ÓBUDAI EGYETEM

BÁNKI DONÁT
GÉPÉSZ ÉS BIZTONSÁGTECHNIKAI MÉRNÖKI KAR

Tóthné Dr. Laufer Edit

Az információ- és kódelmélet alapjai

ÓE-BGK 3075

Budapest, 2019.

Lektor: Dr. habil. Szénási Sándor

ISBN: 978-963-449-135-4

Tartalomjegyzék

Bevezetés

Ez a jegyzet az Óbudai Egyetem Bánki Donát Gépész és Biztonságtechnikai Mérnöki Karán tanuló mechatronikai mérnök BSc szakos hallgatók számára íródott. Célja a hallgatók bevezetése az „Informatika alapjai” tárgyhoz kapcsolódóan az információ és kódelmélet alapvető fogalmaiba, megfelelő alapozást nyújtva a későbbi informatikai tanulmányaikhoz. A jegyzet elsősorban az előadásokon elhangzó anyag magyarázatát tartalmazza és bár az előadásokon való részvétel nélkül is elégséges magyarázattal szolgál, ennek ellenére az órák látogatása javasolt, hiszen ott szóba kerülhetnek olyan témakörök, melyeket a jegyzet nem tartalmaz. Felépítését tekintve a fejezetek mindig a probléma megfogalmazásával kezdődnek, majd ennek megfelelően elsőként az egyes anyagrészek elméleti hátterét mutatjuk be, a szükséges fogalmak bevezetésével, nagy hangsúlyt fektetve az egyes tananyagrészek ok-okozati összefüggésére. Az elméleti áttekintést gyakorlati példák követik a könnyebb megértés segítése érdekében, illetve a téma iránt érdeklődő hallgatóknak lehetőséget ad az egyes témakörök tovább gondolására is. Az anyag matematikai hátterébe csak olyan mélységben megyünk bele, ami a megértés szempontjából feltétlenül szükséges és precíz matematikai definíciók, illetve bizonyítások helyett azok egyszerű, kevésbé matematikai érdeklődésű hallgatók számára is érthető leírását adjuk meg. Az egyes módszerek esetén számos esetben azok algoritmusa is megtalálható a jegyzetben. Ezek az algoritmusok az általános végrehajtás menetét adják meg programozási nyelvtől függetlenül, mondatszerű leírást alkalmazva. Elsődleges céljuk a megértés könnyítése, de esetenként érdemes implementálni is őket egy választott nyelven.

A fejezetek rövid áttekintése

A jegyzet felépítése az „Informatika alapjai” tárgy tematikáját követi, az egyes fejezetek egymásra épülnek, ezért azok áttanulmányozása is a megadott sorrendben ajánlott. Az egyes fejezetekben feltételezzük az előzőek ismertét, hiszen az ott ismertetett fogalmakra is hivatkozunk, illetve bizonyos összefüggések csak a korábbi fejezetek ismeretében érthetők.

Az 1. fejezetben az informatika fogalmát definiáljuk hagyományos és tudományos értelemben is. Áttekintjük az informatika fejlődési tendenciáit,

megnézzük miből fejlődött ki, merre tart és hogyan követhetők ezek a technológiai változások. Kitérünk a napjainkban gyakran alkalmazott lágy számítási módszerekre is. Ezek a módszerek a hagyományos, úgynevezett „kemény” számítási módszerekkel nem megoldható feladatok kezelésére hivatottak azáltal, hogy képesek kezelni a bizonytalanságokat, így a rendszerek adaptív módon valósíthatók meg. A lágy számítási módszerek a jelenleg zajló negyedik ipari forradalom, az IPAR 4.0 részét is képezik, hiszen lehetővé teszi az intelligens eszközök alkalmazását, illetve nagy komplexitású feladatokban, és optimalizálási célokra is előnyösen alkalmazhatók. Az IPAR 4.0 lényegét, fő ágazatait is áttekintjük, valamint megismerkedünk az informatika vonatkozásában gyakran elhangzó konvergencia és szingularitás fogalmával is.

A 2. fejezetben az informatika kulcsfogalmával, az információval foglalkozunk. Áttekintjük, hogy egy informatikai rendszer szempontjából milyen vonatkozásai fontosak, hogyan tekintünk az információra. Megadjuk, hogyan számszerűsíthető az információ, vagyis hogyan határozható meg a szükséges tárhely, vagy csatorna kapacitás az információ tárolásakor, illetve átvitelekor. Kitérünk arra is, hogy a valószínűség milyen hatással van az információ mennyiségére.

A 3. fejezetben a számszerű információkkal foglalkozunk, áttekintjük az informatikában alkalmazott számrendszereket. Megadjuk a számok általános alakját, ami bármely számrendszer esetén érvényes, valamint az egyes számrendszerek közötti átalakítás algoritmusát. Kitérünk arra is, hogy a tízesből kettes számrendszerbe történő átalakítás során milyen pontossági problémák merülhetnek fel valós számok esetén, hiszen ezek ismerete a későbbiekben elengedhetetlen. Megemlítjük azt is, hogy ez a probléma hogyan kezelhető, de részletesebben majd a Programozás I. tárgyon belül foglalkozunk ezzel a kérdéssel.

A 4. fejezetben megnézzük hogyan kezelhetők az előjeles számok. Áttekintjük az előállítás menetét, illetve a műveletvégzés szabályait. Kitérünk arra is, hogy történeti okokból a számítógépek a kivonást összeadásra vezetik vissza, ezért a negatív számokat a pozitívaktól eltérően komplemens alakban tárolják, és kezelik.

Az 5. fejezetben a tört számokkal foglalkozunk. Ezek ismerete mérnöki feladatok esetén különösen fontos, hiszen lényegesen gyakrabban találkozunk velük, mint az egész számokkal. Áttekintjük a lebegőpontos ábrázolás lényegét, megadjuk a számok általános alakját, tárolási módját, és a műveletvégzés szabályait. Megismerjük a karakterisztika eltolt kitevős alakját, illetve a mantissza implicit ábrázolási módját. Kitérünk az adatok bájtjainak tárolási és/vagy

hálózaton való továbbításának sorrendjére is, ami a különböző eszközök, szoftverek közötti hordozhatóságot alapvetően érinti, hiszen a különböző bájtsorrendek alkalmazása esetén teljesen különbözőképpen értelmezhetjük az eredményt, ami komoly problémákat okozhat. A bájtsorrend ismeretében azonban a probléma kezelhető.

A 6. fejezetben először a numerikus kódokat tekintjük át, amelyek a számok bináris reprezentációját jelentik, vagyis tulajdonképpen azok kettes számrendszerbeli alakja. Informatikai rendszerekben azonban rögzítenünk kell a kódhosszt, vagyis azt, hogy a számokat hány biten ábrázoljuk és egységesen az összes szám esetén ezt a méretet kell alkalmaznunk. Kitérünk a hexadecimális kód fontosságára is. A fejezet második része pedig az alfanumerikus kódokat ismerteti, amelyek már a számok mellett karaktereket is tartalmaznak.

A 7. fejezetben azt tekintjük át, hogy az információt hogyan kezelhetjük abban az esetben, ha nem egyedi eseményekkel, hanem rendszerekkel dolgozunk. Bevezetjük a Shannon-entrópia fogalmát, áttekintjük a főbb jellemzőit. Megmutatjuk az entrópia kapcsolatát a korábban definiált Hartley-formulával. Megismerjük az informatika egyik fontos területét, a kereséselmélet fogalmát, ahol igen-nem kérdésekkel próbáljuk megtalálni a keresési tér egy meghatározott elemét. Megmutatjuk, milyen összefüggés van a kereséselmélet és az entrópia között.

A 8. fejezetben az információban rejlő redundancia fogalmával foglalkozunk. A redundancia az üzenetnek az a része, ami elhagyható anélkül, hogy ez az üzenet értelmezhetőségét befolyásolná, ezért a későbbi fejezetekben ismertetésre kerülő tömörítő algoritmusok alapjául szolgál. Definiáljuk a redundancia fogalmát, valamint megadjuk a kiszámítására szolgáló képletet a relatív entrópia fogalmának bevezetése segítségével.

A 9. fejezetben megismerjük az információ kódolás folyamatát, ami informatikai rendszerek esetén nélkülözhetetlen, hiszen a számítógépek minden adatot bináris formában kezelnek, függetlenül attól, hogy ez az adat szám, szöveg, kép, videó, hang, vagy bármi egyéb. Megadjuk a különböző célú kódolási típusok definícióját és felsoroljuk a velük szemben támasztott követelményeket. Ismertetjük a veszteséges és a veszteségmentes adattömörítés lényegét, áttekintjük a különböző algoritmusok hatékonyságát. Megismerjük a kódolás alapvető eszközeit, a kódfát és a prefix kódot, valamint a jellemzésükre szolgáló formulákat és a változó hosszúságú kód jelentőségét.

A 10. fejezetben a veszteségmentes tömörítő algoritmusok egyik nagy csoportjával foglalkozunk, amikor a konkrét üzenet alapján készült statisztika, például a betűgyakoriság szolgál a tömörítés alapjául. Megismerjük a Shannon-

Fano kód, a Gilbert-Moore kód, a Huffman kód, és az aritmetikai kód működését, algoritmusát és példákon keresztül megtanuljuk ezek előállítását. Megadjuk az optimális kód kritériumait és bebizonyítjuk, hogy a Huffman kód optimális kód.

A 11. fejezetben a veszteségmentes adattömörítés másik nagy csoportjával a szótár alapú tömörítéssel foglalkozunk. Ezeknél a módszereknél a kódolás során nem a sorozatok kódját írjuk le, hanem a szótárra hivatkozunk, melyet a kódolás során az aktuális üzenet alapján állítunk elő, miközben az eredeti üzenetet rövidebben írjuk le. Megismerjük a mai tömörítő algoritmusok legtöbbjének alapját képező LZ77 algoritmust, valamint az LZW kódot, ami az ismertetett módszerek közül egyedi abból a szempontból, hogy a szótárt nem kell átküldenünk, mert a dekódolás során önmagát állítja elő.

A 12. fejezetben azzal foglalkozunk, hogy abban az esetben, amikor az átvitel, vagy tárolás során sérül az információ, azt hogyan, milyen módszerekkel vehetjük észre, illetve amennyiben lehetséges, hogyan tudjuk javítani. Definiáljuk a kódellenőrzés, kódjavítás fogalmát. Megemlítjük a redundancia szerepét a hibajavításban és áttekintjük a fontosabb kódellenőrzési módszereket. Bevezetjük a kódszó és a kód Hamming távolságának fogalmát, valamint ismertetjük a különböző paritás ellenőrző módszereket és a CRC működését.

Köszönetnyilvánítás

Elsősorban Tóth Ákosnak tartozom köszönettel, akivel korábban közösen tartottuk az Informatika I. tárgyat a magyar-, illetve az angolnyelvű képzésen. Rengeteg értékes gondolattal, hatalmas szakmai tudással és fáradhatatlan lelkesedéssel fektette le a tárgy alapjait. Éveken át tervezte a jegyzet megírását, amire azonban már nem kerülhetett sor. A jegyzetet az ő emlékének ajánlom.

Továbbá köszönettel tartozom Dr. Kutor Lászlónak, akitől az informatika BSc képzésen a Bevezetés az informatikába (korábban Informatikai Rendszerek Alapjai) tárgyat átvehettem, hiszen ő is számos értékes korábbi előadás anyaggal segítette a munkámat. Szeretnék köszönetet mondani prof. Dr. Pokorádi Lászlónak, aki intézetigazgatóként és kollégaként is mindig mellettem állt és ennek a jegyzetnek a megírását is támogatta. Végül, de nem utolsó sorban köszönöm Dr. habil. Szénási Sándornak, hogy elvállalta a jegyzet lektori feladatait.

1. fejezet

AZ INFORMATIKA FOGALMA, KIALAKULÁSA, FEJLŐDÉSI TENDENCIÁK. INFORMATIKAI ALAPFOGALMAK.

Az informatika napjainkban megkerülhetetlen fogalommá vált. Nem tudunk olyan területet mondani, amihez ne kapcsolódna valamilyen módon. Ebben a fejeztben az informatika fogalmát definiáljuk hagyományos és tudományos értelemben is. Áttekintjük az informatika fejlődési tendenciáit, megnézzük miből fejlődött ki, merre tart és hogyan követhetők ezek a technológiai változások. Kitérünk a napjainkban gyakran alkalmazott lágy számítási módszerekre is, melyek a jelenleg zajló negyedik ipari forradalom, az IPAR 4.0 részét is képezik, hiszen lehetővé teszik az intelligens eszközök alkalmazását, illetve nagy komplexitású feladatokban, és optimalizálási célokra is előnyösen alkalmazhatók. Továbbá megismerkedünk az informatika vonatkozásában gyakran elhangzó konvergencia és szingularitás fogalmával is

1.1. Az informatika fogalma

Az informatika egy viszonylag új tudományterület, ami elsőre furcsának tűnhet, hiszem mára mindent átszőve a mindennapjaink részévé vált. Azonban ha a matematikára vagy a fizikára gondolunk, akkor egyből látjuk, hogy azok megjelenése milyen régmúltra tekint vissza. Hozzájuk képest az informatika valóban egy új tudománynak tekinthető. Mindemellett nagyon heterogén talajból táplálkozik, sokféle tudományterülethez köthető, ezért a mai napig nincs egységes definíció a meghatározására. A következőkben a lehetséges definíciók közül ismerünk meg néhányat, illetve áttekintjük az informatikához kapcsolódó társtudományokat.

Az informatika fogalmának meghatározására szolgáló számos meghatározás közül az alábbiakban láthat néhányat:

- Információ tudomány
- Az információ feldolgozás tudománya
- A számítástechnika alkalmazásainak gyűjtőneve
- Alkalmazási környezetbe ágyazott számítástechnika
- Az információ keletkezésének, leírásának, rendszerezésének tudománya
- Az informatika – mint általános információ tudomány – információrendszerek létrehozását, szerkezetét és működését tanulmányozza
- „Az informatika (mint új tudományterület) a természetes és mesterséges információ feldolgozó rendszerek szerkezetét, viselkedését és interakcióját vizsgálja” University of Edinburgh, School of Informatics

Az utolsó definícióban a természetes és mesterséges információ feldolgozó rendszereket említik. Ehhez kapcsolódóan megjegyezhetjük, hogy informatikában leginkább mesterséges rendszerekkel foglalkozunk, de egyre inkább tolódunk a természetes rendszerek felé. A természetes rendszerekben rejlő óriási erőt egyre inkább kihasználjuk az informatikai rendszerekben is, hiszen a természetes élő rendszerek is alapvetően információ feldolgozó rendszerek. Az információ feldolgozása segítségével képesek a túlélésre. Ezeknek a rendszereknek a működését tanulmányozva, ezek analógiájára építve sokkal hatékonyabb problémamegoldás valósítható meg informatikai rendszerekben is az úgynevezett lágy számítási módszerek alkalmazásával [1].

1.2. Lágy számítási módszerek

A műszaki problémák egy része analitikusan vagy numerikus módszerek, algoritmusok alkalmazásával, vagyis úgynevezett „kemény” számítási módszerek segítségével megoldható. Ezek szabályai azonban szigorúak, nélkülöznek mindenféle bizonytalanságot, azonos bemenetekre mindig azonos kimenetet adnak. Ezzel szemben a „lágy” számítási módszerek kezelni tudják a bizonytalanságokat, képesek a rendszert adaptívvá tenni, többféle eredménnyel szolgálhatnak a körülményekhez alkalmazkodva, és ezek közül választhatjuk a legmegfelelőbbet. Az ilyen típusú rendszerek optimalizálási feladatok, komplex

feladatok esetén biztosítanak hatékony megoldást, intelligens rendszerek létrehozását teszik lehetővé.

„A soft computing nem elvek és technikák homogén egysége, hanem különböző – egymástól akár igen távoli – módszerek együttes használata a vezérelvnek megfelelően. A soft computing legfőbb célja (a cikk írásakor) kiaknázni a pontatlanság és bizonytalanság elviselését a kezelhetőség, robusztusság (határozottság, erőteljesség) és alacsony megoldási költség biztosítása érdekében.” L.A. Zadeh

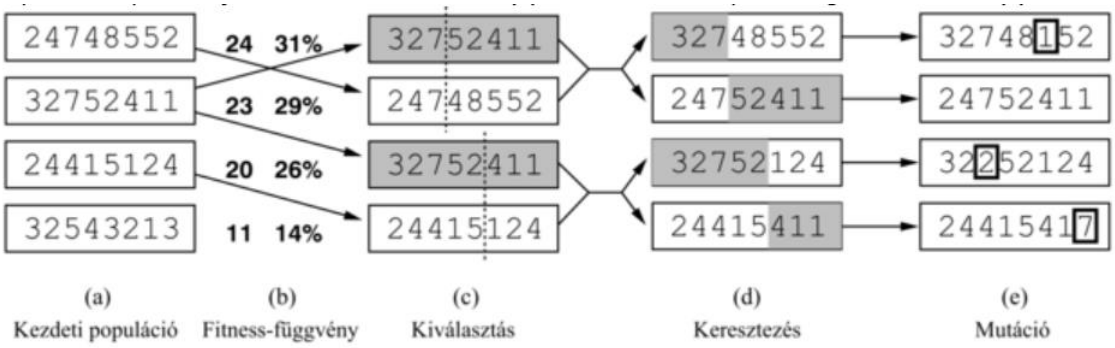
Főbb eszközei a genetikus algoritmusok, neurális hálózatok és a fuzzy logika, melyek lényegét, működési elvét az alábbiakban ismerhetjük meg [2].

Genetikus algoritmusok

A genetikus algoritmusok az evolúció Darwini elmélete alapján jöttek létre az 1970-es évek elején, annak működési elvét utánozzák. Elsősorban keresési, optimalizálási feladatok esetén nyújtanak hatékony megoldást. Olyan kereső rendszerek, amelyek alapja a természetes kiválasztódás, öröklődés mechanizmusa. A feladat lehetséges megoldásait, mint egyedeket (gének) tartalmazó populációval dolgoznak, ahol az egyedekhez, azok „jósága” alapján úgynevezett fitness értéket kapcsolunk. Működésük lényege, hogy különböző operátorok segítségével újabb és újabb populációkat hoznak létre, a lehető legjobb egyedeket kiválasztva, hogy végül megtalálják az optimális megoldást [3], [4]. Az operátorok lehetséges alkalmazási módját mutatja be az 1. ábra.

Az alkalmazott operátorok:

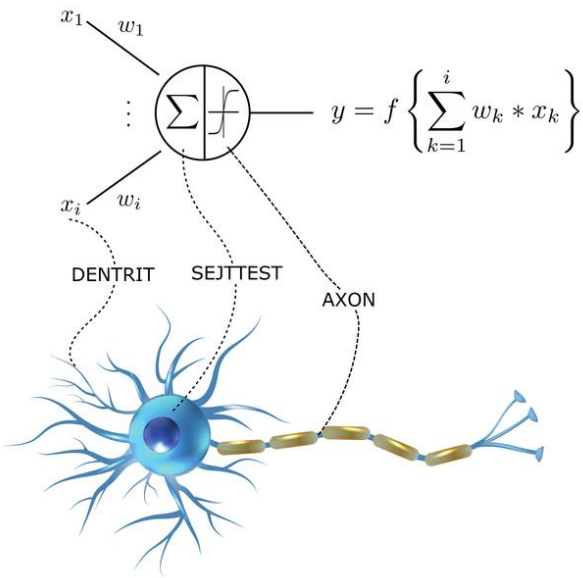
- *reprodukció*: az egyedek közül a fitness érték alapján a legjobbakat választja, ezeket lemásolva hozza létre a következő populációt. A jobb fitness értékkel rendelkező egyedek az új populációban többször is szerepelhetnek, a gyengébb megoldások pedig kimaradhatnak.
- *keresztezés*: több egyedből hoz létre új egyedet, az élőlények szaporodásához hasonlóan mindkét (vagy mindhárom) szülőtől az egyed bizonyos részeit (allélok) örökölve.
- mutáció: az egyed bizonyos bitjei véletlenszerűen megváltoznak.



1. ábra A genetikus algoritmus operátorai

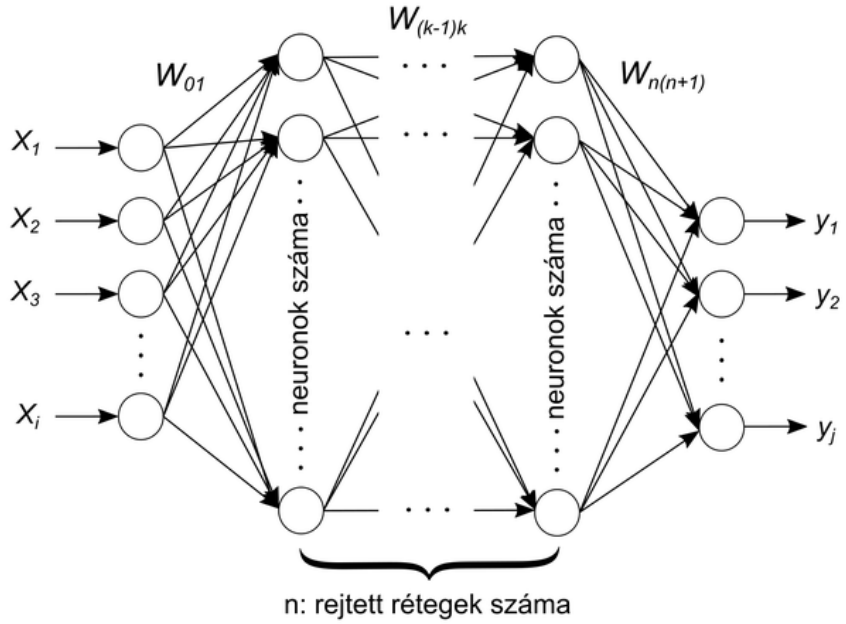
Neurális hálózatok

A mesterséges neurális hálózatok alapjául az emberi agy összekapcsolt neuronjainak bonyolult hálózata szolgált. Alkotóelemei a mesterséges neuronok (egyszerű számítási egységek), melyek egy jól meghatározott struktúra szerint egymással összekapcsolhatók, ezáltal különböző problémaosztályok megoldására alkalmasak. Egyes hálózatok adatminták alapján taníthatók, míg mások automatikusan tanulják, vagy osztályozzák a mintákat. A tanítás után nem csak a tanított minták felismerésére képesek, hanem általánosító képességük alapján a hasonló mintákat is felismerik [3], [5].



2. ábra A természetes és a mesterséges neuron [6]

A természetes és mesterséges neuron analógiája a 2. ábrán látható. A sejttest szerepét veszi át a számítási egység, a dentriteket a mesterséges neuron bemenetei, az axont pedig valamilyen transzfer függvény képviseli. Ezekből a számítási egységekből épül fel a neurális hálózat a 3. ábrán látható módon, a neuronokat rétegekbe szervezve.

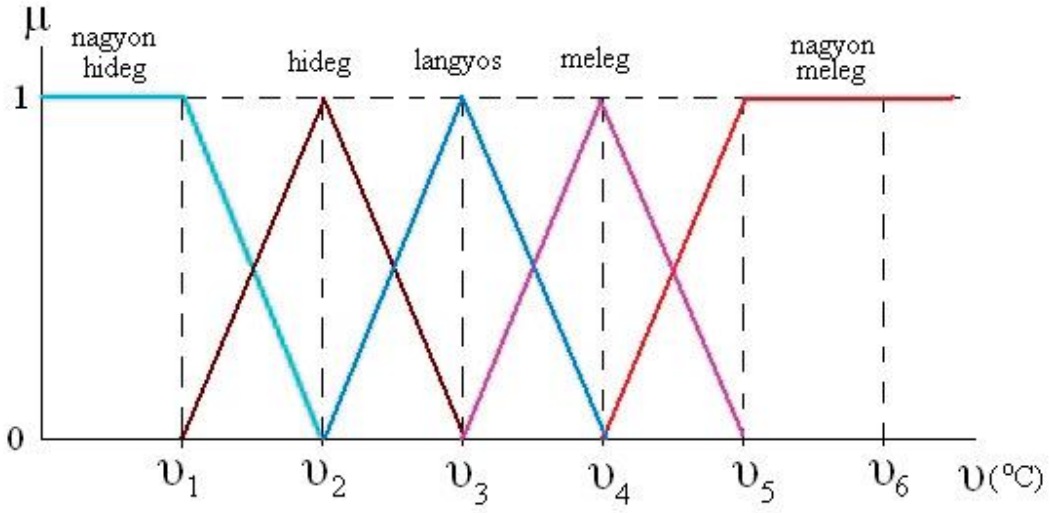


3. ábra Mesterséges neurális hálózat [6]

Fuzzy logika

A fuzzy logika és fuzzy halmazok alapötlete Lotfi A. Zadehtől származik, miszerint az emberi gondolkodás sokkal jobban modellezhető olyan fogalmakkal, amelyeknek nincsenek éles határai, hiszen egy tulajdonság megléte sokszor nem egyértelműen meghatározható. Az ilyen elven működő rendszerek matematikai módszerekkel kezelik a kétértelműségből, pontatlanságból, információhiányból fakadó bizonytalanságot, bevezetve a részleges igazság fogalmát. A fuzzy halmazelmélet a hagyományos halmazelmélet általánosítása alapján jött létre, a halmaz leírása a karakterisztikus függvény általánosításával történik. Nem azt mondjuk meg egy elemről, hogy a halmazhoz tartozik-e (0 vagy 1), hanem azt, hogy milyen mértékben tartozik hozzá (0 és 1 közötti szám). Az ezen az ötleten alapuló szabályozó rendszereket már a 70-es évek közepétől használják ipari alkalmazásokban [7].

A fuzzy logika olyan összetett rendszerekben használható előnyösen, ahol nehéz, vagy lehetetlen a megfelelő rendszermodellt kialakítani; ahol szokásosan emberi szakértő irányít; amelyek folyamatos, vagy közel folyamatos bemenetekkel és nem lineáris kimeneti függvényekkel jellemezhetők; illetve a pontatlanság, bizonytalanság a rendszer gyakori velejárója. A rendszerben számszerű értékek helyett nyelvi jellemzőket használunk, melyre a 4. ábrán láthatunk példát.



4. ábra Nyelvi jellemzők [8]

1.3. Az informatika definíciója tudományos értelemben

A korábbiakban olvashattuk az informatika számos köznapi értelemben vett definícióját, de informatikai rendszerek esetén ezek csak korlátozottan alkalmazhatók. Tudományos értelemben azt mondhatjuk, hogy az informatikai az elektronikus információ feldolgozás tudománya, az információ feldolgozó rendszerek elméletével és gyakorlatával (tervezésével, megvalósításával, ütemezésével) foglalkozik.

Olyan tudomány, ami az információ

- Meghatározásával
- Tárolásával
- Továbbításával
- Tömörítésével
- Titkosításával

- Visszakeresésével

foglalkozik, vagyis minden információhoz kapcsolódó műveletet magában foglal.

Az informatika társtudományai

Ahogy korábban is említettük, az informatika rendkívül heterogén talajból táplálkozik, kialakulását több tudományághoz köthetjük, napjainkban pedig már gyakorlatilag az összes tudományterülettel kapcsolatban áll, összefonódik. A következőkben azokat a társtudományokat ismerhetjük meg, amelyek az informatika alapját képezik [1].

- *Számítástechnika* (Computer Engineering): A számítógép működésével, tervezésével és alkalmazásával foglalkozó tudomány.
- *Számítógép-tudomány* (Computer Science): Az információ feldolgozó gépek tervezésének és használatának elméleti kérdéseit kutatja.
- *Kibernetika* (Cybernetics): Az önműködő rendszerek általános törvényszerűségeivel foglalkozik.
- *Információ elmélet* (Information Theory): Az információ meghatározásával, áramlásával, kódolásával foglalkozó tudomány.
- *Általános rendszerelmélet* (System Theory): A rendszerek működésének körülményeit és tulajdonságait kutatja.
- *Hírközlés* (Communication Theory): A hírek továbbításával foglalkozó tudomány.

A konvergencia fogalma

A fentiek kapcsán fontos megemlíteni a konvergencia fogalmát, ami informatikai vonatkozásban egyre többször hangzik el. Azt mondja ki, hogy a fenti területek összefüggnek, együtt működnek, a közöttük lévő határok átjárhatók, egymás nélkül nem nagyon képzelhetők el. Ez az összemosódás a tudományterületek mellett az eszközök, hálózatok esetén is megfigyelhető. Egyre kevésbé választható külön egymástól például a számítógép és a mobiltelefon fogalma, egyre inkább közösek a különböző eszközök funkciói. A korábbi hálózatok pedig egyetlen komplex hálózatban összevonhatók.

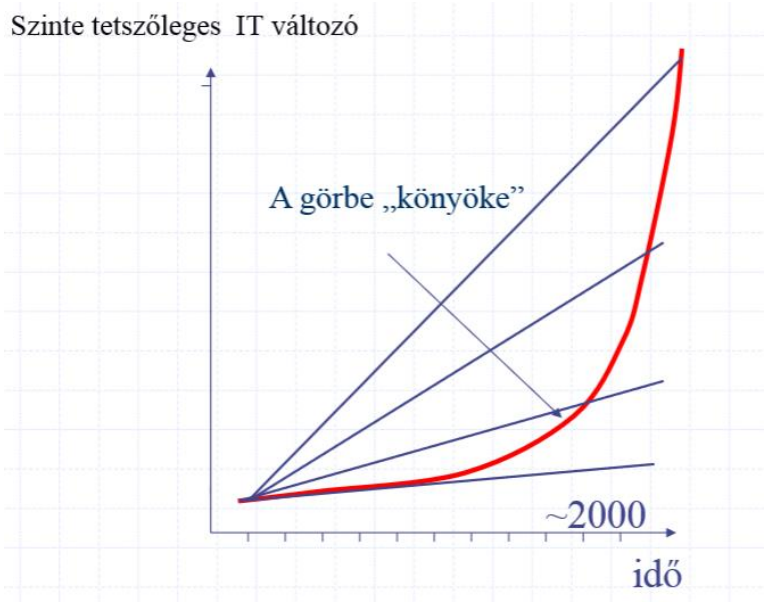
1.4. Az informatika fejlődési tendenciái

Az informatika egy rohamosan fejlődő tudomány. A gyors fejlődés következtében újabb és újabb technológiák születnek, ennek következtében pedig új szakmák alakulnak ki. A következőkben a fejlődési tendenciákba nyerhetünk betekintést szemléltetve azt, hogy informatikai területen a legfontosabb az adaptivitás, hiszen a meglévő technológiák elavulnak, és nem tudjuk mi lép majd a helyükbe. A tendenciákat ismerve azonban rövidtávon valamelyest előre jelezhető a fejlődés iránya, így folyamatosan fejlődve lépést tudunk tartani a változásokkal. A technológiai jellemzők fejlődését szemlélteti az 1. táblázat és az 5. ábra.

Jellemző	Időtartam	Tendencia
Processzorok teljesítménye MIPS-ben	1,8 év	Duplázódik
A processzorok száma	? még nincs elég tapasztalat	Növekszik
Tranzisztorok átlagos ára	1,6 év	Feleződik
Tranzisztorok az Intel mikroprocesszorban	2 év	Duplázódik
Mikroprocesszorok órasebessége	2,7 év	Duplázódik
Számítógépes tárak ára	1,5 év	Feleződik

1. táblázat Informatikai jellemzők felezési vagy kétszerezési ideje [1]

Ahogy az ábrán is látható, az információ-technológia fejlődése exponenciális, ami azt jelenti, hogy a folyamat kezdetben lineárisnak tűnhet, a „könyököt” elérve azonban ugrásszerű növekedés veszi kezdetét. Szakemberek szerint a fordulópont 2000 körül volt, amikor a görbe „könyökét” elérve a fejlődés sokkal gyorsabb üteművé vált [1].

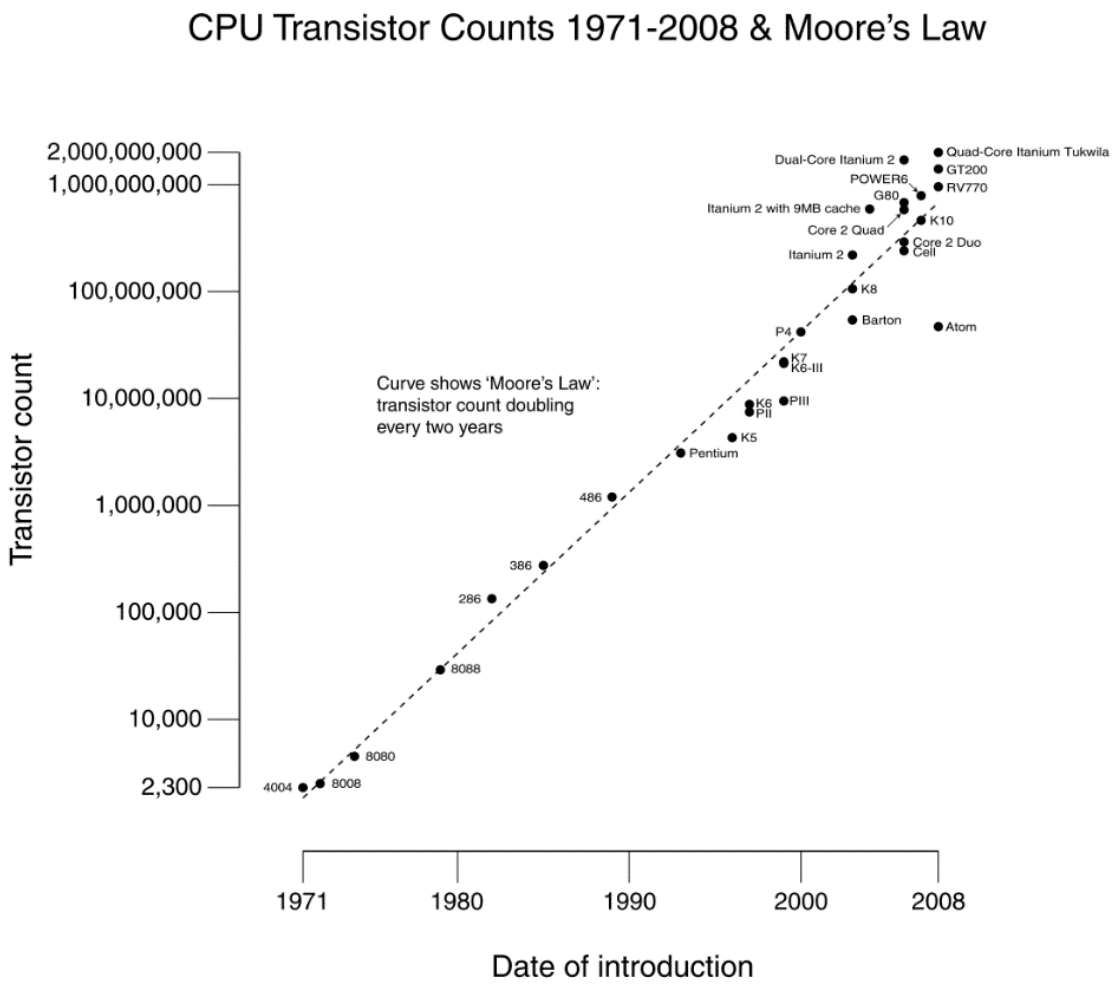


5. ábra Az információ-technológia fejlődését leíró görbe [1]

A következőkben az információ-technológia fejlődését leíró fontosabb tapasztalati törvényeket ismerhetjük meg, amelyek nagymértékben meghatározzák az informatika fejlődésének ütemét és szintén exponenciális jellegűek.

Moore-törvény

A Moore-törvény egy empirikus törvény, ami azt mondja ki, hogy az integrált áramkörök összetettsége, pontosabban a szilíciumchipekben lévő tranzisztorok száma nagyjából évente megduplázódik, ahogy ezt a 6. ábra szemlélteti. A mai napig vitatott, hogy valóban egy tapasztalati törvényről van szó, vagy egy önbeteljesítő jóslatról, aminek a gyártó cégek igyekeznek évről évre eleget tenni. A szakemberek azt jósolják, hogy a jövőben ez a törvény a tranzisztorok száma helyett a magok számáról fog szólni [9].



6. ábra A CPU tranzisztorszámának változása 1971-2008 között [10]

Gilder-törvény

Szintén tapasztalati törvény, amely a kommunikációs rendszerek sávszélességének változásával kapcsolatban tesz megállapítást. Ennek értelmében a kommunikációs rendszerek sávszélessége egy év alatt megháromszorozódik. Ezt a trendet az optikai kábelek kapacitásának a növekedésében ugyanúgy tetten érhetjük, mint a hozzáférési hálózatok technológiai fejlődésében, vagy a gerinchálózati kapacitások terén.

Ruettgers-törvény

Ez a heurisztikus törvény a tárolási kapacitásról szól, amely szerint a felhasznált tárolási kapacitás 12 havonta megkétszereződik.

Metcalf-törvény

A Metcalf-törvény azt mondja ki, hogy a hálózatok értéke négyzetesen arányos a csomópontok számával. Ahogy a hálózat növekszik, a rákapcsolódás értéke exponenciálisan növekszik, míg az egy felhasználóra számított költsége ugyanaz marad, vagy csökken.

