Veszprémi Egyetem

Matematikai és Számítástechnikai Tanszék

DIPLOMADOLGOZAT

A számítástudomány alapjainak szemléltetése az oktatásban

Témavezető:
Dr. Szalkai István

Írta:
Cziráki Tamás

Veszprém
2004.

„Amit egyáltalán meg lehet mondani, azt meg lehet mondani világosan; amiről pedig nem lehet beszélni, arról hallgatni kell... Érezzük, hogy még ha feleletet is adtunk valamennyi lehetséges tudományos kérdésre, életproblémáinkat ezzel még egyáltalán nem érintettük. Akkor persze nem marad egyetlen további kérdés sem... Kétségtelenül létezik a kimondhatatlan."

Ludwig Wittgenstein

Tartalomjegyzék

Előszó

Az első elektromos számítógépeket eredetileg azzal a céllal építették meg a múlt század első felében, hogy a tudományos kutatás eszköztárát bővítsék egy olyan eszközzel, amely könnyebbé teszi a matematikai számítások elvégzését.

Mostanra már nagyon messze kerültünk ettől a céltól, és akkoriban még bizonyára senki sem gondolta volna, hogy egykor majd a számítógépek akkora fontossággal bírnak, hogy egy újfajta társadalmi berendezkedés alapját képezik. Mert manapság már erről van szó. A szociológusok és más társadalomtudományi elemzők, de hétköznapi emberek is egyre gyakrabban használják azt a kifejezést, hogy ***információs társadalom****. Ezt általában úgy értik, hogy ma az információ — pontosabban az a képesség, hogy megszerezzük és birtokoljuk az információt — az, ami alapjaiban határozta meg az emberek életét.*

Mivel ennyire hétköznapivá és természetessé vált együttélésünk a (számító)gépekkel, ez maga után vonta azt, hogy egyre ritkábban teszünk fel kérdéseket, ezen eszközök működésével és képességeivel kapcsolatban. Azonban alapvető fontosságú, hogy legalább azok a szakemberek akik az informatikai tudományokkal foglalkoznak, megismerkedjenek az ilyen jellegű kérdésekkel, sőt maguk is fogalmazzanak meg hasonlókat.

De az is vitathatatlan, hogy ez utóbbi csak akkor történhet meg, ha azok a tanárok, akik a jövő szakembereit képzik, maguk is tisztában vannak a problémákkal, és az oktatás során a legkülönbözőbb formában és eszközökkel tudják bemutatni azokat és a megoldásukra szolgáló alternatívákat.

Az elkövetkezendő oldalakon ez utóbbihoz próbálunk meg hozzájárulni az által, hogy részletesen tárgyalunk néhány, a számítástudománnyal kapcsolatos alapproblémát. Azonban dolgozatunk címében nem véletlenül szerepel az a szó, hogy „szemléltetés". Az elméleti problémák mellett bemutatunk egy számítógépes programot is, amely ez utóbbit próbálja meg szolgálni.

Ezen a helyen szeretnék köszönetet mondani azoknak — tanáraimnak, ismerőseimnek és barátaimnak —, akik segítsége nélkül ez a munka jelen formájában nem készülhetett volna el. Külön köszönet illeti vezető tanáromat, mert — bár Ő mindig lekiismeretesen végezte a munkáját — én nem mindig voltam a legszorgalmasabb diák.

1. fejezet

Bevezetés

Természetesnek vesszük, hogy minden problémát és feladatot meg lehet oldani, mindent ki lehet számítani. Korábban, ha egy problémára nem sikerült megoldást találni, akkor ezt annak tudták be, hogy a tudományos fejlődés még nem ért el arra a pontra, hogy megtalálja a választ, de majd az utókor — ha már bővebbek lesznek az ismeretek —, meg fogja adni a válaszokat.

Ez alól a matematika sem kivétel, és ennek története is számos olyan példával szolgál, amelyre igaz az iménti megállapítás. Hosszú időn keresztül senkiben sem merült fel, hogy esetleg ebben a tudományban is létezhetnek olyan problémák és kérdések, amelyeket nem lehet megválaszolni — még elviekben sem. Éppen ezért, nagy megdöbbenést váltott ki, amikor a múlt század harmincas éveiben ez beigazolódott.

Nem sokkal ezután indult meg az elektromos számítógépek kifejlesztése, és az azokat érintő műszaki jellegű problémák mellett előtérbe kerültek a velük kapcsolatos elméleti kérdések is, olyanok, mint például: mi az algoritmus, vajon minden probléma megoldására létezik–e algoritmus, hogyan lehet definiálni a „gyors algoritmus" és a „lassú algoritmus" fogalmát, és egyáltalán mik a képességei és korlátai ennek az új eszköznek?

Az ilyen jellegű kérdések felvetésében és megválaszolásában nagy szerepe volt egy britt matematikusnak, ***Alan Turing***–nak (1.1 ábra). Ő alkotta meg azt a matematikai modellt — a róla elnevezett ***Turing–gépet*** —, amely lehetővé tette, hogy a matematika eszközeivel vizsgálják a számítógépekkel kapcsolatos problémákat.

Mára már nagyon sokat tárt fel a kutatás ezzel kapcsolatban, azonban még így is vannak fehér foltok ezen területén. Manapság minden tudomány esetében az alapkutatások egyre inkább háttérbe szorulnak. Ez egyenes következménye annak, hogy a tudomány gazdasági erőforrássá vált, és nyilvánvalóan azok határozzák meg a kutatás irányát, akik az anyagi forrásokat adják hozzá.

1.1. ábra. Alan Turing 1912–1954

Ennek a dolgozatnak deklarált célja az, hogy az elkövetkezendő oldalakon felhívjuk a figyelmet az elméleti módszerek jelentőségére. Pontosabban arra szeretnénk felhívni a figyelmet, hogy az oktatás során mennyire fontos hangsúlyt helyezni az eméleti módszerekre.

A számítástudomány alapjainak megközelítése több irányból is lehetséges: a Turing–gépek, a formális nyelvek, a sejtautomata algoritmusok, a rekurzív függvények mind egy lehetséges alternatíva. Igazából mindegy, hogy honnan indulunk el, mert Church tézise értelmében — mint látni fogjuk — bármely modell ekvivalens a Turing–gépekkel, ezért ugyanolyan bonyolultság fogalomhoz és velük kapcsolatos problémákhoz fogunk eljutni.

Éppen ez az oka annak, hogy mi a fentebb felsorolt lehetőségek közül csak egyet ragadunk ki és tárgyalunk részletesen, méghozzá a Turing–gépnek nevezett koncepció és a hozzá kapcsolódó kérdések azok, amely dolgozatunk középpontjában állnak.

A második fejezetben röviden áttekintjük, hogy melyek voltak a főbb történeti állomásai a számítástudomány kialakulásának. A harmadik fejezetben megvizsgáljuk azt, hogy milyen a számítógépek logikai és struktúrális felépítése, és részletesen definiáljuk azokat a fogalmakat, amelyek a Turing–gépekkel kapcsolatosak, és azok megértéséhez elengedhetetlenül szükségesek.

Ezután megnézzük, hogy az elméleti modell, vagyis a Turing–gép milyen kapcsolatban áll a valódi számítógépekkel, vajon tényleg tud–e annyit, mint ez utóbbiak.

Ennek kapcsán vizsgáljuk meg részletesen — szintén ebben a fejezetben — az imént már említett Church tézist. Bizonyítás nélkül kimondunk néhány tételt is, főleg olyan tételeket, amelyek a kiszámíthatósággal kapcsolatban valamilyen érdekes dologra hívják fel a figyelmet. A bizonyításokkal azért nem foglalkozunk részletesen, mert azok egyáltalán nem nehezek és nem bonyolultak, de gyakran igen hosszadalmasak, és ez azt indokolja, hogy terjedelmi okok miatt mellőzzük őket és csak hivatkozásokat közöljünk velük kapcsolatban, hogy azok hol találhatóak meg. Ha valakit a bizonyítások is érdekelnek, azoknak az irodalomjegyzékben szereplő [P] és [LL] valamint [RISz] könyveket ajánljuk, amelyek összefüggéseikben és nagy részletességgel tárgyalják azokat.

A negyedik fejezet a Turing–gépek irányított gráfokkal való leírását vizsgálja. Ebben segítségünkre lesz egy olyan — a szerző által készített, *Automaton* nevű — program, amely szemléletesen mutatja be a Turing–gépek és a gráfok kapcsolatát. Az automaták gráfokkal való leírása azért is szerencsés megoldás, mert lehetővé teszi, hogy bizonyos Turing–gépekkel kapcsolatos, egyszerűbb algoritmuselméleti problémákhoz megtaláljuk a velük ekvivalens gráfelméleti problémát és fordítva. Másrészt ebben az esetben a gráf azt

a szerepet tölti be, mint a folyamatábra, és mivel a legtöbb ember vizuális típus, nagyban segíti a megértést.

Ennek a fejezetnek a második fele az *Automaton* nevű program leírását és használatát tárgyalja. Ennek a progamnak a segítségével modellezhetjük a Turing–gépeket, és ezáltal felhasználhatjuk azt a Turing–gépek vizsgálatára és tanulmányozására.

A program tulajdonképpen arra a koncepcióra épül, hogy konkrét programozási nyelvektől függetlenül is lehet programozni. Hasznos, ha az iskolában oktatott magasszintű programozási nyelvek mellett a diákok megismerkednek egy olyan módszerrel is, amellyel ugyanúgy algoritmusokat lehet megvaslósítani, mint a programnyelveken, de a számítógépektől függetlenül. A program talán segít a diákoknak annak megértésében, hogy az algoritmus egy általánosabb fogalom, mint azt általában gondoljuk, és ezt a fogalmat nem feltétlenül kell összekapcsolni a számítógépekkel. A számítóeszközök ötlete néhány száz éves, viszont algoritmusok már több ezer éve léteznek.

A magasszintű programnyelveket azért találták ki, hogy az emberi gondolkodáshoz nagyon közeli módon lehessen megfogalmazni a számítógépek számára az algoritmust. Azonban a legtöbb olyan fogalom amelyet a magaszintű nyelvekhez kapcsolunk — procedurális– és moduláris absztrakció, ismétlések, feltételes elágazások stb. — nem a programnyelvekhez kötődnek elsősorban, hanem az algoritmizáláshoz általában. Sajnos ez nem minden esetben kap elég hangsúlyt a mai oktatásban. A Turing–gépek és a tárgyalásra kerülő modellező program talán segíthet abban, hogy az oktatás során valamilyen formában érzékeltesse az algoritmus fogalmának igazi jelentését, és segítse az algoritmusok szerkezetének tanulmányozását.

2. fejezet

A „számítások tudományának” kialakulásáról

Bár senki nem tudhatja teljes bizonyossággal, de minden valószínűség szerint a matematika a legősibb tudomány, amelyet az ember művel. A dolgok megszámlálásának igénye és szükségessége annyira magától értetődő dolog, hogy a számolás egyidős magával az emberrel. Elmondhatjuk, hogy amióta ember van a Földön, azóta van matematika is.

Tudjuk azt például, hogy a számfogalom már a kőkorszaki ember által is ismert dolog volt. Különböző tudományágak képviselői — antropológusok, szociológusok, pszichológusok, nyelvészek — azt is kimutatták, hogy hogyan is működhetett a számolás az ősembernél. Kezdetben nem voltak számok, hanem csak az egy, kettő és sok között tettek különbséget. Később alakult ki a többi szám fogalma, és a számrendszerek.

Jelenlegi tudásunk szerint mi emberek vagyunk az egyetlen olyan lények, akik képesek vagyunk, a körülöttünk lévő világ dolgaiban rejlő mennyiségi összefüggéseket elvonatkoztni maguktól a dolgoktól, és számok formájában kifejezni a valóságot, amely körül vesz bennünket. A számokkal viszont műveleteket lehet végezni, a műveletek ilyen értelemben pedig nem mások, mint eljárások a valóság gondolati módosítására. Ezt az alapelvet alkalmazzák már évezredek óta a különféle tudományok akkor, amikor a matematikát nyelvként használják fel. A fizika, a kémia, és a műszaki tudományok, de a társadalom tudományok is a matematika nyelvén fejezik ki tételeiket, törvényeiket, sejtéseiket stb.

Minden korban és minden nép használta tehát a számokat. Az ember mindig tisztelettel, és csodálattal fordult a matematika felé, mert lenyűgözte, hogy az mennyire alkalmas eszköz a természet törvényeinek leírására és rejtett titkainak megragadására. A csillagász Galilei is ezt a vélekedést vetette papírra a XVI. század környékén, amikor egyik művében a következőket írta:

„*A természet könyve a matematika nyelvén van megírva, amelynek a betűi háromszögek, körök és más mértani alakzatok; ezen eszközök nélkül lehetetlen az ember számára, hogy akár csak egy szót is megértsen belőle.*”

Manapság már nehéz olyan tudomány területet mondani, amely ne használná fel valamilyen formában a matematika eredményeit, akár közvetlenül, vagy közvetett módon. Még a humán- és társadalom tudományoknál is így van ez, hiszen például a szociológusok és pszichológusok kísérleteik eredményeit statisztikai eszközökkel elemzik, de igaz ez az orvostudományi és gyógyszerészeti kutatásokra is. Az iparban és műszaki tudományokban egyre általánosabb dolog, hogy a drága és magas költséggel járó kutatásokat, kísérleteket nem végzik el a valóságban, hanem annak matematikai modellje alapján számítógépes szimulációkat hajtanak végere.

Már a kezdetektől fogva meg volt az igény arra, hogy a matematikai számítások elvégzését valamilyen formában megkönnyítsék. Számító- és számolást segítő eszközöket szinte minden korban és kultúrában találunk. Kezdetben a kéz ujjai voltak a segédeszközök, később különféle botok, pálcák, kavicsok, csomózott zsinórok stb. Bizonyára mindenki ismeri az *abakuszt*, vagy annak ma is nagy népszerűségnek örvendő japán változatát a *szorobánt*. Ezt az eszközt már az ókorban is ismerték és minden valószínűség szerint Mezopotámiából származik.

A XVII. század közepén az ismert matematikus–filozófusokat Blaise Pascal[1]–t (1623–1662) és Gottfried Wilhelm Leibniz–et (1646–1716) élénken foglalkoztatta a matematika automatizálásának és számítóeszközöknek a problémája. Mindketten építettek mechanikus elven működő számológépeket, amelyek az alapműveletek elvégzésére voltak alkalmasak.

Az informatika történet jelentős eseményként tartja számon az 1833–as esztendőt, ugyanis ekkor Charles Babbage (1792–1871) brit matematikus és feltaláló megalkotta az általa *Analytical Engine*–nek nevezett gép tervét, és ezzel kidolgozta a modern digitális, programozható számítógép alapelveit. Sajnos az *Analytical Engine* anyagi források hiányában és a kor finommechanikai lehetőségeinek elmaradottsága miatt nem készült el[2], és ezért a matematika igazi automatizálásának problémája továbbra is megoldatlan maradt.[3]

A nagy áttörést a számítógépek fejlődésében az elektromosság felfedezése

[1] Az ő tiszteletére nevezték el a Pascal programozási nyelvet.

[2] Ha megépült volna, egy futballpálya területét foglalta volna el és öt gőzgép energiája kellett volna a működtetéséhez abban az időben. Viszont a londoni *Science Museum* 1991–ben megépítette az első teljes differenciál gépet Babbage születésének kétszázadik évfordulója alkamából. Ez több, mint négyezer részből áll, a súlya pedig több, mint két tonna.

[3] Az informatika és a számítógépek történetéről részletes leírást találhatunk az irodalomjegyzékben szereplő [URL1] web oldalon.

és a vele kapcsolatos tudományos–technikai fejlődés jelentette. 1946-ban a magyar Neumann János hozzá kezdett csoportjával a princetoni *Institute for Advanced Studies* intézetben egy új tárolt programú számítógép tervezéséhez. Ez volt az IAS[4] névre keresztelt gép, amely az összes többi, későbbi általános célú számítógép prototípusának tekintenek. Azonban itt sem maga a gép az, ami fontos, hanem az elv: a Neumann–elv.

A számítógépek technikai fejlődése maga után vonta egy másik tudományterület fejlődését is. Ez a tudományterület előtte nem is nagyon létezett, vagy legalábbis nem volt olyan fontos, azonban az 1950–es és 1970–es évek között lendületes fejlődésnek indult, és mára egyáltalán nem tekinthető újnak. Éppen ezért meglepő, hogy mind a mai napig nincs egységesen elfogadott konvenció az elnevezésére. Sokféle elnevezéssel találkozhatunk a szakirodalomban: nevezik ***számítástudománynak***, ***számítástudomány alapjainak***, ***algoritmus elméletnek***, ***programozás elméletnek***, ***bonyolultság elméletnek***, és még sok másnak is. A későbbiekben ezeket az elnevezéseket felváltva fogjuk használni, de az elnevezés nem lényeges, az viszont annál inkább, hogy mivel is foglalkozik a számítástudomány.

Ha jobban meggondoljuk a dolgot, akkor az ***egész matematika tulajdonképpen nem más, mint algoritmusok összessége***. Már az alapműveletek is, amelyeket még kisgyermekként az általános iskolában megtanulunk algoritmusok. ***Olyan végesszámú, előre meghatározott lépések sorozata, amelyek valamely probléma megoldásához vezetnek***.

A legtöbb ember, ha megkérdezzük őket, hogy szerintük mi is az az algoritmus, akkor valami hasonló meghatározást fognak adni, mint ami az előbbi mondatban szerepel. Ezt szokták az algoritmus naiv definíciójának nevezni. Naiv azért, mert nélkülözi a matemaikai precízséget, de vegyük észre, hogy minden matematikai eljárásra érvényes az alapműveletektől kezdve a másodfokú egyenlet megoldó képletén és a determináns számításon át egészen a differenciál egyenletek megoldásáig. ***Mert minden algoritmus!***

Az algoritmusok megtalálása egy–egy konkrét probléma esetében nem mindig egyszerű feladat. Azonban ha már egy probléma megoldására valaki megtalálta az algoritmust, akkor azt mindenki alkalmazhatja, mert az algoritmusokban az a jó, hogy végrehajtásuk általában nem igényel szakértelmet az adott területen. Nekem nem kell mindig tudnom, hogy egy algoritmus miért működik. Elég ha azt tudom, hogy működik, és hogyan kell végrehajtani.

Joggal merül azonban fel a kérdés, hogy minden esetben és minden problémára létezik–e algoritmus. Ezt a kérdést tették fel azok a matematikusok is a XX. század elsőfelétől kezdve egészen napjainkig, akik munkássága nagyban hozzájárult az algoritmus fogalmának tisztázásához.

[4]Nevét az intézet kezdőbetűiről kapta.

Mint említettük, az algoritmus elméletnek a számítógépek fejlődése és elterjedése adott lendületet. Természetesen a problémák megoldhatóságát a számítógépek megjelenése előtt is sokan vizsgálták, azonban ez nem volt annyira a köztudatban akkoriban. A számítógépek és az algoritmus elmélet összefonódása leginkább a fizika és az analízis összefonódásához hasonlítható.[5] Amikor Isaac Newton (1643–1727) angol fizikus felállította híres elméletét, szembesülnie kellett azzal a ténnyel, hogy az akkori matematikai ismeretek nem tökéletesen felelnek meg az elmélet leírására. Ezért kidolgozta — kortársától, Leibniztől függetlenül — az infinitezimális számítás alapjait, amelyet ma, közismertebb nevén differenciál– és integrálszámításnak nevezünk. Itt tehát a fizika adott lökést a matematika fejlődésének.

Hasonló a helyzet a számítástudomány és a számítógépek fejlődése közti kapcsolat esetén is. Ahogyan fejlődtek, és egyre gyorsabbak lettek a számítógépek, egyre inkább kitágult a számítógéppel reális idő alatt megoldható problémák horizontja, és ezzel együtt egyre újabb és újabb elméleti problémák merültek fel, amelyek korábban vagy egyáltalán nem voltak ismertek, vagy ismertek voltak ugyan, de mivel nem volt gyakorlati jelentőségük, érdemben senki nem foglalkozott velük.

Az 1980–as évek elejétől a számítógépek amúgy is gyors fejlődése exponenciálisan felgyorsult. Köszönhető ez az Amerikai Egyesül Államokban lévő, Szilícium–völgy néven ismert mikroelektronikai–ipari központban zajló kutató–fejlesztő tevékenységnek, amely oroszlán részt vállalt a számítógépek legfontosabb alkatrészének, a processzornak a fejlesztésében.

Bizonyára mindenki előtt ismert az a jóslat, hogy a számítógép processzorok teljesítménye rendszeresen, bizonyos időszakonként meg fog kétszereződni.[6] Az előbbi mondatban a „bizonyos időszakonként” kifejezés helyére mindenki más és más számot szokott behelyettesíteni. Kezdetben három évekről beszéltek, aztán évekről, ma pedig — talán nem túlzás – már néhány hónap is elegendő ehhez.

Felmerül azonban a kérdés, hogy vajon meddig fog ez tartani. Vajon korlátlan a fejlődés lehetősége? Nehéz megmondani, igazából mind a mai napig még senki nem tudott sem igennel, sem nemmel felelni erre a kérdésre. Nap-

[5]Vö. Staar Gyula: A matematikus és a számítógépek c. cikkét, amelyben Lovász Lászlóval beszélget az Élet és Tudomány 2003/26. számában.

[6]Ez az úgynevezett Moore–szabály, amelyet Gordon Moore az Intel cég egyik alapítója állított fel még 1965–ben. Ezt a szabályt egyébként a közgazdaság tudomány is ismeri, ott „bűvös kör” szabályként emlegetik. Lényege, hogy a technológiai fejlődés olcsóbb termékeket eredményez. Az ezekre a termékekre épülő alkalmazások újabb piacokat nyitnak, amelyekre a vállalatok rávetik magukat és megindul a gazdasági verseny. Ez a verseny pedig kikényszeríti az olcsóbb, és minőségi termékek előállítását, vagyis ösztönzi a technológiai fejlődést. Ezen a ponton pedig a kör bezárul, és kezdődik minden elölről. Vö. [TA], 40. o.

jainkban azonban egyre inkább felerősödnek azok a vélemények, amelyek azt hangoztatják, hogy a kezdeti, és egészen máig tartó lendület kezd kifulladni.

Itt megint mindenki máshol húzza meg a határt. Vannak akik egy-két éven belülre jósolják mindezt, mások tízévekről beszélnek. Ebben az esetben igazából a technológiai problémák adnak okot a kételkedésre. A mai leggyorsabb processzorokban a szilícium rétegek, amelyek az áramköröket alkotják, néhány atomréteg vastagságúak. Mivel az atom az a legkisebb egység, amely még az anyag minden tulajdonságát hordozza, nyilvánvaló, hogy egy atomréteg alá nem lehet menni. Ráadásul egy processzor teljesítményének megkétszerezése és az előállítási költségek között nem lineáris kapcsolat van, hanem egy kétszer olyan gyors processzor előállítása általában négyzetesen többe kerül.

