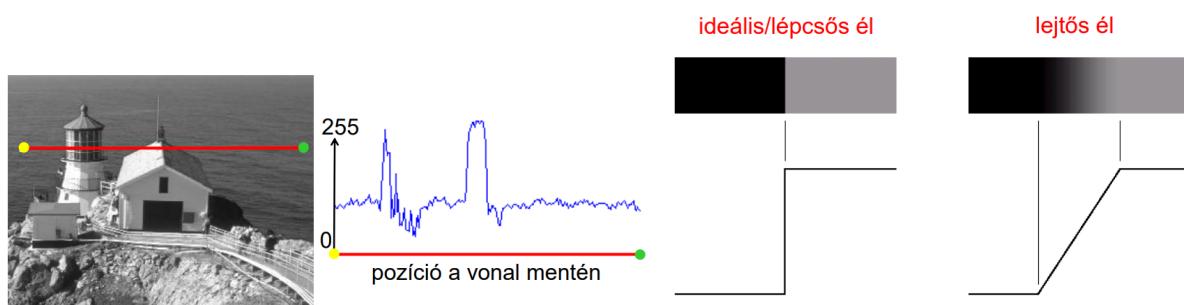
Elárulja, hogy a függvény hol nő és hol csökken és hogy milyen mértékben.

A derivált (meredekség):

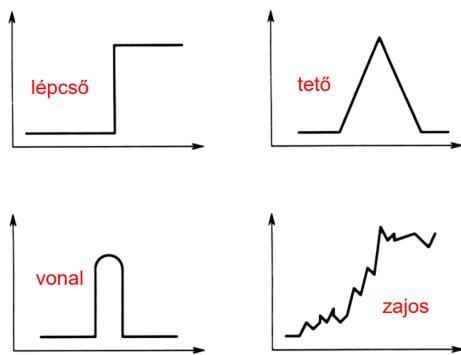
- pozitív, ha a függvény nő,
- negatív, ha csökken

Éldetektálás Gradiens operátorokkal

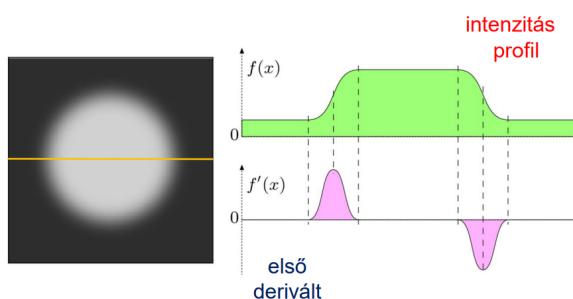
- A képen ott található él, ahol a képfüggvény valamely irány mentén hirtelen változik.



- Tipikus élprofilok



- Az első deriváltat felhasználhatjuk éldetektálásra: ahol kiemelkedőbb lokális maximuma (vagy minimuma) van az első deriváltnak, ott jó esélyel él található. A lokális minimumok miatt abszolútértéket szokás venni, így csak a maximumokra kell odafigyelni



- 2 dimenziós képnél parciális derivált használata: változások detektálása az x és y koordináta mentén

- a két érték alapján tudjuk hogy hol vannak élek



f

$\frac{\partial f}{\partial x}$

$\frac{\partial f}{\partial y}$

- Gradiens nagysága:

$$S = \|\nabla I\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

- Gradiens iránya:

$$\alpha(\nabla f(x, y)) = \arctan \begin{pmatrix} \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial x} \end{pmatrix}$$

- Diszkrét gradiens operátorok:

- Roberts operátor

- a maszkelemek összege 0

- könnyen számítható, de zajérzékeny

$$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Prewitt operátor

- a maszkelemek összege 0

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Sobel operátor

- a maszkelemek összege 0

- simító hatással bír

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Frei-Chen (izotropikus) operátor

- a maszkelemek összege 0

$$\begin{bmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix} \quad \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Gradiens maszk tervezése (x-irányban)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- Feltételek:

1. Szimmetria: $a_{1j} = a_{3j}$ ($j=1,2,3$) 2. Antiszimmetria: $a_{i1} = -a_{i3}$, $a_{i1} > 0$ és $a_{i2} = 0$ ($i=1,2,3$) 3. Nem reagál konstans régióra: $\sum_{i=1}^3 \sum_{j=1}^3$

- 8-irányban élt kereső gradiens operátorok

- Prewitt compass operátor

$$\begin{bmatrix} 1 & 1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 1 \\ -1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

E NE N NW

$$\begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{bmatrix}$$

W SW S SE

- Robinson-3 compass operátor

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

E NE N NW

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

W SW S SE

- Robinson-5 compass operátor

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

E NE N NW

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

W SW S SE

- Kirsch compass operátor

$$\begin{bmatrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{bmatrix} \quad \begin{bmatrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{bmatrix} \quad \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{bmatrix} \quad \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{bmatrix}$$

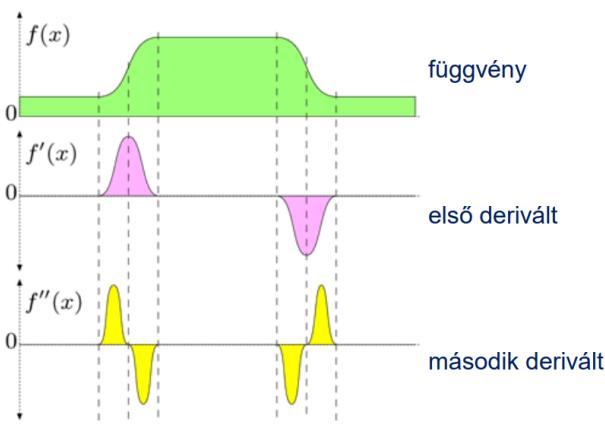
E NE N NW

$$\begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{bmatrix} \quad \begin{bmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{bmatrix} \quad \begin{bmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix} \quad \begin{bmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}$$

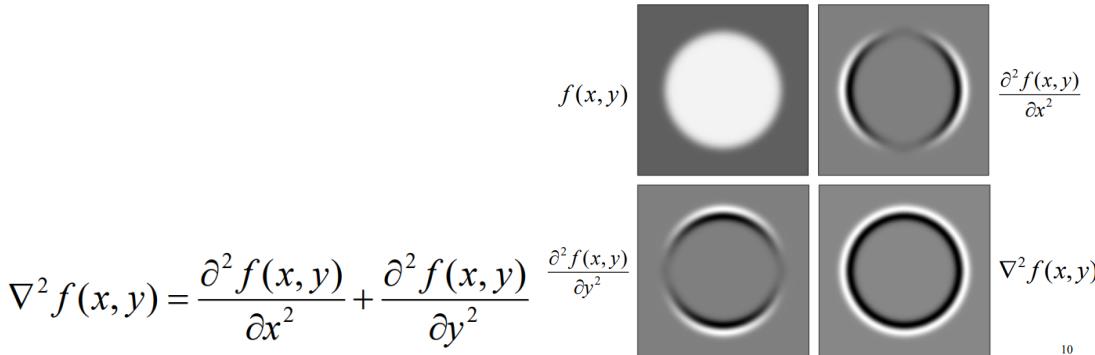
W SW S SE

Laplace éldetektálás

- Másodrendű derivált: az első rendű deriváltal szemben a nullán való áthaladás helyén lesz az él, nem a lokális maximumnál vagy minimumnál



- Kétváltozós függvény Laplace operátora: az x szerinti és y szerinti másodrendű deriváltak összege (a képen látható ahogyan az összegből tényleg megkapjuk az összes élt)



- A Laplace operátor egy lineáris differenciál-operátor a másodrendű derivált közelítésére (a gradiens operátor önmagával vett belső szorzata)

- Tulajdonságai:

- forgásinvariáns
- egyetlen maszkkal számítható
- csak a magnitúdó számítható
- duplán érzékelhet éleket
- zajérzékeny

$$\nabla^2 = \nabla \cdot \nabla = \left[\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_n} \right] \cdot \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{bmatrix} = \sum_{i=1}^n \frac{\partial^2}{\partial x_i^2}$$

- Egy diszkrét Laplace operátor (A maszkelemek összege 0):

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$\approx \frac{\partial^2 f(x,y)}{\partial x^2} \quad \approx \frac{\partial^2 f(x,y)}{\partial y^2} \quad \approx \nabla^2 f(x,y)$$

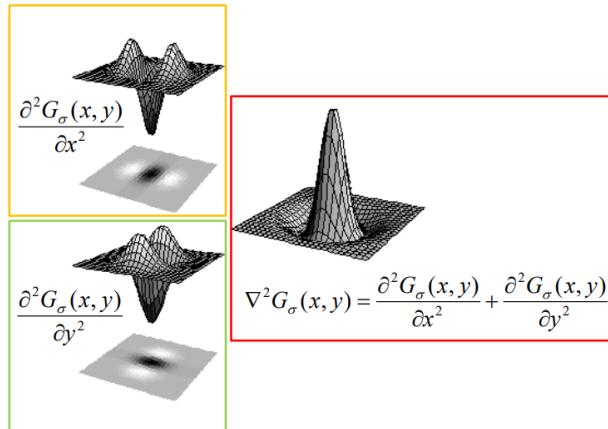
- A másodrendű derivált érzékeny a zajra -> hajtunk végre először Gauss simítást a képen
- Az Laplace operátor és a Gauss operátor is lineáris -> megspórolhatunk egy konvolúciót azzal, ha a Gauss operátoron alkalmazzuk a Laplace transzformációt, amiből egy új konvolúciós maszkot kapunk

$$\nabla^2(G * f) = (\nabla^2 G) * f$$

↑ simítás ↑ (egyetlen) konvolúció
a LoG függvényvel

a simított függvény Laplace transzformáltja

- A Gauss függvény Laplace transzformáltja (LoG – Laplacian of Gaussian)
 - "fordított sombrero"



- A LoG egy diszkrét közelítése:

$$\nabla^2 G \approx \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

Marr-Hildreth éldetektor

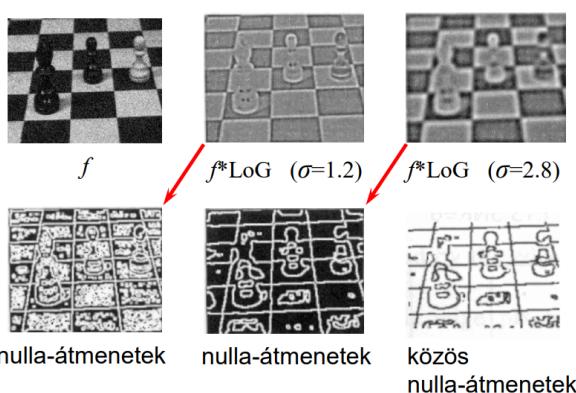
Lényege:

1. Konvolváljuk a képet egy (vagy több) alkalmas LoG függvényel.
 2. Keressünk (közös) nulla-átmeneteket.

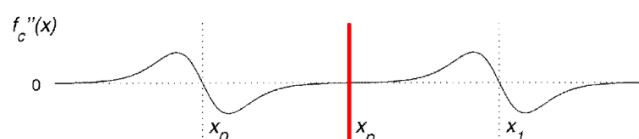
(Nulla-átmenet ott van, ahol az adott pont egy „kis” (pl. 2x2-es vagy 3x3-as) környezetében pozitív és negatív értékek is előfordulnak.)

- Példa:

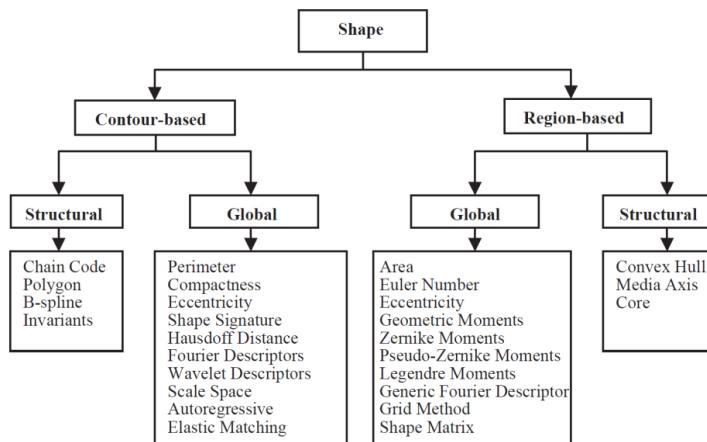
- σ a szórást jelöli



- Nagyon lapos nulla átmeneteknél "fantom" élt is detektálhat

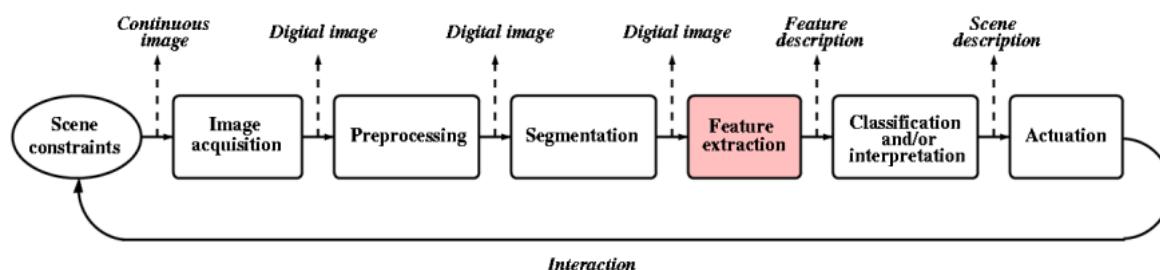


9.2. 2. Alakreprezentáció, határ- és régió-alapú alakleíró jellemzők, Fourier leírás.



Alakzat: pontok összefüggő rendszere

A moduláris gépi látás általános modellje:



- Az alakreprezentáció módszerei:

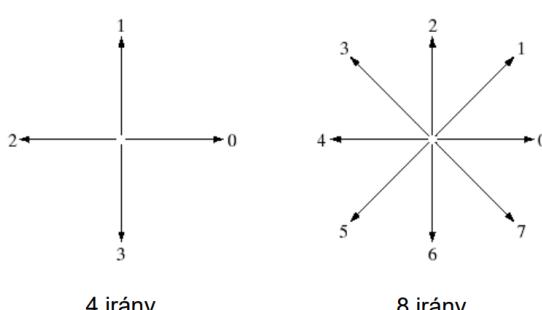
- az objektumot körülvevő **határ** leírása
- az objektum által elfoglalt **régió** leírása
- transzformációs** megközelítés

9.2.1. Határvonal alapú tulajdonságok

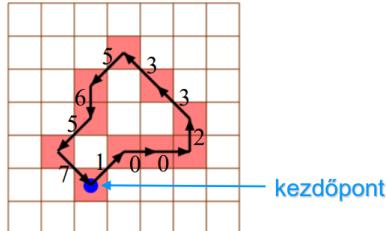
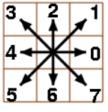
- lánckód, alakleíró szám
- kerület, terület, kompaktság, cirkularitás,
- közelítés poligonnal,
- parametrikus kontúr, határvonal leíró függvény,
- meredekségi hisztorogram,
- görbület, energia
- strukturális leírás

Lánckód (chain code)

- Az alakzat határpontjait követi, láncolja az óramutató járásával ellentétes irányban.
- Hatópont:** az alakzatnak olyan pontja, melynek van az alakzathoz nem tartozó 8-, ill. 4-szomszédja (4, ill. 8 esetén).
- Az elemi elmozdulások kódjai:



- Példa 8-as lánckódra:



lánckód: 1 0 0 2 3 3 5 6 5 7

Normalizált lánckód

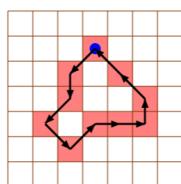
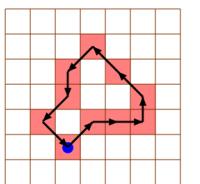
Különböző kezdőpontok választása a lánckód ciklikusan permutált változatait eredményezi.

Különbsékgód: a lánckód első deriváltja, a szomszédos elemek közötti elmozdulások száma.

Normalizálás: addig permutáljuk ciklikusan a különbsékgódot, amíg a legkisebb értékű kódot (a legkisebb 4-es, ill. 8-as számrendszerbeli számot) kapjuk.

Alakleíró szám: a normalizált különbsékgód (nem függ a kezdőpont választásától).

Normalizálás:



lánckód: 1002335657
különbsékgód: 7021021722



alakleíró szám: 0210217227

lánckód: 5657100233
különbsékgód: 1722702102



alakleíró szám: 0210217227

- Az első ábránál (lánckód: 1002335657):

- Különbsékgód:

- ha a 8-irányos iránytűn az óramutató járásával ellentétesen haladunk, akkor 1-től 0-ig 7 lépésekben tudunk eljutni (a lánckód első két elembéből számoljuk a különbsékgód első elemét)
 - második számjegye a 0 és 0 közti távolság, vagyis 0
 - harmadik számjegye 0 és 2 távolsága, ami 2 (még minden az iránytűn, óramutató járásának ellentétesen)
 - ez így megy végig, utolsó szám az elsőhöz lesz hasonlítva

- Alakleíró szám: úgy kell rendezni a különbsékgódot hogy a lehető legkisebb számot adják

- mivel a 2 alakzat megegyezik, ezért minden esetben ugyanaz lesz az alakleíró szám
 - invariáns a forgatásra is, ha a forgatási szög $k \cdot \pi/2$

A lánckód tulajdonságai

- Előnyök (a mátrixos reprezentációval szemben):

- kompakt (tömör),
 - eltolás-invariáns,
 - gyors algoritmus,
 - gyorsan rekonstruálható belőle az alakzat

- Hátrányok:

- nem forgás-invariáns,
 - nem skála-invariáns
 - a pontosság legfeljebb pixelnyi lehet,
 - érzékeny a zajra

Kerület számítása 8-as lánckódból

lánckód: 1 0 0 2 3 3 5 6 5 7

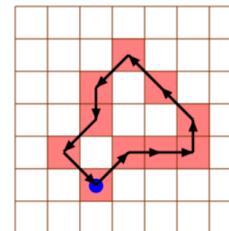
rend (a lánckód hossza): 10

páros elemek száma: 4

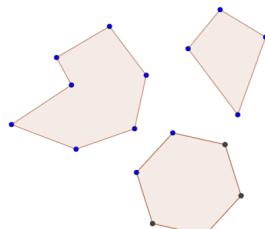
páratlan elemek száma: 6

kerület: 12.485

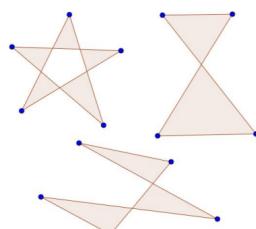
$$\text{kerület} = 1 \cdot (\text{páros elemek száma}) + \sqrt{2} \cdot (\text{páratlan elemek száma})$$



Poligonok

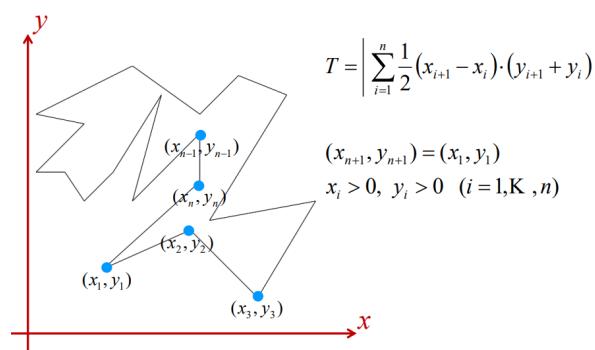


egyszerű
(önmagát nem
metsző)

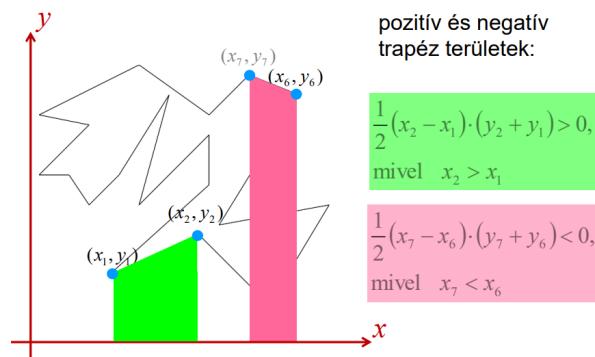


nem-egyszerű

Egyszerű poligon területe:

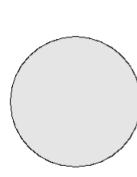


- trapéz területek számítása (negatívak és pozitívak)

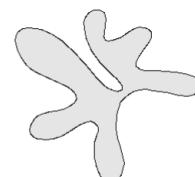


Kompaktság: kompaktság = (kerület)^2 / terület

- Pl:



erősen kompakt

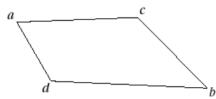
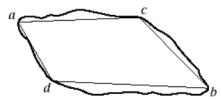
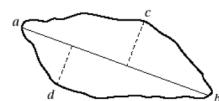


gyengén kompakt

Cirkularitás (körszerűség): cirkularitás = 1 / kompaktság = terület / (kerület)^2

- maximális a körre: 1/(4π) ≈ 0.08

Közeliítés poligonnal



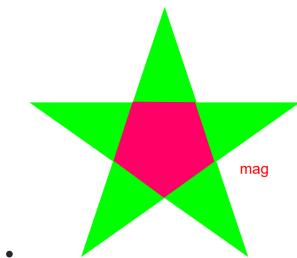
a kezdeti közelítést adó négyszög konstruálása

Leírás egy változós függvényekkel (signatures)

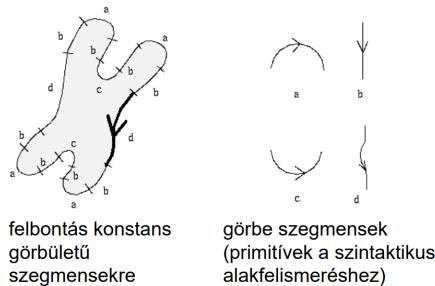
- Pl. a súlypontnak a határtól vett távolságát a szög függvényében fejezi ki.
- Függ az alakzat méretétől és a határon vett kezdőpont megválasztásától, ezért a jellemző normalizálásra szorul.

Csillag-szerű objektum

- Van olyan pontja, amelyből induló tetszőleges irányú sugár a határt egyetlen pontban metszi.
- Az ilyen pont a csillag-szerű objektum magjához tartozik.



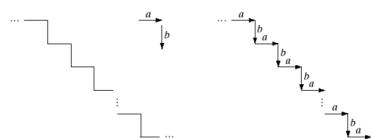
Strukturális leírás



Leírás generatív nyelvtanokkal

Pl:

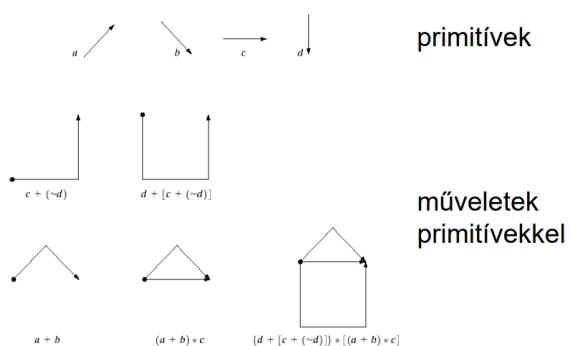
lépcső és kódolt szerkezete:



a CF-nyelvtan:

$$G = (\{S, A\}, \{a, b\}, \{S \rightarrow aA, A \rightarrow bS, A \rightarrow b\}, S)$$

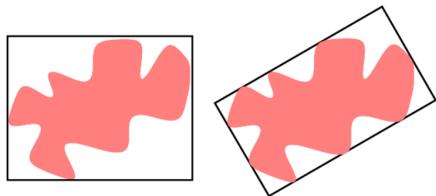
Leírás összefűzött primitívekkel



9.2.2. Régió alapú alakleírás

- befoglaló téglalap, rektangularitás
- főtengely, melléktengely, átmérő, excentricitás, főtengely szöge
- konvex burok, konvex kiegészítés, konkávitási fa, partícionált határ,
- vetületek, törés-költség
- topológiai leírások, Euler-szám, szomszédsági fa,
- váz,
- momentumok, invariáns momentumok

Befoglaló téglalap



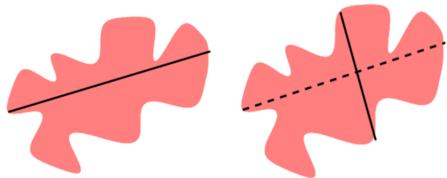
„álló” befoglaló téglalap minimális befoglaló téglalap

Rektangularitás (téglalap-szerűség)

- az alakzat területének és a minimális befoglaló téglalap területének a hányszáma

Fő- és melléktengely

- főtengely: az alakzon belül haladó leghosszabb egyenes szakasz
- melléktengely: az alakzon belüli, a főtengelyre merőleges leghosszabb egyenes szakasz



Átmérő

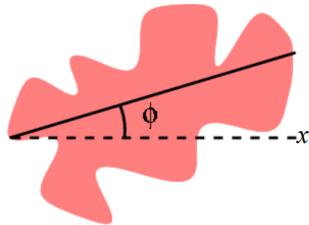
$$Diam(S) = \max_{p,q \in \partial S} \{ d(p,q) \},$$

ahol :

- S : alakzat,
- ∂S : alakzat határa,
- d : távolság.