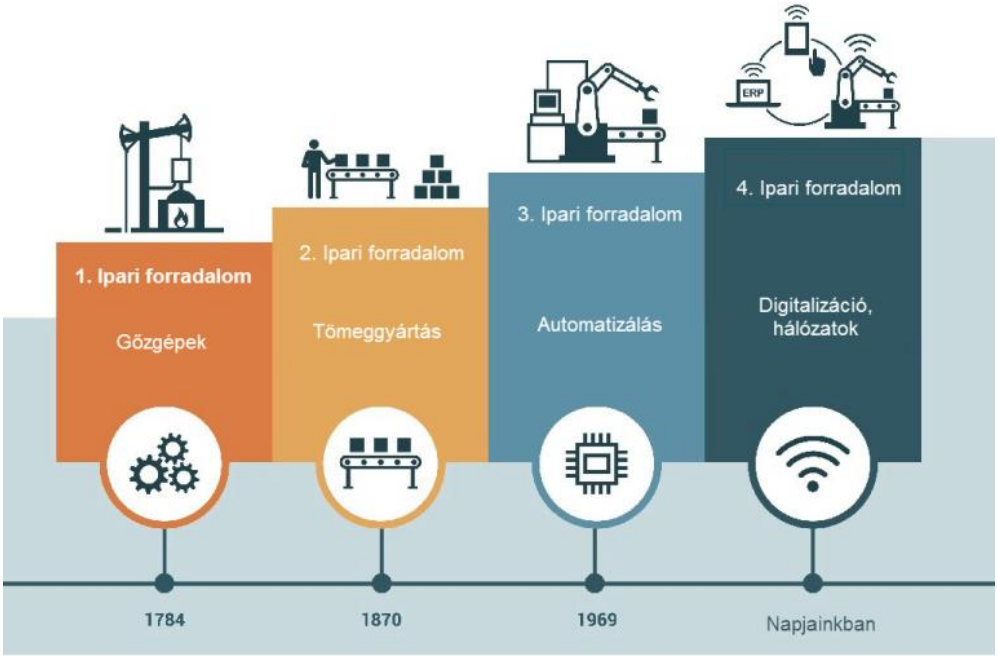
A szingularitás fogalma

Az informatika fejlődésével kapcsolatban megkerülhetetlen fogalom a szingularitás. Szingularitás alatt azt a lehetséges jövőbeli eseményt értjük, amikor az emberfeletti intelligencia megjelenése miatt a technológiai fejlődés és a társadalmi változások olyan módon és sebességgel változtatják meg a környezetet, amit a szingularitás előtt élők képtelenek megbízhatóan jósolni. Jelenlegi jövőmodelljeink nem alkalmasak az előrejelzésre, mivel azok múltbeli tendenciákra épülnek. A jövőkutatók többnyire 2050 körülre jósolják [11].

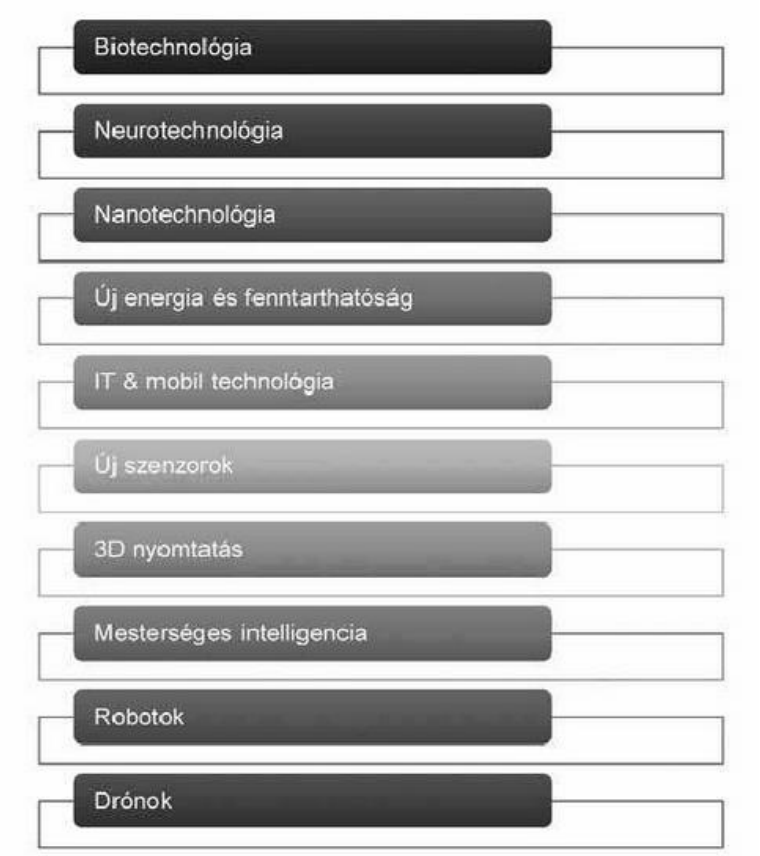
1.5. Ipar 4.0

Az ipari forradalmak mindig jelentős újításokat hoztak az iparba. Az egyes ipari forradalmakat a 7. ábra foglalja össze, és ahogy az ábrán is látjuk, elérkeztünk a 4. ipari forradalomhoz, az ipari digitalizációhoz. Az ipari digitalizálás fő feladata a gyártási folyamat optimalizálása. Olyan technológiai elemek és módszerek alkalmazása válik lehetővé, melyekkel még komplexebb rendszerek működtethetők összehangoltan, teljesen automatizált módon.

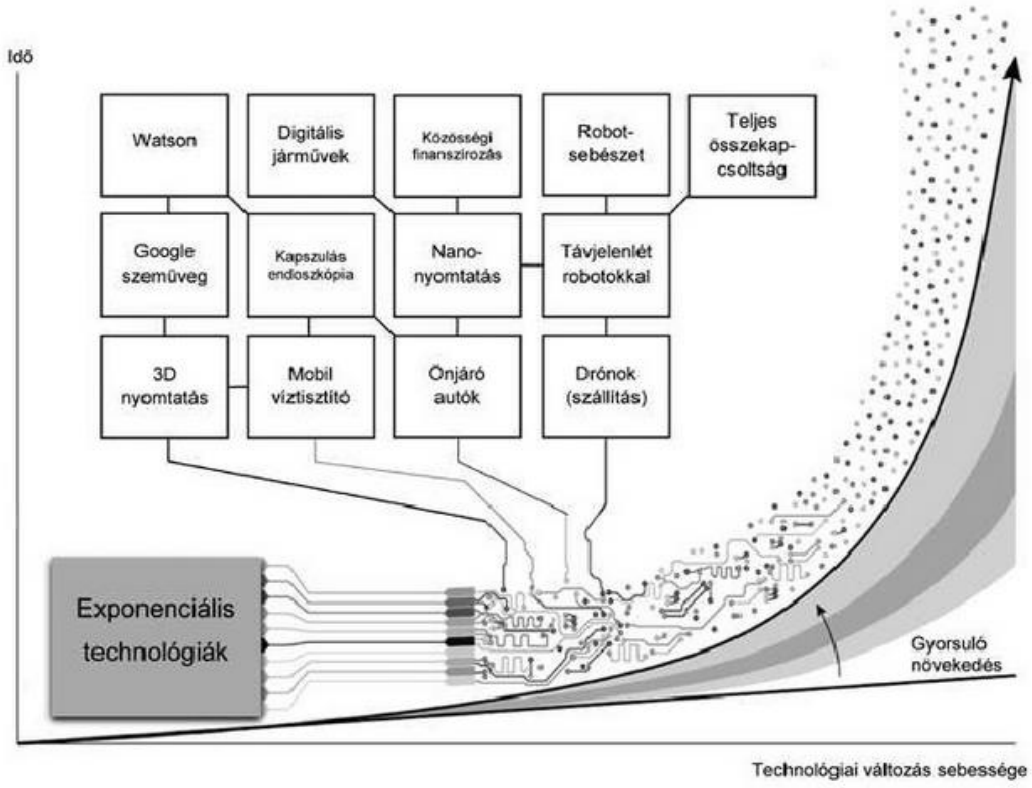
A megoldások alapja az, hogy autonóm, elosztott intelligenciával rendelkező, egymással szoros kapcsolatban álló okos gyárakat alakítanak ki, majd ezek komplex rendszerét átfogó célok alapján optimalizálják. Ez a megoldás a fenntartható fejlődés problémáinak kezelésében is segítséget nyújthat. Az IPAR 4.0-t meghatározó fejlődési irányokat a 8, 9. ábra szemlélteti.



7. ábra Az ipari forradalmak [12]



8. ábra Az IPAR 4.0 fő ágazatai



9. ábra Az IPAR 4.0 fejlődése

2. fejezet

AZ INFORMÁCIÓ FOGALMA, MENNYISÉGE.

Az előző fejezetben megismerkedhettünk az informatika fogalmával és bár láttuk, hogy többféle definíció is létezik, abban mindegyik megegyezik, hogy központi fogalomnak az információt tekinti, ezért ebben a fejezetben információ fogalmával ismerkedhetünk meg. áttekintjük, hogy egy informatikai rendszer szempontjából milyen vonatkozásai fontosak. Megadjuk azokat a definíciókat, amelyek segítségével számszerűsíthető ennek mennyisége, ami ahhoz szükséges, hogy meg tudjuk határozni a szükséges tárhely méretét, vagy a csatorna kapacitását az információ tárolásakor, illetve átvitelekor. Kitérünk arra is, hogy a valószínűség milyen hatással van az információ mennyiségére.

2.1. Az információ fogalma

Az informatikát tudományos értelemben, információ elméleti szempontból definiáltuk, vagyis azt mondtuk, hogy az informatika az információ

- meghatározásával
- tárolásával
- továbbításával
- tömörítésével
- titkosításával
- visszakeresésével

foglalkozó tudomány. A következőkben az információ fogalmát definiáljuk először hagyományos, majd tudományos értelemben.

Hagyományos értelmezés

Amikor hagyományos értelemben beszélünk az információról, annak jelentése lehet tájékoztatás, felvilágosítás, bejelentés, hír, újság, közlés, jellemzés, adat, tudás, vagy értesülés, vagyis minden, amit a rendelkezésre álló adatokból nyerünk. Fontos azonban azt is megjegyezni, hogy az adat újszerűsége is szerepet játszik abban, hogy információnak tekintjük, vagy sem. Abban az esetben, ha egy olyan tényről beszélünk, amelynek megismerésekor olyan tudásra teszünk szert, ami addig nem volt a birtokunkban, akkor ezt információnak tekinthetjük. Azonban ha egy számunkra már ismert adatot közölnek velünk, az már nem információ többé. Például, ha valaki látta már a zárthelyi eredményeket, amiket mi még nem és elmondja nekünk, akkor az információnak tekinthető. Ha már mi is birtokában voltunk ennek az adatnak, akkor ez már nem információ a számunkra.

Tudományos értelmezés

Tudományos értelemben kicsit máshogy tekintünk az információra, bár ahogy látni fogjuk, annak újszerűségét valamilyen szempontból itt is figyelembe vesszük majd, amikor azt vizsgáljuk, hogy mennyire meglepő számunkra ez az információ, vagyis egy esemény bekövetkezéséhez milyen valószínűséget rendelünk. Azt mondhatjuk, hogy az információ olyan „mennyiség”, amely egy eseményrendszer egyik vagy másik eseményének bekövetkezéséről (illetve egy állapottér egyik vagy másik állapotáról) elemi szimbólumok sorozatával közölhető, tehát tudományos értelemben mennyiségként tekintünk rá és hozzá kapcsolódóan az információ-elmélet kérdéseivel foglalkozunk [1], [13].

Az információ-elmélet az információ

- megjelenítésével
- tárolásával
- cseréjével

foglalkozó tudomány. Nagyon fontos megjegyezni, hogy az információ-elmélet nem foglalkozik az információ tartalmával, csak a mennyiségével, vagyis az informatikában az információt fizikai mennyiségként értelmezzük. Ennek az az oka, hogy amikor az információt tárolni, vagy továbbítani szeretnénk, akkor az informatikai rendszer szempontjából mellékes, hogy milyen adatról van szó. Az egyetlen lényeges dolog, hogy elég tárhely, illetve csatorna kapacitás áll-e rendelkezésünkre az információ tárolásához, továbbításához.

Az információ definíciója információ-elméleti szempontból: egy ismert véges halmaz egy elemének megnevezése.

Például abban az esetben, ha feldobunk egy pénzérmét és azt szeretnénk tudni, hogy fejet, vagy írást dobtunk, akkor a lehetséges kimenetek fogják képezni az ismert véges halmazt, vagyis A = {Fej, Írás} és a dobás kimenetele lesz az az elem, amit ebből a halmazból kiválasztunk.

2.2. Az információ mennyisége

A Hartley-formula

Az információ mennyiségének számszerűsítése a szükséges tárkapacitás, illetve csatornakapacitás szempontjából fontos, ezért úgy kell definiálni, hogy az eredmény bitben álljon rendelkezésünkre. Ezt a logaritmus fogalmát alkalmazó Hartley-formula segítségével határozhatjuk meg. A formula eredménye az információ mennyiségét adja meg, vagyis az információ leírásához szükséges bitek számát. Mivel a bitek száma csak egész szám lehet, ezért abban az esetben, ha a képlet eredményéül nem egész számot kapunk, akkor azt a szabályt alkalmazzuk, hogy a kapott értéket felfelé kerekítjük, hiszen a lefelé kerekítéskor biztosan nem lesz elég a rendelkezésre álló bitek száma.

Hartley-formula:

$$I = log_2 n \tag{1}$$

ahol *n* az ismert véges halmaz elemszáma

A formula bármilyen számrendszer esetén működőképes, a logaritmus alapja a választott számrendszertől függ. Az informatikában használt bináris ábrázolás miatt mi csak azzal az esettel foglalkozunk, amikor a logaritmus alapja 2.

A logaritmus kiszámítására vonatkozó általános képlet:

$$log_a b = c \rightarrow a^c = b \tag{2}$$

Például: $log_2 8 = 3$, $log_2 \frac{1}{2} = -1$, $log_2 1 = 0$

Ha a számológépünkkel csak tízes alapú logaritmust tudunk számolni, a következő általános képlet alkalmazásával bármilyen más alapú logaritmus is kiszámítható.

$$log_y x = \frac{log_{10} x}{log_{10} y} \quad (3)$$

ahol *x* az a szám, aminek a logaritmusát szeretnénk meghatározni, *y* pedig a kívánt számrendszer alapszáma, vagyis abban az esetben, ha kettesalapú logaritmust szeretnénk számolni, *y=2*.

Például:

$$log_2 16 = \frac{log_{10} 16}{log_{10} 2} = \frac{1{,}204}{0{,}301} = 4$$

Nézzük meg a Hartley-formula alkalmazását példákon keresztül is:

1. Feladat: feldobunk egy szabályos pénzérmét, hány biten tudjuk megadni, hogy mi a kimenet?

   Megoldás:

   Az ismert véges halmaz: *A = {Fej, Írás}*

   A halmaz elemszáma: *n = 2*

   A Hartley-formulát alkalmazva: $I = log_2 2 = 1$ *bit*

   vagyis a kimenetként kapott elemet 1 biten tudjuk megnevezni. Ez azt jelenti, hogy a halmaz elemeihez hozzárendeljük a 0-t és az 1-t, pl.: *Fej – 0*, *Írás - 1*, így valóban 1 biten írható le az információ, vagyis az, hogy fejet, vagy írást dobtunk, mert a kimenet 0 vagy 1 lesz.

2. Feladat: feldobunk egy szabályos dobókockát, hány biten tudjuk megadni, hogy mi a kimenet?

   Megoldás:

   Az ismert véges halmaz: *B = {1, 2, 3, 4, 5, 6}*

   A halmaz elemszáma: *n = 6*

   A Hartley-formulát alkalmazva: $I = log_2 6 = 2{,}58bit \approx 3bit$

   vagyis a kimenetként kapott elemet 3 biten tudjuk megnevezni. Ez azt jelenti, hogy a halmaz minden egyes eleméhez egy hárombites kódot rendelünk hozzá, pl.: *1 – 000*, 2 - *001*, *3 – 010*, 4 – *011, 5 – 100*, 6 – *101,* ahol rendre az első szám a dobott érték, a második a hozzárendelt kód. Vegyük észre, hogy nem használtuk ki az összes 3 biten felírható kombinációt (*? – 110*, ? – *111*), hiszen 8-féle számsort

tudunk leírni, nekünk pedig csak hatra van szükségünk. A fennmaradó kombinációk képviselik az úgynevezett redundanciát a rendszerben, melynek fogalmáról, jelentőségéről a későbbiekben fogunk részletesen beszélni. Abban az esetben, amikor a Hartley-formula eredménye nem egész szám, mindig számolhatunk redundaciával, vagyis feleslegesen előállított bitkombinációkkal.

A formula mértékegysége különböző alapú logaritmusok esetén.

Ahogy azt korábban is említettük, a Hartley-formula nem csak kettesalapú logaritmus esetén érvényes, hanem általánosan bármilyen számrendszer esetén alkalmazható, a logaritmus alapja a választott számrendszertől függ. Ennek megfelelően a kapott eredmény mértékegysége is más lesz.

$$log_{10}n \rightarrow \text{[Hartley] (digit)}$$

$$log_{2}n \rightarrow \text{[bit]}$$

$$log_{e}n \rightarrow \text{[Nat]}$$

Miért a logaritmust használjuk?

A Hartley-formulával kapcsolatban már említettük, hogy a logaritmus alapja a választott számrendszertől függ, tehát bármilyen számrendszer esetén alkalmazható. Ennek ez az oka, hogy valójában a logaritmus tulajdonságai miatt működik a formula. Nézzük meg, hogy Hartley hogyan indokolta a logaritmus használatát [14]:

1. A műszaki szempontból lényeges paraméterek (pl. idő, sávszélesség, jelfogók száma) a lehetőségek számával lineárisan változnak.

   Például: ha 1 jelfogó egységnyi információt tud kezelni, akkor 3 jelfogónak háromszor annyit kell tudnia kezelni

   1 jelfogó: $2^1 = 2$ állapot $\rightarrow log_2 2 = 1$

   3 jelfogó: $2^3 = 8$ állapot $\rightarrow log_2 8 = 3$

2. Közel áll az intuitív érzésünkhöz. (Szorosan kapcsolódik az 1. ponthoz)

   Az előző példában azt vártuk, hogy 3 jelfogó háromszor annyi állapotot tudjon előállítani, vagy 3 azonos kapacitású tárolóeszköz háromszor annyi információt tudjon tárolni, mint ha csak egy lenne belőle.

Additivitás

Az információt fizikai mennyiségként jellemeztük, ami azt jelenti, hogy teljesülniük kell a mérésre vonatkozó alapszabályoknak, mint például az additivitás. Ez a követelmény szintén a logaritmus használatát indokolja, hiszen ebben az esetben, ha két különböző szempont szerint választunk, akkor a választásban rejlő információ nagyságát leíró függvények összege meg kell, hogy egyezzen azzal, amikor a két szempontot egyszerre vesszük figyelembe és így választunk.

Például, ha egy táblázatból kell választanunk szín és forma szerint

Színek száma: *m*

Színek halmaza = $\{S_1, S_2, \ldots, S_m\}$

Szín szerinti választást leíró függvény: $I(S_i) = f(m)$

Formák száma: *n*

Formák halmaza = $\{X_1, X_2, \ldots, X_n\}$

Forma szerinti választást leíró függvény: $I(X_i) = f(n)$

Szín és forma egyszerre történő választását leíró függvény: $I(S_i X_j) = f(m * n)$

Az additivitási szabály alapján teljesülnie kell a következő összefüggésnek:

$$f(m) + f(n) = f(m * n) \tag{4}$$

A logaritmus azonosságait figyelembe véve kijelenthető, hogy a logaritmus ennek a feltételnek is eleget tesz, hiszen

$$log_a(x) + log_a(y) = log_a(x * y) \tag{5}$$

2.3. A valószínűség hatása az információ mennyiségére

A módosított Hartley-formula

Az előzőekben tárgyalt Hartley-formula nem foglalkozik azzal, hogy mennyire meglepő a kimenet, vagyis nem tulajdonít jelentőséget az esemény bekövetkezési valószínűségének. Abban az esetben, ha az egyes események bekövetkezési valószínűsége különböző, a formula módosítása szükséges, hiszen a valószínűség az információ mennyiségére is hatással van. Tekintsük a

pénzérmés példát. Ha az érme nem szabályos, akkor nyilvánvalóan különböző esélyekkel dobhatunk fejet, vagy írást.

A valószínűség fogalmához kapcsolódó legfontosabb jelölések.

A módosított formula bevezetéséhez, illetve a későbbiekben szereplő kódolási módszerek alapelvének megértéséhez ismerni kell a valószínűség fogalmát, ezért a következőkben a hozzá kapcsolódó alapvető fogalmakat tekintjük át.

Egy „A" esemény bekövetkezési valószínűsége: *P(A)*

Biztos esemény bekövetkezési valószínűsége: *P(A) = 1*

(Biztos eseményről akkor beszélünk, ha ez az esemény garantáltan be fog következni.)

Lehetetlen esemény bekövetkezési valószínűsége: *P(A)=0*

(Lehetetlen eseményről akkor beszélünk, ha tudjuk, hogy ez az esemény garantáltan nem fog bekövetkezni.)

Összefüggő események esetén a bekövetkezési valószínűségek összege:

$$\sum_{i=1}^{n} P(A_i) = 1 \qquad (6)$$

Az összefüggő események közül egyszerre csak egy következhet be.

Az információ hétköznapi értelmezését figyelembe véve azt mondhatjuk, hogy minél kisebb az esemény bekövetkezési valószínűsége, annál nagyobb a hírértéke, vagyis ha olyan kimenetet kapunk, aminek kisebb a valószínűsége, az számunkra meglepőbb. Ez a tulajdonság a tudományos értelemben vett információ kapcsán is megjelenik, vagyis a valószínűség az információ mennyiségére is hatással van. Ennek leírására szolgál a módosított Hartley-formula.

Módosított Hartley-formula:

$$I = log_2 \frac{1}{P(A)} \qquad (7)$$

ahol *P(A)* az „*A*" esemény bekövetkezési valószínűsége.

Az eredeti és a módosított Hartley-formula összehasonlítása

A módosított formulának akkor van jelentősége, ha az egyes események bekövetkezési valószínűsége különböző, hiszen ha minden kimenet ugyanolyan

eséllyel következik be, az információ mennyiségére nincsenek hatással az egyes valószínűségek. A következőkben ezt fogjuk bizonyítani.

Tudjuk, hogy összefüggő események esetén az egyes valószínűségek összege 1, ezért *n* egyenlő valószínűségű eseményre az egyes valószínűségek a $P(A) = \frac{1}{n}$ kifejezéssel határozhatók meg. Ezt behelyettesítve a módosított formulába:

$$I = log_2 \frac{1}{P(A)} = log_2 \frac{1}{\frac{1}{n}} = log_2 n$$

Vagyis visszakaptuk az alap Hartley-formulát, ezért kijelenthetjük, hogy egyenlő valószínűségek esetén az alap formula is használható, a valószínűségnek nincs hatása az információ mennyiségére, különböző valószínűségek esetén azonban a módosított formula alkalmazása szükséges.

Nézzük meg a módosított Hartley-formula alkalmazását példákon keresztül is:

Szabálytalan dobókocka esetén, ahol az egyes számok bekövetkezési valószínűsége rendre: $\frac{1}{2}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}$ a módosított Hartley-formulát alkalmazva:

$$P(A) = \frac{1}{2} \rightarrow I = log_2 \frac{1}{\frac{1}{2}} = log_2 2 = 1\ bit$$

$$P(A) = \frac{1}{10} \rightarrow I = log_2 \frac{1}{\frac{1}{10}} = log_2 10 = \ 3{,}32\ bit\ \rightarrow 4\ bit$$

A fenti példákból is látható, hogy kisebb valószínűség esetén nagyobb az információ mennyisége, vagyis a hírérték, míg nagyobb valószínűség esetén kisebb.

Miért olyan fontos az információ mennyisége?

Az informatika világában minden binárisan, vagyis 0,1 sorozatként tárolódik, továbbítódik függetlenül attól, hogy az információ valamilyen, kép, szöveg, szám, videó, vagy bármi egyéb. Az informatikai rendszerek szempontjából a legfontosabb kérdés, hogy tárolás esetén mekkora tárhelyre, illetve adatátvitek esetén milyen csatorna kapacitásra van szükség az adott információ kezeléséhez. Tudnunk kell azt, hogy mennyire kell esetlegesen tömöríteni az információt.

3. fejezet

SZÁMRENDSZEREK. SZÁMRENDSZEREK KÖZÖTTI ÁTALAKÍTÁS.

Elsőként a számszerű információkkal foglalkozunk, áttekintjük az informatikában alapvető számrendszereket. Megadjuk a számok általános alakját, ami bármely számrendszer esetén érvényes, valamint az egyes számrendszerek közötti átalakítás algoritmusát. Kitérünk arra is, hogy a tízesből kettes számrendszerbe történő átalakítás során milyen pontossági problémák merülhetnek fel valós számok esetén, hiszen ezek ismerete a későbbiekben elengedhetetlen. Megemlítjük azt is, hogy ez a probléma hogyan kezelhető, de részletesebben majd a Programozás I. tárgyon belül foglalkozunk ezzel a kérdéssel.

3.1. Számrendszerek. Számábrázolás.

A legfontosabb számrendszerek

Először tekintsük át azt, hogy a mi szempontunkból mely számrendszerek fontosak.

- **Decimális**
  számjegyei (digit): {0, 1, …, 9}
  Ez a számrendszer megkerülhetetlen, bár az informatikában nem használatos, de mi a mindennapi életben ezt használjuk, tízes számrendszerben gondolkozunk, ezért általában ez a kiinduló számrendszerünk, a számítógép ezt kapja meg bemenetként.

- **Bináris**
  számjegyei (bit): {0, 1}
  Informatikában alapvető, hiszen a számítógép mindent binárisan, 0,1 sorozatként kezel, függetlenül az adattípustól. Az adat lehet akár kép, videó, szöveg, szám, vagy bármi más, a bináris formában történő kezelés mindegyikre érvényes. Ez azt jelenti, hogy a kapott adatot minden esetben bináris kóddá alakítjuk és bármilyen műveletet végzünk vele, az ebben a formában történik.

- **Oktális**
  számjegyei: {0, 1, …, 7}
  Régebben ezt a számrendszert is használták az informatikában, de nem terjedt el.

- **Hexadecimális**
  számjegyei: {0, 1, …, 9, A, …, F}, ahol A értéke 10, B értéke 11, …, F értéke 15.
  Informatikában a kettes számrendszer mellett ez a másik legfontosabb, mert számunkra könnyebben kezelhetővé teszi a kettes számrendszerbeli számokat. Fontos azonban, hogy ez csak egy formális számrendszer, használatával a kettes számrendszerbeli számok rövidebben írhatók le, de ez csak számunkra segítség, a számítógép ezt a formátumot közvetlenül nem használja.

A számok általános alakja

Egy szám általános alakban (bármely számrendszerre érvényes módon) a következőképpen írható fel:

$$N = \pm \sum_{k=-m}^{j} a_k \cdot r^k \qquad (9)$$

ahol *r* a számrendszer alapszáma (radix), *m* a negatív helyiértékek száma, *j* a pozitív helyiértékek száma, a_k az együtthatókat jelenti, *k* a kitevő.

Nézzük meg ezt egy példán keresztül is.

Decimális számrendszerben (r = 10):

$$231_{10} = 2 \cdot 10^2 + 3 \cdot 10^1 + 1 \cdot 10^0$$

$$231{,}4_{10} = 2 \cdot 10^2 + 3 \cdot 10^1 + 1 \cdot 10^0 + 4 \cdot 10^{-1}$$

Bináris számrendszerben (r = 2):

$$10111_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$10111.01_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

3.2. Átalakítás a számrendszerek között

Az átalakítás menetének megismerése elsősorban azért fontos, mert ahogy korábban említettük, mi a számokat tízes számrendszerben adjuk meg, de a számítógép kettes számrendszerbeli számmá alakítva kezeli azokat, majd az eredményt alakítja vissza tízes számrendszerbeli számmá, hogy számunkra is könnyen értelmezhető legyen. Az átalakítás különbözőképpen történik a kisebből a nagyobb, vagy a nagyobból a kisebb számrendszerbe alakítás esetén. A következőkben az átalakítás menetének általános leírását fogjuk megnézni. Fontos, hogy bármilyen irányú átalakítás történik, az egészrészt és a törtrészt mindig külön kell kezelnünk, majd ezeket egymás mellé írva kapjuk a szám végleges alakját.

Átalakítás kisebből nagyobb számrendszerbe

Először az *egészrész* átalakítását nézzük meg, ahol a különböző helyiértékeken szereplő számokat szorozzuk a kiinduló számrendszer alapszámának megfelelő hatványával:

$$N_e = a_0 + a_1 r + a_2 r^2 + \cdots + a_{n-1} r^{n-1} + a_n r^n \qquad (10)$$

ahol *r* a számrendszer alapszáma (radix), a számjegyek száma *n+1.*

A fenti általános formula rövidebben, kevesebb művelet elvégzésével is felírható a Horner módszer alkalmazásával, melynek lényege, hogy *r* –t kiemeljük a következő módon, így kiküszöbölve a hatványozás műveletét [15]:

$$N_e = a_0 + r\big(a_1 + r(a_2 + \cdots + r a_n)\big) \qquad (11)$$

A törtrész előállításakor szintén a különböző helyiértékeken szereplő számokat szorozzuk a kiinduló számrendszer alapszámának megfelelő hatványával, de itt a pozitív kitevők helyett negatív kitevőket használunk.

$$N_t = a_{-1} r^{-1} + a_{-2} r^{-2} + \ldots + a_{-m} r^{-m} \qquad (12)$$

ahol *r* a számrendszer alapszáma (radix), a számjegyek száma *m.*

A *törtrészre* vonatkozó formula szintén felírható rövidebben a Horner módszert alkalmazva, kevesebb művelet elvégzésével r^{-1} kiemelésével a következő módon:

$$N_t = r^{-1}(a_{-1} + r^{-1}(a_{-2} + \cdots + r^{-1}(a_{-m+1} + r^{-1}a_{-m}) \ldots)) \quad (13)$$

Az egészrész és a törtrész megfelelő átalakítása után a teljes számot a következő alakban írhatjuk fel:

$$N = N_e + N_t \quad (14)$$

Nézzük meg az átalakítást egy példán keresztül, ahol egy bináris számot alakítunk át decimális számmá, vagyis a kiinduló számrendszer alapszáma *r = 2*.

Az átalakítandó szám:

N_2 = 11010111.01101

Az átalakítást az egészrésszel kezdjük a legkisebb helyiértékű (2^0) bittől indulva:

$$1 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 = 215$$

ugyanezt az átalakítást felírva a rövidebb, Horner módszer szerinti (11) alakban:

$$1 + 2\left(1 + 2\left(1 + 2\left(0 + 2\left(1 + 2\big(0 + 2(1 + 2 \cdot 1)\big)\right)\right)\right)\right) = 215$$

A törtrész átalakítása (12) alapján történik, a legnagyobb helyiértékű (2^{-1}) bittől indulva:

$$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0{,}40625$$

ugyanezt az átalakítást felírva a rövidebb, Horner módszer szerint (13) alakban:

$$2^{-1}\left(0 + 2^{-1}\left(1 + 2^{-1}\big(1 + 2^{-1}(0 + 2^{-1} \cdot 1)\big)\right)\right) = 0{,}40625$$

A tízes számrendszerbeli érték az egészrész és a törtrész egymás mellé írásából adódik:

N_{10} = 215,40625

Átalakítás nagyobból kisebb számrendszerbe

A kisebből nagyobb számrendszerbe történő átalakításhoz hasonlóan itt is külön kell kezelni a szám egész- és törtrészét. Az egészrész számításakor a számot osztjuk a célszámrendszer alapszámával, majd mindig a hányados egészrészt osztjuk tovább, közben felírva az osztási maradékokat. Ezt a műveletet addig ismételjük, amíg a hányados egészrészére nullát nem kapunk eredményül. A célszámrendszer számát a maradékok visszafelé történő összeolvasásával kapjuk.

Az *egészrész* átváltásának általános algoritmusa:

$$\frac{N}{r} = b_1 + \frac{a_0}{r}$$

$$\frac{b_1}{r} = b_2 + \frac{a_1}{r}$$

$$M \quad \text{amíg } b_j <> 0$$

ahol *N* a nagyobb számrendszerbeli szám, *r* a célszámrendszer alapszáma, b_j a hányados egészrésze, a_{j-1} az osztás maradéka.

A törtrész átalakításakor a számot a célszámrendszer alapszámával szorozzuk, majd az eredményt leírjuk és annak csak a törtrészét szorozzuk tovább. Ezt a műveletet addig ismételjük, amíg a törtrészre nullát nem kapunk eredményül, vagy el nem érjük a kívánt pontosságot. A célszámrendszer számát az egészrészek összeolvasásával kapjuk.

A *törtrész* átváltásának általános algoritmusa:

$$N \cdot r = a_{-1} + b_{-1}$$

$$b_{-1} \cdot r = a_{-2} + b_{-2}$$

$$M \text{ amíg } b_j <> 0$$

ahol *N* a nagyobb számrendszerbeli szám, *r* a cél számrendszer alapszáma, b_i az eredmény törtrésze, a_i pedig az eredmény egészrésze.

Nézzük meg az átalakítást egy példán keresztül, ahol egy decimális számot alakítunk át bináris számmá, vagyis *r = 2* a célszámrendszer alapszáma.

Az átalakítandó szám:

$$N_{10} = 114{,}25$$

Az átváltást a szám egészrészével kezdjük:

	b_j	a_j	
b_0	114	0	a_0
b_1	57	1	a_1
b_2	28	0	a_2
b_3	14	0	a_3
b_4	7	1	a_4
b_5	3	1	a_5
b_6	1	1	a_6
b_7	0		a_7

10. ábra Az átalakítás menete

ahol b_j a kettővel osztás eredményének egészrésze, a_j a kettővel osztás eredményének maradéka.

A számot a jobboldalon szereplő maradékok alulról felfelé történő összeolvasásával kapjuk.

Az alábbiakban azt szemléltetjük, hogy a képletet hogyan alkalmaztuk az átalakítás során:

$$\frac{114}{2} = 57 + \frac{0}{2} \qquad \frac{N}{r} = b_1 + \frac{a_0}{r}$$

$$\frac{57}{2} = 28 + \frac{1}{2} \qquad \frac{b_1}{r} = b_2 + \frac{a_1}{r}$$

$$\vdots \qquad \vdots$$

$$\frac{1}{2} = 0 + \frac{1}{2} \qquad M \quad \text{amíg } b_j <> 0$$

A szám egészrésze bináris alakban: 1110010

Az átváltást a szám törtrészével folytatjuk:

	a_j b_j	
	0,25	
a_{-1}	0,5	b_{-1}
a_{-2}	1,0	b_{-2}

11. ábra Az átalakítás menete

ahol b_j a kettővel szorzás eredményének törtrésze, a_j a kettővel szorzás eredményének egészrésze.

A számot a baloldalon szereplő egészrészek felülről lefelé történő összeolvasásával kapjuk. Ezért is fontos, hogy mindig csak a kapott szorzat törtrészét szorozzuk tovább, mert így garantálható, hogy csak 0, vagy 1 lehet az egészrész.

Az alábbiakban azt szemléltetjük, hogy a képletet hogyan alkalmaztuk az átalakítás során:

$0{,}25 \cdot 2 = 0 + 0{,}5$ $\qquad$ $N \cdot r = a_{-1} + b_{-1}$

$0{,}5 \cdot 2 = 1 + 0$ $\qquad$ $b_{-1} \cdot r = a_{-2} + b_{-2}$

$\vdots$ $\qquad$ $\vdots$

(b_j=0) $\qquad$ M amíg $b_j <> 0$

A szám törtrésze bináris alakban: 0.01

A kettes számrendszerbeli érték az egészrész és a törtrész egymás mellé írásából adódik

N_2=1110010.01

Fontos megjegyezni, hogy a fenti szám a matematikai átalakítás szabályai szerint megfelelő, informatikai rendszerek esetén azonban a számokat rögzített méreten ábrázoljuk, ahogy azt a numerikus kódokkal foglalkozó fejezetben részletesen is látni fogjuk.

Pontossági problémák az átalakítás során

A törtrész átalakításakor nagyobból kisebb számrendszerbe alakításkor előfordulhat olyan eset, amikor a kettővel való sorozatos szorzás eredményeként soha nem kapunk olyan eredményt, ahol a törtrész nulla. Ilyenkor pontos átalakítás nem végezhető, az érték pontossága attól függ, hogy hány bitet szánunk a szám ábrázolására.

