

Záróvizsga Tételek 2022

Algoritmusok és Adatszerkezetek I

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

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 hátizsák feladat

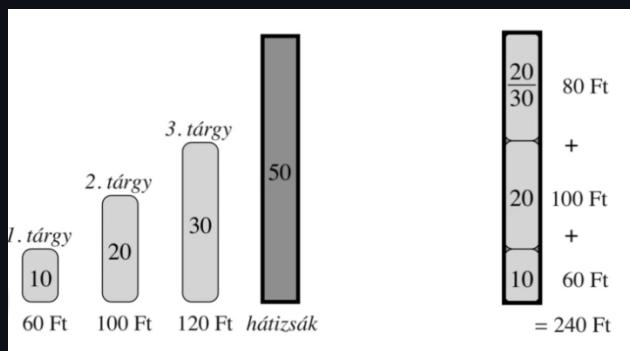
Bemenet: A hátizsák S kapacitása, n tárgy, S_i tárgy súlyok, E_i tárgy értékek

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

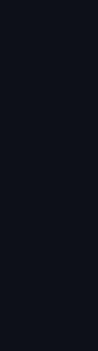
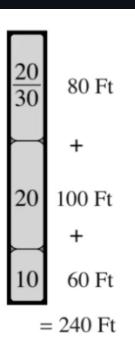


Futás a fenti példán:

- Kiszámoljuk az $\frac{E_i}{S_i}$ értékeket

i. Tárgy: 6

ii. Tárgy: 5



- 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 kitölti a szabad helyet. Jelen esetben a tárgy $\frac{2}{3}$ -át.

A probléma nem-törtedékes verziójára ez a mohó algoritmus nem mindenkor megoldást.

Oszd-meg-és-uralkodj algoritmusok

A feladatot több **részfeladatra** bontjuk, ezek hasonlóak 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.

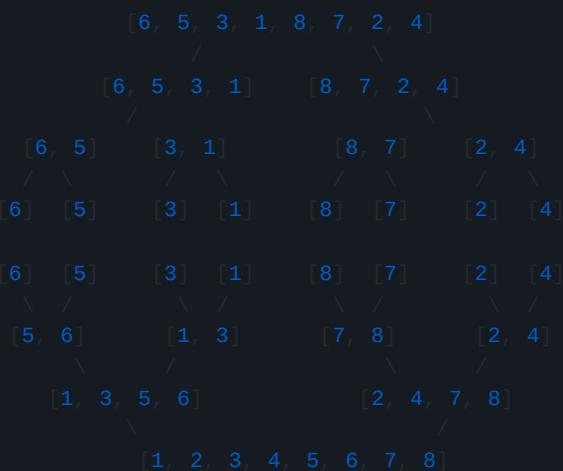
A részfeladatok ne legyenek átfedőek. Bár az algoritmus ettől még működhet, de nem hatékony.

Lépések

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

Példa: Összefésülő rendezés

1. **Felosztás:** Az n elemű rendezendő sorozatot felosztja két $\frac{n}{2}$ elemű részsorozatra.
2. **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.
3. **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, mindenkor a kisebbet fűzzük egy másik, kezdetben üres tömbhöz.

Példa: Felező csúcskereső algoritmus

Vizsgáljuk meg a középső elemet. Ha csúcs, térdünk 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 mindenkor melyik irányba haladunk tovább, egyszerűen azzal, amiről előbb megtudtuk, hogy nagyobb.

- Felosztás:** n elemű sorozatot felosztjuk két $\frac{n-1}{2}$ elemű részsorozatra
- Uralkodás:** A megfelelő részsorozatban rekurzívan tovább keresünk csúcsot
- 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.

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 memóriából 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 mindenet 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 forint.

```
// 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 trivi
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, tujuk, h
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, ás 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

```

Rendező algoritmusok

Rendezés

- Input:** Egészek egy n hosszú tömbje (egy $\langle a_1, a_2, \dots, a_n \rangle$ sorozat)
- Output:** n hosszú, rendezett tömb (az input sorozat egy olyan $\langle a'_1, a'_2, \dots, a'_n \rangle$ permutációja, ahol $a'_1 \leq a'_2 \leq \dots \leq a'_n$)

Ez egy egyszerű eset, a gyakorlatban:

- Van valamilyen iterálható kölleciónk: `Iterálható<Objektum>`)
- Van egy függvényünk, ami megondja képt köllecí-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.

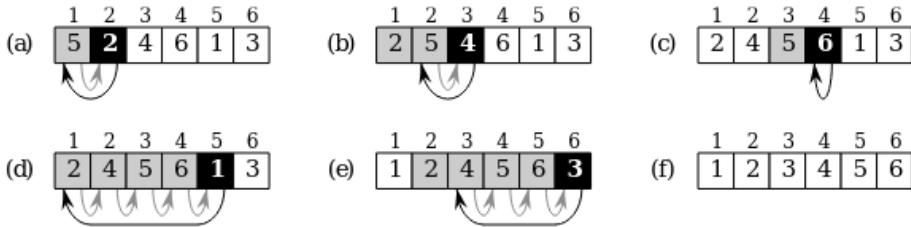
Beszúró rendezés

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.

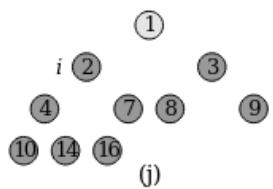
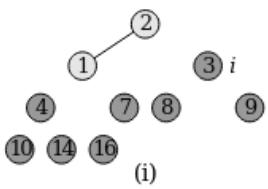
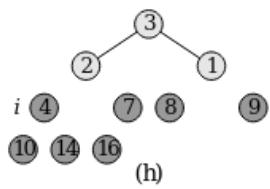
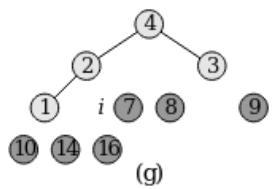
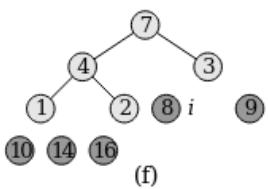
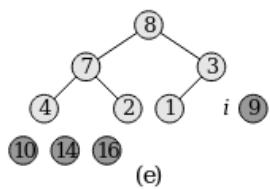
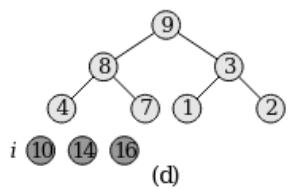
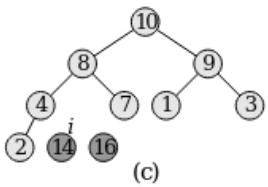
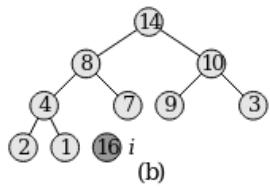
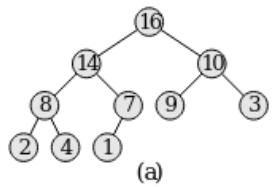
Kupacrendezés

```

const kupacRendezes = (A: number[]) => [
    maximumKupacotEpít(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 éppenvizsgált 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.)



(k)

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

Gyorsrendezés

Összefésülő 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 $A[q+1..r]$ résztömbököt a gyorsrendezés rekurzív hívásával rendezzik.
- 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 mályság $O(log n)$, ami mivel egy hívás futásideje $O(n)$, így az átlagos futásidő $O(n * log n)$. A gyakorlat azt mutatja, hogy ez az algoritmus jól teljesít.

Lehet úgy implementálni, hogy $O(log n)$ tárigénye legyen, ez egy helyben rendező, farok-rekúrziív 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 tömbeemet, 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. pozíción 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]]--;
  }

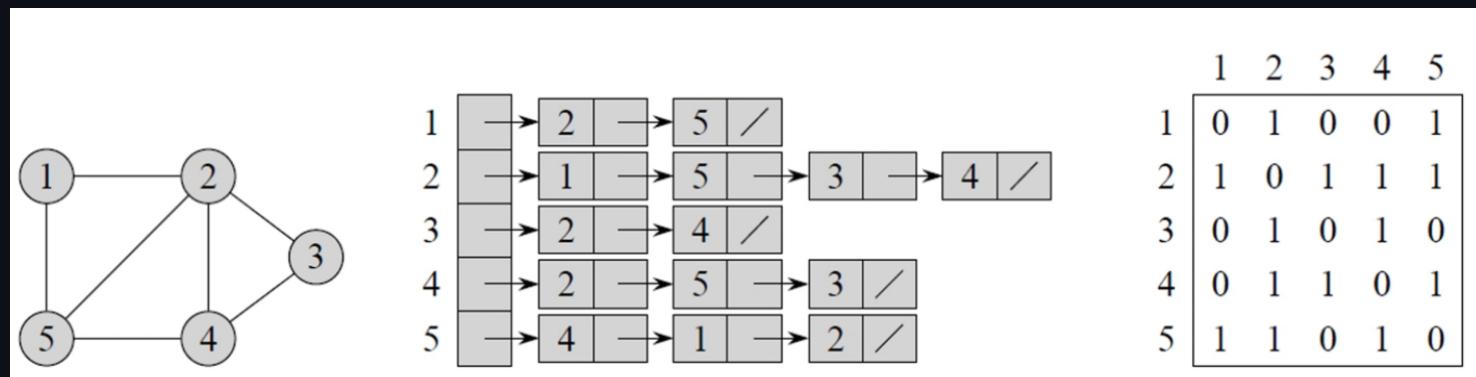
  return B;
};

```

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

Gráfalgoritmusok

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



Szélességi keresés

Gráf bejárására szolgál.

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

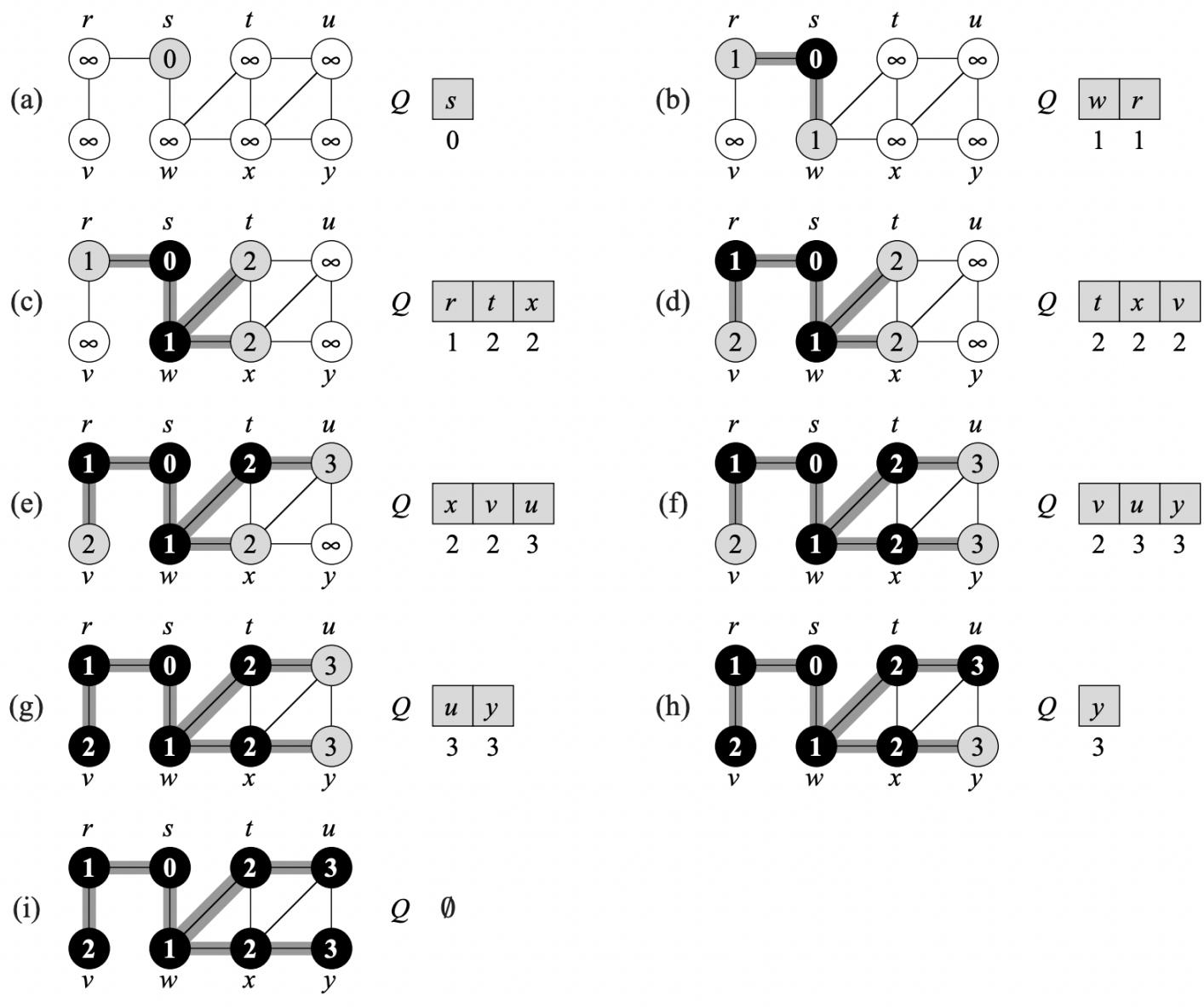
Amilyen távol van a kiindulási csúcstól egy csúcs, az olyan mélységen helyezkedik el ebben a fában.