Excentricitás: a fő- és a melléktengely hosszaránya (főtengely/melléktengely)

Főtengely szöge (az alakzat irányába): a főtengely és az x-tengely által bezárt szög

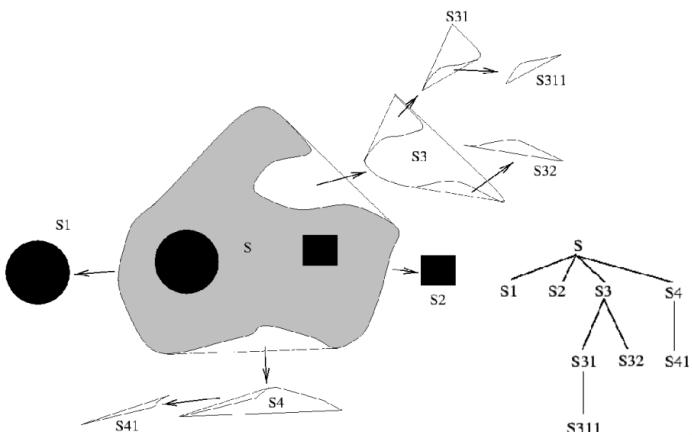


Konvex burok: az alakzatot tartalmazó minimális konvex alakzat

Konvex kiegészítés: a konvex burok és az alakzat különbsége

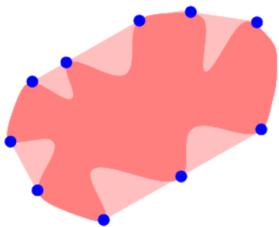
Konkávitási fa:

- A fa gyökere a kiindulási alakzat, az első szinten a konvex különbség alakzatai helyezkednek el, melyekre a faépítést rekúrziív módon folytatjuk.
- A fa elágazási pontjaiban lévő alakzatok nem konvexek, még minden levélalakzat konvex.

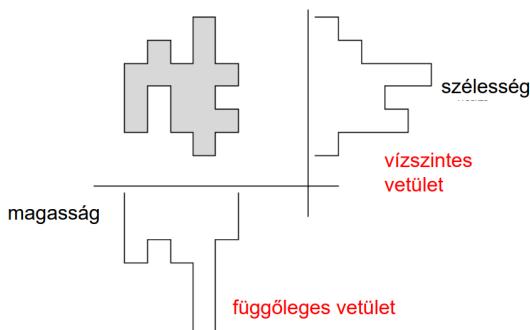


Partitionált határ

- a határ partícionálható az szerint, hogy hol kezdődik, ill. fejeződik be a konvex kiegészítés valamely komponense



Vetületek



Törés költség (break cost): A törés költség meghatározásánál az egyik vetületet (pl. a függőlegeset) vesszük figyelembe. Ekkor egy oszlop költsége a benne található olyan egyesek száma, melyekhez az előző oszlop ugyanazon sorában is egyes található.

Pl.:

1	0	1	0	0	0	1	0	1	0
1	1	1	0	0	0	1	0	0	1
1	0	1	0	0	0	1	0	1	0
1	1	1	0	0	1	1	0	0	1
0	0	1	0	1	1	1	0	1	0
1	0	0	1	0	0	1	1	0	1
0	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	0	0	1
1	0	1	1	0	0	1	0	1	0
1	1	1	0	1	1	1	1	0	1

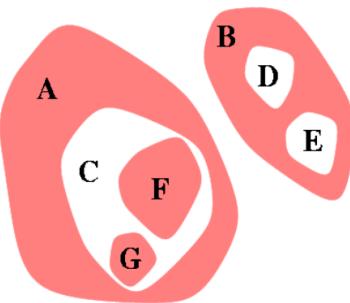
bináris kép

0 4 3 1 0 2 4 3 1 0 törés költség

Topológiai leírás

- Bináris kép:** kétféle érték (1: fekete, alakzat, komponens; 0: fehér, lyuk, háttér)
- Komponens:** maximálisan összefüggő fekete halmaz (bármely két pontja összeköthető a halazon belüli 4-, ill. 8 úttal) régió
- Üreg:** a komplementens/negált kép egy véges komponense

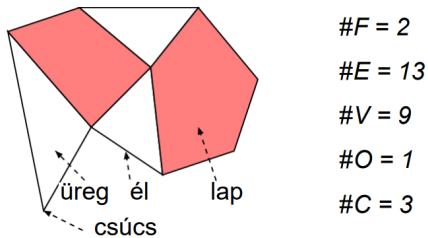
Euler-féle szám: (komponensek száma) - (üregek száma)



komponensek:
A, B, F, G;
üregek:
C, D, E;
Euler-szám:
 $4 - 3 = 1$

Euler szám poligonhálózatokra

- lapok (face) száma: #F
- élek (edge) száma: #E
- csúcsok (vertex) száma: #V
- régiók, objektumok száma: #O
- üregek (cavity) száma: #C
- Euler szám: $#F - #E + #V = #O - #C$

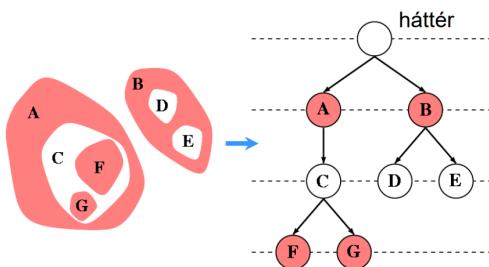


$#F = 2$
 $#E = 13$
 $#V = 9$
 $#O = 1$
 $#C = 3$

$$\#F - \#E + \#V = 2 - 13 + 9 = \#O - \#C = 1 - 3 = -2$$

Összefüggőségi fa

- A bináris képekhez rendelt irányított gráf, ahol:
 - minden egyes szögpontról megfelelő a kép egy (fehér vagy fekete) komponensének
 - a gráf tartalmazza az (X,Y) életet, ha az X komponens „körülveszi” a vele szomszédos Y komponensem.



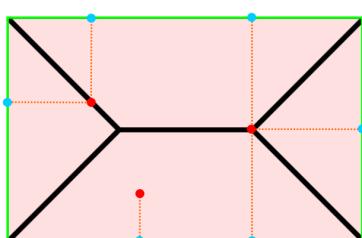
Az Euler-szám és az összefüggőségi fa

- Nem kódolják a képeket, mivel számos képnél megegyezik az Euler száma és/vagy az összefüggőségi fája.

Váz (skeleton): A váz egy gyakran alkalmazott régió-alapú alakleíró jellemző, mely leírja az objektumok általános formáját.

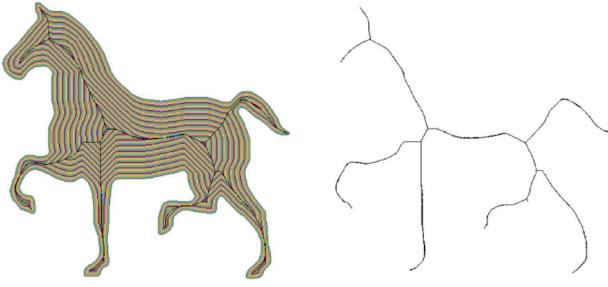
- A váz meghatározásai:

- A váz a középtengely transzformáció (Medial Axis Transform, MAT) eredménye: a vázat az objektum azon pontjai alkotják, melyekre kettő vagy több legközelebbi határpont található.



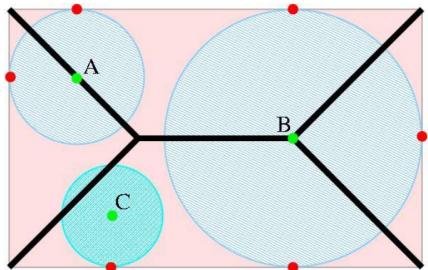
a 2D téglalap **határa** és váza;
objektumpont és **legközelebbi határpontja(i)**

ii. Préritűz-hasonlat: Az objektum határát (minden pontjában) egyidejűleg felgyűjtjük. A váz azokból áll, ahol a tűzfrontok találkoznak és kioltják egymást. (Feltételezzük, hogy a tűzfrontok minden irányban egyenletes sebességgel, vagyis izotropikusan terjednek.)



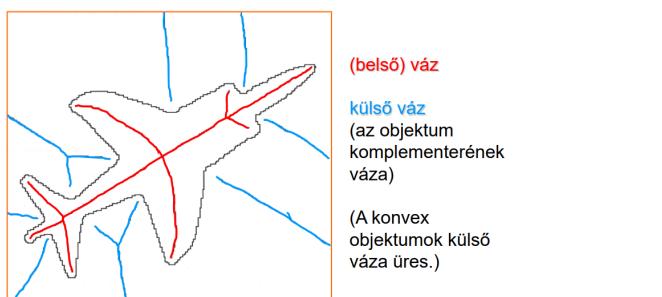
a front-terjedés (diszkrét) modellezése

iii. A vázat az objektumba beírható maximális (nyílt) hipergömbök középpontjai alkotják. Egy beírható hipergömb maximális, ha öt nem tartalmazza egyetlen másik beírható hipergömb sem. A beírható maximális (nyílt) hipergömbök egyesítése a kiindulási objektum egy lefedőrendszerét adja.



beírt körlapok, középpontjaik és az érintési pontok a határon

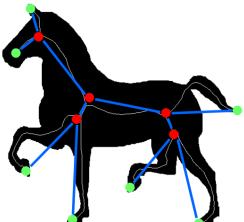
- Belső- és külső váz



- A váz:

- Reprezentálja az objektum
 - általános formáját
 - topológiai szerkezetét és a
 - lokális objektum szimmetriákat
- Invariáns
 - az eltolásra,
 - az elforgatásra és az
 - uniform skálázásra
- Egyszerűbb szerkezet („vékony”, csökkenti a dimenziót).

- Váz gráf:



- Az alakjellemzésben a momentumok előnye:

- számok,
- többszintű képekre is értelmezettek,
- invariánsak (a fontosabb geometriai transzformációkra).

Az I kép $(p+q)$ -adrendű momentumai

$$(p,q=0,1,2,\dots): m_{p,q} = \sum_x \sum_y x^p \cdot y^q \cdot I(x,y)$$

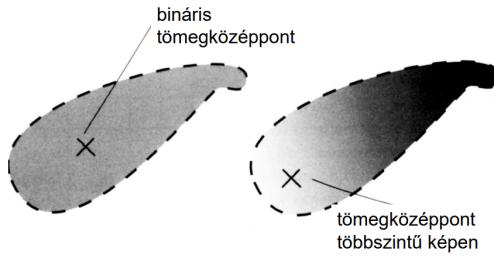
Az S bináris alakzat $(p+q)$ -adrendű momentumai:

$$m_{p,q} = \sum_{(x,y) \in S} x^p \cdot y^q$$

Súlypont:

$$(\bar{x}, \bar{y}) = \left(\frac{m_{1,0}}{m_{0,0}}, \frac{m_{0,1}}{m_{0,0}} \right)$$

Súlypont



Centrális momentumok

Az I kép $(p+q)$ -adrendű centrális momentumai:

$$\mu_{p,q} = \sum_x \sum_y (x - \bar{x})^p \cdot (y - \bar{y})^q \cdot I(x,y)$$

Az S bináris alakzat $(p+q)$ -adrendű centrális momentumai:

$$\mu_{p,q} = \sum_{(x,y) \in S} (x - \bar{x})^p \cdot (y - \bar{y})^q$$

Excentricitás

$$\varepsilon = \frac{(\mu_{2,0} - \mu_{0,2})^2 + 4\mu_{1,1}^2}{(\mu_{2,0} + \mu_{0,2})^2}$$

0 és 1 közötti érték.

Főtengely szöge

$$\Phi = \frac{1}{2} \arctan \left(\frac{2\mu_{1,1}}{\mu_{2,0} - \mu_{0,2}} \right)$$

Normalizált centrális momentumok

$$\eta_{p,q} = \frac{\mu_{p,q}}{\mu_{p,q}^\gamma}$$

$$\gamma = 1 + (p + q)/2$$

Invariáns momentumok és a geometriai transzformációk

Moment Invariant	Original Image	Translated	Half Size	Mirrored	Rotated 45°	Rotated 90°
ϕ_1	2.8662	2.8662	2.8664	2.8662	2.8661	2.8662
ϕ_2	7.1265	7.1265	7.1257	7.1265	7.1266	7.1265
ϕ_3	10.4109	10.4109	10.4047	10.4109	10.4115	10.4109
ϕ_4	10.3742	10.3742	10.3719	10.3742	10.3742	10.3742
ϕ_5	21.3674	21.3674	21.3924	21.3674	21.3663	21.3674
ϕ_6	13.9417	13.9417	13.9383	13.9417	13.9417	13.9417
ϕ_7	-20.7809	-20.7809	-20.7724	20.7809	-20.7813	-20.7809



Transzformáció alapuló alakleírás

Transzformáljuk a határ K darab mintavételezett pontjából (mint komplex s(k) számokból) képzett s vektort. Az eredményül kapott a vektor (komplex a(k) együtthatók) adják a Fourier leírást. Az alakzat rekonstrukciójához az inverz Fourier-transzformációt kell végrehajtani.

A határpontok Fourier-transzformáltja:

$$a(j) = \frac{1}{K} \sum_{k=0}^{K-1} s(k) \cdot e^{-2\pi j k / K} \quad (j = 0, 1, \dots, K-1)$$

Az együtthatók inverz Fourier-transzformáltja:

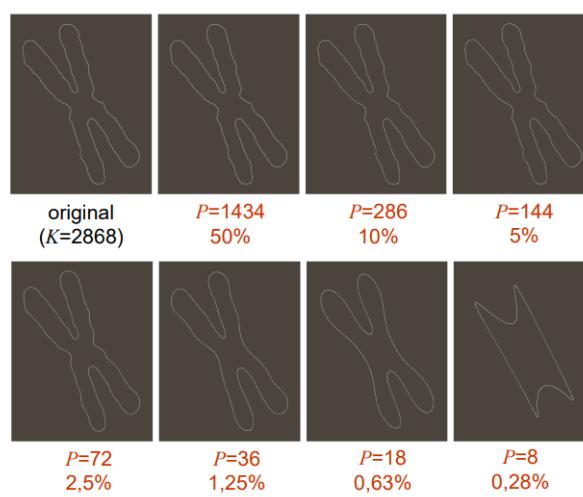
$$s(k) = \sum_{j=0}^{P-1} a(j) \cdot e^{2\pi j k / K} \quad (k = 0, 1, \dots, K-1)$$

Reprezentáció $P \leq K$ darab együttható alapján:

$$\hat{s}(k) = \sum_{j=0}^{P-1} a(j) \cdot e^{2\pi j k / K} \quad (k = 0, 1, \dots, K-1)$$

Példa a Fourier leírásra:

- minél kevesebb Fourier deszkriptort használunk, annál kevésbé lesz pontos az inverz Fourier transzformációból előállított rekonstrukció



10. Programozási nyelvek

10.1. A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai.

10.1.1. Nyelvcsoportok (paradigmák)

- Imperatív, procedurális (pl.: C, C++, Pascal)
- Objektum orientált (pl.: C++, Java, Smalltalk)
- Applikatív, funkcionális (pl.: Haskell, ML)
- Szabály alapú, logikai (pl.: Prolog, HASL)
- Párhuzamos (pl.: Occam, PVM, MPI)

10.1.2. Imperatív programozás

- Az imperatív programozás olyan programozási paradigma, amely utasításokat használ, hogy egy program állapotát megváltoztassa.

- A kifejezést gyakran használják a deklaratív programozással ellentétben, amely arra összpontosít, hogy a program *mit* érjen el, anélkül, hogy meghatározná, hogy a program *hogyan* érje el az eredményt.
- Azok a nyelvek, melyek az imperatív paradigmákba esnek két fő jellemzőjük van: meghatározzák a műveletek sorrendjét olyan konstrukciókkal, amelyek kifejezetten ellenőrzik ezt a sorrendet, és lehetővé tesznek olyan mellékhatásokat, amelyben az állapot módosítható egy időben, egy kód egységen, majd később egy másik időpontban olvasható egy másik kód egységén belül.
- A legkorábbi imperatív nyelvek az eredeti számítógépek gényelvével voltak (assembly)
 - egyszerű utasítások -> könnyebb hardwares megvalósítás, de az összetett programok létrehozása nehezebb

Procedurális programozás

- A megoldandó programozási feladatot kisebb egységekből, vagy eljárásokból (angolul: procedure) építi fel
- Ezek az eljárások a programnyelv kódjában általában jól körülhatárolt egységek (függvény, rutin, szubrutin, metódus – az elnevezés az adott programozási nyelvtől függ), amelyeknek van elnevezésük és jellemzők őket paraméterek és a visszatérési értékük.
- A programok futtatása során gyakorlatilag a főprogramból ezek az eljárások kerülnek sorozatosan meghívásra. Meghíváskor meghatározott paraméterek átadására kerül sor, az eljárás pedig a benne meghatározott logika eredményeként általában valamilyen visszatérési értéket ad vissza, aminek függvényében a főprogram további eljárásokat végezhet.
- Az objektum orientált paradigmával szemben itt háttérbe szorulnak a komplex adatszerkezetek
- Moduláris tervezés
 - Dekompozíció: adott feladat több egyszerűbb részfeladatra bontása
 - Kompozíció: meglévő programegységek újrafelhasználása
 - Érthetőség: a modulok önmagukba is egy értelmes egységet alkossanak
 - Folytonosság: a specifikáció kis változása esetén is csak kis változás legyen szükséges a programban
 - Védelem: egy hiba csak egy (vagy maximum egy pár), modul működésére legyen hatással, ezzel védve a program egészét
- Modularitás alapelvei:
 - Nyelvi támogatás: a modulok külön-külön legyenek lefordíthatók
 - Kevés kapcsolat: a modulok keveset kommunikálnak egymással
 - Gyenge kapcsolat: ha két modulnak kommunikálnia kell egymással, akkor csak annyi információt cseréljenek, amennyi szükséges
 - Explicit interfések: ha két modul kommunikál, akkor legalább az egyikük szövegéből ki kell hogy derüljön
 - Információ-elrejtés: egy modulnak csak az explicit módon nyilvánossá tett információit használhatjuk fel
 - Nyitott és zárt modulok
 - Zárt modul: csak változatlan formában kerülhet felhasználásra
 - Nyitott modul: kiterjeszthető, más szóval bővíthető az általa nyújtott szolgáltatások száma
 - Újrafelhasználhatóság: ugyanazokat a programeleket ne kelljen többször elkészíteni, ügyeljünk viszonylag általanosítható modulok készítésére
 - Típus változatossága: modulok működjenek többféle típusra
 - Adatszerkezetek és algoritmusok változatossága: például egy lineáris kereső eljárás működjön több féle adatszerkezetre (ezeken belül persze más-más algoritmusokkal)
 - Egy típus - egy modul: egy típus műveletei kerüljenek egy modulba
 - Reprezentáció függetlenség: egy adattípus reprezentációjának a megváltozása ne okozzon modulon kívüli változást
- Procedurális programozási nyelvek például a *C*, *Pascal*, *FORTRAN*

Objektum orientált programozás

- Az objektum orientált programozás az objektumok fogalmán alapuló programozási paradigma
- Az objektumok egysége foglalják az adatokat és a hozzájuk tartozó műveleteket (egysége záras).
- A program egymással kommunikáló objektumok összességéből áll.
- A legtöbb objektumorientált nyelv osztály alapú, azaz az objektumok osztályok példányai, és típusuk az osztály.
- Objektumok és osztályok
 - Osztályok:
 - Az adatformátum és az elérhető metódusok definíciója az adott típus vagy a típushoz tartozó objektumok számára.

- Tartalmazhatnak adattagokat és metódusokat, amelyek műveleteket végeznek az osztály adattagjain.
- Összetarto adatok és függvények, eljárások egysége.

- Objektumok:
 - Az osztály példányai.
 - Gyakran megfeleltethetők a való élet objektumainak vagy egyedeinek.
- Pár fontos fogalom:
 - Osztályváltozók: az osztályhoz tartoznak, elérhetők az osztályon, de példányokon keresztül is. minden példány számára ugyanaz.
 - Attribútumok: az egyedi objektumok jellemzői, minden objektumnak sajátja van.
 - Tagváltozók: az osztály- és a példányváltozók együttese
 - Osztálymetódusok: osztály szintű metódusok, csak az osztályváltozókhoz és paramétereikhez férhetnek hozzá, példányváltozókhöz nem.
 - Példánymetódusok: példány szintű metódusok, hozzáférnek az adott példány összes adatához és metódusához, és paramétereik is lehetnek.
- Kompozíció, öröklődés, interfészek
 - Kompozíció: az objektumok lehetnek más objektumok mezői
 - Öröklődés:
 - Osztályok közötti alarendeltségi viszony, majdnem minden osztály alapú nyelv támogatja
 - Ha az A osztályból öröklődik a B osztály, akkor B egyben az A osztály példánya is lesz, ezért megakpja az A osztály összes adattaját és metódusát
 - Több programozási nyelv megengedi a többszörös öröklődést
 - Egyes nyelvekben, mint a Java és a C# megtíltható a leszármazás egyes osztályokból (Javában final, C#-ban sealed a kulcsszó)
 - Interfészek:
 - Nem tartalmazhatnak megvalósítási részleteket, csak előírhatják bizonyos metódusok jelenlétéét, illetve konstansokat definiálhatnak.
 - Olyan nyelvekben, ahol nincs a megvalósítások többszörös öröklődése, interfészkekkel érhető el a többszörös öröklés korlátozott formája
- Objektum orientált nyelvek például a *Java*, *C#*, *Python*, *Smalltalk*

10.1.3. Dekleratív programozás

- A specifikáción van a hangsúly, funkcionális esetben a program egy függvény kiszámítása, logikai esetben a megoldás megkeresését a futtató környezetre bízzuk
- Azok a nyelvek, amely ezt a programozást használják, megpróbálják minimalizálni vagy kiküszöbölni a mellékhatásokat, úgy, hogy leírják, hogy a programnak mit kell elérnie a probléma tartományában, ahelyett, hogy a programozási nyelv primitívjeinek sorozataként írná le, hogyan kell azt megvalósítani
- Deklaratív nyelvek közé tartoznak az adatbázis-lekérdezési nyelvek (pl. SQL, XQuery), a reguláris kifejezések, a logikai programozás, a funkcionális programozás és a konfigurációkezelő rendszerek

Funkcionális (applikatív) programozás

- A funkcionális programnyelvek a programozási feladatot egy függvény kiértékelésének tekintik
- Ugyanannak a feladatnak a megoldására funkcionális nyelven írt programkód általában lényegesen rövidebb, olvashatóbb és könnyebben módosítható, mint az imperatív nyelven kódolt programszöveg, mivel nem léteznek benne változók
- A *rekurzió* a funkcionális programozás egyik fontos eszköze, az ismétlések és ciklusok helyett rekurziót alkalmazhatjuk.
- Rekurziós függvény:
 - Rekurzív hívás minden feltételvizsgálat mögött
 - Rekuzív függvényt két esetre kell felkészíteni
 - Bázis eset: nem kell újra meghívnia magát
 - Rekurzív eset: Meghívja magát újra
 - Biztosítani kell, hogy minden elérjük a bázis esetet
 - Rekurzió speciális esete: iteráció
- Alapjául a Church által kidolgozott lambda-kalkulus szolgál, a tisztán funkcionális nyelvek a matematikában megsokott függvényfogalmat valósítják meg.

- Az ilyen programozás során a megoldandó feladatnál az eredményhez vezető út nem is biztosan ismert, a program végrehajtásához csupán az eredmény pontos definíciója szükséges.
- Tisztán funkcionális programozás esetén tehát nincs állapot és nincs értékkedás.