Ennek a problémának az ismerete azért nagyon fontos, mert tudjuk, hogy a számítógép minden műveletet kettes számrendszerben végez, ami azt jelenti, hogy a tízes számrendszerben kapott értékeket először kettes számrendszerbeli számmá alakítja át, ezután ezekkel az értékekkel kettes számrendszerben végzi a műveleteket. Végül az eredményt visszaalakítja tízes számrendszerbeli számmá. Könnyű belátni, hogy ha már az első lépésben, a kettes számrendszerbe történő átalakításkor pontatlan értéket kapunk, akkor ezzel az értékkel végezve a műveleteket, nyilvánvalóan az eredmény is pontatlan lesz. A probléma a megfelelő információ kódolással, illetve programozási nyelvekben a megfelelő pontosságú típus megválasztásával kezelhető.

4. fejezet

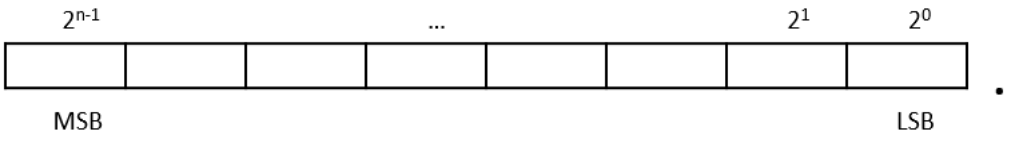
ELŐJELES SZÁMOK BINÁRIS ÁBRÁZOLÁSA. MŰVELETVÉGZÉS.

Az előző fejezetben csak a pozitív számok átalakításával foglalkoztunk. Ebben a fejezetben megnézzük, hogyan kezelhetők az előjeles egész számok. Kitérünk arra is, hogy történeti okokból a számítógépek a kivonást összeadásra vezetik vissza, ezért a negatív számokat a pozitívaktól eltérően komplemens alakban tárolják, és kezelik. Megnézzük az előjeles szám előállításának menetét és áttekintjük a műveletvégzés szabályait is.

4.1. Előjeles számok ábrázolása, komplemens képzés

Egész számok ábrázolása

Először tekintsük át azt, hogy általánosan hogyan ábrázoljuk az egész számokat, egyelőre előjeltől függetlenül. Mivel informatikai rendszerekkel foglalkozunk, fontos, hogy a binárisan ábrázolt számok mérete rögzített legyen. Ez azt jelenti, hogy az összes számot egységesen a meghatározott méreten kell felírni (pl.: 8, 16, 32 bit). Az egész számok általános alakja a következő:



Az ábra fontosabb jelölései:

MSB (Most Significant Bit): a legnagyobb helyiértékű bit, legtöbbször a szám előjele

LSB (Least Significant Bit): a legkisebb helyiértékű bit, ami közvetlenül a bináris pont előtt helyezkedik el

A legnagyobb ábrázolható pozitív szám: 2^n-1, mivel a nullát is ábrázolnunk kell. Például 8 bites szám esetén 2^8=256 féle számot tudunk ábrázolni, de az ábrázolható számok tartománya: [0,255].

Előjeles számok ábrázolása

A fentiekben az előjel nélküli egész számok ábrázolását láttuk. Ez abban az esetben használható, ha például a programban olyan számtípust választunk, ami csak pozitív értéket vehet fel, hiszen ekkor nem kell az előjellel foglalkozni, mivel az egyértelmű. Abban az esetben, ha negatív számokat is szeretnénk tárolni, a számot előjeles számként kell ábrázolni. Ekkor a szám előjelét is tárolni kell, ezért az előbbi alak úgy módosul, hogy az első biten a szám előjelét adjuk meg. Az előjel bit értéke 0, ha a szám pozitív, és 1, ha a szám negatív. Az előjel bit után pedig a szám abszolút értékét ábrázoljuk. Ez az úgynevezett előjeles abszolút értékes alak.

Előjeles számok esetén a legnagyobb ábrázolható pozitív szám: $2^{n-1} - 1$. A kitevő az előjel nélküli számhoz képest eggyel csökken, hiszen eggyel kevesebb bit áll rendelkezésünkre a szám ábrázolására az előjel bit miatt.

A legnagyobb ábrázolható negatív szám: 2^{n-1}. Ez eggyel nagyobb, mint a pozitív számok esetén. Ennek az az oka, hogy a nullát nem ábrázoljuk pozitív és negatív előjellel is, a pozitív számok tartományához tartozónak vesszük.

Negatív számok ábrázolása

A fent látott előjeles abszolútértékes alak számunkra jól kezelhető, de a számítógép számára nem, számára a műveletvégzés bonyolultabb lenne ezzel az alakkal. Régen a számítógépek óriási méretűek voltak, elektroncsövesek, rengeteg pénzbe kerültek, rengeteg energiát fogyasztottak, ezért oda kellett figyelni, hogy a lehető legkevesebb energiát használják. Az alapműveletek közül az összeadás a legfontosabb művelet, hiszen a szorzás ismételt összeadás, az osztás ismételt kivonás, és a kivonás is visszavezethető összeadásra a komplemens képzés által. Ezt kihasználva elég volt, ha összeadó volt a számítógépben, nem kellett külön kivonógépet építeni. Ahhoz, hogy a műveletek ennek megfelelően elvégezhetőek legyenek, a negatív számokat komplemens alakra kell hozni és ilyen formában kell elvégezni velük a műveletet. A fenti okokból a következőkben a komplemens képzést fogjuk megnézni, illetve azt, hogy ez hogyan használható fel kivonáskor [1].

Komplemens képzés

A komplemens képzés során az előjeles abszolútértékes alakból indulunk ki. Fontos, hogy csak a negatív számok esetén van értelme a komplemens alaknak, a pozitív számokat előjeles abszolút értékes formában tároljuk és a műveletvégzés során is ebben az alakban dolgozunk velük.

A komplemens alak előállításának lépései:

1. Előállítjuk a szám előjeles abszolút értékes alakját:

   előjel bit + a szám abszolút értéke

2. Az előjeles abszolút értékes alakot egyes komplemens alakra hozzuk: előállítjuk a szám abszolút értékét (vagyis az előjeles abszolútértékes alaknak a szám abszolút értékét tartalmazó részét, előjel nélkül) 2^{n-1}-1-re (a legnagyobb ábrázolható pozitív számra) kiegészítő számot. Ezt a számot úgy kapjuk meg, ha az előjeles abszolútértékes alakban minden bitet az ellenkezőjére fordítunk, kivéve az előjel bitet.

   Az egyes komplemens alak azonban csak egy eszköz, egy közbülső lépés a kettes komplemens előállításához, közvetlenül nem ezt használjuk fel.

3. Az egyes komplemens alakot kettes komplemens alakra hozzuk:

   egyes komplemens + 1

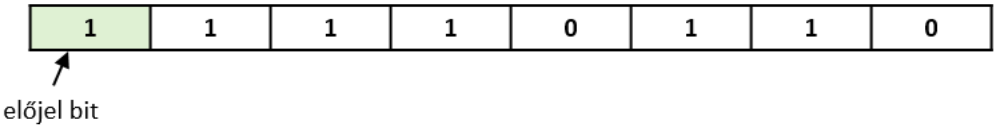
   A negatív számokat ebben a formában kezeljük a műveletvégzés során, a pozitív számokat pedig előjeles abszolút értékes alakban.

Nézzük meg a kettes komplemens alak előállítását egy példán keresztül is.

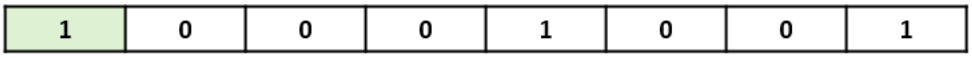
Az átalakítandó szám:

N = -118

1. Az előjeles abszolútértékes alak felírása 8 biten. Az előző fejezetben az egészrész nagyobból kisebb számrendszerbe történő átalakításakor látott módszer szerint átalakítjuk a számot kettes számrendszerbe és ezt az értéket írjuk fel a 2. bittől kezdődően. Az első bit az előjel számára van fenntartva. Mivel a szám negatív, az előjel bit 1.



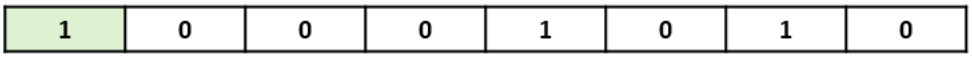
2. Az egyes komplemens alak előállítása (bitek átforgatása az ellenkezőjére, kivéve az előjel bit):



3. A kettes komplemens alak előállítása (egyes komplemens+1) :

1	0	0	0	1	0	0	1	Egyes komplemens
0	0	0	0	0	0	0	1	+1
1	0	0	0	1	0	1	0	Kettes komplemens

4. Vagyis a kapott kettes komplemens alak:



4.2. Műveletvégzés előjeles számokkal

Korábban említettük, hogy a számítógép a kivonást úgy végzi, hogy a kivonandó számot kettes komplemens alakra hozza, míg a pozitív szám előjeles abszolútértékes alakban van megadva, majd összeadást végez. Mielőtt megnéznénk a műveletvégzés menetét, ismerjük meg a bináris műveletvégzés szabályait.

Műveletvégzés bináris számokkal

+	0	1
0	0	1
1	1	10

12. ábra Összeadási táblázat

*	0	1
0	0	0
1	0	1

13. ábra Szorzási táblázat

Megjegyzés: amikor az eredmény 2 bites, a kisebb helyiértékű bitet leírjuk, a nagyobb helyiértékű pedig átvitelként jelenik meg az előző bitnél (pl. 10 esetén a 0-t leírjuk, az 1-et tovább visszük)

Műveletvégzés előjeles számokkal

A következőkben azt nézzük meg, hogy mi a műveletvégzés általános menete, ha a kivonást összeadásra vezetjük vissza, vagyis a kivonandót negatív számként ábrázoljuk, majd a két számot összeadjuk.

1. Felírjuk a számok előjeles abszolútértékes alakját.
2. A pozitív számokat az előjeles abszolútértékes alakban hagyjuk, a negatív számokat (kivonandót) átírjuk kettes komplemens alakra.
3. Elvégezzük az összeadást.
4. Megvizsgáljuk az eredmény előjelét, majd ennek megfelelően járunk el. (Lásd lent)

A művelet eredményének kezelése előjeles számok esetén

1. Ha az eredmény pozitív (előjel bit 0):
    - Az előjeles abszolútértékes alakot kaptuk meg, készen vagyunk, a szám közvetlenül átalakítható tízes számrendszerbe.
2. Ha az eredmény negatív (előjel bit 1):
    - Az eredmény kettes komplemens alakban van, ezért vissza kell alakítanunk előjeles abszolútértékes alakra, hogy átalakíthassuk tízes számrendszerbeli számmá
    - Minden bitet az ellenkezőjére fordítunk, kivéve az előjel bitet.

- Az eredményhez hozzáadunk egyet, így megkapjuk az előjeles abszolút értékes alakot, ami már közvetlenül átalakítható.

Nézzük meg a kettes komplemens alak előállítását példákon keresztül is.

1. Feladat:
   Az elvégzendő művelet: 46 - 111

   Megoldás:

   1. Írjuk fel a számok előjeles abszolútértékes alakját:

46:	0	0	1	0	1	1	1	0
-111:	1	1	1	0	1	1	1	1

   2. Állítsuk elő a negatív szám kettes komplemens alakját:

1	0	0	1	0	0	0	1

   3. Végezzük el a műveletet:

	0	0	1	0	1	1	1	0	46 (előjeles absz. ért.)
+	1	0	0	1	0	0	0	1	-111 kettes kompl.
	1	0	1	1	1	1	1	1	eredmény

   4. Mivel az előjelbit 1, az eredmény negatív, vagyis kettes komplemens alakban van, át kell alakítanunk előjeles abszolútértékes alakra, hogy tízes számrendszerbeli számmá alakíthassuk:

1	1	0	0	0	0	0	0	Forgassuk át a biteket
0	0	0	0	0	0	0	1	+1
1	1	0	0	0	0	0	1	Az eredménye előjeles absz. ért. alakja

   5. Átalakítás tízes számrendszerbe: -(26+20)=-65

2. Feladat:
   Az elvégzendő művelet: 118-116

   Megoldás:

   1. Írjuk fel a számok előjeles abszolútértékes alakját:

118:	0	1	1	1	0	1	1	0
-116:	1	1	1	1	0	1	0	0

   2. Állítsuk elő a negatív szám kettes komplemens alakját:

1	0	0	0	1	1	0	0

   3. Végezzük el a műveletet:

	0	1	1	1	0	1	1	0	118 (előjeles absz. ért.)
+	1	0	0	0	1	1	0	0	-116 kettes kompl.
1	0	0	0	0	0	0	1	0	eredmény

      Az eredmény elején egy plusz bit keletkezik az átvitelek miatt. Ennek értéke 1, ami miatt azt gondolhatnánk, hogy az eredmény negatív. Azonban tudjuk, hogy a számokat 8 biten ábrázoltuk, ezért az eredmény is csak 8 bites lehet. Az a plusz bit ebbe a méretbe nem fér bele, ezért ezt túlcsordulásnak hívják, az eredmény előállításakor ezt elveszítjük. Éppen ezért vele nem kell foglalkoznunk, a 8 bit elején szereplő bit lesz az előjel bit, vagyis 0.

   4. Az eredmény pozitív, vagyis előjeles abszolút értékes alakban adott, nem szükséges további átalakítás.

   5. Átalakítás tízes számrendszerbe: 21=2

5. fejezet

TÖRT SZÁMOK BINÁRIS ÁBRÁZOLÁSA. MŰVELETVÉGZÉS.

Ebben a fejezetben megismerkedünk a valós számok bináris ábrázolási módjaival. Ennek ismerete mérnöki feladatok esetén különösen fontos, hiszen lényegesen gyakrabban találkozunk velük, mint az egész számokkal. Áttekintjük a fixpontos és a lebegőpontos ábrázolás lényegét, megadjuk a számok általános alakját, tárolási módját, és a műveletvégzés szabályait. Megismerjük a karakterisztika eltolt kitevős alakját, illetve a mantissza implicit ábrázolási módját. Kitérünk az adatok bájtjainak tárolási és/vagy hálózaton való továbbításának sorrendjére is, ami a különböző eszközök, szoftverek közötti hordozhatóságot alapvetően érinti, hiszen a különböző bájtsorrendek alkalmazása esetén teljesen különbözőképpen értelmezhetjük az eredményt, ami komoly problémákat okozhat. A bájtsorrend ismeretében azonban a probléma kezelhető.

5.1. Fixpontos ábrázolás

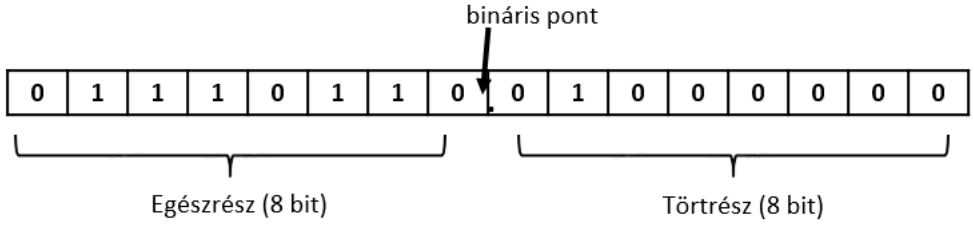
A valós számokat korábban fixpontos alakban ábrázolták, ami onnan kapta a nevét, hogy a bináris pont helye rögzített. Ez azt jelenti, hogy mivel a szám ábrázolására fordítható bitek száma adott és ezen belül a pont helye rögzített, ezáltal az is fixen adott, hogy hány biten tudjuk ábrázolni a szám egészrészt, illetve a törtrészét. A rögzített méret alapvetően meghatározza az ábrázolható számok pontosságát és nagyságrendjét, hiszen ez a pont helyétől függ. Ha a bináris pontot balra toljuk, akkor az ábrázolható szám pontossága nő, a nagyságrendje pedig csökken, mivel több bit áll rendelkezésre a törtrész leírására és kevesebb az egészrészre. Ha jobbra toljuk a bináris pontot, akkor pedig éppen ellenkezőleg, az ábrázolható nagyságrend nő, a pontosság pedig csökken, mivel több bit áll rendelkezésre az egészrész leírására és kevesebb a törtrészre.

Nézzük meg a fixpontos ábrázolás lényegét egy példán keresztül is:

Az ábrázolandó szám:

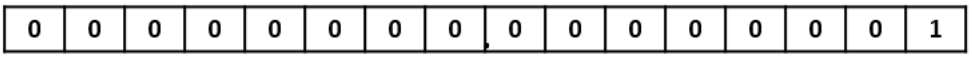
$$N = 118{,}25$$

A szám egészrészét és törtrészét külön, a már ismert módon állítjuk elő, majd az egészrészt a bináris pont elé, a törtrészt pedig a bináris pont mögé írjuk, a megadott méretre nullákkal kiegészítve (az egészrész esetén a bináris alak elé, a törtrész esetén a bináris alak végére írjuk a nullákat).



Ez az ábrázolás rendkívül egyszerű, hiszen a számrendszerek közötti átalakításon kívül nincs más dolgunk. Azt is meg kell említeni azonban, hogy ez az ábrázolási mód tárhely szempontjából nem gazdaságos, mivel a nagyon kicsi és a nagyon nagy számokat csak sok biten tudjuk ábrázolni, feleslegesen tárolva olyan információt, amit lényegesen rövidebben is le lehetne írni.

A gazdaságtalan tárolásra láthatunk példát az alábbiakban, ahol egyetlen bit kivételével mindenhol 0 bitet tárolunk, tehát az egyetlen 0-tól különböző bit leírására összesen 16 bitet foglalunk le:



5.2. Lebegőpontos ábrázolás

A feleslegesen nagy helyfoglalású fixpontos ábrázolás helyett egy olyan ábrázolási módra volt szükség, ami képes a hatékonyabb tárolásra, ezáltal lehetővé téve a számok pontos, ugyanakkor kis helyfoglalású ábrázolását. A lebegőpontos ábrázolás eleget tesz ezeknek a feltételeknek, azáltal, hogy a fixpontos ábrázolásnál látott egyszerű bináris számfelírás helyett a szám normál alakját tárolja, vagyis az egyszerű bináris alakra hozás után a bináris pontot eltoljuk úgy, hogy a pont után az első bit 1 legyen, az egészrész pedig 0. Természetesen annak érdekében, hogy a szám értéke ne változzon, a kapott értéket be kell szorozni az eltolásnak megfelelő kettő hatvánnyal:

$$N = \pm M \cdot 2^{\pm k} \tag{15}$$

ahol *N* a felírandó szám, *M* a mantissza, *k* a karakterisztika

Az ábrázolható számok pontossága a mantissza, míg az ábrázolható számok nagyságrendje a karakterisztika méretétől függ. Ezek méretét IEEE szabványok rögzítik különböző számtípusok esetére. A C# programozási nyelv néhány alapvető számtípusát a 2. táblázat szemlélteti [16].

Típus	Tartomány	Méret	Mantissza	Karakterisztika
Single	**$1{,}5\cdot10^{-45}$...$3{,}4\cdot10^{38}$**	**4 byte**	**23 bit**	**8 bit**
Real	**$2{,}939\cdot10^{-39}$...$1{,}701\cdot10^{38}$**	**6 byte**	**39 bit**	**8 bit**
Double	**$5{,}0\cdot10^{-324}$...$1{,}797\cdot10^{308}$**	**8 byte**	**52 bit**	**11 bit**
Extended	**$3{,}4\cdot10^{-4932}$...$1{,}1\cdot10^{4932}$**	**10 byte**	**63 bit**	**15 bit**

2. táblázat C# alapvető típusainak jellemzői

A mantissza és a karakterisztika ismeretében a számábrázolás a következőképpen történik. Az első bit a szám előjele (mantissza előjel), ezután a karakterisztika méretének megfelelő számú biten ábrázoljuk a karakterisztikát az előjellel együtt, majd a mantissza méretének megfelelő számú biten a mantissza abszolút értékét, amint ezt az alábbi ábrán is láthatjuk.

Mantissza előjel	Karakterisztika az előjellel együtt	Mantissza

14. ábra A lebegőpontos számok általános felírása

A lebegőpontos számábrázolás lépései

1. A szám felírása binárisan a már ismert módon, külön átalakítva az egészrészt és a törtrészt.
2. A kapott szám normál alakra hozása, vagyis úgy toljuk el a bináris pontot, hogy a kapott szám egészrésze 0 legyen, a törtrész első bitje pedig 1. Ehhez természetesen a megfelelő kettő hatvánnyal be kell szoroznunk ezt a számot, hogy az értéke ne változzon.
3. A mantissza és a karakterisztika bináris alakjának segítségével a szám felírása a 14. ábrának megfelelő módon.

Nézzünk meg egy példát is a lebegőpontos számok előállítására

Írjuk fel ugyanazt a számot, mint a fixpontos számábrázolás esetén.

$$N = 118{,}25$$

1. A szám bináris alakja: 1110110.01
2. Normál alakra hozás: $0.111011001*2^7$
3. Mantissza: 111011001

   Karakterisztika: 111 (a 7 bináris alakja)

Megjegyzés: A szám 14. ábrán látható ábrázolását a későbbiekben nézzük meg részletesen, a műveletvégzés során az itt előállított alaknak megfelelő értékekkel dolgozunk.

A lebegőpontos ábrázolás jellemzői

- A kis számokat nagy pontossággal tudjuk ábrázolni. A pontosság a mantissza méretétől, vagyis a választott számtípustól függ.
- A nagy számokat kis pontossággal tudjuk ábrázolni, de itt nem is annyira fontos, hiszen minél nagyobb a szám nagyságrendje, annál inkább elhanyagolható a tizedesek pontossága. A szám nagyságrendje a karakterisztika méretétől, vagyis a választott számtípustól függ.

5.3. Lebegőpontos számok összeadása

Abban az esetben, ha valós számokkal szeretnénk műveletet végezni, bizonyos esetekben a lebegőpontos alak előállítása után még további átalakításra is szükség lehet. Az alábbiakban a lebegőpontos számokkal végzett összeadás menetét ismertetjük.

1. Lebegőpontos számábrázolás esetén a számok normál alakját tároljuk. Mivel ezt a szabályt betartva a számok karakterisztikája különböző lehet, az összeadást nem mindig tudjuk közvetlenül elvégezni. Ahhoz, hogy össze tudjuk adni őket, először át kell alakítani a számokat úgy, hogy a karakterisztikák megegyezzenek. Ezt a műveletet a karakterisztikák illesztésének nevezzük, ami úgy történik, hogy a kisebb karakterisztikát toljuk a nagyobb felé, vagyis azt a számot alakítjuk át, amelynek a normál alakjában a kisebb kettő hatvány szerepel. Ennek a számnak a bináris pontját úgy toljuk el, hogy az eltolás után a karakterisztikája megegyezzen a másik, vagyis a nagyobb nagyságrendű szám karakterisztikájával.
2. Miután a karakterisztikák megegyeznek, elvégezhetjük az összeadást, hiszen a mantisszákban a bitek helyiértéke megegyezik.
3. A kapott eredményt normál alakra hozzuk, ha szükséges.

Nézzünk meg egy példát lebegőpontos számok összeadására.

Végezzük el a következő műveletet kettes számrendszerben lebegőpontos számábrázolást alkalmazva:

$$N_1 = 16{,}5$$

$$N_2 = 7{,}5$$

$$N_1 + N_2 = ?$$

1. Előállítjuk a számok kettes számrendszerbeli alakját:

$$N_1 = 10000.1$$

$$N_2 = 111.1$$

2. A kettes számrendszerbeli alakot normál alakra hozzuk:

$$N_1 = 0.100001*2^5$$

$$N_2 = 0.1111*2^3$$

3. Illesztjük a karakterisztikákat a kisebbet tolva a nagyobb felé. Mivel N_1 a nagyobb karakterisztikájú szám, ezért őt változatlanul hagyjuk, N_2 karakterisztikáját toljuk el úgy, hogy az az N_1 karakterisztikájával egyező legyen. Ez azt eredményezi, hogy kettővel balra toljuk a bináris pontot, hogy a megnövelt karakterisztikával is ugyanazt az értéket kapjuk

$$N_1 = 0.100001*2^5$$

$$N_2 = 0.001111*2^5$$

4. Elvégezzük a műveletet:

$$\begin{array}{r} 0.100001 \\ +\ 0.001111 \\ \hline 0.110000 \end{array}$$

Vagyis az eredmény $0.110000*2^5$

5. Az eredmény normál alakban van, ezért további átalakítás nem szükséges.

6. Ellenőrizzük az eredményt: $11000 = 2^4 + 2^3 = 24$

5.4. Lebegőpontos számok szorzása

A lebegőpontos számok szorzása esetén a normál alakra hozás után már nem szükséges semmilyen átalakítás, a művelet ebben az alakban közvetlenül elvégezhető. Az alábbiakban a lebegőpontos számokkal végzett szorzás menetét ismertetjük.

1. A számokat a korábban látott általános alakban írjuk fel:

$$N_1 = M_1 \cdot 2^{k_1}$$

$$N_2 = M_2 \cdot 2^{k_2}$$

2. Ekkor a szorzás a következőképpen végezhető el:

$$N_1 \cdot N_2 = M_1 \cdot M_2 \cdot 2k_1 + k_2 \qquad (16)$$

vagyis összeszorozzuk a mantisszákat, a karakterisztikákat pedig összeadjuk.

3. Az eredményt normál alakra hozzuk, ha szükséges.

Nézzünk meg egy példát a lebegőpontos számok szorzására:

Végezzük el a következő műveletet kettes számrendszerben lebegőpontos számábrázolást alkalmazva:

$N_1 = 4{,}25$

$N_2 = 2{,}5$

$N_1 * N_2 = ?$

1. Előállítjuk a számok kettes számrendszerbeli alakját:

   $N_1 = 100.01$

   $N_2 = 10.1$

2. A számok kettes számrendszerbeli alakját normál alakra hozzuk:

   $N_1 = 0.10001*2^3$

   $N_2 = 0.101*2^2$

3. Elvégezzük a műveletet:

$$\begin{array}{r} 0.10001 * 0.101 \\ \hline 10001 \\ \quad 00000 \\ \quad\quad 10001 \\ \hline 0.01010101 \end{array}$$

3. A kapott eredmény mantisszája: 0.01010101

   A karakterisztika: $k_1 + k_2 = 3 + 2 = 5$

   Vagyis az eredmény: $0.01010101*2^5$

4. Az eredményt normál alakra hozzuk:

   $0.01010101*2^5 = 0.1010101*2^4$

5. Ellenőrizzük az eredményt: $1010.101 = 2^3 + 2^1 + 2^{-1} + 2^{-3} = 10{,}625$

5.5. A karakterisztika eltolt kitevős ábrázolása. A mantissza implicit alakja

A karakterisztika felírása

Az általános alak felírásakor láttuk, hogy a szám ábrázolásakor a karakterisztika előjelét nem tároljuk külön, hanem a karakterisztikával együtt, ahogy az alábbi ábrán is látható. Ennek az az oka, hogy a kitevő ábrázolása legtöbbször feszített módban, vagyis a karakterisztika eltolásával történik. Ez azt jelenti, hogy a karakterisztikát eltoljuk a pozitív számok tartományába annak érdekében, hogy a negatív kitevő is ábrázolható legyen, miközben az előjelet nem kell külön ábrázolni. Így nem kell külön bitet lefoglalni az előjel számára, a karakterisztika az eltolás miatt magában hordozza az előjelét is [1], [17].

Mantissza előjel	Karakterisztika az előjellel együtt	Mantissza

A pozitív számok tartományába való eltolás azt jelenti, hogy a nulla értéket a karakterisztika tartományának közepébe toljuk el, ami a következőképpen határozható meg:

$$\frac{2^n-1}{2} \tag{17}$$

ahol *n* a karakterisztika mérete, a képlet pedig valójában a következő értéket adja meg:

$$\frac{legnagyobb\ ábrázolható\ pozitív\ szám}{2}$$

A számláló értéke mindig páratlan szám lesz, hiszen egy kettő hatványból vonunk ki egyet, így az eredmény sosem lesz egész szám. Az ábrázolhatóság érdekében azonban egy egész számot kell megadnunk, ezért a szabvány szerint lefelé kell kerekíteni.

Például 8 bit esetén:

$$\frac{2^8-1}{2} = \frac{255}{2} = 127{,}5$$

ekkor a 0 pont, vagyis a 0 értékű karakterisztika 127-nél lesz. A 127-nél kisebb értékek képviselik a karakterisztika negatív tartományát, a nála nagyobbak pedig a pozitív tartományt.

Nézzük meg példákon keresztül is a karakterisztika eltolt kitevős ábrázolását

8 biten ábrázolt karakterisztika esetén, ahogy az előbb levezettük, a 0 pont 127-be tolódik.

1. példa: k = 5

   Mivel a 0 pont a 127-es értéknél van, ennek eltolt kitevős alakját a következőképpen határozzuk meg:

   A nulla pont értéke + k = 127 + 5 = 132

   Ennek bináris alakja, vagyis a karakterisztika: 10000100

2. példa: k = -3

   Mivel a 0 pont a 127-es értéknél van, ennek eltolt kitevős alakját a következőképpen határozzuk meg:

   A nulla pont értéke + k = 127 + (-3) = 124

   Ennek bináris alakja, vagyis a karakterisztika: 01111100

A mantissza implicit ábrázolása

Lebegőpontos számok esetén a mantissza felírásának is van egy „takarékosabb” módja, amivel kevesebb bit szükséges a szám ábrázolásához. A módszer alapötlete az, hogy mivel a lebegőpontos számokat mindig normál alakban ábrázoljuk, aminek az a szabálya, hogy a bináris pont után mindig 1 szerepel, ezt kihasználva lehetővé válik a mantissza implicit ábrázolása. Ez azt jelenti, hogy a bináris pont utáni bitet nem ábrázoljuk, csak az utána következőket. Műveletvégzéskor természetesen vissza kell hozni ezt a bitet is, de tárolás szempontjából gazdaságosabb.

Nézzünk meg egy példát a mantissza implicit ábrázolási módjára:

Írjuk fel a következő szám lebegőpontos alakját a mantissza implicit ábrázolási módját alkalmazva.

$$N = 118{,}25$$

1. A szám bináris alakjának előállítása: 1110110.01
2. A szám felírása normál alakban: $0.111011001*2^7$
3. A szám mantisszája: 111011001

   ennek implicit alakja a bináris pont után 1-et elhagyva: 11011001

(A karakterisztika 7, ha ezt 8 biten szeretnénk ábrázolni, akkor 127-tel kell eltolni, vagyis a 134-et kell ábrázolni → 10000110)

5.6. Bájtsorrend

A bájtsorrend az adatok bájtjainak tárolási és/vagy hálózaton való továbbításának sorrendjét jelöli (egy memóriacímhez relatívan). Ez a sorrend alapvetően érinti a különböző eszközök, szoftverek közötti hordozhatóságot, hiszen a különböző bájtsorrendek alkalmazása esetén teljesen különbözőképpen értelmezhetjük az eredményt, ami komoly problémákat okozhat. Ezért az alkalmazott bájtsorrend ismerete alapvető fontosságú, hiszen a probléma csak ennek ismeretében kezelhető.

Adattárolás és adatátvitel esetén egyaránt egy bájt tekinthető elemi egységnek, ezért az egybájtos adatok sorozata (pl.: ASCII, UTF-8) nem érintett. A többi esetben a bájtsorrend ismeretében kezelhető a probléma.

Lehetséges bájtsorrendek

Little-endian: „kicsi elöl”, vagyis növekvő bájtsorrendet használ, az LSB-t tartalmazó bájt az első, vagyis a legmagasabb helyiértékű bájt kerül a legmagasabb címre.

Például 532 ábrázolása a következőképpen történik:

100								**101**							
0	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Big-endian: csökkenő bájtsorrend, MSB-t tartalmazó bájt az első, vagyis a legmagasabb helyiértékű bájt kerül a legalacsonyabb címre (mi is így írtuk fel a számokat a korábbiakban).

Például 532 ábrázolása a következőképpen történik:

100								101							
0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0

A táblázatok felső sorában a lehetséges memória címek szerepelnek, az alsó sorban pedig az ott tárolt érték.

Hordozhatósági problémák

Ahogy a fentiekben is említettük, az alkalmazott bájtsorrend alapvetően érinti a szoftverek hordozhatóságát, hiszen a különböző bájtsorrendek alkalmazása esetén teljesen különbözőképpen értelmezhetjük az eredményt. Bináris formában tárolt adat értelmezésekor használt bitmaszk esetén különböző eredményt kapunk a bájtsorrendtől függően, ami komoly problémákat okozhat. A probléma csak az alkalmazott bájtsorrend ismeretével kezelhető.

A probléma megoldási lehetőségei olyan szoftverek esetén, amelyeknek információt kell megosztaniuk különböző eltérő bájtsorrendű hálózati csomópontok között:

- Kiválasztanak egy adott bájtsorrendet és csak azt használják
- Az adathoz hozzáfűzve közlik, hogy milyen bájtsorrendet használtak

A bájtsorrend kezelés mindkét esetben a fogadó dolga a fenti lehetőségek alapján. A legtöbb Internet szabvány az első megoldást támogatja.

A bájtsorrendből adódó problémák kezelése

- UTF-16 kód esetén tetszőleges bájtsorrend alkalmazható, de lehetőség van a bájtsorrend jelzésre egy 2 byte-os string használatával (Byte Order Mark – BOM). UTF-32 esetén ugyanez a lehetőség adott, de ott 4 byte-on adjuk meg a bájtsorrend jelzésére szolgáló stringet.
- Az Internet Protocol big-endian hálózati bájtsorrendet definiál, vagyis minden csomag fejlécében, több magasszintű protokollban, és fájlformátumban is, amit IP szerinti kezelése terveztek, kötelező ennek használata.

6. fejezet

NUMERIKUS KÓDOK. ALFANUMERIKUS KÓDOK

Ebben a fejezetben a numerikus kódokat tekintjük át, amelyek a számok bináris reprezentációját jelentik, vagyis tulajdonképpen azok kettes számrendszerbeli alakja. Informatikai rendszerekben azonban rögzítenünk kell a kódhosszt, vagyis azt, hogy a számokat hány biten ábrázoljuk és egységesen az összes szám esetén ezt a méretet kell alkalmaznunk. Kitérünk a hexadecimális kód fontosságára is. A fejezet második része pedig az alfanumerikus kódokat, azok fejlődését ismerteti, amelyek már a számok mellett karaktereket is tartalmaznak.

6.1. Numerikus kódok

A 3. fejezetben megismertük a számrendszerek közötti átalakítás menetét, láttuk azt, hogy egy tízes számrendszerbeli szám hogyan írható fel bináris alakban. Amikor matematikai értelemben beszélünk a számrendszerekről, akkor nem fontos, hogy a kapott szám hány biten írható le. Informatikai rendszerekben azonban rögzítenünk kell a kódhosszt, vagyis azt, hogy a számokat hány biten ábrázoljuk és egységesen az összes szám esetén ezt a méretet kell alkalmaznunk. Ekkor már nem egyszerűen kettes számrendszerbeli számokról beszélünk, hanem numerikus kódokról. A numerikus kódok kizárólag számok ábrázolására alkalmasak.

Bináris kód

A bináris kód tulajdonképpen a szám kettes számrendszerbeli alakja, azonban ahogy a bevezetőben is említettük, nagyon fontos, hogy míg egyszerű átalakítás esetén nem foglalkozunk a számjegyek (bitek) számával, addig numerikus kód esetén meg kell egyezni a kódhosszban és egységesen az

összes számot a meghatározott méreten kell felírni (pl.: 8, 16, 32 bit). Abban az esetben, ha a szám kevesebb biten is leírható lenne, mint a rögzített kódhossz, a számot nullákkal ki kell egészíteni úgy, hogy az értéke ne változzon. Ez a számok egészrésze esetén azt jelenti, hogy a szám elé írjuk a nullákat, törtrész esetén pedig a szám végére.

Például 8 bites kódhossz esetén:

N=7

a szám bináris alakja: 111

8 biten ábrázolva: 00000111

N=0,75

a törtrész bináris alakja: 0.11

8 biten ábrázolva: 0.11000000

Hexadecimális kód

A hexadecimális kód valójában a szám tizenhatos számrendszerben felírt értékét jelenti, de a bináris kódhoz hasonlóan itt is fontos, hogy mivel kódról van szó, nem egyszerűen a szám hexadecimális alakjáról, ezért rögzíteni kell a kódhosszt. A hexadecimális kód nem a számítógép által közvetlenül alkalmazott kód, csak formálisan használjuk azért, hogy a binárisan felírt értékeket rövidebben, kezelhetőbb formára tudjuk alakítani. A hexadecimális kód értékei a számok 0-9, valamint az ABC betűi A-F. Mivel a tizenhat egy kettő hatvány, ezért a kettes és a tizenhatos számrendszer közötti átalakítás pontos, ami szintén fontos szempont az alkalmazhatóság tekintetében.

Az átalakítás bináris kódból hexadecimálisba úgy történik, hogy négyesével csoportosítjuk a biteket és felírjuk ezeknek a négybites csoportoknak a hexadecimális értékét.

Bináris alak:	0011	0110	0100	1010	1100	1100	0101	0111
Hexadecimális:	3	6	4	A	C	C	5	7

A fenti példából is jól látható, hogy a hexadecimális kód sokkal rövidebb, könnyebben értelmezhető, könnyebben kezelhető.