Irányított, irányítatlan gráfog esetén is alkalmazható.

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

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

Futásidő

- Minden csúcsot 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

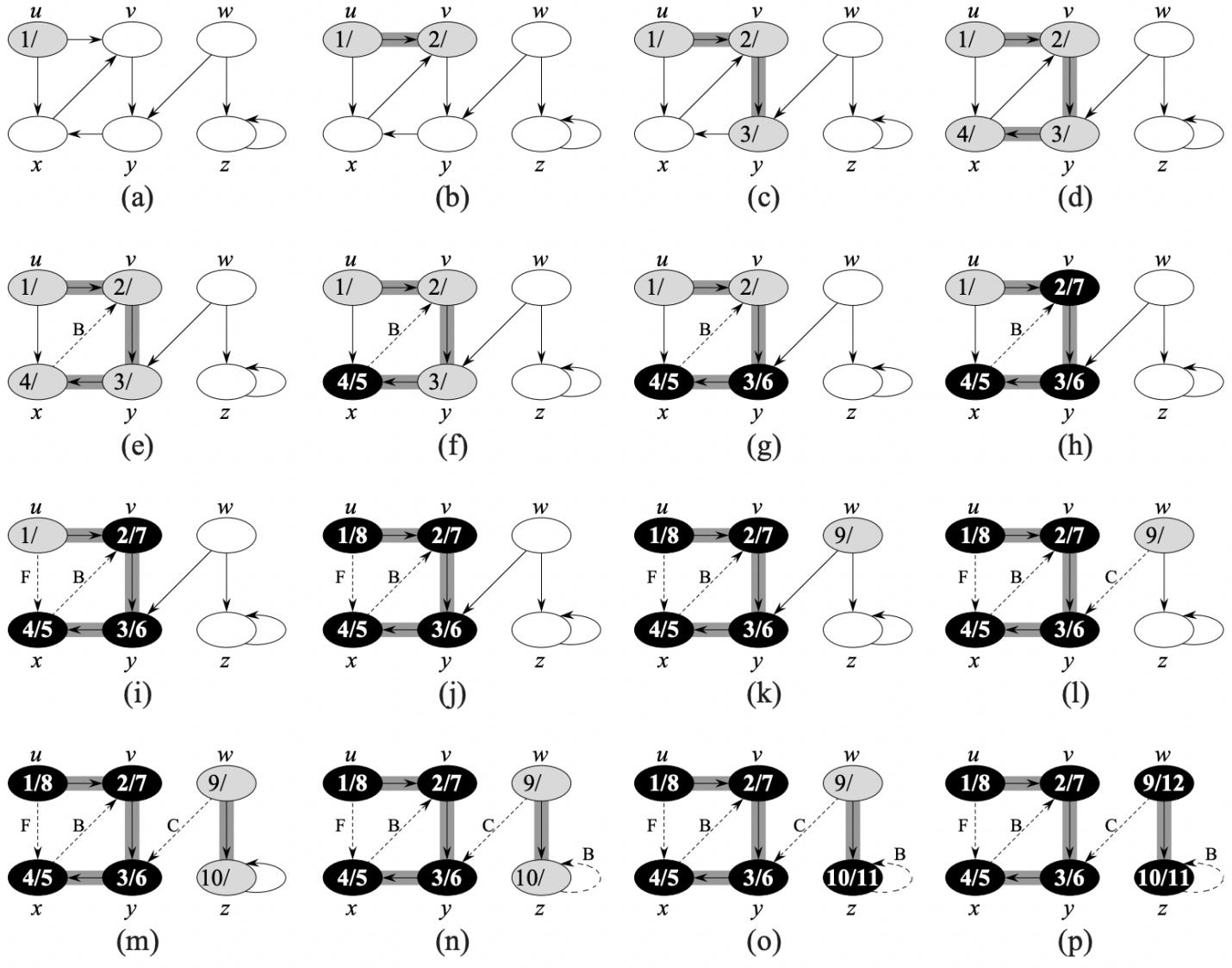
Addig megy a kivezető élek mentén, ameddig tud, majd visszafele indulva minden érintett csúcs kivezető élein addig megy mélyre, amíg lehet.

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

melysegiBejaras(u) {
    szin[u] = "szürke"
    idő++
    d[u] = idő // Ekkor értük el
    for u minden v szomszédjára {
        if (szin[v] === "fehér") {
            szülő[v] = u
            melysegiBejaras(v) // Azonnal már indulunk is el a talált csúcsból
        }
    }
    szin[u] = "fehete"
    idő++
    f[u] = idő // Ekkor hafytuk 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 eloszlatva, de **összessen** ennyi szomszédot vizsgál meg például a melysegiBejárás().

Minimális feszítőfák

Cél: megtalálni éleknek azon **körmentes** részhalmazát, amely élek mentén minden **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 csúcs.

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 = Φ
```

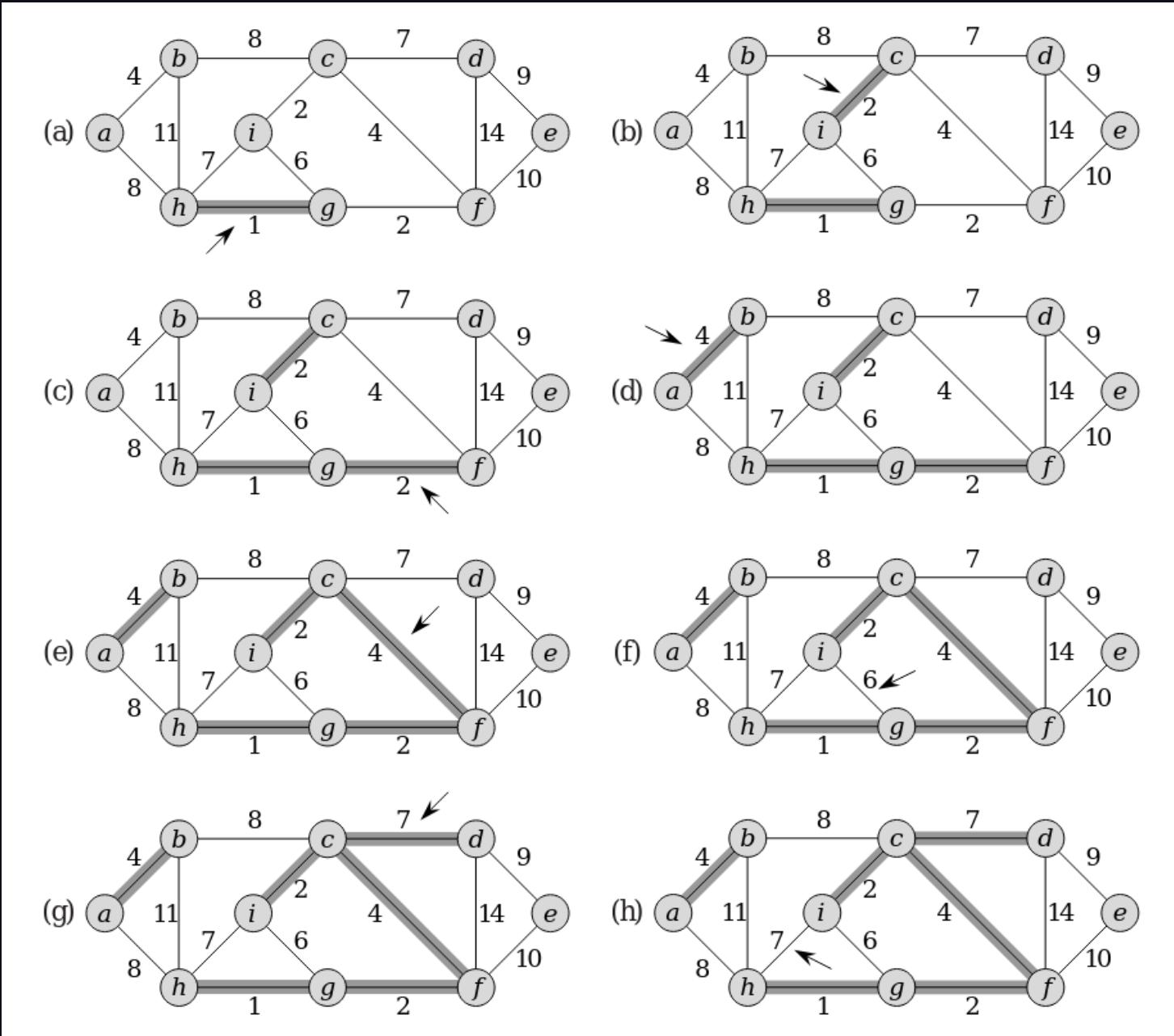
```

for minden v csúcsra {
    halmaztKeszit(v)
}

for minden (u, v) elre, az elősűlyok 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 jasátossága. 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 $O(E * \log E)$.

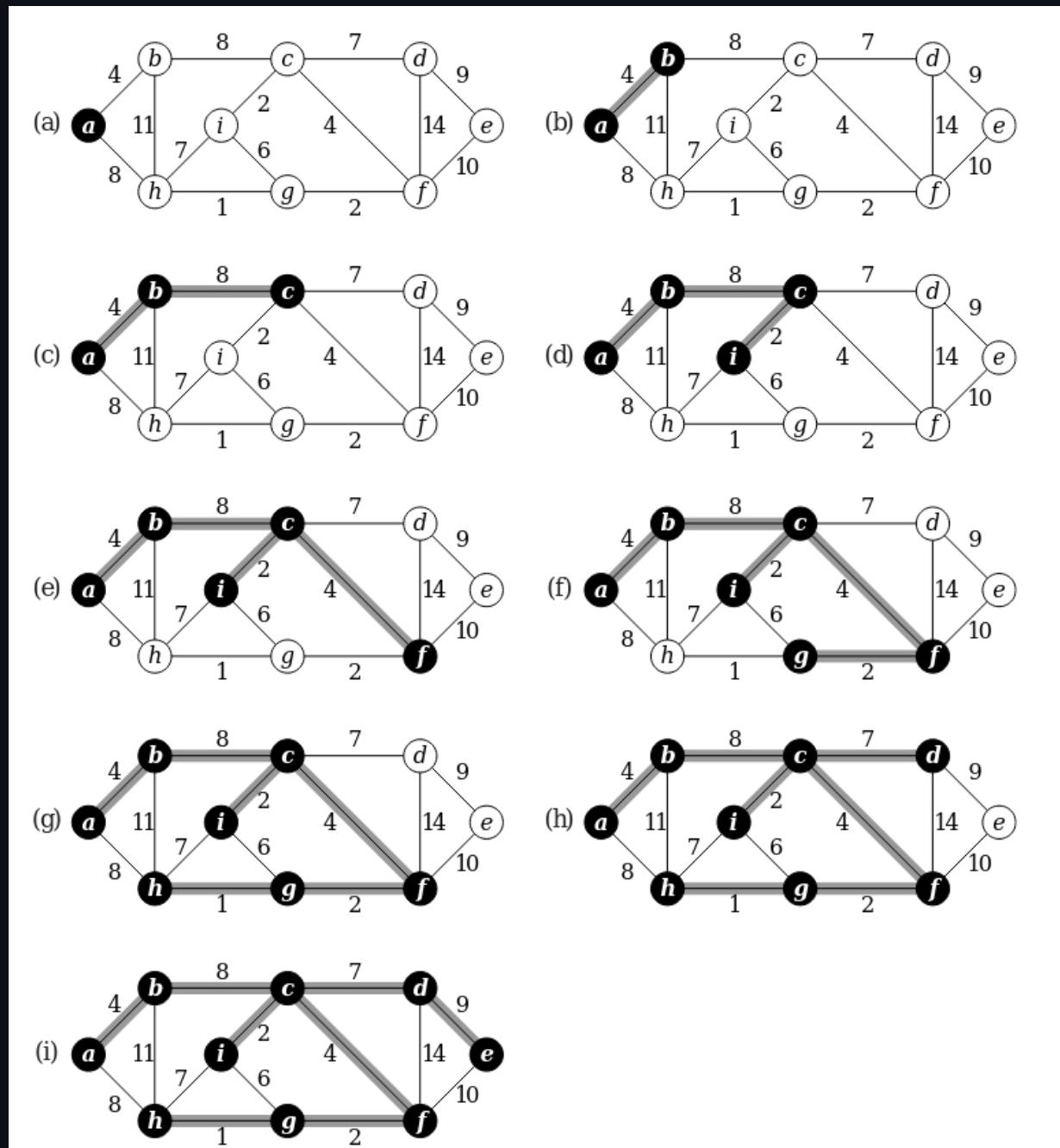
Így a teljes futásidő $O(E * \log E)$.

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 értékadások: $O(V)$

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 kulcsnak való értékadás valójában egy 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 elsőt 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 mégegyszer végig haladva mindig kisebb súlyú utat kapunk.