A fejlődés lelassulásával kapcsolatban ellenérvként azt szokták felhozni, hogy nem az általános fejlődés fog lelassulni, hanem csak a jelenlegi (elektronikus elven működő) technológia az, amely kezdi elérni lehetőségei végső határát, de majd más, új technológiák képesek lesznek túllépni a problémákon. Ebben minden kétséget kizáróan van igazság. Számos kutatás folyik új technológiák után. Némelyek kvantum számítógépekről, mások molekuláris méretű logikai áramkörökről beszélnek,[7] és a lehetőségek határát senki nem ismeri.

Sokan vannak azonban, akik, ha csak elméleti síkon is, de vitatkoznak az iménti kijelentéssel. Van egy olyan határ, amely mégis csak gátat szab a lehetőségeket. A későbbiekben definiálni fogjuk a Turing–gép modellt. Mint látni fogjuk ennek a számítóeszköz koncepciónak egyik legfontosabb jellemzője a belső állapotok halmaza, és az hogy a belső állapot a számítás során lépsről–lépésre változik. Ez a belső állapot változás azonban, ha a Turing–gép fizikai ralizációját tekintjük, akkor minden esetben valamilyen fizikai állapot változást jelent. Konkrétan az elektromosság elvén működő személyi számítógépek esetében például a 0–1, igen–nem, van áram–nincs áram váltakozását.

Nehéz elképzelni olyan (számító)gépet, amelynek a belsejében soha semmilyen változás nem zajlik le, hanem minden fizikai paraméter állandó. Nem nyilvánvaló, hogy egy ilyen eszközzel hogyan lenne lehetséges számításokat végezni.

[7] Az Élet és Tudomány 2002/47. számában érdekes cikket olvashatunk arról, hogy az IBM kutatóközpontjában sikerült szén–monoxid molekulák egymás mellé rakásával a mai félvezető csipek logikai elemeinél 260 ezerszer kisebb méretű szerkezetet összeállítani, amellyel sikeresen szimulálták a logiaki ÉS–kapu illetve a logikai VAGY–kapu működését. A cikkben elolvashatjuk, hogy még a kísérlet során felépített legbonyolultabb áramkör is olyan parányi, hogy egy 7 milliméter átmérőjű ceruzaradír tetején is 190 milliárd darab férne el belőle.

Azonban a fizikai állapot változásoknak — legalábbis jelenlegi fizikai tudásunk szerint — van egy abszolút felső határa. Nevezetesen az, hogy semmilyen fizikai hatás nem mehet végbe fénysebességnél gyorsabban. Elviekben tehát nem létezhet tetszőlegesen gyors számítóeszköz, hanem létezik egy olyan számítógép, amelynél gyorsabbat nem lehet építeni, mert a körülöttünk lévő világ természeti törvényei olyanok, hogy ezt nem teszik lehetővé.

Természetesen a fenti eszmefuttatás pusztán elméleti jellegű, és jelen pillanatban nincs olyan mérnök, fizikus, matematikus, vagy bármilyen más komolyan vehető tudós, aki bármit is tudna mondani arról, hogy a technika mennyire közel illetve távol van ennek a gépnek a megépítésétől. Valószínűleg nagyon messze, vagy az is lehet, hogy soha nem fog eljutni eddig a fejlődés, de ezt a problémát meghagyjuk a mérnök kollégáknak, had törjék rajta ők a fejüket, miközben a jövő processzorait tervezik.

2.1. Hardver vagy szoftver?

Számunkra egyetlen egy dolog lényeges az elmondottakból ***A hardver teljesítményének növelésével nem győzhetünk le minden nehézséget. Szükség van szoftveres gyorsításra is, vagyis nem nélkülözhetjük az elméleti módszereket.***

Életem első találkozása a számítógépekkel az volt, amikor az 1990–es évek elején általános iskolai szakkör keretében elkezdtünk COMMODORE16–os gépeken BASIC–et programozni. Látszólag azóta nagy fejlődés ment végbe. Például az akkori fekete–fehér ORION televíziót, amit monitorként használtunk mára felváltották a színes monitorok, és nem vitatható, hogy a mai gépek gyorsabbak, mint a COMMODORE–ok voltak.

Azonban mégis van egy nem elhanyagolható szempont, ami nem változott, és amely közös a korábban már említett Charles Babbage féle *Analytical Engine*–ben, a COMMODORE–ban és abban a PENTIUM4®–es számítógépben, amelyen ezt a dolgozatot most éppen írom. Technikai paramétereik ugyan nagyban különböznek, azonban mindegyik a problémák ugyanazon osztályának megoldására alkalmas. A mai modern gépek semmivel sem tudják a problémák szélesebb osztályát megoldani, mint bármely elődjük. Ennek pedig nagyon egyszerű oka van. Nevezetesen az, hogy ***mindegyik a Turing–gépnek nevezett modell egy–egy fizikai realizációja, és éppen ezért minden, ami érvényes a Turing–gépekre, érvényes bármely számítógépre is***. Ha egy probléma nem oldható meg Turing–géppel, akkor semmilyen más számítóeszközzel sem. Legalábbis ezt állítja a Church–tézis néven ismert posztulátum, amelyet a későbbiekben még — több különböző formában is — részletesen fogunk tárgyalni.

Viszont az nem vitatható, hogy a COMMODORE korszakban, és azt megelőzően a programozóknak sokkal ügyesebbnek és trükkösebbnek kellett lenniük, mint manapság, amikor a memóriát már megabájtokban, a processzor sebességét pedig gigahertzekben mérjük. Az akkori technológiai megoldások rákényszerítették a programozókat arra, hogy kifejlesszenek olyan programozói fogásokat és algoritmusokat, amelyek kitalálása ugyan nem ment egykönnyen, de az lett az eredményük, hogy a szűkös memória kihasználtsága 110 %–ra ugrott.

Manapság már más szemlélet uralkodik. Ha kevés a memória, akkor sokkal könnyebb elmenni a sarki szaküzletbe, és venni egy új memória modult, mint szakkönyvekben, vagy az INTERNET–en olyan algoritmusokat keresni, amelyek implementálása ugyan sokkalta nehezebb, de a memória igénye negyede annak, mint amit egy másik algoritmus megkövetel.

Ha pedig egy feladat megoldása lassúnak tűnik, és az a bosszantó kinézetű homokóra sokáig forog a képenyőn, akkor — gondolják sokan — egyszerűbb felemeli néhánnyal a gigahertz–ek számát, mint átgondolni az alkalmazott módszert. A meglepetés csak akkor következik be, amikor mondjuk valaki egy 1024 bites RSA kulcsot szeretne valaki faktorizálni, mert akkor nem hogy a gigaherc–ek, de még a terra–, peta–, és exahertzek sem jelentenek megoldást. ***És ma már rengeteg ehhez hasonló nehézségű problémát ismerünk, nem is beszélve arról a számtalanról, amelyekre egyáltalán nem létezik megoldó algoritmus. Ebben esetben a „számtalan” találó kifejezés, mert a problémák többen vannak, mint a lehetséges megoldó algoritmusok. Előbbiek számossága kontinuum, míg utóbbiak megszámlálhatóak.***

Ennek a dolgozatnak a szerzője vitatja, hogy minden esetben a fenti módszer lenne a célravezető megoldás és a követendő példa. Programozni tudni, és programozási nyelveket ismerni két, lényegileg különböző dolog. Programozni megtanulni hosszú évek munkája. Egy konkrét programozási nyelvet megtanulni — ha rendelkezésre áll az adott nyelvről egy jó referencia leírás —, néhány nap.

Ahhoz viszont, hogy ezt megértsük, tisztázni kell, hogy mit jelent ez a szó: *algoritmus*. Pontosan ez a kérdés az, amely életre hívta kevesebb, mint egy évszázaddal ezelőtt a „*számítások tudományát*”, és mi is ezt kutatjuk ebben a dolgozatban.

3. fejezet

A számítástudomány alapjai

Ebben a fejezetben részletesen definiáljuk azokat a fogalmakat, amelyek az egész tárgyalás alapját képezik. Erre azért van szükség, mert — mint látni fogjuk — több olyan koncepció is létezik, amelyekre ráragasztják a bélyeget, hogy Turing–gép. Igazából ezekről a modellekről könnyen be lehet látni, hogy számítási erejük megegyezik, és egyik sem tud többet a másiknál[1]. Valójában csak „ízlés dolga", hogy melyik szakirodalom mit nevez Turing–gépnek. Mi a továbbiakban az irodalomjegyzékben szereplő [SZ0] és [SZ1] könyvekben lévő modellt fogjuk használni[2], mi erről állítjuk, hogy EZ A TURING–GÉP.

3.1. Neumann–elvű számítógépek

A legtöbb szakirodalom általában először definiálja a Turing–gép fogalmát, majd ezek után röviden vázolja azokat az elméleti megfontolásokat, amelyek elegendő indokot szolgáltatnak arra, hogy a valódi, fizikailag is létező számítóeszközöket ezen elméleti modell egy–egy raelizációjának tekintsük, vagyis minden, ami érvényes a modellre, érvényes az asztalunkon lévő gépre is.

Mi most egy fordított utat fogunk bejárni. Először a valódi számítógépekből kiindulva elemezzük azok fizikai és logikai struktúráját, és működésük alapelvét, azután pedig azt fogjuk meg vizsgálni, hogy miként lehetne ezt a koncepciót egyszerűsíteni, vagyis melyek azok az elemei a modellnek, amelyek nem feltétlenül szükségesek abban, és ezért elhagyhatóak, vagy más megoldásokkal helyettesíthetőek.

Természetesen minden esetben szem előtt kell tartanunk azt az alapelvet, hogy csak akkor hagyhatjuk el ezeket az elemeket, ha az ily módon nyert

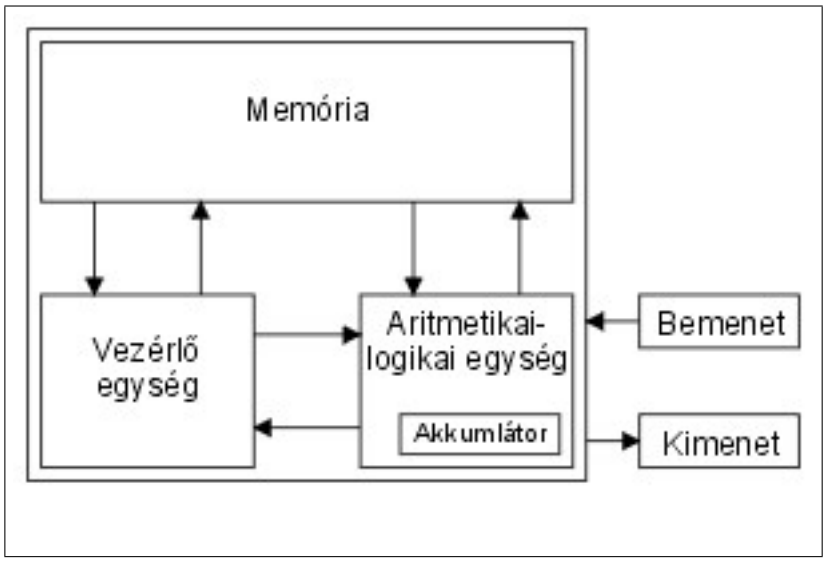
[1] Ezt úgy kell érteni, hogy legfeljebb polinomiális időveszteség árán az egyik képes szimulálni a másik működését.

[2] Némi elhanyagolható módosítással, amelyeket a megfelelő helyen részletesen kifejtünk.

szimplifikált modell számítási ereje megegyezik a kiindulásként szolgáló modell számítási erejével.

Ahogyan egyre inkább egyszerűsítjük a számítógépek szerkezetét, előbb utóbb el fogunk jutni egy olyan modellhez, amelyet már eléggé egyszerűnek ítélhetünk ahhoz, hogy a matematikai és logikai tárgyalás alapjául szolgáljon. Mint látni fogjuk, ez a modell (pontosabban az egyik lehetséges ilyen modell) a Turing–gép lesz.

A Neumann–elvű tárolt programú digitális számítógépet tekintik az összes ma használt, modern számítóeszköz alapjának. Az iménti mondatban minden olyan fontos jelző benne foglaltatik, amelyek miatt a Neumann által kigondolt koncepció többet jelentett elődeinél, és messze felülmúlta a korábbi számítógépeket. ***A két legfontosabb kulcsfogalom a „digitális" és a „tárolt programú".*** Az elméleti modell konkrét változata az IAS nevű gép volt, amelyet Neumann két munkatársával Athur W. Burks–szal és Hermann H. Goldstein–nel 1952–ben épített meg a princetoni *Institute for Advanced Studies* intézetben. A gép felépítésének vázlatát a 3.1 ábrán láthatjuk. Az igazsághoz hozzá tartozik, hogy a digitális számítógép ötlete,



3.1. ábra. A Neumann-elvű gép vázlata

már Neumann–t megelőzően is létezett, sőt mi több, a korábban már említett Charles Babbage volt az ötletgazda, még az 1830–as években. Azonban Neumann idejében a legtöbb gép analóg elven működött, illetve — igazság szerint — az analóg és digitális technikát (elektronikai–logikai áramköröket) ötvözte valamilyen formában.

3.1.1. Az analóg elv

Az analóg számítógépek lényege, hogy a számokat és mennyiségeket valamilyen fizikai paraméter nagyságával ábrázolják, oly módon, hogy az adott paraméter meghatározott egységekre vonatkozó értéke egyenlő legyen a számmal. Ez a paraméter lehet például az áramerősség vagy a feszültség nagysága, vagy azoknak az ívfokoknak a száma, amellyel egy bizonyos tárcsa elfordul a tengelye körül stb.

Másik fontos jellemzőjük ezeknek az analóg gépeknek, hogy a használt fizikai paraméter, az előre megadott minimális és maximális értéke között lévő tetszőleges értéket felvehet, vagyis folyamatosan változhat az értékhatárok között, és nem csak diszkrét értékekei lehetnek, ez utóbbi ugyanis majd a digitális gépekre lesz jellemző. Ha például feszültség értékeket használunk és az egységnek a mV–ot választjuk, akkor a 2004 szám úgy jelenik meg a gépben, hogy egy maghatározott vezetékben $2004mV = 2.004V$ a feszültség érték.

Az ilyen elven működő analóg gépek a műveleteket — összeadást, kivonást, szorzást és osztást — úgy hajtják végre, hogy valamilyen olyan folyamatot indítanak el, amely a kívánt módon hat a mennyiségeket ábrázoló fizikai paraméterekre. Például áramerősségeket úgy lehet kivonni egymásból, hogy egy változtatható nagyságú ellenállást kapcsolunk az áramkörbe, amely nagyságát az az áramerősség szabályozza, amely a kivonandó szám értékét jeleníti meg.

Ezek a gépek tehát attól lesznek analógok, hogy valamilyen fizikai változás hasonlatosságára[3] állítják elő az elvont matematikai mennyiségek változásait.

Az analóg elven működő gépeknek van egy sajnálatos hátrányos tulajdonságuk. Mivel a számokat fizikai mennyiségekkel ábrázolják olyan körülményeket kellene teremteni, hogy ha az adott paraméter fel vett egy meghatározott értéket, akkor azt meg is tartsa amíg szükséges — esetenként akár hosszabb időn keresztül is —, és ne változzon meg egyéb, külső tényezők hatására. Sajnos olyan szerkezetet építeni, amelyben ezek a körülmények fenn állnak egyáltalán nem problémamentes, ugyanis tudjuk, hogy minden csatornán, amely információt tárol, vagy közvetít, jelen van egy bizonyos nagyságú zaj, amely véletlenszerű torzulásokat idézhet elő.

Önmagában a zaj még jelentene problémát, akkor ha volna valamilyen olyan módszer, amellyel el tudnánk távolítani a végeredményként kapott értékből. Ilyen módszer azonban sajnos nem ismeretes, ezért ha kapunk egy eredményt, akkor nem tudjuk meghatározni, hogy abban milyen mértékben

[3] *Analógia*=hasonlóságon alapuló egyezés.

van jelen a minket érdeklő szignifikáns eredmény és mennyi a szerkezet véletlen fluktuációja következtében ráterhelődött zaj.

Ez tehát azt jelenti, hogy nem létezik olyan analóg gép, amely két szám szorzatát, hányadosát, összegét stb. állítja elő, hanem amit előállít, az az eredmény plussz–mínusz még valami, ami a fizikai folyamatok ingadozása miatt lép fel.

Mivel a zaj minden berendezésben jelen van, és nem tudjuk megszüntetni, az alapvető kérdéssé az lesz, hogy miként tudjuk a lehető legteljesebb mértékben minimalizálni. Ez a konkrét berendezések esetében különféle technikai megoldásokkal lehetséges, amelyek csökkentik ugyan a zaj nagyságát, de soha nem szüntetik meg teljesen.

3.1.2. A digitális elv

A analóg technikával szemben a digitális számítógépek jelentik a megoldási alternatívát, amelyek a „mindent vagy semmit" elvén működnek. Ez konkrétan azt jelenti, hogy míg az analóg gépek egy fizikai paraméter nagyságával kódolják a számokat, addig ***a digitális gépek egy fizikai paraméter meglétével, vagy meg nem létével ábrázolják ugyanezeket a mennyiségeket. Maga az ábrázolás ugyanolyan módon történik, mint amikor mi leírjuk a számokat egy papírra kézi számolásnál, vagyis számjegyek sorozataként.***

Ebben az esetben is fizikai paraméterek jelenítik meg a számokat, azonban ezek most csak meghatározott, diszkrét értékeket vehetnek fel, vagyis nem lehetnek bármekkorák. Például ha a $+3V$ feszültségértéket megfeleltetjük a 0–nak a $+5V$–ot pedig az 1–nek, akkor a vezetékben lévő feszültség csak ezen két érték valamelyike lehet, és elméletileg semmilyen körülmények között nem vehet fel más értéket.

Azonban ez tényleg csak elméletileg van így, mert a zaj a digitális berendezéseket is érinti, és ezek működésére is hatással van, viszont ebben az esetben sokkal könnyebb kiküszöbölni és kezelni, mint az analóg gépeknél. Ez azon alapul, hogy általában tudunk valamilyen feltevéssel élni a zaj nagyságát illetően, vagyis meg lehet becsülni, hogy maximálisan mekkora annak nagysága. Ezek után már könnyen lehet olyan digitális szerkezetet építeni, amely már kevésbé érzékeny ezekre a hatásokra. Egyszerűen úgy kell megválasztani a számábrázolásra használt paraméter két szomszédos szintjét, hogy a köztük lévő távolság nagyobb legyen, mint a várható zaj abszólútértékének kétszerese. Ebben az esetben ugyanis ha a jel torzulást szenved, akkor egyértelműen meg lehet mondani, hogy mi volt eredetileg, egyszerűen csak venni kell a hozzá legközelebb eső megengedett értéket.

Ma általánosan alkalmazott megoldás a digitális gépekben a bináris aritmetika, vagyis azoknak a diszkrét értékeknek a halmaza, amelyet a számábrázolásra felhasznált paraméterek felvehetnek kételemű.

Azonban Neumann idejében nem ez volt az általános, hanem más alapszámot, általában tízes számrendszert használtak.[4] Neumann nagy ötlete a digitális technikával kapcsolatban a tisztán bináris alapú aritmetika bevezetése volt, amelynek több előnyös tulajdonsága is van a decimális aritmetikával szemben.

Az egyik legtipikusabb előnye, hogy a kettes számrendszerben a legegyszerűbb a matematikai műveletek végzése, mert itt csak két számjegy van, és itt tűnik ki leginkább a matematikai műveletek logikai jellege[5] (3.2 ábra). Ezen utóbbi megállapításnak egyenes következménye, hogy sokkal

+	0	1
0	0	1
1	1	10

·	0	1
0	0	0
1	0	1

3.2. ábra. A bináris aritmetika műveletáblái

könnyebb olyan áramköröket építeni, amely a bináris aritmetikát valósítja meg, mint olyat, amely a decimálisat. Figyelembe véve Neumann korának elektronikai–technológiai fejlettségét, ennek a ténynek hallatlanul nagy a jelentősége. Ugyanis minél bonyolultabb egy áramkör, annál több alkatrész kell hozzá, és minél több az alkatrészek száma, annál nagyobb az esélye annak, hogy valamelyikük meghibásodik.

Márpedig akkoriban nem az volt az elsődleges szempont, hogy hatékonyan működő számítógépet építsenek, hanem az, hogy egyáltalán működő számí-

[4]Vö. [NJ] 166. o. és 265. o. Ne értsük félre a „digitális” szó jelentését. Ha manapság valamire azt mondják, hogy digitális, mindenki a kettes számrendszerű elektronikára asszociál, aminek valószínűleg az az oka, hogy mára már kizárólag a bináris logikát alkalmazzák a digitális gépekben. Azonban attól, hogy egy eszköz digitális elven működik, ez még nem feltétlenül jelenti azt, hogy kettes számrendszerben végzi a számításokat. Pusztán azt jelenti, hogy a számok kódolására számjegy sorozatokat használ, hiszen a szó jelentése is ez *digit*=számjegy az angol nyelvben. A számrendszer alapszámát az határozza meg, hogy a számábrázolásra használt fizikai paraméter hány megengedett értéket vehet fel. Ma általánosan két érték a megengedett: az igen–nem, 0–1, van áram–nincs áram. Azonban a megengedett értékek halmaza lehet tíz elemű is, ekkor a gép tízes alapú számrendszerben dolgozik. Például ha $+1V$ jelenti az 0–át, $+2V$ a 1–t,..., stb., $+10V$ pedig a 9–et, akkor a 2004 szám ábrázolása úgy történik, hogy négy vezeték közül az elsőben $+3V$ a feszültség, a másodikban $+1V$, a harmadikban $+1V$, a negyedikban pedig $+5V$. Ez a gép, bár tízes számrendszert használ, de digitális, hiszen a számokat számjegy sorozatokként jeleníti meg.

[5]Vö. [NJ] 267.–270. o.

tógépet építsenek, vagyis olyat, amely képes úgy üzemelni, hogy a nap 24 órájából kevesebb, mint 24 órát kelljen a gép javításával és karbantartásával eltölteni.[6]

3.1.3. A digitális gépek potenciális hibaforrásai

Mint már korábban említettük, a digitális gépek esetében a mindenhol jelenlévő zaj következtében fellépő véletlen torzulások nem jelentenek olyan nagy problémát, mint az analóg gépek esetében, mert megfelelő technikai megoldásokkal kiküszöbölhetőek. Azonban ez nem jelenti azt, hogy nem merülnek fel más jellegű nehézségek, amelyeknek ugyanúgy nem kívánatos következményei lehetnek, ha nem járunk el elég körültekintően.

A digitális modellre épülő gépeknél is fellép számos probléma, méghozzá olyanok, amelyek kiváltó oka nem kifejezetten a technikai megoldások tökéletlenségében keresendő. Ezek a problémák a véges számábrázolás következményei, vagyis konkrétan abból származnak, hogy a számokat csak véges pontossággal tudjuk megjeleníteni a számítógép memóriájában.