BCD kód (Binary Coded Decimal)

A kód nevének jelentése „binárisan kódolt decimális”, ami arra utal, hogy a decimális szám minden egyes számjegyéhez egy bináris kódot rendelünk hozzá. Ez a hozzárendelés egy táblázat alapján történik. A BCD tulajdonképpen egy kódcsaládnak nevezhető, mivel többféle változata létezik, ezért több ilyen táblázat áll rendelkezésünkre. A következőkben a kód alapvető változatait fogjuk megnézni.

BCD – 8421

A 8421 tulajdonképpen a legkisebb 2 hatványok ($2^3 2^2 2^1 2^0$) egymás után történő felírásából adódik, vagyis az elnevezés arra utal, hogy a decimális számjegy 4 biten felírt kettes számrendszerbeli alakját használjuk a kódolásnál.

Decimális érték	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

3. táblázat A 8421 BCD kódtábla

Írjuk fel a következő szám BCD_{8421} alakját:

N = 519

BCD_{8421} alak: 0101 0001 1001

BCD – Háromtöbbletes

A háromtöbbletes változat esetén az elnevezés arra utal, hogy a decimális számjegy 4 biten felírt kettes számrendszerbeli alakjához hármat hozzáadva kapjuk a kódot.

Decimális érték	Háromtöbbletes
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

4. táblázat A háromtöbbletes BCD kódtábla

Ennek a kódnak az a jelentősége, hogy egyenletesebben oszlanak el a 0, 1-ek, aminek következtében az energiafelhasználás, melegedés egyenletesebb lesz.

Írjuk fel a következő szám BCD_{8421} alakját:

N = 519

$BCD_{háromtöbbletes}$ alak: 1000 0100 1100

BCD – Gray

A Gray kód jelentősége az, hogy a szomszédos számjegyek csak egy bitben térnek el egymástól. Ez elsősorban logikai áramkörökben léptetés kódolására alkalmazható (pl. számjegyvezérlés, CNC esetén).

Decimális érték	Gray
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101

5. táblázat A BCD Gray kódtáblája

Írjuk fel a következő szám BCD_{Gray} alakját:

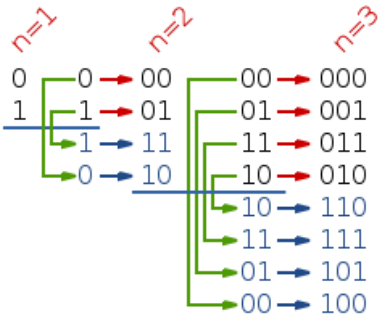
N = 519

BCD_{Gray} alak: 0111 0001 1101

A Gray kód előállítása:

Egybites kódból indulunk ki, ahol a lehetséges értékek 0 és 1. Ezeket egymás alá felírjuk, majd húzunk alá egy vízszintes vonalat, amire tükrözzük az aktuális szint kódjait, majd a következő szinten a vonal feletti résznél 0-t írunk a kapott kódok elé, a vonal alatti szinten pedig egyet. Minden egyes lépésben 1 bittel növekszik a kódméret. Addig ismételjük a tükrözést, amíg a megfelelő méretet elő nem állítottuk. Az előállítás menetét a 15. ábra szemlélteti, ahol az

aktuális szint (bitszám) kódjai rendre az n=1, n=2, n=3 oszlopokban találhatók, közöttük pedig a tükrözés lépése látható.



15. ábra A Gray kód előállítása

A BCD kód tulajdonságai

A BCD kód jól alkalmazható a korábban ismertetett, a számrendszerek közötti átalakítás esetén jelentkező pontossági problémák kezelésére, illetve ahogy a fentiekben is láttuk, bizonyos speciális feladatok esetén, mint a logikai áramkörökben való léptetés kódolására. A következőkben a kód jellemző tulajdonságait foglaljuk össze, melyek általánosan érvényesek a kód összes változatára.

Előnyei:

- Könnyű átalakítás, hiszen az értékek egy táblázatból olvashatók ki
- Pontos átalakítás oda-vissza, vagyis elkerülhető az a pontatlansági probléma, ami a tízesből kettes számrendszerbe alakításnál felmerül. Szintén abból adódik, hogy táblázatból olvassuk ki az értékeket.

Hátrányai

- Nagyobb méretű mint az egyszerű bináris kód, hiszen egy számjegy ábrázolásához 4 bit szükséges
- Redundáns, mivel 4 biten 2^4=16 féle érték ábrázolható, de csak 10 számjegyünk van.
- A műveletvégzés lassabb a művelet során keletkező átvitelek miatt.

6.2. Alfanumerikus kódok.

Az alfanumerikus kódokat olyan esetben alkalmazzuk, amikor már nem csak számokat szeretnénk ábrázolni, hanem más szimbólumokat is. A következőkben ezeket a kódokat vesszük sorra a kezdetektől.

Telex kód

- 5 bites kód, vagyis 32 különböző jelet tudunk vele kódolni (2^5)
- számokat és az angol ABC betűit tartalmazza a kódtábla
- minden kódszó kettős jelentétartalommal bír (2 kódtábla van), kapcsolójel jelzi a jelfolyamban a táblák közötti váltást, betűk előtt az 11111, számok előtt az 11011 kódot kell alkalmazni. A számváltó, illetve a betűváltó addig érvényes, amíg másik vezérlőkód nem érkezik.

KÓDJEL	BETŰ	SZÁM	KÓDJEL	BETŰ	SZÁM	
11000	A	-	11100	Q	1	
10011	B	?	01010	P	4	
01110	C	:	10100	S	,	
10010	D	Ki az?	00001	T	5	
10000	E	3	11100	U	7	
10110	F	"megj."	01111	V	=	
01011	G	"megj."	11001	W	2	
00101	H	"megj."	10111	X	/	
01100	I	8	10101	Y	6	
11010	J	csengő	10001	Z	+	
11110	K	(	00000	üres		vezérlő kódok
01001	L	)	11111	betűváltó		
00111	M	.	11011	számváltó		
00110	N	,	00100	szóköz		
00011	O	9	00010	kocsi-vissza		
01101	P	0	01000	soremelő		

16. ábra A Telex kód [18]

ISO kód

Régen a különböző gépeken különböző kódokat használtak, ami kezdetben nem okozott problémát, mert a gépek egymástól elszigetelten működtek, nem kellett megoldani a közöttük történő kommunikációt. Később azonban felmerült az igény a kódok egységesítésére, ekkor jött létre ez a kódtípus [13].

- Szervezet: International Organization for Standardization, nemzetközi szabványt határoz meg
- 7 bit információt tartalmaz + 1 bit paritást
- az ASCII őse

EBCDIC

Az EBCDIC az „Extended Binary Coded Decimal Interchange Code" elnevezés rövidítése, vagyis valójában a BCD kód kiterjesztett változata. Érdekessége, hogy a kódképzés nem egy egyszerű hozzárendelés alapján történik, hanem a karakterek sorszáma alapján. Az ASCII kóddal egyidőben, de attól függetlenül történt a fejlesztése. Elterjedésének oka, hogy bár az IBM volt az egyik fő támogatója az ASCII szabványosításának, ugyanakkor az IBM-nek nem volt ideje elkészítenie az ASCII perifériáit (mint például kártyalyukasztó), ezért a System/360 számítógép rendszereit EBCDIC kódolással szállította, mivel a cég időre csak az EBCDIC perifériáit tudta használni. A rendszer széles körben sikeres lett, vele együtt az EBCDIC is [19]. Az EBCDIC kódtáblát a 17. ábra szemlélteti.

Az EBCDIC jellemzői:

- 8 bites kód
- IBM fejlesztés
- az ASCII vetélytársa volt
- mainframe-eken ma is használják

Dec	Hx	Oct	Char		Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	nul	(Null)	65	41	101		130	82	202	b	195	c3	303	C
1	1	001	soh	(Start of Heading)	66	42	102		131	83	203	c	196	c4	304	D
2	2	002	stx	(Start of Text)	67	43	103		132	84	204	d	197	c5	305	E
3	3	003	etx	(End of Text)	68	44	104		133	85	205	e	198	c6	306	F
4	4	004	pf	(Punch Off)	69	45	105		134	86	206	f	199	c7	307	G
5	5	005	ht	(Horizontal Tab)	70	46	106		135	87	207	g	200	c8	310	H
6	6	006	lc	(Lower Case)	71	47	107		136	88	210	h	201	c9	311	I
7	7	007	del	(Delete)	72	48	110		137	89	211	i	202	ca	312	
8	8	010	ge		73	49	111		138	8a	212		203	cb	313	
9	9	011	rlf		74	4a	112	¢	139	8b	213		204	cc	314	
10	a	012	smm	(Start of Manual Message)	75	4b	113	.	140	8c	214		205	cd	315	
11	b	013	vt	(Vertical Tab)	76	4c	114	>	141	8d	215		206	ce	316	
12	c	014	ff	(Form Feed)	77	4d	115	(	142	8e	216		207	cf	317	
13	d	015	cr	(Carriage Return)	78	4e	116	+	143	8f	217		208	d0	320	}
14	e	016	so	(Shift Out)	79	4f	117	|	144	90	220		209	d1	321	J
15	f	017	si	(Shift in)	80	50	120	&	145	91	221	j	210	d2	322	K
16	10	020	dle	(Data Link Escape)	81	51	121		146	92	222	k	211	d3	323	L
17	11	021	dc1	(Device Control 1)	82	52	122		147	93	223	l	212	d4	324	M
18	12	022	dc2	dc2 (Device Control 2)	83	53	123		148	94	224	m	213	d5	325	N
19	13	023	tm	(Tape Mark)	84	54	124		149	95	225	n	214	d6	326	O
20	14	024	res	(Restore)	85	55	125		150	96	226	o	215	d7	327	P
21	15	025	nl	(New Line)	86	56	126		151	97	227	p	216	d8	330	Q
22	16	026	bs	(Backspace)	87	57	127		152	98	230	q	217	d9	331	R
23	17	027	il	(Idle)	88	58	130		153	99	231	r	218	da	332	
24	18	030	can	(Cancel)	89	59	131		154	9a	232		219	db	333	
25	19	031	em	(End of Medium)	90	5a	132	!	155	9b	233		220	dc	334	
26	1a	032	cc	(Cursor Control)	91	5b	133	$	156	9c	234		221	dd	335	
27	1b	033	cu1	(Customer Use 1)	92	5c	134	*	157	9d	235		222	de	336	
28	1c	034	ifs	(Interchange File Separator)	93	5d	135	)	158	9e	236		223	df	337	
29	1d	035	igs	(Interchange Group Separator)	94	5e	136	;	159	9f	237		224	e0	340	\
30	1e	036	irs	(Interchange Record	95	5f	137		160	a0	240		225	e1	341	
31	1f	037	ius	(Interchange Unit Separator)	96	60	140	-	161	a1	241	~	226	e2	342	S
32	20	040	ds	(Digit Select)	97	61	141	/	162	a2	242	s	227	e3	343	T
33	21	041	sos	(Start of Significance)	98	62	142		163	a3	243	t	228	e4	344	U
34	22	042	fs	(Field Separator)	99	63	143		164	a4	244	u	229	e5	345	V
35	23	043			100	64	144		165	a5	245	v	230	e6	346	W
36	24	044	byp	(Bypass)	101	65	145		166	a6	246	w	231	e7	347	X
37	25	045	lf	(Line Feed)	102	66	146		167	a7	247	x	232	e8	350	Y
38	26	046	etb	(End of Transmission Block)	103	67	147		168	a8	250	y	233	e9	351	Z
39	27	047	esc	(Escape)	104	68	150		169	a9	251	z	234	ea	352	
40	28	050			105	69	151		170	aa	252		235	eb	353	
41	29	051			106	6a	152	|	171	ab	253		236	ec	354	
42	2a	052	sm	(Set Mode)	107	6b	153	,	172	ac	254		237	ed	355	
43	2b	053	cu2	(Customer Use 2)	108	6c	154	%	173	ad	255		238	ee	356	
44	2c	054			109	6d	155		174	ae	256		239	eF	357	
45	2d	055	enq	(Enquiry)	110	6e	156	<	175	af	257		240	f0	360	0
46	2e	056	ack	(Acknowledge)	111	6f	157	?	176	b0	260		241	f1	361	1
47	2f	057	bel	(Bell)	112	70	160		177	b1	261		242	f2	362	2
48	30	060			113	71	161		178	b2	262		243	f3	363	3
49	31	061			114	72	162		179	b3	263		244	f4	364	4
50	32	062	syn	(Synchronous Idle)	115	73	163		180	b4	264		245	f5	365	5
51	33	063			116	74	164		181	b5	265		246	f6	366	6
52	34	064	pn	(Punch On)	117	75	165		182	b6	266		247	f7	367	7
53	35	065	rs	(Reader Stop)	118	76	166		183	b7	267		248	f8	370	8
54	36	066	uc	(Upper Case)	119	77	167		184	b8	270		249	f9	371	9
55	37	067	eot	(End of Transmission)	120	78	170		185	b9	271		250	fa	372	|
56	38	070			121	79	171	`	186	ba	272		251	fb	373	
57	39	071			122	7a	172	:	187	bb	273		252	fc	374	
58	3a	072			123	7b	173	#	188	bc	274		253	fd	375	
59	3b	073	cu3	(Customer Use 3)	124	7c	174	@	189	bd	275		254	fe	376	
60	3c	074	dc4	(Device Control 4)	125	7d	175	'	190	be	276		255	ff	377	eo
61	3d	075	nak	(Negative Acknowledge)	126	7e	176	=	191	bf	277					
62	3e	076			127	7f	177	"	192	c0	300	{				
63	3f	077	sub	(Substitute)	128	80	200		193	c1	301	A				
64	40	100	Sp	(Space)	129	81	201	a	194	c2	302	B				

17. ábra EBCDIC kódtábla [20]

ASCII kód

A kód neve az „American Standard Code for Information Interchange” rövidítéséből adódott.

- 8 bites kód. A 8. bit az esetek egy részében a hibafelfedést segítő paritás bit, de a mikroszámítógépek többsége a 8 bites kibővített változatát használja, amelyben a nemzeti karakterek mellett a táblázatrajzoló és grafikus karakterek is helyet kaptak, ahogy az a 18. ábrán is látható.
- 0-127 bit szabványos (angol ABC, számok, írásjelek, vezérlő kódok)
- többi bővített a nemzeti karaktereknek megfelelően, ötletszerűen bővítették, nem minden van benne

0		30	▲	60	<	90	Z	120	x	150	ľ	180	┤	210	Ď	240	
1	☺	31	▼	61	=	91	[	121	y	151	Ś	181	Á	211	Ë	241	˝
2	☻	32		62	>	92	\	122	z	152	ś	182	Â	212	ď	242	˛
3	♥	33	!	63	?	93	]	123	{	153	Ö	183	Ě	213	Ň	243	ˇ
4	♦	34	"	64	@	94	^	124	|	154	Ü	184	Ş	214	Í	244	˘
5	♣	35	#	65	A	95	_	125	}	155	Ť	185	╣	215	Î	245	§
6	♠	36	$	66	B	96	`	126	~	156	ť	186	║	216	ě	246	÷
7	•	37	%	67	C	97	a	127	⌂	157	Ł	187	╗	217	┘	247	¸
8	◘	38	&	68	D	98	b	128	Ç	158	×	188	╝	218	┌	248	°
9	○	39		69	E	99	c	129	ü	159	č	189	Ż	219	█	249	¨
10	◙	40	(	70	F	100	d	130	é	160	á	190	ż	220	▄	250	·
11	♂	41	)	71	G	101	e	131	â	161	í	191	┐	221	Ţ	251	ű
12	♀	42	*	72	H	102	f	132	ä	162	ó	192	└	222	Ů	252	Ř
13	♪	43	+	73	I	103	g	133	ů	163	ú	193	┴	223	▀	253	ř
14	♫	44	,	74	J	104	h	134	ć	164	Ą	194	┬	224	Ó	254	■
15	☼	45	-	75	K	105	i	135	ç	165	ą	195	├	225	ß	255	
16	►	46	.	76	L	106	j	136	ł	166	Ž	196	─	226	Ô		
17	◄	47	alt	77	M	107	k	137	ë	167	ž	197	┼	227	Ń		
18	↕	48	0	78	N	108	l	138	Ő	168	Ę	198	Ă	228	ń		
19	‼	49	1	79	O	109	m	139	ő	169	ę	199	ă	229	ň		
20	¶	50	2	80	P	110	n	140	î	170	¬	200	╚	230	Š		
21	§	51	3	81	Q	111	o	141	Ź	171	ź	201	╔	231	š		
22	▬	52	4	82	R	112	p	142	Ä	172	Č	202	╩	232	Ŕ		
23	↨	53	5	83	S	113	q	143	Ć	173	ş	203	╦	233	Ú		
24	↑	54	6	84	T	114	r	144	É	174	«	204	╠	234	ŕ		
25	↓	55	7	85	U	115	s	145	Ĺ	175	»	205	═	235	Ű		
26	→	56	8	86	V	116	t	146	í	176	░	206	╬	236	ý		
27	←	57	9	87	W	117	u	147	ô	177	▒	207	¤	237	Ý		
28	∟	58	:	88	X	118	v	148	ö	178	▓	208	đ	238	ţ		
29	↔	59	;	89	Y	119	w	149	Ľ	179	|	209	Đ	239	´		

18. ábra ASCII kódtábla [21]

ISO kódlapok

Az ASCII kód problémáinak kezelésére hozták létre, hiszen a nemzeti karakterek mindegyike nem ábrázolható az ASCII kódtábla 8 bitjén. Az ISO kódlapok ezt a bővítést teszik lehetővé. Többféle kódlap létezik, melyek már maradéktalanul tartalmazzák a különböző nemzeti karaktereket. Ezek az úgynevezett Latin1 (ISO/IEC 8859-1), Latin2 (ISO/IEC 8859-2) kódlapok.

- Addig jól működtek, amíg elszigetelten egy konkrét gépen, célszoftverrel használták
- Általánosabb alkalmazásokban, világszerte sok különböző karakterbeállítású gépen kell helyesen működnie (adatcsere más gépekkel, emberekkel) – ezt nem tudta teljesíteni

Unicode

Az ISO kódlapos megoldás nem bizonyult a legjobbnak, mert igaz, hogy a nemzeti karakterek is jól kezelhetők velük, de a programoknak számon kell tartaniuk, hogy melyik az a kódlap, amelyik éppen aktív és a különböző kódlapon lévő karaktereket nem lehet keverni. Az ilyen jellegű problémák kezelésére kellett egy olyan kódolás, ami egymagában képes az összes nyelv összes karakterét ábrázolni kódlapok alkalmazása nélkül. Így született meg a Unicode, amely meghatározó szerepet játszik a szoftverek nemzetközivé tételében. A mai operációs rendszer alapértelmezett Unicode támogatással rendelkeznek. A belső feldolgozás és tárolás is legtöbbször Unicode alapú.

- 8 bit helyett már 16, 32 bittel dolgozik, ezért nagyobb a helyfoglalása. A Unicode szabvány 16 biten tárolt síkokra osztja a Unicode kódpontokat. A 17 síkon 1 114 112 kódpont található, amelyből a legutóbbi változat 137 kódpontot foglal le a karakterek számára. Az első síkot alapszintű többnyelvű síknak hívjuk, melyben a legtöbb ma használatos jel megtalálható.
- Az alsó 128 érték megegyezik a hagyományos ASCII kóddal, ezért az ASCII és a Unicode közötti konverzió könnyen elvégezhető.
- Általában hexadecimális kódként adjuk meg

UTF (Unicode Transformation Format)

A Unicode szövegeket különböző karaktertárolással tárolhatjuk. A Unicode szabványhoz tartozik az UTF-8, UTF-16 és az UTF-32 karakter kódolás. Amikor különböző alkalmazások, rendszerek közötti információcsere történik, akkor általában a Unicode szabvány UTF-8 ábrázolási módját alkalmazzuk kisebb helyfoglalása és ASCII kompatibilitása miatt.

- UTF-8: változó hosszúságú kódolást használ, ami azt jelenti, hogy az ASCII karaktereket 1 byte-on önmagukkal reprezentáljuk, ennek következménye az, hogy kompatibilis az ASCII-val. A többi karaktert 2-4 byte-on ábrázoljuk (16-32 bit).

- UTF-16: változó hosszúságú kódolást használ, az alapszintű többnyelvű síkhoz 16 bitet, a többi karaktert pedig 4 byte-on ábrázolja.
- UTF-32: fix kódhosszúságú, minden karakterhez 4 byte-ot használ, így képes minden Unicode kódpontot kódolni. Nagy hátránya viszont az, hogy a fix kódhossznak köszönhetően nagy helyfoglalású, ezért ritkábban, meghatározott célra használják.

7. fejezet

AZ ENTRÓPIA FOGALMA. KERESÉSELMÉLET.

Ebben a fejezetben azt tekintjük át, hogy rendszerek esetén hogyan kezelhető az információ. Bevezetjük a rendszer bizonytalanságát leíró Shannon-entrópia fogalmát, részletesen elemezve az entrópia függvény tulajdonságait. Megmutatjuk az entrópia kapcsolatát a korábban definiált Hartley-formulával. Megismerjük az informatika egyik fontos területét, a kereséselmélet fogalmát, ahol igen-nem kérdésekkel próbáljuk megtalálni a keresési tér egy meghatározott elemét. Megvizsgáljuk azt is, hogy a kereséselméletben hogyan tudjuk felhasználni az entrópia tulajdonságait annak érdekében, hogy a keresés optimális legyen

2.4. Az információ fogalma rendszerekben

A korábban, a 2. fejezetben megismert Hartley formula eredeti és módosított változata is egyedi eseményekkel foglalkozik. A gyakorlatban azonban általában rendszerekkel dolgozunk, amikor már ezek alkalmazása önmagában nem elég, valamilyen összetettebb formulára van szükség, ami összekapcsolható a Hartley-formulával is. Rendszerek esetén az információ mennyisége egymást követő egyedi események sorozataként definiálható, vagyis elemi szimbólumok bizonyos készletéből egy sorozatot választ ki. Ebben a kiválasztásban, vagyis az üzenet előállításában a valószínűség is szerepet játszhat.

Az aktuális esemény kiválasztása függhetnek az előző választástól (pl.: nyelvi jellegzetességek, betűgyakoriság). Ebben az esetben az információ mennyiségének meghatározására már nem a Hartley formulát használjuk. Például nyelvi jellegzetességek esetén: az „Arra gondoltam,” mondatrész után a „vagy” szó valószínűsége nagy, az „elefánt” szó valószínűsége pedig egészen

kicsi. A nyelvtől függő betűgyakoriság esetén pedig, az egyes betűk különböző eséllyel fordulnak elő a szövegben. Például a magyar nyelvben az „e” betű gyakorisága nagy, az „x” gyakorisága kicsi, míg más nyelvekben ez nem feltétlenül igaz.

Amikor az aktuális esemény kiválasztása nem függ az előző eseménytől, a Hartley formulával számolható az információ mennyisége:

$$k \cdot log_2 n \tag{8}$$

ahol *k* az egymást követő elemi események száma, amelyeket ugyanabból az *n* elemű sokaságból választunk ki.

A valószínűség rendszerre gyakorolt hatása alapján megkülönböztethetünk sztochasztikus és Markov rendszereket.

Sztochasztikus folyamat

Sztochasztikus folyamatnak nevezzük azt a rendszert, amely bizonyos valószínűségek szerint szimbólumok sorozatát állítja elő.

Markov folyamat vagy Markov lánc

A sztochasztikus folyamat speciális esete, amikor a valószínűségek a megelőző eseménytől függnek.

7.1. A Shannon-entrópia fogalma

Az entrópia fogalma a fizikában.

A fizikában használt entrópia formula, amely a rendszerek rendezetlenségi fokát írja le, analóg az információ mennyiségét leíró képlettel, de itt a logaritmus alapja *e*, mivel nem informatikai rendszerekben használatos, nem bitben kell meghatározni az eredményt.

$$S = k \cdot log_e D \tag{18}$$

ahol *S* az entrópia nagysága, *k* a Boltzmann állandó (1,38*10^{-23} [J/K]), *D* pedig a rendszer különböző állapotainak száma. Mértékegysége: [Nat]

Shannon entrópia.

Informatikai rendszerekben hasonló összefüggést használunk, de itt a rendszer rendezetlensége helyett annak bizonytalanságát írjuk le. Annak érdekében, hogy megkülönböztessük a fizikai entrópia fogalmától, a Shannon entrópia nevet kapta, de a továbbiakban egyszerűen entrópiaként hivatkozunk

rá, tekintve, hogy informatikai rendszerek esetén egyértelműen a Shannon-entrópiáról beszélünk. A formula eredménye egy olyan szám, amely összehasonlíthatóvá tesz két rendszert olyan szempontból, hogy melyikben nagyobb a bizonytalanság.

A következőkben a Shannon entrópia formula levezetését nézzük meg [1].

Adottak a következők:

$X_1, X_2, \ldots, X_n$ - egyedi közlemények

$p(X_1), p(X_2), \ldots, p(X_n)$ - a közleményekhez rendelt valószínűségek

$S = X_1 + X_2 + \cdots + X_n$ - az összes üzenet

k - a kibocsátott üzenetek száma (nem az összeset küldjük)

Meghatározandó:

H(S) – az üzenet információ mennyisége

Az egyes elemek (X_i) előfordulásának száma a valószínűségük alapján határozható meg:

$$g_i = k * p(x_i) \tag{19}$$

Az i-edik üzenet információ tartalma a Hartley-formulával határozható meg:

$$I(x_i) = log_2 \frac{1}{p(x_i)} = -log_2 p(x_i) \tag{20}$$

Ebből a kibocsátott üzenetek információ tartalma:

$$I_k = g_1 \cdot I(x_1) + g_2 \cdot I(x_2) + \cdots + g_n \cdot I(x_n) \tag{21}$$

g_i-t és $I(x_i)$- t behelyettesítve *k* üzenetre: $I_k = -k \sum_{i=1}^{n} p(x_i) log_2 p(x_i)$

Az entrópia formula tehát a következő képlettel írható le:

$$H = -k \sum_{i=1}^{n} p_i log_2 p_i \tag{22}$$

ahol *H* az entrópia mennyisége; *k* konstans, kettes számrendszerben *k=1*; *n* a független jelek vagy üzenetek száma; p_i pedig a jelekhez tartozó valószínűség. Mértékegysége: [bit]

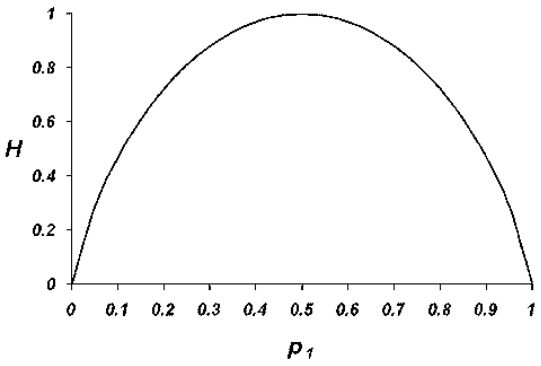
7.2. Az entrópia függvény tulajdonságai

Az entrópia függvény görbéjét két eseményre ábrázoljuk, mivel minden egyes plusz esemény eggyel növeli a függvény dimenzióinak számát. Ahhoz azonban, hogy a tulajdonságait elemezni tudjuk, két esemény is elegendő, a jellegzetességek így is jól láthatók.

Két eseményre az események valószínűségei rendre: p_1, p_2

Mivel tudjuk, hogy összefüggő eseményekre $\sum_{i=1}^{n} p_i = 1$ ebből következik, hogy $p_2 = 1 - p_1$

Felhasználva ezt az összefüggést, egy diagramon ábrázolható mindkét eseményre vonatkozóan az entrópia. Például a p_1=0,2 esetén p_2=1-0,2=0,8, vagyis a 0,2-hez és a 0,8-hez tartozó értéket kell leolvasnunk.



19. ábra Az entrópia függvény két eseményre

Az ábrán jól látható, hogy a függvény jellege alátámasztja azt az állítást, ami szerint nem az információ tartalma, hanem a mennyisége a fontos, hiszen bármelyik irányba megyünk, az entrópia értéke ugyanúgy változik. $p_1=0,5$ értéktől bármely irányban haladva ugyanolyan mértékben csökken. A másik szembeötlő dolog, hogy a szélső helyeken nincs bizonytalanság, vagyis $H(0)=0$ és $H(1)=0$. Ez a valószínűség fogalmából következik, hiszen $p_1=0$ a lehetetlen eseményt jelenti, $p_1=1$ pedig a biztos eseményt. Mivel az entrópia a rendszer bizonytalanságának jellemzésére szolgál, belátható, hogy ez az érték mind a lehetetlen, mind a biztos esemény esetén nulla kell, hogy legyen, hiszen mindkét szélsőséges esetben egyértelmű a rendszer kimenete, nem lehet bizonytalanság.

Az entrópia függvény tulajdonságai a következők:

1. Folytonos függvény p_i –n
2. Maximuma a Hartley-formulával adható meg
3. Nemnegatív
4. Monoton növekvő függvénye az elemszámnak
5. Szimmetrikus
6. Kommutatív
7. Elágazási fától független

A következőkben a fenti tulajdonságokat részletezzük.

Az entrópia függvény folytonos p_i-n

Ez a tulajdonság azt jelenti, hogy a valószínűség (p_i) kismértékű változása nem okozhat nagymértékű változást a kimenetben, vagyis az entrópia értékében, a rendszer bizonytalanságában.

Formálisan:

A és *B* rendszer entrópiáját tekintve elmondható, hogy

$$H_A(p_1, p_2, \dots, p_n) \approx H_B(p_1 + \delta, p_2 - \delta, \dots, p_n) \qquad (23)$$

ahol δ egy nagyon kicsi érték.

Az entrópia függvény maximuma

A függvény akkor éri el a maximumát, amikor teljesen szabadon választhatunk a független jelek, vagy üzenetek közül, vagyis abban az esetben, amikor a valószínűségek egyenlők.

A maximum értéke a Hartley-formulával számítható ki:

$$\log_2 \mathrm{n} \qquad (24)$$

Fontos megjegyezni, hogy csak 2 eseményre igaz az, hogy az entrópia maximuma 1, mivel $\log_2 2 = 1$, más elemszám esetén ennél nagyobb értéket fogunk kapni.

Nézzük meg példákon keresztül az entrópia formula használatát.

Egyenlő valószínűségek esetén

$p_1 = \frac{1}{2}, p_2 = \frac{1}{2}$

$$H\left(\frac{1}{2}, \frac{1}{2}\right) = -\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{2} log_2 \frac{1}{2}\right) = -\left(\frac{1}{2} \cdot -1 + \frac{1}{2} \cdot -1\right) = 1$$

vagyis ugyanazt az értéket kapjuk, mint a Hartley-formula alkalmazásakor ($log_2 2 = 1$), hiszen egyenlőek a valószínűségek.

Különböző valószínűségek esetén

$p_1 = \frac{1}{4}, p_2 = \frac{3}{4}$

$$H\left(\frac{1}{4}, \frac{3}{4}\right) = -\left(\frac{1}{4} log_2 \frac{1}{4} + \frac{3}{4} log_2 \frac{3}{4}\right) = -\left(\frac{1}{4} \cdot -2 + \frac{3}{4} \cdot -0{,}42\right) = 0{,}815$$

A példából is látható, hogy különböző valószínűségek esetén az entrópia értéke kisebb, mint a Hartley-formulával meghatározott maximum érték. Minden más különböző valószínűség esetén is hasonló eredményre jutnánk.

Az entrópia függvény nemnegatív

Az entrópia függvény soha nem vehet fel negatív értéket.

Formálisan:

$$H(P) \geq 0 \qquad (25)$$

Ezt mesterségesen szabályozzuk azáltal, hogy az entrópia formula elején egy mínusz előjelet használunk. Ennek oka, hogy így garantálhatóan mindig pozitív eredményt fogunk kapni, ami inkább használható a bizonytalanságok mérőszámaként, hiszen így nagyobb érték nagyobb bizonytalanságot jelöl. Vezessük végig, hogy miért szükséges a negatív előjel használata. Tudjuk, hogy a valószínűség egy 0 és 1 közötti szám, vagyis $0 \leq p(x_i) \leq 1$. Az 1-nél kisebb számok logaritmusa pedig mindig negatív (pl. $log_2 \frac{1}{2} = -1$). Az entrópia formula alkalmazásakor ($H = -k \sum_{i=1}^{n} p_i log_2 p_i$) ezeket a negatív értékeket összegezzük, vagyis egy negatív előjelű összeget kapunk. A képlet elején szereplő *k* értéke kettes számrendszer esetén 1, vagyis egy pozitív szám, ezért a teljes függvény eredménye negatív lenne, amiből a szumma előtt szereplő negatív előjel „csinál” pozitív számot.

Az entrópia függvény monoton növekvő függvénye az elemszámnak

Ez a tulajdonság azt mondja ki, hogy a különböző elemszámú rendszerekben, ahol az egyes független jelek, vagy üzenetek bekövetkezési valószínűsége egyenlő, annak a rendszernek az entrópiája nagyobb, amelyiknek nagyobb az elemszáma.

Formálisan:

„A” rendszer elemszáma: *n*

„B” rendszer elemszáma: *m*

ahol *n>m*, és a valószínűségek egyenlőek, vagyis $p_{A_i} = \frac{1}{n}$, $p_{B_i} = \frac{1}{m}$

Ekkor a két rendszer entrópiájára a következő összefüggés érvényes:

$$H_A\left(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}\right) > H_B\left(\frac{1}{m}, \frac{1}{m}, \dots, \frac{1}{m}\right)$$

vagyis több elem esetén nagyobb a bizonytalanság a rendszerben.

Nézzünk meg egy példát erre a tulajdonságra.

A két rendszerünket egy-egy doboz képviseli. A dobozokban golyók vannak, ezek közül kell húznunk bekötött szemmel.

1. doboz („A” rendszer): 25 db golyó, amiből 1 piros

2. doboz („B” rendszer): 50 db golyó, amiből 1 piros

Melyikben nagyobb a bizonytalanság? Hol van kisebb esélyünk a piros golyót kihúzni? Számoljuk ki az entrópia értékét mindkét rendszerre!

1. doboznál:

$$p_i = \frac{1}{25} \rightarrow H_A\left(\frac{1}{25}, \frac{1}{25}, \dots, \frac{1}{25}\right) = -25\left(\frac{1}{25} log_2 \frac{1}{25}\right) = log_2 25 = 4{,}6443$$

2. doboznál:

$$p_i = \frac{1}{50} \rightarrow H_B\left(\frac{1}{50}, \frac{1}{50}, \dots, \frac{1}{50}\right) = -50\left(\frac{1}{50} log_2 \frac{1}{50}\right) = log_2 50 = 5{,}6444$$

Ekkor a két rendszer entrópiájára a következő összefüggés érvényes:

$$H_A > H_B$$

vagyis több elem esetén nagyobb a bizonytalanság a rendszerben, hiszen ha több golyó közül kell kiválasztani az egyetlen pirosat, kisebb az esélyünk, hogy sikerül.

Az entrópia függvény szimmetrikus

A szimmetrikusság a két eseményre ábrázolt függvény esetén is nyilvánvalóan látszik, de általánosságban, vagyis több eseményre is igaz. Tetszőleges számú eseményre entrópia függvény szimmetrikussága azt jelenti, hogy a valószínűségeket bármilyen sorrendben rendeljük az egyes eseményekhez, a rendszer entrópiája nem változik.

Például: szabálytalan pénzérme esetén mindegy, hogy

$p(Fej) = \frac{3}{4},\ p(\text{Írás}) = \frac{1}{4}$ vagy $p(Fej) = \frac{1}{4},\ p(\text{Írás}) = \frac{3}{4}$

Formálisan:

$$H(p_1, p_2, \dots, p_n) = H(p_{r_1}, p_{r_2}, \dots, p_{r_n}) \tag{26}$$

ahol p_{r_1}, p_{r_2} ,…, p_{r_n} a $p1, p2, \dots, p_n$ valószínűségek egy tetszőleges permutációja.

Az entrópia függvény kommutatív

A kommutativitás az entrópia függvény esetén azt jelenti, hogy az eseményeket bármilyen sorrendben vesszük, a rendszer entrópiája ugyanannyi marad. Ez a tulajdonság látszólag megegyezik az előzővel, de lényeges különbség a kettő között, hogy míg a szimmetrikusság esetén az események sorrendje rögzített, csak a hozzájuk rendelt valószínűségek sorrendje változik, addig a kommutativitás esetén az eseményekhez rendelt valószínűségek állandóak, csak az események sorrendje változik.