- Funkcionális nyelvek például a *Haskell* és *Scala*

Logikai programozás

- A logikai program egy modellre vonatkozó állítások (*axiómák*) egy sorozata
- Az állítások a modell objektumainak tulajdonságait és kapcsolatait, szaknyelven *relációit* írják le
- Az állítások egy adott relációt meghatározó részhalmazát predikátumnak nevezik
 - A program futása minden esetben egy az állításokból következő tételek konstruktív bizonyítása, azaz a programnak feltett kérdés vagy más néven cél/megválaszolása
- Az első logikai programozási nyelv a Prolog volt
 - Egy Prolog program csak az adatokat és az összefüggéseket tartalmazza. Kérdések hatására a "programvégrehajtást" beépített következtető-rendszer végzi
 - Programozás Prologban:
 - Objektumok és azokon értelmezett relációk megadása
 - Kérdések megfogalmazása a relációkkal kapcsolatban
 - A programnak meg kell adnunk egy célfeliratot (célklózt), ezután a program ellenőrzi, hogy a célklóz a logikai (forrás)program logikai következményei között van-e
 - Gyakran használják mesterséges intelligencia-alkalmazások megvalósítására, illetve a számítógépes nyelvészeti eszközöként

10.1.4. Párhuzamos programozás

- Egyszerre több szálon történik a végrehajtás
- Végrehajtási szál: folyamat (process)
- Előnyei:
 - Termézeszetes kifejezésmód
 - Sebességnövekedés megfelelő hardver esetén
- Hátrányai
 - Bonyolultabb a szekvenciálisnál
- A párhuzamos programok alapvetően nem determinisztikusak
- Sokféle párhuzamos programozási modell van
- Közös problémák:
 - Adathozzáférés folyamatokból
 - Közös memória (shared memory)
 - Osztott memória (distributed memory) + kommunikáció
 - Folyamatok létrehozása, megszüntetése, kezelése
 - Folyamatok együttműködése (interakciója)
 - Független
 - Erőforrásokért versengő
- A párhuzamos program:
 - Sebességfüggő: a folyamatok relatív sebessége minden futáskor más lehet
 - Nemdeterminisztikus: ugyanarra az inputra különböző output
 - Holt pont (deadlock): kölcsönös egymásra várakozás
 - Éhezés (starvation): Nincs holt pont, egy folyamat mégsem jut hozzá az erőforrásokhoz
- Occam

- Imperatív, folyamatok saját memóriával rendelkeznek, üzenetküldéssel kommunikálnak
- Occam program részei:
 - Változók
 - Folyamatok
 - Elindul -> csinál valamit -> befejeződik (terminál)
 - Befejeződés helyett holt pontba is kerülhet, erre különös figyelmet kell fordítani
 - Elemi és összetett folyamato
 - Csatornák: két folyamat közötti adatátvitelre szolgál

11. Programozás 1 & 2

11.1. 1. Objektum orientált paradigma, és annak megvalósítása a JAVA és C++ nyelvekben. Az absztrakt adattípus, az osztály. AZ egységbe zárás, az információ elrejtés, az öröklődés, az újrafelhasználás, és a polimorfizmus. A polimorfizmus feloldásának módszere.

TODO

11.2. 2. Objektumok életciklusa, létrehozás, inicializálás, másolás, megszüntetés. Dinamikus, lokális, és statikus objektumok létrehozása. A statikus adattagok és metódusok, valamint szerepük a programozásban. Operáció, és operátor overloading JAVA és C++ nyelvekben. Kivételkezelés.

TODO

11.3. 3. JAVA és C++ programok fordítása és futtatása. Parancssori paraméterek, fordítási opciók, nagyobb projektek fordítása. Absztrakt-, interfész-, és generikus osztályok, virtuális eljárások. A virtuális eljárások megvalósítása, szerepe, használata.

11.3.1. Java program fordítása és futtatása

Java program futtatásához szükség van JVM-re, ami a Java Virtual Machine. A Java forrás ugyanis egy köztes kódra fordul, ami architektúra függetlenül (de Java verziótól függő lehet) a JVM futtat.

Ezek ellenére a JIT compilerrel lefordítható a Java forrás natív gépi kódra.

Fordítás: `javac HelloWorld.java`

Ennek outputja `HelloWorld.class` file, ezt futtatni a `java HelloWorld.class` parancssal lehet.

Itt a JVM futtat (`java` parancs)

JIT compilation automatikusan történik a gyakran használt kód elemekre.

Nagyobb projektek fordítása

Sok file, összetett projekt struktúra esetén nem jó megoldás simán a `javac` parancssal fordítani

Ekkor érdemes használni egy build rendszert, pl.: **maven**, vagy **gradle**

Érdemes alkalmas fejlesztői környezetet is választani, gyakran a build rendszerek a fejlesztői környezetek felületéről kezelhetőek. Pl. IntelliJ esetén.

- **mvn clean:** Cleans the project and removes all files generated by the previous build.
- **mvn compile:** Compiles source code of the project.
- **mvn test-compile:** Compiles the test source code.
- **mvn test:** Runs tests for the project.
- **mvn package:** Creates JAR or WAR file for the project to convert it into a distributable format.
- **mvn install:** Deploys the packaged JAR/ WAR file to the local repository.
- **mvn deploy:** Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests and building the project.

`pom.xml`-ben projekt adatai, dependency

Fordítási (és futtatási) opciók

`javac :`

- `-g` : Debugging információk generálása
- `-nowarn` : Warning üzenetek kikapcsolása
- `-verbose` : Információt logol a compiler tevékenységeiről

- `-Werror` : Fordítási folyamat terminálása, ha egy warning történik

`javac -version`

<https://docs.oracle.com/en/java/javase/13/docs/specs/man/javac.html>

`java`

- `-version`
- `-jar filenév` : Futtat egy `.jar` fileba enkapszulált Java programot
- `-verbose:class` egyek class-ok betöltéséről logol információkat
- `-verbose:gc` Garbage Collection eventekről logol információkat
- Opciók megadása után felsorolhatjuk a programunk parancssori paramétereit

<https://docs.oracle.com/en/java/javase/13/docs/specs/man/java.html>

<https://docs.oracle.com/en/java/javase/13/docs/specs/man/java.html#overview-of-java-options>

Parancssori paraméterek

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // args-ban  
    }  
}
```

Argumentumok száma: `args.length`

Más nyelvekkel ellentétben a **0. indexű argumentum nem a futtatott file neve**, hanem ténylegesen az első argumentum

11.3.2. C++ program fordítása és futtatása

Java-val ellentétben nem egy virtuális gép kódjára fordul, hanem natív gépi kódra.

Fordítás: `g++ main.cpp`

Output: `a.out` bináris, ezt futtathatjuk: `./a.out`

Nagyobb projektek fordítása

Nagyobb projektek esetén ugyan az a helyzet, mint Java-nál, nem ideális egy `g++` parancshoz felsorolni a linkelendő egységeket.

Használunk egy build rendszert, mint a Visual Studio-é, vagy `make`-et.

Fordítási opciók

`-o` : Kimeneti file névnek specifikálása

`-Wall` : minden warning mutatása

`-lm` : Matematikai függvénykönyvtárak linkelése

Preprocesszor → Fordítás assemblyre → Fordítás gépi kódra (assembler) → Linker (szerkesztés, futtatható állomány előállítása az egységekből)

Parancssori paraméterek

```
int main(int argc, char* argv[]) {  
    // ...  
    return 0;  
}
```

`argc` : Argumentumok száma

`argv` : Argumentumok tömbje

0. indexen a file neve áll!

11.3.3. Java absztrakt-, interfész-, generikus osztályok

Absztrakt osztályok

Amikor egy osztályban nem feltétlen szükséges implementálni egy metódust, csak az azt öröklő osztályokban.

Ekkor megjelölhetjük a metódust, mint `abstract` metódus. Viszont ekkor maga az egész osztály is `abstract` kell, hogy legyen.

Minden osztály `abstract`, ha van legalább egy `abstract` metódusa, de nem kell, hogy legyen, egy osztály `abstract` metódus nélkül is lehet `abstract`.

Ekkor ezt az `abstract` metódust nem implementáljuk, és ez az osztály nem példányosítható (azaz nem lehet meghívni a konstruktőröt).

De az öt öröklő osztályok konstruktőrből meg lehet hívni (`super`), így lehet konstruktora, közös logikákat lehet bele szervezni.

`abstract` metódus nem lehet:

- `private`: Mivel ha nem érik el a gyerekek, akkor nem is tudják felülríni. Csak esetleg egy ugyan olyan signature-jű metódust létrehozni, de nem lesz `override`.
- `final`: Mivel a `final` metódusok nem overridelhetők.

Példa

```
abstract class Hangszer {  
    abstract public void szolj(Hang h);  
}
```

A `hangszer` mint olyan egy absztrakt fogalom, nem tud egy adott hangon szólni. Viszont mondjuk a belőle öröklő csellő már tud csellő hangon szálni:

```
public class Csello extends Hangszer {  
    @Override  
    public void szolj() {  
        System.out.println("csellohang");  
    }  
}
```

Az `@override` annotáció segít olvasni a kódot, illetve ha valójában nem is egy `override` a metódus, akkor szól a linter / fordító.

Interfészek

Akkor használjuk, ha minden metódus `abstract`.

`class` helyett `interface` kulcsszó

Megkötések:

- Nem lehet konstruktőr
- Metódusok impliciten `public abstract`-ök.
- Adattagok impliciten `public static final`-ök.
 - `public`: Mivel nem példányosítható, valószínűleg ezt szeretnénk, bár a `protected` is elkövethető.
 - `static`: Mivel nem példányosíthatók, nem lehet őket objektumhoz (példányhoz) kötni, csak osztályhoz.
 - `final`: Mivel nincs konstruktőr, mindenképpen legyenek inicializálva.

Leírja, milyen módon lehet használni az öt implementáló osztályokat. De csak a működés lehetőségeit írja elő, az implementáció az öröklő osztályokban van.

Visszafelé kompatibilitási okok miatt lehetséges az interface-ben törzset is adni (implementációt) az osztályoknak, de ez csak akkor jelentős, ha meglévő kódbázisokban használt interfacen akarunk módosítani (nem biztos, hogy az új metódust implementálják a másik kódbázis osztályai). Ez a `default` kulcsszóval lehetséges a metódus signaturejében.

Példa

```
interface Hangszer {  
    void szolj(Hang h); // impliciten public és abstract  
}  
class Zongora implements Hangszer {  
    public void szolj(Hang h) {  
        System.out.println("zongorahang");  
    }  
}
```

Generikus osztályok

Generikus: típus paraméter

Szintaxis:

- `interface InterfeszNev<T> { ... }`
- `class OsztalyNev<T> { ... }`
- `<T> T fuggvenyNev(T p);`

Használatkor hasonlóan kacsacsőrök között tudjuk megadni a konkrét típust, amit alkalmazni szeretnénk:

```
OsztalyNev<Long> i = new OsztalyNev<Long>(new Long(1));
```

Csak referenciatípus lehet, primitív, pl. `int` nem. (De `Integer` igen)

Példa

```
class Generikus<E> {  
    private E x;  
    public Generikus(E x) {  
        this.x = x;  
    }  
    public E getX() {  
        return x;  
    }  
    public void setX(E x) {  
        this.x = x;  
    }  
}  
class GenMetodus {  
    public String toString() {  
        return "GenMetodus";  
    }  
    <T> T f(T p) {return p;}  
}  
class GenericsPelda {  
    public static void main(String[] args) {  
        //Generikus<long> a1 = new Generikus<long>(1); // fordítasi hiba  
        Generikus<Long> a1 = new Generikus<Long>(new Long(1));  
        Generikus<String> a2 = new Generikus<String>("egy");  
        GenMetodus b = new GenMetodus();  
        Generikus<GenMetodus> a3 = new Generikus<GenMetodus>(b);  
        System.out.println(b.f("valami"));  
        System.out.println(b.f(9));  
    }  
}
```

Példa beépített generikus osztállyal

```
List<String> strings = new ArrayList<>();  
//... add String instances to the strings list...  
  
for (String aString : strings) {  
    System.out.println(aString);  
}
```

11.3.4. C++ virtuális metódusok, absztrakt osztályok, templatek

Virtuális metódusok

Alapvetően (`virtual` kulcsszó nélkül) C++-ban nem valósul meg polimorfikusság. Hiába egyezik meg a metódus signature-je teljesen az ősével, ha egy függvény az ős szerint vár paramétert, az az ős-beli metódust fogja meghívni.

Ennek az az oka, hogy ilyenkor statikus kötés történik, a fordító azt látja, hogy `Hangszer`-t vár a metódus, így statikusan aként adja át a `Csello` példányt is.

Ekkor fordítási időben eldől a típus.

Ez virtualizációval módosítható.

Az ősben azokat a metódusokat, amelyeket felül szeretnénk definiálni egy leszármazottban, meg kell jelölni `virtua` kulcsszóval. Ezek a metódusok már polimorfikusak lesznek.

Ekkor csak futási időben dől el a típus.

`override` kulcsszó: hasonló a Java-féle annotációhoz, jelzi, ha egy metódus valójában nem is override (nincs neki vitruális megfelelője az ős osztályban).

`final`: Ha a leszármazottban meg akarjuk akadályozni, hogy az ő leszármazottja megint csak felülírja a metódust.

A `final` értelmezett osztályokra is, ilyen osztályból nem lehet örökölni.

Virtuális destruktur: Figyelni kell rá, ha objektumot törlünk őstípus alapján (polimorfikusan), akkor legyen virtual a destruktur!

Absztrakt osztályok

Ha egy osztály rendelkezik **pure virtual** metódussal, akkor az absztrakt, és nem lehet példányosítani.

A leszármazott köteles implementálni az örökölt pure virtual metódusokat, különben ő is absztrakt lesz.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Az `= 0` jelzi, hogy nem sime virtual, hanem pure virtual metódus.

Templatek

Mint a Java generikusok.

```
template <typename T> // typename T vagy class T ugyanaz
class Example {
    T t;
public:
    Example(const T& t) : t(t) { }
    bool is_empty() const {
        return t.empty();
    }
};

int main () {
    Example<int> ot(5);
    // cout << ot.is_empty() << endl; -> HIBA LENNE !
    Example<string> pelda("pelda");
    cout << pelda.is_empty() << endl;
}
```

Template osztályokat a fordító többször "példányosítja" (többet hoz létre).

Minden típushoz, amivel használjuk, generál egyet.

Emiatt csak használatkor derül ki, hogy lehet-e azzal a típussal használni.

Template-ben lehet nem-típus paraméter is:

```
template <typename T, unsigned N>
class Tomb {
    T t [N];
    /* ... */
};

int main () {
    Tomb <int, 5> tomb;
}
```

12. Programozás alapjai

12.1. 1. Algoritmusok vezérlési szerkezetei és megvalósításuk C programozási nyelven. A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés.

12.1.1. Vezérlési módok

Segítségükkel azt fejezzük ki, hogy egyszerűbb műveletekből hogyan építhetünk fel összetettebb műveleteket és ennek milyen lesz a vezérlése, azaz milyen sorrendben kell végrehajtani az öt felépítő utasításokat.

Négy fő vezérlési módot különböztetünk meg:

- **Szekvenciális:** Véges sok művelet rögzített sorrendben egymás után történő végrehajtása
- **Szelekciós:** Véges sok művelet közül adott feltétel alapján valamelyik végrehajtása
- **Ismétléses:** Adott műveletet adott feltétel szerinti ismételt végrehajtása
- **Eljárás:** Adott művelet alkalmazása adott argumentumokra, ami az argumentumok értékének meghatározott változását eredményezi

Ezek nyelv független fogalmak, amikor egy imperatív programozási nyelvet el akarunk sajátítani, a legfontosabb annak megismerése, hogy ezeket a vezérlési módokat milyen utasításokkal tudjuk (ha tudjuk) megvalósítani.

12.1.2. Algoritmusok leírása

Több féle képpen meg tudjuk adni egy algoritmus vezérlését, azaz azt az előirást, amely az algoritmus minden lépéssére kijelöli, hogy a lépés végrehajtása után melyik lépés végrehajtása következik.

- **Természetes nyelvi leírás:** Legegyszerűbb megközelítés, szövegesen, minden lépést foglalva írja le az algoritmust. Nagyon távol áll egy gépi megvalósítástól.
- **Pszeudo kód:** Egy programozási nyelv szerű struktúrált nyelv, de sokkal szabadabb, mint egy valódi programozási nyelv, nem kell minden részletet definiálni.
- **Folyamatábra:** Grafikus, kevésbé strukturált gráf reprezentációja a végrehajtásnak, amely a működési folyamatra koncentrál
- **Szerkezeti ábra:** Szintén grafikus, strukturált leírása az algoritmus felépítésének leírására, amely leírja a működési folyamatot is

12.1.3. Folyamatábra

Akkor használjuk, ha csak a kész algoritmus működését szeretnénk leírni, és a szerkezete kevésbé fontos.

Az algoritmus egyes lépéseiit egy gráf csúcspontjaiban definiáljuk, amely pontokat irányított nyilakkal kötjük össze, ezzel kijelölve a végrehajtás irányát.

Közeli áll az assembly nyelvhez.

Szintaxis

Legyenek $M = M_1, \dots, M_k$ műveletek, és $F = F_1, \dots, F_l$ feltételek.

Az (M, F) feletti folyamatábra olyan irányított gráf, amelyre teljesül a következő 5 feltétel:

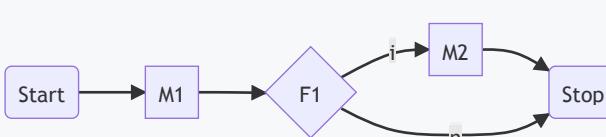
- Van egy olyan pontja, ami a **Start** művelettel van címkézve, és ebbe a pontba nem vezet él.
- Van egy olyan pontja, ami a **Stop** művelettel van címkézve, és ebből nem indul ki él.
- minden pontja vagy egy M -beli művelet, vagy egy F -beli feltétel a **Start** és **Stop** pontokon kívül.
- Ha egy pont
 - M -beli művelettel van címkézve, akkor belőle egy él indul ki
 - F -beli feltéttel van címkézve, akkor belőle két él indul ki, és ezek az **i** (igen), illetve **n** (nem) címkéket viselik.
- A gráf minden pontja elérhető a **Start** címkéjű pontból.

Szemantika

Egy folyamatábrát a következőképpen kell értelmezni:

- A végrehajtást a **Start** pontból kell kezdeni.
- Az összetett utasítás akkor ér véget, ha elérük a **Stop** pontot, azaz a vezérlést megkapja a **Stop** pont.
- A gráf egy pontjának a végrehajtását attól függően definiáljuk, hogy az M -beli utasítással, vagy F -beli címkével van címkézve.
 - Ha a pontban M -beli művelet van, akkor a művelet végrehajtódik és a vezérlés a gráf azon pontjára kerül, amelybe a pontból kiinduló él vezet.
 - Ha a pont F -beli feltéttel van címkézve, akkor kiértékelődik a feltétel. Ha az értéke igaz, akkor az a pont kap vezérlést. amelybe az **i** (igen) címkéjű él vezet, egyébként az a pont kapja meg a vezérlést, amelybe az **n** (nem) címkéjű él vezet.

Példa



1. Start pontból a vezérlés rákerül az M_1 utasítást tartalmazó blokkra.

2. M_1 végrehajtása után az F_1 feltétel kiártákelése történik.

- Ha a feltétel igaz volt, akkor végrehajtjuk az M_2 utasítást

3. Akár végre hajtottuk az M_2 utasítást, akár nem, ezen a ponton eljutunk a Stop csúcsig

12.1.4. Szekvenciális vezérlés

Szekvenciális vezérlésről akkor beszélünk, amikor a P probléma megoldását úgy kapjuk, hogy a problémát P_1, \dots, P_n részproblémákra bontjuk, majd az ezekre adott megoldásokat (részalgoritmusokat) sorban, egymás után végrehajtjuk.

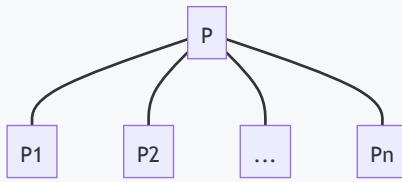
P_1, \dots, P_n lehetnek elemi műveletek, de lehetnek összetettek is, amiket utána tovább kell bontani.

Folyamatábra



Szerkezeti ábra

Itt az látszódik, hogy a P problémának a megoldását a P_1, \dots, P_n problémák megoldásával kapjuk. A sorrendiséget csak a felsorolás sorrendje jelzi.



C-ben

```
{  
    P1;  
    ...  
    Pn;  
}
```

12.1.5. Szelekciós vezérlés

A kiválasztás módjától függően megkülönböztetünk pár altípust:

- Egyszerű szelekciós vezérlés
- Többszörös szelekciós vezérlés
- Esetkiválasztásos szelekció
- A fentiek kiegészítve **egyébként** ágakkal

Egyeszerű szelekciós vezérlés

Egyetlen művelet, és egyetlen feltétel van.

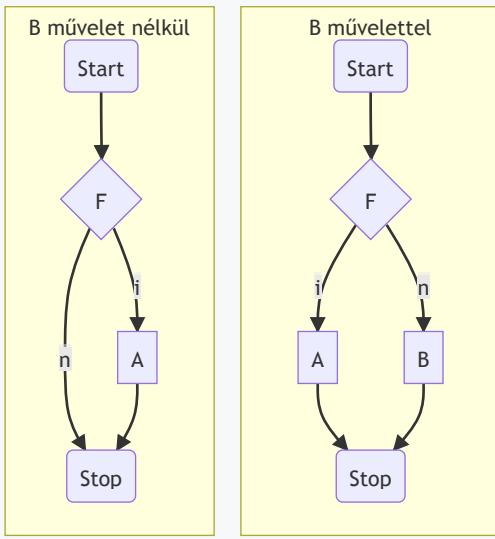
Maga a művelet persze lehet összetett.

Legyen F egy logikai kifejezés, A pedig tetszőleges művelet. Az F feltételből és az A műveletből képzett **egyszerű szelekciós vezérlés** a következő vezérlési előírást jelenti:

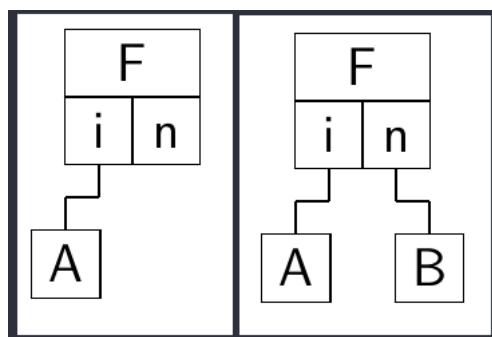
1. Értékeljük ki az F feltételt és folytassuk a 2. lépéssel
2. Ha F értéke igaz, akkor hajtsuk végre az A műveletet, és fejezzük be az összetett művelet végrehajtását
3. Egyébként ha F értéke hamis, akkor fejezzük be az összetett művelet végrehajtását

A vezérlés bővíthető úgy, hogy a 3. pontban üres művelet helyett egy B műveletet hajtunk végre. (`else` ág, minimális módosításokkal felírható hasonló definíció)

Folyamatábra



Szerkezeti ábra



C-ben