A véges tizedestörtek esetében ez csak akkor jelent nehézséget, ha a szám nem esik bele a rendelkezésre álló biteken ábrázolható legkisebb és legnagyobb szám által meghatározott intervallumba.[7] De mit kezdjünk az olyan számokkal, mint a $\sqrt{2}$, a $\sqrt{3}$, a $\lg 2$, a $0.10110111011110\ldots$, a π vagy az e és még hosszan sorolhatnánk.[8] Ezek a számok irracionálisak, vagyis végtelen, nem szakaszos tizedes törtek, tehát ha teljesen pontosan szeretnénk ábrázolni őket a számítógépen, akkor ehhez végtelen nagy memóriára van szükség, ami nyilvánvalóan képtelenség. Ezekben az esetekben csak az a lehetőség marad, hogy valamilyen algoritmus szerint kerekítjük a számokat, vagyis bizonyos (véges) számú tizedesjegyüket használjuk csak fel a számításainkban, a többit pedig egyszerűen figyelmen kívül hagyjuk.

Ezek a kerekítések azonban további potenciális hibaforrást jelentenek, ugyanis kellemetlen velejárójuk, hogy a belőlük származó hiba öröklődik a számítások folyamán, ráadásul nem egyformán érinti az egyes műveleteket, mert más a hiba nagysága akkor, ha két kerekített számot összeszorzunk, és más akkor ha összeadjuk őket (és természetesen az eredményt is kerekítjük).

[6] Vö. [NJ] 238. o. és 249. o.

[7] Általánosan igaz, hogy b alapú számrendszerben t darab számjegy (helyiérték) felhasználásával b^t darab (pozitív) számot lehet előállítani, amelyek közül a legkisebb a 0, a legnagyobb pedig $b^t - 1$.

[8] De még milyen hosszan, hiszen az irracionális számok halmaza még csak nem is megszámlálható, hanem kontinuum számosságú halmaz.

Vagyis nem teljesen mindegy, hogy milyen sorrendben hajtjuk végre a műveleteket, és melyikből hányat. Az $a \cdot \big(b + c\big) = a \cdot b + a \cdot c$ disztributív törvényt ezek szerint csak matematika órán állíthatjuk teljes bizonyossággal, amikor pedig programot írunk, akkor lehet, hogy az „=” jelet a „≅” jellel kell felcserélnünk, mert a két oldal csak közelítőleg egyenlő egymással.

Az ebből következő komplikációk nem különösebben feltűnőek abban az esetben, ha csak kis számú műveletet végzünk ugyanazokkal a számokkal. Azonban a számítógépek nagy erőssége éppen az iterációs számításokban, vagyis az ismétlésekben rejlik, amelyeknek alapvető szerepe van az olyan algoritmusokban, amelyek valamilyen rekurzív definícióval megadott lépéssorozatot foglalnak magukban.[9] A rekurzió lényege, hogy van egy kiindulási szám, úgynevezett inicializációs paraméter, amelyre alkalmazunk valamilyen műveleteket. Ezek után az eredményként kapott számra újfent alkalmazzuk ugyanazokat a műveleteket, vagyis az eredményt visszahelyettesítjük a függvénybe, és így tovább. Ha számítógéppel hajtjuk végre a számításokat, akkor minden egyes iterációs lépésben kerekítési hiba keletkezik, amely tovább öröklődik a következő lépésre, és mire a folyamat a végére ér, elképzelhető, hogy a kapott eredménynek semmi köze a valósághoz, mert a hibák olyan mértékben felhalmozódtak, hogy teljesen eltorzítják azt. Ebben az esetben azt mondjuk, hogy az algoritmus a kerekítési hibákra nézve instabil.

Nyilvánvaló, hogy az ilyen jellegű hibák teljesen függetlenek a számítóeszközök mindenkori technikai megvalósításától és annak esetlegességeitől, és a véges számábrázolás következményeiként állnak elő. Ez azonban egyben azt is jelenti, hogy nem tudjuk őket a fizikai hardver módosításával kiküszöbölni, mint tettük ezt korábban a zaj esetében. Azonban itt is lehetőség van arra, hogy behatároljuk, és algoritmusunk megfelelő elemzésével úgy módosítsuk a műveletek végrehajtásának sorrendjét, hogy ezzel minimalizáljuk az elkövetett kerekítési hibát. Mivel nem tartozik szorosan témánkhoz, ennek mikéntjét nem tárgyaljuk részletesen ebben a dolgozatban, azonban az irodalomjegyzékben szereplő [HF] könyv első fejezete behatóan foglalkozik az imént említett problémákkal és a hibaanalízissel.

A digitális gépekben tehát a hibák legtipikusabb forrása a kerekítés, azonban nem ez az egyetlen kritikus pont a rendszerben. Más tényezők is vannak, amelyek hibákat eredményezhetnek, és sajnos mivel ezek nem olyan nyilvánvalóak, felderítésük és kezelésük is sokkal nehezebb és problematikusabb.

Neumann gépe még nem ismerte a lebegőpontos számokat, de nem azért, mert nem tudták megoldani ezek ábrázolását, hanem azért, mert Neumann úgy gondolta, hogy minden valamirevaló matematikus fejben tudja tartani

[9] Igen gyakoriak például az olyan numerikus algoritmusok, amelyek esetében egy rekurzív definícióval megadott számsorozat határértéke adja a feladat megoldását.

a tizedesvessző helyét, ezért nem érdemes azzal bajlódni, hogy ezt külön jelöljük a gépben valamilyen formában.

A mai processzorok azonban számos adattípussal dolgoznak, vagyis attól függően, hogy milyen adattípusról van szó, ugyanaz a bitsorozat mást és mást jelenthet. A legtöbb konfliktust az okozza, amikor egyik adattípust át kellene konvertálni egy másikra, tehát például amikor azt próbáljuk meg a gép tudomására hozni, hogy ugyanazt a bitsorozatot amely eddig egy egész szám kódjaként kezelt, mostantól tekintse egy lebegőpontos szám kódjának.

Ha magasszintű programnyelven írunk programot, az egyes nyelvek más és más módon valósítják meg az ilyen típus konverziókat, de az ehhez hasonló műveletek gyakran járhatnak adatvesztéssel. Bizonyos programnyelvek nagyobb mértékben támogatják a típus átalakítást, mások pedig kevésbé. Például a C/C++ nyelv ebből a szempontból meglehetősen rugalmas, és nagy szabadságot ad a programozónak a különféle típus konverziókra; ezzel szemben a Pascal az úgynevezett erősen típusos nyelvek családjába tartozik, és nem engedi meg, hogy mondjuk egy egész számot csak úgy lebegőpontossá alakítsunk. Ha ezt akarjuk tenni, akkor külön szubrutinok állnak rendelkezésre ehhez, amelyek nyilvánvalóan kiszűrik a hibákat, de csökkentik a rugalmasságot.

A típus átalakításhoz hasonlóan potenciális hiba forrás lehet a túlcsordulás, vagyis az a szituáció, amikor egy művelet elvégzése során az eredményül kapott szám több számjegyből áll, mint amennyit az ábrázolására felhasználhatunk. Például két tízjegyű szám szorzata húsz jegyű szám lesz. De ha csak tíz jegyet használhatunk fel az ábrázolásra, akkor könnyen keletkezhet túlcsordulás. A részletekbemenő elméleti fejtegetés helyett vizsgáljuk meg ezt a problémát egy pár soros Pascal nyelven megírt kódrészleten keresztül! Nézzük meg figyelmesen az 3.3 valamint az 3.4 ábrákon látható Pascal eljárásokat.

Ha egy középiskolás diáknak, aki már tanult valamelyest programozni, feltesszük a kérdést, hogy mi lesz a különbség a két eljárás kimenete között, ha lefuttatjuk őket, valószínűleg rövid gondolkodás után rávágja, hogy semmi, hiszen a két algoritmus teljesen ugyanaz, mindössze a for–ciklus magjában lévő értékadó műveletben van másképpen implementálva a zárójelezés. Az 3.3 ábrán lévő alprogram először kiszámítja a ciklusváltozó reciprokát, majd az eredményt megszorozza önmagával; ezzel szemben az 3.4 ábrán lévő kódrészlet először négyzetre emeli a ciklusváltozót, és utána veszi annak reciprokát. Mivel $\frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n \cdot n}$ nincs értelme vitát nyitni arról, hogy a két algoritmus valóban ugyanazt a kimenetet produkálja.

Gyakorlati teszteléssel azonban könnyen meggyőződhetünk róla, hogy ez nem így van. Az első eljárást lefuttatva a kimeneten a 3.14159169866047

```
procedure GetPi1;
var
  P: Extended;
  n: LongInt;
begin
  P := 0;
  for n := 1 to 1000000 do
    begin
      P := P + ((1 / n) * (1 / n));
    end;
  P := Sqrt(6 * P);
  WriteLn(P);
end;
```

3.3. ábra. A π–t közelítő algoritmus (első változat)

```
procedure GetPi2;
var
  P: Extended;
  n: LongInt;
begin
  P := 0;
  for n := 1 to 1000000 do
    begin
      P := P + (1 / (n * n));
    end;
  P := Sqrt(6 * P);
  WriteLn(P);
end;
```

3.4. ábra. A π–t közelítő algoritmus (második változat)

szám jelenik meg.[10] Azonban a második eljárást futtatva az nem olyan viselkedést mutat, mint amilyet elvárnánk tőle. Attól függően, hogy milyen operációsrendszer alatt futtatjuk a programot különféle hibaüzeneteket kapunk: DOS operációs rendszeren „Runtime error 200 at FFFF:FFFF.", WINDOWS és LINUX rendszereken pedig a „Flaoting point division by zero." (vagy valami ehhez nagyon hasonló) szöveg jelenik meg a képernyőn.

Első látásra érthetetlennek és misztikusnak tűnhet egy ilyen hibaüzenet, hiszen ha jól megnézzük a kódot, akkor látható, hogy sehol sem vesznek fel a benne szereplő változók nulla értéket, viszont ennek ellenére a hibaüzenetek mégis arról informálnak, hogy nullával való osztás történt. Egy gyakorlott és tapasztalt programozó azonnal rájöhet a hiba okára, ami a változók típusának megadásában keresendő. A ciklus változó, vagyis az n típusa LongInt, tehát hosszúegésznek van deklarálva. Ez az adattípus konkrétan egy 32 bites előjeles számot jelent a PASCAL nyelvben, tehát értékkészlete a -2147483648..2147483647 tartományba esik. A for–ciklus 1–től 1000000–ig fut, azonban abban a pillantban, amikor a ciklus változó felveszi a $65536 = 2^{16}$–ot értékként, és ezt négyzetre emeli, az eredmény 2^{32} lesz, ami kettes számrendszerben leírva: $1\underbrace{00\ldots00}_{\text{32 db nulla}}$. Vagyis 33 bit kell az ábrázolásához, viszont nekünk csak 32 bit áll a rendelkezésünkre. Ilyenkor a processzor csak a megengedett 32 bitet veszi figyelembe, méghozzá azokat, amelyek sorszáma 0–tól a 31–ig terjed, és a számozás jobbról balra értendő, vagyis a fenti felírásban a legjobboldalibb számjegy a 0. sorszámú, hátulról a második az 1. sorszámú stb. Az összes többit egyszerűen figyelmen kívűl hagyja, mintha ott sem lennének. Mivel a 0..31 sorszámú bitek mind nullák, így már érthető, hogy miért történik nullával való osztás.

A hiba kiküszöbölésére szolgáló egyik lehetséges megoldás az, hogy a ciklus változó típusát olyannak deklaráljuk, hogy a négyzetreemelés eredménye is elférjen rajta, például az Extended lebegőpontos adattípus jelen esetben tökéletesen megfelel erre a célra. Ennek a megoldásnak az implementációját a 3.5 ábrán láthatjuk. Mivel a PASCAL nyelv a for–ciklus esetében csak egész típusú változó használatát engedi meg ciklusváltozóként, a for–ciklust itt lecseréltük és működését egy while–ciklussal szimuláltuk.

3.1.4. A digitális számítógépek előnye

Az analóg gépekkel kapcsolatban korábban azt mondtuk, hogy amit kiszámítanak, az az eredmény plussz–mínusz mégvalami, ami a zaj következtében

[10] Analízisből ismeretes, hogy $\lim_{k\to\infty} \sum_{n=1}^{k} \frac{1}{n^2} = \frac{\pi^2}{6}$. Vagyis a fenti algoritmus a π–t közelíti.

jelenik meg. Legutóbbi megfontolásaink alapján azonban nyilvánvalóvá vált, hogy a digitális gépek működtetése sem teljesen problémamentes, és itt is számos potenciális hibaforrással kell számolnunk, amelyek befolyással vannak az elvégzett számításokra.

Mint láttuk a véges számábrázolás következtében fellépő kerekítési hiba az egyik legkritikusabb, és ennek a kezelése a legmunkaigényesebb. ***Tehát a digitális gépekre is igaz, hogy amit kiszámítanak az az eredmény plussz–mínusz mégvalami, jelen esetben a kerekítési hiba.***

Kérdés az, hogy melyik jellegű hiba a nagyobb, vagyis melyik technikai megvalósítást érdemes alkalmazni az analógot, vagy a digitálisat. Most nem fogjuk részletesen kielemezni és összehasonlítani ennek a két fajta hibaforrásnak a nagyságát ugyanis a központi kérdés nem ez, hanem az, hogy melyiket és hogyan lehet hatékonyabban kezelni. Ezzel kapcsolatban viszont felhívjuk a figyelmet egy igen figyelemreméltó különbségre.

Az analóg gépeknél a hiba forrása a zaj, vagyis a gépben lejátszódó fizikai folyamatok véletlen fluktuációja. Ennek megfelelően a belőlük származó hiba is a véletlen törvényszerűségeknek engedelmeskedi, vagyis matematikai szempontból valószínűségi változó. Így, ha bármit mondani akarunk erről a fajta hibáról, akkor

```
procedure GetPi3;
var
  P: Extended;
  n: Extended;
begin
  P := 0;
  n := 1;
  while n <= 1000000 do
    begin
      P := P + (1 / (n * n));
      n := n + 1;
    end;
  P := Sqrt(6 * P);
  WriteLn(P);
end;
```

3.5. ábra. A π–t közelítő algoritmus (harmadik változat)

a valószínűségszámítás eszköztárát kell igénybevennünk, tehát végső soron kockajátékot játszunk.

Ezzel szemben a véges számábrázolás miatt fellépő kerekítési hiba nem valószínűségi változó, hanem aritmetiakilag egyértelműen meghatározott. Itt tehát nem csak azt tudjuk megmondani, hogy egy bizonyos valószínűséggel mekkora a hiba nagysága, hanem azt, hogy mekkora az az érték, amelynél száz százalékos biztonsággal nem nagyobb. Másrészt ha kétszer egymás után végezzük el ugyanazokat a műveleteket, akkor a digitális gépben a kerekítési hiba mindegyik esetben ugyanakkora lesz; analóg gépnél viszont más és más értéket vesz fel a valószínűségi változó, amely leírja a hibát.

Ezen utóbbi érvek nagymértékben a digitális technika javára billentik a mérleg nyelvét és nagyjából érzékeltetik azt, hogy miért jelentett nagy előrelépést a Neumann által alkalmazott digitális elv a számítógépek fejlődésében, az akkoriban uralkodó analóg gépekkel szemben. A továbbiakban részletesen megvizsgáljuk ennek a gépnek a logikai szerkezetét és struktúrális felépítését.

3.2. A számítógépek logikai–struktúrális szerkezete

Térjünk vissza a 3.1 ábrához, amely a Neumann–elvű számítógépek szerkezeti felépítésének vázlatát mutatja. Mint már fentebb említettük, ezt a koncepciót tekintik az összes, ma általánosan használt digitális számítógép modelljének. Valóban nem nehéz felfedezni a hasonlóságot az asztalunkon álló gép, és a Neumann–gép között. Igaz ugyan, hogy az IAS még egy teljes szobát betöltött, a mai gépeket pedig félkézzel is könnyedén fel lehet emelni, nem is beszélve az egyre népszerűbb *notebook*–okról, *laptop*–okról, valamint egyéb mini– és zsebszámítógépekről, de ha elkezdünk darabokra szedni egy mai gépet, akkor könnyen beazonosíthatjuk annak alkotóelemeit az 3.1 ábrán lévő részekkel, amelyek a következők:

1. Aritmetikai egység.
2. Tároló egység, vagy más néven memória.
3. Vezérlő egység.
4. Bemenő egység.
5. Kimenő egység.

Az aritmetikai egység

Az aritmetikai egység, vagy ahogyan a mai gépekben nevezik aritmetikai–logikai egység (*Arithmetical–Logical Unit*=ALU), mint az elnevezése is mutatja az aritmetikai és logikai műveletek elvégzését célozza. Valójában tényleg nem tud egyebet, mint összeadni, kivonni, osztani, szorozni, és végrehajtani néhány alapvető logikai műveletet, mint az AND, az OR, a XOR, a NOT stb.

Ennek ellenére ez az egyik legfontosabb szerve a gépnek, mert ez végzi a tényleges munkát, itt realizálódik a megprogramozott algoritmus végrehajtása, és a többi egység tulajdonképpen csak „aszisztál" az ALU–nak. Az elmondottak azonban azt is jelentik, hogy végsősoron minden algoritmus — még a legbonyolultabbak is —, lefordíthatóak az imént említett primitív műveletek egymásutánjára, vagyis végsősoron a formállogika nyelvére. Sőt mi több, a nemsokára tárgyalásra kerülő Church–tézis azt állítja, hogy egyáltalában csak olyan problémák oldhatóak meg algoritmikusan, amelyeket le lehet írni az említett formális műveletek soraként.

Ez tehát az első pont, ahol megmutatkozik, hogy milyen szoros összefüggés van a logika és a számítóeszközök között. Olyannyira szoros ez az kapcsolat, hogy a logika határai egyben (mindenkori) gépünk határait is jelentik. Márpedig a logika korlátairól 1931. óta Kurt Gödel nyomán igen meglepő dolgokat tárt fel a tudomány.

Ha vetünk egy pillantást az 3.1 ábrára, akkor láthatjuk, hogy az aritmetikai egység belsejében egy akkumulátornak nevezett komponens van. Ez nem egyéb, mint egy memória regiszter, amely számok tárolására képes. Amikor az ALU valamilyen aritmetikai, vagy logikai műveletet hajt végre, akkor előtte ezekbe a regisztrekbe (merthogy több is lehet belőle) tölti be az operandusokat, vagyis azokat a számokat, amelyere végre kell hajtani az aktuális műveletet. Az akkumulátor úgy van kilakítva, hogy az elérési ideje, tehát a számok beírásához és kiolvasásához szükséges idő minimális legyen. Mivel az ALU belsejében van, valóban lényegesen rövidebb idő alatt hozzáférhető, mint egy általános memória cella, amely egy az ALU–tól teljesen különálló egységben, a tárolóban van.

A tároló egység

A tároló egységben olyan cellák összességét, találjuk, amelyek képesek arra, hogy belső állapotukban egy–egy számot kódoljanak, és a felvett belső állapotot hosszabb időn keresztül megőrizzék, valamint ha az szükséges, akkor az ALU, vagy a gép bármely más szervének kérésére a beléjük írt számot „visszaszolgáltassák".

A tárolóban, eredetileg azok a számok kaptak helyet, amelyekkel a műveleteket végezte az aritmetikai–logikai egység, vagyis az adatok. A tároló cellái véges sokan vannak, ami azt jeleni, hogy megsorszámozhatóak 1–től n–ig. Egy–egy cella sorszámát az adat memória címének, vagy egyszerűen csak címének nevezzük. Ha az ALU–nak valamilyen műveletet kell végeznie, akkor az adott címről átírja az adatokat az akkumulátorokba, majd a végrehajtás után visszatölti az eredményt egy másik címre, vagy akár ugyanoda, ahonnan kiolvasta az operandust.

A korai számítógépeknél a program végrehajtása, és vezérlése általában úgy történt, hogy amikor az ALU elvégezte a műveletet, akkor összeköttetést kellett létesíteni az akkumulátor, és aközött a memória cella között, ahova az eredményt el kellett tárolni. Ehhez hasonlóan a művelet előkészítő szakaszában pedig a forrás operandusokat tároló cím, és az akkumulátorok között kellett kapcsolatot teremteni. Ezt a kezdet–kezdetén manuálisan oldották meg, vagyis az operátor egyszerűen összekötötte egy dróttal a megfelelő cellákat, ahhoz hasonlóan, ahogyan a régi fajta, nem automata telefonközpontban a telefonos kisasszony összekötött két állomást, amely beszélni akart egymással. Később a manuális megvalósítást felváltották az elektromos jelfogók amelyek állását mágneses tekercsek elektromos gerjesztésével szabályozták, ez utóbbiak gerjesztő áramát pedig lyukszalag vezérelte.[11] Végsősoron tehát a lyukszalagon volt a tulajdonképpeni program, amely meghatározta, hogy mikor melyik memóriacellát kell összekötni az akkumulátorral, majd hova kell visszahelyezni az eredményt.

Neumann volt az első, aki felvetette az ötletet, hogy mi lenne akkor, ha az utasításokat is el tárolnánk a memóriába az adatok mellett. Ennek alapja egyszerűen az, hogy az utasítások is véges sokan vannak, tehát ezeket is sorszámozhatjuk. Ezek után az egymásutáni utasítások sorszámát beírjuk a memóriába, és építünk egy egységet, amely „megérti” ezeket az utasításokat, vagyis ha beolvasott egy számot, akkor tudja, hogy ez a szorzás, összeadás stb. kódja, és végrehajtatja azt az aritmetikai–logikai egységgel (hogy milyen módon, arra még visszatérünk).

A vezérlő egység

A vezérlő egység az a szerve a gépnek, amely a memóriában tárolt program értelmezéséért felelős. Alapvetően az alábbi dolgokat kell tudnia:

1. Beolvas egy számot, és megállapítja, hogy az melyik utasítás kódja.
2. Megállapítja azt, hogy hány operandusa van a műveletnek, és mi ezen operandusok tároló címe.

[11] Vö. [NJ] 271. o.

3. Betölti az operandusokat az iménti lépésben meghatározott címről az akkumulátorokba.

4. Végre hajtatja a műveletet az aritmetikai–logikai egységgel.

5. Megállapítja, hogy hova kell elhelyezni a művelet eredményét és visszatölti azt arra a címre.

6. Megállapítja, hogy hol van a következő végrehajtandó utasítás memória címe, majd az egész folyamatot megismétli az 1–es lépéstől kezdve.

Természetesen az utasítások, amelyeket a vezérlő egység „megért", igen sokfélék lehetnek, és az egyszerű programozhatóság érdekében kívánatos is, hogy sokfélék legyenek. Azonban ezek az utasítások típusokba sorolhatóak, és osztályozhatóak. Mi a teljesség igénye nélkül csak a legfontosabbakat emeljük ki:

– **Matematikai operátorok**: aritmetikai és logikai utasítások, amelyet az ALU hajt végre.

– **Vezérlés átadó utasítások**: olyan utasítások, amelyek alapján a vezérlő egység meghatározza a következő végrehajtandó utasítás tároló címét. Feltétel nélküli vezérlés átadás esetén az utasítás egyértelműen meghatározza azt, hogy milyen memória címen van a következő végrehajtandó utasítás. Feltételes vezérlés átadásnál a vezérlő egységnek „választási lehetősége" van, hogy hol keresse a következő utasítást. A döntés, amit meghoz, általában az alapján realizálódik, hogy az utoljára elvégzett műveletnek mi volt az eredménye.