Kör a legrövidebb útban: Negatív összsúlyú tehát 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 csúcsból indul
    for minden v csúcsra {
        f[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
    }
}
```

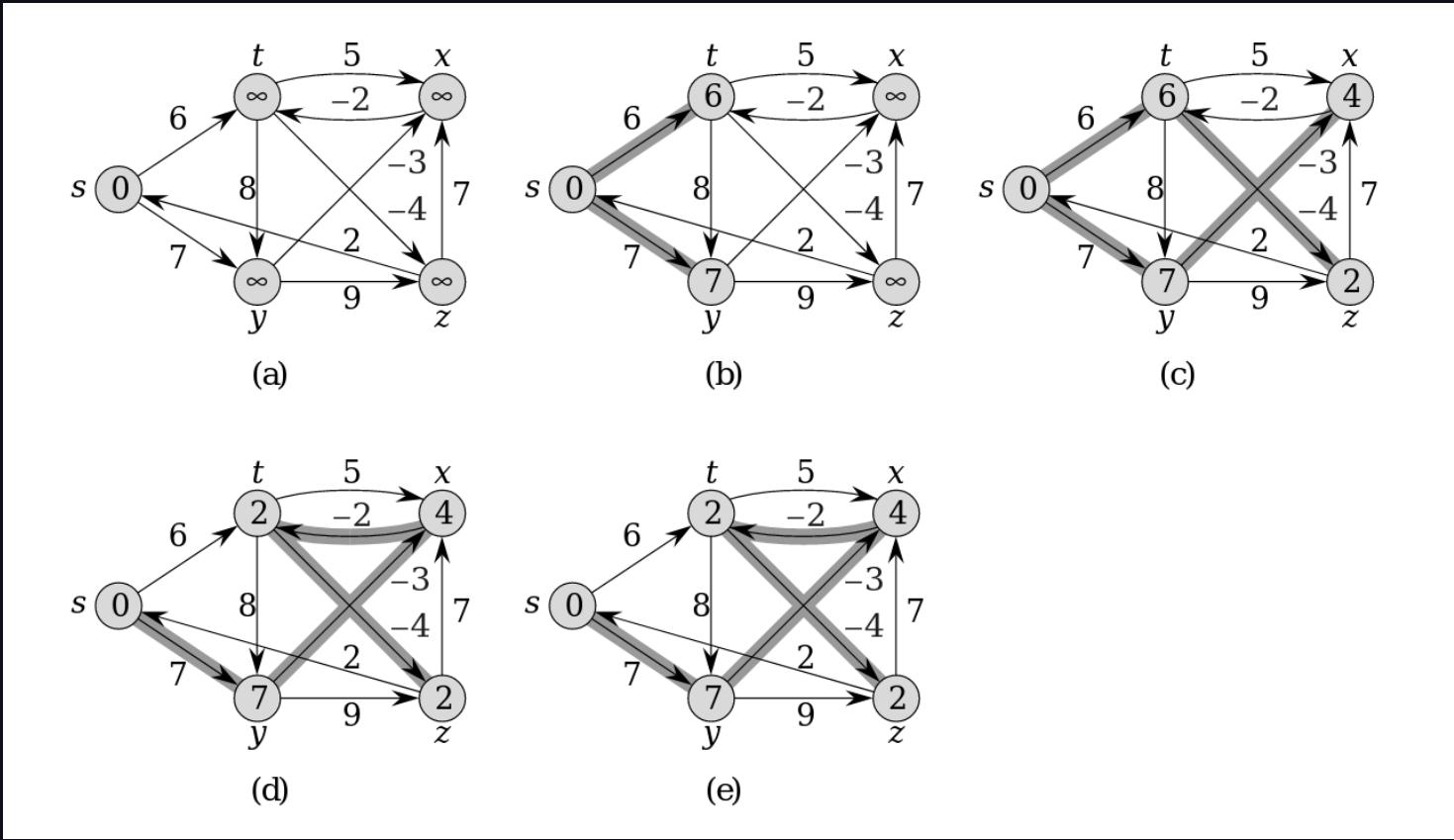
Bellman-Ford algoritmus

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) érre {
            közelít(u, v, w)
        }
    }
    for minden (u, v) érre { // Itt ellenőrzi, hogy volt-e negatív kör
        if d[v] > d[u] + w(u, v) {
            return false
        }
    }
}
return true
}

```



Futásidő

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

Dijkstra algoritmusa

Nemnegatív élsúlyok esetén működik.

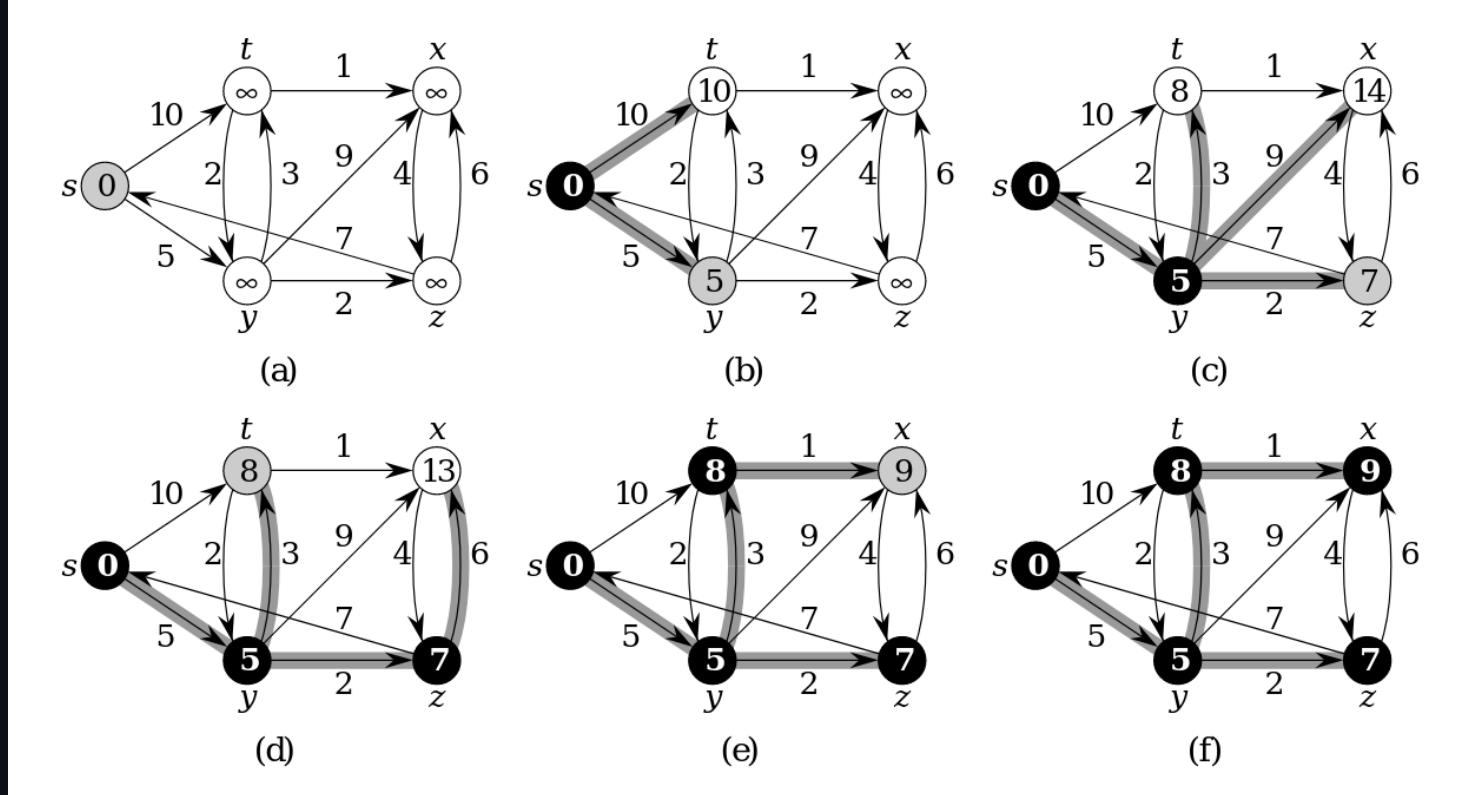
S halmaz: Azon csúcsok kerülnek bele, amikhez már meghatározta a legrövidebb utat a kezdőcsúcsból.

```

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 legfejlebb $O(E)$ -szer hívódik meg.

Az összesített futásidő nagyban függ a **prioritási sor implementációtól**, a legegyszerűbb eset, ha egy **tömbbel implementáljuk**. Ekkor a `beszűr` 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 látalánosságban 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-e 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 szomszedsagi martix
    n = sorokSzama(W)
    D(0) = W
    for k = 1-tol n-ig { // Ezt vizsgaljuk mindig majd, mint köztes csúcs
        for i = 1-tol n-ig {
            for j = 1-tol 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 csúcsot**.

Futásidő

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

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 módosítást.

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

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

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

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

Listák

Absztrakt adatszerkezet.

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

Művelet	Magyarázat
ÉRTÉK(H , i)	i . pozíció (index-en) a kulcs értékének visszaadása
ÉRTÉKAD(H , i , k)	i . pozíció levő értéknek a k érték értéküladása
KERES(H , k)	A k kulcs (érték) megkerekése a listában, indexének visszaadása
BESZÚR(H , k , i)	Az i -edik pozíctól 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
$\text{ÉRTÉK}(h, i)$	$O(1)$
$\text{ÉRTÉKAD}(h, i, k)$	$O(1)$
$\text{KERES}(h, k)$	$O(n)$
$\text{BESZÚR}(h, k, i)$	$O(n)$
$\text{TÖRÖL}(h, k)$	$O(n)$

Beszúrásnál újra kellhet allokálni egyel nagyobb emmóriaterületet.

Jellemzően úgy implementáljuk, hogy definiálunk egy **kapacitást**, és amikor kell, akkor eggyivel 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	Futásidő
ÉRTÉK(h, i)	$O(n)$
ÉRTÉKAD(h, i, k)	$O(n)$
KERES(h, k)	$O(n)$
BESZÚR(h, k, i)	$O(1)$
TÖRÖL(h, k)	$O(1)$

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

Előnye: Nem egy nagy összefüggő memória blokk kell.

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

Verem

Lista, amiben csak a legutoljára beszűrt 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 csökk
}
}

```

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

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

Sor

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

Lefoglalunk egy valamekkora egybefüggő memória szegmenst, de nem minden használjuk az egészet. 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 "körvelfordult" 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 "körvelfordult" a sor a lefoglalt memóriaterületen.
    } else {
        fej[S]++
    }
}

```

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

Prioritási sor

Absztrakt adatszerkezet.

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

Művelet	Magyarázat
BESZÚR(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 gyerekeié, 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

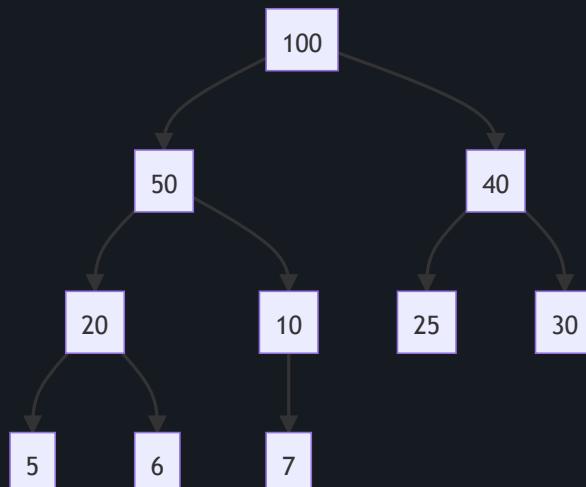
Egybefüggő memóriaterületen van a teljes kupac.

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] inkább 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] inkább 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 kicseréljük, és rekurzívan meghívjuk rá a `maximumKupacol()`-t, mert lehet, az új szülőjénél/gyerekénél is nagyobb.

`maximumKupacol()` futásidje $O(log n)$, mert ennyi a majdnem teljes bináris fa mélysége, és legrosszabb esetben az egészen 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.

```

kupacMaxima(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 részekén
    maximumKupacol(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
<code>BESZÜR(H, k)</code>	$O(log n)$
<code>MAX(H)</code>	$\Theta(1)$
<code>KIVESZ-MAX(H)</code>	$O(log n)$

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.

Gyerek 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.

Bináris keresőfák

Absztrakt adatszerkezet a következő műveletekkel:

Művelet	Magyarázat
KERES(T, x)	Megkeresi a fában az x kulcsot, és visszaadja azt a csúcsot
BESZÚR(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.

```

fabanKeres(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

```

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

```

A maximum elem a "legjobboldali" elem

```

fabanMaximum(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.

TODO: Hasonlóan a megelőzőre.