Formálisan:

$$H(A, B) = H(B, A) \tag{27}$$

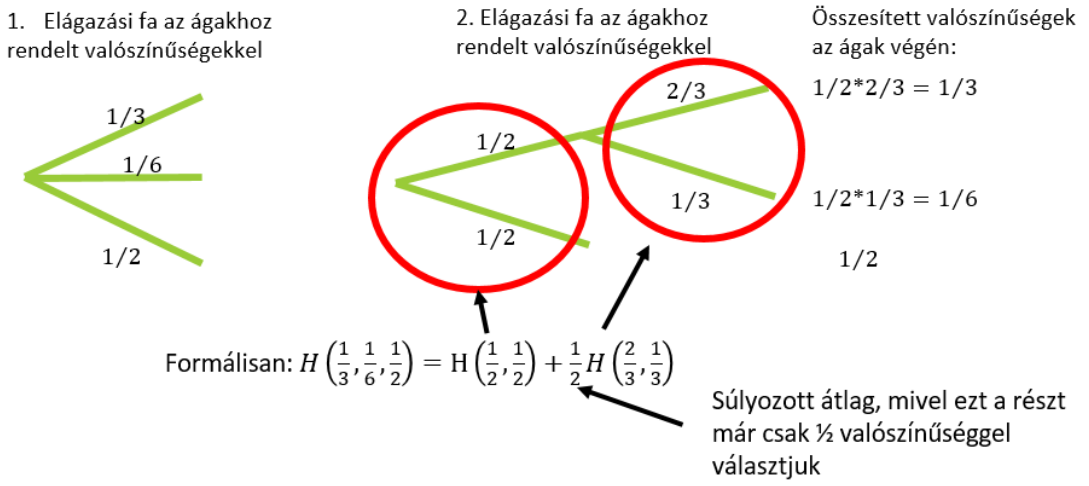
Például: szabálytalan pénzérme esetén, ha

$p(Fej) = \frac{3}{4},\ p(\text{Írás}) = \frac{1}{4}$ akkor $H(Fej, \text{Írás}) = H(\text{Írás}, Fej)$

Az entrópia függvény elágazási fától független

Az elágazási fától való függetlenség azt jelenti, hogy az információ megszerzési módja nem befolyásolja az információ mennyiségét. Ez más szavakkal azt jelenti, hogy mindegy hogy egy bonyolult, vagy egy egyszerű kérdéssorozaton keresztül jutunk el a válaszig, az információ mennyisége ugyanannyi lesz.

Ezt a tulajdonságot szemlélteti a 20. ábra. Az első esetben három lehetőség közül választunk $p_1 = \frac{1}{3}, p_2 = \frac{1}{6}, p_3 = \frac{1}{2}$ valószínűséggel, a második esetben pedig szintén három lehetőség közül választunk, de azzal a különbséggel, hogy először $p_1 = \frac{1}{2}, p_2 = \frac{1}{2}$ valószínűséggel választjuk a harmadik, vagy együttesen az első két lehetőséget, majd az első kettő közül $p_1 = \frac{2}{3}, p_2 = \frac{1}{3}$ valószínűséggel választjuk valamelyiket. Az ágakon szereplő értékeket összeszorozva azt kapjuk, hogy az első ágon $p = \frac{1}{2} * \frac{2}{3} = \frac{1}{3}$, a második ágon $p = \frac{1}{2} * \frac{1}{3} = \frac{1}{6}$, a harmadikon pedig $p = \frac{1}{2}$ az összesített valószínűség, ugyanúgy, mint az első esetben, amikor közvetlenül választunk a lehetőségek közül. Az entrópia értékének meghatározásakor minden egyes elágazást, mint külön alrendszert kell figyelembe venni, és az alacsonyabb szinteken már a hozzájuk vezető ág valószínűségével súlyozott értéket számolunk.



20. ábra Elágazási fa

7.3. Az entrópia és a kereséselmélet kapcsolata

A kereséselmélet az entrópia fogalmához, illetve az információ mennyiségéhez szorosan kapcsolódik. A következőkben ezt a kapcsolatot fogjuk megvizsgálni.

A kereséselmélet az informatika egy fontos területét képezi. Olyan típusú feladatokat foglal magában, ahol az a feladatunk, hogy egy véges halmaz egy ismeretlen elemét megtaláljuk. Ez a keresés kérdések sorozatával történik, ahol egy kérdésre adott válasz megmondja, hogy az ismeretlen elem benne van-e egy adott részhalmazban, vagy nincs. Ez azt jelenti, hogy csakis igen-nem kérdéseket tehetünk fel, vagyis az ilyen típusú feladatokba a válasz binárisan kódolható. Ez az oka annak, hogy a kereséselmélet az informatikának egy fontos területévé válhatott.

Az entrópia fogalmával kapcsolatban tudjuk, hogy a rendszer bizonytalanságát írja le, azzal foglalkozik, hogy mit választunk, és ezt milyen valószínűséggel tesszük.

Kereséselmélet esetén az a célunk, hogy minél kevesebb kérdéssel találjuk meg a választ. Ezt megpróbálhatjuk úgy elérni, hogy nagy halmazokat próbálunk kizárni, és ha ez sikerül, akkor a megmaradt kicsi halmazból kérdezünk tovább, ami jelentősen csökkentheti a kérdések számát. Ehhez azonban szerencse kell, mert ha pont a nagy halmazban van a keresett elem, akkor csak a kicsi halmaz elemei zárhatók ki és a nagy halmazzal kell folytatnunk. Következésképpen ez nem lehet jó megoldás. Az entrópia tulajdonságainál említettük, hogy a bizonytalanság akkor a legnagyobb, amikor a valószínűségek egyenlőek. Ebből az következik, hogy meg kell próbálnunk a keresési teret közel egyforma méretű részhalmazokra osztani, így minden egyes kérdéssel a lehető legnagyobb bizonytalanságot tudjuk megszüntetni.

Összefoglalva az előbbieket az entrópia és a kereséselmélet kapcsolatáról elmondható:

- Cél a maximális bizonytalanság megszüntetése (egyenlő valószínűségek esetén, ahol az entrópia maximális) – vagyis arra kell törekednünk, hogy minden egyes lépésben úgy kérdezzünk, amivel két nagyjából egyenlő elemszámú részhalmazra osztjuk a halmazunkat

- Ezzel a technikával egy optimális átlagos lépésszámot kapunk, ami azt jelenti, hogy szerencsés esetben kevesebb, szerencsétlen esetben pedig több kérdéssel jutunk a válaszhoz, de informatikában nem

hagyatkozhatunk a szerencsére, az átlagos lépésszámot kell optimalizálnunk, ami garantált.

8. fejezet

A REDUNDANCIA

Ebben a fejezetben az információban rejlő redundancia fogalmával foglalkozunk. A redundancia az üzenetnek az a része, ami elhagyható anélkül, hogy ez az üzenet értelmezhetőségét befolyásolná, ezért a későbbi fejezetekben ismertetésre kerülő veszteségmentes tömörítő algoritmusok alapjául szolgál. Definiáljuk a redundancia fogalmát, megnézzük hogyan kapcsolható ez a fogalom az entrópiához és megadjuk a kiszámítására szolgáló képletet a relatív entrópia fogalmának bevezetése segítségével.

8.1. A redundancia fogalma

Az információ fogalmához hasonlóan a redundanciát is először hétköznapi értelemben vizsgáljuk. Ebben az esetben a jelentése feleslegesség, terjengősség (pl. beszédben) vagyis általában egy negatív értelmezés kapcsolódik hozzá. Informatikai rendszerekben a redundanciára egy kicsit más szemszögből tekintünk annak ellenére, hogy itt is a bizonyos szempontból vett feleslegességet jelenti az üzenetben. Itt éppen ezt a feleslegességet kihasználva tudunk tömörebb információt létrehozni.

Informatikai definíciója: a redundancia az üzenetnek az a része, ami szükségtelen abban az értelemben, hogy ha az a rész hiányozna, az üzenet akkor is lényegében teljes, vagy teljessé tehető lenne.

Nézzük meg egy példán keresztül is a redundancia, illetve a redundáns rész elhagyásának hatását. A példa lényege, hogy egy teljes üzenetből hagyunk el bizonyos részeket és azt vizsgáljuk, hogy mennyire marad értelmezhető ezután.

Eredeti üzenet:

„Az üzenetnek az a része, ami szükségtelen abban az értelemben, hogy ha az a rész hiányozna, az üzenet akkor is lényegében teljes, vagy teljessé tehető lenne.”

3 karakterenként mindig az elsőt elhagyjuk:

z ze et ek az a és e, am s ük ég el n bb n z rt le be , og h a a ré z iá yo na a ü en t kk r s én eg be t lj s, va y el es é eh tő le ne

5 karakterenként mindig az elsőt elhagyjuk:

z üz netn k az rés e, a i sz kség elen abba az rtel mben hog ha z a ész iány zna, az ü enet akko is énye ében telj s, v gy t ljes é te ető enne

Látható, hogy bizonyos karaktereket, vagyis az üzenet egy részét elhagyva az üzenet olvasható, értelmezhető marad. Ez azt jelenti, hogy az eredeti üzenet redundáns, az eredti alaknál tömörebben is leírható.

8.2. Redundancia jelentősége

Rendszerek esetén az információ mennyiségének leírására az entrópia formula szolgál, melynek legnagyobb értéke a Hartley-formulával definiálható. Az előző példában azonban láttuk, hogy bizonyos karaktereket elhagyva a szöveg ugyanúgy értelmezhető marad, amennyiben ismerjük a nyelvi jellegzetességeket, vagyis nem biztos, hogy valóban szükséges az entrópia formula által meghatározott bitek száma az információ leírásához. A nyelvi jellegzetességek ugyanis azt is jelentik, hogy nyelvtől függően a különböző karakterekhez különböző előfordulási gyakoriság társul, ami alapján előállítható az üzenet hiányzó része. Célunk, ennek a „felesleges” résznek a meghatározása annak érdekében, hogy az üzenet tömöríthető legyen.

A gyakorlatban fix hosszúságú kódokat használunk ahogy a korábbi fejezetekben láttuk (pl.: ASCII, Unicode), ami azt jelenti, hogy minden egyes szimbólumot ugyanolyan hosszúságúbitsorozattal írunk le. Az ASCII kódtáblát a 12. ábra szemlélteti, ahol egy 8 bites kódot használunk a különböző szimbólumok leírására. Ez a tárolási forma nagyobb méretű üzenetek előállítását teszi lehetővé, hiszen a gyakran előforduló szimbólumokat ugyanannyi biten írja le, mint a ritkábbakat. Az így kapott kódot a redundanciát kihasználva tömörebben is leírhatjuk, így kisebb tárhelyre, illetve csatornakapacitásra lesz szükségünk az információ tárolása, illetve továbbítása során. A szimbólumok különböző előfordulási valószínűsége esetén lehetőségünk van arra, hogy a

szimbólumsorozat egy részét elhagyva tömörebben írjuk le. A következő fejezetekben az erre szolgáló módszereket fogjuk megnézni.

0		30	▲	60	<	90	Z	120	x	150	ľ	180	┤	210	Ď	240	
1	☺	31	▼	61	=	91	[	121	y	151	Ś	181	Á	211	Ë	241	˝
2	☻	32		62	>	92	\	122	z	152	ś	182	Â	212	ď	242	˛
3	♥	33	!	63	?	93	]	123	{	153	Ö	183	Ě	213	Ň	243	ˇ
4	♦	34	"	64	@	94	^	124	|	154	Ü	184	Ş	214	Í	244	˘
5	♣	35	#	65	A	95	_	125	}	155	Ť	185	╣	215	Î	245	§
6	♠	36	$	66	B	96	`	126	~	156	ť	186	║	216	ě	246	÷
7	•	37	%	67	C	97	a	127	⌂	157	Ł	187	╗	217	┘	247	¸
8	◘	38	&	68	D	98	b	128	Ç	158	×	188	╝	218	┌	248	°
9	○	39		69	E	99	c	129	ü	159	č	189	Ż	219	█	249	¨
10	◙	40	(	70	F	100	d	130	é	160	á	190	ż	220	▄	250	˙
11	♂	41	)	71	G	101	e	131	â	161	í	191	┐	221	Ţ	251	ű
12	♀	42	*	72	H	102	f	132	ä	162	ó	192	└	222	Ů	252	Ř
13	♪	43	+	73	I	103	g	133	ů	163	ú	193	┴	223	▀	253	ř
14	♫	44	,	74	J	104	h	134	ć	164	Ą	194	┬	224	Ó	254	■
15	☼	45	-	75	K	105	i	135	ç	165	ą	195	├	225	ß	255	
16	►	46	.	76	L	106	j	136	ł	166	Ž	196	─	226	Ô		
17	◄	47	alt	77	M	107	k	137	ë	167	ž	197	┼	227	Ń		
18	↕	48	0	78	N	108	l	138	Ő	168	Ę	198	Ă	228	ń		
19	‼	49	1	79	O	109	m	139	ő	169	ę	199	ă	229	ň		
20	¶	50	2	80	P	110	n	140	î	170	¬	200	╚	230	Š		
21	§	51	3	81	Q	111	o	141	Ź	171	ź	201	╔	231	š		
22	▬	52	4	82	R	112	p	142	Ä	172	Č	202	╩	232	Ŕ		
23	↨	53	5	83	S	113	q	143	Ć	173	ş	203	╦	233	Ú		
24	↑	54	6	84	T	114	r	144	É	174	«	204	╠	234	ŕ		
25	↓	55	7	85	U	115	s	145	Ĺ	175	»	205	═	235	Ű		
26	→	56	8	86	V	116	t	146	í	176	░	206	╬	236	ý		
27	←	57	9	87	W	117	u	147	ô	177	▒	207	¤	237	Ý		
28	∟	58	:	88	X	118	v	148	ö	178	▓	208	đ	238	ţ		
29	↔	59	;	89	Y	119	w	149	Ľ	179	│	209	Đ	239	´		

21. ábra ASCII kódtábla [21]

8.3. A relatív entrópia és a redundancia kiszámítása

A relatív entrópia formula

A redundancia számszerű meghatározásához először a rendszer relatív entrópiáját kell meghatároznunk. A relatív entrópia jelentése az üzenet kialakítása során, az azt alkotó szimbólumok kiválasztási szabadságához kapcsolódik, vagyis azt írj le, hogy a forrás üzenet kialakításához felhasználható szimbólumok kiválasztása során milyen mértékű szabadsággal rendelkezünk, ahhoz a lehetőséghez képest, amikor ezeket a szimbólumokat teljesen szabadon lehet kiválasztani. A kapott érték mindig egy egynél kisebb szám lesz, mivel a pillanatnyi entrópia értéket, a nála nagyobb maximális értékkel osztjuk (28).

$$H_{rel} = \frac{H_S}{H_H} \tag{28}$$

$$\frac{pillanatnyi\ entrópia\ (Shannon\ entrópia\ formula)}{maximális\ entrópia\ (Hartley - formula)}$$

Például:

H_S=1,6; H_H=2

$$H_{rel} = \frac{H_S}{H_H} = \frac{1{,}6}{2} = 0{,}8$$

Az eredmény azt jelenti, hogy az üzenet kiválasztásakor 80%-os szabadsággal rendelkezünk.

A redundancia kiszámítása

A redundanciáról tudjuk, hogy az üzenetnek az a része, ami elhagyható anélkül, hogy ez problémát okozna az üzenet értelmezésében, tehát veszteségmenetesen tudjuk az információ méretét csökkenteni. Értéke a relatív entrópiából számítható, és mindig 1-nél kisebb lesz.

$$Redundancia = 1 - H_{rel} \qquad (29)$$

Példa: Az előző példa folytatásaként, ha H_{rel}=0,8, akkor a fenti képlet alapján számítható. Redundancia=1-0,8=0,2 vagyis az üzenet 20%-a elhagyható.

9. fejezet

A KÓDOLÁS FOLYAMATA. KÓDFA. PREFIX KÓD.

Ebben a fejezetben megismerjük az információ kódolás folyamatát, ami informatikai rendszerek esetén nélkülözhetetlen, hiszen a számítógépek minden adatot bináris formában kezelnek, függetlenül attól, hogy ez az adat szám, szöveg, kép, videó, hang, vagy bármi egyéb. Megadjuk a különböző célú kódolási típusok definícióját és felsoroljuk a velük szemben támasztott követelményeket. Ismertetjük a veszteséges és a veszteségmentes adattömörítés lényegét, áttekintjük a különböző algoritmusok hatékonyságát. Megismerjük a kódolás alapvető eszközeit, a kódfát és a prefix kódot, valamint a jellemzésükre szolgáló formulákat. Megnézzük, hogyan tehetjük a kódot egyértelműen dekódolhatóvá, vagyis elválasztó karakterek nélkül is értelmezhetővé. Tárgyaljuk a változó hosszúságú kód jelentőségét is.

9.1. A kódolás folyamata

Ahogy korábban is volt róla szó, az üzeneteket az adatátvitelhez kódolni kell annak érdekében, hogy 0, 1 sorozatként a számítógép számára értelmezhető legyen. Az átvitel illetve a tárhely szempontjából az a szerencsés, ha ezt az üzenetet minél tömörebben tudjuk leírni, így kisebb tárhely, illetve csatorna kapacitás szükséges az információ tárolásához, illetve továbbításához. Egy szöveges üzenet esetén ez azt jelenti, hogy először a karakterek kódolása történik ASCII, vagy Unicode hozzárendelésével, de ezek fix hosszúságú kódok, tehát minden karaktert ugyanannyi biten írunk le függetlenül attól, hogy milyen gyakran fordulnak elő. Abban az esetben azonban, ha valójában különböző gyakoriságú karakterekről van szó, akkor a kód tartalmaz redundanciát. Ezt kihasználva, lehetőségünk van arra, hogy tömörebben írjuk le az üzenetet. Az alábbiakban ennek folyamatát és eszközeit ismerjük meg.

A kódolás során először annak az üzenetnek a kiválasztása történik, amit kódolni szeretnénk. Ezt a kiválasztást a *Forrás* végzi, az eredmény maga a szöveg, amit továbbítani szeretnénk, ekkor még kódolatlan formában. Annak érdekében, hogy ez a kiválasztott üzenet továbbítható legyen, bináris formára kell alakítani, vagyis kódolni kell. Ezt a műveletet az *Adó* végzi, melynek feladata az üzenet jellé alakítása. Abban az esetben, ha az üzenetet tömöríteni is tudjuk, a tömörítést is az *Adó* végzi. A kódolt üzenetet a *Csatornán* keresztül továbbítjuk a *Vevő* felé. Ez a csatorna lehet kommunikációs, vagy akár tároló csatorna is. A *Vevő* feladata a csatornán keresztül érkező jel visszaalakítása üzenetté, vagyis ő végzi az üzenet dekódolását. Abban az esetben, ha az üzenetet tömörített formában kaptuk meg, annak kitömörítését is ő végzi. Végezetül a dekódolt üzenet megérkezik a kódolás folyamatának utolsó állomására, a *Rendeltetési helyre*. A kódolás folyamatát a 22. ábra szemlélteti.

22. ábra A kódolás folyamata

Az ábrát szemlélve látható, hogy a vett jelet és a jelet különbözőképpen jelöljük. Ennek az az oka, hogy zajos csatorna esetén az üzenet sérülhet, így nem biztos, hogy a vevőhöz a küldött jel érkezik meg. A célunk az, hogy ilyen esetben az eredeti üzenet visszaállítható legyen, de legalább annak hibás voltát fel tudjuk ismerni. A kódolás során elérendő cél: U=U*.

9.2. A kódolás típusai

A kódolás célja alapján három kódolási módot különböztetünk meg. Ezek a kódolási típusok különböző más-más módszereket használnak, ezért választjuk őket külön, de az üzenet kódolásakor mindháromra szükségünk van.

1. Forráskódolás
2. Csatornakódolás
3. Titkosító kódolás

Forráskódolás

A forráskódolás célja az üzenet méretének csökkentése, ezért ebbe a kódolás típusba sorolhatók a különböző tömörítő eljárások. A módszerek sokfélesége ellenére mindegyikben közös, hogy a fájlméret csökkentését a redundancia kihasználásával érik el. Ebben az esetben a kódolás után kapott jel mérete kisebb lesz, mint az eredeti üzenet, vagyis az ábrán látható jelöléseket használva: A<U.

Csatornakódolás

A csatornakódolás a veszélyes csatornák hatására sérült üzenet visszaállíthatóságát hivatott elősegíteni. Ennek alapfeltétele a redundancia növelése, ami látszólag a forrás kódolás során történő redundancia csökkentéssel ellentétes folyamat, hiszen a kódolás eredményeként az előállított jel mérete nagyobb lesz, mint az eredeti üzenet, vagyis az ábra jelöléseit alkalmazva: A>U. A forrás kódolás esetén azonban az üzenetben természetesen megtalálható redundanciát igyekszünk csökkenteni a tömörebb leírás érdekében, ügyelve arra, hogy az üzenet továbbra is értelmezhető maradjon. Csatornakódolás során pedig egy mesterségesen hozzáadott redundanciát használunk, ami valamilyen algoritmus alapján az üzenetből állítható elő, és kimondottan annak javíthatóságára szolgál. Ezt a kódolást veszélyes csatorna esetén alkalmazzuk, amikor károsodhat az információ.

Titkosító kódolás

A titkosító kódolás célja az információ védelme. Ebben az esetben nem az átvitel során keletkező sérüléstől óvjuk az üzenetünket, hanem egy esetleges harmadik féltől. Azt próbáljuk megakadályozni, hogy egy esetleges bepillantó ember ne tudja értelmezni az üzenetet. Ilyenkor a redundancia változatlan, nincs szerepe a kódolás során, vagyis az eredeti és a kódolt üzenet mérete megegyezik, azaz az ábra jelöléseit alkalmazva: A=U

Fontos megjegyezni, hogy tökéletes biztonság nem létezik, a titkosító kódolásokkal nem tudunk olyan üzenetet előállítani, ami biztosan törhetetlen, de arra kell törekedni, hogy minél nehezebb legyen egy külső személy számára az üzenet értelmezése.

A kódolással szemben támasztott követelmények

1. Az előállított jel egyértelműen dekódolható legyen, vagyis a vevő oldalon ugyanabból a jelből ne lehessen több különböző üzenetet előállítani. Ez mindhárom kódolási típusnál elengedhetetlen alapkövetelmény.

2. Forráskódolás esetén a jel előállításakor a minimális szóhosszra törekszünk, vagyis az üzenet szimbólumaihoz minél rövidebb kódszavakat rendelünk, hiszen itt az a cél, hogy a lehető legtömörebben írjuk le az információt, lehetőleg veszteségmentesen.

3. Csatornakódolásnál az üzenetet olyan módon kell kódolnunk, hogy ezáltal lehetővé tegyük a hibák észlelését és javítását annak érdekében, hogy a sérült üzenetből is előállítható legyen az eredeti.

4. A titkosító kódolás során a megfelelően nehéz dekódolhatóságra törekszünk. Ez látszólag ellentmond az első követelménynek, ami az üzenetek egyértelmű dekódolhatóságát írja elő, de valójában a két követelmény egymás mellett, egyidejűleg alkalmazható. Míg az első követelmény esetén az egyértelmű dekódolhatóság a vevőre vonatkozik, neki kell biztosítani, addig a titkosító kódolás során a nehéz dekódolhatóság a külső, harmadik fél számára kell, hogy érvényesüljön, hiszen azt szeretnénk, hogy számára ne legyen hozzáférhető az üzenet, míg a vevő számára igen. Azt kell mondanunk, hogy sajnos teljes biztonság nincs, de arra törekszünk, hogy minél nehezebb legyen az eredeti információhoz hozzáférni egy külső fél számára.

A tömörítő algoritmusok hatékonysága

Tömörítő algoritmusból sokféle létezik. Ezek a módszerek különböző hatékonysággal tudják csökkenteni a fájl méretét, de ezek hatékonysága nem általánosan ugyanolyan minden információ típusra, hanem az alkalmazott algoritmustól függően a különböző módszerek különböző fájlszerkezetek esetén működnek hatékonyan. A leggyakrabban alkalmazott módszerek: Huffman kód, LZW, aritmetikai kód, PKZIP, ARJ.

A megfelelő módszer kiválasztáskor az elsődleges szempont természetesen az, hogy megengedhető-e a veszteség a tömörítés során, vagy mindenképpen veszteségmentes tömörítésre van szükségünk. Következőként azt kell megfontolnunk, hogy akinek küldjük a kódolt üzenetet, annak rendelkezésére áll-e a visszatömörítő program, hiszen ennek hiányában nem tudja értelmezni az üzenetet. Ezután, a következő kérdés az, hogy mire szeretnénk optimalizálni. Optimalizálhatunk fájlméretre, vagy akár tömörítési

időre is, attól függően, hogy számunkra mi a fontos, mi az elérendő cél. Ennek megfelelően a tömörítés paraméterezhető, például megadható, hogy milyen sűrűségű tömörítést alkalmazzon a program, illetve a tömörítés milyen gyors legyen. A 6. táblázatban a leggyakrabban alkalmazott módszerek hatékonyságát a fájlméret szempontjából hasonlítjuk össze, különböző fájltípusokra megadva a fájl kiinduló méretét és az egyes tömörítő módszerek után kapott méretet bájtban. A felsorolt tömörítő algoritmusok mindegyike veszteségmentes.

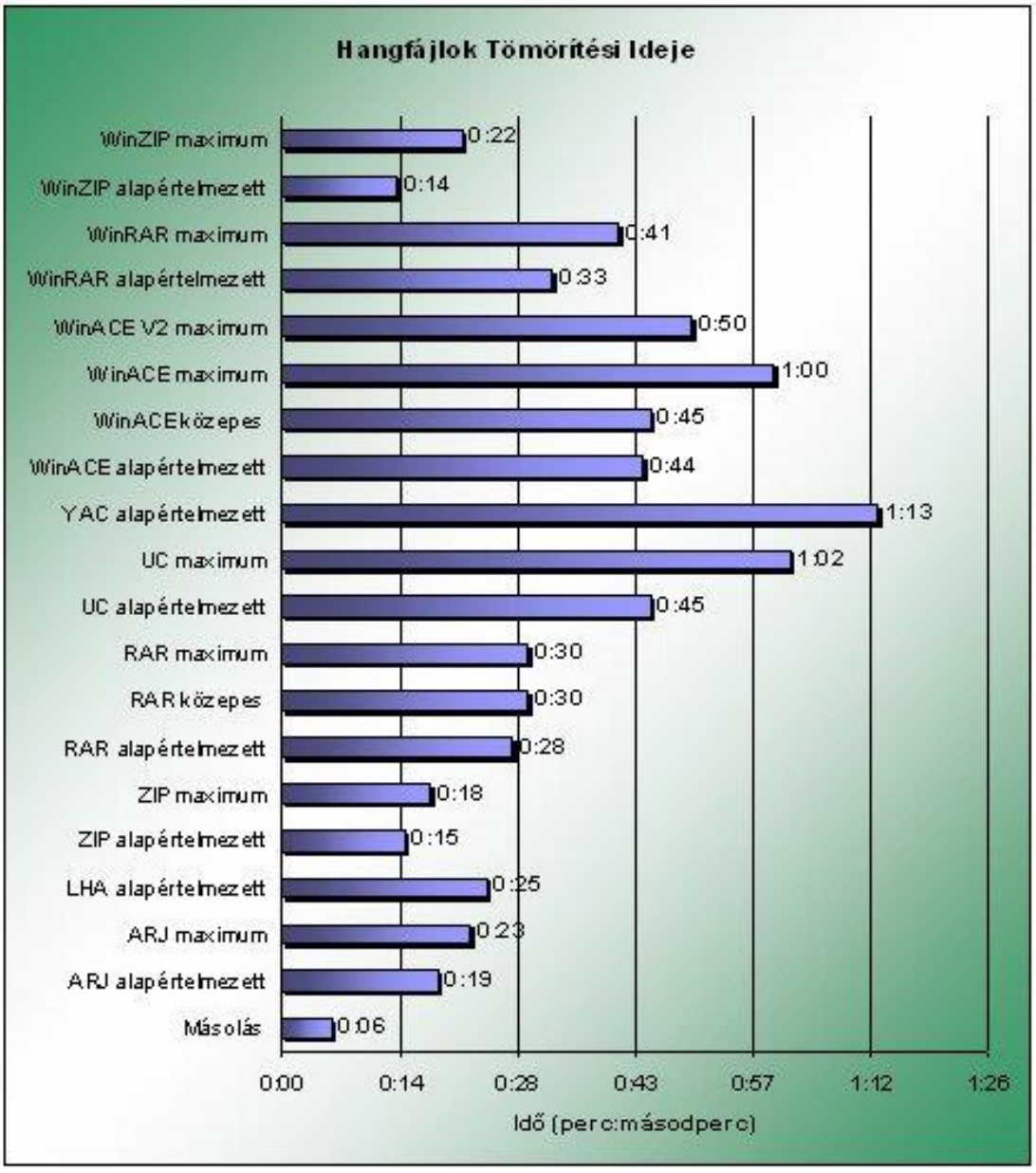
Fájltípus	.exe	.img	.txt
Kiinduló fájlméret	277 766	168 974	151 579
Huffman	103 408	57 383	42 576
LZW	117 811	55 108	48 322
Aritmetikai	177 042	79 870	101 322
PKZIP	96 525	56 380	39 953
ARJ	92 560	50 236	36 913

6. táblázat Néhány tömörítő módszer eredményeinek vizsgálata

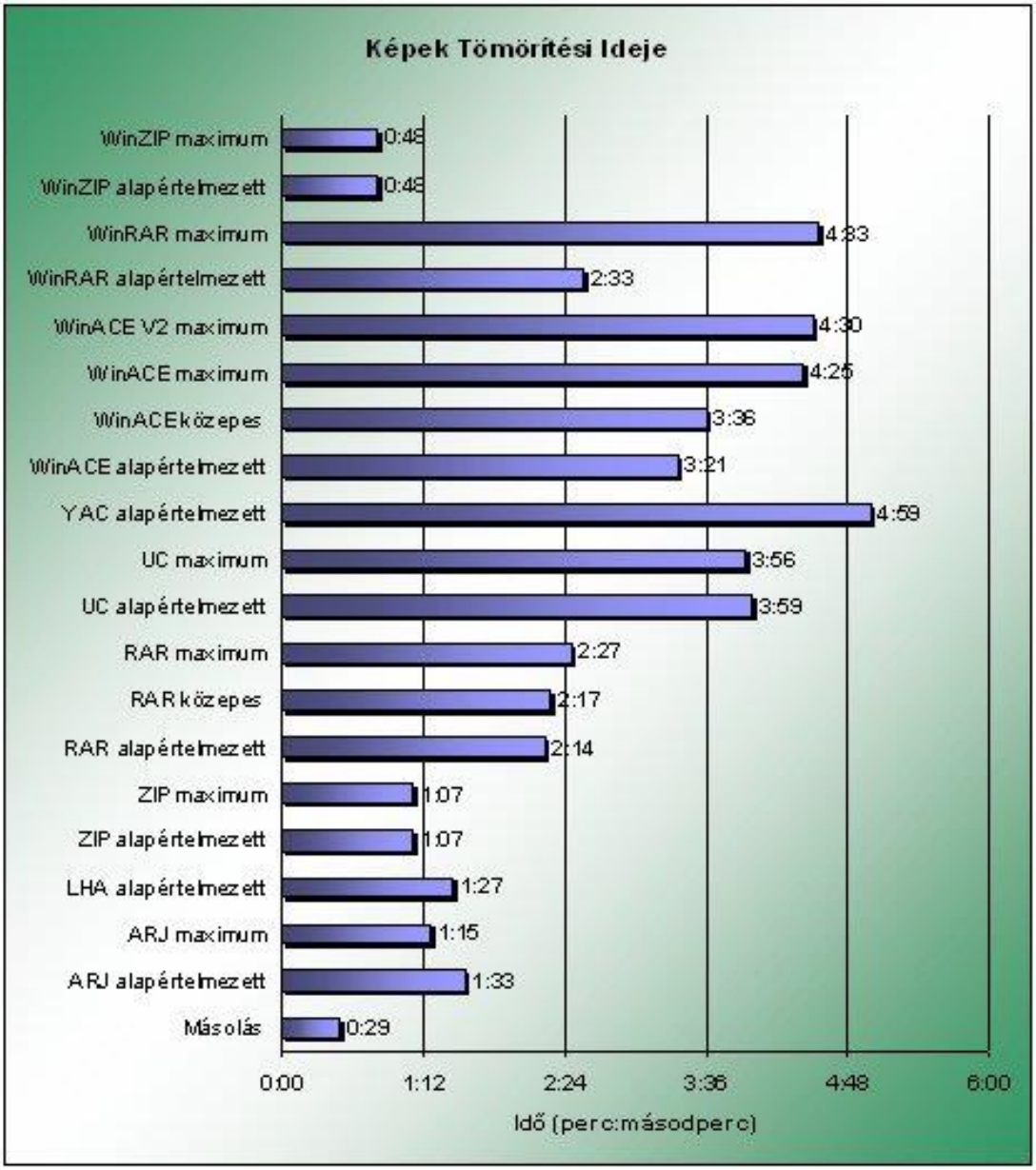
Veszteséges vagy veszteségmentes tömörítés

A forrás kódolás esetén eddig csak azt az esetet említettük, amikor az üzenetben rejlő redundanciát kihasználva tömörítjük az üzenetünket olyan módon, hogy a kódolt üzenet értelmezhetőségében ez ne okozzon problémát. Arról azonban eddig nem esett szó, hogy az értelmezhetőség alatt pontosan mit értünk. A tömörítés eredménye abban az esetben is értelmezhető maradhat, ha abból valamit elhagyunk, vagyis veszteséges tömörítést alkalmazunk. Azonban az hogy a tömörítés során megengedünk-e veszteséget vagy nem, szintén a fájl szerkezetétől függ, hiszen a szerkezet határozza meg az alkalmazható algoritmusokat. Egy program tömörítésekor például fontos a veszteségmentesség, hiszen minden bitnek szerepe van. Zene, illetve kép fájlok tömörítése esetén azonban az emberi észlelés korlátai miatt megengedhető hogy a tömörítés veszteséges legyen. Hang fájlok esetén a számunkra nem hallható tartomány elhagyása ugyan veszteséget okoz az információban, problémát mégsem okoz. Képfájlok esetén ugyanez igaz a felbonthatóságra. Az agyunk

számára egy bizonyos felbontáson túl már nem ad plusz információ a kép, ezért ilyen finomságú felbontást már nem érdemes alkalmazni a fájl méretének rovására. A fentiekből következően kép-, és hangfájlok esetén mindig megállapodás kérdése, hogy mit tekintünk zajnak, és mit tekintünk információnak és ezt figyelembe kell venni a tömörítés során. Képek esetén a nagy összefüggő területeket tudjuk rövidebben kódolni, hang esetén pedig a számunkra nem hallható tartományt hagyhatjuk el. Az érzékelés kép és hang esetén is átsimítja a veszteségeket. Természetesen ehhez figyelembe kell vennünk, hogy mi az, ami még elhagyható az információ károsodása nélkül. Ezzel sokat nyerünk, mert a fájlméret viszont töredéke lesz az eredetinek.



23. ábra Hangfájlok tömörítési ideje különböző módszerek alkalmazásával [22]



24. ábra Képek tömörítési ideje különböző módszerek alkalmazásával [22]

A 14, 15. ábrákon a hang-, illetve a képfájlok tömörítési idejét vizsgáljuk a különböző módszereket összehasonlítva.

9.3. A kódolás alapvető eszközei

Amint azt a fentiekben láttuk, forráskódoláskor arra törekszünk, hogy az információt minél tömörebben írjuk le úgy, hogy az üzenet továbbra is értelmezhető maradjon. Arra keressük a választ, hogy ez hogyan, milyen

eszközök segítségével oldható meg. Lehet-e hatékonyabban kódolni, mint a fix hosszúságú kódok esetén (pl. ASCII, Unicode), ahol minden egyes szimbólumhoz ugyanolyan hosszúságú kódszavakat rendelünk, miközben az egyértelmű dekódolhatóság követelményének is eleget teszünk. A következőkben azokat az alapvető eszközöket fogjuk megismerni, amelyek ezt lehetővé teszik.

A kódolás alapvető eszközei:

- kódfa
- prefix kód

Gráf

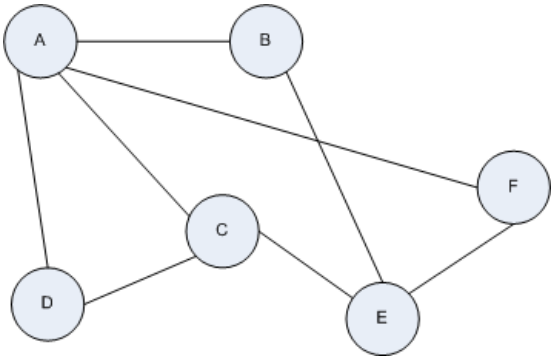
Ahhoz, hogy a kódfát definiálni tudjuk, előbb a gráf fogalmával kell tisztában lennünk.