– **Relációs operátorok**: olyan utasítások, amelyek összehasonlítanak két memóriacímen lévő számot, tipikusan a $<$, a $>$, az $=$, a $\leqq$, és a $\geqq$ relációkat valósítják meg. Szoros kapcsolatban vannak az imént említett vezérlés átadó utasításokkal, ugyanis általában egy kitüntetett tároló címen lévő szám előjelét változtatják meg annak függvényében, hogy a két operandusra alkalmazva fenn áll–e a reláció, vagy sem. Ezek után egy vezérlésátadó utasítás ez alapján határozza meg a folytatást. Így a kétfajta utasítás családot kombinálva már komolyabb, elágazásokat is tartalamzó algoritmusok is implementálhatóak a gépen.

– **Adatmozgató utasítások**: adatokat töltenek be egy memória címről az akkumulátorokba (LOAD), és adatokat töltenek vissza a tárolóba az akkumulátorokból (STORE).

– **Egyéb utasítások**: ezek olyan utasítások, amelyek nem feltétlenül szükségesek, de kényelmesebbé tehetik a programozást. Ide tartoznak például a program futását befejező termináló utasítások. Ha vezérlő egység egy ilyet hajt végre, akkor annak az lesz a következménye, hogy a program futása befejeződik, és a gép megáll.

Minél több utasítást ismer egy gép, annál könyebb programozni. Azonban ez nem jelenti azt, hogy bármivel többet tud, és a problémák szélesebb osztályára alkalmazható, mint egy kevesebb utasítással rendelkező gép. Ez nagyon jól érzékelhető a RAM–programoknak nevezett számítóeszköz modell esetében. A RAM–program modell, egy ugyanolyan elméleti absztrakciója a számítóeszközöknek, mint a Turing–gép, csak kevésbé elvont, és szerkezetileg közelebb áll a Neumann–elvű géphez, mint az általunk tárgyalandó Turing–gép. Az irodalomjegyzék [P], [LL], és [RISz] könyveiben leírást találhatunk a RAM–programokról[12] valamint a Turing–gépekkel való ekvivalenciájukról, és például a [P] irodalom megadja egy olyan RAM gép utasítás készletét, amely mindössze 11 utasításból áll[13], és bizonyítja, hogy ezen 11 elemű assembly jellegű programnyelven bármely olyan program megvalósítható, amely implementálható Turing–gépen is, és fordítva.[14]

A bemenő– és kimenő egységek

Ezen egységek feladata, hogy az alapvető kommunikációs csatornát biztosítsák ember és gép között. Nyilvánvaló, hogy a feladat megoldására szolgáló programot, és az adatokat, amelyen az dolgozik, valamilyen formában a gép tudomására kell hozni (bevitel), illetve ha az befejezte a működést, az eredményeknek hozzáférhetőknek kell lenniük a felhasználó számára (kivitel). Ha ez nem lenne megoldott, akkor teljesen értelmetlen lenne gépet használni a feladatok megoldására, hiszen nem tudnánk, hogy milyen választ adott a program. A bemeneti– és kimeneti eszközök nagyon sokfélék lehetnek. Korábban például mind a bemnetet, mind pedig a kimenetet lyukszalaggal oldották meg, de a bemeneti eszközök ma alkalmazott legtipikusabb megoldása a billentyűzet és az egér, a kimeneti eszköz pedig a képernyő, vagy a nyomtató. Ettől részletesebb tárgyalásuk, és egyéb eszközök vizsgálata témánk szempontjából nem bír jelentőséggel.

[12] Vö. [P], 40.–50. o.; [RISz] 239.–345. o.; [LL], 12.–16. o.

[13] [P], 42. o.

[14] Csak összehasonlításképpen: a PENTIUM® processzorok több, mint 600 assembly utasítást ismernek, márpedig ezek sem tudnak többet a Turing–gépeknél.

3.2.1. A tárolt programú gép lehetőségei

Korábban már említettük, hogy Neumann volt az első, aki felvetette azt, hogy a végrehajtandó program — az adatokhoz hasonló módon — tárolható a gép memóriájában. Már önmagában ez is egy igyen figyelemreméltó elgondolás, azonban Neumann — nem hiába volt zseni —, ennél sokkal merészebb dolgot látott meg a tárolt programú gép lehetőségeiben, ennek lényege pedig a következő: a gép számára nem bírnak semilyen jelentéssel a memóriájában tárolt számok, tehát amikor a memória egy konkrét celláján dolgozik, nem tud különbséget tenni aközött, hogy az aktuális cella tartalma egy adat, vagy pedig a programnak egy utasítása. Viszont bármelyik cella tartalmát szabadon módosíthatja, vagyis ha az aktuális cella egy program utasítást tartalmaz, akkor minden további nélkül átírhatja azt bármilyen más utasításra. Ez pedig nagyon izgalmas lehetőségeket vet fel, ugyanis íly módon lehet olyan programot írni, amely saját maga kódját képes megváltoztatni, vagyis amely bizonyos feltételek függvényében átírja önmagát.[15]

Neumann–t élénken foglalkoztatta a mesterséges élet gondolata. Például sok egyéb munkája mellet *Az automaták általános és logikai elmélete*[16] című dolgozatában részletesen összehasonlítja az élő szervezeteket a gépekkel, és elemzi, hogy miként lehetne olyan szerkezetet építeni, amely az élő szervezetekhez hasonlatos tevékenységeket mutat, tehát például átalakítja a környezetében fellelhető anyagokat, reprodukálja önmagát, reagál a környezetében bekövetkező változásokra és alkalmazkodik azokhoz stb.

A végletekig leegyszerűsítve a következőről van szó: tegyük fel, hogy van egy raktár, amelyben olyan alkatrészek vannak, amelyekből robotokat lehet építeni. A kérdés az, hogy vajon meg tudunk–e építeni egy robtot úgy, hogy az a következő tevékenységsort hajtsa végre: fel-alá rohangál a raktárban, és megkeresi a polcokon a saját komponenseit, majd ezekből az alkatrészekből összeszereli önmaga tökéletes mását. Természetesen úgy, hogy a másolat is működőképes legyen, vagyis ha elkészült, akkor már ketten rohangálnak össze–vissza a raktárban, ahol mindegyikük újfent elkezdi legyártani önmaga másolatát és így tovább a végtelenségig.

Ez csak egy elméleti absztrakció, nyilvánvaló, hogy előbb–utóbb kiürül a raktár, tehát legfeljebb csak potenciálisan tekinthetjük végtelennek. Azonban Neumann matematikus lévén elméleti ember volt, és nem az érdekelte, hogy a fenti konstrukció a maga fizikai valóságában véghezvihető–e, hanem az, hogy elméletileg, elviekben megvalósítható–e, vagy esetleg van valamilyen olyan logikai érv, amely ellene mond egy ilyen szerkezet létezésének és megépítésének. Neumann elkezdett foglalkozni a problémával, és lefektette egy

[15] Vö. [NJ] 274.–280. o.

[16] Megtalálható az irodalomjegyzékben lévő [NJ] könyvben.

máig is igen érdekes kutatási terület alapjait, amelyet sejtautomata algoritmusoknak nevez a tudomány.[17] Ennek a vizsgálódásnak a végső konklúziója az volt, hogy semmilyen olyan elvi megfontolás nincs, amely kizárja egy ilyen önreprodukáló objektum létezését.

Azonban most már egy kissé messze elkalandoztunk témánktól, és nem teljesen világos, hogy mi köze ennek az egésznek ahhoz, ahonnan elindultunk, vagyis a Neumann János által felvetett tárolt programú számítógép ötletéhez. A válasz egyszerű. Ha elméletileg lehetséges önmagát lemásoló objektum, akkor ezt minden további nélkül kiterjeszthetjük a számítógépes programokra is. Valóban vannak olyan programok, amelyek lemásolják magukat, és képesek olyan viselkedést mutatni, mint a fentebb tárgyalt robot. Láttuk azt is, hogy olyan program is van, amely saját maga kódját módosítja. Nem nehéz belátni, hogy ennek következtében olyan program is lehetséges, amely más program kódját változtatja meg. Ezek után már nem kell hozzá nagy fantázia, hogy összeálljon a kép, és olyan programot képzeljünk el, amely az iménti mondatokban említett tulajdonságokkal egyszerre rendelkezik. Manapság, köznapi szóhasználattal az ilyen programokat úgy nevezik, hogy számítógép–vírus. Ezek a programok vagy a memóriában lévő, vagy a háttértárolón elhelyezkedő programok kódját írják át úgy, hogy a program indulásakor elsőként ők kapják meg a vezérlést, és így lemásolhassák, és más programokhoz hozzáfűződve működőképessé tegyék magukat, vagyis az alapelv ugyanaz, mint az említett robot esetében.

Mivel nem csak más programok kódját, hanem a sajátjukat is átírhatják, intelligensebb példányaik meg is teszik ezt. Ők a polymorfikus– és permutáló vírusok, amelyek kódjuk megváltoztatásával rejtik el magukat a meghatározott mintát kereső antivírus programok elől. Természetesen Neumann korában még ez a fogalom még nem létezett. A terminológia csak akkor született meg, amikor rájöttek arra, hogy az ilyen jellegű programok terjedése ugyanolyan valószínűség eloszlásokat és matematikai–statisztikai törvényszerűségeket követ, mint amelyek a járványos betegségeket okozó biológiai vírusok terjedését leírják. Ettől kezdve nevezik őket vírusoknak. Lám nincs új a nap alatt. Neumann János, aki a modern kor egyik legnagyobb áldásának, a tárolt programú, digitális számítógépnek az alapjait lerakta, tudtán és akaratán kívűl, az azt sújtó egyik legnagyobb átok, a számítógép–vírus elvi koncepcióját is kidolgozta.

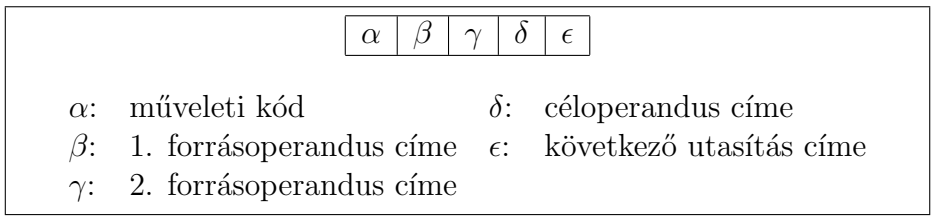
[17] A sejtautomaták tulajdonképpen egyik fajtája a számítógépeknek. A téma terjedelme miatt nem áll módunkban, hogy itt részletesen foglakozzunk velük, azonban az irodalomjegyzékben lévő [VR] könyv általános bevezetést nyújt az elméletükbe, valamint az alábbi két cikkben egyéb adalékokat olvashatunk a témához: Kós Géza: *Kutyák a marsról*, KöMal, 1996/3. 138.–144. o. illetve Totik Vilmos: *Az élet játék(a)*, Polygon, 1996/VI. 9.–21. o.

3.2.2. A memória címzése

Az imént tárgyalt szekezeti egységek képezik tehát a Neumann–gép fő részeit, és könnyű felfedezni, hogy a mai gépek is ugyanezen az elven szerveződnek. Térjünk vissza néhány mondat erejéig a vezérlő egység működésére, és arra hogy milyen módon valósítja meg az utasítások feldolgozását és a memória írást/olvasást, egyszóval a címzést.

Korábban már említettük azt, hogy minden algoritmus leképezhető elemi lépésekre, vagyis egyszerű matematikai és a logikai utasítások sorozatára. Láttuk azt is, hogy ha minden egyes utasításnak adunk egy sorszámot, úgynevezett utasításkódot, akkor az utasítássor is bevihető a számítógép memóriájába, ahonnan a vezérlő egység kiolvassa, és értelmezi ezt a kódot, majd végrehajtatja a neki megfelelő műveletet az ALU–val. Ez azonban további problémákat vet fel, például olyanokat, hogy honnan állapítja meg a vezérlő egység, hogy hol van a következő végrehajtandó utasítás, honnan tudja azt, hogy hol van a művelet végrehajtásához szükséges operandus, hová tegye az eredményt a memóriába, ha egy művelet befejeződött stb. A számítógépek fejlődése során különféle megoldások és címzések alakultak ki ezeknek a problémának a megoldására.

A mai gépekben az aritmetikai–logikai egység és a vezérlő egység egyetlen szerkezeti komponensbe integrálva jelenik meg, amelyet *Central Processing Unit*–nak (CPU), tehát központi feldolgozó egységnek, vagy egyszerűen csak processzornak neveznek. Nyilvánvaló, hogy ha betöltünk egy programot a gép memóriájába, akkor az első és legfontosabb információ a processzor számára, hogy melyik az a memória cella, ahol a program legelső utasítása megtalálható, tehát hol van a program belépési pontja. Ezután az egyik lehetséges megoldás a vezérlésre az utasítások megfelelő szerkezeti kialakítása, például az 3.6 ábrán látható séma szerint. Ebben az utasításban az



3.6. ábra. A négycímes processzor utasítás szerkezete

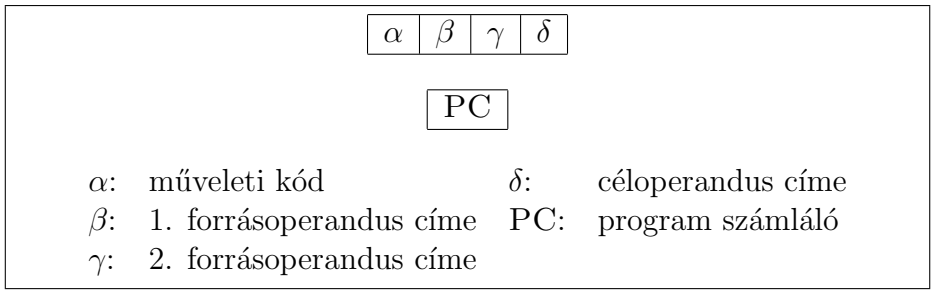
első komponens mondja meg a vezérlő egységnek, hogy melyik utasítást kell végrehajtatni az ALU–val; a második és harmadik összetevő jelöli az operandusok címét, amelyekre alkalmazni kell a műveletet (természetesen ha csak egy operandusos a művelet, akkor csak az egyik cím van használatban); a

harmadik összetevő mutatja a címet, ahova az eredményt el kell helyezni; és végül az utolsó komponens adja a következő végrehajtandó utasítás címét. Mivel ebben az utasítás szerkezetben négy darab memória hivatkozás van, ezért az ilyen megoldást használó processzorokat négycímes processzoroknak nevezik.

Az iménti konstrukciót alkalmazó gépeken tetszőleges algoritmust megprogramozhatunk, azonban van egy nem elhanyagolható hibájuk a négycímes processzoroknak, ami különösen a számítógépek fejlődésének korai szakaszában jelentett nagy problémát. Mégpedig az, hogy meglehetősen pazarlóan bánnak a memóriával, a sok címhivatkozás miatt, ami egy–egy program utasításban előfordul. Ezért elkezdtek keresni olyan megoldásokat, amelyek csökkentik az utasítások hosszát. Az első lépés ezen az úton a négycímes utasítás szerkezet utolsó komponensének, vagyis a következő utasítás címének elhagyása volt. De ha ezt tesszük, akkor hogyan fogja a vezérlő egység megtalálni a következő utasítást? A trükk nagyon egyszerű, és azon alapul, hogy minden esetben ismerjük, hogy mekkora egy–egy utasítás hossza, vagyis mekkora területet foglal el a memóriában. Ezek után egyszerűen csak azt kell tenni, hogy a processzorba beépítünk egy speciális számláló regisztert, amely kezdetben a program belépési pontját tartalmazza, majd utána minden egyes utasítás végrehajtása után megnövelni ennek a regiszternek a tartalmát annyival, amilyen nagyságú memória területet az utoljára végrehajtott utasítás elfoglalt. Ezt a speciális regisztert több elnevezéssel is illetik az egyes szakkönyvek: gyakran nevezik *Program Counter*–nek (PC), máshol — főleg az Intel processzorok estében — az *Instruction Pointer* (IP) elnevezéssel találkozhatunk. Ennek a megoldásnak a bevezetésével megoldódott az a probléma, hogy hogyan állapítja meg a vezérlő egység a következő utasítás helyét, akkor ha az nem szerepel közvetlenül az utasításban. Illetve még nem teljesen. Ugyanis ezzel a koncepcióval csak szekvenciális programokat tudunk implementálni, vagyis olyanokat, amelyek semmiféle elágazást nem tartalmaznak, hanem az utasításaik az eredeti sorrendnek megfelelően, szigorú egymásutániságban hajtódnak végre. Azonban ezt a nehézséget is kiküszöbölhatjük a már említett vezérlésátadó utasítás család bevezetésével, amelyek éppen az által teljesítik feladatukat, hogy közvetlenül képesek megváltoztatni a PC regiszter tartalmát, vagyis arra valók, hogy az ott található memória címhez egy számot hozzáadjanak, vagy kivonjanak belőle valamennyit, vagy egyszerűen csak felülírják azt egy másik címmel. Így már a programunk tetszőleges pontjáról bármely másik pontra átadhatjuk a vezérlést.

Azonban vegyük észre azt a tényt, hogy amint egyszerűsítettük az utasítás szerkezetet, ez azonnal automatikusan maga után vonta azt, hogy növelni kellett a processzor utasítás készletét. A négycímes processzoroknak nincs szükségük vezérlésátadó utasításokra, mert az utasítás szerkezet ezt helyet-

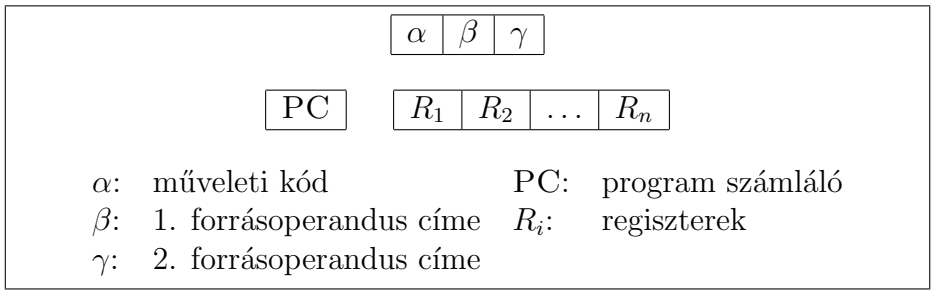
tesíti. Viszont a 3.7 ábrán lévő háromcímes utasítás szerkezetet használó



3.7. ábra. A háromcímes processzor utasítás szerkezete

processzoroknak külön utasításokkal kell megvalósítaniuk a ciklusokhoz és elágazásokhoz elengedhetetlenül szükséges ugrásokat, és már önmagában ez is bonyolítja fizikai szerkezetüket, másrészt az is, hogy beléjük kell építeni a program számláló regisztert. Így kaptuk tehát a háromcímes processzorokat, amelyek utasítás szerkezete elhagyja a következő utasításra mutató hivatkozást, és ezáltal kevesebb memóriát igényel a program kód tárolásához.

A sikeren felbuzdulva alkalmazzuk újra az iménti ötletet, és hagyjuk el megint a legutolsó memória hivatkozást, vagyis most a céloperandus címét. Igaz ugyan, hogy ekkor nem fogjuk tudni, hogy hova tegyük a művelet eredményét, de ha tovább növeljük processzorunk komplexitását, a probléma megint csak kiküszöbölhetővé válik. Az egyik megoldás, hogy az eredményt

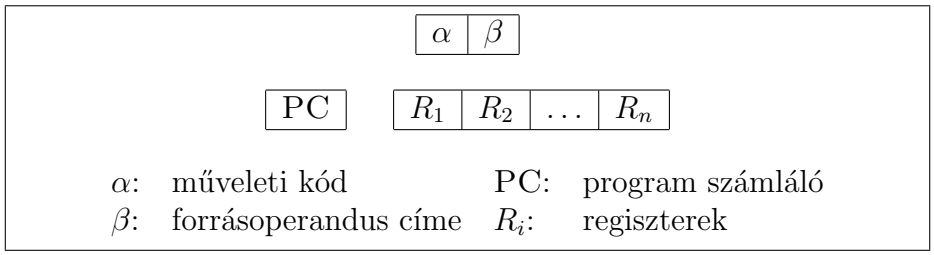


3.8. ábra. A kétcímes processzor utasítás szerkezete

a CPU valamelyik belső regiszterében, például a már említett akkumulátorban tároljuk el, és utána a STORE adatmozgató utasítással visszaírjuk valamelyik memória címre. Itt megint látjuk érvényesülni az imént felfedezett törvényszerűséget, mert megint növelni kellett a processzor utasítás készletét: az utasítás szerkezet egyszerűsítése érdekében be kellett vezetni az adatmozgató utasításokat (LOAD és STORE). Azonban van más lehetőség is ebben

az esetben. Megtehetjük egyszerűen azt, hogy az eredményt nem egy belső regiszterben tároljuk el, hanem felülírjuk vele valamelyik forrásoperandust, tehát az utasításban szereplő β vagy γ címre helyezzük el. Természetesen ez által az eredetileg ott lévő adat elveszik, ezért ha még a későbbiekben is szükségünk van rá a számítások során, akkor azt el kell mentenünk egy más címre, és ez a megoldás — bár nem feltétlenül —, de megint új utasítások definiálását követelheti meg.

Az elmondottak után nem látszik különösebb akadálya annak, hogy még tovább egyszerűsítsük az utasítás szerkezetet, és az egyik forrásoperandus elhagyásával még tovább csökkentsük az utasítások hosszát, és ezáltal az egész programkód hosszát. Ha viszont elhagyjuk az egyik forrásoperandust, ak-



3.9. ábra. Az egycímes processzor utasítás szerkezete

kor az egyetlen megoldás az, hogy a másik operandust vagy egy regiszterbe kell betöltenünk a LOAD utasítással még a művelet végrehajtásának megkezdése előtt, vagypedig konstansként eltárolni magában az utasításban. Az eredményt pedig az előző meggondolásaink alapján szintén vagy a (nem konstans) forrásoperandusban, esetleg egy regiszterben kapjuk majd vissza. Ez lesz az egycímes processzor utasításszerkezete, amelyet manapság a legtöbb CPU architektúra alkalmaz (3.9 ábra).

Az eddigiekben áttekintettük a mai számítógépek működésének alapelvét, és szerkezeti–logikai struktúráját, amelyet a Neumann–elvként ismert koncepcióból kiindulva írtunk le.[18] A továbbiakban arra törekszünk, hogy

[18]Mi csak vázlatosan érintettük a témát, és a legtöbb hozzá kapcsolódó problémát, azonban az irodalomjegyzékben lévő [TA] könyv részletesen ismerteti az egyes számítógép architektúrákat; illetve Neumann koncepcióját elsőkézből megismerhetjük az alábbi három dolgozatból, amelyek mindegyike megtalálható az [NJ] irodalomban:

1. Neumann János: *Az automaták általános és logikai elmélete*
2. Neumann János: *Az újabb matematikai gépek fejlődése és kihasználása*
3. Neumann János: *A számológép és az agy*

ezt a modellt leegyszerűsítsük egészen addig a pontig, ahol már mind a modellt, mind pedig az általa végzett számítás sorozatot olyan precíz matematikai fogalmakkal tudjuk kapcsolatba hozni, mint a *függvény*, a *halmaz* vagy például a *reláció* stb.