```
if(F) {
    A;
}
```

```
if(F) {
    A;
} else {
    B;
}
```

Feltételes kifejezés (ternary): `a ? b : c`

A C nyelv egyetlen 3 operandusú művelete.

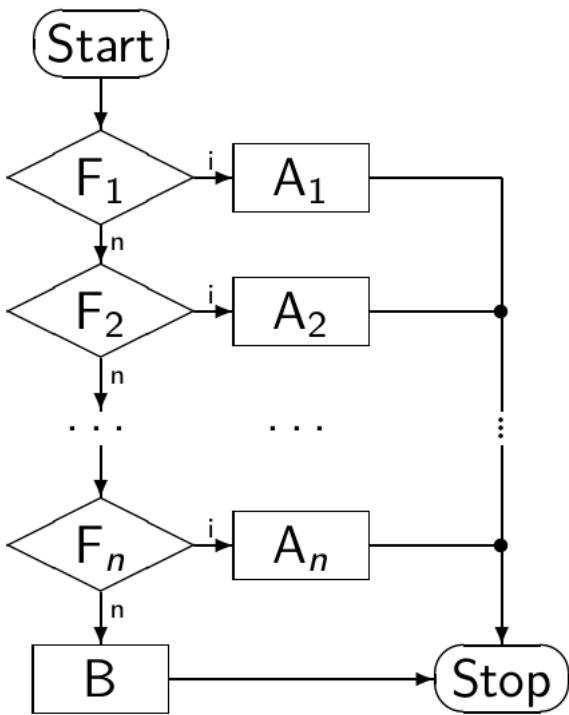
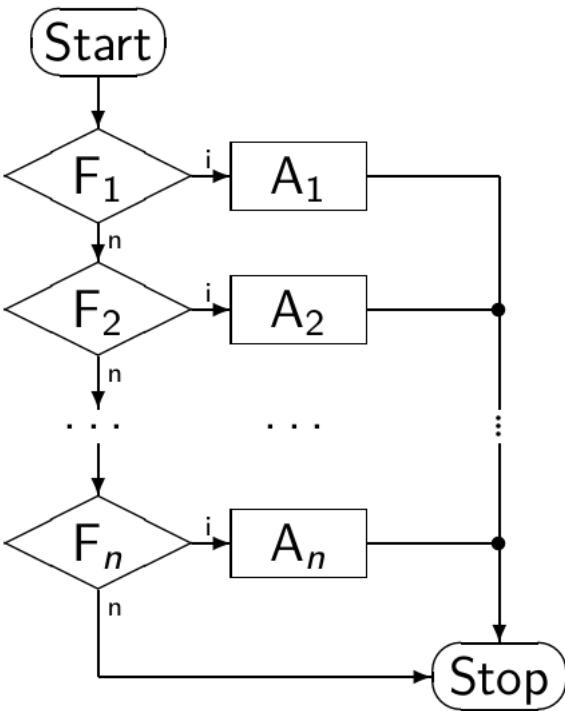
Többszörös szelekciós vezérlés

Több feltétel, több művelettel.

Legyenek F_i logikai kifejezések, A_i pedig tetszőleges műveletek $1 \leq i \leq n$ -re (azaz minden feltételből, mint A műveletből van n darab). Az F_i feltételekből és A_i műveletekből képzett többszörös szelekciós vezérlés a következő vezérlési előírást jelenti:

1. Az F_i feltételek sorban történő kiértékelésével adjunk választ a következő kérdésre: van-e olyan i ($1 \leq i \leq n$), amelyre teljesül, hogy az F_i feltétel igaz és az összes F_j ($1 \leq j < i$) feltétel hamis? (Azaz keressük az első F_i feltételt, ami igaz.)
2. Ha van ilyen i , akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
3. Egyébként, vagyis ha minden F_i feltétel hamis, akkor fejezzük be az összetett művelet végrehajtását.

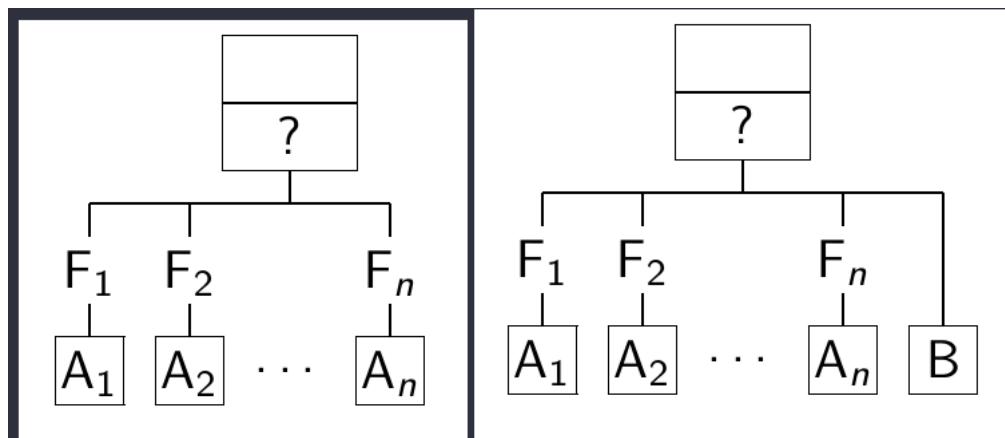
Folyamatábra



Az utóbbi rendelkezik egyébként ággal

Szerkezeti ábra

Mivel a fenti dobozba nem lehetne beírni minden feltételt, és nem is lenne egyértelmű, hogy melyikhez melyik tartozik, csak egy ?-et írunk.



Valójában összeépíthető az egyszerű szelekciós vezérlés szerkezeti ábrájával, a hamis ágegy újabb egyszerű szelekcióba vezet, és így tovább ahány feltétel van (és a végén egy esetleges `else` ág).

C-ben

```
if(F1) {  
    A1;  
} else if(F2) {  
    A2;  
    ...  
} else if(Fn) {  
    An;  
} else {  
    B;  
}
```

Fontos, hogy a zárójelezésre figyeljünk, az alábbi két blokk ekvivalens, hiába tűnhet úgy, mintha a másodikban az `else` ág az első `if`-hez tartozna:

```
if(F1) {  
    if(F2)  
        A1;  
    else  
        A2;  
}
```

```
if(F1)  
    if(F2)  
        A1;  
else  
    A2;
```

A C nyelv nem whitespace érzékeny.

Esetkválasztásos szelekciós vezérlés

Akkor alkalmazhatjuk, ha a többszörös szelekció feltételeit átírhatjuk úgy, hogy **elemek valamelyen halmazba tartozását** vizsgálják.

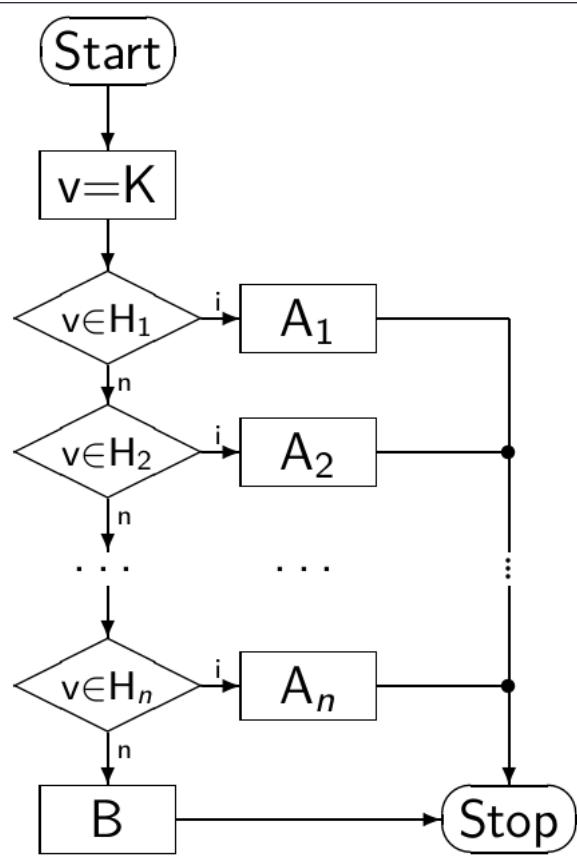
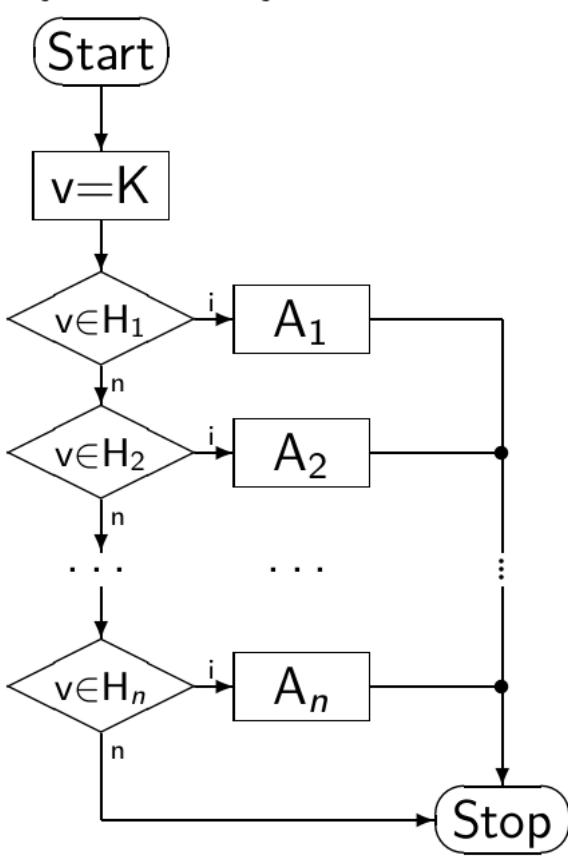
`switch`

Legyen K egy adott típusú kifejezés, és legyenek H_i -k olyan halmazok, melynek elemeinek típusa megegyezik K típusával. Legyenek továbbá A_i tetszőleges műveletek, ahol $1 \leq i \leq n$ teljesül. A K szelektor kifejezésből, H_i kiválasztó halmazokból és A_i műveletekből képzett esetkválasztásos szelekciós vezérlés a következő vezérlési előírást jelenti:

1. Értékeljük ki a K kifejezést és folyassuk a 2. lépéssel.
2. Adjunk választ a következő kérdésre: Van-e olyan i ($1 \leq i \leq n$), amelyre teljesül, hogy a $K \in H_i$, és $K \notin H_j$, ahol ($1 \leq j < i$)?
3. Ha van ilyen i , akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
4. Egyébként, vagyis ha K nem eleme egyetlen H_i halmaznak sem, akkor fejezzük be az összetett művelet végrehajtását.

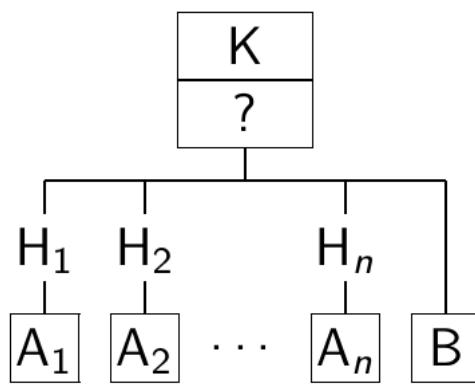
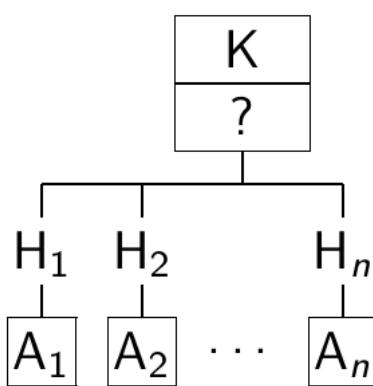
Itt is lehet egyébként ág (`default`), minimálisan módosul a definíció ha jelen van.

Folyamatábra



Utóbbi a `default` ágas

Szerkezeti ábra



K az adott kifejezés, aminek a H_i -k be tartozását vizsgáljuk.

C-ben

```

switch(K) {
    case H1:
        A1;
        break;
    ...
    case Hn:
        An;
        break;
    default:
        B;
        break;
}
  
```

Alapból azt fejezzük ki, hogy melyik `case`-től kezdődően hajtsuk végre az A_i utasításokat (mindhatni, hogy a `case`-ek belépései pontot határoznak meg). Így ha `break`-el, vagy `return`-el zárunk minden utasítást (esetleg kivéve az utolsót), akkor esetkiválasztásos szelekciót valósít meg a struktúra.

Itt a `case`-ek után egy elem állhat, nem egy halmaz, ezt a `case` működéséből adódóan (azaz abból, hogy egy belépései pontot határoz meg) viszont a következőképpen meg tudjuk oldani:

```

case x_i1:
case x_i2:
...
case x_ini:
    Ai;
    break;
  
```

Ekkor az A_i utasítás akkor fog kiválasztódni, ha $K \in H_i = x_{i,1}, x_{i,2}, \dots, x_{i,n_i}$

12.1.6. Eljárásvezérlés

Egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.

Két fajta: eljárásművelet, függvényművelet.

Függvényművelet

Matematikai függvények álrólánosítása

Függvényművelet specifikációja:

- Művelet elnevezése

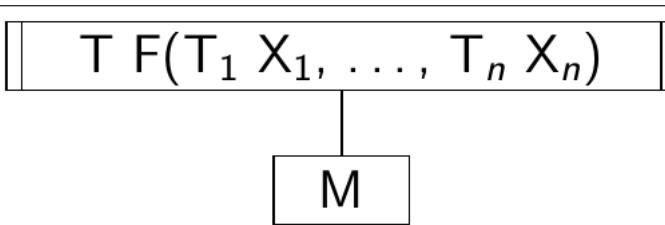
- Paraméterek felsorolása
- Paraméterek típusa
- A műveletek hatásának leírása
- Eredménytípus

Jelölés: $T \ F(T_1 \ X_1, \dots, T_n \ X_n)$

- T : a függvényművelet eredménytípusa
- F : a függvényművelet neve
- T_i : i . paraméter típusa
- X_i : i . paraméter azonosítója

Zárójeleket akkor is kirakjuk, ha a paraméterlista üres

Szerkezeti ábra



Formális paraméter: Függvényművelet leírásában használt paraméterek

Argumentum: Amire konkrét esetben végre szeretnénk hajrani a műveletet

C-ben

```

T F(T1 X1, ... , Tn Xn)
{
    M;
}
  
```

Eljárásművelet

Alkalmazása adott argumentumokra az argumentumok értékének pontosan meghatározott megváltozását eredményezi.

Minden eljárásműveletnek rögzített számú paramétere van, és minden paraméter rögzített adattípusú.

Három mód:

- **Bemenő mód:** Ha a művelet végrehajtása nem változtathatja meg az adott argumentum értékét.
- **Kimenő mód:** Ha a művelet eredménye nem függ az adott argumentum végrehajtás előtti értékétől, de az adott argumentum értéke a művelet hatására megváltozhat.
- **Be- és kimenő (vegyes) mód:** Ha a művelet felhasználhatja az adott argumentum végrehajtás előtti értékét és az argumentum értéke a művelet hatására meg is változhat.

Ezek a módok C-ben éppenséggel ugyan úgy működnek függvényműveletek esetén is, de nem feltétlen van ez így minden nyelv esetén

C-ben a kimenő mód pointerekkel valósítható meg. Deklarációban: `T_i *X_i`, függvénytörzsben: `*X_i`-vel dereferáljuk. Híváskor: `&A_i`-vel pointert adunk át.

Eljárásművelet specifikációja:

- Művelet elnevezése
- Paraméterek felsorolása
- Paraméterek adattípusai
- Művelet hatásának leírása

Eljárásművelet általános jelölése: $P(m_1 \ X_1 : T_1; \dots; m_n \ X_n : T_n)$

- P : az eljárás neve
- m_i : az i . paraméter kezelési módja

- X_i : az i . paraméter azonosítója
- T_i : az i . paraméter adattípusa

C-ben

```
void visszatérési érték típus.
```

A függvényműveletekkel ellentétben nem lehet egy összetett művelet részkifejezése (pont azért, mert nem vesz fel értéket, mivel nincs visszatérési értéke).

12.1.7. Ismétléses vezérlés

Ötféle ismétléses vezérlés:

- Kezdőfeltételes
- Végfeltételes
- Számlálásos
- Hurok
- Diszkrét

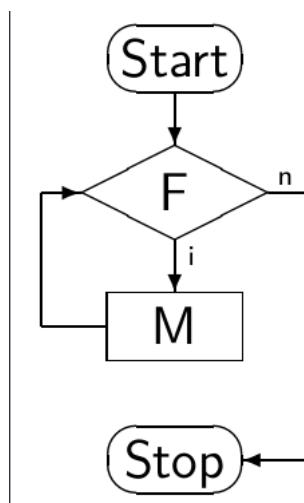
Kezdőfeltételes ismétléses vezérlés

A ciklusmag ismételt végrehajtását egy belépési feltételhez kötjük.

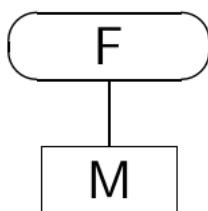
Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F ismétlési feltételből és az M műveletből (a ciklusmagból) képzett kezdőfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:

1. Értékeljük ki az F feltételt és folytassuk a 2. lépéssel.
2. Ha F értéke hamis, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
3. Egyébként, vagyis ha az F értéke igaz, akkor hajtsuk végre az M műveletet, majd folytassuk az 1. lépéssel.

Folyamatábra



Szerkezeti ábra



C-ben

```
while (F) {
    M;
}
```

Végfeltételes ismétléses vezérlés

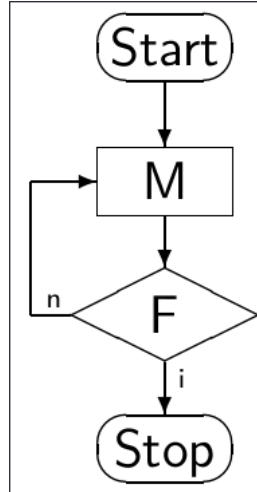
A ciklusmag elhagyását kötjük egy kilépési feltételhez.

Legyen F egy logikai kifejezés, M pedig tetszőleges művelet. Az F kilépési feltételből és az M műveletből (ciklusmagból) képzett **végfeltétes ismétléses vezérlés** a következő vezérlési előírást jelenti:

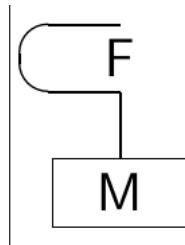
1. Hajtsuk végre az M műveletet majd folytassuk a 2. lépéssel.
2. Értékeljük ki az F feltételt és folytassuk a 3. lépéssel.
3. Ha F értéke igaz, akkor az ismétléses vezérlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
4. Egyébként, vagyis ha az F értéke hamis, akkor folytassuk az 1. lépéssel.

Azaz a ciklusmag legalább egyszer lefut

Folyamatábra



Szerkezeti ábra



C-ben

```
do {  
    M;  
} while (!F);
```

Számlálásos ismétléses vezérlések

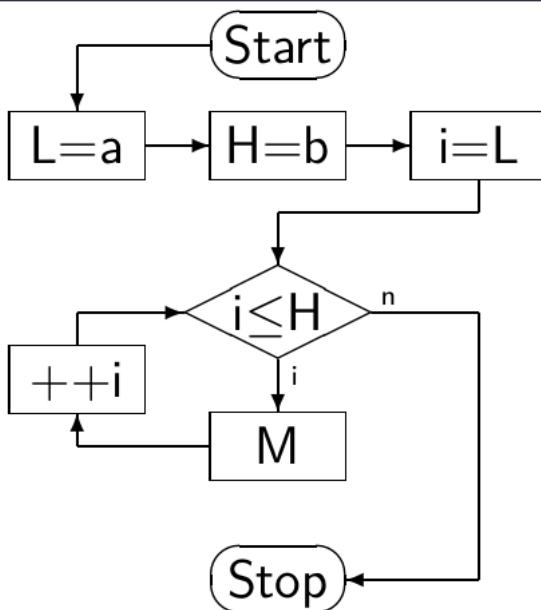
Legyen a és b egész érték, i egész típusú változó, M pedig tetszőleges művelet, amelynek nincs hatása a , b és i értékére.

Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett **növekvő (csökkenő) számlálásos ismétléses vezérlés** az alábbi vezérlési előírást jelenti:

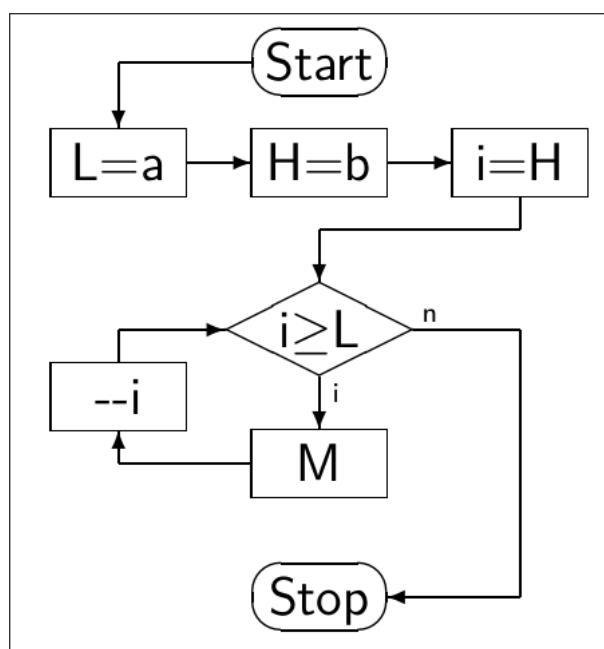
1. Legyen $i = a$ ($i = b$) és folytassuk a 2. lépéssel.
2. Ha $b < i$ ($i < a$), akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
3. Egyébként, vagyis ha $i \leq b$ ($a \leq i$), akkor hajtsuk végre az M műveletet, majd folytassuk a 4. lépéssel.
4. Növeljük (csökkentsük) i értékét 1-gyel, és folytassuk a 2. lépéssel.

Folyamatábra

Növekvő:

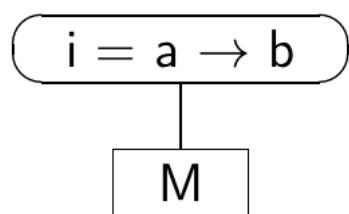


Csökkenő:

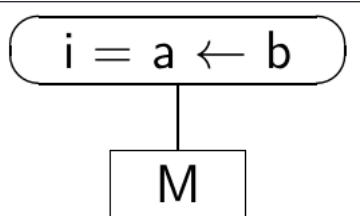


Szerkezeti ábra

Növekvő:



Csökkenő:



C-ben

```

for (kif_11, kif_12, kif_13;
    kif_2;
  
```

```

    kif_31,kif_32) {
utasítás;
}

```

Ez szemantikailag egyenértékű a következővel:

```

kif_11; kif_12; kif_13;
while (kif_2) {
    utasítás;
    kif_31;kif_32;
}

```

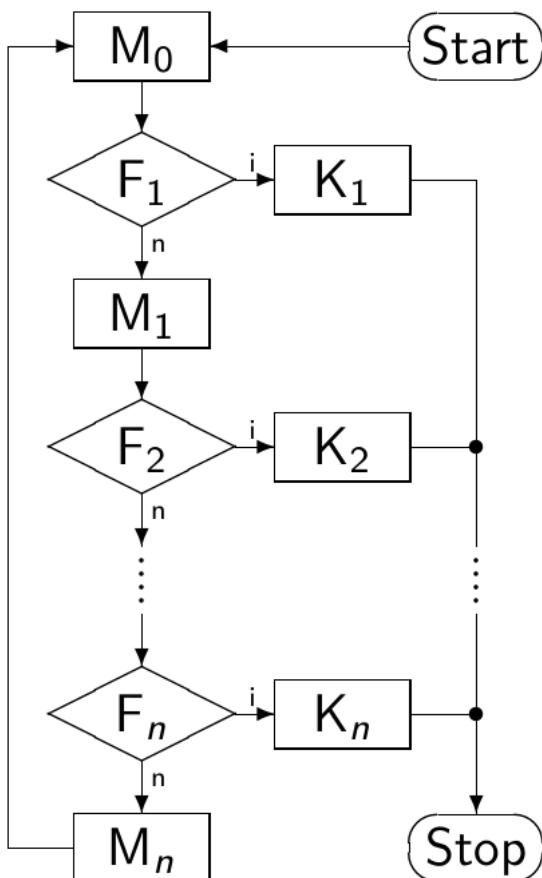
Hurok ismétléses vezérlés

Amikor a ciklusmag ismétlését a ciklusmagon belül vezéreljük úgy, hogy a ciklus különböző pontjain adott feltételek teljesülése esetén a ciklus végrehajtását befejezzük, hurok ismétléses vezérlésről beszélünk.

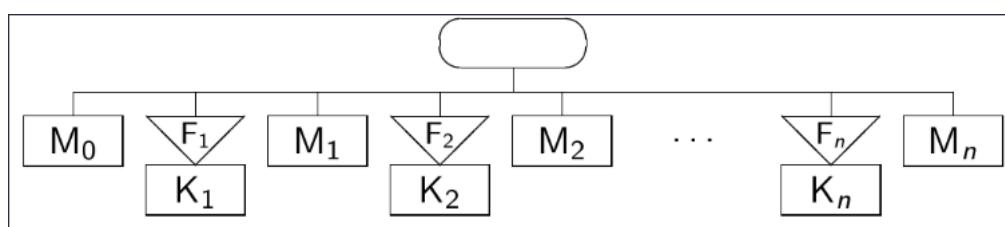
Legyenek F_i logikai kifejezések, K_i és M_j pedig tetszőleges (akár üres) műveletek $1 \leq i \leq n$ és $0 \leq j \leq m$ értékekre. Az F_i kijáratú feltételekből, K_i kijáratú műveletekből és az M_i műveletekből képzett hurok ismétléses vezérlés a következő előírást jelenti:

1. Az ismétléses vezérlés következő végrehajtandó egysége az M_0 művelet.
2. Ha a végrehajtandó egység az M_j művelet, akkor ez végrehajtódik. $j = n$ esetén folytassuk az 1. lépéssel, különben pedig az F_{j+1} feltétel végrehajtásával a 3. lépésben.
3. A végrehajtandó egység az F_i feltétel $1 \leq i \leq n$, akkor értékeljük ki. Ha F_i igaz volt, akkor hajtsuk végre a K_i műveletet, és fejezzük be a vezérlést. Különben a végrehajtás az M_i műveettel folytatódik a 2. lépésben.

Folymatábra



Szerkezeti ábra



Nincs rá olyan vezérlési forma, amivel körvezlenül megvalósítható.

Kezdőfeltételes ismétléses vezérlés és egyszerű szelekció segítségével kifejezhetjük.

Nyelvfüggetlen megközelítés