A gráf

- pontok valamint rajtuk értelmezett összeköttetések (élek) halmaza, vagyis az a kérdés, hogy a pontok hogyan vannak összekötve, össze vannak-e kötve
- lehet irányított vagy irányítatlan, attól függően, hogy két pont között csak meghatározott irányban haladhatunk-e. Irányított gráfok esetén a definiált irányt az él végére rajzolt nyíllal jelöljük

Irányítatlan gráf

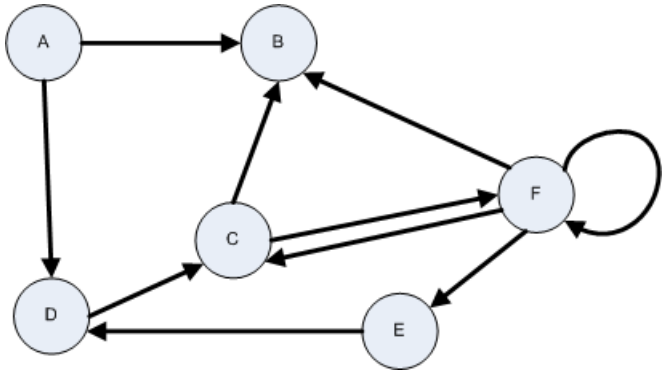
- Az élek egyszerű összekötő vonalak
- A csúcsokat maximum egy él köti össze
- Nem teszünk különbséget A-ból B-be és B-ből A-ba menő élek között



25. ábra Példa irányítatlan gráfra

Irányított gráf

- Az élek nem egyszerű összekötő vonalak, hanem nyilak hangsúlyozzák az élek irányultságát
- A csúcsokat maximum két él köti össze, a különböző irányok jelölésére különböző éleket alkalmazunk
- C-ből F-be és F-ből C-be menő éleknek különböző, irányított élek felelnek meg
- Tipikus alkalmazási példa az útvonal hálózatok leírása



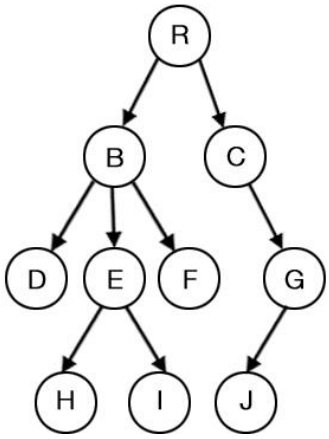
26. ábra Példa irányított gráfra

Fa

A gráf egy komplex alakzat, ebből következően a feldolgozó algoritmusok is komplexek. Természetesen vannak olyan feladatok, amelyek máshogy nem kezelhetők, de kódolás esetén helyette a jóval egyszerűbb szerkezetű fát használhatjuk.

Fa jellemzői

- Speciális jellemzőkkel rendelkező irányított gráf. (Az irányítottság informatikai feladatokban mindig jelen van.)
- Egyszerűsített gráf olyan értelemben, hogy a csomópontok között maximum egy él van
- Összefüggő gráf, vagyis bármely két csúcs között van út
- Körmentes gráf, vagyis egyetlen csúcsához sincs belőle kiinduló és ugyanott végződő irányított út



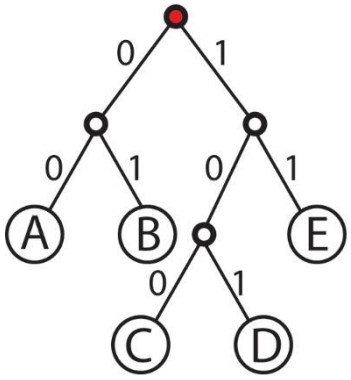
27. ábra Példa fa szerkezetre

Kódfa

A kódfa a fa egy speciális esete, ahol a fent felsorolt jellemzőkön kívül a fának további kritériumoknak kell eleget tennie. Ebben az esetben fontos, hogy minden csomópontból maximum két él induljon ki, vagyis a fa bináris legyen és felépítése a 19. ábrán szemléltetettnek megfelelő.

A kódfa elemei

- Gyökérpont: olyan pont amiből csak kifelé mennek élek, az ábrán a pirossal jelölt pont
- Levélpont: olyan pont, amibe csak befelé megy él. A levélpont jelöli a kódelemeket, vagyis a kódolandó szimbólumkat. Ebből következően a kódfának annyi levélpontja van, ahány kódolandó szimbólum (az ábrán a kódolandó szimbólumok: A, B, C, D, E, a levélpontok száma: 5)
- Csomópont: olyan pontok, amelyekbe be- és kifelé is mennek élek
- A kódjeleket, amelyekből előállítjuk majd a kódot (0,1) az ágakhoz rendeljük hozzá.



28. ábra Egy lehetséges kódfa

A kód leolvasása

A kód leolvasása definíció szerint mindig a gyökértől a levél felé történik, ezért hagyhatók el a nyilak az ágakról annak ellenére, hogy irányított gráfról van szó. A 19. ábrán látható kódfa esetén a kódolandó szimbólumokhoz a következő kódszavakat rendelhetjük:

A: 00

B: 01

C: 100

D: 101

E: 11

Vegyük észre, hogy a kapott kódszavak hossza különböző, ilyen esetben változó hosszúságú kódról beszélünk.

A kódfához kapcsolódó definíciók

A kódfa által előállított kódszavak jellemzése érdekében definiálnunk kell az őket leíró fogalmakat.

Ágszám *L(xᵢ)*: azt adja meg, hogy hány ágon kell keresztülmenni, hogy eljussunk a gyökértől a szimbólumig. Az előző példában A, B, E esetén: 2; C, D esetén: 3

Abban az esetben, ha szimbólumokhoz előfordulási valószínűségeket is rendelünk, a teljes kódtáblára megadható az **átlagos kódhossz**, ami a következőképpen számítható:

$$L_{átl} = \sum_{i=1}^{n} L(x_i)p(x_i) \qquad (30)$$

ahol $p(x_i)$ az i-edik szimbólum valószínűsége, $L(x_i)$ pedig az i-edik szimbólum ágszáma

Az átlagos kódhossz egy olyan mérőszám, ami megmutatja, mennyire jól, mennyire hatékonyan kódolunk. Minél kisebb a kapott érték, annál hatékonyabb a kódolás. Következésképpen az átlagos kódhossz értékét minimalizálni kell. A minimalizálás úgy lehetséges, hogy a gyakrabban előforduló szimbólumokhoz rövidebb, míg a ritkábban előforduló szimbólumokhoz hosszabb kódszót rendelünk, így változó hosszúságú kódot állítunk elő. Az átlagos kódhosszt azonban nem lehet a végtelenségig csökkenteni, van egy alsó határ arra vonatkozóan, hogy mi a legkisebb elérhető átlagos kódhossz. Ennél jobb érték nem érhető el alternatív keresési stratégiák esetén sem. Az átlagos kódhossz alsó határa a következő képlettel számolható:

$$L_{átl} \geq \frac{H(x)}{log_2(s)} \qquad (31)$$

ahol *H(x):* a rendszer entrópiája, *s:* az egy pontból kiinduló maximális élek száma (bináris esetben 2), $log_2 s$: egy lépés maximális információ tartalma (bináris esetben 1)

Prefix kód

Míg fix hosszúságú kódok esetén a kódolással szemben támasztott követelmények első kritériumát, vagyis az egyértelmű dekódolhatóságot biztosan teljesíteni tudjuk, addig változó hosszúságú kód esetén külön figyelmet kell

fordítanunk erre a kritériumra. Ebben az esetben ugyanis fennáll a veszélye annak, hogy ha nem megfelelően állítjuk elő a kódszavakat, akkor az üzenet nem lesz egyértelműen dekódolható. A perfix kód egy eszköz ahhoz, hogy ez a helyzet ne állhasson elő, hiszen a tömörítés nem történhet az értelmezhetőség rovására. A következőkben azokat a követelményeket nézzük meg, amelyek ahhoz szükségesek, hogy egy kód prefix legyen.

Az alkalmazott jelölések:

y: kódjelek halmaza, pl.: {0, 1}

Y: *y*-ból alkotott kódszavak halmaza, pl.: {0, 1, 01, 10, 00, 11, 100, 110 …}

Prefix kód – előzetes definíciók

Kódszavak egyenlőek (Y elemei): két kódszó egyenlő, ha azok egyforma hosszúak és minden pozícióban megegyeznek. Például: 00=00, 01=01, de 00≠01 mivel a hosszuk egyenlő, de a 2. pozíción különböznek, 000≠00 mivel már a hosszuk sem egyenlő.

u kódszó folytatása v-nek: ha *u, v ϵ Y* kódszavak és *u*=*v*, vagy ha *u* megkapható *v*-ből plusz karakterek hozzáfűzésével. A karaktereket csak *v* kódszó végéhez fűzhetjük hozzá. Például: ha v=01 és u=010, akkor *u* folytatása *v*-nek, mivel a *v*-hez egy karaktert hozzáfűzve megkapjuk *u*-t

Kód: $g: X \to Y$ leképezés, ahol $x \in X$ szimbólum, $y \in Y$ kódszó, tehát a szimbólumokhoz rendelünk kódszavakat

Prefix kód: $g: X \to Y$ leképezés, ahol $x \in X$ szimbólum, $y \in Y$ kódszó, ha a kódszavak mind különbözőek és egyik kódszó sem folytatása a másiknak.

Például.:

a=1

b=01

c=00

Kijelenthető, hogy a kód prefix, mert teljesül rá a definíció, hiszen a kódszavak mind különbözőek és egyik kódszó sem folytatása a másiknak.

Például:

a=1

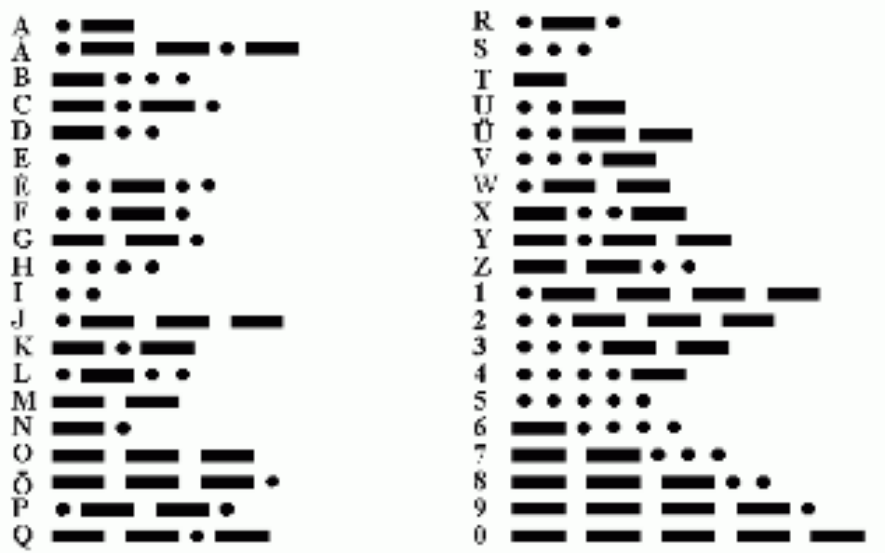
b=01

c=10

Kijelenthető, hogy a kód nem prefix, mert igaz, hogy a kódszavak mind különbözőek, de a c-hez tartozó kódszó folytatása az a-hoz tartozó kódszónak.

A prefix kód jelentősége

A prefix kód esetén nem kell elválasztó karaktereket használni annak jelölésére, hogy az üzenetben hol ér véget az egyik szimbólum kódja és hol kezdődik a másik. A prefix kód enélkül is teljesíti a kódolással szemben támasztott első kritériumot, vagyis mindig egyértelműen dekódolható. Olyan változó hosszúságú kódok esetén, amelyek nem prefixek, mint például a Morse kód mindig kell szünetjelet használni, különben az üzenet dekódolása nem egyértelmű, hiszen nem fogjuk tudni, hogy hol vannak a szimbólumok határai az üzenetben.



29. ábra Morse kódtábla [23]

Morse kód esetén „A” folytatásai: Á, J, L, P, R, W, 1

„D” folytatásai: B, X, 6

stb.

Például a következő üzenet esetén: • • • – – – • • •

A lehetséges megfejtések: SOS, EÉE, V7, IJS, IETGI

Prefix kóddal kapcsolatos állítások

- A prefix kód mindig egyértelműen dekódolható.
- Ha egy kód nem prefix, akkor valamilyen elválasztó karakterrel prefixszé tehető.
- Ha egy kód kódhossza állandó, akkor biztosan prefix. Ez a kódszavak egyenlőségére vonatkozó definícióból következik. Fix kódhossznál alapfeltétel, hogy egyforma hosszúak legyenek a kódszavak, viszont két különböző szimbólumhoz nem rendelhetjük ugyanazt a kódszót, ezért nem egyezhetnek meg minden pozícióban, vagyis minden kódszó különböző.

A prefix kódhoz kapcsolódó definíciók

Átlagos kódhossz:

$$L_{átl} = \sum_{x \in X} \|g(x)\| p(x) \tag{32}$$

ahol $g: X \to Y$ leképezés, $x \in X$ szimbólum, $y \in Y$ kódszó, $\|g(x)\|$ az előállított kódszó hossza, $p(x)$ az x szimbólum előfordulási valószínűsége

A prefix kód átlagos kódhossza ekvivalens a kódfa átlagos kódhosszával, hiszen

$$(x_i) = \|g(x)\| \tag{33}$$

Az átlagos kódhossz alsó korlátja:

$$L_{átl} \leq \frac{H(x)}{log_2 s} \tag{34}$$

A kódfa és a prefix kód kapcsolatáról elmondható, hogy egy kód akkor és csak akkor prefix, ha rajzolható hozzá kódfa.

A változó hosszúságú kód jelentősége

A változó hosszúságú kód alkalmazásának az az oka, hogy szeretnénk minél kisebb átlagos kódhosszt elérni. Ennek alapjául szolgál a szimbólumok különböző előfordulási gyakorisága szolgál (pl. betűgyakoriság). Ekkor a

gyakrabban előforduló szimbólumokhoz rövidebb, míg a ritkábban előforduló szimbólumokhoz hosszabb kódszót rendelünk. Az átlagos kódhossz definíciója alapján belátható, hogy így a teljes üzenet kódhossza rövidebb lesz, mint abban az esetben, ha fix kódhosszt használunk. Ennek az az oka, hogy a rövidebb kódhosszú szimbólumok fordulnak elő gyakrabban, vagyis a súlyozott összegben ezek szerepelnek többször, míg a hosszabb kódhosszú szimbólumok ritkábban, ezért ők a súlyozott összegben is kevesebbszer szerepelnek.

10. fejezet

STATISZTIKA ALAPÚ ADATTÖMÖRÍTÉS

Ebben a fejezetben a veszteségmentes tömörítő algoritmusok egyik nagy csoportjával foglalkozunk a statisztika alapú tömörítő algoritmusokkal. Ennek a kódolástípusnak az alapja az, hogy az üzenetben előforduló szimbólumok gyakorisága alapján változó hosszúságú kódot rendelünk az egyes szimbólumokhoz, ezzel csökkentve az átlagos kódhosszt. Definiáljuk az optimális kód kritériumait és bemutatásra kerül egy optimális kód, amelynek esetén bizonyítjuk is, hogy teljesíti az optimális kód kritériumait.

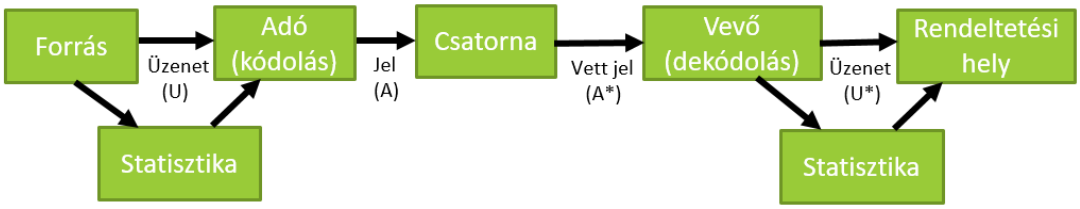
10.1. A statisztika alapú adattömörítés folyamata

A statisztika alapú adattömörítés a forrás kódolás egy lehetséges változata, vagyis az a célunk, hogy az információt rövidebben tudjuk leírni.

Alapötlet: az egyes szimbólumok gyakorisága az üzenetben különböző, ezt kihasználva a ritkábban előforduló szimbólumokhoz hosszabb, míg a gyakrabban előfordulókhoz rövidebb kódot rendelünk. A kódolás folyamata a 30. ábrán látható módon a korábban látott elemekhez képest a statisztikával bővül. Ezt a statisztikát az aktuális üzenet alapján állítjuk elő és az üzenettel együtt kapja meg az adó, hogy el tudja végezni a kódolást. Itt az adó feladata már nem csak az üzenet jellé alakítása, hanem a statisztika alapján, annak tömörebb formában történő leírása is. Mivel a statisztika nem általános érvényű, hanem az aktuális üzenet alapján állítható elő, ezért a jellel együtt továbbítani kell, hogy a vevő képes legyen dekódolni a kapott jelet.

Előnye: biztosítja a minimális átlagos kódhosszt

Hátránya: a statisztikát is továbbítani kell az üzenettel együtt, hogy a vevő értelmezni tudja az üzenetet, de ez növeli a továbbítandó adatmennyiséget.



30. ábra A statisztikára épülő adattömörítés folyamata

A következőkben néhány olyan kódolást fogunk megismerni, amelyek a szimbólumok gyakoriságát kihasználva, tehát a statisztika alapján rendelnek kódszavakat az egyes szimbólumokhoz.

10.2. Shannon-Fano kód

Az algoritmus abból indul ki, hogy összefüggő események esetén a valószínűségek összege 1. Ennek megfelelően a konkrét üzenetre meghatározott statisztika, vagyis az egyes szimbólumok előfordulási valószínűségei segítségével a [0,1] intervallum felosztását végzi el, ez az algoritmus kiinduló lépése. Az algoritmus pontos megismerése előtt rögzítsük az alkalmazott jelöléseket.

$X = \{x_1, \ \ldots, x_n\}$ a kódolandó szimbólumok halmaza

$P = \{p_1, \ \ldots, p_n\}$ a szimbólumokhoz rendelt valószínűségek halmaza

$Y = \{y_1, \ \ldots, y_n\}$ az egyes szimbólumokhoz rendelt kódszavak halmaza

A Shannon-Fano kód előállításának lépései

1. A szimbólumok előfordulási valószínűségének meghatározása.
2. A szimbólumok valószínűség szerinti csökkenő sorrendbe rendezése.
3. Az elemek hozzárendelése a [0, 1] intervallumhoz valószínűség alapján.
4. Az intervallum felosztása két egyenlő részre, amíg az intervallumban egynél több szimbólum van. Abban az esetben, ha a szimbólum éppen az intervallumok határán helyezkedik el, a jobboldali intervallumba tartozónak vesszük.

5. Kódfa generálás

6. Kódjelek hozzárendelése az ágakhoz.

7. Kódszavak leolvasása a gyökérponttól a levél felé.

Shannon-Fano kód példa

1. Szimbólumok előfordulási valószínűségének meghatározása.

$$p(x_1) = 0{,}1$$

$$p(x_2) = 0{,}5$$

$$p(x_3) = 0{,}15$$

$$p(x_4) = 0{,}2$$

$$p(x_5) = 0{,}05$$

2. Szimbólumok valószínűség szerinti csökkenő sorrendbe rendezése:

$$p(x_2) = 0{,}5$$

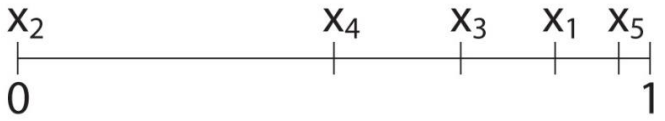
$$p(x_4) = 0{,}2$$

$$p(x_3) = 0{,}15$$

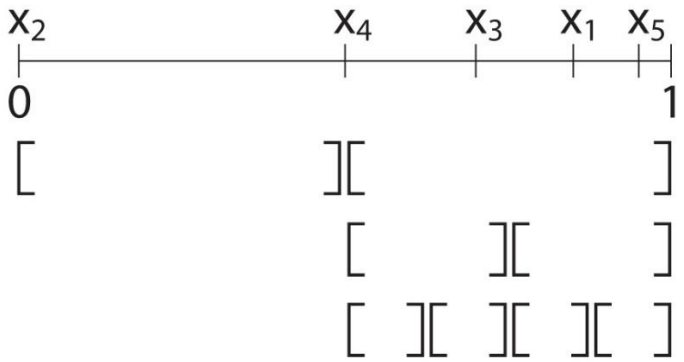
$$p(x_1) = 0{,}1$$

$$p(x_5) = 0{,}05$$

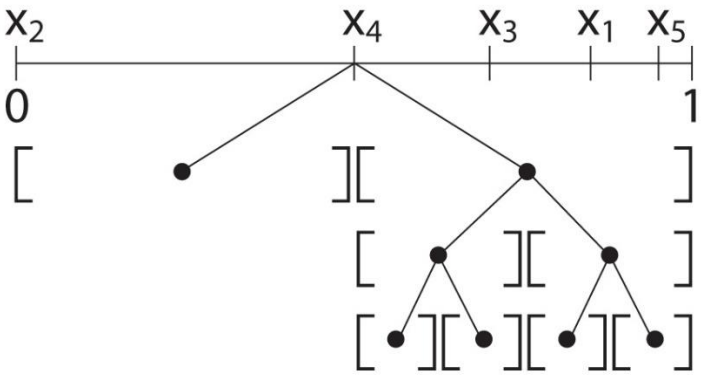
3. Elemek hozzárendelése a [0,1] intervallumhoz a valószínűségek alapján. Az intervallum felosztása a csökkenő sorrendbe rendezett valószínűségek alapján történik. A 0 ponttól először a $p(x_2)$ valószínűséget mérjük fel a következő szimbólum helyének meghatározása érdekében, a szimbólumokat pedig mindig a hozzájuk tartozó intervallum baloldalára írjuk, ebben az esetben a $p(x_2)$ intervallum baloldalához, vagyis a 0 ponthoz az x_2 szimbólumot rendeljük. Ezután sorban a többi valószínűséget is rámérjük a [0,1] intervallumra az előző osztástól indulva, vagyis az alábbi ábrán $d(x_2, x_4) = p(x_2)$, $d(x_4, x_3) = p(x_4)$, $d(x_3, x_1) = p(x_3)$, $d(x_1, x_5) = p(x_1)$, $d(x_5, 1) = p(x_5)$, ahol $d(x_i, x_j)$ az x_i, x_j pontok távolsága.



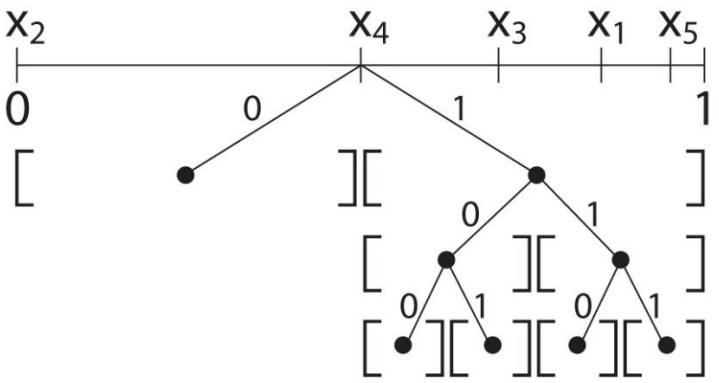
4. A [0,1] intervallum felosztása két egyenlő részre, majd az egyes lépésekben kapott intervallumokkal is ugyanígy járunk el, amíg az intervallumon belül egynél több szimbólum található. Az x_4 szimbólum két intervallum határán van, így a definíció szerint a jobboldali intervallumhoz tartozónak kell venni.



5. Kódfa generálás az intervallumok közepét összekötve. A kódfa gyökérpontja a [0,1] intervallum közepe, a levélpontok pedig a legalsó szinten lévő intervallumok középpontjai.



6. A kódjelek hozzárendelése az ágakhoz.



7. Kódszavak leolvasása a gyökértől a levélig. A szimbólumok mindig ahhoz a levélponthoz tartoznak, amelyik intervallumon belülre esnek.

$$x_1 = 110$$

$$x_2 = 0$$

$$x_3 = 101$$

$$x_4 = 100$$

$$x_5 = 111$$

A Shannon-Fano kód tulajdonságai

- Prefix: ezt a kódfa és a prefix kód kapcsolatára megfogalmazott definíció alapján jelenthetjük ki. Az algoritmus egy kódfát állít elő és a definícióból tudjuk, hogy ha tudunk kódfát rajzolni, akkor a kód biztosan prefix.
- Figyelembe veszi a valószínűségeket, hiszen a második lépésben a valószínűségek alapján rendezzük sorba a szimbólumokat. Ennek következménye, hogy a gyakoribb szimbólumokhoz rövidebb, míg a ritkábbakhoz hosszabb kódot rendelünk. Ez könnyen belátható, mivel a gyakoribb elemhez hosszabb intervallum tartozik, így kevesebbszer kell felosztani az intervallumot ahhoz, hogy csak egy szimbólum legyen az intervallumban, vagyis az adott szimbólumnál kevésbé lesz mély a kódfa.
- Az átlagos kódhossz felső korlátja:

$$\frac{H(x)}{log_2 s} + 1 \qquad (35)$$

ahol *H(x)* a rendszer entrópiája, *s* az egy pontból kiinduló maximális ágszám.
A felső korlát azt jelenti, hogy az átlagos kódhossz biztosan nem lehet ennél nagyobb.

- Jól közelíti a prefix kódra megadott alsó korlátot: láttuk, hogy a felső korlátja, vagyis a legrosszabb esetre megadott átlagos kódhossz a prefix kódra általánosan megadott alsó korlátot jól közelíti. A Shannon-Fano kód átlagos kódhossza az általános alsó korlát és a (35)-ben definiált felső korlát között van.

$$\frac{H(x)}{log_2 s} \leq L \leq \frac{H(x)}{log_2 s} + 1 \qquad (36)$$

10.3. Gilbert-Moore kód

A Gilbert-Moore kód a Shannon-Fano kód egy speciális változata. A két algoritmus szinte teljes mértékben megegyezik azzal a különbséggel, hogy a szimbólumok valószínűség szerinti rendezésének lépése a Gilbert-Moore kód esetén kimarad.

A Gilbert-Moore kód előállításának lépései

1. A szimbólumok előfordulási valószínűségének meghatározása.
2. Az elemek hozzárendelése a [0, 1] intervallumhoz valószínűség alapján.
3. Az intervallum felosztása két egyenlő részre, amíg az intervallumban egynél több szimbólum van. Abban az esetben, ha a szimbólum éppen az intervallumok határán helyezkedik el, a jobboldali intervallumba tartozónak vesszük.
4. Kódfa generálás
5. Kódjelek hozzárendelése az ágakhoz.
6. Kódszavak leolvasása a gyökérponttól a levél felé.

Gilbert-Moore kód példa

A következőkben nézzük meg ugyanazt a példát, mint a Shannon-Fano kód esetén.

$$p(x_1) = 0{,}1$$

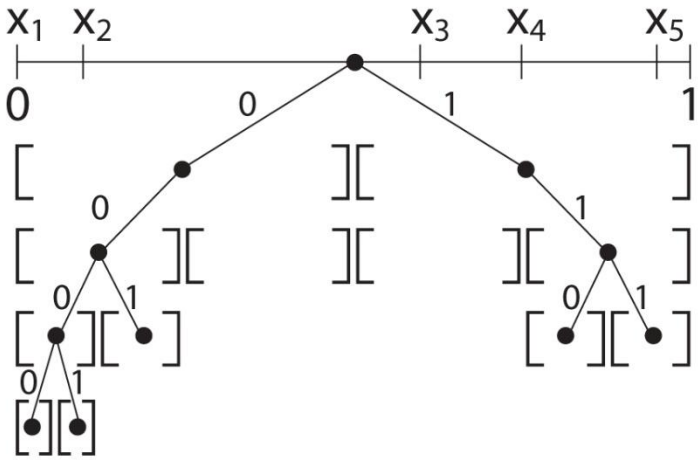
$$p(x_2) = 0{,}5$$

$$p(x_3) = 0{,}15$$

$$p(x_4) = 0{,}2$$

$$p(x_5) = 0{,}05$$

Ebben az esetben a valószínűségeket ugyanebben a sorrendben mérjük fel a [0,1] intervallumra, hiszen a rendezés lépése kimarad. Majd ugyanúgy, ahogy a Shannon-Fano kód esetén, az intervallumok két egyenlő részre osztása következik addig, amíg végül minden intervallumban maximum egy szimbólum szerepel. Ezt követően megrajzoljuk a kódfát, melyet az alábbi ábrán látható módon kapunk meg.



31. ábra A feladat megoldása Gilbert-Moore kód esetén

Az egyes szimbólumok kódjai a leolvasás után a következőképpen állnak elő:

$$x_1 = 0000$$

$$x_2 = 001$$

$$x_3 = 10$$

$$x_4 = 110$$

$$x_5 = 111$$

A Gilbert-Moore kód tulajdonságai

- Prefix, mivel tudunk kódfát rajzolni
- Az átlagos kódhossz felső korlátja:

$$\frac{H(x)+1}{log_2 s} + 1 \qquad (37)$$

  ahol *H(x)* az entrópia, *s* az egy pontból kiinduló maximális ágszám.

- Hátránya: hosszabb kódot állít elő, mint a Shannon-Fano kódolás, abból adódóan, hogy kihagyjuk a rendezés lépését, így nem garantálható, hogy a nagyobb valószínűségű szimbólumoknál kevésbé legyen mély a fa, hiszen az intervallumok méretüket tekintve „össze-vissza" lesznek felmérve. Ez alatt itt azt értjük, hogy nem méret szerint rendezett sorrendben.
- Előnye: az elemeket nem rendezzük át, ezért az egymásmellettiséget megtartja, ami bizonyos esetekben elengedhetetlen (pl.: áramkörök, keresési feladatok esetén)
- Jól közelíti a prefix kódra megadott alsó korlátot: értéke a prefix kódra általánosan megadott alsó korlát és a Gilbert-Moore kódra megadott felső korlát között van.

$$\frac{H(x)}{log_2 s} \leq L \leq \frac{H(x)+1}{log_2 s} + 1 \qquad (38)$$

A Shannon-Fano és a Gilbert-Moore kód összehasonlítása

A következőkben a fenti példában látott Shannon-Fano és Gilbert-Moore kódolás során kapott kódszavakat hasonlítjuk össze. A kódolandó szimbólumokhoz tartozó valószínűségek a következők:

$$p(x_1) = 0{,}1$$

$$p(x_2) = 0{,}5$$

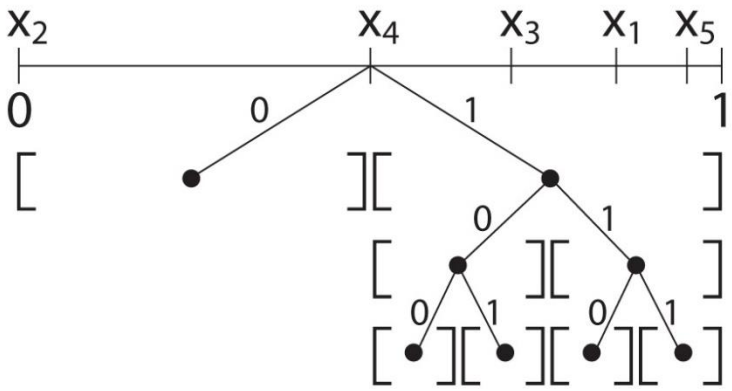
$$p(x_3) = 0{,}15$$

$$p(x_4) = 0{,}2$$

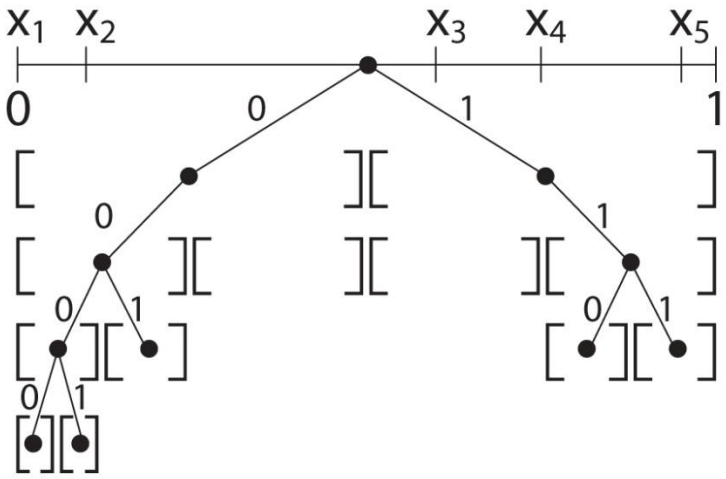
$$p(x_5) = 0{,}05$$

A már ismert módon elvégezve a kódolást, a Shannon-Fano kód esetén a 32. ábrán látható kódfa áll elő, míg a Gilbert-Moore kód esetén a 33. ábrán látható eredményt kapjuk.

Ahogy az a Shannon-Fano és a Gilbert-Moore kód tulajdonságaiból is következik, az átlagos kódhossz felső határa a Gilbert-Moore kód esetén valamelyest rosszabb. Ez abból a különbségből adódik, hogy Gilbert-Moore kód algoritmusából a rendezés lépése kimarad, így az intervallumok méretüket tekintve „össze-vissza” lesznek felmérve, ezért nem garantálható, hogy nagyobb valószínűségű szimbólumoknál kevésbé legyen mély a fa. Shannon-Fano kód esetén a rövidebb kód éppen a rendezés által garantálható, hiszen ha nagyobb a valószínűség, akkor ritkábban helyezkednek el a szimbólumok az intervallumon belül, és kevesebb felosztásra van szükség. Ez a különbség az alábbi ábrákon is jól látható.



32. ábra A Shannon-Fano kód eredménye



33. ábra A Gilbert-Moore kód eredménye

Az ábrákról leolvasva a különböző kódolások eredményeként kapott kódszavakat az alábbiakban láthatjuk. Ezeket összehasonlítva is megállapítható, hogy a Shannon-Fano kód esetén a legnagyobb valószínűséggel előforduló x_2 szimbólum kódszava a legrövidebb, míg a többi szimbólum kódszava ennél hosszabb. Ugyanez az összefüggés a Gilbert-Moore kód esetén egyáltalán nem érvényes, hiszen ebben az esetben az x_3 szimbólum kódszava a legrövidebb, melynek valószínűsége éppen a középső a sorrendben. Ugyanakkor megfigyelhető, hogy a nála kisebb, illetve nagyobb valószínűségű szimbólumokhoz is hosszabb kódszó tartozik.

Shannon-Fano kód	Gilbert-Moore kód
$x_1 = 110$	$x_1 = 0000$
$x_2 = 0$	$x_2 = 001$
$x_3 = 101$	$x_3 = 10$
$x_4 = 100$	$x_4 = 110$
$x_5 = 111$	$x_5 = 111$

10.4. Az optimális kód

Tudjuk, hogy az átlagos kódhossz prefix kódra meghatározott alsó határa az az érték, ami a lehető legjobb kódolást jelenti, hiszen ennél rövidebben nem tudjuk leírni az információt. Az előzőekben látott két kódolás esetén nem sikerült elérni ezt a határt. A következőkben azt fogjuk megnézni, hogy milyen kritériumokat kell teljesítenie egy kódnak ahhoz, hogy ilyen átlagos kódhosszt sikerüljön produkálni.

Cél: a kód átlagos kódhossza egyezzen meg a prefix kódra megadott átlagos kódhossz alsó határával, vagyis $L_{átl} = \frac{H(x)}{log_2 s}$.

Az optimális kód kritériumai:

1. Legyen

   $X = \{x_1, \ \dots, \ x_n\}$ a kódolandó szimbólumokat tartalmazó halmaz,

   $P = \{p_1, \ \dots, \ p_n\}$ a szimbólumokhoz rendelt valószínűségek,

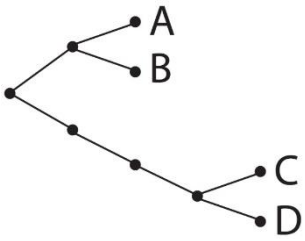
   $L(x_1), L(x_2), \ \dots, L(x_n)$ az egyes szimbólumokhoz tartozó kódhossz.

   Ekkor teljesül a következő állítás:

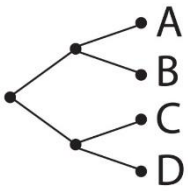
   $$\text{ha } p_1 \geq \cdots \geq p_n \text{ akkor } L(x_1) \ \leq \cdots \leq L(x_n)$$

   vagyis a nagyobb valószínűségű elemekhez rövidebb kódhossz tartozik, a kisebb valószínűségűekhez pedig hosszabb.

2. A kód alkosson teljes kódfát, vagyis minden csomópontból pontosan *s* él induljon ki (bináris esetben s=2). A teljes kódfa kiegyensúlyozott, ami rövidebb kódhosszt eredményez azáltal, hogy a kódfa mélysége minimális. Az alábbi ábrákon jól látható, hogy míg a teljes kódfa esetén minden egyes szimbólumhoz 2 hosszúságú kódszó fog tartozni, mivel mindegyiket 2 ágon keresztük érjük el, addig a nem teljes kódfa esetén a legrövidebb kódszó 2, míg a leghosszabb 4 bitből áll. Az átlagos kódhossz képlete alapján kiszámítható, hogy az átlagos kódhossz a teljes kódfa esetén jobb lesz. Ez annak a következménye, hogy teljes kódfa esetén minden szinten a maximális lehetőségeket használjuk ki.