Beszűr

```
fábaBeszűr(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 gyerekes belső csúcs

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

Két gyerekes 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).

Futásidők

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 kiegensúlyozott fák esetén jó, hiszen akkor $O(\log n)$ -es.

Pl. AVL-fa, bináris kereső fa kiegensúlyozott.

Halmaz

Absztrakt adatszerkezet.

Egy elem legfejlebb egyszer szerepelhet benne.

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

Egyéb extra műveletek definiálhatóak, pl.: `METSZET`, `UNIÓ`

Közvetlen címzésű táblázat

Egy akkora 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.

Szótár

Absztrakt adatszerkezet.

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

dict, asszociatív tömb, map

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 réseire.

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

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

Mivel az unicerzum, a lehetséges kulcsok száma nagyobb, mint réseké (különben csinálhatnánk tömbös megvalósítást), így elkerülhetetlen, 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 résekben 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 lsitán is végig kelhet menni.

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

m : rések száma

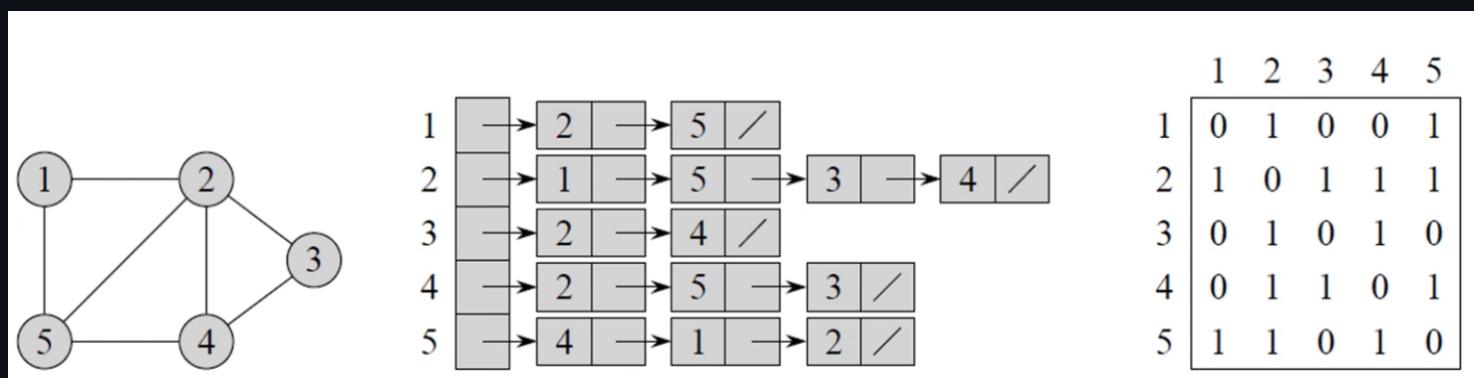
n : elemek a táblában

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 rések számát úgy, hogy az α egy konstans legyen, ekkor **KERES**, **TÖRL**, **BESZÚR** minden $O(1)$.

Gráfok számítógépes reprezentációja



- 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 először megfontolni, 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.

Bonyolultságelmélet

A P osztály

R 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**.

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

Hatókony 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 készíti **választató** módon, azaz minden x inputra $A(x) = B(f(x))$

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

Ebben az esetben ha B eldönthető, akkor A is eldönthető, illetve ha A eldönthetetlen, akkor B 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 mondtunk 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 készíti **választató** 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 $\text{PÁROSÍTÁS} \leq \text{SAT}$

PÁROSÍTÁS

Input: Egy CNF (konjunktív normálformájú formula)

Output: Kielégíthető-e?

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

SAT

Input: Egy G gráf

Output: Van-e G -ben teljes párosítás?

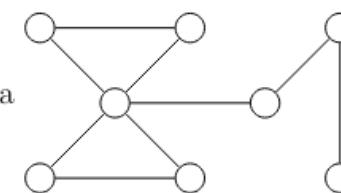
Közös csúccsal nem rendelkező éllek halmaza, amik lefednek minden csúcsot.

Visszavezetés

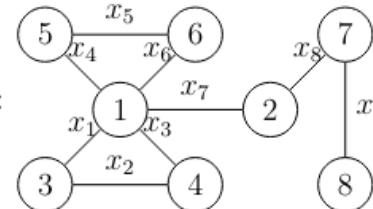
Tehát a cél egy G gráfból egy ϕ_G CNF előállítása választartó módon, polinomidőben úgy, hogy G -ben pontosan akkor legyen teljes párosítás, ha ϕ_G kielégíthető.

- minden élhez 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 felírunk, hogy pontosan egy él illeszkedik rá, majd ezeket összeéseljük. Ha így egy csúcsra sikerül megfelelő CNF-et alkotni, akkor azok összeéselése is CNF, hiszen CNF-ek összeéselése CNF.
- Egy csúcshoz annak formalizálása, hogy pontosan egy él fedi: legalább egy él fedi ÉS legfeljebb egy él fedi.
 - Legalább egy: Egyetlen CNF kell hozzá: $(x_1 \vee x_2 \vee \dots \vee x_k)$.
 - Legfeljebb egy: Négyzetesen sok klóz kell hozzá, minden csúcspárra 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)

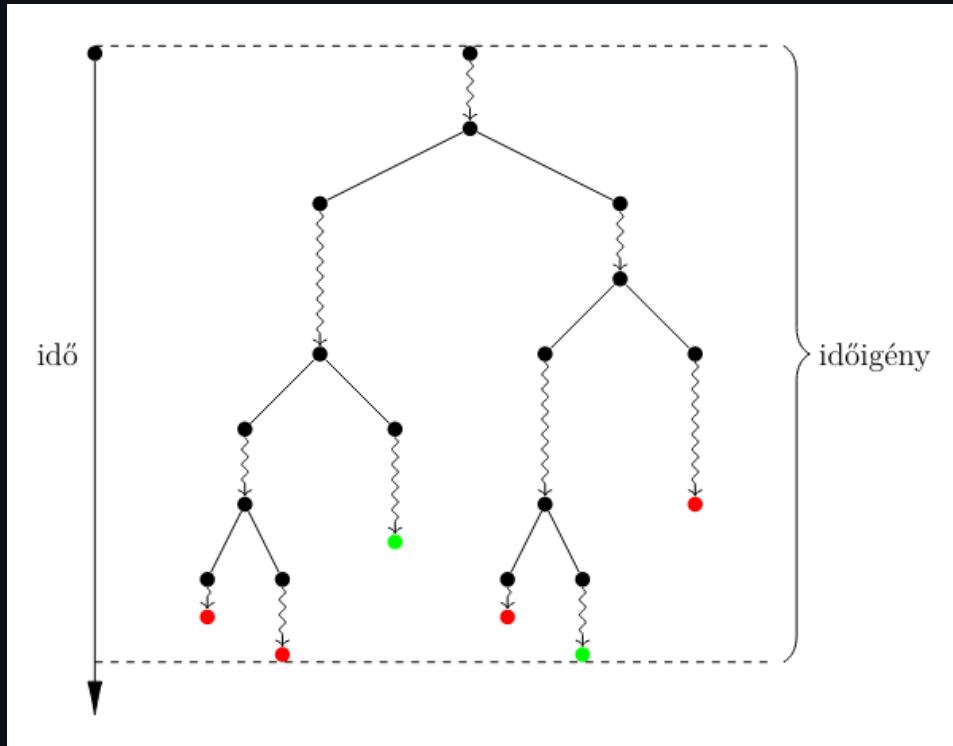
$$\begin{aligned}
& (x_4 \vee x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \\
& \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_7 \vee x_8) \wedge (\neg x_7 \vee \neg x_8) \wedge (x_8 \vee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

Nemdeterminizmus

Nemrealisztikus számítási modell: Nem tudjuk hatékonyan szimulálni.

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

Ezzel nemdeterminisztikusan adunk értéket egy bitnek, amit úgy lehet elképzelni, 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**.

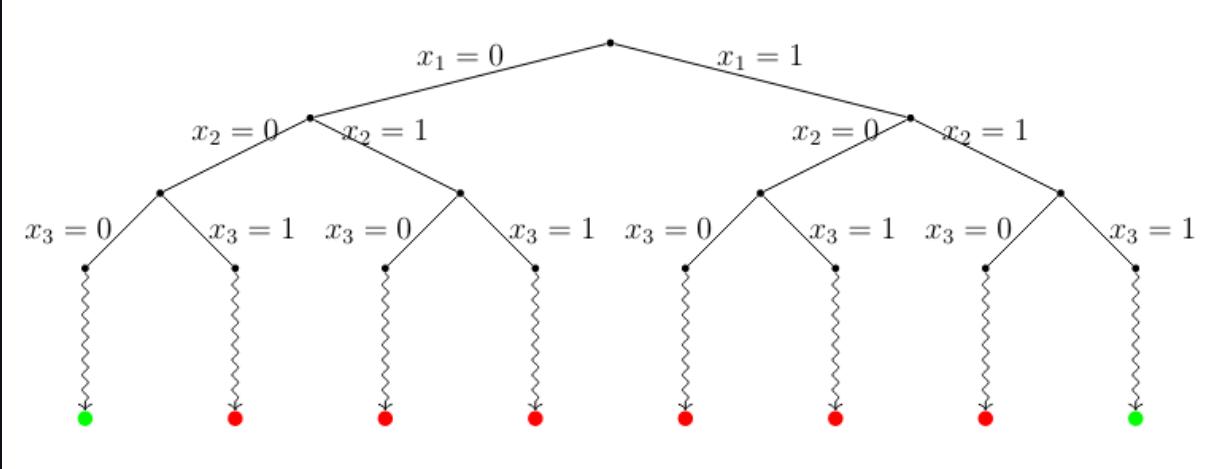
Nemdeterminisztikus algoritmus a SAT-ra

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 inphozhoz a számítási fa:



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

Az NP osztály

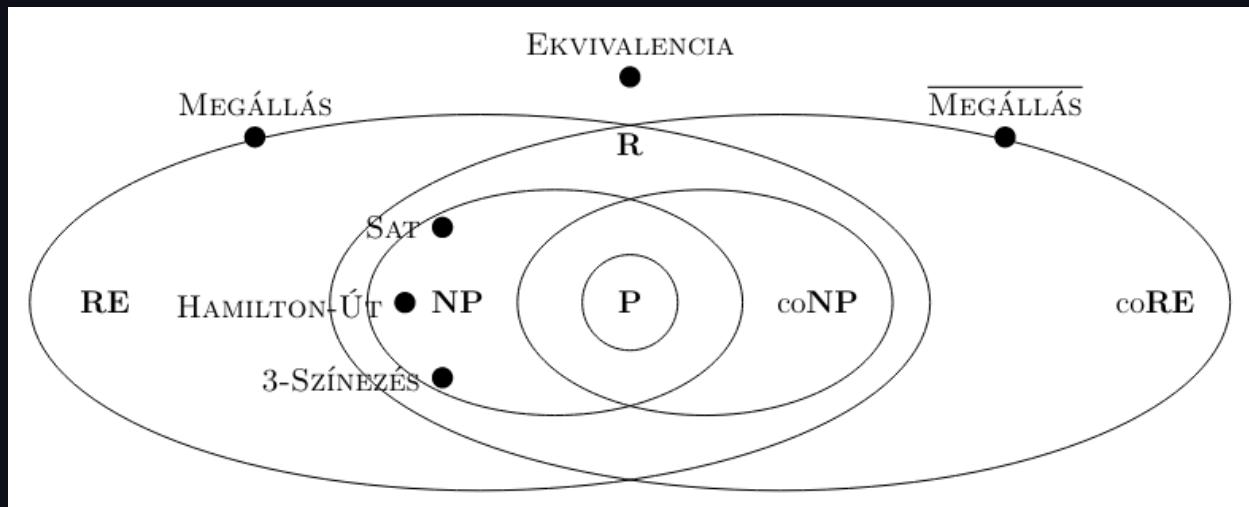
Nemdeterminisztikus algoritmussal polinomidőben eldönthető problémák osztálya.

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.



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 dönteneni, 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 irányíthatlan 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

Párosítás

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?

Hármasítás

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 $\frac{\sum_{i=1}^k a_i}{2}$?

Hátról

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

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:

$$PSPACE = NPSPACE$$

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 eldöntésére. 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** minden elődönthető 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.

Offline, vagy lyuksalagos tárigény

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

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.

Az NL-osztály

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

Immermann-Szelepcsényi tételet 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 $\log^3 = 3 * \log n$ biten, tehát $O(\log n)$ a tárkorlátja.
- Az **input valamelyik elemére rámutatni** egy pointerrel, hiszen lényegében ez is egy 0-tól n -ig értéket felvevő változó.

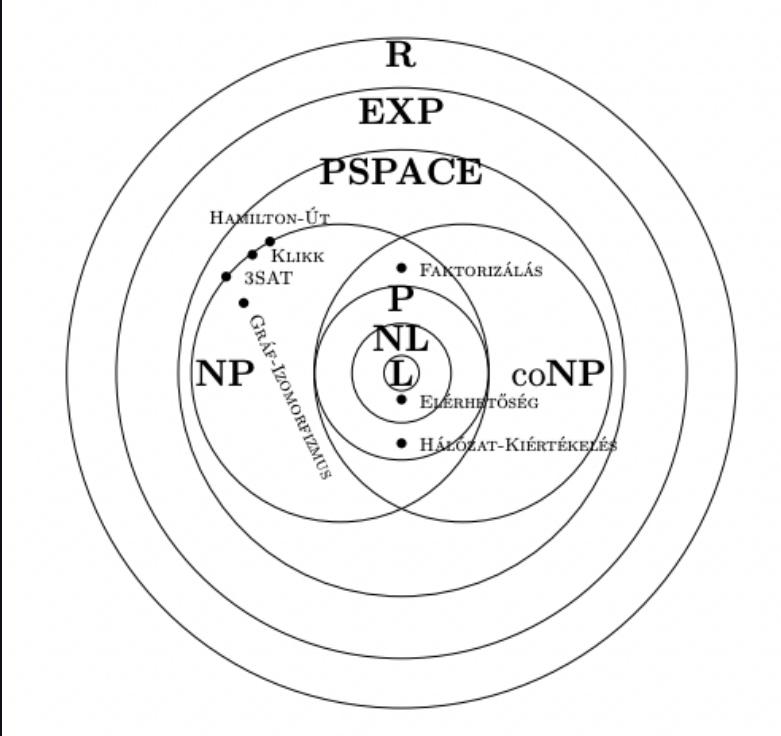
Elérhetőség

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

Nemdeterminisztikusan bele férünk a logtárra. Ekkor "nemdeterminisztikusan bolyongunk" a gráfban, és ha N lépésben elérünk a csúcstig, akkor `true` amúgy `false`. Tehát minden iterációban átlépünk nemdeterminisztikusan minden szomszédra, ha megtaláltuk a cél csúcst, `true`, ha nem tudunk már tovább lépni, vagy lefutott minden az N iteráció, akkor `false`.

Ezek alapján tehát:

$\text{ELÉRHETŐSÉG} \in \text{NL}$



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 lyuksalagos, 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 különböző 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.

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)

PSPACE-teljes problémák

QSAT

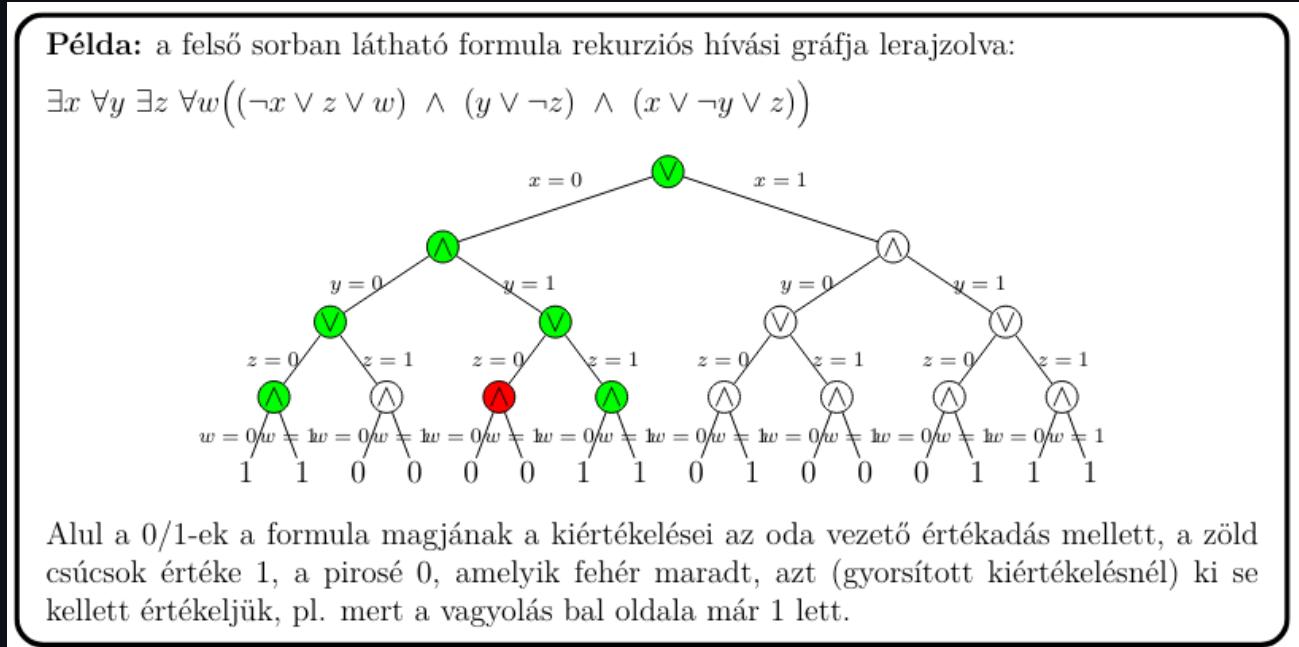
Input: Egy $\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{2m} \phi$ alakú **kvantifikált ítéletlogikai** formula, melynek magja, a ϕ konjunktív normálformájú, **kvantormentes** formula, melyben csak az x_1, \dots, x_{2m} változók fordulnak elő.

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:



Alul a 0/1-ek a formula magjának a kiértékelése 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 bábuját a gráf kezdőcsúcsára.
- Ezután a második játékos lép, majd az első, stb., felváltva, mindenkor a bábut 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

Formális Nyelvek

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

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

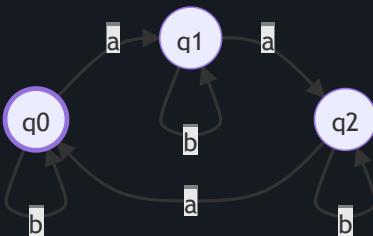
Gráf csúcsai az automata állapotai

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



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

A korábbi példa automata megadása gráffal:



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

Automata megadása táblázatként

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

A korábbi 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 ás 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 megszokott regexeknél

Felismert nyelv

Az $M = (Q, \Sigma, \delta, q_0, F)$ automata által felismert nyelven az $L(M) = w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon)$ 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 mindig állapotoknak 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 megnöhet (akkor 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 $L(M) = w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon)$ 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 definiálhatóvá tehetők "csapda" állapot bevezetésével, anélkül, hogy a felismert nyelv megváltozna.

- Felvészünk 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 automaza 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, amikből 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.

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 őt 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 legszűkebb 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 tehát 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

$3 \rightarrow 1$ az ekvivalencia tételeben.

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

Ez R struktúrája szerinti indukcióval belátható.

Reguláris nyelvek felismerhetők automatával

$1 \rightarrow 2$ 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

$2 \rightarrow 3$ az ekvivalencia tételeben

Minden, automatával felismerhető nyelv reprezentálható reguláris kifejezással.

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 nyelvezet 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.

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.

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.

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

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 -fel 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 átirással (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:

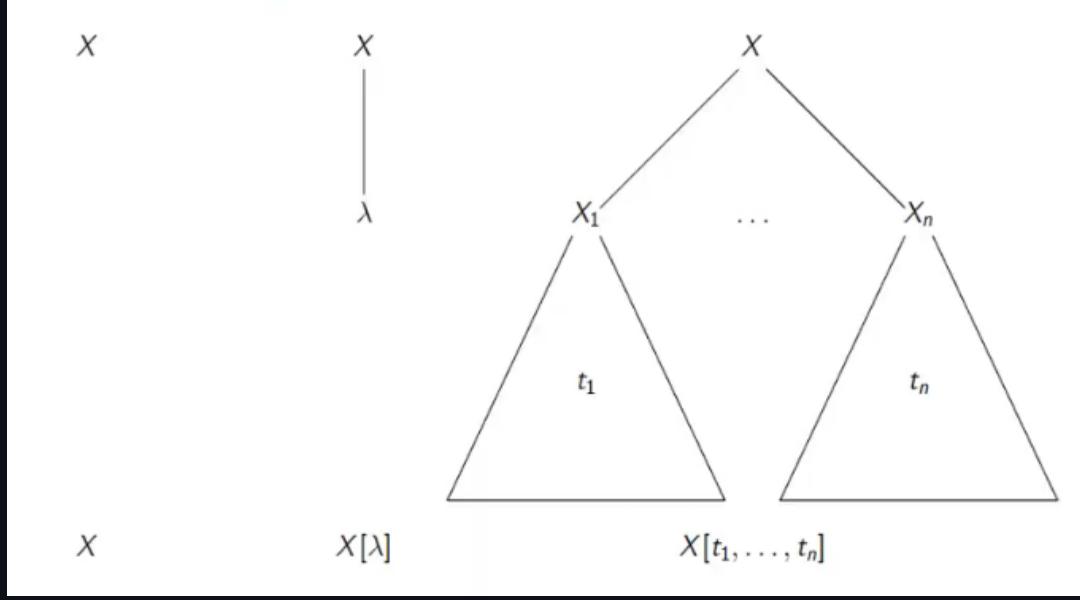
$$L(G) = w \in \Sigma^* \mid S \Rightarrow_G^* w$$

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

Derivációs fák, kapcsolatuk a derivációkkal

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

- Az a fa, amelynek egyetlen szög pontja (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 szög pontból 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 leolvásásával kapunk.

Az összefüggés derivációs fák, és derivációk között

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 tételek 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 nemterminálist lehet csak helyettesíteni)
 - van olyan S gyökerű derivációs fa, amelynek határa w .

Generált nyelv definíálása derivációs fákkal

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

Közeliítő és szimbolikus számítások

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

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

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

Lineáris egyenletrendszer 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 beszorzássá.
- Egy egyenlet konstansszorosának hozzáadása egy másik egyenlethez.

Egyenletrendszer megoldása

Ilyen átalakításokkal próbálunk háromszögmátrixot vagy diagonális mátrixot létrehozni. Ez azért jó, mert ilyen alakban az egyenletrendszer könnyen megoldható:

\$\$

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

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

=

$$\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

\$\$

Ilyen az esetben a megoldás könnyen kifejezhető:

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

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 $G_j \in \mathcal{R}^{n \times n}$ **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_{\underline{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 harmadik koordinátája is nulla.

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 =

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

$G_{\underline{j}} =$

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

\$\$

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 ($a_{11} = et$ módosítottam 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 oszloban ténylegesen kinullázódott két sor, már csak a második oszlopan kellene az utolsó sort kinullázni, és egy könnyen megoldható egyenletrendszer együtthatómátrixát kapnánk.

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 $L U x = b$ egyenletrendszeret 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 "permutálandó" 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

Ríkka 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.

Pozitív definit = minden sajátértéke pozitív

Matlab implementáció

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

A \top operátor transponál, a \backslash pedig: $\top R \backslash y := A z$ $R x = y$ 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.

Lineáris egyenletrendszer 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, mindeneket megkapott értéket behelyettesítjük a kifejezett báltozó 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 lépésben 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 kellene?

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.

Emllett 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

$$\left(\begin{array}{c} \rho(A) \\ \rho(A) \end{array} \right)$$

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.

Matlabban