```
tovabb = 1;
while (tovabb) {
    M0;
    if (F1) {
        tovabb = 0;
        K1;
    } else {
        M1;
        ...
        if (Fn) {
            tovabb = 0;
            Kn;
        } else {
            Mn;
        }
    }
}
```

A C nyelvben a ciklusmag folyamatos végrehajtásának megszakítására két utasítás használható:

- `break` : megszakítja a ciklust, a program végrehajtása a ciklusmag utáni első utasítással folytatódik.
- `continue` : megszakítja a ciklusmag aktuális lefutását, a vezérlés a ciklus feltételének kiértékelésével (`while`, `do while`) illetve az inkrementáló kifejezés kiértékelésével (`for`) folytatódik.

C-ben `break` használatával:

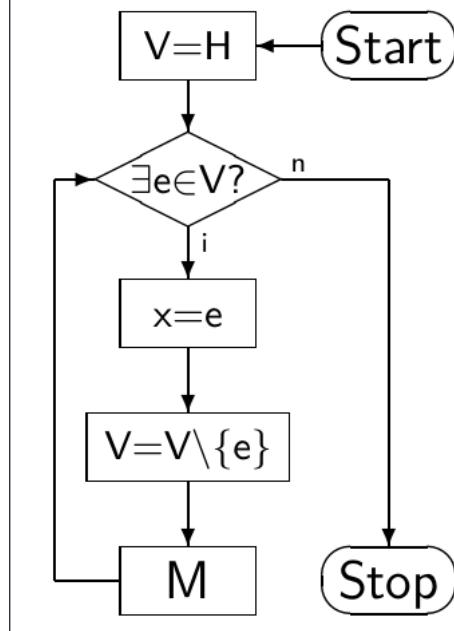
```
while(1) {
    M0;
    if (F1) {
        K1; break;
    }
    M1;
    ...
    if (Fn) {
        Kn; break;
    }
    Mn;
}
```

Diszkrét ismétléses vezérlés

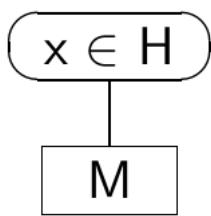
Diszkrét ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani egy halmaz minden elemére tetszőleges sorrendben. Legyen `x` egy `T` típusú változó, `H` a `T` értékkészletének részhalmaza, `M` pedig tetszőleges művelet, amelynek nincs hatása `x` és `H` értékére. A `H` halmazból, `x` ciklusváltozóból és `M` műveletből (ciklusmagból) képzett diszkrét ismétléses vezérlés az alábbi vezérlési előírást jelenti:

1. Ha a `H` halmaz minden elemére végrehajtottuk az `M` műveletet, akkor vége a vezérlésnek.
2. Egyébként vegyük a `H` halmaz egy olyan tetszőleges `e` elemét, amelyre még nem hajtottuk végre az `M` műveletet, és folytassuk a 3. lépéssel.
3. Legyen `x = e` és hajtsuk végre az `M` műveletet, majd folytassuk az 1. lépéssel.

Folyamatábra



Szerkezeti ábra



C-ben

A diszkrét ismétléses vezérlésnek nincs közvetlen megvalósítása a C nyelvben.

A megvalósítás elsősorban attól függ, hogy az ismétlési feltételben megadott halmazt hogyan reprezentáljuk.

12.2. 2. Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük, és műveleteik. Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord, és az unió típus. Az egyes típusok szerepe, használata.

Az **adattípus** a programnak egy olyan komponense, amely két összetevője, az értékhalmaz és az értékhalmaz elemein végezhető műveletek által meghatározott.

12.2.1. Egyszerű adattípusok

Az elemi adattípusok közé tartoznak azok az adattípusok, amelyek értékhalmaza elemi értékekből áll, azaz nem összetett adatok alkotják.

Ezeket az elemi adattípussal rendelkező értékeket már nem lehet további, önmagukban is értelmes részekre bontani.

C típus	méret (bit)	alsó határ	felső határ
char	8	fordító függő	fordító függő
signed char	8	$-128(-2^7)$	$127(2^7 - 1)$
unsigned char	8	0	$255(2^8 - 1)$
short int	16	$-32768(-2^{15})$	$32767(2^{15} - 1)$
signed short int	16	$-32768(-2^{15})$	$32767(2^{15} - 1)$
unsigned short int	16	0	$65535(2^{16} - 1)$
int	32	$-2147483648(-2^{31})$	$2147483647(2^{31} - 1)$
signed int	32	$-2147483648(-2^{31})$	$2147483647(2^{31} - 1)$
unsigned int	32	0	$4294967295(2^{32} - 1)$
long int	32	$-2147483648(-2^{31})$	$2147483647(2^{31} - 1)$
signed long int	32	$-2147483648(-2^{31})$	$2147483647(2^{31} - 1)$
unsigned long int	32	0	$4294967295(2^{32} - 1)$

<code>long long int</code>	64	-2^{63}	$2^{63} - 1$
<code>signed long long int</code>	64	-2^{63}	$2^{63} - 1$
<code>unsigned long long int</code>	64	0	$2^{64} - 1$
<code>float</code>	32	$-3.4028234663852886E + 38$	$3.4028234663852886E + 38$
<code>double</code>	64	$-1.7976931348623157E + 308$	$1.7976931348623157E + 308$
<code>long double</code>	64	$-1.7976931348623157E + 308$	$1.7976931348623157E + 308$

Egész

C-ben `int` és `char` típusok használhatóak egészek tárolására

Értelmezési tartomány az alábbiak szerint módosítható:

- `signed`: A típus előjeles értéket fog tartalmazni. Egy bit az előjelhez lesz felhasználva, nem az érték nagyságához.
- `unsigned`: A típus előjel nélküli, nemnegatív értéket fog tartalmazni, minden bit felhasználható az érték nagyságához.
- `short`: kevesebb biten tárolódik, így kisebb az értelmezési tartománya. Ez a módosító már nem tehető ki a `char` elő.
- `long`: Ábrázolására több bit áll a rendelkezésre, azaz több érték ábrázolható vele. Akár duplán is alkalmazható (`long long`), és a `char` elő szintén nem kerülhet oda.

Pontosság

Az értelmezési tartomány határain belül valamennyi számot pontosan ábrázolnak.

Különböző architektúrákon az egyes típusok mérete lehet más, de minden C megvalósításra igaz, hogy a `short` legfeljebb akkora, mint az `int`, ami legfeljebb akkora, mint a `long`, ami legfeljebb akkora, mint a `long long`

Pontos határok elérhetőek a `limits.h` headerben, ami preprocesszor definíciókkal közli velünk, mik a határok az adott architektúrán, pl.: `#define UCHAR_MAX 255`

Műveletek

- Aritmetikai műveletek: `+`, `-`, `*`, `/`, és `%`.
 - `/` ekkor a maradékos osztás egész része
 - `%` ekkor a maradékos osztás maradéka
- Relaciós műveletek: `<`, `<=`, `>`, `>=`, `==`, `!=`.
- Bitenkénti logikai műveletek: `&`, `|`, `^`, `<<`, `>>`, `~`

Egész kifejezések típusa: Az eredmény típusa a két operandus típusa közül a nagyobbik lesz.

Számábrázolás

n bit esetén 2^n állapotot tudunk megkülönböztetni, ez ennyi különböző szám ábrázolását jelenti.

- `unsigned` esetben: $[0, \dots, 2^n - 1]$ zárt intervallumból vesz fel értéket. Az n db egymás utáni bitet bináris számként értelmezve kapjuk meg a reprezentált értéket.
- `signed` esetben: $[-2^{n-1}, \dots, 2^{n-1} - 1]$ zárt intervallumból vesz fel értéket.
 - Nemnegatív értékek: 0 értékű bittel kezdődő n bites bitsorozat a szám kettés számrendszerbeli alakját ábrázolja (bevezető 0 bitekkel kiegészítve).
 - Negatív értékek: Eltároláskor adjunk hozzá a negatív értékhez 2^n -t, és az eredményt tároljuk el. 1-essel kezdődő bitsorozat pozitív számként értelmezett értékéből levonva a 2^n értéket kapjuk a reprezentált negatív értéket.

Ez a jellegű negatív szám tárolás a **kettes komplement**

Úgy is megadható, hogy a reprezentálandó értéket negáljuk (0-ák, és 1-esek felcserélése), és hozzáadunk 1-et.

Karakter

`char`-ban tárolt értéket lehet számként, és karakterként is értelmezni.

A `char`-ban tárolt számok egy meghatározott kódtáblából hivatkoznak egy karakterre. Mivel ez az alkalmazott táblától függ, érdemes a kódban karakterrel hivatkozni, ha például egy feltételben ellenőrzünk, nem számmal (`'a'` helyett, hogy `97`).

A C alapvetően az ASCII táblát használja. Így a nem ASCII karakterekkel vigyázni kell, mert pl. az UTF8 kódolás esetén egyes karakterek több byteon tárolódnak, ezt a C nem tudja kezelni.

Escape szekvenciák

Nem megjeleníthető karakterek, valami más hatásuk van. \ karakterrel kezdődően adjuk meg.

Megnevezés	Escape szekvencia
újsor	\n
vízszintes tab	\t
backslash	\\

Pontosság, műveletek, számábrázolás az egészkről szóló részben

Logikai

Eredetileg nem volt része a nyelvnek, a C⁹⁹ szabvány vezette be.

_Bool típus 0, 1 értékkészlettel.

A logikai, és az egész típusok konvertibilisek C-ben.

Ez onnan ered, hogy C-ben a 0 a hamis, minden más igaz.

stdbool.h header: Definiál egy elegánsabb bool típust, plusz a false és true literálokat.

Pontosság, műveletek, ábrázolás megintcsak megegyezik az egészekkel, mivel az egészek is viselkedhetnek logikai értékként.

Valós

float és double

Méretük lehet architektúrafüggő, de float legfeljebb akkora, mint double, ami legfeljebb akkora, mint long double.

Pontosság

Nem tudunk, csak diszkrét értékeket pontosan ábrázolni.

Valósok esetén annyit tudunk garantálni, hogy az értékkészlet határain belül minden értéket képesek vagyunk egy e relatív pontossággal ábrázolni, azaz minden a valós számhoz megadható az az a -hoz legközelebbi az adott valós típuson ábrázolható x érték, amelyre $(|x - a| \leq e)$ teljesül.

A pontossági problémák miatt a == és != operátorokkal vigyázni kell. Előfordulhat, különösen nagyon kicsi számok esetén, hogy az elvileg egyező értékek különböznek egymástól az ábrázolás miatt.

Megoldás lehet egy megadott toleranciával dolgozó, összehasonlítást végző makró használata: #define EQUALS(X, Y) (((X) > (Y)) ? ((X) - (Y)) : ((Y) - (X))) <= 1e-10)

A nagyon kicsi számokkal az aprobléma, hogy a tört részben sok vezető 0 lesz (pl. 0.000000000035), így kevés számjegy marad a valódi érték ábrázolására.

Számábrázolás

Egy valós értéket tároló memóriaterület három részre osztható lebegőpontos számábrázolás esetén:

- Eljelbit
 - 0: Pozitív szám
 - 1: Negatív szám

Mivel mindenkorban van, ezért nincs signed és unsigned módosítója a valós típusoknak.

- Tört
- Kitevő

Mindegyik fix hosszúságú biten van tárolva

Valós szám ábrázolása:

1. A számot kettes számrendszerbeli $1.m * 2^k$ normál alakra hozzuk.
2. m bináris számjegyeit tároljuk a tört részen
 - i. Meghatározza az ábrázolás pontosságát
3. k egy típusfüggő, b korrekciós konstanssal megnövelt értéket tároljuk a kitevőnek fenntartott helyen egész számként
 - i. Kitevő meghatározza az értéktartományt

[Binary 4 – Floating Point Binary Fractions 1 - YouTube](#)

- float : 32 biten tárolja a valós számokat

- 1 előjel bit
- 8 kitevő bit ($b = 2^7 - 1 = 127$ korrekciós értékkel)
- 23 tört bit
- `double` : 64 biten tárolja a valószínűszámokat
 - 1 előjel bit
 - 11 kitevő bit ($b = 2^{10} - 1 = 1023$ korrekciós értékkel)
 - 52 tört bit (azaz 1b dupla pontos)

12.2.2. Típusképzés

`typedef` kulcsszóval lehet típusokat elnevezni, saját típusokat definiálni.

```
typedef tipus uj_típusnev;
```

Például az igen hosszú `unsigned long long int` típust a `typedef unsigned long long int ulli;` utasítással `ulli`-nek kereszthetjük.

Ha utólag jövünk rá, hogy bizonyos helyeken elég mondjuk egy kisebb méretű típus használata, akkor hasznos, ha ezt az adott műveletet eleve egy általunk megadott type alias-al használtuk, mert akkor egy helyen elég átírni.

12.2.3. Összetett adattípusok

Azokat a típusokat, amelyek értékei tovább bonthatóak, illetve további értelmezésük lehetséges, **összetett adattípusoknak** nevezzük.

Pointer

Dinamikus változónak nevezzük azt a változót, amely bármely blokk aktivizálásától (végrehajtásától) független hozhatunk létre, illetve szüntethetünk meg. Az ilyen dinamikus változók megvalósításának eszköze lesz a pointer típus.

```
típus * változónév;
```

Pointer deklaráció két értelmezése (`int * p;`):

- `*p` egy `int` típusú (dinamikus) változó
- `p` egy `int*` típusú (azaz egy `int`-re mutató) változó

Az előbbi szerencsébb lehet, ugyanis a `*` a változóhoz tartozik, például `int * p, q;` esetén `q` egy szimpla `int` lesz.

Lehet pointer típust is definiálni: `typedef tipus *pointertípusnév;`

Változóhivatkozás

Szintaktikus egység, meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven, tehát egy kód részlet. Azaz egy `p = 3;` utasítás esetén maga a `p` jel(sorozat) a változóhivatkozás.

Változó

A program futása során a program által lefoglalt memóriaterület egy része, amelyen egy adott (elemi vagy összetett) típusú érték tárolódik. Például egy `int p;` deklaráció esetén a memoriában lefoglalódik egy `int` típusú érték tárolására alkalmas hely, és ezen a területen tárolódik majd a változó értéke.

Pointer, mint absztrakt adattípus

Dinamikus változóhivatkozáshoz tartozó változók a pointer típus műveleteivel hozhatók létre és szüntethetők meg.

Művelet megnevezése	Művelet leírása
<code>NULL</code>	Konstans, érvénytelen pointer érték, ha egy pointerhez ezt az értéket kapcsoljuk, akkor annak jelentése, hogy a pointerhez nem tartozik dinamikus változó.
<code>Létesít(← x : PE)</code>	Új <code>E</code> típusú dinamikus változó létesítése, amely elérhetővé válik az <code>x PE</code> (azaz <code>E</code> típusra mutató) pointer által.
<code>Értékadás(← x : PE , → y : PE)</code>	Az <code>x</code> pointer felveszi az <code>y</code> pointer értékét, azaz <code>x</code> és <code>y</code> a művelet végrehajtása után ugyanarra a dinamikus változóra mutatnak.
<code>Törlés(↔ x : PE)</code>	Az <code>x</code> által hivatkozott dinamikus változó törlésre kerül. Az <code>x</code> ezen túl nem hivatkozik semmiről.
<code>Dereferencia(→ x : PE) : E</code>	Visszaad egy <code>E</code> típusú változó hivatkozást, amivel a dinamikus változót el tudjuk érni, amivel arra hivatkozhatunk.
<code>Egyenlő(→ p : PE , → q : PE): bool</code>	Összehasonlítja, hogy <code>p</code> és <code>q</code> ugyanarra a dinamikus változóra hivatkoznak-e (beleértve, hogy egyik sem hivatkozik változóra)!
<code>NemEgyenlő(→ p : PE , → q : PE): bool</code>	Összehasonlítja, hogy <code>p</code> és <code>q</code> más-más dinamikus változóra hivatkoznak-e (beleértve, hogy legfeljebb az egyik nem hivatkozik változóra)!

←: kimenő paraméter

→: bemenő paraméter

↔: be- és kimenő módú paraméter

Pointer, mint virtuális adattípus

Művelet absztrakt szinten	Művelet virtuális megvalósítása
NULL	NULL
Létesít(← X : PE)	X = malloc(sizeof(E));
Értékadás(← X : PE, → Y : PE)	X = Y
Törlés(↔ X : PE)	free(X); X = NULL;
Dereferencia(→ X : PE) : E	*X
Egyenlő(→ p : PE, → q : PE) : bool	p == q
NemEgyenlő(→ p : PE, → q : PE) : bool	p != q
Cím(→ v : E, ← p : PE)	p = &v;

A & operátorral (memória cím lekérés) lokális változók esetén vigyázni kell, ugyanis azok az adott scope végén felszabadulnak (stack-en vannak), az oda mutató pointer így már felszabadított területre fog mutatni.

Létrehozáson és megszüntetésen kívül minden megvalósítható közvetlenül a C nyelvi elemeivel.

Ezt a kettőt pedig a `stdlib.h` headerből elérhető `malloc` és `free` utasításokkal tudjuk megvalósítani.

Pointer mérete

32 bites architektúran 4 byte, 64 bitesen 8 byte-on tartalmazza a hozzá tartozó dinamikus változó kezdőcímét.

`void*`

A `void*` egy speciális, úgynevezett típustalan pointer. Az ilyen típusú pointerek "csak" memóriacímek tárolására alkalmasak, a dereferencia művelet alkalmazása rájuk értelmetlen.

Viszont minden típusú pointerrel kompatibilisek értékadás és összehasonlítás tekintetében.

Valójában a `malloc` függvény visszatérési értéke is `void*`, és ez kasztolódik impliciten a megfelelő típusú mutatóvá.

Függvény argumentumok módjainak kezelése

Alapvetően az érték szerint átadott argumentumok bemeneti argumentumok.

Kimenő, vagy be- és kimenő argumentumos keseén pointereket kell alkalmaznunk.

Például a következő függvény:

```
void csere (int x, int y) {
    int m;
    m = x;
    x = y;
    y = m;
}
```

nem végzi el ténylegesen a cserét, ugyanis érték szerint kapja a paramétereket, így csak a stack-re kerülő két lokális változót fogja cserélni, ami csak a saját scope-ján belül történik meg.

Módosítani kell ezt a függvényt, hogy ne érték szerint várja az értékeket:

```
void csere (int *x, int *y) {
    int m;
    m = *x;
    *x = *y;
    *y = m;
}
```

Tömb

Rögzített számú, ugyan olyan typusú elemek sorozata

Művelet megnevezése	Művelet leírása
Kiolvas ($\rightarrow A : T, \rightarrow i : I, \leftarrow x : E$)	Az A tömb i . értékét kiolvassa és eltárolja az x változóban.
Módosít ($\leftrightarrow A : T, \rightarrow i : I, \rightarrow x : E$)	Az A tömb i . értékét módosítja az x értékkal.
Értékkapás ($\leftarrow A : T, \rightarrow X : T$)	Az A tömb felveszi az X tömb típusú kifejezés értékét.

 T : tömb típus E : tömb-beli érték típusa I : index típus (0, ..., $n - 1$ intervallumból egy érték)

Tömb, mint virtuális adattípus

A változó neve kiegészül egy szögletes zárójel párral, ami között megadjuk a tömb méretét, azaz azt az elemszámot, amennyi elemet el szeretnénk tárolni a tömbben.

`típus változónév[elemszám];`

Természetesen a tömb típussal is definiálhatunk új típust:

`typedef típus újnév[elemszám];`

Művelet absztrakt szinten	Művelet virtuális megvalósítása
Kiolvas ($\rightarrow A : T, \rightarrow i : I, \leftarrow x : E$)	$x = A[i];$
Módosít ($\leftrightarrow A : T, \rightarrow i : I, \rightarrow x : E$)	$A[i] = x;$
Értékkapás ($\leftarrow A : T, \rightarrow X : T$)	Nincs direkt megvalósítás! Használható a <code>memcpy()</code> , vagy a C ¹¹ szabvány után a biztonságosabb <code>memcpy_s()</code> függvény.

Fontos, hogy C-ben nincs indexhatár ellenőrzés, azzaz alul- vagy felülindexelhetünk egy tömböt, és ez látszólag nem fog problémát okozni, azonban futási hibákat eredményezhet amiatt, hogy a memória egy olyan területét érjük el, amin más adat van.

Több dimenziós tömbök

A műveletek nagyon hasonlóan definiálhatóak, de I itt indexek egy (i_0, \dots, i_k) vektora (k dimenziós tömb esetén).

Kiolvasás: $x = A[i_1 \dots i_k];$ Módosítás: $A[i_1 \dots i_k] = x;$ Értékkapás: `memcpy()` vagy `memcpy_s()`

Tömb inicializálása

```
int t[][3] = {
    { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }
};

int t1[4][3] = {
    { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }
};

int t2[][3] = {
    { 1 }, { 4 }, { 7 }, { 9 }
};
```

Üresen hagyott méreteket (`[]`)-et az inicializációból kikövetkezteti a nyelv.

Inicializálatlan cellák értéke 0

Tömb, mint fizikai adattípus (hogy néz ki a memóriában)

Tömb típusú változó számára történő helyfoglalás azt jelenti, hogy minden tömbelem, mint változó számára memóriát kell foglalni. Feltehetjük, hogy egy adott tömb változóhoz a tömbelemek számára foglalt tárterület összefüggő mezőt alkot.

A tömb azonosítója valójában egy pointer az első elemre.

Az indexelés pedig nem más, mint egy shorthand arra, hogy az első elemre mutató pointerrel, és egy offsettel érünk el egy bizonyos elemet.

Egy dimenziós esetben egyszerűen számolható ez az offset, hiszen csak az index és egy elem méretének szozata adja meg.

Rekord

Különböző típusú, de logikailag összefüggő értékek kezelésére alkalmas.

Szorzat-rekord absztrakt adattípus

Művelet megnevezése	Művelet leírása
$Kiolvas_i(\rightarrow A : T, \leftarrow x : T_i)$	Az $A : T$ típusú szorzat rekord i . mezőjét kiolvasó művelet, amely az i . mező típusának megfelelő x változóba teszi a kiolvasott mező értékét.
$Módosít_i(\leftrightarrow A : T; \rightarrow x : T_i)$	Az $A : T$ típusú szorzat rekord i . mezőjét módosító (beállító) művelet, amely az i . mező típusának megfelelő x változót értékül adja az i . mezőnek.
$Értékkadás(\leftarrow A : T; \rightarrow x : T)$	Az $A : T$ típusú szorzat rekord változónak értékül adja az $x : T$ típusú szorzat rekord változót.

Szorzat-rekord virtuális adattípus

```
typedef struct T {  
    T1 M1;  
    ...  
    Tk Mk;  
} T;
```

Az első T a struktúra neve, ami akár el is maradhatna, a második T viszont az újonnan, a `typedef` által bevezetett típus neve.

Ha a struktúrának nem adunk nevet, és nem is definiálunk hozzá egy új típust, akkor mindenkor, amikor adott elemeket tartalmazó struktúra változót szeretnénk létrehozni, le kell írni a teljes struktúra definícióját. Ez persze már túl hosszú ahhoz, hogy ezt többször megadjuk. Így ha van neve a struktúrának, akkor a következő esetben már `struct T` is elég a megadásához, ha pedig típust is képeztünk belőle, akkor a típusképzés után már a T típusazonosító is egyértelműen hivatkozza az adott struktúrát.

A fenti típusképzésben az $M1 \dots Mk$ azonosítókat mezőazonosítóknak (tagnak, membernek) hívjuk és lokálisak a típusképzésre nézve.

Művelet absztrakt szinten	Művelet virtuális megvalósítása
$Kiolvas_i(\rightarrow A : T, \leftarrow x : T_i)$	$x = A.M_i$
$Módosít_i(\leftrightarrow A : T; \rightarrow x : T_i)$	$A.M_i = x$
$Értékkadás(\leftarrow A : T; \rightarrow x : T)$	$A = x$

Szorzat-rekord fizikai adattípus

Mivel a struktúra egy összetett adat, így az, hogy ő konkrétan mekkora részt foglal a memóriában függ attól, hogy a benne levő típusok megkorák, illetve tudnunk kell azt, hogy ezek hogyan tárolódnak.

Összefüggő memóriaterületen tárolódik.