34. ábra Nem teljes kódfa (kiegyensúlyozatlan)



35. ábra Teljes kódfa (kiegyensúlyozott)

3. Ha a kód optimális, akkor a két legkisebb valószínűségű elem kódhossza megegyezik, vagyis $L(x_{n-1}) = L(x_n)$.

   Ez a kritérium az előzőből is következik, hiszen, ha minden csomópontból pontosan 2 él indul ki, akkor a két legkisebb valószínűségű elem ugyanabból a csomópontból ágazik ki, tehát ugyanannyi ágon keresztül jutunk el hozzájuk.

10.5. Huffman kód

Az algoritmus alapja az üzenetből előállított statisztika. A kódolás az aktuális üzenet betűgyakoriságából indul ki annak érdekében, hogy a gyakrabban előforduló betűkhöz rövidebb, a ritkábban előfordulókhoz hosszabb kódszavakat rendel. Az algoritmus megismerése előtt rögzítsük az alkalmazott jelöléseket.

$X = \{x_1, \ \dots, \ x_n\}$ kódolandó szimbólumok halmaza

$P = \{p_1, \ \dots, \ p_n\}$ a szimbólumokhoz rendelt valószínűségek halmaza

$\mathrm{F} = \{f_1, \ \dots, \ f_n\}$ a szimbólumokhoz rendelt gyakoriságok halmaza

$Y = \{y_1, \ \dots, \ y_n\}$ kódszavak halmaza

P vagy *F* halmaz közül bármelyik tetszőlegesen használható. A példákban, levezetésekben a gyakorisággal dolgozunk, hiszen az egész számokkal történő műveletvégzés egyszerűbb.

A Huffman kód előállítása

1. A szimbólumok gyakoriságának meghatározása az aktuális szöveg alapján, vagyis megszámoljuk, hogy melyik betű hányszor fordul elő az üzenetben.
2. A szimbólumok gyakoriság szerinti csökkenő sorrendbe rendezése. Ez a lépés kihagyható, az algoritmus eredményét nem befolyásolja olyan értelemben, hogy a rendezés lépésének kihagyásakor is teljesülni fog az a követelmény, hogy a gyakoribb elemekhez rövidebb, míg a ritkábbakhoz hosszabb kód tartozik. A különbség annyiban mutatkozik meg, hogy az eltérő sorrend miatt eltérő kódszavakat rendelünk a különböző betűkhöz, de ez az algoritmus hatékonysága szempontjából nem releváns.
3. A két legkisebb gyakoriságú elem összekötése, majd az összekötő ág értékének meghatározása a kiválasztott elemek gyakoriságának összegzése által. Ezt a lépést ismételjük, amíg az összes szimbólum sorra nem kerül, a következő lépésben már az összegzett gyakorisággal dolgozunk tovább.
4. A kódjelek (0,1) ráírása az élekre.
5. Kódszavak leolvasása a gyökértől a levél felé.

Huffman kód példa

Kódolandó szöveg: CSACSKAMACSKA

1. A szimbólumok gyakoriságának meghatározása, vagyis az egyes betűk száma:

$$f(C) = 3$$

$$f(S) = 3$$

$$f(A) = 4$$

$$f(K) = 2$$

$$f(M) = 1$$

2. A szimbólumok gyakoriság szerinti csökkenő sorrendbe rendezése (opcionális):

$$f(A) = 4$$

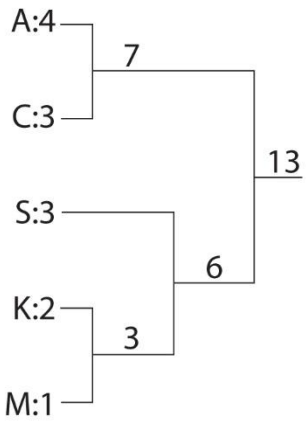
$$f(C) = 3$$

$$f(S) = 3$$

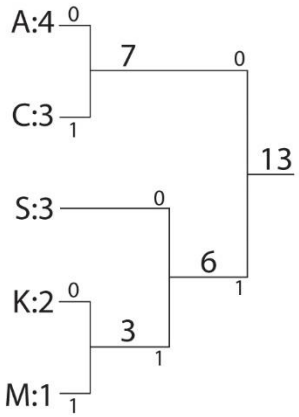
$$f(K) = 2$$

$$f(M) = 1$$

3. A két legkisebb gyakoriságú elem összekötése, amíg az összes szimbólum sorra nem kerül, az összekötő ágakra a kiválasztott elemek összegzett gyakoriságát írjuk.



4. A kódjelek ráírása az élekre.



5. A kódszavak leolvasása a gyökértől a levélig.

A: 00

C: 01

S: 10

K: 110

M: 111

A Huffman kód tulajdonságai

1. Prefix, mivel a kódolás során egy kódfát állítottunk elő és tudjuk, hogy ha rajzolható kódfa, akkor a kód prefix.
2. Átlagos kódhossz:

$$\frac{H(x)}{log_2 s} \tag{39}$$

3. Azonnal kódfát ad. Amikor az elemeket páronként kötjük össze, akkor valójában a kódfa ágait rajzoljuk meg. Ellentétben a Shannon-Fano és a Gilbert-Moore kóddal, ahol a kódfa megrajzolása előtt az intervallumokat egymás után többször fel kell osztani, itt nincs közbülső lépés, egyből rajzoljuk a kódfát.
4. Az egymásmellettiséget megtart(hat)ja. A sorba rendezés lépése nem kötelező, ezért ahogy a Gilbert-Moore kód esetén már említettük, olyan feladatok esetén, ahol fontos lehet a szimbólumok sorrendjének megtartása (pl.: áramkörök, keresési feladatok) szintén alkalmazható.
5. Karakteres kódolás esetén működik a legjobban.
6. Maximális hatásfokú prefix kód, mivel elértük az átlagos kódhossz alsó határát, amiről azt mondtuk, hogy ennél rövidebb kódhossz nem érhető el.
7. Optimális kód (teljesíti a kritériumokat, amit a következőkben bizonyítunk)

A Huffman kód optimális – bizonyítás

1. Állítás:

$$\text{ha } p_1 \geq \cdots \geq p_n \text{ akkor } L(x_1) \leq \cdots \leq L(x_n)$$

vagyis a nagyobb valószínűségű elemekhez rövidebb kódhossz tartozik, kisebb valószínűségűekhez pedig hosszabb

Bizonyítás:

ez a kritérium az algoritmus következményeként teljesül, hiszen mindig a két legkisebb gyakoriságú elemet kötjük össze, a legnagyobbakat hagyjuk a végére. A kódfa ágai pedig az összekötések során állnak elő, így az előbb összekötött ágak esetén nagyobb lesz a fa mélysége, több ágon keresztül érjük el őket, mint például a legnagyobb gyakoriságú szimbólum esetén, amelyet legutoljára kötünk össze.

2. Állítás:

A kódfa teljes kódfát alkot, minden csomópontból pontosan 2 él indul ki.

Bizonyítás:

Ezt a kritériumot szintén az algoritmus garantálja, hiszen a szimbólumokat páronként kötjük össze.

3. Állítás:

A két legkisebb valószínűségű elem kódhossza megegyezik, vagyis $L(x_{n-1}) = L(x_n)$.

Bizonyítás:

Ez a kritérium is teljesül, hiszen a két legkisebb gyakoriságú (valószínűségű) elemet kötjük össze először, ezért ők ugyanazon a szinten lesznek levélpontok.

10.6. Aritmetikai kód

Ez az algoritmus a Huffman kódhoz hasonlóan a karakterek előfordulási gyakoriságán alapul, de ezek feldolgozása más elven történik. A gyakoriság alapján minden szimbólumhoz hozzárendelünk egy tartományt, a gyakoribbhoz kisebbet, a ritkábbhoz nagyobbat. A kódolás során először kijelöljük az első karakterhez tartozó tartományt, majd a következő karakter tartományát már ezen a tartományon belül arányosan határozzuk meg. Végül az utolsó szimbólum így kapott tartományából választunk egy konkrét értéket, ez lesz az üzenet kódja. Az aritmetikai kódolás jelentősége, hogy ezzel a módszerrel 2byte-ból akár egy egész oldalnyi szöveg is visszanyerhető [24].

Aritmetikai kódolás példa

A kódolandó üzenet: ALMA

Az üzenet alapján előállított gyakoriságok a 7. táblázatban láthatók. A következőkben ezek előállítását nézzük meg.

1. Megszámoljuk, hogy melyik karakter hányszor fordul elő a kódolandó üzenetben (Táblázatban az Előfordulás oszlop)
2. Kiszámítjuk az egyes karakterek relatív gyakoriságát úgy, hogy az előfordulások számát elosztjuk a teljes üzenet karakterszámával (pl.: „A” betű esetén 2/4 = 0,5)
3. A tartomány meghatározása úgy történik, hogy a [0,1] intervallumot felosztjuk a relatív gyakoriságoknak megfelelően, vagyis rendre felmérjük a relatív gyakoriságokat, mint intervallum hosszokat 0-tól indulva (hiszen ezek összege 1).

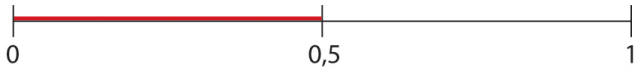
Karakter	Előfordulás	Relatív gyakoriság	Tartomány
A	2	0,5	0 – 0,5
L	1	0,25	0,5 – 0,75
M	1	0,25	0,75 - 1

7. táblázat A kódolandó szöveghez tartozó tartományok

4. A táblázat előállítása után következhet maga a kódolás, az alábbiak szerint:

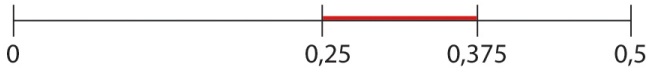
   1. **karakter „A”** : A [0,1] intervallumban keressük az „A” betű 7. táblázatban megadott tartományát arányosan. A kapott tartományt pirossal jelöljük, határait a képlet alapján számoljuk.

      Alsó_határ = 0 + (1-0)*0 = 0
      Felső_határ = 0 + (1-0)*0,5 = 0,5

   2. **karakter „L”** : itt a [0,1] intervallum helyett az előző lépésben kapott [0, 0.5] intervallumban keressük „L” tartományát.

      Alsó_határ = 0 + (0,5-0)*0,5 = 0,25
      Felső_határ = 0 + (0,5-0)*0,75 = 0,375

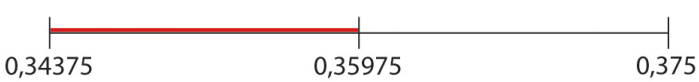
      

   3. **karakter „M”** : az előző lépésben kapott [0.25, 0.375] intervallumban keressük „M” tartományát.

      Alsó_határ = 0,25 + (0,375-0,25)*0,75 = 0,34375
      Felső_határ = 0,25 + (0,375-0,25)*1 = 0,375

   4. **karakter „A”** : az előző lépésben kapott [0.34375, 0.375] intervallumban keressük „M” tartományát.
      Alsó_határ = 0,34375 + (0,375-0,34375)*0 = 034375
      Felső_határ = 0,34375 + (0,375-0,34375)*0,5 = 0,35975

5. Miután az utolsó karakter tartományát is meghatároztuk, tudjuk, hogy az ALMA szót a 0,34375 - 0,35975 tartomány képviseli. Ebből a tartományból választunk értéket. Pl. 0,345.

   A kódolt üzenet: 0,345

Az aritmetikai kódolás algoritmusa

```
Alsó_határ = 0;
Felső_határ = 1;
Ciklus i=1-től hosszig
   Be: szimbólum_alsó_határ, szimbólum_felső_határ
   Intervallum = Felső_határ – Alsó_határ;
   Felső_határ = Alsó_határ + Intervallum*szimbólum_felső_határ;
   Alsó_határ = Alsó_határ + Intervallum*szimbólum_alsó_határ;
Ciklus vége
Kód = Véletlen(Alsó_határ, Felső_határ);
```

Aritmetikai kódolás - dekódolás

A dekódoláshoz az üzenet kódján kívül az egyes karakterek 7. táblázatban található tartományai, és a kódolt üzenet karaktereinek száma is rendelkezésünkre áll.

1. Első lépésben azt vizsgáljuk, hogy a táblázat alapján ez a kód melyik karakter intervallumába esik és az ehhez az intervallumhoz tartozó karakter lesz az eredmény.

2. Majd annak érdekében, hogy a következő karakterhez tartozó kódot is megkapjuk, az aktuális kódból kivonjuk az utoljára előállított karakterhez tartozó intervallum alsó határát és az így kapott értéket elosztjuk ugyanennek a karakternek az intervallum hosszával. A kapott érték lesz a következő karakter kódja, amivel az 1. lépés szerint járunk el, majd ezeket a lépéseket ismételjük, amíg el nem érjük a megadott karakterszámot.

Aritmetikai kód – dekódolás példa

Nézzük meg az előző példában kapott kód dekódolását.

A kapott kód: 0,345, az üzenet hossza: 4 karakter, a karakterek tartománya a 8. táblázatban látható.

Karakter	Tartomány
A	0 – 0,5
L	0,5 – 0,75
M	0,75 - 1

8. táblázat A karakterek tartományai

1. A táblázatból látható, hogy ez az érték a 0-0,5 tartományba esik, vagyis az „A” kódja, ezért ezt a szimbólumot írjuk le.

   Az aktuális dekódolt üzenetünk = A

   Kiszámoljuk a következő értéket:

   $$(0{,}345 - 0) / 0{,}5 = 0{,}69$$

2. A következő lépésben a 0,69-hez tartozó intervallumot keressük.

   Ez a kód a 0,5 – 0,75 tartományba esik, ami az „L”-hez tartozik, vagyis az üzenet következő karaktere: L

   Az aktuális dekódolt üzenetünk = AL

   Kiszámoljuk a következő értéket:

   $$(0{,}69 - 0{,}5) / 0{,}25 = 0{,}76$$

3. Keressük a 0,76-hoz tartozó intervallumot, ami 0,75-1, vagyis a következő karakter: M

   Az aktuális dekódolt üzenetünk = ALM

   Kiszámoljuk a következő értéket:

   $$(0{,}76 - 0{,}75) / 0{,}25 = 0{,}04$$

4. Ez a kód a 0 – 0,5 intervallumba esik, vagyis az „A”-t jelenti.

   Az aktuális dekódolt üzenetünk = ALMA

   Mivel tudjuk, hogy az üzenet 4 karakterből állt, a kódolás befejeződött

   A dekódolás eredménye: ALMA

Aritmetikai kód – dekódolás algoritmusa

```
Be: kód, hossz
üzenet = üres
Jel = kód tartományához tartozó szimbólum
üzenet = üzenet + jel
Ciklus i=1-től hosszig
    Új kód = (kód-alsóhatár)/intervallum_hossz
    Jel = Új kód tartományához tartozó szimbólum
    üzenet = üzenet + jel
    Kód=új kód
Ciklus vége
```

11. fejezet

Szótár alapú tömörítő algoritmusok

Ebben a fejezetben a veszteségmentes adattömörítés másik nagy csoportjával a szótár alapú tömörítéssel foglalkozunk. Ezeknél a módszereknél a kódolás során nem a sorozatok kódját írjuk le, hanem a szótárra hivatkozunk, melyet a kódolás során az aktuális üzenet alapján állítunk elő, miközben az eredeti üzenetet rövidebben írjuk le. Megismerjük a mai tömörítő algoritmusok legtöbbjének alapját képező LZ77 algoritmust, valamint az LZW kódot, ami az ismertetett módszerek közül egyedi abból a szempontból, hogy a szótárt nem kell átküldenünk, mert a dekódolás során önmagát állítja elő.

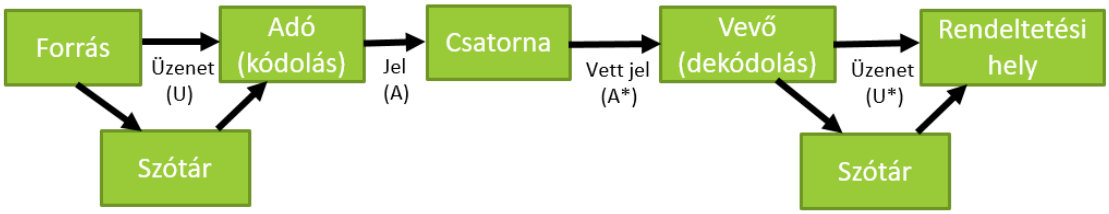
11.1. A szótár alapú adattömörítés folyamata

A szótár alapú adattömörítés a forrás kódolás egy lehetséges változata, amikor az a célunk, hogy az információt rövidebben tudjuk leírni.

Alapötlet: a módszer alapötlete az, hogy nem magát az üzenetet, illetve annak kódját küldjük át, hanem egy szótárt hozunk létre a különböző hosszúságú karaktersorozatok tárolására, az átküldendő jelsorozatba pedig a szótárra történő hivatkozások kerülnek. Ahhoz, hogy egy ilyen típusú kódolt üzenet dekódolható legyen, a fogadó oldalon természetesen ismerni kell a szótárt, vagyis a kóddal együtt azt is továbbítani kell.

Előnye: biztosítja a minimális átlagos kódhosszt

Hátránya: a szótárt is továbbítani kell az üzenettel együtt, hogy a vevő értelmezni tudja, ami növeli az adatforgalmat.



36. ábra A szótár alapú tömörítés folyamata

A fenti ábra már ismerős, hiszen a kódolás folyamatánál is ezt az ábrát láthattuk, melyet itt a szótárral egészítünk ki. Ahogy az ábrán is jól látható, az Adó oldalán is megjelenik a szótár, hiszen azt a kódolás során állítjuk elő, majd ezt a jellel együtt továbbítjuk és a Vevő oldalon ennek segítségével történik a dekódolás.

A következőkben néhány olyan kódolást fogunk megismerni, amelyek a karaktersorozatok ismétlődését kihasználva, egy szótárt előállítva kódolják az üzenetet, vagyis szótár alapú tömörítést végeznek.

11.2. Futamhossz kódolás (RLE - Run Length Encoding)

Ez a kódolás az egymás utáni sok azonos karakter ismétlődését használja ki, ezek tömörebb leírását teszi lehetővé. A módszer működési elve szerint, ha ismétlődő karaktereket találunk, akkor az ismétlődések számát kell meghatároznunk és ahelyett, hogy ezt az üzenetrészt változatlanul írnánk le, az ismétlődések száma és maga az ismétlődő karakter kerül az üzenetbe. Ahol pedig nincs ismétlődés, a karaktert változatlanul leírjuk. Annak érdekében, hogy meg tudjuk különböztetni a dekódoláskor, hogy a soron következő üzenetrész éppen az ismétlődések számát, vagy magát a karaktert tartalmazza, az ismétlődés esetén az ismétlődések száma elé egy vezérjelet írunk (pl.: $). Ekkor tudjuk, hogy a vezérjel után először az ismétlődések számát találjuk adott bitszámon, majd az ismétlődő szimbólumot, ahol nincs vezérjel, ott pedig maga a szimbólum szerepel.

Egy példán keresztül szemléltetve:

Az eredeti üzenet: 000000000101101110000000000

A kódolt üzenet: $90 10110111 $80

Fontos megjegyezni a módszerrel kapcsolatban, hogy minimum 4 ismétlődéstől érdemes használni. Ha ennél rövidebb szakaszon ismétlődik a szimbólum, akkor is változatlan formában írjuk le ugyanúgy, mintha nem lenne ismétlődés. Ennek

oka az, hogy ha 4 karakternél rövidebb sorozat helyett a vezérjelet, az ismétlődések számát és magát a karaktert is le kellene írnunk, akkor a tömörített üzenet akár hosszabb is lehetne, mint az eredeti, vagyis éppen ellentétes hatást érnénk el, mint amit szeretnénk.

A futamhossz kódolást elsősorban fax esetén, színskála-alapú képeknél használjuk, folytatólagos színváltásnál nem működik jól. A jpg formátumnál is jól működik, de elsősorban nem erre találták ki. Veszteségmentes tömörítő algoritmus, hiszen az információból semmit nem hagyunk el, csak rövidebb formában írjuk le.

11.3. LZ77 kódolás (Lempel-Ziv, 1977)

A módszert Abraham Lempel és Jakob Ziv publikálta 1977-ben, innen kapta a nevét. Ez a módszer egy úgynevezett csúszóablakos technikát használ, aminek az a lényege, hogy egy *k* hosszúságú ablakot csúsztatunk végig a kódolandó üzeneten. Ez az ablak két részből áll, egy keresőablakból és egy előre mutató ablakból és azt vizsgálja, hogy az előremutató ablakban található szövegrész szerepelt-e már korábban. Ha az előre mutató ablakban olyan bájt csoportot találunk, ami szerepel a keresőablakban, akkor a keresőablakbeli helyét és hosszát tároljuk, vagyis egy hivatkozást tárolunk az ismétlődő szövegrészlet helyett. A kódolás során előállított kód saját maga a kódtábla.

Az LZ77 (Lempel-Ziv) szótár felépítése

A szótár hivatkozásokat tartalmaz a keresőablakra. Ezek a hivatkozások az aktuális (előremutató ablakban szereplő) szövegrészlet keresőablakbeli helyét és hosszát mutatják.

A szótár minden egyes sora a következő felépítésű: index, hossz, karakter.

> Index: ha volt már ilyen karaktersorozat a pufferben (keresőablakban), akkor annak helyét adjuk meg. A hely megadása úgy történik, hogy visszafelé megszámoljuk az aktuális karaktertől kezdve hány karakterrel korábban szerpelt már ez a részlet. A leghosszabb egyezést keressük. Ha az aktuális részlet még nem szerepelt korábban, vagy csak egy karakternyi hosszú rész ismétlődne, akkor az index értéke 0.
>
> Hossz: ha találtunk ismétlődést a pufferben, akkor az indexen kívül meg kell adni azt is, hogy hány karakter hosszú az a rész, ami megegyezik

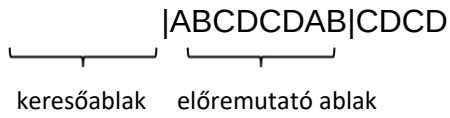
Karakter: abban az esetben, ha találtunk ismétlődést, akkor az egyező rész utáni karakter az előremutató ablakban, ha nem találtunk ismétlődést, akkor pedig maga az előremutató ablak aktuális karaktere.

LZ77 példa

Nézzük meg a módszer működését egy példán keresztül.

Kódolandó üzenet: ABCDCDABCDCD

A csúszóablak mérete 16 karakter, melyet egyenlően osztunk fel, vagyis a keresőablak és előremutató ablak tartalma is 8 karakter. Ekkor a kódolás kiinduló állapot a következő:



1. Vegyük az előre mutató ablak első karakterét: A

   A keresőablakban visszafelé haladva nézzük meg, hogy szerepelt-e már korábban ilyen karakter. Mivel a keresőablak még üres, biztosan nem szerepelt, ezért a kimenet:

Index	Hossz	Karakter
0	**0**	**A**

   Mivel 1 karakterből álló sorozatot kódoltunk, az ablakot eltoljuk eggyel jobbra:

   | A|BCDCDABC|DCD

2. Folytassuk a következő karakterrel, vagyis az előre mutató ablak első karakterével: B

   A keresőablakban visszafelé haladva nézzük meg, van-e ilyen karakter. Mivel nincs, ezért a kimenet:

Index	Hossz	Karakter
0	0	A
0	0	B

Az ablakot ismét eltoljuk jobbra eggyel:

| AB|CDCDABCD|CD

3. A soron következő karakter a C, ezzel ugyanúgy járunk el, mint az előző két karakterrel.

Index	Hossz	Karakter
0	0	A
0	0	B
0	0	C

Ezután az ablakot ismét eltoljuk jobbra eggyel:

| ABC|DCDABCDC|D

4. A soron következő D szintén nem szerepelt még.

Index	Hossz	Karakter
0	0	A
0	0	B
0	0	C
0	0	D

Az ablakot ismét eltoljuk jobbra eggyel:

| ABCD|CDABCDCD|

5. A soron következő karakter a C. A keresőablakban visszafelé haladva megnézzük van-e már ilyen karakter. Mivel 2 karakterrel korábban találunk ilyet (máshol nem), az index értéke 2. Ezután azt is meg kell néznünk, hogy milyen hosszúságban van egyezés. Mivel CD egyezik, a hossz 2. Az

első nem egyező karakter pedig a keresőablakban a CD után következő A.

Index	Hossz	Karakter
0	0	A
0	0	B
0	0	C
0	0	D
2	2	A

Az ablakot most 3 karakterrel csúsztatjuk el, mivel 2 karakter egyezett és az utána következő karaktert szintén beírtuk a táblába.

Az eltolás utáni állapot:

| ABCDCDA|BCDCD |

6. Az előremutató ablak első karakterét, B-t keressük a keresőablakban. 6 karakterrel korábban találunk B-t (máshol nem) és 5 karakter hosszban egyezik a keresőablak és az előremutató ablak tartalma. A * azt jelöli, hogy az üzenet végére értünk, nincs több karakter.

Index	Hossz	Karakter
0	0	A
0	0	B
0	0	C
0	0	D
2	2	A
6	5	*

Fontos megjegyezni, hogy amennyiben több, az előremutató ablakbeli résszel azonos kezdetű szakaszt találunk a keresőablakban, akkor a leghosszabb egyezés indexét írjuk a szótárba.

Az LZ77 tulajdonságai

- Veszteségmentes tömörítés
- Az ARJ és a ZIP is ezen az elven működik
- Információ-elméleti alapon nehéz bizonyítani, hogy miért működik jól, de a tapasztalatok azt bizonyítják, hogy jól működik
- Nincs lehetőség javításra a csúszás miatt. Ebből következően, ha elveszik egy bit, olvashatatlan lesz a teljes üzenet. Ezt sokszor tapasztaljuk is tömörített fájlok sérülése esetén.
- Csak a kódolandó karaktersorozatot közvetlenül megelőző adatsoron belül („a szótárban”) tud keresni.
- Az egy kóddal lekódolható adat hossza korlátozott. Annak eldöntése, hogy milyen hosszú ablakot használunk, nagyon nehéz kérdés. Ha túl kicsi ablakot választunk, akkor a tömörítés nem lesz elég hatékony. Ha túl nagy az ablakméret, akkor pedig nagyobb helyet foglal. Az optimumot kell megtalálni a tárhely és a tömörítés mértékének kérdését mérlegelve.
- A fenti hátrányos tulajdonságok ellenére a ma leggyakrabban használt módszerek legtöbbje ezen alapszik.

LZ78 (Lempel-Ziv, 1978)

Az algoritmust szintén Abraham Lempel és Jakob Ziv publikálta egy évvel az LZ77 után, annak módosított változataként. Adaptív szótár-alapú kódolás, mert mindig egyre hosszabb sorozatot kódol az LZ77-hez hasonlóan. A szótárt lépésről lépésre építi fel, a szótárba a soron követező sorszám mellé az a karakterfüzér kerül, ami korábban még nem szerepelt. Az algoritmus egyre jobb kódot hoz létre, menet közben tanul. Veszteségmentes tömörítő módszer [25].

Az LZ78 algoritmusa

```
Induláskor a szótár és az s üres
Ciklus i=1-től hosszig
  ch = következő karakter
  Ha a szótárban van s + ch akkor
    s = s + ch
  különben
    Ha s üres akkor
      index=0;
      szótárba ch
    különben
      index = s helye a szótárban
      szótárba s + ch
      s = üres
    Elágazás vége
    kimenet = kimenet + index + ch + " "
  Elágazás vége
Ciklus vége
```

11.4. LZW kódolás (Lempel-Ziv-Welch)

Az LZW kód algoritmusát Abraham Lempel, Jakob Ziv és Terry Welch publikálta 1984-ben. A nevek rövidítéséből adódik az algoritmus elnevezése, és az LZ78 finomított változatának tekinthető. Szintén a szótár alapú tömörítő algoritmusok körébe tartozik. A szótár általában 12 bites kódszavakat tartalmaz, melynek első elemei az üzenet egybetűs szimbólumai, ezt követik az üzenet feldolgozása során dinamikusan előállított kódszavak [26]. A módszer lehetővé teszi, hogy a dinamikus bővítés során egyre hosszabb karakter sorozatokat kódoljunk. Nagy előnye, hogy a szótárt nem kell átküldeni, hiszen a dekódolás során önmagát építi fel [13].

Az LZW lépései

1. Szótár inicializálás, az egybetűs szimbólumok felvétele a szótárba.
2. A szótárban keressük a jelenlegi részlettel egyező leghosszabb elemet.
3. A talált részlethez tartozó kódszót hozzáfűzzük a kódhoz.
4. A szótárba felvesszük kódolt részletet a következő karakterrel kiegészítve.
5. A folyamatot a 2. lépéstől ismételjük, amíg az üzenet végére nem érünk.

Az LZW kódolás algoritmusa

```
String = olvass 1 karaktert
Ciklus amíg van input karakter
   ch = olvass 1 karaktert
   Ha string+ch benne van a szótárban akkor
        string = string+ch
    Különben
        Ki: string kódja
        string+ch-t a szótárba
        string = ch
    Elágazás vége
Ciklus vége
Ki: string kódja
```

LZW kódolás példa

Nézzünk meg egy példát az előző algoritmus alkalmazására.

Kódolandó üzenet: ABABBAABBBAACABBBCAB

1. A szótár alapállapota: az egybetűs szimbólumokat tartalmazza a hozzájuk rendelt kódszavakkal

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**

2. Elkezdjük olvasni az üzenetet az első karaktertől indulva. A leghosszabb sorozat, ami még szerepel a szótárban: A
   A kódolt üzenet az A-hoz tartozó kódszó: **#1**

   A szótárba felvesszük a kódolt karaktert a következő karakterrel kibővítve: *AB*ABBAABBBAACABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**

   A kódolandó üzenet: A|BABBAABBBAACABBBCAB

3. A következő karaktertől folytatjuk (B). A leghosszabb sorozat, ami még szerepel a szótárban: B

   Ennek kódját hozzáírjuk a kódolt üzenethez, ekkor ennek tartalma: **#1#2**

   A szótárba felvesszük a kódolt szimbólumot a következő karakterrel kibővítve:

   A*BA*BBAABBBAACABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**

Kódolandó üzenet: AB|ABBAABBBAACABBBCAB

4. A következő karaktertől folytatjuk (A). A leghosszabb sorozat, ami még szerepel a szótárban: AB
   Ennek kódját hozzáírjuk a kódolt üzenethez, ekkor ennek tartalma: **#1#2#4**

   A szótárba felvesszük a kódolt szimbólumot a következő karakterrel kibővítve: AB*ABB*AABBBAACABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**

Kódolandó üzenet: ABAB|BAABBBAACABBBCAB

5. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: BA
   Ennek kódjával kibővítve a kódolt üzenet: **#1#2#4#5**

   A szótárba felvesszük a kódolt szimbólum sorozatot a következő karakterrel kibővítve: ABAB*BAA*BBBAACABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**

Kódolandó üzenet: ABABBA|ABBBAACABBBCAB

6. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: ABB
   Ennek kódjával kibővítve a kódolt üzenet: **#1#2#4#5#6**

   A szótárba felvesszük ezt a következő karakterrel kibővítve: ABABBA*ABBB*AACABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**

Kódolandó üzenet: ABABBAABB|BAACABBBCAB

7. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: BAA
   Ennek kódjával kibővítve a kódolt üzenet: **#1#2#4#5#6#7**

A szótárba felvesszük ezt a következő karakterrel kibővítve: ABABBAABB*BAAC*ABBBCAB

Kódolandó üzenet: ABABBAABBBAA|CABBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**

8. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: C

   Ennek kódjával kibővítve a kódolt üzenet: **#1#2#4#5#6#7#3**

   A szótárba felvesszük ezt a következő karakterrel kibővítve: ABABBAABBBAA*CA*BBBCAB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**
CA	**#10**

Kódolandó üzenet: ABABBAABBBAAC|ABBBCAB

9. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: ABBB
   Ennek kódjával kibővítve a kódolt üzenet: #1#2#4#5#6#7#8

   A szótárba felvesszük ezt a következő karakterrel kibővítve: ABABBAABBBAAC*ABBBC*AB

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**
CA	**#10**
ABBBC	**#11**

   Kódolandó üzenet: ABABBAABBBAACABBB|CAB

10. A következő szimbólumtól kezdve keressük a leghosszabb sorozatot, ami már szerepel a szótárban: CA
    Ennek kódjával kibővítve a kódolt üzenet: **#1#2#4#5#6#7#8#10**

    A szótárba felvesszük ezt a következő karakterrel kibővítve: ABABBAABBBAACABBB*CAB*

Karaktersorozat	Kódszó
A	#1
B	#2
C	#3
AB	#4
BA	#5
ABB	#6
BAA	#7
ABBB	#8
BAAC	#9
CA	#10
ABBBC	#11
CAB	#12

Ekkor már csak a B marad az üzenet végén, ennek kódját az üzenet végéhez írjuk, a szótárba már nem kerül új elem.

A kódolt üzenet: **#1#2#4#5#6#7#8#10#2**

LZW dekódolás algoritmusa

Be: régikód

Ki: régikódhoz tartozó szimbólum

Ciklus amíg van input

  Be: újkód

  string = újkódhoz tartozó szimbólum

  Ki: string

  ch = string első karaktere

  régikód+ch-t a szótárba

  régikód = újkód

Ciklus vége

LZW dekódolás példa

Nézzünk meg az előző példában előállított kód dekódolását.

Dekódolandó jel: **#1#2#4#5#6#7#3#8#10#2**

1. A szótár alapállapota: az egybetűs szimbólumok kódja. A vevő felé semmi mást nem küldünk át, a szótár a dekódolás során felépíti önmagát.

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**

2. Leírjuk az első két kódszóhoz tartozó szimbólumot a szótárból: A, majd ugyanígy a második kódszóhoz tartozó szimbólumot: B.

   Mivel a szótárt a dekódolás során dinamikusan építjük fel, hasonlóan a kódoláshoz az előző kódhoz tartozó szimbólumot, az aktuális kódhoz tartozó szimbólum sorozat első karakterével kibővítve felvesszük a szótárba.

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**

   Üzenet: **AB**

   Dekódolandó jel: #1#2|#4#5#6#7#3#8#10#2

3. A következő kódtól folytatjuk (#4). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: AB*AB*

   Majd a kódtáblába beírjuk az előző kódhoz (#2) tartozó sorozatot az aktuális sorozat (#4) 1. karakterével (A) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**

Üzenet: **ABAB**

Dekódolandó jel: #1#2#4|#5#6#7#3#8#10#2

4. A következő kódtól folytatjuk (#5). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: ABAB*BA*

   Majd a kódtáblába beírjuk az előző kódhoz (#4) tartozó sorozatot az aktuális sorozat (#5) 1. karakterével (B) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**

   Üzenet: **ABABBA**

   Dekódolandó jel: #1#2#4#5|#6#7#3#8#10#2

5. A következő kódtól folytatjuk (#6). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: ABABBA*ABB*

   Majd a kódtáblába beírjuk az előző kódhoz (#5) tartozó sorozatot az aktuális sorozat (#6) 1. karakterével (A) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**

Üzenet: **ABABBAABB**

Dekódolandó jel: #1#2#4#5#6|#7#3#8#10#2

6. A következő kódtól folytatjuk (#7). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: ABABBAABB*BAA*

   Majd a kódtáblába beírjuk az előző kódhoz (#6) tartozó sorozatot az aktuális sorozat (#7) 1. karakterével (B) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**

Üzenet: **ABABBAABBBAA**

Dekódolandó jel: #1#2#4#5#6#7|#3#8#10#2

7. A következő kódtól folytatjuk (#3). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: ABABBAABBBAA*C*

   Majd a kódtáblába beírjuk az előző kódhoz (#7) tartozó sorozatot az aktuális sorozat (#3) 1. karakterével (C) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**

   Üzenet: **ABABBAABBBAAC**

   Dekódolandó jel: #1#2#4#5#6#7#3|#8#10#2

8. A következő kódtól folytatjuk (#8). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: BABBAABBBAAC*ABBB*

   Majd a kódtáblába beírjuk az előző kódhoz (#3) tartozó sorozatot az aktuális sorozat (#8) 1. karakterével (A) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**
CA	**#10**

Üzenet: **BABBAABBBAACABBB**

Dekódolandó jel: #1#2#4#5#6#7#3#8|#10#2

9. A következő kódtól folytatjuk (#10). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: BABBAABBBAACABBB*CA*

   Majd a kódtáblába beírjuk az előző kódhoz (#8) tartozó sorozatot az aktuális sorozat (#10) 1. karakterével (C) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**
CA	**#10**
ABBBC	**#11**

Üzenet: **BABBAABBBAACABBBCA**

Dekódolandó üzenet: #1#2#4#5#6#7#3#8#10|#2

10. A következő kódtól folytatjuk (#2). Az ehhez tartozó kódszót hozzáírjuk az üzenethez: BABBAABBBAACABBBCA*B*

    Majd a kódtáblába beírjuk az előző kódhoz (#10) tartozó sorozatot az aktuális sorozat (#2) 1. karakterével (B) bővítve:

Karaktersorozat	Kódszó
A	**#1**
B	**#2**
C	**#3**
AB	**#4**
BA	**#5**
ABB	**#6**
BAA	**#7**
ABBB	**#8**
BAAC	**#9**
CA	**#10**
ABBBC	**#11**
CAB	**#12**

Az LZW szótár jellemzői

A szótárban szerepló kódszavak általában 12 bitesek, ami azt jelenti, hogy 2^{12} féle, vagyis 4096 különböző bejegyzés kerülhet a szótárba. Ebből általában az első 256 hely a karakterek számára van lefoglalva (ASCII kód esetén), a fennmaradó helyekre pedig a kódolás során dinamikusan előálló karaktersorozatok kerülnek. A tömörítés korlátja a kódméret lehet, hiszen alapvetően meghatározza, hogy meddig bővíthetjük a szótárt, hányféle kódszó kerülhet bele.