3.3. Turing–gépek és az algoritmusok elmélete

Láttuk azt, hogy az egycímes processzorok utasításszerkezete az egyik lehető legegyszerűbb. Ennek egyenes következménye egyrészt, hogy ha ugyanazt az algoritmust implementáljuk egycímes– illetve négycímes processzorra, akkor az utóbbi esetében a program kódjának a hossza lényegesen nagyobb lesz, mint az egycímes processzor esetében, de más részről azt is megállapítottuk, hogy minél egyszerűbb az utasítások szerkezete, annál több utasítást kell ismernie a gépnek ugyanazon problémák megoldásához.

A továbbiakban eltekintünk attól, hogy könnyen programozható és áttekinthető működésű gépet építsünk. Az egyetlen szempont, amit figyelembe veszünk az az, hogy olyan gépet konstruáljunk, ahol az egyes számítási sorozatok elemi lépésekre és műveletekre való felbontása a lehetőségek végső határáig megvalósul. Például két szám összeadása nem elemi művelet, mert még egyszerűbb logikai jellegű bitmanipulációkra lehet visszavezetni, ahogyan ez a 3.2 ábrán lévő műveletáblákból ránézésre is kiderül.

3.3.1. A Turing–gépek definíciója

Az eddig elmondottak alapján nyilvánvalóvá vált, hogy a memória írása és olvasása nem különösebben bonyolult művelet, pusztán a címzés okozott némi bonyodalmat, de különböző trükkös megoldásokkal sikerült megszabadulni ezektől a nehézségektől. Sőt a végereményként kapott megoldás olyan hatékonynak bizonyult, hogy segítségével nem csak szekvenciális programokat tudtunk megvalósítani, hanem bonyolult, elágazásokat és iterációkat is tartalamzó algoritmusokhoz is logikailag jól struktúrált, és áttekinthető programkódot tudtunk hozzárendelni.

Éppen azért, mivel ilyen jól működő megoldást kaptunk, első megközelítésben talán merész ötletnek tűnik, de a továbbiakban megpróbáljuk még tovább egyszerűsíteni a modellt, azáltal, hogy valamilyen módon megpróbáljuk kiküszöbölni a memória megcímzésének szükségességét.

Azonban nem nyilvánvaló, hogy ezt miként tudnánk megvalósítani. Hogyan hivatkozzunk az adatokra, ha nem tudjuk, hogy hol vannak? Némi

fejtörés után viszont szinte magától adódik a következő ötlet. A memóriát ezentúl tekintsük úgy, mint egy szalagot, amely cellákra van felosztva, ahol egy–egy cella tartalmaz egy–egy adatot, ahogyan azt alább látjuk:

...	σ_{i_1}	σ_{i_2}	σ_{i_3}	σ_{i_4}	...	σ_{i_n}	...

Van továbbá egy író–olvasó fej, amely minden egyes pillanatban pontosan egy, és csak egy cellára mutat a szalagon. Amelyik cellára mutat, annak tartalmát kiolvashatja, és oda bármit visszaírhat, de nem férhet hozzá más cellákhoz. A fej minden egyes lépésben legfeljebb ± 1 pozícióval változtathatja meg az aktuális helyzetét, tehát konkrétan legfeljebb egy cellát léphet balra, illetve jobbra. Ezeket a szabályokat figyelembe véve tehát, ha hozzá akarunk férni valamelyik másik — például a jelenlegitől öt cellával jobbra lévő — cella tartalmához, akkor ezt csak úgy tudjuk megvalósítani, hogy egyesével ellépkedünk pozitív irányba addig a celláig. Ha pedig a jelenlegi celltól balra lévő cellában helyezkedik el a kívánt adat, akkor negatív irányba lépkedünk.[19]

Ezt a bonyolult megoldást korábban egy egyszerű* LOAD *utasítással helyettesítettük volna. Azonban az imént deklaráltuk azt célt, hogy a végletekig leegyszerűsített elemi lépésekre akarunk felbontani minden műveletet, amely a gép belsejében lejátszódik. Márpedig az, hogy a fej egyet jobbra, vagy egyet balra mozdul, ez tényleg tekinthető elemi lépésnek, míg a* LOAD *utasítás végrehajtása esetében — amely egy magasabb absztrakciós szintet képvisel — nem nyilvánvaló, hogy milyen fizikai folyamatok komplex összjátéka eredményeként jelenik meg a memória tartalma a processzor belsejében.

Most már csak az a kérdés, hogy miként vezéreljük az író–olvasó fej működését és mozgását. Ennek érdekében tegyük fel, hogy az egész gépnek vannak belső állapotai, amelyeket ugyanúgy kódolhatunk valamilyen fizikai paraméterrel, mint ahogyan korábban a digitális– és analóg gépekkel foglakozó fejezetben fizikai jelenségek (például feszültség, áramerősség, fordulatszám stb.) mérőszámait rendeltük hozzá a számokhoz, és így ábrázoltuk azokat. Tegyük fel továbbá azt is, hogy ezek a lehetséges belső állapotok véges sokan vannak, diszkrét értékek, tehát nincs köztük átfedés, és a gép nem lehet egyszerre több belső állapotban, hanem aktuális állapota mindig egyértelműen meghatározott.

[19]Matematikailag ez azt jelenti, hogy a szalag celláit sorszámozhatjuk, és ez a sorszámozás — ha a szalagot mindkét irányba végtelennek tekintjük —, nem más, mint a egy kölcsönösen egyéretlmű leképezés a cellák, és az egész számok halmaza, vagyis $\mathbb{Z}$ között. Azonban éppen az a lényege a fenti konstrukciónak, hogy ne a sorszámukkal (=címükkel) hivatkozzunk a cellákra, hanem ennek szükségességét elemi műveletekkel feloldjuk.

Ezek után a gépünk által elvégzett számítási sorozat leírható úgy, mint az író–olvasó fej és a szalag kölcsönös egymásra hatása, mégpedig a következő képpen: a fej azáltal tud hatni a szalagra, hogy az aktuális cella tartalmának, és a gép aktuális belső állapotának függvényében változtatja meg a cella tartalmát[20], a szalag pedig úgy tud visszahatni a fejre, hogy az aktuális cellában lévő szimbolum elolvasása — attól függően, hogy milyen a gép aktuális belső állapota —, a gépet belső állapotának megváltoztatására és a fejet -1, 0, vagy $+1$ irányú elmozdulásra készteti, ahol -1 jelenti az egy cellányi balra mozgást, 0 a helyben maradást, $+1$ pedig a jobbra mozgást. Ez a lépéssorozat képezi a gép működésének egy–egy ütemét, és ezt a működést az alábbi típusú utasítások sorozatával írhatjuk le:

$$(q, \sigma) \rightarrow (q', \sigma', m)$$

amit úgy értelmezünk, hogy ha a fej q belső állapotban van és a szalag aktuális cellájából (vagyis amelyik felett éppen tartozkodik) az σ szimbólumot olvassa, akkor ennek hatására belső állapotát q'–re változtatja, az aktuális cellába a σ' szimbólumot írja, és m irányú elmozdulást végez, ahol $m = \{-1, 0, +1\}$, a fentebb meghatározott jelentéssel. Természetesen azt is megengedjük, hogy esetlegesen $q = q'$ és/vagy $\sigma = \sigma'$.

Ha ezt az utasítás szerkezetet összehasonlítjuk a korábban tárgyalt akárhány címes processzorok utasítás szerkezetével, az első, ami a legszembetűnőbb, hogy ebben az utasításban mindösszesen háromfajta „cím” hivatkozás van, a -1, a 0 és a $+1$ és tulajdonképpen ezt a megvalósítást az egycímes processzor még további egyszerűsítésének foghatjuk fel.

Az iménti eszköz konstrukciója nem más, mint maga a Turing–gép, és szerkezeti sémáját a 3.10 ábrán láthatjuk. Ránézésre is megállapítható, hogy ez a szerkezet lényegesen egyszerűbb, és kevesebb komponensből áll, mint a Neumann–gép. Először is a memóriát egy — mindkét irányban végtelennek tekintett — szalag helyettesíti. Itt is találhatunk egy logikai egység nevű szerkezeti elemet, azonban ez is teljesen más célt szolgál, mint korábban. Működése a fentebb leírt ütemekre tagolható, pontosabban ez felelős egy–egy ütem végrehajtásáért. (A logikai egység belsejében lévő a Q regiszter csak arra szolgál, hogy a fej aktuális belső állapotát tárolja két ütem végrehajtása között.)

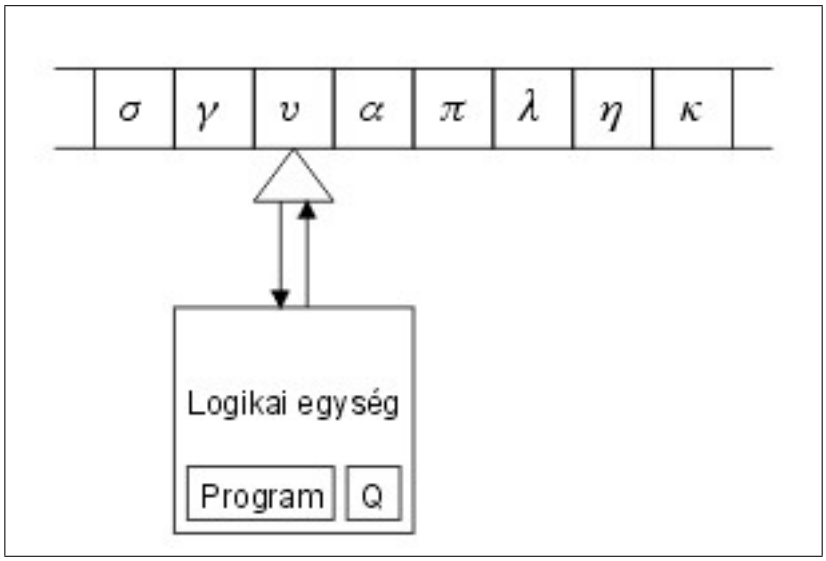
Egy másik feltűnő dolog ebben a gépben, hogy nincs bemeneti– és kimeneti egysége. Azonban erre nincs is szükség, mert a szalag tartalma közvetlenül hozzáférhető, vagyis ide írjuk a gép indulásakor a kezdő információt, és amikor az megállt, akkor a szalag

[20]Ebbe azt is beleértjük, amikor a fej ugyanazt a szimbólumot írja vissza az aktuális cellában, amelyet éppen kiolvasott.

aktuális tartalmát tekintjük a számítás eredményének, tehát a kimenet közvetlenül leolvasható a szalagról. Emlékeztetünk arra, hogy a korai gépek esetében is hasonló volt a helyzet, ugyanis ott is lyukszalaggal kellett bevinni az utasításokat és az adatokat a gépbe, és a kimenet is a lyukszalagon volt hozzáférhető.

Most már csak az nem világos, hogy a gép mikor, és hogyan fejezi be a működését, illetve, miként értesülünk mi ennek a ténynek a bekövetkezéséről. Azért, hogy ez egyértelmű legyen előre definiálunk olyan speciális állapotokat, amelyekbe ha a gép a futása során eljut, akkor automatikusan megáll.

Nevezzük ezeket az állapotokat terminál állapotoknak, hiszen ezek a korábban már említett terminál utasítások megfelelői a Turing–gépek esetében. A továbbiakban mi három ilyen terminál állapotot fogunk elkülöníteni, amelyekre a HALT, a YES és a NO elnevezésekkel fogunk hivatkozni. Ha a gép a HALT állapotba kerül, akkor egyszerűen csak megáll és a számítás eredményének a szalag aktuális tartalmát tekintjük, a YES és NO állapotok pedig az olyan jellegű problémák megoldását fogják jelezni, amelyekre igennel illetve nemmel lehet válaszolni, és ha a gép YES állapotban fejezi be a működését, akkor ezt a tényt úgy tekintjük, hogy a gép elfogadta a bemenetet, a NO állapotot pedig úgy értékeljük, hogy a gép elutasította a bemenetet. (Ennek a megkülönböztetésnek az értelme majd akkor válik világossá, amikor a magasszintű programozásból már ismert iterációkat és ciklusokat próbáljuk



3.10. ábra. A Turing–gép vázlata

megvalósítani a Turing–gépen egy későbbi fejezetben.)

Van még egy speciális, és egyértelműen meghatározott állapot, amelyet a START elnevezéssel jelölünk. Ez a kezdő állapot, és megállapodunk abban, hogy minden számítási folyamat megkezdése előtt a gép ebben az állapotban van.

Bizonyos szakirodalmak azt is kikötik, hogy a szalagnak van egy kitüntetett cellája, a kezdő cella, amely felett az író–olvasó fej áll a számítás megkezdése előtt. Ennek pozíciója többféleképpen is megadható, általában azt a legbaloldalibb cellát szokták választani, amely nem üres cella. Mi nem teszünk ilyen jellegű kikötést, ugyanis — mint később látni fogjuk —, ez nem szükséges, tehát úgy tekintjük, hogy elvileg bármely cellára beállíthatjuk a fejet a számítás megkezdése előtt.

Továbbá definiálunk egy (egyértelműen meghatározott) speciális szimbólumot, az üres szimbólumot (*blank symbol*). Legyen ez a jel a „#". Ha tehát valamelyik cella „#" jelet tartalmazza, akkor ezt a tényt azzal tekintjük egyenértékűnek, hogy a cella üres.

Ha összehasonlítjuk a Turing–program elemi utasításait, vagyis a $(q, \sigma) \rightarrow (q', \sigma,' m)$ ***típusú utasítások egy halmazát a Neumann–gép programjával, akkor nyilvánvaló, hogy a Turing–gépet sokkal komplikáltabb programozni, mint a másikat. A Neumann–gép programja egy magasabb absztrakciós szintet képvisel, és az emberi gondolkodáshoz és problémamegoldáshoz közelebbi szimbólumrendszerrel tudjuk a „tudomására hozni" a megvalósítandó algoritmust, vagyis a programot. A Turing–gépet viszont szándékosan úgy alkottuk meg, hogy minden egyes utasítása a lehető legelemibb szintet képviselje, cserébe viszont egy nehezen áttekinthető és struktúrálatlan szimbólumrendszerrel tudjuk csak vezérelni.***

A Turing–gépnek azonban van egy nagy előnye az egyéb típusú gépekkel szemben. Nevezetesen, ha az asztalunkon álló gép működését és az általa végzett számítás sorozatot le szeretnénk írni valamilyen formális definícióval, akkor nehézségekbe ütközünk, mert a gép túlzottan komplikált ahhoz, hogy minden egyes komponensének állapotát — amely a számítások szempontjából jelentőséggel bír — pillanatról–pillanatra végig kövessük. Ezzel szemben a Turing–gép eléggé egyszerű ahhoz, hogy matematikai fogalmakkal beszéljünk róla.

Az eddigiekben mi úgy beszéltünk a Turing–gépről, mintha az egy valóságos, fizikailag is létező gép lenne. Ez nem hiba, mert a fentebb megadott konstrukcióval bárki megépítheti ezt a gépet, azonban a Turing–gép lényege éppen az, hogy egy formális, matematikai absztrakció, méghozzá az alábbi értelemben:

3.3.1. Definíció (Turing–gép). *Formálisan a Turing–gépet az alábbi rendezett kilencessel írhatjuk le:*

$$\mathcal{M} = (\Sigma, K, \delta, \mathrm{START}, H, \mathrm{HALT}, \mathrm{YES}, \mathrm{NO}, LR)$$

ahol az egyes komponensek rendre a következő jelentéssel bírnak:

- Σ *és* K *tetszőleges, véges (de nem üres) halmazok;* Σ *a külső ábécé, belőle kerülnek ki a szalagra írható szimbólumok;* K *a belső ábécé, az összes olyan belső állapot halmaza, amelyeket a gép futása során felvehet. Tehát az egész futás során minden egyes lépésben a gép valamely* $q \in K$ *állapotban van, és minden cella egy egyértelműen meghatározott* $\sigma \in \Sigma$ *jelet tartalmz. Kikötjük továbbá, hogy* $\Sigma \cap K = \emptyset$*, valamint azt is, hogy* $\# \in \Sigma$*, ahol „#" az üres szimbólum.*
- $\mathrm{START} \in K$ *a kezdő állapot;*
- $H \subset K$ *a termináló állapotok halmaza;*
- $\mathrm{HALT} \in H$ *a végállapot állapot;*
- $\mathrm{YES} \in H$ *az elfogadó állapot;*
- $\mathrm{NO} \in H$ *az elutasító állapot;*
- $LR = \{-1, 0, +1\}$ *a fej mozgási irányai.*
- *és végül a legfontosabb, a* $(q, \sigma) \to (q', \sigma', m)$ *típusú utasítások halmza, amit a*

  $$\delta : K \times \Sigma \to K \times \Sigma \times LR$$

  tetszőleges (úgynevezett átmeneti) függvény definiál, a gép programja.

Megjegyzések

- A Turing–gép szalagját az aktuális cellától (tehát amely felett a fej éppen tartózkodik) jobbra is és balra is végtelennek tekintjük. Ez az egyetlen olyan tény, amelyben a modell felülmúlja a valódi számítóeszközöket, mert a valóságban nem létezik végtelen nagy memória.
- Nyilvánvaló, hogy a számítások megkezdése előtt a szalagnak csak véges sok olyan cellája van, amely a „#"–tól különböző szimbólumot tartalmaz. Továbbá ha a gép véges sok lépés után befejezi a működését, akkor szintén csak véges számú olyan cella van, amely nem a „#" jelet tartalmazza, hiszen véges lépésben csak véges számú cellát érinthetett a fej.

– Ha a gép valamelyik H–beli állapotba kerül, akkor befejezi a működését. Ebből következik, hogy a δ függvény nincs értelmezve az összes $(q, s) \in H \times \Sigma$ pontban, vagyis azokban az állapototokban amelyek terminál állapotok. Viszont a $(K \setminus H) \times \Sigma$ halmazon a δ teljesen definiált, mert így elkerülhetjük azt, hogy a gép olyan konfigurációba kerüljön, ahol az aktuális belső állapota és az olvasott szimbólum alapján nem képes meghatározni a következő lépést.

– Korábban már említettük, hogy az $\mathcal{M}$ gép akkor fejezi be a működését, ha a H–beli állapotok valamelyikébe eljut. Ha ez az állapot a YES állapot, akkor úgy tekintjük, hogy a gép a bemenetet elfogadta, vagyis arra a problémára, amelyet a Σ ábécé szimbólumaival kódoltunk — legyen az a szimbólumsorozat most x —, és az indulás előtt inputként a szalagra írtunk, igen a válasz. Ezt a tényt a $\mathcal{M}(x) = \text{YES}$ formában jelöljük. Ennek analógiájára, ha a végállapot NO, akkor az x–el kódolt problémára nem a válasz, vagyis jelekkel kifejezve $\mathcal{M}(x) = \text{NO}$. Végül ha a gép a HALT állapotban áll meg, akkor a számítás sorozat eredményének azt a szalagon lévő szimbólum sorozatot tekintjük, amely azokból a cellákból áll, amelyeket a gép a futás során legalább egy alkalommal érintett. Ez a szimbólum sorozat — jelöljük most y–nal — egyértelműen meghatározott, és ezt az eredményt a $\mathcal{M}(x) = y$ formulával rövidítjük. Ezzel minden olyan lehetőséget áttekintettünk, amely a gép megállásához vezet. Olyan nem fordulhat elő, hogy a gép „elakad”, hiszen a δ függvényről kikötöttük, hogy a teminál állapotok kivételével teljesen definiált függvény a $K \times \Sigma$ halmazon, tehát a gép aktuális belső állapota és a fej által olvasott szimbólum minden egyes lépésben egyértelműen determinálja a végrehajtandó cselekvéssort. Ezek alapján már csak egyetlen olyan eset maradt, amely a gép által tanúsított lehetséges viselkedést leírja, ez pedig az a lehetőség, amikor a gép soha nem áll meg, hanem végtelen ciklusba keveredik. Ezt a $\mathcal{M}(x) = \infty$ jelöléssel fejezzük ki.

– A HALT, YES és NO állapotok megkülönböztetése csak kényelmi szempont és arra szolgál, hogy megkönnyítse a gép által végzett számítássorozat eredményének értelmezését. Elegendő lenne egyetlen egy terminál állapotot definiálni[21], és minden egyéb információt (bemenet elfogadása, elutasítása stb.) a gép egyszerűen felírhatna a szalagra valamilyen kódolt formában, mielőtt a HALT állapotba kerülne. Ezt például megteheti oly módon, hogy ha elfogadja a bemenetet, akkor nem a YES állapotba megy át, hanem egy olyan állapotba, amelyben

[21] Ahogyan az [SZ0] irodalom is teszi. (Vö. 390. o.)

megkeresei a szalagon lévő inputszó elejét. Ha ezt megtalálta, akkor átlép egy másik állapotba, amelyben elindul jobbra, és törli a szalag tartalmát. Ha ez is meg van, akkor újfent egy másik állapotba lép, és valamilyen speciális jelentéssel bíró jelet ír fel a szalagra, mondjuk „1"–et, és utána a HALT állapotba lép. Ennek analógiájára történik az elutasítás jelzése, azzal a különbséggel, hogy akkor egy másik szimbólumot például „0"–t ír a szalagra. Így tehát $\mathcal{M}(x) = 1$, ha a gép elfogadta a bemenetet, és $\mathcal{M}(x) = 0$, ha elutasította azt, és nincs szükség a YES és NO állapotok megkülönböztetésére. Azonban ez a megoldás sokkal körülményesebb, mint az, hogy több terminális állapotot adunk meg, és egyiküket „kinevezünk" elfogadó/elutasító állapotnak, amelybe a gép egyszerűen átlép és megáll, ha ez szükséges, íly módon jelezve a számítás eredményét. Természetesen a H halmaznak egyéb elmei is lehetnek, nem csak a HALT, YES és NO állapotok, de az iménti megfontolások azt mutatják, hogy ha $|H| = 1$, az is tökéletesen elegendő.

– A Σ és K halmazok tetszőleges halmazok, de végesek, és az LR halmaz is véges, így ezek megszámlálhatóak, és elemeiket kódolhatjuk természetes számokkal például a következő séma szerint: legyen $\Sigma' = \{1, 2, 3, \ldots, |\Sigma|\}$ és $K' = \{|\Sigma| + 1, |\Sigma| + 2, |\Sigma| + 3, \ldots, |\Sigma| + |K|\}$, valamint $LR' = \{|\Sigma|+|K|+1, |\Sigma|+|K|+2, |\Sigma|+|K|+3\}$. Nyilvánvaló, hogy Σ ekvivalens Σ'–vel, K ekvivalens K'–vel, LR pedig LR'-vel, ezért szabadon felcsrélhetőek. Az így kapott δ' függvény $\mathbb{N}^2 \to \mathbb{N}^3$ típusú lesz. Ez a kódólás csak az egyik a számtalan lehetésges kódólás közül, nem a módszer a lényeges, hanem az, hogy ilyen kódolás egyáltalán létezik. Ennek két szempontból van jelentősége: egyrészt rámutat arra, hogy a számítógépek működése, vagyis végsősoron a kiszámíthatóság elmélete és a természetes számok aritmetikája kapcsolatba hozható egymással[22]; másrészt ez teszi lehetővé, olyan gép létezését, amely bármely más gép működését képes szimulálni.