```
sizeof(E) = sizeof(T1) + ... + sizeof(Tk) + igazítás.
```

Valamennyi mező a deklaráció sorrendjében egymást követő, növekvő memóriacímen kezdődik. Az első mező memóriacíme megegyezik a teljes `struct` típusú érték címével.

Bitmezők

```
struct {  
    unsigned int flag1 : 1;  
    unsigned int flag2 : 1;  
    unsigned int flag3 : 2;  
} jelzok;  
jelzok.flag1 = jelzok.flag2 = 0;  
jelzok.flag3 = 1;  
if (jelzok.flag1 == 0 && jelzok.flag3 == 1) {  
    /* ... */  
}
```

Meg lehet adni, melyik mező hánny bites legyen.

A \rightarrow operátor

Legyen tp egy struktúra típusú változóra mutató pointer. Ilyenkor ha a struktúra egy mezőjére szeretnénk hivatkozni, azt a `(*tp).meznev` alakban tudunk.

Mivel ez egy gyakori eset, egy egyenértékű, egyszerűbb megvalósítás: `tp->meznev`

Elképzelhető, hogy úgy szerethnénk egységesen hivatkozni adatokra, hogy a konkrét megvalósításban még nincs fogalmunk arról, hogy a program adott futásakor milyen típusú adatot kapunk az adott ponton.

Egyesített-rekord absztrakt adattípus

Művelet megnevezése	Művelet leírása
$Változat(\rightarrow A : T ; \leftrightarrow V : T_0)$	Változat kiolvasása. A művelet végrehajtása után $V = c_i$, ha $A = (c_i, a)$.
$Kiolvasi(\rightarrow A : T , \leftarrow x : T_i)$	Adott $i \in 1 \dots k$ -ra az A rekord i . komponensének kiolvasása adott $x : T_i$ típusú változóba. A művelet végrehajtása után $x = a$, ha $A = (c_i, a)$. A művelet hatástan, ha A első komponense nem c_i .
$Módosít_i(\leftrightarrow A : T ; \rightarrow x : T_i)$	Adott $i \in 1 \dots k$ -ra az A rekord i . komponensének módosítása adott $x : T_i$ típusú értékre. A művelet végrehajtása után $A = (c_i, x)$.
$\Értékkadás(\leftarrow A : T ; \rightarrow X : T)$	Az $A : T$ típusú szorzat rekord változónak értékül adja az $X : T$ típusú szorzat rekord változót.

Egyesített-rekord virtuális adattípus

```
typedef union T {  
    T1 M1;  
    ...  
    Tk Mk;  
} T;
```

Ebben a típusképzésben is az M_1, \dots, M_k azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve. Illetve az `union` szó utáni névre és a típus nevére hasonlóak igazak, mint a `struct` esetében.

Megjegyzendő, hogy a C megvalósításában (megfelelő környezetben) minden hivatkozhatunk bármelyik mezőre, függetlenül attól, hogy az `union` aktuálisan melyik mező értékét tárolja.

Önmagában a C `union` típusképzésében nem adhatunk meg változati mezőazonosítót (T_0), így nincs lehetőségünk az aktuális változatról információ tárolására. Éppen ezért, ha ez a változati mezőazonosítót is el szeretnénk tárolni, akkor kombináljuk a `struct` és `union` lehetőségeit:

```
typedef struct T {  
    T0 Milyen;  
    union {  
        T1 M1;  
        ...  
        Tk Mk;  
    };  
} T;
```

Művelet megnevezése	Művelet leírása
$Változat(\rightarrow A : T ; \leftrightarrow V : T_0)$	$V = A.Milyen$
$Kiolvasi(\rightarrow A : T , \leftarrow x : T_i)$	<code>if (A.Milyen == ci) { x = A.Mi; }</code>
$Módosít_i(\leftrightarrow A : T ; \rightarrow x : T_i)$	<code>{ A.Milyen = ci; A.Mi = x; }</code>
$\Értékkadás(\leftarrow A : T ; \rightarrow X : T)$	$A = X$

Egyesített-rekord fizikai adattípus

Láttuk, hogy az `union`-on belül egy adott időpillanatban egy mező lesz az érvényes, így egyszerre nem szükséges minden mezőt eltárolni. Így ha a legnagyobb mező méretének megfelelő memória foglalódik az `union` számára, abban valamennyi mezőn tárolt érték elfér majd.

Azaz egy `T union` típus mérete a következő módon alakul: `sizeof(T) = max{sizeof(T1), ..., sizeof(Tk)}`.

A `struct` és `union` típusok inicializálása

Mind a `struct`, mind az `union` változók kaphatnak kezdőértéket, az adattagok értékeit a `{}`-ek között kell felsorolni.

Az értékek szimpla felsorolásával `struct`-ban az adattagok sorrendjében kell megadni az egyes tagok értékét, de nem kötelező minden, az `union` esetében viszont csak az első mező típusának megfelelően inicializálható.

A C⁹⁹ szabvány lehetővé teszi tetszőleges mezők inicializását a mezők neveit felhasználva. Ekkor a `{}` zárójelek között az egyes értékek elő a `.mezo = -t` írva jelezhetjük, hogy az adott érték mely mező kezdőértéke lesz:

```
Idom i = {
    .UU = {
        .U1 = {
            .C = 3, .B = 4, .A = 5
        }
    },
    .Fajta = haromszog
};
```

13. Rendszerfejlesztés 1.

13.1. 1. Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei.

13.1.1. A szoftverfejlesztés folyamata

- **A szofverfolyamat:** tevékenységek és kapcsolódó eredmények, amely során elkészítjük a szoftvert
- A folyamat összetett, kreatív munka kell hozzá
- Csak korlátozott automatizálás
- Nincs ideális folyamat, viszont modellek léteznek
- minden folyamat egyedi, sokszor kombinációkat használnak

Folyamat szerepe:

- Szoftverfejlesztés = folyamat +menedzsment + technikai módszerek + eszközök használata
- Minőségi szoftver biztosítéka
- Folyamat meggyőzíti hogy elveszítik az uralmat a projekt felet
- Adaptálás adott projekthez és környezethez

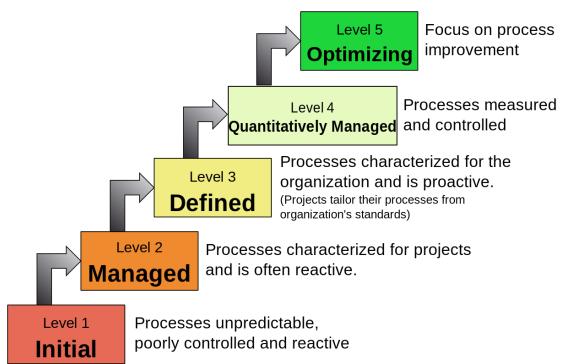
Folyamat elemek:

- Fő elemek
 - Feladatok, termékek
 - Határidők, átadandók
- Kiegészítő elemek
 - Projektmenedzsment
 - Konfiguráciomenedzsment
 - Dokumentáció
 - Minőségbiztosítás, kockázatmenedzsment
 - Mérés

Folyamat fejlettsége

- Egy szervezetnél alkalmazott folyamat minősítése meghatározhatja a megrendelők bizalmát
- SEI CMM(I) (Capability Maturity Model Integration)
- **Szintjei:**
 - Kezdeti
 - Reprodukálható
 - Definiált
 - Ellenőrzöt
 - Optimalizált

Characteristics of the Maturity levels



A szoftverfolyamat fázisai

- minden folyamatnak elemei:
 - **Specifikáció**: szoftver funkcionálitása, megszorítások („mit”)
 - **Fejlesztés**: tervezés és implementáció specifikáció alapján („hogyan”)
 - **Verifikáció és Validáció**: fejlesztés megfelel-e a specifikációnak és a követelményeknek
 - **Evolúció**: változás kezelése, szoftver „utóélete”
- **Specifikáció**
 - Szoftver definiálása
 - Milyen funkciókat, szolgáltatásokat követelünk meg a rendszertől
 - Követelménytervezés
 - Kritikus szakasz: itt a legkisebb a változtatások költsége
 - Eredmény: követelményspecifikáció dokumentum, esetleg prototípusok
 - Végfelhasználónak: magas szintű
 - Fejlesztőknek: részletes, technikai
- **Követelménytervezés fázisai**
 - Megvalósíthatósági tanulmány (feasibility study)
 - Költséghatékonyúság ellenőrzése
 - Követelmények feltárása és elemzése:
 - Rendszermodellek, prototípusok
 - Követelményspecifikáció: Egységes dokumentum
 - Követelmény validáció
- **Tervezés**
 - Szoftver struktúrája, adatok, interfészek
 - Rendszermodellek különböző absztrakciós szinteken
 - **Tevékenységei**:
 - Architektúra tervezés: alrendserek meghatározása
 - Absztrakt specifikáció: alrendserek szolgáltatásai
 - Interfész tervezés: alrendserek között
 - Komponens tervezése
 - Részletek: adatszerkezetek, algoritmusok
 - Gyakorlati folyamatok speciálisan definiálják ezeket
- **Tervezési módszerek**
 - Ad hoc (átfogó rendezés nélkül)
 - Strukturált
 - Structured Design (SD)
 - SSADM

- Jackson

- Objektumorientált

- pl.: UML (Unified Modeling Language, szabványos, általános célú modellező nyelv, üzleti elemzők, rendszertervezők, szoftvermérnökök számára)

- Közös: grafikus rendszermodellek, szabványos jelölésrendszer, CASE (Computer Aided Software Engineering) támogatás

- **Implementáció**

- Programozás és nyomkövetés
- kritikus rendszerekben részletes tervezés alapján
- Programozás (kódolás): adottság kell hozzá, személyes technikák, stílusok
- Minőségbiztosítás érdekében kódolási stílust lehet megkövetelni
- Nyomkövetés (debugging): hiba lokalizálás, eltávolítás, újratesztelek, programszöveg manuális vizsgálata, eszközökkel támogatás

- **Szoftver validáció**

- Verifikáció és validáció (V & V): Rendszer megfelel-e a specifikációnak, és a megrendelő elvárásainak
- Tesztelés különböző szinteken történik, inkrementálisan
 - Egység tesztelése (unit test): komponensek független tesztelése, programozó feladata
 - Modul tesztelése: függő és kapcsolódó komponenseket együtt, szintén a programozó feladata
 - Alrendszer tesztelése: például interfések illeszkedése, független tesztelő csapat feladata
 - Rendszer tesztelése: előre nem várt kölcsönhatások felfedezése, validáció specifikációhoz tesztadatokon, független tesztelő csapat
 - Átvételi tesztelés: megrendelő adataival, valós környezetben, alfa tesztelés (fejlesztő vagy teszt csapat végzi)
 - Béta tesztelés: potenciális vásárlók által tesztelt, előre nem látható hibák keresése

- **Evolúció**

- Szoftver flexibilitás miatt nagy, összetett rendszerek születnek
- Változás bár költséges, de
 - Eredményesebb meglévő rendszerekből kialakítani az újat
 - Kevés a teljesen új szoftver
 - Egy szoftver sosincs kész
- Követelményspecifikáció után a meglévő rendszereket kiértékeljük
 - Manuális vizsgálat
 - Automatikus elemzés (reverse engineering)
 - Kimenet: dokumentáció, magasabb szintű rendszermodell
- Rendszermódosítások után jön létre az „új” rendszer
 - Újratervezés (re-engineering)
 - Folyamatos evolúció (roundtrip engineering)

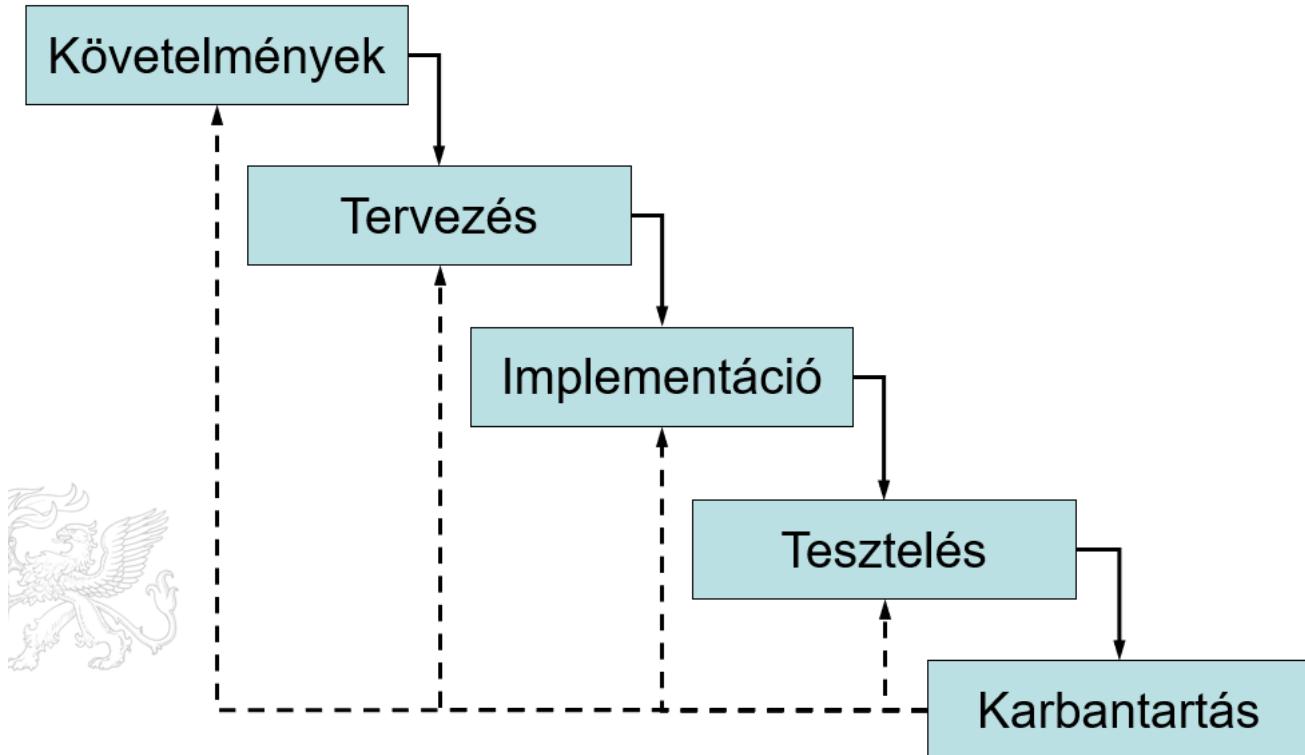
13.1.2. A folyamat modelljei

Kategóriák

- Triviális: lineáris, vízesés
 - Tevékenységek különálló fázisok
 - Iteratív modellek: prototípus, vízesés, RAD
- Evolúciós: prototípusok gyors gyártása, finomítása
- Formális módszerek: matematikai rendszer, transzformációk
- Újrafelhasználható, komponens alapú

Vízesés modell

- Első publikált, „klasszikus” modell (életciklus modell)
- Lényegében egy szekvenciális modell
 - Fázisok lépcsősen kapcsolódnak
 - Visszacsatolás is van
- Fázisok kimenetei teljesen el kell, hogy készüljenek, mielőtt továbbmegyünk
 - A hibákat összegyűjtik a fázisok végén
 - Javításra a folyamat végén van lehetőség
 - Iteráció közvetve van jelen
- Ha jó a specifikáció, akkor működőképes



- Problémái
 - Ritkán van egyszerű lineáris fejlesztés
 - Követelményeket nehéz pontosan specifikálni a legelején
 - A megrendelő csak a legvégén látja meg először a terméket
 - Sok hiba ekkor derül ki, melyek javítási költsége nagy
- Előnye: megrendelő könnyebben tud megállapodni, mert a specifikáció pontos (Viszont nehezen módosítható szoftver alakul ki)

Iteráció, inkrementalitás

- Folyamat iterációja elkerülhetetlen
 - Ha a követelmények változnak, akkor a folyamat bizonyos részeit is változtatni kell
- Iteráció szélsőséges esetei:
 - Vízesés modellnél minimális lehetőség
 - Prototípus (vagy evolúciós) modellnél minimális a specifikáció, fejlesztésben sok iteráció van, és menet közben alakul ki a végleges specifikáció

Evolúciós fejlesztés

- Prototípus modell: az evolúciós fejlesztés egy szélsőséges iteratív+inkrementális példája
 - Durva specifikáció megrendelő részéről
 - Ezután gyors fejlesztés, eredménye prototípus
 - Prototípus kiértékelése után követelményspecifikáció újraíródik

- Sok-sok iteráció a végtermékig
- Nagy dilemma: a prototípusból lesz-e a végtermék, vagy az csak eldobható
- Intenzív kapcsolat kell a megrendelővel
- Kész komponensek alkalmazása előnyös
- Prototípus modell problémái:
 - A megrendelő azt gondolja, hogy a prototípus kész rendszer, nehéz ellenállni, hogy ne használja
 - Gyors fejlesztés miatt minőség romolhat, kevésbé hatékony megoldások alkalmazása miatt, amik beépülhetnek a végső rendszerbe
 - Megrendelő sokszor vállalja a rizikókat, mert:
 - Szeret „belelátni” a fejlesztésbe
 - Kezdetben pontosan tudja, hogy mit szeretne, de a részletekről fogalma sincs
 - Hibrid megoldások kellenek, vízesés modellel

Inkrementális modell

- Vízesés és evolúciós fejlesztés kombinációja (robosztusság és felxibilitás)
- Nagy körvonalakban specifikáljuk a rendszert
 - „Inkremensek” meghatározása
 - Funkcionalitásokhoz prioritásokat rendelünk
 - Magasabbakat előbb kell biztosítani
- Architektúrát meg kell határozni
- További inkremensek pontos specifikálása menet közben történik
- Egyes inkremensek kifejlesztése történhet akár különböző folyamatokkal is (vízesés vagy evolúciós, amelyik jobb)
- Az elkészült inkremenseket akár szolgálatba is lehet állítani
 - Tapasztalatok alapján lehet meghatározni a következő inkremenseket
- Az új inkremenseket integrálni kell a már meglévőkkel
- Előnyei:
 - A szoftver már menet közben használható
 - Korábbi inkremensek prototípusként használhatók, a későbbi követelmények pontosítása érdekében
 - Ha határidő csúszás van kilátásban, inkrementális modell bevethető
 - Teljes projekt nem lesz kudarcra ítélezve, esetleg csak egyes inkremensek
 - A legfontosabb inkremensek lesznek többször tesztelve (mivel azokkal kezdtük a megvalósítást)
- Hátrányai:
 - Megfelelő méretű inkremensek meghatározása nem triviális feladat
 - Ha túl kicsi: nem működőképes
 - Ha túl nagy: elveszítjük a modell lényegét
 - Bizonyos esetekben számos alapvető funkcionálitást kell megvalósítani
 - Egész addig nincs működő inkremens
 - Csak akkor pörög be a rendszer, ha minden összeállt

eXtreme Programming (XP)

- Szélsőséges inkrementális modell
 - Nagyon kis funkcionálitású inkremensek
 - Megrendelő intenzív részvételle
- Programozás csoportos tevékenység (többen ülnek egy képernyő előtt)
- Az utóbbi időben sok kiegészítés készül, sajnos kezdi kinőni az eredeti elképzést

- Sok támadója van

RAD

- Rapid Application Development
- Extrém rövid életciklus (Működő rendszer 60-90 nap alatt)
- Vízesés modell „nagysebességű” adaptálása
 - Párhuzamos fejlesztés
 - Komponens alapú fejlesztés
- Fázisok:
 - Üzleti modellezés: milyen információk áramlanak funkciók között
 - Adatmodellezés: finomítás adatszerkezetekre
 - Adatfolyam processzus: adatmodell megvalósítása
 - Alkalmazás generálás: 4GT (negyedik generációs technikák) alkalmazása, automatikus generálás, komponensek
 - Tesztelés: csak komponens tesztelés
- Problémái:
 - Nagy emberi erőforrásigény
 - Fejlesztők és megrendelők intenzív együttműködése
 - Nem minden típusú fejlesztésnél alkalmazható
 - Modularizálhatóság hiánya problémát jelenthet

Spirális modell

- Olyan evoluciós modell, amely kombinálja a prototípus modellt a vízesés modellel
- Inkrementális modellhez hasonló, csak általánosabb megfogalmazásban
- Nincsenek rögzített fázisok, mindig egyedi modellek
- Más modelleket ölelhet fel, pl.:
 - Prototípuskészítés pontatlan követelmények esetén
 - Vízesés modell egy későbbi körben
 - Kritikus részek esetén formális módszerek
- A spirál körei a folyamat egy-egy fázisát reprezentálják
- minden körben a kimenet egy „release” (modell vagy szoftver)
- Körök céljai pl.:
 - Megvalósíthatóság (elvi prototípusok)
 - Követelmények meghatározása (prototípusok)
 - Tervezés (modellek és inkremensek)
 - (javítás, karbantartás, stb.)
- A körök szektorokra oszthatók (3-6 db)
 - 4 szektorral:
 - Célok kijelölése
 - Kockázat becslése és csökkentése
 - Fejlesztés és validálás
 - Következő spirálkör megtervezése
 - 6 szektorral:
 - Kommunikáció megrendelővel
 - Tervezés

- Kockázatelemzés
- Fejlesztés
- Megvalósítás és telepítés
- Kiértékelés megrendelő részéről

Újrafelhasználás-orientált

- Komponens alapú fejlesztés
 - Elérhető, újrafelhasználható komponensek
 - Ezek integrációja
- Hagyományos modellekkel megegyezik
 - Követelményspecifikáció és validáció
- Közte levő fázisok eltérnek
 - Komponens elemzés
 - Követelménymódosítás
 - Rendszertervezés újrafelhasználással
 - Fejlesztés és integráció
- Előnyök:
 - Kevesebb fejlesztendő komponens, csökken a költség
 - Gyorsabb leszállítás
- Hátrányok
 - Kompromisszumok követelményekkel szemben
 - Evolúció során a felhasznált komponensek új verziói már nem integrálhatók
- Objektumorientált paradigmá jó alap
 - UML használata
 - Rational Unified Process (RUP) egy iteratív, inkrementális és komponens alapú folyamat

Formális módszerek

- Vizesés modellre hasonlít
 - Specifikáció: formális, matematikai apparátus
 - Kidolgozás: ekvivalens transzformációk
 - Verifikáció: hagyományos értelemben nem szükséges
- Kisebb lépésekben áll, amelyek finomítják az egyes formális modelleket, így könnyebb a formális bizonyítás
- Speciális területeken alkalmazható (Pl. kritikus (al)rendszerknél, ahol elvárt a bizonyítottság)
- Kölcsönhatások nem minden formalizálhatók

Cleanroom módszer

- Fejlesszünk (bizonyítottan) hibátlanul és akkor nem kell tesztelni
- Csak rendszertesztes kell, modulhelyesség bizonyított
- Az egyik legismertebbek formális módszer
- Inkrementális fejlesztésen alapul
- Fejlesztőszerek egyszerűbbek, szigorúbbak (Pl. csak strukturált programnyelvek)
- Dobozokkal reprezentálják a rendszert
- Képzett, elkötelezetett tervezők

4GT

- Negyedik generációs technikák

- Magas szintű reprezentáció (absztrakció)
 - 4G (vizuális) nyelvek, grafikus jelölés
 - Automatikus kódgenerálás
- Vizuális eszközök: adatbázis lekérés, riportgyártás, adatmanipuláció, GUI, táblázatkezelés, HTML-oldalak, web, stb.
- Előnyei:
 - Rövidebb fejlesztési idő
 - Jobb produktivitás
 - Kis és közepes alkalmazásoknál jó
- Hátrányai:
 - Vizuális nyelvet nem könnyebb használni
 - Generált kód nem hatékony
 - Karbantarthatóság rosszabb
 - Nagy alkalmazásoknál nem előnyös
- Komponens alapú technikával alkalmazva még jobb

Egyéb (aktuális) modellek

- Kliens/szerver modell
 - Kliens adatokat/szolgáltatást kér, szerver szolgáltatja
- Web fejlesztés
 - Hagyományos módszerek + kliens/szerver + 4GT + OO + komponensek
 - Web tartalom és design tervezés is ide tartozik
- Nyílt forráskódú fejlesztés
 - Ad hoc fejlesztés
 - Fejlesztési ütemezés, költségvetés nem definiált
 - Nem strukturált folyamat
 - Közösségi ellenőrzés
 - Bizalmatlanság megbízhatóság terén
 - Nyílt forráskód, bárki mérheti a minőséget és javíthat
 - Nem feltétlenül jobb a kereskedelmi termék
 - Sokszor ingyenes licensz

13.2. 2. Projektmenedzsment. Költségbecslés, szoftvermérés

13.2.1. Projektmenedzsment

Tényezők (4P)

- **Munkatársak (people)** – a sikeres projekt legfontosabb tényezői
- **Termék (product)** – a létrehozandó termék
- **Folyamat (process)** – a feladatok, tevékenységek halmaza a munka elvégzése során
- **Projekt** – minden olyan tevékenység, ami kell ahhoz, hogy a termék létrejöjjön

Projekt sikertelenségének okai

- Nem reális a határidők megválasztása
- A felhasználói követelmények változnak
- A szükséges ráfordítások alulbecslése
- Kockázati tényezők
- Technikai nehézségek

- A projekt csapatban nem megfelelő a kommunikáció
- A projekt menedzsment hibái

Emberek menedzsmentje

- Szoftverfejlesztő szervezet legnagyobb vagyona az emberek
 - Szemelmi töke
 - Lehető legjobban kamatozzon!
- Sok projekt bukásának legfőbb oka a rossz humánmenedzsment
- Egyik legfontosabb feladat az emberek motivációja
 - Szociális szükségek, megbecsülés, önmegvalósítás igénye

Csoportmunka

- Valódi szoftvereket 2-1000 fős csapatok készítik (team)
- Hatékony együttműködés fontos
 - Csapatszellemet kell kialakítani (csoport sikere fontosabb mint az egyéné)
 - Csoportépítés (pl.: szociális tevékenységek)
- Munkakörnyezet fontos (közös és privát területek fontosak)
- Befolyásoló tényezők:
 - Csoport összetétele
 - egymást kiegészítő személyiségek
 - nemkívánatos vezető végzetes lehet
 - Csoportösszetartás (pl.: csoportos programozás)
 - Csoportkommunikáció
 - Csoport szerkezete
 - informális szervezés
 - vezető programozó-csoport: kell tartalék programozó és adminisztrátor is

Csapatfelépítés szempontjai

- A megoldandó probléma nehézsége
- A programok mérete (LOC vagy funkciótartam)
- A team működésének időtartama
- A feladat modularizálhatósága
- A létrehozandó rendszer minőségi és megbízhatósági követelményei
- Az átadási határidők szigorúsága
- A projekt kommunikációs igénye
- Csapatfelépítés lehet,
 - **Zárt forma** – hagyományos strukturális felépítés
 - **Véletlenszerű forma** – laza szerkezet, egyedi kezdeményezések a döntők
 - **Nyitott forma** – a zárt és a véletlenszerű paradigmák előnyeinek kombinálása
 - **Szinkronizált forma** – az adott probléma felosztása szerint történik a team szervezése, egyes csoportok között kevés kommunikáció van

Emberek kiválasztása

- Különböző tesztekkel történhet
 - Programozási képesség
 - Pszichometrikus tesztek
- Sok tényező: alkalmazási terület, platform, programozási nyelv, kommunikációs készség, személyiségek, stb.

- Szakmai karrier megállhat egy szinten, ha vezetői szerepkört kap
 - Azonos értékű kell hogy legyen a szakember és a vezető!

Termék (product)

- Szoftver hatásköre
 - Környezet
 - Input-output objektumok
- Probléma dekompozíció

Folyamat (process)

- A megfelelő folyamat kiválasztása
- Előzetes projekt terv
- 4CPF (common process framework)
 - Felhasználói kommunikáció
 - Tervezés
 - Kockázat analízis
 - Fejlesztés
 - Release
 - Felhasználói kiértékelés

13.2.2. Szoftverköltség becslése

- Projekt tevékenységeinek kapcsolódása a munka-, idő- és pénzköltségekhez
- Becslésekkel lehet és kell adni
 - Folyamatosan frissíteni
- Projekt összköltsége:
 - Hardver és szoftver költség karbantartással
 - Utazási és képzési költség
 - Munkaköltség

Projekt

- W5HH módszer
 - Miért fejlesztjük a rendszert? (why?)
 - Mit fog csinálni? (what?)
 - Mikorra? (when?)
 - Ki a felelős egy funkcióért? (who?)
 - Hol helyezkednek el a felelősök? (where?)
 - Hogyan megy a technikai és menedzsment munka? (how?)
 - Mennyi erőforrás szükséges? (how much?)
- Szoftver projekt tervezésénél meg kell becsülni:
 - Mennyi pénz?
 - Mennyi ráfordítás?
 - Mennyi idő?

Munkaköltség

- Legjelentősebb
- Fejlesztők fizetése, kisegítő személyzet fizetése, bérleti díj, rezsi, infrastruktúra, szórakozás, adó, stb.

Termelékenység

- Ipari rendszerben a legyártott egységek száma / emberórák
- Szoftvernél nehézkes
 - Egyik kód hatékony, a másik karbantartható, stb.
- Ezért mérik a szoftver valamely jellemzőjét (metrika)
- Két típus:
 - Méret-alapú (pl. programsorok száma)
 - Funkció-alapú (funkciót, objektumpont)

Méret alapú mérés

- LOC = Lines Of Code
- Több technika
 - Csak nem üres sorok
 - Csak végrehajtható sorok
 - Dokumentáció mérete
- Félrevezető lehet (különböző nyelveken ugyanaz a funkcionálítás mint)

Funkciót pont számítás

- Jobb, de nehezebben határozható meg
- Nyelv független
- Rendszer funkcionálitásának „mennyisége”
- Több programjellemző súlyozott kombinációja
 - Külső bemenetek és kimenetek
 - Felhasználói interaktivitás
 - Külső interfések
 - Használt állományok
- Vannak további módosító tényezők
 - Projekt összetettsége
 - Teljesítmény
 - Ezek nagyon szubjektívek
- Sorok átlagos száma
 - Assembly: 200-300 LOC/FP
 - 4GL(negyedik generációs prog nyelv): 22-40 LOC/FP

Objektumpontok

- Nem az osztályok vagy objektumok száma!
- 4GL nyelvekhez
- Súlyozott becslés:
 - Megjelenítendő képernyők száma (1-3 pont)
 - Elkészített jelentések száma (2-8 pont)
 - 3GL modulok száma (modulonként 10 pont)

Dekompozíciós technikák

- Szoftver méret (ennek meghatározása a legfontosabb)
 - Fuzzy-logic: approximációs döntési lépések
 - FP méret
 - Szabványos komponens méretek használata

- Változás alapú méret (létező komponenseket módosítunk)

- Probléma alapú becslés

- Funkciókra való bontás a lényeges
- „Baseline” metrikák (alkalmazás specifikus)
- LOC becslés - dekompozíció a funkciókra
- FP becslés - dekompozíció az alkalmazás jellemzőire koncentrál

- 4Folyamat alapú becslés

- Meghatározzuk a funkciókat
- minden funkcióhoz megadjuk a végrehajtandó feladatokat
- A feladatokra becsüljük a feladatok költségeit

Tapasztalati becslés modellek

- Erősen alkalmazás-függő
- Több féle számítási modell, pl:

► Erősen alkalmazás-függő

► Általános alak

- $E = A + B * (x)^C$
- ahol $x = \text{LOC}$ vagy FP
- E: emberhónap
- A,B,C: konstansok

- COCOMO modell (Constructive Cost Model)

- Iparban a COCOMO 2 használt
- Regressziós modell, LOC-on alapul
- Feladatok nehézsége be van sorolva, feladathoz szükséges idő megbecsülve

- Dinamikus modell

- $E = [\text{LOC} * B^{0.333} / P]^{3/4} * (1/t^4)$
- E: pm ráfordítás
- B: speciális képzettségi igény
- P: produktivitási paraméterek
- t: projekt időtartam

A szoftverminőség

- mindenki célja: termék vagy szolgáltatás minőségének magas szinten tartása
- Nem egyszerű definiálni itt, a felhasználó igényeinek (a specifikációk) és a fejlesztők igényeinek (pl.: karbantarthatóság) is eleget kell tennie

CMM(I): a szoftver folyamat mérése

- Capability Maturity Model (Integration)
- Cél: a szoftverfejlesztési folyamat hatékonyságának mérése
- Egy szervezet megkaphatja valamely szintű minősítését
- 5 besorolási szint (a fölsőbb szintek magába foglalják az alsókat)
 - Kezdeti: csak néhány folyamat definiált, a többségük esetleges (Alapszint)
 - Reprodukálható: az alapvető projekt menedzsment folyamatok definiáltak. Költség, ütemezés, funkcionalitás kezelése megoldott és megismételhető. (Bevésesi szint)
 - Definiált: a menedzsment és a fejlesztés folyamatai is dokumentáltak és szabványosítottak az egész szervezetre. (Véglegesítési szint)
 - Ellenőrzött: a szoftver folyamat és termék minőségének részletes mérése, ellenőrzése. (Bevezetési szint)
 - Optimalizált: a folyamatok folytonos javítása az új technológiák ellenőrzött bevezetésével (Optimalizálási szint)
- A nagyobb szinteknél teljesülni kell a korábbi szintek követelményeinek is

Konfigurációkezelés

- A rendszer változásainak kezelése

- Változások felügyelt módon történjenek
- Eljárások és szabványok fejlesztése és alkalmazása
- Fejlesztés, evolúció, karbantartás miatt van rá szükség
- Sokszor hiba-követéssel egybekötött
- Verziók kezelése
- Változások forrásai
- Új piaci feltételek
- Vásárló, megrendelő új követelménye
- Szervezet újraszervezése (pl. felvásárlás)
- Új platform támogatása

Hibamenedzsment

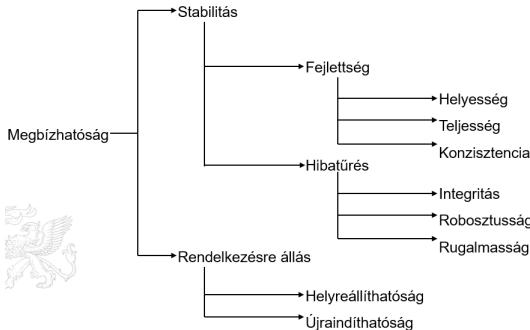
- Hiba-követés
 - Fontos, mert sok hiba van/lesz: kategorizálás, prioritások felállítása, követés elengedhetetlen
 - Hibaadatbázis, minden hibának egyedi aonosító
 - Számon van tartva a hiba felvezője, a hiba súlyossága, meg van-e javítva stb.
 - Fontos a hiba életútjának rögzítése
- Általánosabb: változtatás-menedzsment
 - CR (change request) adatbázis nyilvántartása

13.2.3. Szoftvermérés, metrikák

- **Szoftvermérés:** termék vagy folyamat valamely jellemzőjét numerikusan kifejezni (metrika)
 - Ezen értékekből következtetések vonhatók le a minőségre vonatkozóan
- Szisztematikus szoftvermérés még nem elterjedt
 - Mérési eredmény használata még nem kiforrott
 - Mérés szabványosításának (metrikák, eszközök) hiánya
- Metrikák két csoportja:
 - **Vezérlési metrikák:** Folyamattal kapcsolatosak, pl. egy hiba javításához szükséges átlagos idő
 - **Prediktor metrikák:** Termékkel kapcsolatosak, pl. LOC, ciklomatikus komplexitás, osztály metódusainak száma
- Mindkettő befolyásolja a vezető döntéshozatalt

Minőségi jellemzők mérése

- Jellemzőket lehetetlen közvetlenül mérni
 - Magasabb szintű absztrakciók, sok mindenről függnek
 - Hierarchikus összetétel (jellemzők származtatása)
 - Sokszor szervezet- vagy termékfüggő
 - Több metrika együttes vizsgálata
 - Metrikák változása az idő függvényében
 - Statisztikai technikák alkalmazása
- Metrikák (belőjük jellemző) és (külső) jellemzők közötti kapcsolatokra fel kell állítani egy modellt (Sok projekt esettanulmányának vizsgálata)
- Példa jellemző származtatásra:



Mérési folyamat

1. Alkalmazandó mérések kiválasztása
2. Mérni kívánt komponensek kiválasztása