Megoldások arra az esetre, ha betelik a szótár (nem tudunk több új kódszót hozzáírni):

- Nem kódolunk tovább. Ez valószínűleg nem lesz jó megoldás.
- Nagyobb kódméretet választunk amennyiben ez lehetséges. Így a kódolható karaktersorozatok száma többszörösére növekedhet, hiszen minden egyes bittel való bővítés kétszerezi az előállítható kódszavak számát. Ez a legkézenfekvőbb megoldás.
- Töröljük a szótár elejét, ahol a rövidebb karaktersorozatok kódja szerepel. Ezzel kevésbé rontjuk a tömörítés hatékonyságát.
- A legkisebb gyakoriságú karakter sorozatokat töröljük a szótárból, szintén a tömörítés hatékonyságának érdekében.

Az LZW kódolás tulajdonságai

- Veszteségmentes tömörítés
- Radikális fájlméret csökkenés érhető el vele abból adódóan, hogy egyre hosszabb sorozatokhoz rendelünk kódot, így egyre hosszabb sorozat kódolható egyetlen kóddal (pl. képekben sok azonos színű pixel). Képek esetén gyakran jobb eredményt ad mint a jpeg, későbbi korrekció szempontjából előnyösebb lehet, hiszen veszteségmentes eljárás.
- Hatalmas előnye, hogy nem kell átküldeni a kódtáblát, mert a dekódolás során önmagát építi fel.

12. fejezet

HIBATŰRŐ RENDSZEREK ALAPJAI. HIBÁK ÉRZÉKELÉSE, JAVÍTÁSA.

Ebben a fejezetben azt fogjuk megvizsgálni, hogy abban az esetben, amikor az átvitel, vagy tárolás során sérül az információ, azt hogyan, milyen módszerekkel vehetjük észre, illetve amennyiben lehetséges, hogyan tudjuk javítani. Definiáljuk a kódellenőrzés, kódjavítás fogalmát. Megemlítjük a redundancia szerepét a hibajavításban és áttekintjük a fontosabb kódellenőrzési módszereket. Bevezetjük a kódszó és a kód Hamming távolságának fogalmát, valamint ismertetjük a különböző paritás ellenőrző módszereket és a CRC működését.

12.1. Hibatűrő rendszerek alapjai. Kódellenőrzés, kódjavítás fogalma.

A csatorna kódolás feladata

A kódolás folyamatát már korábban megismertük. Ilyenkor a forrás által kiválasztott üzenetet kódoljuk a megfelelő módszerrel, általában a forráskódolást, csatornakódolást és titkosító kódolást együttesen alkalmazva. Az így kódolt üzenetet jelnek hívjuk, ez már egy bináris formában tárolt információ, ezt küldjük át a csatornán. A vevő oldalon a kapott jel dekódolása történik annak érdekében, hogy előálljon a küldött üzenet. Az átvitel során azonban károsodhat az információ, így előfordulhat, hogy a vevő már nem a küldött jelet kapja meg,

hanem ennek sérült változatát. A cél természetesen az, hogy a küldött és a vett jel megegyezzen, vagyis U=U* teljesüljön.

Az eddig megismert kódolások mind a forráskódolások közé tartoztak, melynek célja a redundancia csökkentése annak érdekében, hogy minél tömörebben írjuk le az információt.

Ebben a fejezetben viszont azzal foglalkozunk, hogyan érhetjük el azt, hogy ne okozzon észrevehető hibát, ha egy bit megsérül. Ennek érdekében csatornakódolást alkalmazunk, amelynek célja a redundancia növelése által az információ hibájának felfedése, annak javíthatósága. Itt egy mesterségesen hozzáadott redundanciáról beszélünk, ami nem azonos azzal a redundanciával, amit a forrás kódolás során csökkenteni szeretnénk.

A kódolás célja

Informatikai rendszerekben a kódolás egyidejűleg több követelménynek is eleget kell, hogy tegyen. Ebből következően a különböző kódolásokat egy időben kell alkalmaznunk.

A kódolással szemben támasztott követelmények:

- A kódolt adat legyen tömör – erre a célra a forráskódolást alkalmazzuk, a redundanciát minimalizálva, ahogy azt a 10-11. fejezetben ismertetett módszereknél láttuk.
- Az üzenet szükség esetén visszaállítható legyen – ez a csatornakódolás segítségével valósítható meg, ami biztosítja a hiba felfedését és a javítás lehetőségét egy hozzáadott redundancia által. Ennek módszereit ismerjük meg ebben a fejezetben.
- Az adat titkosítása azért, hogy egy harmadik fél ne tekinthessen bele a kommunikációba, vagy minél nehezebben tehesse ezt meg – ezt a titkosító kódolással biztosíthatjuk.

A redundancia szerepe

Forráskódolás esetén definiáltuk a redundancia fogalmát, ami szerint a redundancia az üzenetnek az a része, ami szükségtelen abban az értelemben, hogy ha az a rész hiányozna, az üzenet akkor is lényegében teljes, vagy teljessé tehető lenne, vagyis az üzenet értelmezhetőségét nem rontja, ha ezt a részt elhagyjuk.

Hibatűrő rendszerek esetén azonban a redundanciáról kicsit más szemszögből beszélünk, bár a fenti definíció továbbra is igaz lesz. Csatorna

kódolás esetén a redundanciát mesterségesen adjuk hozzá az üzenethez valamilyen meghatározott szabály alapján. Itt nem az üzenetben rejlő, a tömörítés során kihasznált redundanciáról van szó. Itt nem pejoratív értelemben használjuk ezt a fogalmat, hanem elengedhetetlen feltétele a hibajavítás lehetőségének, ez a hibatűrés feltétele. A fenti definíció olyan értelemben lesz igaz a továbbiakban, hogy ha ezt a mesterségesen hozzáadott redundanciát elhagynánk, az az üzenet tartalmát nem befolyásolná. Ennek a hozzáadott redundanciának kizárólag a hibák felfedhetősége és javíthatósága szempontjából van jelentősége.

Kódellenőrzés, kódjavítás fogalma

A lehetséges módszerek áttekintése előtt fontos tisztázni a kódellenőrzés és a kódjavítás fogalmát, azok közötti különbséget, hogy azokat mindig a helyes értelemben használjuk.

Kódellenőrzés: annak vizsgálata, hogy ténylegesen az az üzenet érkezett-e meg, amit küldtünk.

Kódjavítás: az esetlegesen hibás üzenetet próbáljuk meg kijavítani, hogy visszaállítsuk az eredeti üzenetet.

A következőkben a módszerek ismertetése során arról is lesz szó, hogy nem minden esetben tudjuk javítani a hibát. Áttekintjük, hogy mikor áll elő az a helyzet, hogy csak felfedni tudjuk a hibát és mikor tudjuk javítani is.

12.2. Kódellenőrzési módszerek

A fenti definíciók alapján itt elsősorban a hiba felfedésére, vagyis a kódellenőrzésre koncentrálunk, de természetesen a végső cél a felfedett hiba javítása, amit azonban nem mindig tudunk megtenni.

Hozzáadott redundancia nélküli kódellenőrzési módszerek

A legegyszerűbb ellenőrzési módszer, amikor nem adunk hozzá semmilyen plusz információt az üzenethez, hanem egészen egyszerűen változatlan formában kétszer küldjük át a vevőnek. A fogadó oldalon összehasonlítjuk az átküldött üzeneteket. Ha megegyeznek, akkor azt mondjuk, hogy az üzenet nem sérült. Ha különböznek, akkor pedig tudjuk, hogy valami hiba történt, de nem tudjuk melyik üzenet volt a helyes. Azért, hogy ez egyértelművé váljon, újra át kell küldeni az üzenetet, majd egy újabb összehasonlítást végzünk, ekkor már mindhárom üzenetet összehasonlítva. Amelyik korábban kapott üzenettel

egyezik a harmadik üzenet, az lesz a helyes. Technikailag ez a legegyszerűbb módszer, de nem hatékony, mivel hiba esetén háromszor kell átküldeni az üzenetet, vagyis a hatékonysága 33% és azt mondhatjuk, hogy ennek a módszernek a legnagyobb a redundanciája annak ellenére, hogy nem adunk plusz redundanciát az üzenethez. A redundancia itt úgy jelenik meg, hogy legjobb esetben is kétszer kell átküldeni az üzenetet, tehát a redundancia mérete ilyenkor az üzenet méretével egyezik meg. Olyan esetben, amikor hibás az üzenet, ez a redundancia kétszeresére nő.

Hozzáadott redundancián alapuló kódellenőrzési módszerek

Az üzenet többszöri átküldése helyett egy mesterségesen generált plusz redundancia hozzáadása hatékonyabb módszert eredményez. Ebben az esetben a forráskódolás után kapott jelet kibővítjük, így a többlet bitek segítségével, amelyeket valamilyen szabály szerint adunk az üzenethez, lehetővé válik a hibafelfedés, esetlegesen a javítás is. A továbbiakban ennek legalapvetőbb módszereivel foglalkozunk.

Paritás bit hozzáadása

Ennél a módszernél az üzenet minden egyes bájtjához egy plusz bitet fűzünk, vagyis a bájtok felépítése:

7 bit információ + 1 bit paritás

A paritás bitet az adó rendeli hozzá az üzenethez úgy, hogy az összkód páros legyen. Ez azt jelenti, hogy megszámolja hány 1 van az üzenet adott bájtjában, és ha ez a szám páratlan, akkor a paritás bit 1 lesz, ha páros, akkor 0. Vagyis a paritás bittel együtt minden bájtban páros számú 1-nek kell lennie.

A vevő a párosságot ellenőrzi blokkonként, ha páros számú 1 van a blokkban, feltételezi, hogy jó.

Nézzük meg a paritás bit alkalmazását példákon keresztül

1. példa:

   az aktuális bájt: 0110010

   megszámoljuk az 1-eket, páratlan számú 1 van benne

   paritás bit: 1

   a blokk a paritás bit hozzáfűzése után: 01100101

2. példa:

   aktuális bájt: 0110110

   megszámoljuk az 1-eket, páros számú 1 van benne

   paritás bit: 0

   a blokk a paritás bit hozzáfűzése után: 01101100

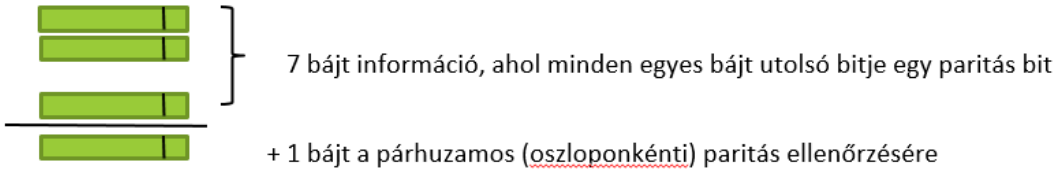
A módszer hátránya, hogy csak páratlan számú hibát tud felfedni, mivel páros számú hiba esetén az összkód páros marad, elfedve a hibát.

Paritás bájt hozzáadása

A paritás bájt hozzáadása az előző módszerhez hasonlóan működik. Ugyanúgy minden egyes bájt utolsó bitje egy paritás bit, de ezt kiegészítjük azzal, hogy minden 8. bájt egy hozzáadott paritás bájt lesz, amit az előző 7 bájtból számolunk. A paritás bájt létrehozásának elve ugyanaz, mint a paritás bit esetén, csak itt oszloponként számoljuk meg az 1-eket és ha a számuk páros, akkor a paritás bájt adott oszlopbeli bitje 0 lesz, ha páratlan, akkor 1.

A paritás bájt felépítése:

7 bájt információ (bájtonként 7 bit információ + 1 bit paritás) + 1 bájt paritás



Nézzük meg a paritás bájt hozzáadását egy példán keresztül:

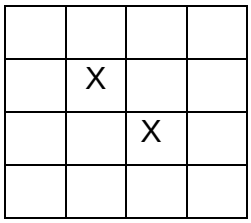
A paritás bájt hibafelfedő képessége

Könnyen belátható, hogy paritás bájt esetén a hiba felefedő képesség lényegesen jobb, mint ha csak paritás bitet használunk, hiszen a vízszintes és a függőleges paritást is ellenőrizzük.

Vegyük sorra a lehetséges eseteket:

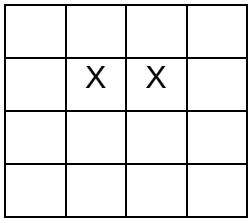
1 bit hiba: nem csak észrevesszük a hibát, hanem a valószínű helyére is következtethetünk, hiszen vízszintesen és függőlegesen is páratlan lesz az 1-ek száma.

2 bit hiba: a hiba felfedhető, mert vagy vízszintesen, vagy függőlegesen páratlan lesz az 1-ek száma, de a helyét nem mindig tudjuk meghatározni. A példákban az egyszerűség kedvéért a bájtoknak csak a hibás részét ábrázoljuk, a teljes bájtok esetén egy 8x8-as táblázatot kapnánk. A 37. ábrán arra az esetre látunk példát, amikor a hiba helye is felfedhető, hiszen az X-el jelölt bitek hibásak, és ebben az elrendezéseben mind a vízszintes, mind a függőleges paritás páratlan lesz.



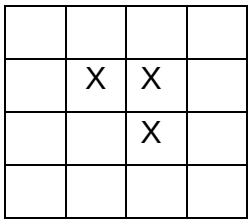
37. ábra A hiba helye is felfedhető

A 38. ábrán pedig egy olyan hiba elrendezést láthatunk, ahol a hiba helye nem fedhető fel, mivel a függőleges paritás ugyan páratlan lesz, de a vízszintes páros, vagyis nem fogjuk tudni, hogy melyik sorban (bájtban) van a hiba.

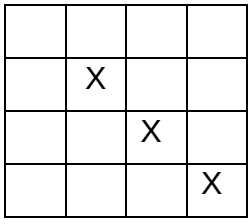


38. ábra A hiba helye nem fedhető fel

3 bit hiba: a hiba felfedhető, de nem mindig javítható. A 39. ábrán egy olyan hiba elrendezést látunk, ahol egyik hiba helyét sem tudjuk felfedni, mert vagy a vízszintes, vagy a függőleges paritásuk lesz páros, ami alapján nem vesszük észre a hibát. A 40. ábrán mindhárom hiba helye megadható, mert mindegyiknél mind a vízszintes, mind a függőleges paritás páratlan.



39. ábra A hiba helye nem fedhető fel



40. ábra A hiba helye is felfedhető

4 bit hiba: elhelyezkedéstől függ, hogy egyáltalán felfedhető-e a hiba. A 41. ábrán arra látunk példát, amikor annak ellenére, hogy 4 hibás bitünk van, nem vesszük észre, hogy hibás az üzenet.

	X	X	
	X	X	

41. ábra A hiba helye is felfedhető

12.3. Hamming távolság

A kódellenőrzés során felmerülő problémák

Vegyük sorra, hogy a vevő oldalon milyen problémákat tapasztalhatunk az üzenet sérülése esetén, amikor hozzáadott redundancia nélkül is észrevehetjük, ha az üzenet hibás.

1. Olyan kódszó érkezik, ami nincs a kódtáblában. Ilyenkor a jel dekódolása során a sérült kódszónál egyértelműen észrevesszük, hogy hibás az üzenet.

   Kódszavak:

   A: 000

   B:011

   C:101

   D:110

   Kapott üzenet: 011 001 011 000

   B ? B A

2. A bitek sérülése következtében másik értelmes kódszót kapunk. Ilyenkor a dekódolás során nem vesszük észre a hibát, hiszen minden kódszó szerepel a kódtáblában. A hibát csak az üzenet értelmezésekor fedezhetjük fel.

   Kódszavak:

   A: 000

   B:001

   C:010

Kapott üzenet: 001 000 001 001

B A B B

Eredeti üzenet: 001 000 001 000

B A B A

Kódszavak Hamming távolsága

Az előzőekben ismertetett problémával kapcsolatban egy újabb fogalmat kell megismernünk, ugyanis a 2. esetben, mikor a bit sérülése következtében egy másik értelmes kódszót kapunk, a hiba nem fedhető fel egyértelműen.

Definíció: a kódszavak Hamming távolsága az a szám, amely megadja, hogy hány bitet kell megváltoztatni a kódszóban ahhoz, hogy egy másik érvényes kódszót kapjunk. Jelölése: $D(kódszó_1,kódszó_2)$

Például a következő kódtábla esetén:

A: 000

B: 010

C:111

A kódszavak Hamming távolságait minden lehetséges kódszópárra meg kell néznünk:

D(A,B)=1 {a 2. bit különbözik}

D(A,C)=3 {minden bitjük különbözik}

D(B,C)=2 {az 1. és a 3. bitjük különbözik}

Kód Hamming távolsága

Definíció: Az összes lehetséges kódszópár távolságának minimuma. Jelölése: D_G

Például a következő kódtábla esetén:

A: 000

B: 010

C:111

Először a kódszavak Hamming távolságát kell meghatároznunk:

D(A,B)=1, D(A,C)=3, D(B,C)=2

Ebből a kód Hamming távolsága: D_G=1

A Hamming távolság a kódellenőrzés, kódjavítás esetén alapvető fontosságú fogalom, hiszen hiba érzékelhetőség és javíthatóság feltétele a megfelelő Hamming távolság.

Hiba felfedhetőség és javíthatóság

k bit hiba jelzéséhez szükséges Hamming távolság:

$$D_G = k + 1 \tag{40}$$

k bit hiba javításához szükséges Hamming távolság:

$$D_G = 2k + 1 \tag{41}$$

Vegyük sorra a lehetséges eseteket, amelyekből látható, hogy a fenti összefüggések valóban teljesülnek.

$D_G = 0$: nem lehetséges, hiszen azt jelenti, hogy két kódszó megegyezik egymással, vagyis két különböző szimbólumhoz ugyanazt a kódszót rendeltük. Ez ellentmond a kód definíciójának.

$D_G = 1$: nem garantálható a hiba felfedése sem, hiszen ha 1 bit módosul, egy másik érvényes kódszót kaphatunk.

Például A: 000, B: 001 esetén 000 helyett 001-et kapunk.

$D_G = 2$: 1 bit hiba felfedhető, ugyanis azt jelenti, hogy 2 bit megváltoztatása szükséges ahhoz, hogy egy másik érvényes kódszót kapjunk. 1 bit megváltoztatásával a kód érvénytelen lesz. Javítani viszont nem tudjuk a hibát, mert nem tudjuk, hogy melyik szomszédos kód a helyes (pont a kettő között van).

Például A: 000, B: 011, C: 101 esetén 000 helyett 001-et kapunk, akkor észrevesszük, hogy a kódszó érvénytelen, hiszen nem találjuk a kódtáblában, de a kapott kódszóból 1 bit megváltoztatásával előállítható A: 00<u>0</u>, B: 0<u>1</u>1, C: <u>1</u>01 közül bármelyik szimbólum, ezért nem tudjuk melyik a helyes.

D_G = 3: ebben az esetben 1 bit hiba felfedhető és javítható, mivel 1 bit hiba esetén 1 bit távolságra pontosan 1 érvényes kódszó található (a másik 2 bit távolságra van)

Például A: 0000, B: 1110, C: 1101 esetén 0000 helyett 0001-et kapunk, akkor észrevesszük, hogy a kódszó érvénytelen, hiszen nem találjuk a kódtáblában, és a kapott kódszóból 1 bit megváltoztatásával csak A: 0000 szimbólum állítható elő, ezért egyértelműen javítható a hiba.

12.4. Hamming kód

A Hamming kód a lineáris hibajavító kódok egy speciális csoportjába tartozik. A fentiekhez hasonlóan paritás ellenőrzést végez, de a korábbiaktól eltérően 3 bit paritással dolgozik. A paritással kiegészített kódszavakra továbbra is érvényes, hogy az összkód páros lesz, vagyis a benne szereplő 1-ek száma páros [13].

Hamming kódtábla

Többféle Hamming kódtábla létezik, ezek közül a 8421-es BCD kódhoz tartozót a 42. ábrán láthatjuk. Ez egy 7 bites kód, ahol 4 biten írjuk le az összes tízes számrendszerbeli számjegy BCD kódját, majd ezeket kiegészítjük 3 bit paritással. A 8, 4, 2, 1 fejlécű oszlopokban szerepel maga a számjegy 8421 kódja, a 0 fejlécű oszlopokban pedig a paritás bitek. A kód automatikus hibajavítást tesz lehetővé 1 bit hiba esetén.

	0	**0**	**8**	**0**	**4**	**2**	**1**
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1

42. ábra A Hamming kód kódtáblája

A kapott üzenet ellenőrzése úgy történik, hogy az ellenőrzés során előállítjuk a kapott 7 bites üzenetből a következő bit sorozatokat (ahol az index a bit sorszáma az üzenetben):

$$\mathrm{E}_1\colon \mathrm{A}_1\mathrm{A}_3\mathrm{A}_5\mathrm{A}_7$$

$$\mathrm{E}_2\colon \mathrm{A}_2\mathrm{A}_3\mathrm{A}_6\mathrm{A}_7$$

$$\mathrm{E}_3\colon \mathrm{A}_4\mathrm{A}_5\mathrm{A}_6\mathrm{A}_7$$

E_i értéke:

0 – ha a kifejezés értéke páros

1 – ha a kifejezés értéke páratlan

A kapott E_i értékek alapján az alábbi képlettel határozható meg, hogy melyik a hibás bit:

$$E_1 + E_2 \cdot 2 + E_3 \cdot 2^2 \qquad (42)$$

Nézzünk meg egy példát arra, hogyan javítható a hiba a Hamming kód alkalmazásával:

Eredeti üzenet: 0101010

Kapott üzenet: 0001010

Előállítjuk az E_i bitsorozatokat a definíció szerint:

$E_1\colon A_1A_3A_5A_7$ = 0000 (a kapott üzenet 1, 3, 5, 7 indexű bitjei)

$E_2\colon A_2A_3A_6A_7$ = 0010 (a kapott üzenet 2, 3, 6, 7 indexű bitjei)

$E_3\colon A_4A_5A_6A_7$ = 1010 (a kapott üzenet 4, 5, 6, 7 indexű bitjei)

Kiszámoljuk az E_i bitsorozatok értékét:

$E_1 = 0$ (mert a kifejezés értéke páros)

$E_2 = 1$ (mert a kifejezés értéke páratlan)

$E_3 = 0$ (mert a kifejezés értéke páros)

A kapott E_i értékekből (42) segítségével kiszámoljuk, melyik bit a hibás:

$E_1 + E_2 \cdot 2 + E_3 \cdot 2^2$ = 0 + 1*2 + 0*2^2 = 2

A 2. bit hibás (0<u>0</u>01010), ezt javítva kapjuk: 0101010, ami az eredeti üzenet, a kódtáblában kikeresve a hozzá tartozó decimális számjegy 2.

12.5. Ciklikus redundancia ellenőrzés (CRC - Cyclic Redundancy Check)

A korábban ismertetett paritás ellenőrzési módszerek nem elég hatékonyak. Ahogy láttuk, sok esetben csak arra alkalmasak, hogy a kódellenőrzés során észrevegyük, ha hibás a kapott üzenet, de még ez sem minden esetben garantált. Ahhoz, hogy a hatékonyságot növelni tudjuk, egy hosszabb bitsorozatot kell fűznünk az üzenethez.

A CRC működése

A CRC szintén a hiba felfedésére szolgáló általános módszer, amely egy hozzáadott redundancia segítségével ellenőrzi az üzenet hibátlanságát, vagyis a csatorna kódolás egyik változata. Az úgynevezett ellenőrző összeget az üzenet végéhez rendeljük hozzá a hitelesítés érdekében. Ez a módszer csoportos bithibák felismerésére szolgál a polinom aritmetika (polinom osztás) 2-es maradékosztályán (modulo2) alkalmazásával [13].

A kódot alkotó biteket egy polinom együtthatóiként kezeljük. Egy *n* bites kód esetén z együtthatók x^{n-1} és x^0 közötti értékek lesznek. Így a kapott polinom a következőképpen írható fel:

$$x^{n-1} + x^{n-2} + \cdots + x^1 + x^0 \qquad (43)$$

Ez azt jelenti, hogy ha az *M(x)*-szel jelölt kódunk 10111001, akkor csak azokat a tagokat felírva, ahol az eredeti kódban 1 szerepel, az előállított polinom a helyiértékeknek megfelelően:

$$x^7 + x^5 + x^4 + x^3 + x^0$$

A polinomok alkalmazásakor a kommunikációban résztvevőknek meg kell egyezniük egy közös polinomban, amelyet generátor polinomnak nevezünk és általában *G(x)*-el jelöljük. A generátor polinommal kapcsolatban alapvető követelmény, hogy a legalsó és a legfelső bitjének 1-esnek kell lennie, valamint a továbbítandó üzenet legyen hosszabb, mint maga a generátor polinom. A CRC előállításakor el kell döntenünk, hogy hányadfokú polinomot használjunk (8, 12, 16, 32), majd ennek megfelelő számú 0-t fűzünk az eredeti üzenethez. Ezt a hozzáfűzött redundáns részt *R*-rel jelöljük. Ezután elvégezzük a kapott polinom maradékos osztását a generátor polinommal [19]:

$$r(x) = (M(x) + R) \; mod \; G(x) \qquad (44)$$

Az alábbi táblázatokban a modulo 2-ben végzett számítási szabályokat tekintjük át [27]. Vegyük észre, hogy a modulo 2-ben végzett összeadás

eredménye megegyezik a XOR művelet eredményével, mivel az 1+1 esetén a túlcsordulást nem írjuk le.

+	0	1
0	0	1
1	1	0

9. táblázat Összeadás mod 2

*	0	1
0	0	0
1	0	1

10. táblázat Szorzás mod 2

A kapott maradék maximum *R*-ed fokú lesz. Ezt kivonva az eredeti polinomból megkapjuk *T(x)*-et, ez lesz a továbbítandó üzenet.

$$T(x) = (M(x) + R) + r(x) \qquad (45)$$

Az így kapott üzenet (*T(x)*) által meghatározott polinomnak oszthatónak kell lennie a generátor polinommal, ezért a vevő oldalon úgy ellenőrizhető a kapott üzenet, hogy a generátor polinommal osztva megnézzük, keletkezik-e maradék [17]. Ha van maradék, az azt jelenti, hogy hiba történt az átvitel során.

Az eljárás matematikailag meglehetősen bonyolult, de a számítógépek ezt nagyon gyorsan képesek elvégezni. 1961-ben Peterson és Brown bebizonyította, hogy az ellenőrző kód léptető-regiszterekkel előállítható. Ezt a hardvert ma már a legtöbb hibajavítással kapcsolatba kerülő áramkör használja. A *m*-bites ellenőrző bittel ellátott polinomkód legfeljebb *m*-bites csoportos bithibát képes jelezni [17]. A CRC-t nemzetközi szabvány polinomok segítségével hozzuk létre (valamilyen számítási algoritmussal bitek közötti logikai kapcsolat alkalmazásával)

A gyakorlatban három polinom vált szabvánnyá:

- CRC-12 = $x^{12} + x^{11} + x^2 + x^1 + 1$, akkor használjuk, amikor a karakterhossz 6 bit.
- CRC-16 = $x^{16} + x^{15} + x^2 + 1$, 8 bites karakterekhez használható.
- CRC-CCITT = $x^{16} + x^{12} + x^5 + 1$, szintén 8 bites karakterekhez alkalmazhatjuk
- CRC-32 = $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ IEEE 802.3 (Ethernet) szabványban felhasznált generátor polinom.

A CRC algoritmusa

```
Be: számláló, puffer
B=0;
Ciklus amíg számláló<>0
   tmp1 = (Eltolás 8-al jobbra) and 00FFFFFF
   p = p+1
   tmp2 = CRCtábla[CRC xor puffer[p] and FF]
   CRC = tmp1 xor tmp2
   számláló = számláló -1
Ciklus vége
```

A ciklussal végig megyünk a puffer elemein, aminek a CRC-jét szeretnénk előállítani. Az and szerepe *tmp1* esetén az első két bit levágása, a többi változatlanul hagyása mellett, *tmp2* esetén pedig a túlcsordulások kezelése. A CRCtábla 8 bites CRC esetén 256 féle elemet tartalmaz, ebből választjuk ki a *CRC xor puffer[p] and FF* művelet által meghatározott indexű elemet [13].

CRC átvitel

A fogadó *m(x)* üzenetet fogadja.

- Abban az esetben, ha nem történt hiba az átvitel során, akkor az

$$m(x)\ mod\ G(x) = 0$$

vagyis *m(x)=T(x),* a küldött üzenet és a kapott üzenet megegyezik.

- Abban az esetben, ha hiba történt az átvitel során, akkor

$$m(x) = T(x) + E(x)$$

ahol E(x) a hibapolinom. A bit hibákra vonatkozóan a következő összefüggés írható fel:

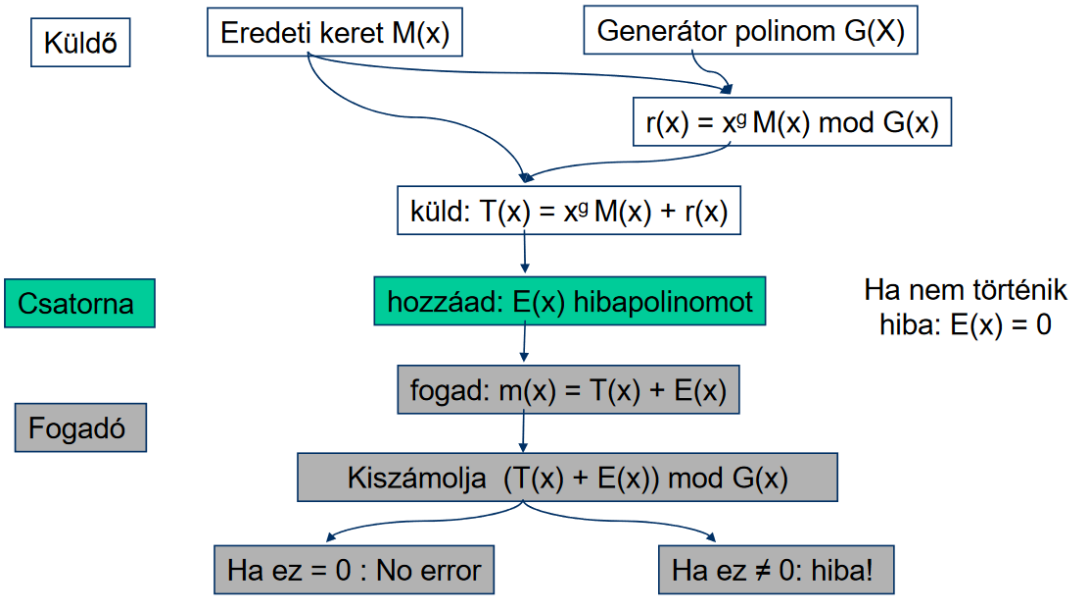
$$m(x)\ mod\ G(x) = \big(T(x) + E(x)\big)\ mod\ G(x)$$

vagyis

$$m(x)\ mod\ G(x) = T(x)\ mod\ G(x) + E(x)\ mod\ G(x)$$

ahol $T(x)\ mod\ G(x) = 0$ és $E(x)\ mod\ G(x)$ a hiba indikátor [27].

A CRC működésének áttekintése a 43. ábrán látható.



43. ábra A CRC áttekintése [27]

A generátor polinom megválasztása

A bithibák felismerése szempontjából alapvető fontosságú a megfelelő generátor polinom megválasztása. A hibákat csak abban az esetben nem ismerjük fel, ha a hibapolinom (*E(x)*) többszöröse a generátor polinomnak (*G(x)*-nek) [27].

- 1 bit hiba: $E(x) = x^i$, ekkor a hiba az i-edik pozícióban van.
  Ha *G(x)* legalább 2 nemnulla együtthatót tartalmaz, akkor *E(x)* nem többszörös.
- 2 bit hiba: $E(x) = x^i + x^j = x^j\left(x^{i-j} + 1\right)$, ahol *i>j*.
  G(x) nem szabad, hogy osztója legyen $(x^h + 1)$-nek semmilyen *h*-ra, $0 \leq h \leq k$ a maximális üzenet hosszig.
- Páratlan számú hiba: ekkor *E(x)* nem többszöröse *(x+1)*-nek
  Ötlet: legyen *(x+1)* osztója *G(x)*-nek, ekkor *E(x)* nem többszöröse *G(x)*-nek.

IRODALOMJEGYZÉK

[1] Kutor László, Bevezetés az informatikába, Óbudai Egyetem, 2014

[2] Retter Gyula, Fuzzy, neurális, genetikus, kaotikus rendszerek, Akadémiai Kiadó, 2006

[3] Imre J. Rudas, Hybrid Systems, Encyclopedia of Information Systems, Vol. 2, Elsevier, 2003

[4] Álmos Attila, Dr. Horváth Gábor, Dr. Várkonyiné dr. Kóczy Annamária, Győri Sándor, Genetikus algoritmusok, Typotex, 2002

[5] Altrichter Márta, Horváth Gábor, Pataki Béla, Strausz György, Takács Gábor, Valyon József, Neurális hálózatok, Panem, 2006

[6] Tóth Bálint Pál, Neurális hálózatok, Beszélő számítógépek mély gondolatokkal, Élet és Tudomány, 2016/09/15, http://www.eletestudomany.hu/neuralis_halozatok

[7] Kóczy T. László, Tikk Domonkos, Fuzzy rendszerek, Typotex, https://www.tankonyvtar.hu/hu/tartalom/tkt/fuzzy-rendszerek-fuzzy/adatok.html

[8] Oláh Ferenc, A fuzzy logika – alapismeretek, https://autotechnika.hu/cikkek/motor-eroatvitel/8313/a-fuzzy-logika-alapismeretek

[9] Hlács Ferenc, Ötven éve diktál a Moore-törvény, 2015.04.20, https://www.hwsw.hu/hirek/53856/moore-torveny-jubileum-intel.html

[10] Gordon Moores Law Of Exponential Growth And The Singularity, http://www.kidskunst.info/linked/gordon-moores-law-of-exponential-growth-and-the-singularity-676f72646f6e.htm

[11] Gáspár Merse Előd, Mi az a technológiai szingularitás, és mikor jön már el? 2018.01.03, https://qubit.hu/2018/01/03/mi-az-a-technologiai-szingularitas-es-mikor-jon-mar-el
[12] A negyedik ipari forradalom, Ipari digitalizáció, https://digitalizationindustry.com/hu/2017/07/26/a-negyedik-ipari-forradalom/
[13] Tóth Ákos, Információelmélet, Budapesti Műszaki Főiskola, 2002
[14] Claude E. Shannon, Warren Weaver, A kommunikáció matematikai elmélete, Országos Műszaki Információs Központ és Könyvtár, 1986
[15] Lócsi Levente, Numerikus módszerek, ELTE IK, 2013
[16] Sergyán Szabolcs, Algoritmusok és adatszerkezetek I, ÓE-NIK 5014, 2016
[17] Anrew S. Tannenbaum, Számítógéphálózatok, Panem, 2013
[18] Számrendszerek, kódolás, I+K technológiák, BME_KJIT, 2018, http://kjit.bme.hu/images/stories/targyak/ik_technologiak/I_K_foliak.pdf
[19] Kovács Emőd, Bíró Csaba, Perge Imre, Bevezetés az informatikába, Eszterházy Károly Főiskola, 2013
[20] http://www.lookuptables.com/ebcdic_scancodes.php
[21] Kódolás, dekódolás, https://slideplayer.hu/slide/12277576/
[22] Tömörítő programok, http://www.winace.hu/sajto/2001_02_25_internews/index.htm
[23] A Morse-féle távíró, http://dongo.biz/dm/03-villany-mint-informacio/04
[24] Győrfi László, Győri Sándor, Vajda István, Információ- és kódelmélet, Typotex, 2002
[25] Sallai András, LZ78 algoritmus, SzitWiki, https://szit.hu/doku.php?id= oktatas:programozás: algoritmusok:lz78
[26] Hogyan működik az LZW tömörítő algoritmus, Dynamicart, http://dynamicart.hu/blog/hogyan-mukodik-az-lzw-tomoritesi-algoritmus.html
[27] Lukovszki Tamás, Adatkapcsolati réteg – CRC utólagos hibajavítás, Számítógépes hálózatok, ELTE, 2012