Az eddig elmondottak értelmében nyilvánvaló, hogy ha adott egy az 3.10 ábrán vázolt „valódi" Turing–gép, akkor könnyen meg tudjuk hozzá adni a 3.3.1 definícióban szereplő leírását, és fordítva, ha adott a gép $\mathcal{M}$ leírása, akkor meg tudjuk hozzá konstruálni a „valódi" gépet. Természetesen ha valaki nem jártas a mérnöki tudományokban, és nem szeret barkácsolni, akkor nem

[22]Pontosan ezt teszi az irodalomjegyzékben szereplő [M] könyv, amely bevezetést ad a rekurzív függvények elméletébe. Alapvetően arról van szó, hogy definiálunk bizonyos alapfüggvényeket, és alapoperátorokat, amelyek biztosan kiszámíthatóak algoritmussal, majd ezután azt a posztulátumot tekintjük az algoritmus definíciójának, hogy csak az a függvény kiszámítható, amely felépíthető az alapfüggvényekből az alapoperátorok felhasználásával.

szükséges a „valódi” gépet megépítenie, elegendő, ha egy olyan számítógépes porgramor ír, amely a gép $\mathcal{M}$ leírása alapján tetszőleges bemenet esetén szimulálja annak működését. Mi is ezt fogjuk tenni a következő fejezetben. Előtte azonban néhány, a Turing–gépekkel kapcsolatos további definíciót és tételt fogunk tárgyalni, amelyekre még a későbbiekben hivatkozni fogunk. Korábbi megállapodásunknak megfelelően a tételek bizonyítását általában nem közöljük, de mindig megjelöljük a forrást, ahol a bizonyítás megtalálható.

3.3.2. Bonyolultság

Célunk az eddigiekben az volt, hogy egy olyan számítóeszköz modelljét építsük fel, amely eléggé egyszerű ahhoz, hogy az általa végzett számítás sorozatot is könnyedén megragadhassuk a matematika fogalmaival. ***Csak az a kérdés, hogy mi az, ami alapján egy számítás sor hatékonyságát lemérhetjük. Nyilvánvaló, hogy valamilyen erőforrás igényt kellene alapul venni, és ezt az input valamilyen paraméterének függvényében kifejezni. Több ilyen erőforrás igényt is szoktak definiálni, amelyek közül a legtipikusabbak a futási idő, illetve a program által igényelt tár nagysága. A futási idő egy meglehetősen reletív mennyiség, hiszen nagymértékben függ az egyes számítógépek teljesítményétől, ezért helyette célszerűbb, az algoritmus végrehajtásához szükséges elemi lépések számával jellemzni a hatékonyságot, mert így már egy gépfüggetlen fogalomhoz tudjuk azt kötni.*** Mi is ez alapján fogunk vizsgálódni a továbbiakban, azonban még egy tényre fel kell hívnunk a figyelmet ezzel kapcsolatban. Itt nem arról van szó, hogy minden egyes futás estén, teljesen pontosan akarjuk tudni, hogy az hány lépésből állt. Nem konkrétan az algoritmus egyes futásait vizsgáljuk hanem általánosságban az algoritmus viselkedését, illetve még precízebben a lehetséges „legrosszabb” viselkedését, vagyis azt a lehetséges inputot amelyen az algoritmus a legtovább dolgozik. Ennek oka az, hogy konkrét esetekben előre semmiféle feltevéssel nem élhetünk az algoritmus futásával kapcsolatban, ezért ha korlátokat akarunk megadni a lépésszámra, akkor célszerű, mindig a legrosszabbat feltételezni. Egyszóval a lépések száma nagyságrendileg bír csak jelentőséggel a számunkra, és nem a valódi érték az, ami számít. Ezt az alapelvet pontosítja az alábbi definíció:

3.3.2. Definíció („nagy ordó”). *Legyenek adottak az* $f, g : \mathbb{N}^+ \to \mathbb{R}^+$ *függvények. Az*

$$f(n) = \mathcal{O}(g(n))$$

jelölés[23] *azt fejezi ki, hogy léteznek olyan* $c \in \mathbb{R}$ *és* $n_0 \in \mathbb{N}^+$ *számok, hogy minden* $n \geqq n_0$ *esetén*

$$f(n) \leqq c \cdot g(n)$$

teljesül.

A 3.3.2 definíció szemléletes tartalma az, hogy f legfeljebb olyan gyorsan növekszik, mint g. Éppen ezért a benne szereplő $g(n)$ függvény tökéletesen alkalmas arra, hogy a Turing–gép által produkált számítás sorozat lépés számára felső korlátot adjunk meg vele. Ezt felső korlátot a továbbiakban nevezzünk bonyolultságnak. Már csak az a kérdés, hogy milyen is ez a $g(n)$ függvény, és hogyan adhatjuk meg. Először is a benne szereplő n az nem más, mint az input hossza, vagyis a számítás megkezdése előtt a szalagra felírt x karaktersorozat szimbólumainak a száma, amit az $|x| = n$ formában szokás jelölni. Már említettük, hogy bennünket az érdekel, hogy milyen az algoritmus futásának legrosszabb esete. Ezt most egészítsük annyival, hogy

1. $g(n) = c$ – ***konstans***
2. $g(n) = \log(n)$ – ***logaritmikus***
3. $g(n) = n$ – ***lineáris***
4. $g(n) = n \cdot \log_a(n)$ – ***szemilineáris***
5. $g(n) = n^2$ – ***négyzetes***
6. $g(n) = n^k$ – ***polinomiális***
7. $g(n) = a^n$ – ***exponenciális***
8. $g(n) = n^n$ – ***hiperexponenciális***

3.11. ábra. Bonyolultsági osztályok

az is jelentőséggel bír vizsgálataink során, hogy milyen a g függvény viselkedése, abban az esetben, ha az input hossza minden határon túl nő, vagyis ha $|x| \to \infty$. Már csak az van hátra, hogy magát a függvényt megadjuk. Az igazság az, hogy a g számos „egyszerű” függvény típus lehet, és ezen függvény típusok alapján az algoritmusokat bonyolultsági osztályokba lehet

[23]Ejtsd: „$f(n)$ egyenlő nagyordó $g(n)$–nel”, vagy „f legfeljebb g nagyságrendű”.

besorolni (3.11 ábra). Az [SZ0] könyv grafikonnal és táblázattal is szemlélteti a 3.11 ábrán lévő bonyolultsági osztályok növekedését és egymáshoz való viszonyát.[24]

Mi csupán két bonyolultsági osztályt emelünk ki, ugyanis azt szeretnénk, hogy a „gyors algoritmus" illetve a „lassú algoritmus" nehezen meghatározható és homályos fogalmát valamilyen módon kötni tudjuk ezekhez a bonyolultsági osztályokhoz. Ismeretes, hogy egy polinom növekedési ütemét a legmagasabb fokú tag kitevője határozza meg, tehát ha $p(n) \in P[k]$ egy k–ad fokú polinom, akkor a 3.3.2 definíció értelmében $p(n) = \mathcal{O}\left(n^k\right)$. Ezt a bonyolultsági osztályt, ahol tehát egy polinommal tudjuk becsülni az algoritmus végrehajtásához szükséges lépések számát, **P**–vel jelöljük, és az ide tartozó problémákat polinomiális, vagy **P**–beli problémáknak nevezzük. Másrészt viszont analízisbeli eszközökkel bizonyítható, hogy minden polinom lassabban növekszik, mint bármely exponenciális függvény. Ezért a másik, számunkra fontossággal bíró bonyolultsági osztály az, amelybe azok a problémák tartoznak, amelyek megoldására szolgáló algoritmusoknak $\mathcal{O}(g(n))$ lépést kell tenniük, mielőtt terminálnának, ahol $g(n) = a^n$ (az alap tetszőleges). Ezeket a problémákat exponenciális bonyolultságúnak nevezzük.

Azért ezt a két osztályt emeltük ki, mert azokat az algoritmusokat, amelyek polinomiális, vagy annál lassabban növekvő függvényekkel jellemezhetőek, általában gyors algoritmusokként tartjuk nyilván, míg az exponenciális, vagy annál is gyorsaban növő bonyolultságú problémákat lassúnak tekintjük.

Számos olyan probléma ismeretes, amelyekről korábban azt gondolták, hogy exponenciális bonyolultságúak, aztán később kiderült, hogy létezik rájuk polinomiális időben végrehajtható algoritmus is. Egy tipikus példa, és az utóbbi idők egyik legjelenősebb felfedezése prímtesztelés problémája. Régóta az egyik fontos kérdés volt a matematikában hogy ha adott egy $n \in \mathbb{N}$ szám, akkor hogyan tudjuk meghatározni, hogy vannak–e valódi osztói ennek a számnak, vagyis a kérédés az, hogy n prímszám, vagy sem. Számos polinomiális algoritmust sikerült megadni, amely a számok egy–egy speciális osztályára kiválóan működött[25], de olyan determeinisztikus algoritmust, amely tetszőleges szám esetére működött volna, csak olyat sikerült találni, amelynek a bonyolultsága exponenciális volt. Az iménti mondatban fontos kitétel a „determinisztikus". Ismertek voltak ugyanis úgynevezett randomizált algoritmusok a probléma megoldására, amelyek lényege az, hogy nem azt mondják meg az inputon beadott számról, hogy az prímszám, vagy sem,

[24] [SZ0], 404. o.

[25] Ilyen például a *Lucas–Lehmer teszt*, amely csak az úgynevezett *Mersenne–prímekre* működik. Egy prímet ekkor nevezünk *Mersenne–prímnek*, ha felírható $2^p - 1$ alakban, ahol p prímszám. Például a $2^{13466917} - 1$ ilyen prímszám. (Ennek tízes számrendszerben való leírásához több, mint 4 millió számjegyre van szükség!!!)

hanem csak azt, hogy mekkora a valószínűsége annak, hogy prímszám, és mekkora a tévedés valószínűsége. Ellenben a determinisztikus algoritmusok nem játszanak kockajátékot, hanem 100%–os bizonyossággal adnak választ egy–egy kérdésre. A prímtesztelséssel kapcsolatban a probléma csak 2002–ben dőlt el véglegesen, ekkor sikerült megtalálni az első olyan deteminisztikus algoritmust, amely tetszőleges számról polinom időben képes eldönteni, hogy az adott szám prím, vagy sem. A részleteket az [AKS] cikkben találhatjuk meg.

Sajnos számos olyan probléma van, ahol nem volt ilyen szerencséje a kutatóknak, mint a prímtesztelés esetében. Vannak olyan problémák, amelyekre mind a mai napig nem sikerült polinomiális algoritmust találni, de azt sem sikerült bebizonyítani, hogy nincsen ilyen algoritmus. Azonban ismerjük a problémák egy olyan speciális osztályát, amelyekre jelenlegi ismereteink szerint csak exponenciális bonyolultságú algoritmus van, de sikerült kimutatni azt, hogy ezek a problémák egyfajta univerzalitással rendelkeznek, vagyis ha közülük bármelyikre sikerülne találni egy polinomiális algoritmust, akkor ez automatikusan maga után vonná, hogy ilyen algoritmus az osztály összes többi problémájára is létezik. Ezeket a problémákat **NP**–teljesnek nevezik, és az egyik legismertebb, a Boole–függvények kielégíthetőségének problémája. Ennek lényege, hogy adott egy $f(a_1, a_2, a_3, \ldots, a_n)$ logikai Boole–formula n változóval, és a kérdés az, hogy létezik–e olyan $T : \{a_1, a_2, a_3, \ldots, a_n\} \rightarrow \{Igaz, Hamis\}$ kiértékelése az $a_1, a_2, a_3, \ldots, a_n$ változóknak, hogy $f(a_1, a_2, a_3, \ldots, a_n) = Igaz$. Az egyik legtriviálisabb megoldása a feladatnak, ha az összes lehetséges T kiértékelést ellenőrizzük. Azonban nyilvánvaló, hogy n változó esetén ez $\mathcal{O}(2^n)$ lépést igényel, tehát exponenciálisan lassú.[26] Ez a probléma azért bír olyan nagy jelentőséggel, mert ez volt az első olyan, amelynek **NP**–teljességét sikerült bizonyítani.[27]

Természtsen ez csak egy a sok közül, mára már számos problémáról sikerült kimutatni, hogy **NP**–teljes.[28] A hozzájuk kapcsolódó általános kérdés pedig az, hogy vajon létezik–e rájuk gyors algoritmus, vagyis olyan, amely polinomiális lépésszámot igényel. Ez a híres $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ kérdés, amely mindmáig nyitott kérdés a matematikában, tehát sem bizonyítani, sem pedig cáfolni

[26]Megjegyezzük, hogy természetesen itt is olyan algoritmust keresünk, amely tetszőleges formulára alkalmazható. Ez azért fontos, mert itt is az a helyzet, mint a fentebb már említett prímtesztelés esetében, hogy a formulák bizonyos osztályaira létezik polinom időben végrehajtható algoritmus. Például a [P] irodalom közöl egy ilyen algoritmust, amely a Boole–formulák egy speciális szerkezettel rendelkező osztályára alkalmazható, és polinomiális lépészámot igényel. (88. o.)

[27]Lásd: [P], 196. o. illetve [SZ0], 416. o.

[28]Például az [LL] és [P] könyvek tételesen közölnek válogatást ezek közül a problémák közül, és **NP**–teljességüket is bizonyítják.

nem tudta eddig senki. Természetesen az is lehetséges, hogy a kérdés nem megválaszolható, mert független a matematika axiómáitól, de minden olyan kísérlet, amely ennek kimutatására irányult, eddig szintén kudarcba fulladt.

Mi csak a teljesség kedvéért említettük meg az iménti problémákat, hogy érzékeltessük, az algoritmusok futásideje egyáltalán nem mellékes dolog, hanem komoly problémákat vet fel, sőt ezek adják az algoritmuselméleti alapkutatás olyan központi kérdéseit, amelyeknek gyakorlati következményei is vannak. Nyilvánvaló, hogy ha egy problémáról beigazolódik, hogy algoritmikusan megoldhatatlan, akkor ez ellen nem sokat tehetünk azon kívűl, regisztráljuk ezt a tényt. Ellenben ha egy problémára van algoritmus, de az lassú, akkor központi kérdéssé válik, hogy vajon van–e gyorsabb.

Mi a továbbiakban nem nagyon foglalkozunk azzal, hogy konkrét algoritmusok bonyolultságát elemezzük, de erre számos példát találhatunk a [P] könyvben, amely a bolyolultsági osztályokat és a velük kapcsolatos kérdéseket is nagyon részletesen tárgyalja.

3.3.3. Turing–gépek és nyelvek

Láttuk, hogy a Turing–gép működése során valamilyen szimbólum sorozatból egy másik szimbólum sorozatot állít elő. Eddigi vizsgálódásaink alapján úgy tűnik, hogy Turing–gépek egyik nagy erőssége, a szófüggvének kiszámítása, éppen ezért minden feladatot, amelyet meg akarunk oldani a géppel vissza kell vezetnünk valamilyen módon erre a feladatra. Az alábbi definíciók pontosítják ezt a megállapítást.[29]

3.3.3. Definíció. *Legyen A tetszőleges halmaz. Ekkor A^*–gal jelöljük az A elemeiből képezhető összes lehetséges véges(!) hosszúságú jelsorozatok halmazát, és azt mondjuk, hogy A^* az A feletti szavak halmaza.*

Emlékeztetünk arra a halmazelméletből ismert tényre, hogy ha A véges vagy megszámlálható, akkor A^* is megszámlálható számosságú halmaz.

3.3.4. Definíció (Nyelv). *Legyen A tetszőleges halmaz. Ekkor az A^* halmaz bármely $L \subseteqq A^*$ részhlmazát nyelvnek nevezünk.*

3.3.5. Definíció (Rekurzív nyelv). *Legyen A tetszőleges halmaz továbbá legyen $L \subseteqq A^*$ egy nyelv. Ha van olyan $\mathcal{M}$ Turing–gép, hogy minden $x \in A^*$ bemenő szóra $\mathcal{M}(x) = \mathrm{YES}$, ha $x \in L$, és $\mathcal{M}(x) = \mathrm{NO}$, ha $x \notin L$, akkor azt mondjuk, hogy az $\mathcal{M}$ gép eldönti az L nyelvet, és L–t eldönthető, vagy rekurzív nyelvnek nevezzük.*

[29] A definíciókat a [P] irodalomból vettük át.

3.3.6. Definíció (Rekurzívan felsorolható nyelv). *Legyen A tetszőleges halmaz továbbá legyen $L \subseteqq A^*$ egy nyelv. Ha van olyan $\mathcal{M}$ Turing–gép, hogy minden $x \in A^*$ bemenő szóra $\mathcal{M}(x) = \mathrm{YES}$, ha $x \in L$, és $\mathcal{M}(x) = \infty$, ha $x \notin L$, akkor azt mondjuk, hogy az $\mathcal{M}$ gép felismeri vagy elfogadja az L nyelvet, és L–t rekurzívan felsorolható nyelvnek nevezzük.*

A rekurzívan felsorolható nyelvek halmazát az RE szimbólummal szokás jelölni, ami az angol *recursive enumerable* szónak a rövidítése. Vegyük észre az 3.3.6 definícióban rejlő érdekes asszimetriát. Ha az x szó eleme az L nyelvnek, akkor ez véges időn belül ki fog derülni, ha elindítjuk az $\mathcal{M}$ gépet az x bemenettel, de ha az x nem eleme az L halmaznak, akkor a gép végtelen ciklusba fog kerülni, és ezért soha sem fogjuk megtudni, hogy nincs benne a halmazban. Ugyanis ha már vártunk mondjuk $10^{(10^{1000})}$ évig, és eddig még nem állt meg a gép, ez semmilyen szempontból sem jelent garanciát arra nézve, hogy az elkövetkezendő 10 másodpercben nem fog valamelyik terminálló állapotba eljutni.

Az iménti definícióknak és a nyelveknek azért van olyan nagy jelentősége a Turing–gépek szempontjából a számítástudományban, mert minden problémát át lehet fogalamzni a „*melyek az L nyelv elemei*" típusú kérdéssé. Hogy ezt konkrétan hogyan lehet megtenni, arra láthatunk példákat a [T] könyvben[30], illetve Kiss Emil: *Hogy lehet, hogy nem lehet?* című cikkében, amely a [HA] könyvben olvasható; emellet általánosságban foglakozik a formális nyelvekkel az [FG] jegyzet és nagyon részletesen a [BI] könyv.

Itt csak egy dologra hívjuk fel a figyelmet ezzel kapcsolatban. Az imént említettük, hogy minden feladat átfogalmazható nyelvekkel kapcsolatos problémákká. Ha ezt tesszük, vagyis a — problémákat azonosítjuk a nyelvekkel — akkor egyszerű halmazelméleti megfonotlások alapján adódik, hogy vannak algoritmikusan megoldhatatlan kérdések is. Ugyanis a nyelvek halmaza nem megszámlálható, hanem kontinuum számosságú, ezzel szemben a Turing–gépek kódolhatóak véges hosszúságú jelsorozatokkal — ez lesz az alapja az univerzális Turing–gépnek amit a következő alfejezetben tárgyalunk —, vagyis megszámlálhatóan sokan vannak. Ez tehát azt jelenti, hogy nem tudjuk „párba állítani" a lehetséges problémákat a lehetséges megoldó algoritmusokkal (Turing–gépekkel), hanem lesznek olyanok, amelyekhez nem tudunk megoldó algoritmust hozzárandelni.

Az egyik ilyen tipikus probléma a ***megállási probléma***, ezt fogalmazza meg az alábbi 3.3.1 tétel:

3.3.1. Tétel (Megállási probléma). *Nem létezik olyan algoritmus, amely egy tetszőleges algoritmusról és inputról véges sok lépésben képes eldönteni,*

[30] [T], 43.–46. o.

hogy az adott algoritmus az adott inputtal elindítva (véges sok lépésben) terminál vagy sem.

Ez volt az egyik első olyan probléma, amely algoritmikus eldönthetetlenségét sikerült kimutatni[31], és azért játszik fontos szerepet a számítástudományban, mert számos más problémát vissza lehet erre vezetni, vagyis kimutatni, hogy ekvivalens a megállási problémával, és ezért algoritmikusan nem megoldható.[32]

3.3.4. Az univerzális Turing–gép

Az eddig elmondottak alapján úgy tűnhet, hogy a Turing–gép valójában csak szófüggvények kiszámítására, és igen–nem kérdések eldöntésére használható. Ráadásul még az is nyilvánvaló, hogy minden egyes probléma megoldására külön gépet kell szerkeszteni, hiszen a 3.3.1 definícióban szereplő Σ és K halmazok valamint a δ függvény egyértelműen meghatározzák azt, hogy milyen feladatot old meg a gép, vagyis melyik az a nyelv, amelyet felismer vagy eldönt stb. Ezzel szemben az általános számítógép látszólag ennél sokkal többre képes, hiszen ugyanaz a gép számos probléma megoldására alkalmazható. Azonban éppen erre a vélekedésre cáfol rá az alábbi tétel:

3.3.2. Tétel (Univerzális TM). *Létezik olyan $\mathcal{M}_0$ Turing–gép, amely képes arra, hogy tetszőleges másik Turing–gép működését szimulálja.*

Az 3.3.2 tétel[33] az, amely az amúgy látszólag korlátolt és behatárolt képességekkel rendelkező primitív Turing–gépet pillanatok alatt az általános célú számítógépekkel egy kategóriába emeli. A tétel tulajdonképpen azon alapul, hogy tetszőleges $\mathcal{M}$ Turing–gép működését leíró szabálysorozat (vagyis a δ függvény) és a gép x input szava ügyes trükkökkel kódolható olyan formában, hogy ha ezt a kódolt információt felírjuk az $\mathcal{M}_0$ gép szalagjára, akkor az éppen úgy fog működni, mint az $\mathcal{M}$ gép, tehát ugyan azt a nyelvet fogja felismerni/eldönteni stb.

Az iménti megállapítások látszólag elméleti jellegűek, azonban a gyakorlatban már hosszú ideje alkalmazzák azt az alapelvet, hogy az egyik számítógép felhasználható egy másik számítógép működésének szimulálására. Egy tipikus példa erre a JAVA programozási nyelv esete, amely manapság már a legelterjedtebb programnyelv az Internetes alkalmazások fejlesztésére. Röviden összefoglalva a történet a következő volt: 1990–es évek elején a SUN

[31] A bizonyítást lásd az alábbi helyeken: [SZ0], 385. o.; [RISz], 216. o.; [P], 66. o.

[32] Vö. [RISz] 215.–228. o.; [SZ0], 386. o.; [LL], 21.–27. o.