```
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 `./` komponensenként való osztás.

Az egyes iterációban kapott eredmények:

- `r`: Rayleigh-féle hányszámosval kapott eredmény.
- `x`: Komponensenkénti becsléssel kapott eredmény.

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

Érintő 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.

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.

Így 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 **köztefogják** a x^* gyököt.
- Ha $f'(x^*) > 0$, és $f''(x^*) > 0$, akkor x^* -nál nagyobb, de ahoz közeli kezdőértékekkel **szigorúan monoton konvergencia** érhető el.

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.

Konjugált gradiens eljárás

Optimalitálás elvein alapuló módszer.

Szimmetrikus pozitív definit mátrixú lineáris egyenletrendszer megoldására alkalmas.

Pontos aritmetikával ugyan váges sok lépésben 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 lépésközt, 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 lépéseket, 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 lépésben, de felismerték hogy ez a meredek falú, enyhén lejtő völgyeszerű 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 lépésenkénti megfelelő irányváltoztatással 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 értékek 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;
    r = rr;
end
```

```

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 ártákül `r`-nek (`rr`-t).

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 pontokon 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 meg a 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 függvénygel.

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.

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.

Kvadratúra-formula

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** nevezünk.

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

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:

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

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

|| Az alappontról 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 általánostásaként 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.

Ekvidisztáns alappontokat használnak.

|| Azaz a szomszédosak között ugyanannyi a távolság.

Ha az integrálhatárai szerepelnek az alappontok között, akkor **zárt**, ha a határok nem alappontok, 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.

Logika és informatikai alkalmazásai

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

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 **válzotó**, 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).

- **Egyésgklóz:** 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.
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.

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 igazságátblával.

* 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-függvénynek 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.

Bármennyik 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
- alkalmaz 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

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

Lényegében ezzel azt írtuk le, hogy az x_n értéke 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 alkalmaz 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.

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 ilyan 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 értékkedés 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 írunk.

Leválasztási következtetés, vagy modus ponens

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

Ha F -et, és $F \rightarrow G$ -t már levezettük, azaz az eredeti formuláknak 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 értékkedés. Definiáljuk \mathcal{B} értékkedást 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: $\mathcal{B}(F) = \mathcal{A}(F[p_1/F_1, \dots, p_n/F_n])$

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

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 $\Sigma \vDash F_i$
- Ha $F_i \in \Sigma$, akkor $\Sigma \vDash F_i$
- 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 $\Sigma \vDash F_i$ 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 $\Sigma \vDash F_j$ és $\Sigma \vDash F_k$ (feltételezzük, hogy a korábban felvett formulák már logikai következmények)
 - Tehát $\Sigma \vdash F_j, F_k$
 - MP def miatt $F_k = F_j \rightarrow F_i : \Sigma \vdash F_j, F_j \rightarrow F_i$
 - A leválasztási következtetés: $F_j, F_j \rightarrow F_i \vDash F_i$
 - A tranzitivitás miatt tehát $\Sigma \vDash F_i$ (tranzitivitást kihasználjuk, mivel $\Sigma \vdash F_j, F_j \rightarrow F_i \vDash F_i$)

Így a Hilbert-rendszer egy helyes következtető rendszer.

Teljesség

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

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

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

Ekvivalens 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 halmmazt fel lehet fűjni."