3. Mérés (metrika számítás)
4. Magasabb szintű jellemző meghatározása modell alapján
5. Rendellenes értékek összehasonlítása (korábbi mérésekkel szemben)
6. Rendellenes komponensek részletes vizsgálata

Termékmetrikák

- Dinamikus
 - Szorosabb kapcsolat egyes minőségi jellemzőkkel
- Statikus
 - Közvetett kapcsolat
- Fajták:
 - Méret
 - Komplexitás, csatolás, kohézió
 - Objektumorientáltsággal kapcsolatos metrikák
 - Rossz előjelek, tervezési, kódolási problémák száma

Tesztelés „mérése” és fejlesztése

- Tesztelési környezet is minőségvizsgálatra szorul
- Metrikák, amiket definiálhatunk:
 - Lefedettség: adott változtatás hány %-át érintik a tesztek
 - Regressziós teszt hatékonysága
 - Tesztesetek redundanciája
- Tesztelési/fejlesztési költségek becsülhetővé válnak
- Súlyos összegek takaríthatók meg

Folyamat és projekt metrikák

- Folyamat mutatók: az aktuális folyamat hatékonysága
- Projekt mutatók: a jelenlegi projekt státusza
- A mérések alapján becsléseket készíthetünk (költség, ütemezés, minőség)
- A metrika olyan mutató, ami bepillantást nyújt a szoftver folyamatba (projektbe)

Szoftverfolyamat javítása

- Az alapvető cél a minőség és a hatékonyság növelése
- A technológia és a termékek bonyolultsága is befolyásoló tényező
- A minőség és hatékonyság szempontjából a legfontosabb tényező a munkatársak képzettsége és motiváltsága

- Személyes metrikák
 - Hiba riportok
 - Sorok száma modulonként
 - PSP (Personal Software Process) – személyre szabott folyamat
- Publikus metrikák: projekt szinten összegzett metrikák
- Hiba analízis
 - hibák forrása, javítási költségeik, kategórizálásuk stb.

Projekt metrikák

- Régi projektek mérési adatait használjuk új projektek költség- és időbecslésére
- Hatékonyság (funkció pontok, dokumentációs oldalak, LOC)
- Minőség (hibák szoftverfejlesztési feladatonként)
- Egy másik modell
 - Input(az erőforrások mérése)
 - funcOutput(a létrejött termék mérése)
 - Eredmény(a létrejött termék 'átadandó' használhatósága)
- Projekt menedzser használja ezeket a metrikákat

Méret alapú metrikák

- Széleskörűen használják ezeket a metrikákat, de nagyon sok vita van alkalmazásokról (könnnyű számolni, de prog nyelveknél eltérő)
- PL.: Költség / LOC, Hibák / KLOC, Költség / dokumentációs oldal

Funkció alapú metrikák

- Felhasználói inputok száma - alkalmazáshoz szükséges adatok
- Felhasználói outputok száma-riportok,képernyők,hibaüzenetek
- Felhasználói kérdések száma - on-line input és output
- Fájlok száma- adatok logikai csoportja
- Külső interfések száma - az összes gépi interfész (pl.adatfájlok), ami adatokat továbbít
- Az aktuális szoftver bonyolultsági kategorizálása szubjektív
- Funkció pont számítása: $FP = \text{Count total} \times [0.65 + 0.01 \times \sum (F_j)]$
- Az F_j ($j=1 \dots 13$) a bonyolultságot befolyásoló tényezők
- A tényezők számításához kérdések (minden kérdést 0-5 skálán pontozunk):
 - A rendszer megköveteli-e a biztonsági mentéseket és helyreállításokat?
 - Adatkommunikáció szükséges-e?
 - Kritikus-e a hatékonyság?
 - A rendszer intenzíven használt környezetben működik?
 - Van on-line adatbevitel?
 - Az on-line adatbevitelhez szükség van összetett képernyő kezelésre?
 - A fájlok aktualizálása on-line módon történik?
 - Bonyolultak az inputok,outputok,fájlok vagy lekérdezések?
 - Bonyolult a belső feldolgozás?
 - A forráskód újrafelhasználhatóra lett tervezve?
 - A konverzió és az installáció a tervezés része?
 - A rendszer többszöri installációra lett tervezve különböző szervezeteknél?
 - Változások támogatása lett tervezve?

- FP programozási nyelv független
- Hátránya, hogy sok szubjektív elemet tartalmaz, nincs konkrét fizikai jelentése

Kiterjesztett FP metrikák

- Az eredeti FP mérték információs rendszerekre lett tervezve, egyéb rendszereknél kiegészítésekre van szükség
- 3D Funkció pont mérték
 - Adat dimenzió
 - Funkcionális dimenzió – a belső műveletek (transzformációk) száma
 - Vezérlési dimenzió – átmenetek állapotok között. PI telefonnál automatikus hívás állapotba pihenő állapotból
 - Számítás: Index=input+output+lekérdezés+fájlok+külső interfész +transzformáció+átmenetek

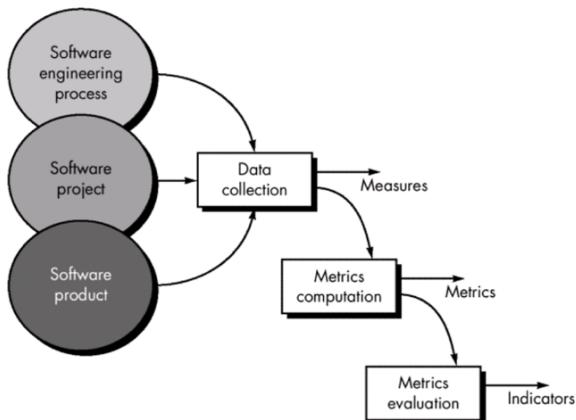
Semantic statements	1-5	6-10	11+
Processing steps			
1-10	Low	Low	Average
11-20	Low	Average	High
21+	Average	High	High

Metrikák alkalmazása szoftver-minőség mérésére

- Elsődleges a hibák és hiányosságok mérése
- A minőség mérése:
 - Helyesség (hiányosság/KLOC)
 - Karbantarthatóság (nincs mérőszám)
 - Integritás: külső támadások elleni védelem
 - Fenyegetettség: annak valószínűsége, hogy egy adott típusú támadás bekövetkezik egy adott időszakban
 - Biztonság: annak valószínűsége, hogy egy adott típusú támadást visszaver a rendszer
 - Integritás = $\Sigma [1-(fenyegetettség \times (1-biztonság))]$ (Összegzés a különböző támadás típusokra történik)
 - Használhatóság – a felhasználó barátság mérése (milyen könnyű használni stb.)
 - DRE (defect removal efficiency)
 - DRE = E/(E+D), ahol E olyan hibák száma, amelyeket még az átadás előtt felfedezünk, D pedig az átadás után a felhasználó által észlelt hiányosságok száma
 - Cél a DRE növelése(minél több hiba megtalálása az átadás előtt)
 - Fontos, hogy a hibákat a fejlesztés minél korábbi fázisában találjuk meg (analízis, tervezés)
- Néhány tipikus kérdés, amikre metrikákkal tudunk válaszolni:
 - Milyen felhasználói igények változnak a leggyakrabban?
 - A rendszer melyik komponensében várható a legtöbb hiba?
 - Mennyi tesztelést tervezünk a komponensekre?
 - Mennyi és milyen típusú hibát várhatunk el a tesztelés kezdetekor?

Metrikus baseline

- Kell egy hosszabb idejű, több projekten alapuló összehasonlítási alap (baseline)
- Korábbi projektek adatai alapján
- Metrikák gyűjtésének folyamata:



Statisztikai folyamat vezérlés

- Statisztikailag érvényes trendek megállapítása
- Vezérlési diagramm(metrikák stabilitásának mérése)
 - ER=hibák száma/ellenőrzésre fordított idő
- A változási tartomány mérése
 - mR bar – középvonal,UCL – felső vezérlési korlát

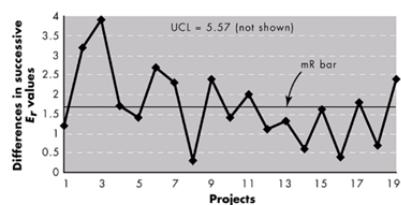


Figure 04.09
Moving range control chart

Összegzés

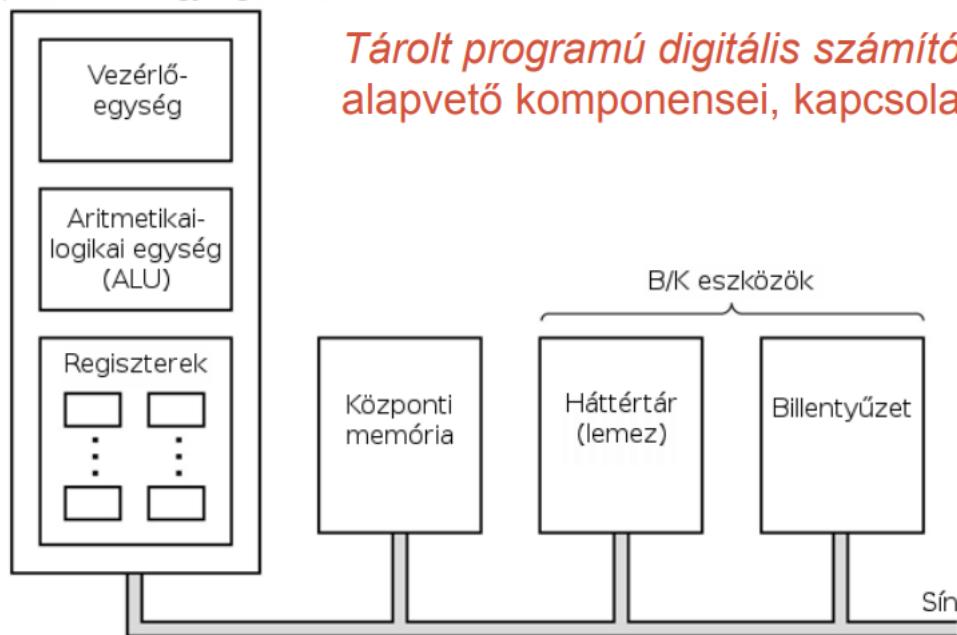
- A metrika alapú mutatók fontosak a menedzsment és a technikai vezetők számára
- Folyamat metrikákat stratégia szempontok alapján kell kiértékelni
- Méret- és funkció pont alapú metrikákat széleskörűen használnak az iparban.
- A szoftver minőség javításának alapja a metrika alapú „baseline”

14. Számítógép architektúra

14.1. 1. Neumann-elvű gép egységei. CPU, adatút, utasítás-végrehajtás, utasítás- és processzorszintű párhuzamosság. Korszerű számítógépek tervezési elvei. Példák RISC (UltraSPARC) és CISC (Pentium 4) architektúrákra, jellemzőik.

14.1.1. Neumann-elvű gép sematikus váza

Központi vezérlőegység (CPU)



Tárolt programú digitális számítógép alapvető komponensei, kapcsolatai

Központi memória: a program kódját és adatait tárolja, számokként

Központi feldolgozóegység (CPU): A központi memoriában tárolt program utasításainak beolvasása és végrehajtása

- **Vezérlőegység:** utasítások beolvasása a memoriából és típusának megállapítása
- **Aritmetikai és logikai egység (ALU):** Utasítások végrehajtásához szükséges aritmetikai és logikai műveletek elvégzése
- **Regiszterek:** kisméretű, gyors elérésű memóriarekeszek, részeredmények tárolása, vezérlőinformációk

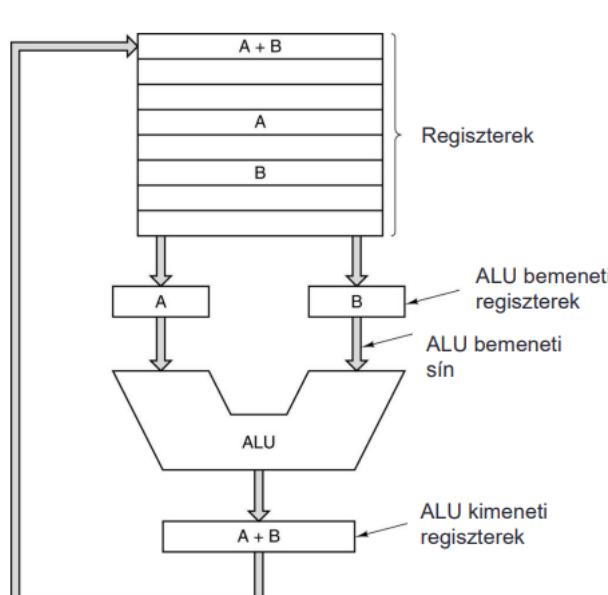
Külső sín: részegységek összekötése (kábel, huzalozás), adatok, címek, vezérlőjelek továbbítása különböző buszokkal

Belső sín: CPU részegységei közötti kommunikáció (vezérlőegység, ALU, regiszterek)

Beviteli és kiviteli eszközök: felhasználóval való kapcsolat, adattárolás háttértárakon, nyomtatás stb.

(Működést biztosító járulékos eszközök: gépház, tápellátás, stb.)

14.1.2. Adatút



• Példa

- Összeadás elvégzése ALU-val

• Legtöbb utasítás

- Regiszter-memória
- Regiszter-regiszter

• Adatciklus

- Két operandus ALU-n való átfutása és az eredmény regiszterben tárolása

14.1.3. Utasítás végrehajtás

Betöltő-dekódoló-végrehajtó ciklus

1. Soron következő utasítás beolvasása a memoriából az utasításregiszterbe az utasításszámláló regiszter mutatta helyről
2. Utasításszámláló beállítása a következő címre

3. Utasításszámláló beállítása a következő címre
4. A beolvasott utasítás típusának meghatározása
5. Ha az utasítás memóriára hivatkozik, annak lokalizálása
6. Ha szükséges, adat beolvasása a CPU egy regiszterébe
7. Az utasítás végrehajtása

Probléma: A memória olvasása lassú, az utasítás és az adatok beolvasása közben a CPU többi része kihasználatlan

Gyorsítási lehetőségek:

- Órajel frekvenciájának emelése (korlátozott)
- Utasításszintű párhuzamosság
 - Csővezeték
 - Szuperskaláris architektúrák
- Processzorszintű párhuzamosság
 - Tömbszámítógépek
 - Multiprocesszorok
 - Multiszámítógépek

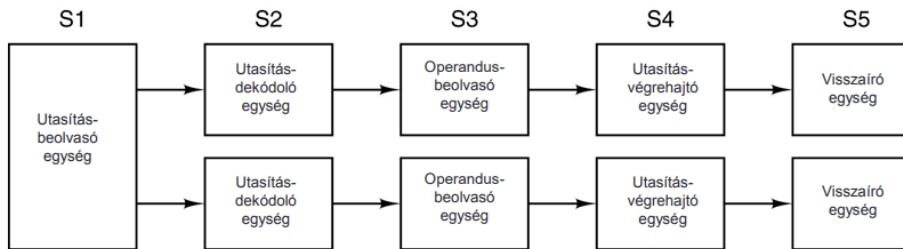
Késleltetés: utasítás végrehajtásának időigénye

Áteresztőképesség: MIPS (millió utasítás mp-enként)

14.1.4. Utasításszintű párhuzamosság

Párhuzamos csővezetékek

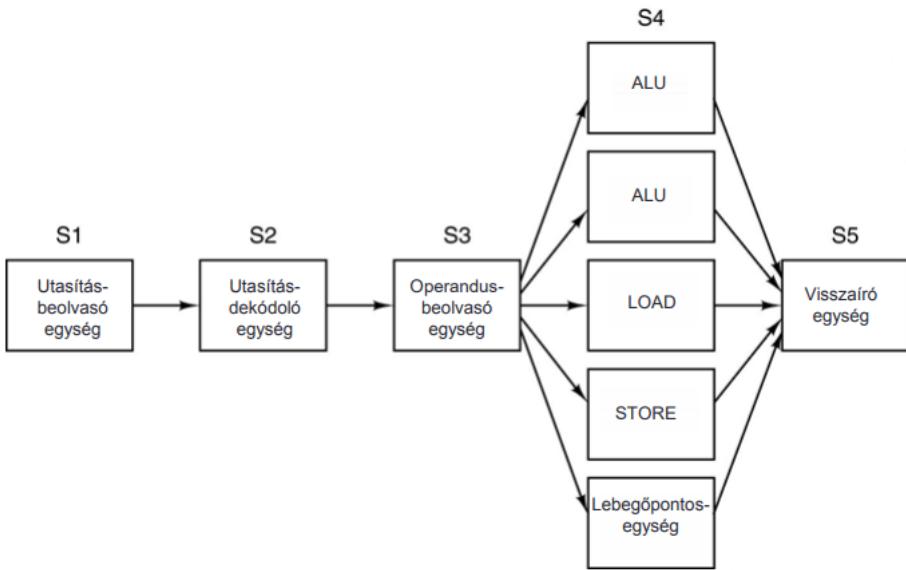
- Közös utasítás-beolvasó egységgel
- A csővezetékek saját ALU-val rendelkeznek (párhuzamos végrehajtás, ha nincs erőforrás-használat ütközés)
- Általában 2 vagy 4 csővezeték



- Pentium hasonlót alkalmaz
 - Fő csővezeték: tetszőleges Pentium utasítás
 - Második csővezeték: csak egész műveletek

Szuperskaláris architektúrák

- Egy csővezeték, de több funkcionális egységgel
- Feltételezzük hogy S1-S3 fázis sokkal gyorsabb, mint S4
- Funkcionális egységek ismétlődhetnek, pl. több ALU is lehet



- Hasonló a Pentium 4 architektúrája

14.1.5. Processzorszintű párhuzamosság

- **Tömbszámítógépek**
 - Ugyanazon műveletek elvégzése különböző adatokon → párhuzamosítás
- **Multiprocesszorok (szorosan kapcsolt CPU-k)**
 - Több CPU, közös memória → együttműködés vezérlése szükséges
 - Sínrendszer
 - 1 közösen használt (lassíthat)
 - Emellett a CPU-k akár saját lokális memóriával is rendelkezhetnek
 - Jellemzően max. pár száz CPU-t építenek össze
- **Multiszámítógépek (lazán kapcsolt CPU-k)**
 - Nincs közös sín, processzor-kommunikáció üzenetküldéssel
 - Általában nincs minden gép összekötve egymással (pl. fa-struktúra)
 - Több ezer gép is összeköthető

14.1.6. RISC és CISC

- **RISC (Reduced Instruction Set Computer)**
 - Csökkentett utasításkészletű számítógép
 - Csak olyan utasítások legyenek, amelyek az adatút egyszeri bejárásával végrehajthatók
 - Tipikusan kb. 50 utasítás
- **CISC (Complex Instruction Set Computer)**
 - Összetett utasításkészletű számítógép
 - Sok utasítás (akár több száz), mikroprogram interpretálással
 - Lassabb végrehajtás

Intel: A kezdeti CISC felépítésbe integrálták egy RISC magot (80486-tól) a leggyakoribb utasításoknak

14.1.7. Korszerű számítógépek tervezési elvei

- **Minden utasítást közvetlenül a hardver hajson végre**
 - A gyakran használtakat minden képpen
 - Interpretált mikroutasítások elkerülése
- **Maximalizálni az utasítások kiadási ütemét**

- Párhuzamos utasításkiadásra törekedni

- Az utasítások könnyen dekódolhatók legyenek

- Kevés mezőből álljanak, szabályosak, egyforma hosszúak legyenek, ...

- Csak a betöltő és a tároló utasítások hivatkozzanak a memóriára

- Egyszerűbb utasításforma, párhuzamosítást segíti

- Sok regiszter legyen

- Számítások során ne kelljen a lassú memóriába írni

14.2. 2. Számítógép perifériák: Mágneses és optikai adattárolás alapelvei, működésük (merevlemez, Audio CD, CD-ROM, CD-R, CD-RW, DVD, Bluray). SCSI, RAID. Nyomtatók, egér, billentyűzet. Telekommunikációs berendezések (modem, ADSL, KábelTV-s internet).

14.2.1. Mágneslemezek

Részei:

- Mágnesezhető felületű, forgó alumíniumkorong

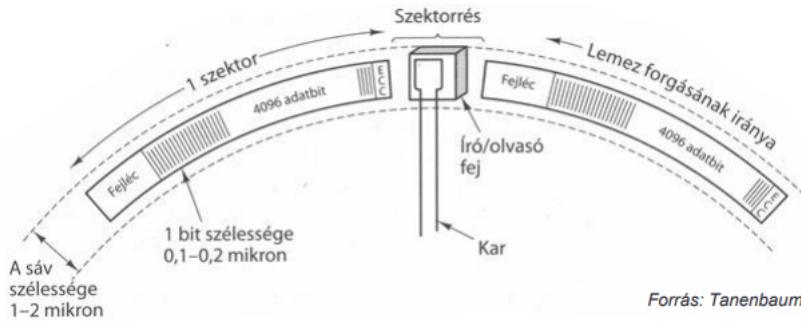
- Átmérő kezdetben 50 cm, jelenleg 3-12 cm

- Indukciós tekercset tartalmazó fej

- Lebeg vagy érinti a felszínt

- Kar

- Sugárirány mentén a fej egyvonalú mozgatása



Forrás: Tanenbaum

Írás: Pozitív vagy negatív áram az indukciós tekercsben, alegel

adott helyen mágneseződik

Olvasás: Mágnesezett terület felettelhaladva pozitív vagy negatív áram indukálódik a mágneses polarizációnak megfelelően

Adattárolás:

- Sáv

- Koncentrikus körök mentén
- Egy teljes körülfordulás alatt felírt bitsorozat
- Centiméterenként 5-10 ezer sáv (szélesség)

- Szektor

- 1 sávon több szektor

- Fejléc: fej szinkronizálásához
- Adat (pl. 512 bájt)
- Ellenőrző kód
- Hamming vagy Reed-Solomon
- Szektorrész

- Lineáris adatsűrűség

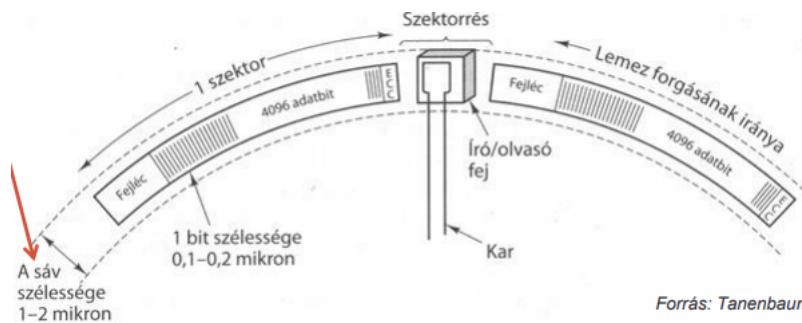
- Kerület mentén, 50-100 ezer bit/cm

- Merőleges rögzítés

- Tárolás hosszirány helyett „befelé” történik

- Kapacitás**

- Formázott és formáztatlan



Forrás: Tanenbaum

Felépítés:

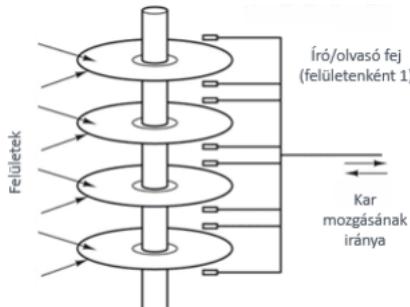
- Fontos a tisztaság és a pormentesség → zárt merevlemezek

Lemezegység

- Közös tengelyen több lemez (6-12), azonos fej pozíció!
- Cilinder: adott sugárpozícióban lévő sávok összessége

Teljesítmény

- Keresés (seek): fej megfelelő sugárirányba állítása (kar)
 - 1 ms: egymás utáni sávok
 - 5-10 ms: átlagos (véletlenszerű)
- Forgási késleltetés
 - A kerület mentén a fej alá fordul a kívánt terület
 - Fél fordulat ideje, 3-6 ezredmásodperc (5400, 7200, 10880 fordulat / perc mellett)



Jellemzők:

- Mechanikai sérülés előfordulhat (Fizikai behatásra a fej megsértheti a lemezt)
- Lemevezérlő lapka
 - Sokszor saját CPU-t is tartalmaz
 - Szoftverből érkező parancsok fogadása
 - READ, WRITE, FORMAT
 - Kar mozgatása
 - Hibák felismerése és javítása
 - Javíthatatlan hiba esetén fizikai áthelyezés
 - Bájtok oda- és visszaalakítása bitek sorozatává
 - Pufferelés (gyorsítás)

Típusok:

SCSI

- Small Computer System Interface

- „kis számítógép-rendszerek interfésze”, kiejtése „szkazi”

- Olyan szabványegyüttes, melyet számítógépek és perifériák közötti adatátvitelre terveztek
- A SCSI szabványok definiálják a parancsokat, protokollokat, az elektromos és optikai csatolófelületek definíciót
- A SCSI merevlemezek fizikai mérete ugyanakkor, mint az ATA és SATA winchestereké – lemezeinek átmérője 3,5 inch –, viszont percenkénti fordulatszáma azoknál nagyobb, haladóbb eszközök

RAID

- Redundant Array of Inexpensive Disks - Olcsó lemezek redundáns tömbje
- Ellentéte: SLED (Single Large Expensive Disk), egyetlen nagy drága lemez
- Több merevlemez egységbe foglalása (SCSI alkalmazása a párhuzamossága miatt)
- A rendszer felé egy nagy lemez kint jelenik meg
- Az adatok a lemezeken szétosztásra kerülnek
- redundancia javítja a megbízhatóságot
- Többféle szervezési mód (RAID 0 - RAID 6), megvalósítása lehet hardveres vagy szoftveres

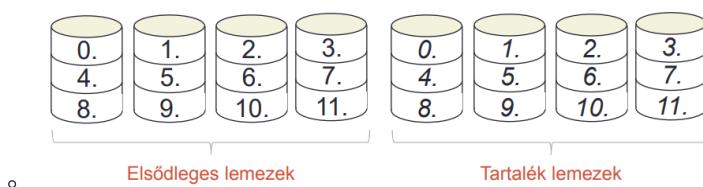
- RAID 0**

- Adatok párhuzamos tárolása a lemezeken
 - k darab szektorból álló csíkok (stripes)
 - Csíkok egy-egy lemezen tárolódnak
- Nincs hibajavítási képessége, nem „igazi” RAID (így gyorsabb)
- Nagyméretű blokkokkal működik legjobban



- RAID 1**

- Adatok írása két példányban (két különböző lemezre) csíkozással (4 elsődleges és 4 tartalék lemez)
- Olvasás párhuzamosítható, egyes szektorok az elsődleges, mások a tartalék lemezekről
- Hibás lemezegység cserélhető, csak rá kell másolni a „párja” tartalmát



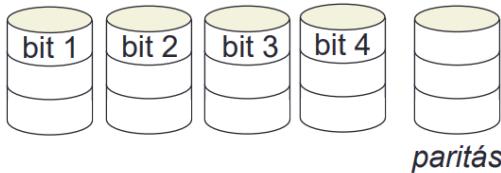
- RAID 2**

- Bájt- vagy szó-alapú tárolás (szektorcsoportok helyett)
- A lemezeknek és a karoknak szinkronban kell mozogniuk
- Adat + Hamming kód bitjeinek egyidejű tárolása külön lemezeken (4 adatbit + 3 paritásbit = 7 tárolandó bit; 7 szinkron lemez kell)
 - Hamming távolság: két azonos hosszságú bináris jelsorozat eltérő bitjeinek a száma
 - minimális hamming távolság segítségével detektálja és javítja a hibákat (ha d a minimális Hamming távolság, akkor $d-1$ hibát tud detektálni és $\lfloor(d-1)/2\rfloor$ hibát tud javítani)
- Sok merevlemez esetén használható jó!
- Vezérlőnek plusz munka a Hamming kód kezelése!
- Hamming kóddal pótolható a kieső lemez tartalma -> hibatűrő



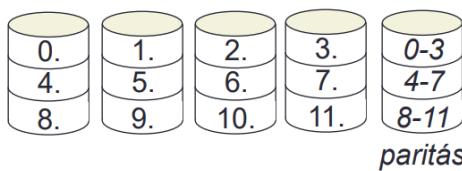
- RAID 3

- A RAID 2 egyszerűsített változata
- minden adatszóhoz egyetlen paritásbit
- Paritásbit tárolása dedikált lemezegységen
- Itt is szükséges a lemezek szinkron kezelése
- Javításra is alkalmas, ha tudjuk, hogy melyik lemezegység romlott el, de egyszerre maximum 1, tehát cseréljük gyorsan!



- RAID 4

- Csíkozással dolgozik -> nem szükséges a lemezegységek szinkron kezelése, Csíkonkénti paritást felírja egy dedikált paritás lemezegységre
- Kieső meghajtó tartalma előállítható a paritásmeghajtó segítségével
- Probléma: Íráshoz olvasni is kell minden lemezeiről, nagyon leterhel a paritásmeghajtó



- RAID 5

- RAID 4-hez hasonló elv, de nincs dedikált paritásmeghajtó
- A paritásbileket körbejárásos módszerrel szétosztja a lemezegységek között
- Legalább 3 lemezegység kell, legalább 4 ajánlott

14.2.2. Optikai lemezek

CD írás folyamata:

- Üveg mesterlemez: írás nagy energiájú lézerrel
- A mesterlemezről negatív öntőforma készül
- A negatív öntőformába olvadt polikarbonát gyantát öntenek
- Megszilárdulás után tükröző alumínium réteget visznek rá
- Védő lakk réteggel vonják be és rányomtatják a címkét

CD olvasás folyamata

- Olvasás kis energiájú infravörös lézerrel
- Az üregből visszavert fény fél hullámhossznyival rövidebb utat tesz meg, mint az üreg pereméről visszavert, ezért gyengíteni fogják egymást

Audio CD adattárolása

- Spirál alakban, belülről kifelé haladva, kb. 5,6 km hosszú
- A jel sűrűsége állandó a spirál mentén
- Állandó kerületi sebesség biztosítása, változó forgási sebesség
- nincs hibajavítás, de mivel audio ezért nem gond

CD-ROM

- Digitális adattárolásra

- Többszintű hibajavítás bevezetése (a hanggal ellentétben itt nem lehet adatvesztés!)
- nehezebb az olvasó fejet pozícionálni mint a merevlemezeknél (koncentrikus körök helyett spirál)
- Meghajtó szoftvere nagyjából a célterület fölött viszi az olvasófejet, Fejlécet keres, abban ellenőrzi a szektor sorszámot

CD-R

- CD-ROM-okhoz hasonló polikarbonát felépítés
- Saját író berendezéssel rögzíthető az adat
- *Újdonság*
 - Író lézernyaláb
 - Alumínium helyett arany felület
 - Üregek és szintek helyett festékréteg alkalmazása
 - Írás: a nagy energiájú lézer roncsol → sötét folt marad véglegesen
 - Olvasás: az ép és a roncsolt területek detektálása

CD-RW

- Újraírható optikai lemez
- *Újdonság*
 - Más adattároló réteg
 - Ezüst, indium, antimon és tellűr ötvözeti
 - Kétféle stabil állapot: kristályos és amorf (más fényvisszaverő képesség)
 - 3 eltérő energiájú lézer
 - Legmagasabb energia: megolvad az ötvözeti → amorf
 - Közepes energia: megolvad → kristályos állapot
 - Alacsony energia: anyag állapotnak érzékelése, de meg nem változik

DVD

- CD koronggal egyező méret
- Nagyobb jelsűrűség (kisebb üreg, szorosabb spirál)
- Vörös lézer
- Több adat (egy/két oldalas, egy/két rétegű (4,7 GB – 17 GB))
- Új filmipari funkciók: Szülői felügyelet, hatcsatornás hang, képarány dinamikus választása (4:3 vagy 16:9), régiókódok

Blu-Ray

- Kék lézer használata a vörös helyett
 - Rövidebb hullámhossz, jobban fókuszálható, kisebb mélyedések
 - 25 GB (egyoldalas) és 50 GB (kétoldalas) adattárolási képesség
 - 4,5 MB/mp átviteli sebesség

14.2.3. Kimenet/bemenet

Nyomtatók

- Mátrixnyomtatók
 - Monokróm nyomat
 - Tintaszalag + elektromágnesesen irányítható tük
 - Olcsó technika, elsősorban cégeknél (volt) jellemző
 - Pontmátrix karakterek
- Tintasugaras nyomtatók
 - Elsősorban otthoni használatra

- Lassú, de relatíve olcsó
- Tintapatront tartalmazó, mozgatható fej, lapra tintát permetez
- Fajtái
 - Piezoelektronos: Tintapatron mellett kristály, amely feszültség hatására deformálódik → tintacseppet présel ki
 - Hővezérlésű vagy festékbuborékos: Fúvókákban kis ellenállás, amely feszültség hatására felhevül, a festék felforr és elpárolog, túlnyomás keletkezik, papírra kerül, fúvókát lehűti, a keletkező vákuum újabb tintacseppet szív be a tartályból
- Lézernyomtatók
 - Kiváló minőségű kép, gyors működés
 - Saját CPU, memória
 - Elsősorban monokróm, de van színes változata is
- 3D nyomtatás
 - Digitális tervrajzokból → 3D tárgy
 - Porréteg + ragasztó komponens
 - jelenleg még drága
 - Prototípusok gyors készítése, egyedi tárgyak, objektumok készítése
 - Tárgyak helyett tervek küldése nagy távolságokra

Egér

- Grafikus felületen egy mutató mozgatása
- Egy, kettő vagy akár több nyomógomb vagy görgő
- Típusai
 - Mechanikus: Kerekek vagy gumi golyó, potenciométerek
 - Optikai: LED fény, visszaverődés elemzése
 - Optomechanikus: Golyó, két tengely forgat (merőlegesek), résekkel ellátott tárcsák, LED fény, mozgás hatására fényimpulzusok
- Működése: Bizonyos időnként (pl. 0,1 sec) vagy esemény hatására 3 adatos (általában 3 bájtos) üzenetet küld a soros vonalon (PS-2 vagy USB) a számítógépnek

Billentyűzet

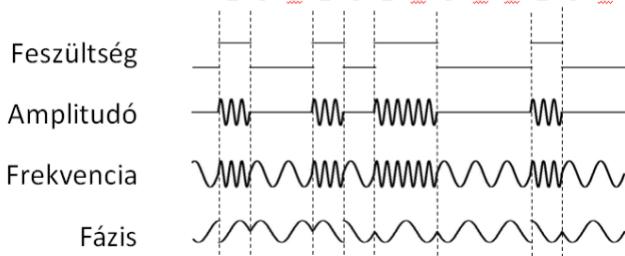
- Egy-egy billentyű leütése áramkört zár
- Megszakítás generálódik
 - Az operációs rendszer kezeli és továbbítja a programoknak

14.2.4. Telekommunikációs berendezések

Modem

- Adatkommunikáció analóg telefonvonalon
 - Az analóg vonalat hangátvitelre találták ki
 - Adatátvitelhez: vivőhullám (1000-2000 Hz-es szinusz hullám)
 - A bitek csak sorasan, egymás után vihetők át
 - 1 bájt átvitele: start bit + 8 adatbit + stop bit = 10 bit

Modulációk Jel 1 0 0 1 0 1 1 0 0 0 1 0 0



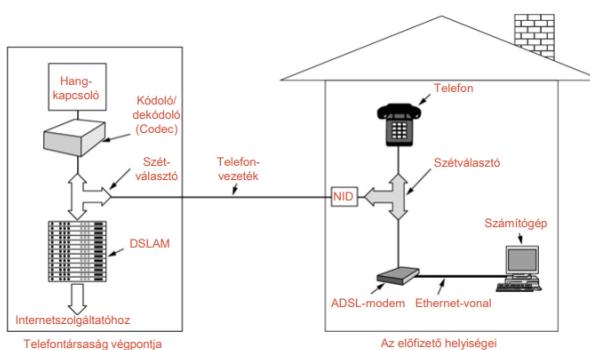
- Modulációk
 - amplitúdó, frekvencia módosítása
 - Fázis: díbit kódolás
 - 45, 135, 225 és 315 fokos fáziseltolódások az időintervallumok elején
 - 2 bit átvitele egységnyi idő alatt (45 fok: 00, 135 fok: 01, ...)
 - Kombinálva is használhatók

- Definíciók

- Baud: Jelváltás / másodperc
 - 1 jelváltás több bitnyi információt is hordozhat (lásd díbit kódolás)
- Adatátviteli sebesség: bit / másodperc
 - Jellemzően 28800 vagy 57600 bit / mp, jóval alacsonyabb baud értékkel!
- Kommunikációs vonal típusa
 - Full-duplex (kétirányú kommunikáció egyidőben)
 - fél-duplex (egyszerre csak 1 irányban)
 - szimplex (egyirányú kommunikáció lehetséges csak)

ADSL (Asymmetric Digital Subscriber Line)

- Szélessávú adatforgalom analóg telefonvonalon
- Hangátvitel: 3000 Hz-es szűrő alkalmazása a vonalon
- DSL technika: 1,1 MHz méretű tartomány használata
 - 256 darab 4 kHz-es csatorna
 - Szétválasztó (splitter)
 - Az alsó tartomány leválasztása hangátvitelre: 0. csatorna
 - A felső tartomány az adatátvitelé: 4-8 Mbps sebesség
 - 1-5. csatornák nem használtak (ne zavarja a hangátvitelt)
 - Két vezérlő csatorna a le- és feltöltés vezérlésére, a többi az adatátvitelre
- Tipikus hálózati konfiguráció



Kábeltévés internet

- Kábeltévé társaságok
 - Fő telephely + fejállomások
 - Fejállomások üvegkábelben a fő telephelyhez kapcsolódnak
 - A felhasználók felé induló vonalakon sok eszköz osztózik
 - Kábelek sávszélessége 750 MHz körüli
 - A sávszélesség függ a felhasználók pillanatnyi számától!!
 - Bonyolultabb kommunikáció a fejállomás és az előfizetői eszközök között

- Sávkiosztás

- 54 – 550 MHz: TV, rádió (lejövő frekvenciák)
 - 5 – 42 MHz: felmenő frekvenciák adatfeltöltésre és vezérlésre
 - 550 – 750 MHz: lejövő frekvenciák adatletöltésre
 - Aszimmetrikus adatkommunikáció
- Szükséges eszköz: kábelmodem