[33] A bizonyítást, vagyis azt, hogy miként lehet megkonstruálni az $\mathcal{M}_0$ gépet, lásd az alábbi helyeken: [P], 65. o.; [RISz], 205. o.; [BI], 208. o.; [T], 132.–137. o.

MICROSYSTEM kutatói kitaláltak egy számítógép architektúrát, vagyis megadták a műszaki leírását, utasítás készletét stb. egy új számítógépnek. Ez a gép a JVM (*Java Virtual Machine*) elnevezést kapta. Megalkottak hozzá egy új programnyelvet is, a JAVA–t, amely egy a C++ nyelvhez nagyon hasonló programnyelv, de ősével szemben nagy előnye, hogy sokkal biztonságosabban használható Internetes alkalmazások fejlesztésére. Írtak egy fordítóprogramot is, amely a JAVA nyelvű forráskódot lefordította a JVM gép utasításkészletére, hogy azon futtatható legyen. Azonban maga a gép soha nem került legyártásra (nem is ez volt a cél), hanem helyette készítettek egy interpretert, vagyis értelmező programot, amely arra képes, hogy a JAVA nyelven megírt, és a JVM gépre lefordított programkódot értelmezze, és végrehajtsa, vagyis szimulálja a JVM gép működését. Ezt az értelmezőt aztán a legkülönfélébb számítógéptípusokra és operációs rendszerekre elkészítették, és így ma már szinte mindenhol lehet JAVA programokat futtatni, anélkül, hogy maga a gép, amely ilyen programok futtatására rendeltetett egyáltalán (a maga fizikai valóságában) létezne. De természetesen más példát is felhozhatunk erre, ugyanis ma már se szeri se száma az olyan programoknak, amelyek IBM PC számítógépen futnak, és arra írták őket, hogy a COMMODORE64–es gépeket szimulálják, vagyis ha egy ilyen programot elindítunk a gépünkön, akkor lehetővé válik, hogy azt ugyanúgy használjuk, mintha COMMODORE64 lenne.

Az univerzális Turing–géppel kapcsolatban van még egy fogalom, amely ide kívánkozik. Ezt fogalmazza meg a következő definíció:

3.3.7. Definíció (Turing–teljesség). *Turing–teljesnek nevezünk azokat az objektumokat, amelynek megvan az a tulajdonsága, hogy minden olyan probléma, amely kiszámítható/megoldható Turing–géppel, az kiszámítható/megoldható az adott objektummal is.*

Az iménti megfogalmazás némi magyarázatra szorul. Először is a benne szereplő „objektum” szó semmi esetre sem cserélendő fel az objektum orientált programozásból imert objektum fogalommal. Láttuk, hogy a Turing–gépek az algoritmusok szinonímáiként tekinthetőek. Viszont azt is láttuk, hogy van univerzális Turing–gép, tehát olyan, amely tetszőleges másik gépet szimulálni tud. A Turing–gép önmagában egy elméleti absztrakció, de számos olyan objektum van, amellyel relizálni tudjuk a működését. Például építhetünk egy olyan automata szerkezetet az univerzális gép megvalósítására, amely a 3.10 ábrán látható. Ha egy ilyen automatát építünk, akkor az Turing–teljes lesz, mert tetszőleges Turing–gépekkel megoldható problémát megoldhatunk vele. De ugyanezen az elven írhatunk olyan számítógépes programot, amely az univerzális gép működését valósítja meg [34], vagy alkothatunk olyan elektro-

[34] Ezt tesszük a következő fejezetben

nikai áramkört, amely ugyanezt teszi. De például a programozási nyelvek is Turing–teljesek, mert bármely Turing–gépekkel kiszámítható problémát is veszünk alapul, a programozási nyelveken megfogalmazhatunk olyan lépéssort, amely ugyanúgy elvezet a feleadt megoldásához, mintha a géppel oldanánk meg. Természetesen nem minden mesterséges nyelv Turing–teljes. Az olyan egyszerű leíró nyelevek, mint a HTML nem Turing–teljsesk, tehát pusztán ennek a nyelvnek szintatkitkáját felhsználva nem lehet leírni egy tetszőleges Turing–gép által kiszámítható probléma megoldásához vezető lépéssort.

Látjuk tehát, hogy nagyon széles azoknak az objektumoknak a skálája, amelyek képesek a Turing–gépek működését realizálni. Egy igen érdekes, és első hallásra meghökkentő példáját találhatjuk meg ennek a [V] cikkben, amelynek címe *C++ Templates are Turing Complete*. A template–k, vagy magyarul a sablonok olyan szintaktikai bővítése a C++ nyelvnek, amely segítségével könnyedén lehet különböző adattípusokat kezelő függvény– és osztálycsoportokat készíteni, anélkül, hogy minden egyes adattípusra külön elkészítenénk a függvények és osztályok definícióját. Az említett cikkben elolvashatjuk, hogy miként lehet egy olyan, szándékosan hibás programot megírni C++ nyelven a sablonok segítségével, hogy a fordítás közben keletkező hibaüzenetek egymásutánja egy tetszőleges Turing–gép futásának egymásutáni konfigurációit kódolja. Valószínű, hogy a C++ nyelvet megtervező informatikus mérnököknek és matematikusoknak nem ált szándékában, hogy a nyelv ilyen tulajdonságokkal is rendelkezzen, hanem ez véletlenül alakult így. Azonban mindez egy nagyon is pozitív, és önbizalmat növelő üzenetet hordoz a (kezdő és tapasztalt) programózok számára: nem baj, ha a programunk nem működik, mert elviekben egy teljesen működésképtelen és hibás program is alkamas lehet ugyanolyan feladatok megoldására, mint a működő programok. A továbbiakban már csak a fantáziánk szab határt annak, hogy elképzeljük, a jövőben még mi mindenről fog kiderülni, hogy valójában nem más, mint egy Turing–gép, csak ezt eddig nem tudtuk róla.

3.3.5. A Church–tézis

Most térjünk vissza eredeti kérdésünkhöz, vagyis annak vizsgálatához, hogy tulajdonképpen mi is az algoritmus. A fejezet eddigi részében már elejtettünk néhány utalást erre vonatkozólag, de még nem neveztük meg egyértelműen, hogy miről is van szó, és nem adtunk definíciót az algoritmus fogalmára. Nos ezt most sem fogjuk megtenni, ugyanis az algoritmus fogalmára nem tudunk olyan egyértelmű, és explicit definíciót megadni, mint más matematikai fogalmak esetében. Csak azt tudjuk megmondani (a definíció helyett), hogy mi az a konstrukció, amelyről ésszerű azt feltételezni, hogy az algoritmus fogalmával azonos. Nem nehéz kitalálni, hogy ez a konstrukció a Turing–gép.

Ezt a feltevést először Alonzo Church fogalmazta meg, ezért nevezik Church–tézisnek. A lényege pedig az, hogy bármely olyan probléma, amely megoldására létezik algoritmus, az megoldható Turing–géppel is, és fordítva, csak azok a problémák algoritmikusan megoldhatóak, amelyekre Turing–gép szerkeszthető.

Első hallásra ez nagyon merész kijelentésnek tűnik, és úgy gondolhatnánk, hogy a magaszintű programnyelevek bonyolult elágazásokat, és iterációkat is tartalmazó szintaxisával sokkal több problémát tudunk megfogalmazni és megoldani, mint a primitív Turing–géppel. Azonban midezidáig senki sem tudott megadni olyan problémát, amelyekre a hagyományos programnyelveken lehet programot írni, viszont Turingg–géppel nem oldhatóak meg. A következő fejezetben vázolni fogjuk azt, hogy a magaszintű nyelveken való programozás és algoritmizálás minden alapelemének (iterációk, ciklusok, szelekciók stb.) meg tudunk feleteni egy–egy Turing–gépet, és ezáltal az ilyen programokat átfogalamazni a Turing–gépek szintjére.

A Church–tézissel kapcsolatban ne hagyjuk figyelmen kívűl azt a tényt, hogy ez az állítás nem tétel, hanem egyfajta posztulátum, tehát olyan állítás, amelynek igazságát sem bizonyítani, sem pedig cáfolni nem lehet, de mégis elfogadjuk hogy igaz, azért, hogy a tudományos megismerés útján megtehessük a következő lépést. Felmerülhet bennünk a kétely, hogy esetleg a modellben van a hiba, és ha másként közelítjük meg a számítástudomány alapjait, akkor nem állja meg a helyét a Church–tézis. Azonban eddig minden számítóeszköz modellről kiderült, hogy az sem tud többet a Turing–gépeknél.

Van a Church–tézisenk egy másik formája, amely ugyanazt állítja, amit fentebb már leírtunk, csak másként van megfogalmazva. Ez így hangzik: ***Az algoritmusok és időigényük matematikai eszközökkel való modellezésére tett bármely ésszerű kísérlet szükségképpen olyan modellhez és hozzá tartózó időigény–fogalomhoz vezet, amely polinomiálisan ekvivalens a Turing–géppel.***[35]

Valóban eddig minden modell esetében kiderült, hogy ha a modellben valamely probléma megoldásához $\mathcal{O}(f(n))$ lépésre van szükség, akkor lehet olyan Turing–gépet szerkeszteni, amely ugyanazt a feladatot $\mathcal{O}(g(f(n)))$ lépésben oldja meg, ahol a g egy polinom. A legtipikusabb példa a közvetlen hozzáférésű gép, amelyet korábban RAM–program modell néven említettünk.[36] De hasonló a helyzet a Turing–gép különféle módosított változatainak esetében is. Például az olyan gép, ahol a szalag csak az egyik irányban végtelen, vagy az, ahol nem csak egy, hanem több darab szalaggal van ellátva

[35]Vö. [P], 41. o.

[36]Vö. ezen dolgozat 31. oldalával, valamint a RAM–gépekkel kapcsolatban lásd az alábbi helyeket: [P], 40.–50. o.; [RISz] 239.–345. o.; [LL], 12.–16. o.

a gép, illetve az olyan konstrukció, ahol a jobbra és balra végtelen szalagot egy kétdimenziós négyzetrács helyettesíti, mind olyan módosításai Turing–gépnek, amelyek működése polinom időben szimulálható az általunk megadott Turing–gép modellel, tehát az ilyen jellegű kiterjesztések nem növelik a gép számítási erejét.

Ezek tehát azok a megfontolások, amelyek indokot szolgáltatnak arra, hogy „higgyünk" a Church–tézisben, és elfogadjuk annak igazságát. Ha ezt tesszük, akkor egyenlőség jelet teszünk a Turing–gépek és az algoritmusok közé. Márpedig a legtöbb szakirodalom manapság elfogadja a Church–tézis igazságát, mégpedig azért, mert a más irányú feltételezések még nem vezettek eredményre. A fenti állításban azonban nem véletlenül szerepel az a kifejezés, hogy „ésszerű". Lehet kidolgozni ugyanis olyan modelleket is, amelyek valamilyen olyan „trükköt" alkalmaznak, amelyek megvalósítása nem teljesen evidens.

Ezen alapul például a *nemdeterminisztikus Turing–gép* modellje, amely teljesen megegyezzik a 3.3.1 definícióban leírt hagyományos (vagy másként determinisztikusnak nevezett) Turing–géppel azzal az egyetlen, de lényeges különbséggel, hogy a δ függvény nem a $K \times \Sigma \times LR$ halmazba képez, hanem annak hatványhalmazába, tehát $\delta : K \times \Sigma \to \mathcal{P}(K \times \Sigma \times LR)$ típusú, vagyis minden egyes lépésben az olvasott szimbólumhoz és a gép belső állapotához a $K \times \Sigma \times LR$ halmaz egy részhalmaza van hozzárendelve, tehát nincs egyértelműen meghatározva a következő lépés. Ebben az esetben a gép működését úgy képzeljük el, hogy az „választ" a lehetséges lépések közül, hogy melyiket hajtja végre, és akkor mondjuk, hogy a gép megold egy problémát $\mathcal{O}(g(n))$ lépésben, ha elméletileg(!) létezik egy olyan futása, amely a megoldáshoz vezet és ez legfeljebb $\mathcal{O}(g(n))$ lépésből áll. Ennek a konstrukciónak meglehetősen nagy a számítási ereje, azonban nem világos, hogy mi alapján realizálja a lehetőségek közti választását, és éppen ez a tény az, ami a Church–tézis szempontjából „ésszerűtlenné" teszi a modellt, ami most nem úgy értendő, hogy irracionális, hanem úgy, hogy a gép jellegénél fogva egy tisztán elméleti konstrukció, és ennél fogva valódi számítások elvégzésére nem alkalmas.

Ne értsük félre az iménti megállapítást: nyilvánvaló, hogy a hagyományos, determinisztikus Turing–gép képes szimulálni a nemdeterminisztikus változat működését, azáltal, hogy a minden egyes lépés során lehetséges választásokat szisztematikusan végignézi és szimulálja, hogy miként folytatódna a nemdeterminisztikus gép futása az egyes választások megvalósulása esetén. Azonban egyszerű halamzelméleti megfonolások alapján az is nyilvánvaló, hogy ezt csak exponenciális időveszteséggel tudja végrehajtani.[37] Vagyis ez

[37] Vö. [SZ0], 413. o.

a modell polinomiálisan nem ekvivalens a determinisztikus Turing–géppel, és látszólag kivonja magát a Church–tézis érvénye alól, azonban a gyakorlatban teljesen hasznavehetetlen, mert az ilyen gépeknek meg van az a hátrányos tulajdonságuk, hogy nem léteznek.

Vegyük észre azt is, hogy az iménti megfontolások által felvetett probléma tulajdonképpen nem más, mint a már említett $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ kérdés, ami itt úgy jelenik meg, hogy vajon lehetésges–e a nemdeterminisztikus Turing–gépek működésének determinisztikus Turing–géppel való szimulálása polinomiális időben?

4. fejezet

A Turing–gépek szemléltetése

Az előző fejezetben részletesen elemzetük a valódi számítógépek logikai–struktúrális szerkezetét, és megvizsgáltunk a Turing–gépnek nevezett modellt, amelyről azt állítottuk — a Church–tézis igazságába vetett bizalomra támaszkodva —, hogy minden olyan feladat, amely egyáltalán algoritmussal megoldható, ahhoz meg tudjuk szerkeszteni azt a Turing–gépet, amely megoldja ugyanezt a feladatot. A Turing–géphez olyan módon jutottunk el a Neumann–elvű gépből kiindulva, hogy elkezdtük szimplifikálni ezen utóbbinak a szerkezetét, amíg az már annyira leegyszerűsödött, hogy egyszerű matematikai fogalmakkal is meg tudtuk ragadni az íly módon nyert modell által végzett számítássorozatot. Azonban visszatekintve úgy tűnhet, hogy az áttérés a Neumann–gépről a Turing–gépre, nem teljesen nyilvánvaló módon történt meg, és felmerülhet bennünk a kétely, hogy esetleg valamilyen olyan lépést hajtottunk végre, amely miatt a Turing–gép a maga egyszerűségével kevesebb probléma megoldására lesz csak alkalmas, mint a valódi gépek. Ebben a fejezetben ezt a kérdést szeretnénk tisztázni azáltal, hogy megvizsgáljuk, hogy azok az alapelemek amelyekből a valódi gépekre megírt programok felépülnek, valamilyen módon átfogalmazhatóak–e Turing–gépre. Ha igen, akkor ez azzal egyenértékű, hogy minden, ami a valódi számítógépek programozási nyelvein leírható, az leírható a Turing–gépek primitív nyelvén is, tehát egyik sem tud többet a másiknál. Ha pedig elfogadjuk azt, hogy a valódi gépeken elvileg mindenre tudunk programot írni, amire egyáltalán lehetséges, akkor ez igaz kell hogy legyen a Turing–gépekre is, tehát ezeken is minden algoritmus megvalósítható. Természetesen ezt szigorú értelemben véve nem tudjuk bebizonyítani, hiszen ez nem más, mint a Church–tézis, csak most éppen megint egy más formájában mondtuk ki, de nem is célunk az, hogy ezt bizonyítsuk, csupán a szemléletbeli megfontolásokra támaszkodva szeretnénk tovább mélyíteni azt a „hitet”, hogy a Church–tézis igaz. Azért, hogy kissé elszakadjuk a pusztán elméleti jellegű tárgyalástól, a továbbiakban

a segítségünkre lesz az *Automaton* nevű számítógépes program, amely abból a célból készült, hogy ha egy tetszőleges Turing–gép szerkezeti leírását megadjuk neki, akkor ez alapján szimulálja az adott gép működését tetszőleges bemenet esetére.

4.1. A Turing–gépek szerkezetének leírása grafokkal

A programozási technikák és a programozás módszertanának fejlődése során számos olyan módszer alakult ki, amely segít a programozóknak a feladatot leíró algoritmus megfogalmazásában, és az algoritmus szerkezetének áttekintésében. A legismertebb ilyen eszközök a folyamatábra, a Jackson–féle funkcionális leírás, a Chapin–féle struktogram, vagy a mondatszerkezeti leírás, a pszeudó–kód. Programozója válogatja, hogy ki melyiket szereti, és melyik módszer segítségével tudja egyetlen pillantással átfogni egy–egy algoritmus szerkezetét, de az tény, hogy a legtöbb ember vizuális típus, és ezért egy ábra, vagy diagram több információt hordozhat, mint a leírt programkód. Ha ezt az elvet kiterjesztjük a Turing–gépekre, akkor magától adódik az ötlet, hogy ezek szerkezetét is valamilyen ábrával próbáljuk meg leírni. A szakirodalomban több ilyen módszert is találhatunk, mi most egy olyat fogunk tárgyalni, amely könnyen áttekinthető, és a már említett *Automaton* program is az itt bemutatásra kerülő leírásban kéri a Turing–gépek szerkezeti megadását, és aztán ebből generálja magának a gép programját, tehát a δ függvényt.

4.1.1. A Turing–gép bináris növelésre

Vegyünk egy egyszerű problémát, amelyet megpróbálunk Turing–géppel megvalósítani. A feladat legyen az, hogy egy kettes számrendszerben felírt számot inkrementálunk, vagyis megnövelünk eggyel. Erre a feladatra több féle Turing–gépet is szerkeszthetünk, mi a 4.1 ábrán látható konstrukciót vesszük alapul, de mielőtt rátérnénk a részletes tárgyalásra, tennünk kell néhány megjegyzést.

Először is fentebb azt modtuk, hogy a δ függvényt teljesen definiáltnak tekintjük a $(K \setminus H) \times \Sigma$ halmazon. Ennek ellenére a továbbiakban a δ függvény értékét csak azokon a $(q, \sigma) \in K \times \Sigma$ helyeken fogjuk megadni, amelyeknek tényleges szerepe van a számításokban. Tehát ha mondjuk semilyen körülmények között nem fordulhat elő, hogy a gép a q_i állapotban a σ_k szimbólumot olvassa, akkor a (q_i, σ_k) helyen nem fogjuk külön megadni a δ értékét. Ezeken

a helyeken, és minden egyéb helyen, ahol δ–t nem definiáljuk külön, úgy értelmezük, hogy $\delta(q_i, \sigma_k) = (q_i, \sigma_k, 0)$, vagyis ha a gép mégis a σ_k jelet olvassa q_i állapotában, akkor ugyanazt a jelet írja vissza, amit olvasott, belső állapotát nem változtatja meg, és a fej helyben marad. Nem nehéz észre venni, hogy ha ez bekövetkezik, akkor az végtelen ciklus lesz. Továbbá a 3.3.1 definícióban szereplő HALT, YES és NO állapotok közül is csak azokat adjuk meg a konkrét Turing–gépek leírásában, amelyeket a gép tényleg használ, a többit egyszerűen elhagyjuk.

Vegyük szemügyre a 4.1 ábrán látható Turing–gépet. Ez nem csinál egyebet, mint azt, hogy a szalagjára felírt kettes számrendszerbeli számot megnöveli eggyel, majd utána megáll. A gép START állapotban kezdi meg

$\Sigma = \{0, 1, \#\}$
$K = \{s, h, q_0\}$
$H = \{h\}$
START=s
HALT=h

	q	σ	$\delta(q, \sigma)$
1.	s	0	$(s, 0, +1)$
2.	s	1	$(s, 1, +1)$
3.	s	#	$(q_0, \#, -1)$
4.	q_0	0	$(h, 1, 0)$
5.	q_0	1	$(q_0, 0, -1)$
6.	q_0	#	$(h, 1, 0)$

4.1. ábra. A bináris inkrementálást megavalósító Turing–gép

működését, és a számítások megkezdésekor fej azon a legbaloldalibb cellán áll, amely a „#”–tól különböző jelet tartalmaz. Ha a gép START állapotában 0–t vagy 1–et olvas, akkor csupán azt teszi, hogy az ugyanazt a jelet, amit olvasott, visszaírja a cellába, és eggyel jobbra lépteti a mutatót (1. és 2. szabály). Mivel a szalagra felírt szám véges sok 0–ból és 1–ből áll, előbb–utóbb a jobbraléptetések miatt ezek elfogynak, és a gép rátalál egy üres cellára, vagyis olyanra, amely a „#” jelet tartalmazza. Ha ez bekövetkezik, akkor a gép a „#” jel elolvasásának hatására aktuális belső állapotát (ami a START), q_0 állapotra változtatja — amely állapotot nevezhetünk „dolgozó” állapotnak, hiszen ebben az állapotban végzi a tényleges műveleteket —, az aktuális cella tartalmát nem változtatja meg, és balra lépve egyett rááll a bináris szám legkisebb helyiértékű bitjére (3. szabály). Látható tehát, hogy az 1., 2. és 3. szabályok csupán azért kerültek be a progrmba, hogy

a legkisebb helyiértékű bitet a gép megkereshesse. Említettük azt, hogy a kezdő cella, bármelyik cella lehet, amelyre a futás megkezdése előtt ráállítjuk a fejet. Ha nem a legbaloldalibb „#"–tól különböző cellára állítjuk a fejet, az elején, hanem a legjobboldalibbra, akkor az 1., 2. és 3. szabályra nem is lenne szükség. Hogy mégis bekerültek a programba, annak két oka van. Egyrészt ez is mutatja, hogy az a tény, hogy nem tettünk kikötést a kezdő cella pozíciójára vonatkozólag, nem jelent semmiféle meg nem engedett lépést, mert ügyesen kitalált szabályokkal tetszőleges celláról tetszőleges másik cellára tudjuk mozgatni a fejet anélkül, hogy eközben a szalag tartalma megváltozna; másrészt — és ez a lényegesebb — azt szerettük volna demonstrálni, hogy az ilyen jellegű szabályokkal teljes mértékben megvalósítható és helyettesíthető a Neumann–gépnél tárgyalt közvetlen memória címzés. Látható, hogy a Turing–gép definíciójában szereplő $LR = \{-1, 0, +1\}$ halmaz, vagyis a „balra lép", „helyben marad", „ jobbra lép" primitív háromelemű címzési mód, bár sokkal bonyolultabban, de képes azt helyettesíteni, mint amikor az akárhány címes processzorral közvetlen módon „közöljük" az operandusok címét. Nyilvánvaló, hogy ez a címzés sokkal komplexebb, mit a másik, de nem az a lényeg, hogy bonyolult vagy sem, hanem az, hogy lehetséges. Úgy tűnik tehát, hogy ezen a ponton, vagyis a memória címzés szemszögéből nem találunk olyan indokot, ami arra engedne következtetni, hogy a Turing–gép kevesebbet tud, mint a valódi számítógépek, és hogy egyik ne lenne visszavezethető a másikra.