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: $\Sigma \vDash F \Leftrightarrow \Sigma \vdash F$

Azaz most itt egyszerre van belátva minden F esetén, hogy $\Sigma \vDash F$ és $\Sigma \vdash F$.

Sorban minden ekvivalenciát tovább feltünk ekvivalencia mentén:

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

Ennek az alapja egy téTEL: $\Sigma \models F$ pontosan akkor igaz, ha $\Sigma \cup \neg F$ kielégíthetetlen. Ez van itt felírva 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-rendszere helyes, és teljes**. Azaz tetszőleges Σ halmzból Hilbert rendszerében **pontosan** Σ következményei vezethetőek le.

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.

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 $\Sigma \models C_1, C_2$ (nyilván, összevagyonlani 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 kövezkeménye 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 **oldalklózájának**.

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 benn 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 $\Sigma'|_{l=0}$ és $\Sigma'|_{l=1}$ 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 szerinte van \square -nak egy C_1, C_2, \dots, C_m lineáris rezolúciós vezetése, melynek $C_1 = C - l$ a bázisa.
- "Visszaemelve" a $\Sigma|_{l=0}$ cátolatot Σ' fölötti vezetéssé, az új vezetésben 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 egységeklóznál jár a lineáris rezolúciós vezetés.
- 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 cátolata
- Az előző fázisban kapott l egységeklózt tudjuk rezolválni ezzel a C klózzal, tehát a $\Sigma'|_{l=1}$ cátolatát "fel tudjuk emelni" Σ' fölötti vezetéssé.
- A felemelt vezetés végén vagy \square -t, vagy \bar{l} -t kapunk. Utóbbi esetben még egyszer rezolválunk l -lel mint oldalklózzal, és kész vagyunk

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

* Elsőrendű logika alapfogalmak

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

Alapterm: Olyan term, amiben nincs változó

* 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" értéket ad, 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 $A^n \rightarrow \{0, 1\}$ predikárum
 - 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 befejtetésítjük 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

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 $Q_1x_1Q_2x_2\dots Q_nx_n(F)$ 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: $\neg\exists xF \equiv \forall x\neg F$
- $\exists xF \vee G \equiv \exists x(F \vee G)$
 - Ha x nem szerepel G -ben szabadon, ezért kell előtte kiigazírani!

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 eltűntetjük 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 vűtétozók.

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 konjukciójá, pl.: $(p(x) \vee \neg q(y, c)) \wedge \neg p(x)$

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

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.

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 szerinte 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, köztük \square -t)

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

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 klózból csak pozitív előfordulásokat, a másikból csak negatívakat. Így lesz esély arra, hogy egyesíthető legyen.

- Ha C egyesíthető az s legáltalánosabb 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.

Algoritmus

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

| Atért 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 rálerül a listára, akkor Σ kielégíthetetlen.

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

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

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

Helyesség

- A helyesség a rezolvensképzés helyességből 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 felítjük 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 klóz csak önmagának példánya, így $C_n = \square$ kell legyen.

Programozási nyelvek

A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai.

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)

Imperatív programozás

- Az imperatív programozás olyan programozási paradigmá, 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 gépnyelvei voltak (assembly)
 - egyszerű utasítások -> könnyebb hardwares megvalósítás, de az összetett programok létrehozása nehezebb

Procedurális programozás

- A megoldandó programozási feladatot kisebb egységekből, avagy 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 eljáráshívásokat 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
 - Érhetőség: 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 interfész: 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: kiterjeszhető, más szóval bővíthető az általa nyújtott szolgáltatások száma
- Újrafelhasználhatóság: ugyanazokat a programeleket ne kelljen többször elkészíteni, ügyeljünk viszonylag általánosítható 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 paradigmája
- Az objektumok egységbe foglalják az adatokat és a hozzájuk tartozó műveleteket (egységbe zárás).
- A program egymással kommunikáló objektumok összességeből 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.
 - Összetartozó 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 osztályváltozókhöz és paramétereikhez férhetnek hozzá, példányváltozókhöz nem.
 - Példánymetódusok: példány szintű metódusok, hozzáférnek az adott példány összes adatához és metódusához, és paramétereik is lehetnek.
- Kompozíció, öröklődés, interfések
 - Kompozíció: az objektumok lehetnek más objektumok mezői
 - Öröklődés:
 - Osztályok közötti alárendeltségi 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 adattagját és metódusát
 - Több programozási nyelv megengedi a többszörös öröklőést

- Egyes nyelvekben, mint a Java és a C# megttiltható a leszármazás egyes osztályokból (Javában final, C#-ban sealed a kulcsszó)
- Interfészek:
 - Nem tartalmazhatnak megvalósítási részleteket, csak előírhatják bizonyos metódusok jelenlétét, 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*

Dekleratív programozás

- A specifikáció 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, megróbálják 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
 - Rekurzív 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 megszokott 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 nevezzük

- A program futása minden esetben egy az állításokból következő tételes 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 célfórmulát (célklózt), ezután a program ellenőrzi, hogy a célklóz a logikai (forrás)program logikai következményei között van-e
 - Gyakran használják mesterségesintelligencia-alkalmazások megvalósítására, illetve a számítógépes nyelvészeti eszközöként

Párhuzamos programozás

- Egyszerre több szálon történik a végrehajtás
- Végrehajtási szál: folyamat (process)
- Előnyei:
 - Természetes 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
 - Nem determinisztikus: 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 holtpontba is kerülhet, erre különös figyelmet kell fordítani
 - Elemi és összetett folyamato
 - Csatornák: két folyamat közötti adatátvitelre szolgál

Rendszerfejlesztés 1.

1. Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei.

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 meggátolja hogy elveszítsük az uralmat a projekt felet
- Adaptálás adott projekthez és környezethez

Folyamat elemek:

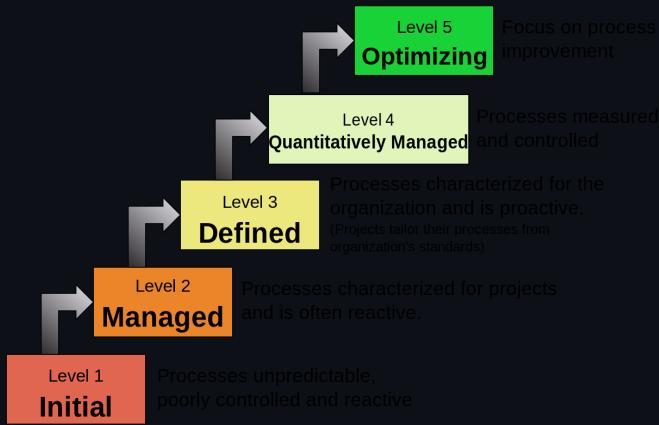
- Fő elemek
 - Feladatok, termékek
 - Határidők, átadandók
- Kiegészítő elemek
 - Projektmenedzsment
 - Konfiguráciomenedzsment
 - Dokumentáció
 - Minőségbiztosítás, kockázatmenedzsment
 - Mérés

Folyamat fejlettsége

- Egy szerveztnél 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 absztraktiós szinteken
- *Tevékenységei:*
 - Architektúra tervezés: alrendszerek meghatározása
 - Absztrakt specifikáció: alrendszerek szolgáltatásai
 - Interfész tervezés: alrendszerek 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 rendszereknél részletes tervezés alapján
- Programozás (kódolás): adottság kell hozzá, személyes technikák, stílusok
- Minőségbiztosítás érdekében kódolási stílust lehet megkövetelni
- Nyomkövetés (debugging): hiba lokalizálás, eltávolítás, újratesztelés, programszöveg manuális vizsgálata, eszköztámogatás

• 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 interfészek 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)

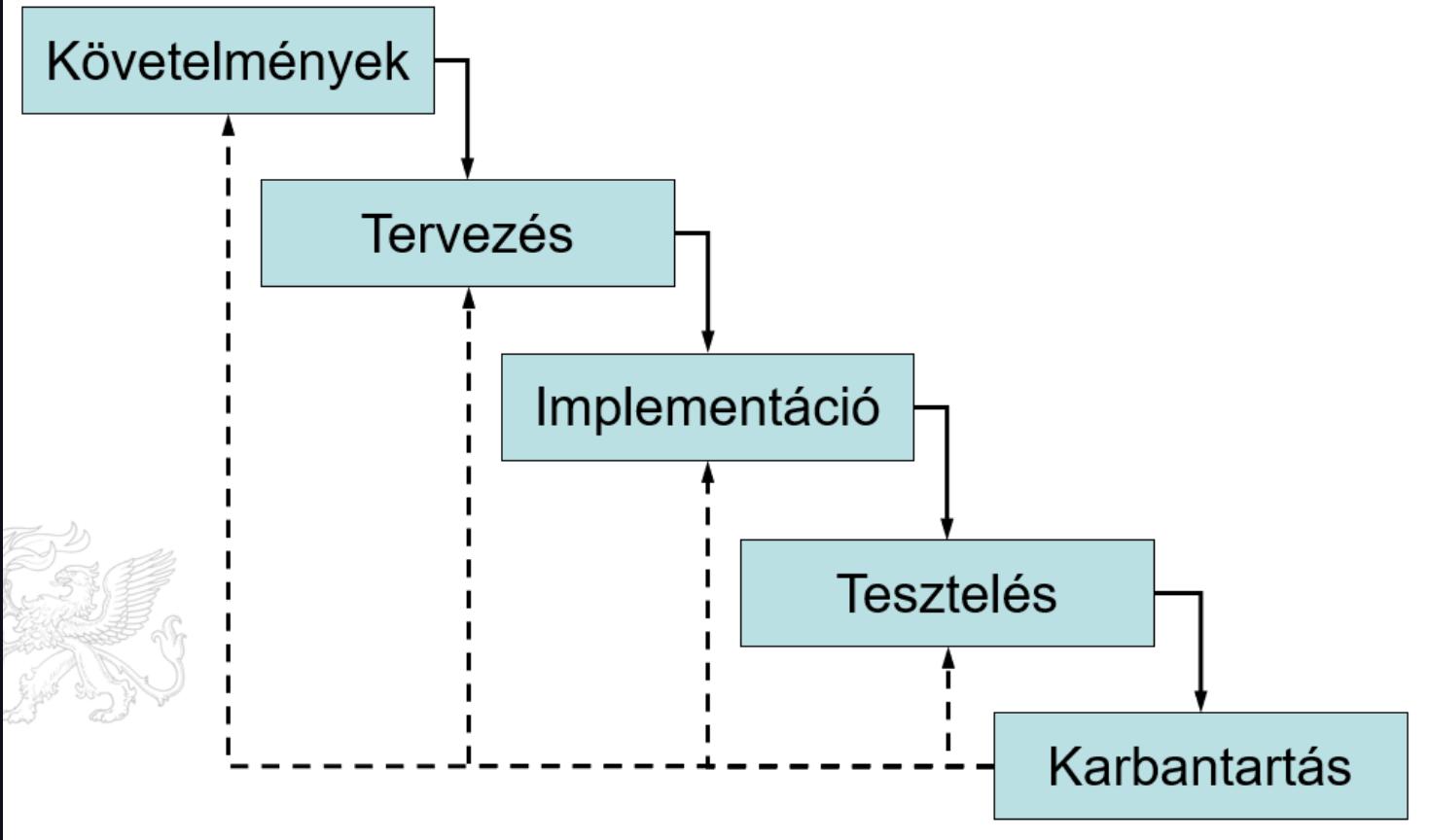
A folyamat modelljei

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 részvétel
- Programozás csoportos tevékenység (többen ülnek egy képernyő előtt)
- Az utóbbi időben sok kiegészítés készül, sajnos kezdi kinöni 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 evolúciós 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
- Hárányok
 - Kompromisszumok követelményekkel szemben
 - Evolúció során a felhasznált komponensek új verziói már nem integrálhatók
- Objektumorientált paradigma jó alap

- UML használata
- Rational Unified Process (RUP) egy iteratív, inkrementális és komponens alapú folyamat

Formális módszerek

- Vízesés modellre hasonlít
 - Specifikáció: formális, matematikai apparátus
 - Kidolgozás: ekvivalens transzformációk
 - Verifikáció: hagyományos értelemben nem szükséges
- Kisebb lépésekkel 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 mindenkor formalizálhatók

Cleanroom módszer

- Fejlesszünk (bizonyítottan) hibátlanul és akkor nem kell tesztelni
- Csak rendszertesztelés kell, modulhelyesség bizonyított
- Az egyik legismertebb formális módszer
- Inkrementális fejlesztésen alapul
- Fejlesztőszközök egyszerűbbek, szigorúbbak (Pl. csak strukturált programnyelvek)
- Dobozokkal reprezentálják a rendszert
- Képzett, elkötelezett 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

2. Projektmenedzsment. Költségbecslés, szoftvermérés

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 menedzselése

- Szoftverfejlesztő szervezet legnagyobb vagyona az emberek
 - Szemelmi tőke
 - Lehető legjobban kamatozzon!
- Sok projekt bukásának legfőbb oka a rossz humánmenedzsment

- Egyik legfontosabb feladat az emberek motivációja

- Szociális szükségletek, megbecsülés, önmegvalósítás igénye

Csoporthoz köthető

- 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 funkciót pont)
- 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 kombinálás
 - **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

Szoftverköltség becslése

- Projekt tevékenységeinek kapcsolódása a munka-, idő- és pénzköltségekhez
- Becsléseket 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 meggy 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 = 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 = [LOC*B ^{0.333} /P] ^{3*} (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 specifikációnak) 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, funkcionális kezelése megoldott és megismételhető. (Bevezési 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-kötéssel 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

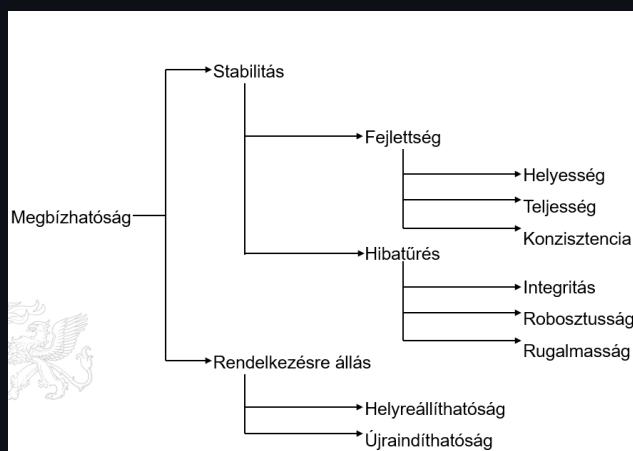
- Számon van tartva a hiba felvezője, a hiba súlyossága, meg van-e javítva stb.
- Fontos a hiba életútjának rögzítése
- Általánosabb: változtatás-menedzsment
 - CR (change request) adatbázis nyilvántartása

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 vezetői 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 (belsı jellemzı) és (külsı) jellemzı közötti kapcsolatokra fel kell állítani egy modellt (Sok projekt esettanulmányának vizsgálata)
- Példa jellemzı 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, kategorizálásuk 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 szümmolni, 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 interfészek 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
- Hárnya, 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

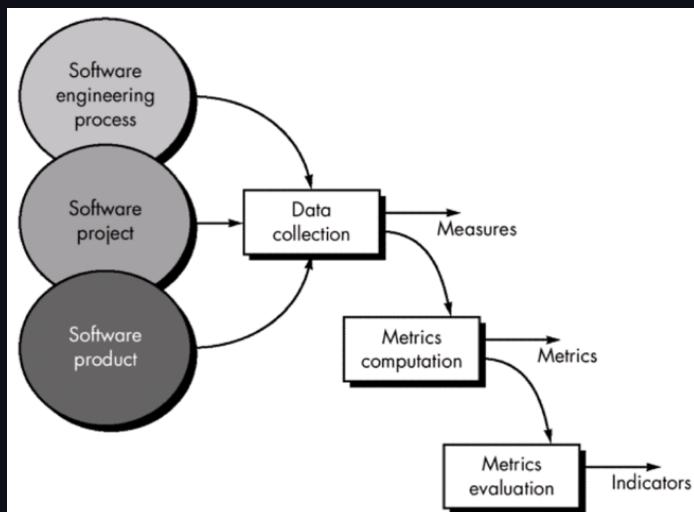
Metrikák alkalmazása szoftver-minőség mérésére

- Elsődleges a hibák és hiányosságok mérése
- A minőség mérése:
 - Helyesség (hiányosság/KLOC)
 - Karbantarthatóság (nincs mérőszám)
 - Integritás: külső támadások elleni védelem
 - Fenytegettség: annak valószínűsége, hogy egy adott típusú támadás bekövetkezik egy adott időszakban
 - Biztonság: annak valószínűsége, hogy egy adott típusú támadást visszaver a rendszer
 - Integritás = $\Sigma [1 - (\text{fenyegetettség} \times (1 - \text{biztonság}))]$ (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?

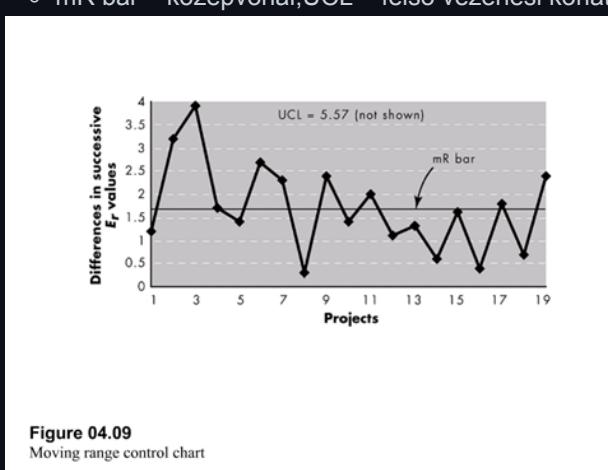
Metrikus baseline

- Kell egy hosszabb idejű, több projekten alapuló összehasonlízási 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



Összegzés

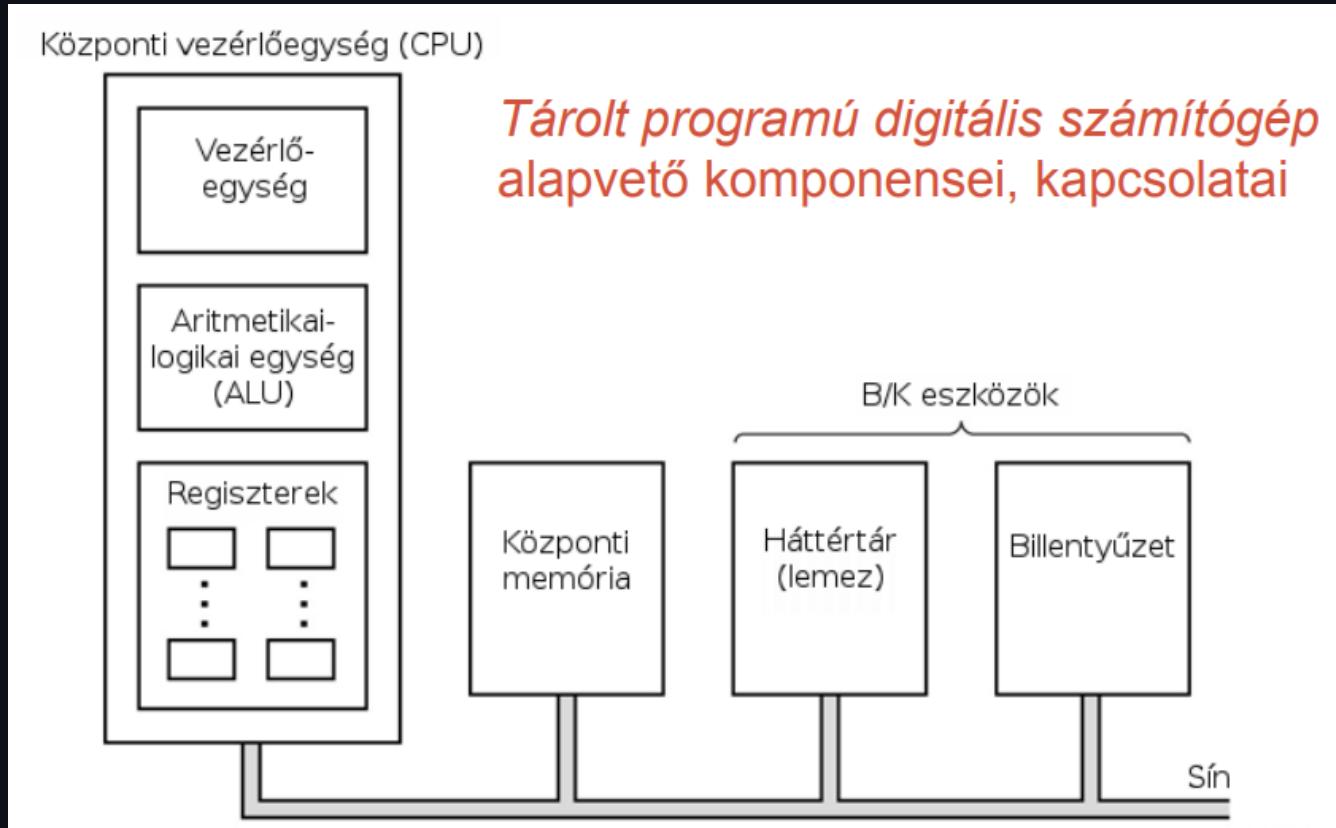
- A metrika alapú mutatók fontosak a menedzsment és a technikai vezetők számára
- Folyamat metrikákat stratégia szempontok alapján kell kiértékelni
- Méret- és funkció pont alapú metrikákat széleskörűen használnak az iparban.

- A szoftver minőség javításának alapja a metrika alapú „baseline”

Számítógép architektúra

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.

Neumann-elvű gép sematikus váza



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

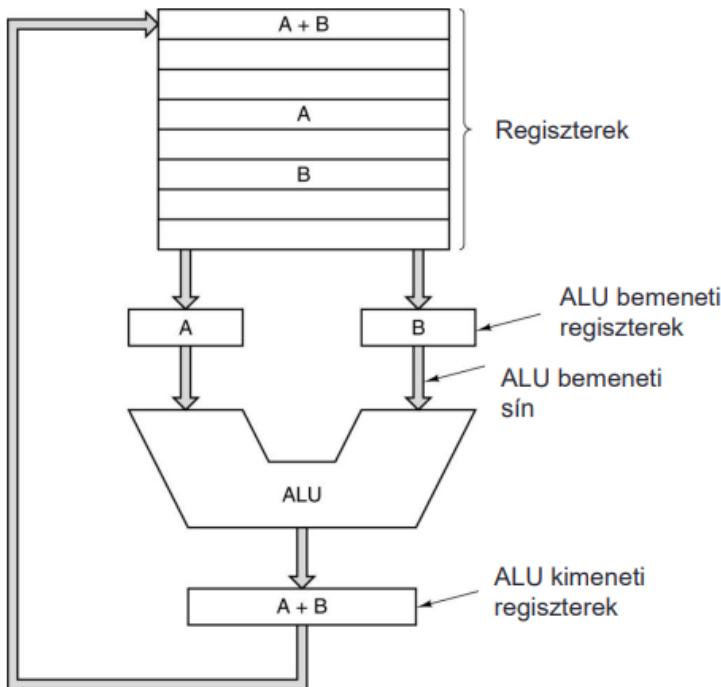
Külső sín: részegységek összekötése (kábel, huzalozás), adatok, címek, vezérlőjelek továbbítása különböző buszokkal

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.)

Adatút



Példa

- Összeadás elvégzése ALU-val
- Legtöbb utasítás
 - Regiszter-memória
 - Regiszter-regiszter
- Adatciklus
 - Két operandus ALU-n való átfutása és az eredmény regiszterben tárolása

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 memoriára 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

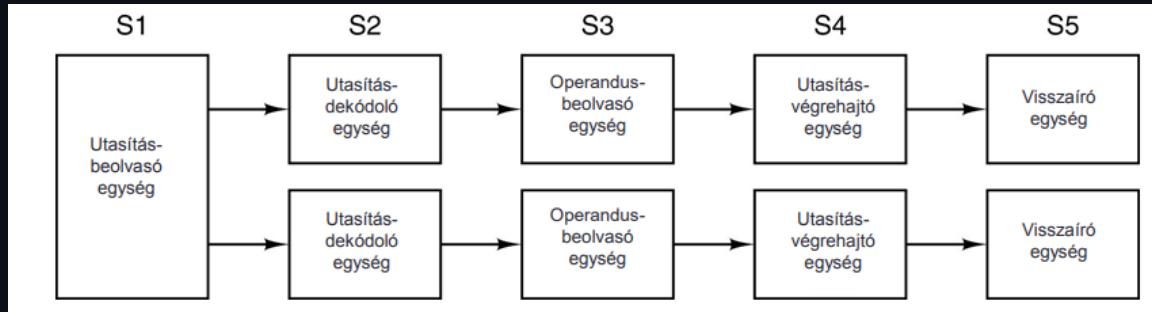
Késleltetés: utasítás végrehajtásának időigénye

Áteresztőképesség: MIPS (millió utasítás mp-enként)

Utasításszintű párhuzamosság

Párhuzamos csővezetékek

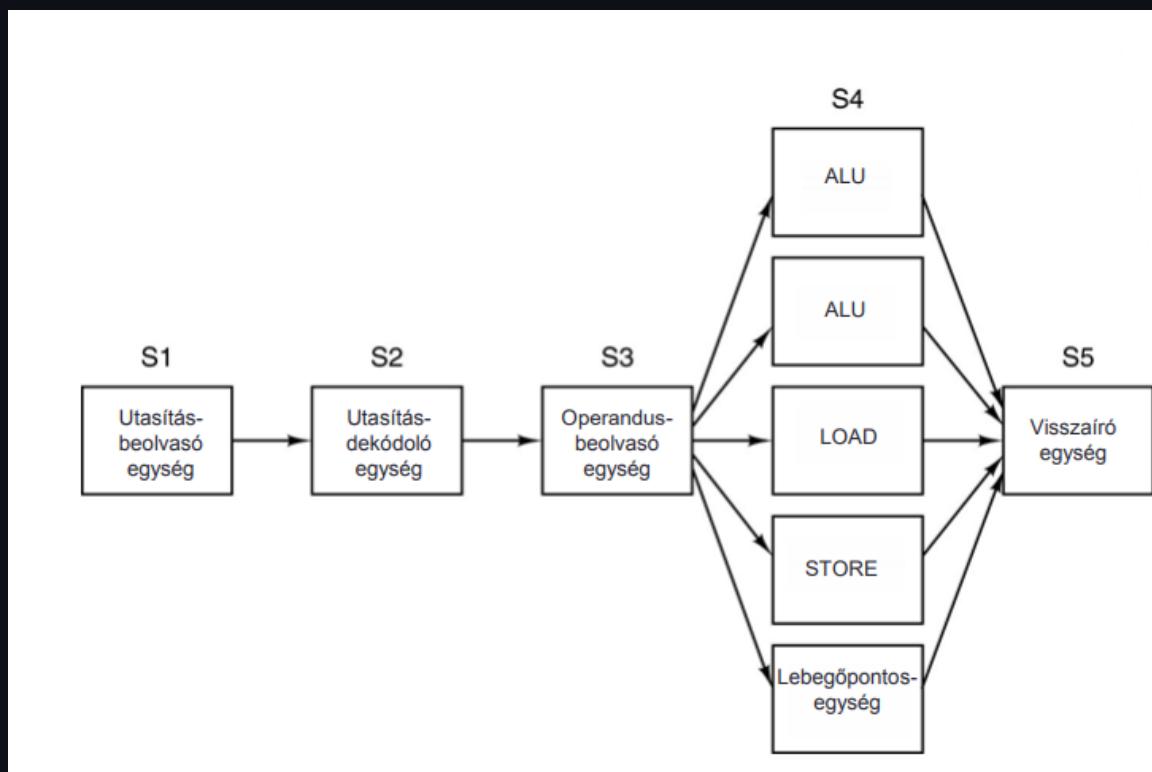
- Közös utasítás-beolvasó egységgel
- A csővezetékek saját ALU-val rendelkeznek (párhuzamos végrehajtás, ha nincs erőforrás-használat ütközés)
- Általában 2 vagy 4 csővezeték



- 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

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

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 integráltak egy RISC magot (80486-tól) a leggyakoribb utasításoknak

Korszerű számítógépek tervezési elvei

- **Minden utasítást közvetlenül a hardver hajtson végre**

- A gyakran használtakat mindenkorban
- 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 utasításformája, párhuzamosítást segíti

- **Sok regiszter legyen**

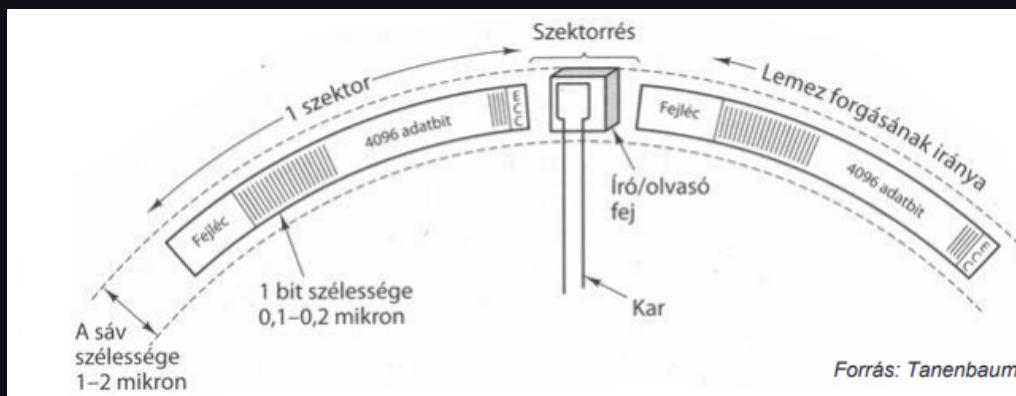
- Számítások során ne kelljen a lassú memoriába írni

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).

Mágneslemezek

Részei:

- Mágneszhető 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

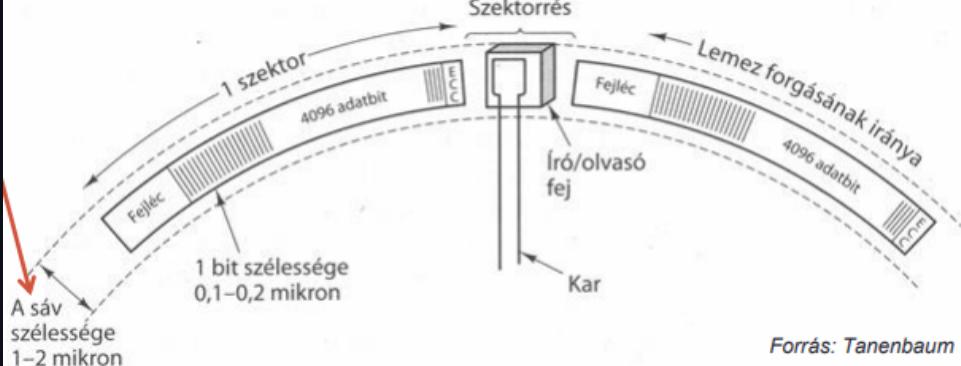
Írás: Pozitív vagy negatív áram

az indukciós tekercsben, alemez adott helyen mágnesződik

Olvasás: Mágneszett terület felettes haladva pozitív vagy negatív áram indukálódik a mágneses polarizációnak megfelelően

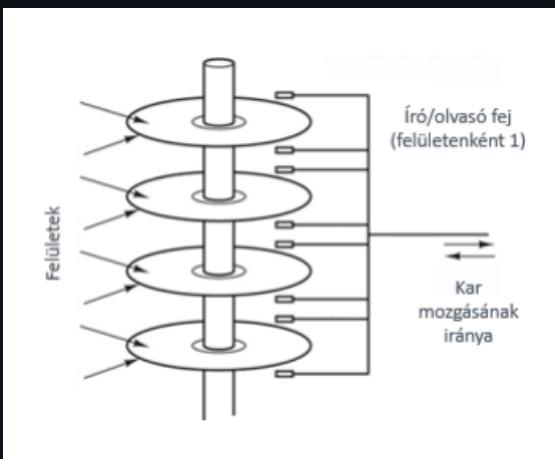
Adattárolás:

- **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
 - Szektorrés
- **Lineáris adatsűrűség**
 - Kerület mentén, 50-100 ezer bit/cm
- **Merőleges rögzítés**
 - Tárolás hosszirány helyett „befelé” történik
- **Kapacitás**
 - Formázott és formáztatlan



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)
- Lemezvezérlő 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 sorozatával
 - Bufferelés (gyorsítás)

Típusok:

SCSI

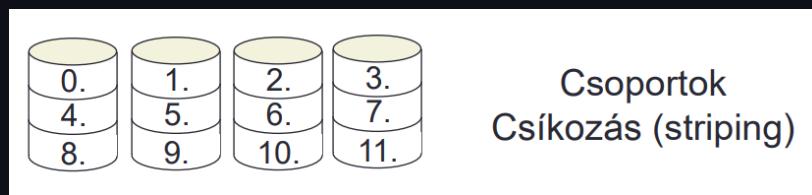
- Small Computer System Interface
 - „kis számítógép-rendszer interfésze”, kiejtése „szkazi”
 - Olyan szabványegyüttes, melyet számítógépek és perifériák közötti adatátvitelre terveztek
 - A SCSI szabványok definiálják a parancsokat, protokollokat, az elektromos és optikai csatlófelületek definíciót
 - A SCSI merevlemezek fizikai mérete ugyanakkora, mint az ATA és SATA winchesterek – lemezeinek átmérője 3,5 inch –, viszont percenkénti fordulatszáma azokénál 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 ként 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**

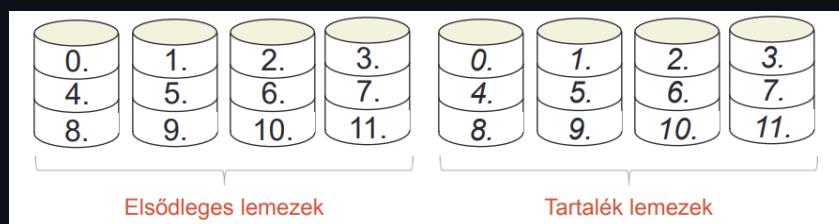
- RAID 0

- Adatok párhuzamos tárolása a lemezen
 - 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 ($4 \text{ adatbit} + 3 \text{ paritásbit} = 7 \text{ tárolandó bit}$; 7 szinkron lemez kell)
 - Hamming távolság: két azonos hosszúságú 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 jól
- Vezérlőnek plusz munka a Hamming kód kezelése!
- Hamming kóddal pótolható a kieső lemez tartalma -> hibatűrő



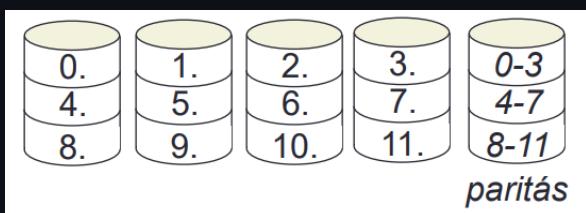
- RAID 3

- 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 lemezről, nagyon leterheli a paritásmeghajtót



- RAID 5

- RAID 4-hez hasonló elv, de nincs dedikált paritásmeghajtó
- A paritásbiteket körbejárásos módszerrel szétesztja a lemezegységek között

- Legalább 3 lemezegység kell, legalább 4 ajánlott

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 pozicionálni mint a merevlemezknél (koncentrikus körök helyett spirál)
- Meghajtó szoftvere nagyjából a célterület fölé 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 ötvözöt
- Kétféle stabil állapot: kristályos és amorf (más fényvisszaverő képesség)
- 3 eltérő energiájú lézer
 - Legmagasabb energia: megolvad az ötvözöt → 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 filmparai 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

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 → tintacséppet présel ki
 - Hővezérlésű vagy festékbuborékos: Fűvökákban kis ellenállás, amely feszültség hatására felhevül, a festék felforr és elpárolog, túlnyomás keletkezik, papírra kerül, fűvökát lehűtik, a keletkező vákuum újabb tintacséppet 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 tervezek 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 tengelyt 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

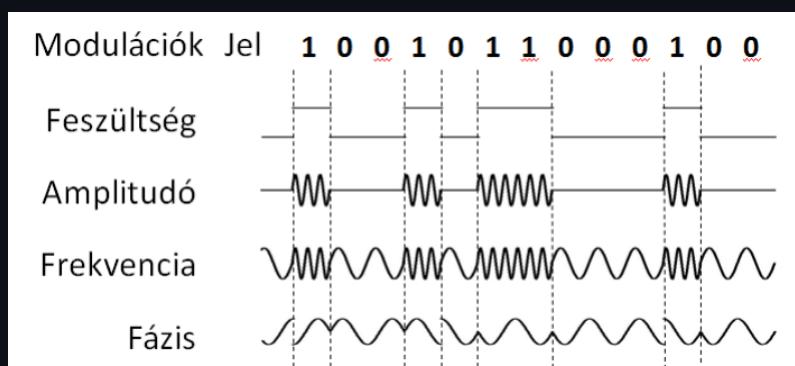
Billentyűzet

- Egy-egy billentyű leütése áramkört zár
- Megszakítás generálódik
 - Az operációs rendszer kezeli és továbbítja a programoknak

Telekommunikációs berendezések

Modem

- Adatkommunikáció analóg telefonvonalon
 - Az analóg vonalat hangátvitelre találták ki
 - Adatátvitelhez: vivőhullám (1000-2000 Hz-es szinuszos hullám)
 - A bitek csak sorasan, egymás után vihetők át
 - 1 bájt átvitele: start bit + 8 adatbit + stop bit = 10 bit



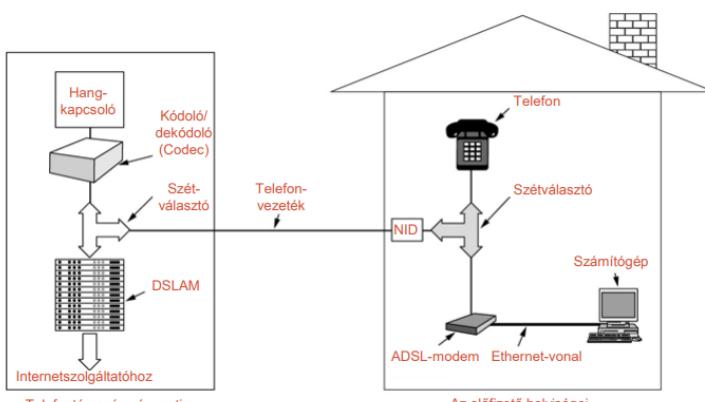
- Modulációk
 - amplitúdó, frekvencia módosítása

- Fázis: dabit kódolás
 - 45, 135, 225 és 315 fokos fáziseltolódások az időintervallumok elején
 - 2 bit átvitеле egységnyi idő alatt (45 fok: 00, 135 fok: 01, ...)
- Kombinálva is használhatók
- Definíciók
 - Baud: Jelváltás / másodperc
 - 1 jelváltás több bitnyi információt is hordozhat (lásd dabit kódolás)
 - Adatátviteli sebesség: bit / másodperc
 - Jellemzően 28800 vagy 57600 bit / mp, jóval alacsonyabb baud értékkel!
 - Kommunikációs vonal típusa
 - Full-duplex (kétirányú kommunikáció egyidőben)
 - félf-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 adatátvitelre: 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 osztozik
 - 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