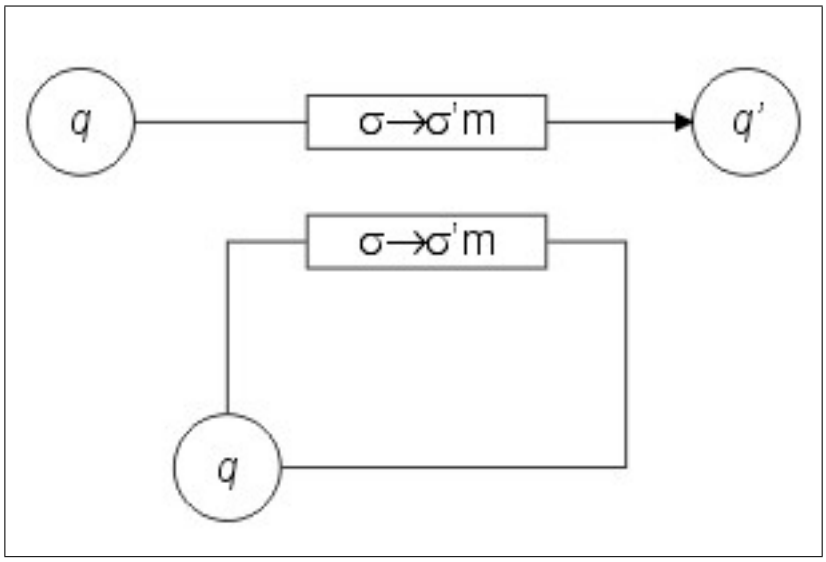
Térjünk vissza a bináris inkrementálást megvalósító gépünkhöz, és kövessük tovább a lehetséges futását. Addig jutottunk, hogy a fej elgyalogolt a legkisebb helyiértékű bitre, és felvette a q_0, vagyis a „dolgozó" állapotot. Ebben az állapotban a gép elkezdi balra mozgatni a fejet, és feldolgozni az inputot. A kérdés csak az, hogy mi módon? Ha vetünk egy pillantást a 20. oldalon közölt 3.2 ábrára, amely a bináris aritmetika művelettábláit tartalmazza, akkor leolvasható, hogy kettes számrendszerben $1 + 0 = 0 + 1 = 0$ és $1 + 1 = 10$. Amikor 0–hoz adunk hozzá 1–et, akkor ezt triviális módon tudja a gép végrehajtani, mert egyszerűen a 0–t lecseréli 1–re. Viszont az $1 + 1 = 10$ már problematikusabb, mert egy lépésben csak egy jelet tud változtatni. Éppen ezért célszerűbb ezt úgy értelmezni, hogy $1 + 1 = 0$ és(!) átvitel keletkezik, méghozzá az átvitel 1. Tehát ha a gép 1–et olvas, akkor azt cserélje le 0–ra, de ezzel még nem fejezheti be mert az átvitelt hozzá kell adni az eggyel nagyobb helyiértékű bithez, ami szintén lehet 0 vagy 1, de bármi is az, a cselekvéssor már viszzavezethető az imént tárgyalt esetek valamelyikére. Összefoglalva tehát a gép a következő módon működik a q_0 állapotban: ha 1–et olvas, akkor azt lecseréli 0–ra, állapotát nem változtatja és balra lép (5. szabály); ezzel a balralépegetéssel megkeresi az első 0–t, és ha ezt olvassa, akkor azt lecseréli 1–re és a HALT állapotba kerülve megáll (4.

szabály). Az utóbbi esetben, mivel a gép úgyis megáll, nincs jelentősége annak, hogy a fej merre mozdul, így akár helyben is maradhat. Úgy tűnik, hogy ezzel befejeztük a gép működésének leírását, azonban ha nem vagyunk eléggé körültekintőek, akkor további komplikációk adódhatnak. Mi történik például akkor, ha a következő input szót írjuk a szalagra: $\ldots\#\#11111111\#\#\ldots$? Ha végig követjük a gép működését ezen a bemeneten, akkor a következő történik: a fej elmegy a szó végére, és átlép q_0 állapotba; mivel minden egyes szimbólum az 1, ezért ezeket elkezdi lecserélgetni 0–ra, és minden egyes csere után balra lép, ahogyan ezt az 5. szabály előírja; így viszont előbb–utóbb eljut a szó legelejére a legnagyobb helyiértékű bithez, de mivel az is 1, lecsréli 0–ra és ismét balra lépve egy üres cellára áll, a q_0 állapotot továbbra is megtartva; a probléma a következő lépésben áll elő, amikor is a „#" szimbólumot olvassa, mert az eddigiekben erre az esetre még nem adtuk meg a végrehajtandó cselekvéssort. Két lehetőségünk van: vagy azt mondjuk, hogy $\delta(q_0, \#) = (h, \#, 0)$, tehát a gép ebben az esetben megáll, és nem változtaja meg az üres szimbólumot, amit úgy értelmezhetünk, hogy az eredmény nem fér el az ábrázolásra felhasznált cellákban, tehát túlcsordulás történt; vagypedig azt mondjuk, hogy $\delta(q_0, \#) = (h, 1, 0)$, amely esetben helyes eredményt kapunk, de az output szó hossza nagyobb, mint az input szó hossza. Mi ez utóbbi esetet választottuk, ezt fogalmazza meg a 6. szabály. Ezzel már tényleg megkaptuk azt az $\mathcal{M}$ Turing–gépet, amelyre tetszőleges $x \in \{0,1\}^*$ szó esetén $\mathcal{M}(x) = x + 1$, ahol a „+" művelet jelenését az 3.2 ábrán látható táblázat definiálja.

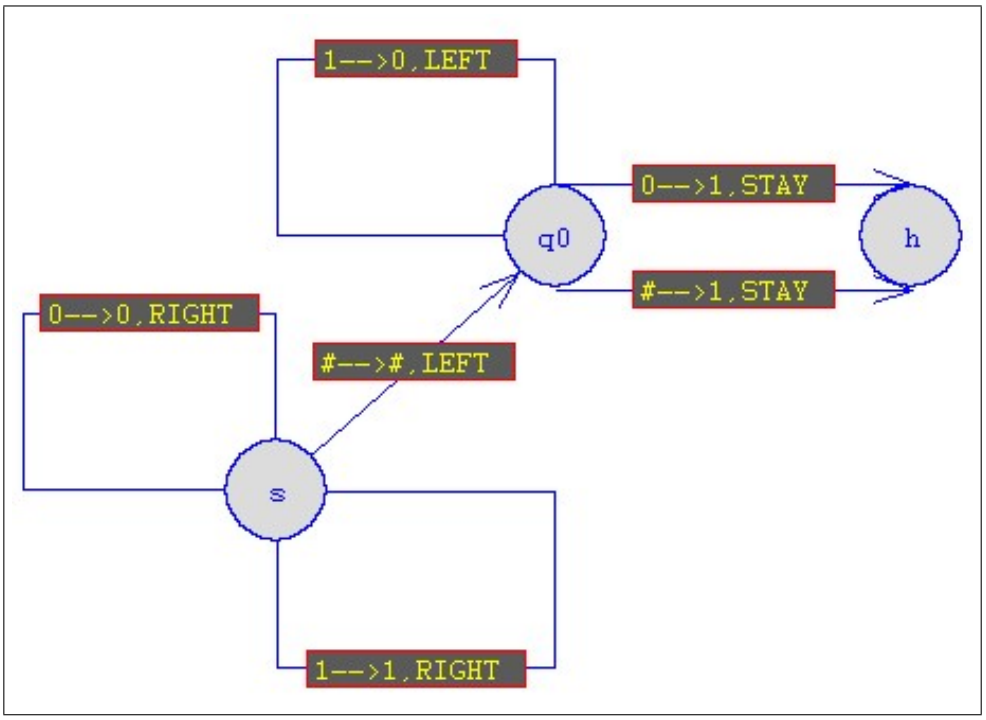
4.1.2. Turing–gráfok

Most térjünk vissza arra, hogy miként lehet a Turing–gépek programjának szerkezetét a programozásból már ismert folyamatábrához hasonló módon felrajzolni. Magasszintű algoritmizálás esetén a folyamatábra több komponensből áll, mert az algoritmusok alapelemeit, tehát az iterációkat, ciklusokat, feltételes elágazásokat stb. külön–külön komponens jeleníti meg az ábrán. Viszont a Turing–gépek szerkezete sokkal egyszerűbb a valódi gépek szerkezeténél, és ebből az következik, hogy a Turing–programok „folyamatábrája" is egyszerűbb, és sokkal kevesebb komponensből felépíthető. ***Nevezetesen két darab ilyen komponens van, méghozzá a kör, illetve az azokat összekötő vonalak.*** Ha a köröket csomópontoknak tekintjük, a vonalakat pedig éleknek, akkor világos, hogy a Turing–programok folyamatábrája egy gráf. Ezt a gráfot úgy kell felépíteni, hogy valamilyen módon tükrözze a gép programját, tehát a $(q, \sigma) \to (q', \sigma', m)$ típusú utasítások összességét, vagy ha úgy tetszik más felírással, a $\delta(q, \sigma) = (q', \sigma', m)$ értékeket.

Ennek érdekében minden $q \in K$ ***belső állapotnak feleltessünk***



4.2. ábra. A Turing–gráfok alapelemei



4.3. ábra. A bináris inkrementálást megvalósító gép Turing–gráfja

meg egy (és csakis egy) csomópontot a gráfban. Majd ezek után, ha $\delta(q,\sigma) = (q',\sigma',m)$***, akkor ezt úgy ábrázoljuk, hogy rajzolunk egy irányított élet a*** q ***csomópontból a*** q' ***csomópontba (ez jelenti az átmenetet a*** q ***állpotból a*** q' ***állapotba), és az élre címkeként ráírjuk, hogy milyen cselekvéssort ír elő az adott utasítás. Cselekvéssor alatt most azt értjük, hogy mi az olvasott szimbólum, mit kell visszaírni a cellába és milyen irányba kell lépni, amelyeket a*** $\sigma \to \sigma' m$ ***formában írunk fel az élre.*** Ezt a sémát láthatjuk a 4.2 ábrán felül, amelyet balról jobbra haladva úgy értelmezünk, hogy ha a gép q állapotban van és a σ szimbólumot olvassa, akkor σ' jelet ír vissza, m irányba lép és q' állapotba kerül.

Természetesen előfordulhat, hogy a gép nem változtatja meg a belső állapotát valamely lépésben, tehát az is megengedett dolog, hogy az iménti utasításban $q = q'$ legyen. Ha a gép egy ilyen utasításhoz érkezik, akkor a végrehajtás előtt és a végrehajtás után is a belső állapota q. Ez a gráfban úgy jelenik meg, mint egy olyan él, amelynek forrás– és célcsomópontja azonos, tehát hurokél. Mivel ezekben az esetekben az él irányítása nem bír jelentőséggel, hurokélek esetében nem nyilat rajzolunk, hanem egy négyzet alakú törött vonalat, ahogyan azt a 4.2 ábrán alul láthatjuk, és amely értelmezése az imént elmondottak alapján már nyilvánvaló.

A Turing–program minden egyes utasításnak megfeleltetünk tehát egy a 4.2 ábrán látható csomópont–él–csomópont hármast, és ilyenekből építjük fel a teljes progrmaszerkezetet leíró gráfot. Az ilyen ábrázolást többféleképpen is nevezik az egyes szakirodalmak, a legelterjetteb elnevezés az ***állapot diagram*** és a ***Turing–gráf***. Mi a továbbiakben ezeket az elnevezéseket felváltva fogjuk használni.

Az eddig elmondottakat felhasználva már meg tudjuk rajzolni a 4.1 ábrán lévő bináris inkrementálást végrehajtó Turing–gép állapot diagramját. Ezt láthatjuk a 4.3 ábrán. Ha figyelmesen tanulmányozzuk az ábrát, könnyen beazonosíthatjuk az egyes utasításokat a gráf megfelelő részgráfjaival. A 4.3 ábrát az *Automaton* program segítségével rajzoltuk meg, és a jelölések pusztán annyiban térnek el az eddig használttól, hogy az $LR = \{-1, 0, +1\}$ halmaz elmeihez, amelyek a fej mozgását leírják, rendre a LEFT, STAY és RIGHT elnevezéseket rendeltük hozzá.[1]

[1] Az elnevezés nem lényeges, de mivel fentebb a START, HALT, YES és NO állapotok esetében az angol nyelvű szakirodalomból átvett szavakat használtuk, akkor leszünk következetesek, ha továbbra is ragaszkodunk ehhez a konvencióhoz, és angol nyelvű elnevezéseket használunk.

4.1.3. A Turing–gép bináris csökkentésre

Látjuk, hogy a Turing–gráfok már valamelyest szemléletesebbek, mint ha pusztán a δ függvény értékeit tekintjük. A továbbiakban kiterjesztjük a gráfok szerkezetét felépítő alapelemeket, hogy méginkább megtudjuk velük ragadni a Turing–program algoritmusát. Azonban, az elméleti fejtegetés helyett, kapcsolódjunk ennek során megint egy példához. A feladat legyen az, hogy készítsük el azt a Turing–gépet, amely az inkrementálás művelet inverzét, a dekrementálást valósítja meg bináris számrendszerben, tehát ha egy kettes számrendszerbeli számot felírunk a gép szalagjára, akkor a gép csökkenti a szám értékét eggyel. Láttuk, hogy az inkerementálás esetében sem volt túlságosan bonyolult a gép, és ebből sejthetjük, hogy a mostani feladat sem különösebben nehéz. Természetesen itt is igaz az, hogy több lehetséges mód kínálkozik a megvalósításra. Mi most egy olyan gépet fogunk definiálni, amely kihasználja a bináris aritmetika egy érdekes matematikai tulajdonságát, nevezetesen azt, hogy kettes számrendszerben a kivonás visszavezethető az összeadásra.

Ha adott egy bináris szám — jelöljük x–szel —, akkor a szám komplemensének nevezzük azt a számot, amelyet úgy kapunk x–ből, hogy annak minden bitjét az ellentettjére változtatjuk, tehát 0–ból 1–et csinálnk, és 1–ből 0–t. A továbbiakban jelöljük az x komplemensét az $\overline{x}$ szimbólummal. Például ha $x = 1011001$, akkor $\overline{x} = 0100110$. A bináris aritmetika érdekes tulajdonsága, hogy a ha az x–ből ki akrjuk vonni az y számot[2], akkor ezt megtehetjük úgy, hogy az x komplemenséhez hozzáadjuk y–t, és az eredményt újra komplementáljuk, vagyis ez azt jelenti, hogy $x - y = \overline{\overline{x} + y}$. Ezzel a kivonást máris visszavezettük az összeadásra.[3]

Pontosan ezt az elvet fogjuk alkalmazni a dekrementálást megvalósító gép estében: a gép először végig szalad az inputszón, és annak minden bitjét ellentetjére változtatja; azután a kapott eredményhez hozzáad egyet, amelyet például a 4.1 ábrán már definiált géppel lehet megvalósítani; ha elvégezete az inkrementálást, visszaszalad a szalag elejére, és újfent komplementálja a biteket, ha pedig ezzel is végzett, akkor megáll.

A gép leírása a 4.4 ábrán, gráfja pedig a 4.5 ábrán látható. A gép ebben az esetben is a START állapotban kezdi meg a működést, és a fej induláskor

[2]Az egyszerűség kedvéért tegyük fel, hogy $y \leqq x$.

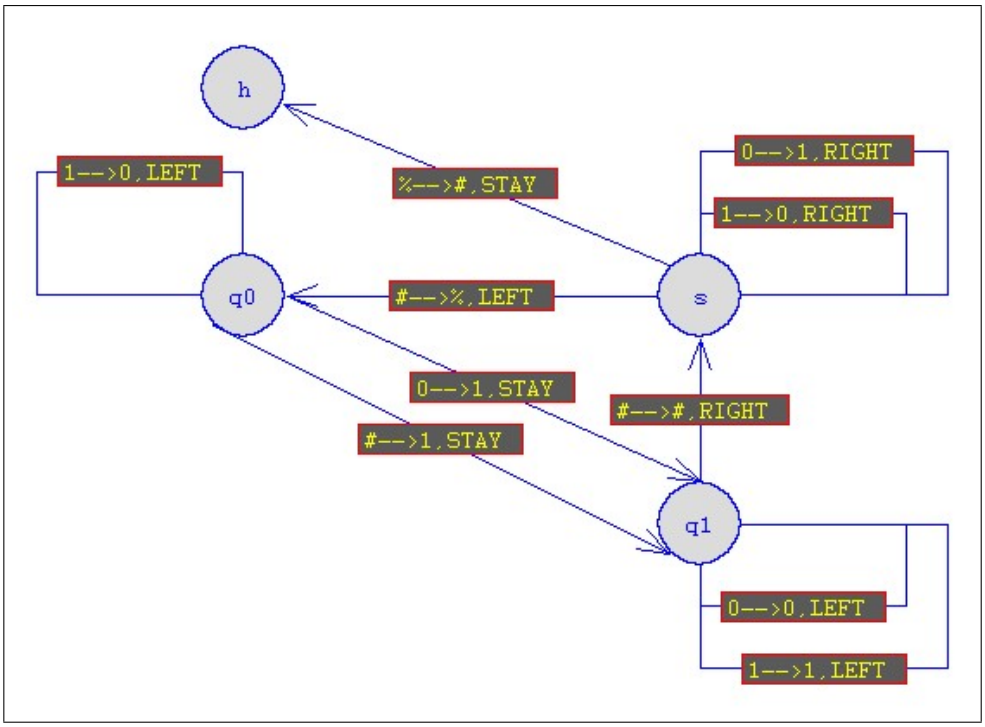
[3]Könnyű utánagondolni, hogy ez miért van így, de mi csupán annyit jegyzünk meg, hogy tulajdonképpen tetszőleges számrendszerben is működik a dolog, ha a komplementálás műveletét felcseréljük a -1–gyel való szorzásra. Ekkor a kivonás itt is visszavezethető az összeadásra, mert $x - y = -(-x + y)$, ami a fenti képlet analógiája. Ezek alapján a bináris szám komplemensét felfoghatjuk a szám mínuszegyszereseként is. Valóban erről van szó, mert egyes számítógépek ezen a módon ábrázolják a negatív számokat. A számábrázolásokról a [HF] könyvben olvashatunk bővebben.

most is az első olyan legbaloldalibb cellán áll, amely nem az üres szimbólumot tartalmazza. Az inkrementálást megvalósító gépen a START állapotnak csak annyi volt a szerepe, hogy a fejet a legkisebb helyiértékű bitre mozgassa. Most is ez történik, de miközben a fej START állapotban jobbra lépdel, egyből végrehajtja az inputszó komplementálását, tehát minden 1–ből 0–t csinál, és minden 0–ból 1–et (1. és 2. szabály). Ha a fej ebben az állapotban elért a szó végére, akkor ezt onnan tudja, hogy a „#" jelet olvassa. Korábban az inkrementáló gép nem változtatta meg ezt a szimbólumot, mostani gépünk viszont lecseréli „%"–ra (ennek oka hamarosan világossá válik), és balra lép egyet a legkisebb helyiértékű bitre, valamint az állapotát is q_0–ra változtatja (3. szabály). A q_0 állapot a „dolgozó állapot", ebben az állapotban hajtja végre a már komplementált inputszó értékének eggyel való megnövelését, teljesen azon elvek szerint, amelyet az inkrementáló gépnél alkalmaztunk (5., 6. és 7. szabály). Ezen utóbbi gép, ha elvégezte a növelést, akkor egyszerűen HALT állapotba lépett át és megállt. Most nem ez történik, hanem a gép felveszi a q_1 állpotot (6. és 7. szabály). A q_1 állapot arra szolgál, hogy a fejet visszaküldje az inputszó legelejére, abból a célból, hogy a gép újra elvégezze eredmény komplementálását. Ennek megfelelően ha a gép q_1 állapotban 0–t vagy 1–et olvas, akkor azokat nem változtatja meg, hanem csak balra lépdel (8. és 9. szabály). Ha a szó elejét elérte, erről

$\Sigma = \{0, 1, \#, \%\}$
$K = \{s, h, q_0, q_1\}$
$H = \{h\}$
START=s
HALT=h

	q	σ	$\delta(q, \sigma)$
1.	s	0	$(s, 1, +1)$
2.	s	1	$(s, 0, +1)$
3.	s	#	$(q_0, \%, -1)$
4.	s	%	$(h, \#, 0)$
5.	q_0	1	$(q_0, 0, -1)$
6.	q_0	0	$(q_1, 1, 0)$
7.	q_0	#	$(q_1, 1, 0)$
8.	q_1	0	$(q_1, 0, -1)$
9.	q_1	1	$(q_1, 1, -1)$
10.	q_1	#	$(s, \#, +1)$

4.4. ábra. A bináris dekrementálást megavalósító Turing–gép



4.5. ábra. A bináris dekrementálást megvalósító gép Turing–gráfja

onnan értesül, hogy a „#” jelet olvassa. Ezt is érintetlenül hagyja, de jobbra lép egyet, és visszatér START állapotba, mert — emlékezzünk vissza — ebben az állapotban végezi a komplementálást (10. szabály). Vegyük észre azt is, hogy újra előállt ugyanaz a szituáció, mint ami a futás megkezdésének legelején volt: újra START állapotban vagyunk, és a fej megint a legelső cellán áll. Vagyis megint egy komplementálást végrehajtó ciklus kezdődik, de most ha a fej a START állapotban elér a szó végére, akkor nem a „#” jelet, hanem a „%” szimbólumot fogja ott találni, amit korábban írt oda. Viszont ennek a jelnek a hatására a START állapotú gép megáll, de előtte még a jelet visszacseréli „#”–ra (4. szabály). Itt válik érthetővé, hogy az első komplemenáló ciklus alkalmával miért cserélte le a gép a szó végét jelző „#” szimbólumot valami másra. Ha nem ezt tette volna, akkor a második komplementálás után újra elkezdődött volna a szalagon lévő szó eggyel való megnövelése, és könnyű belátni, hogy ezzel egy végtelen ciklusba léptünk volna. A „%” szimbólum tehát azt a kódolt üzenetet hordozza a gép számára, hogy „korábban már jártál itt, most állj meg”.

4.1.4. A Turing–gépek, mint modulok és alprogramok

Ezzel tehát már eljutottunk addig, hogy van egy Turing–gépünk, amivel meg tudunk növelni egy bináris számot eggyel, és van egy másik Turing–gépünk, amivel csökkenteni tudunk egy bináris számot szintén eggyel. Azonban a gépek szerkezetét leíró gráfok még mindig eléggé nehezen értelmezhetőek, mivel csak két komponensből épülnek fel: körökből, és vonalakból. Ha megnézzük a második gépünket, vagy pontosabban azt a feladatot, aminek végrehjtására terveztük, akkor azonnal feltűnik, hogy a végrehajtás — kissé elnagyolva — négy, egyértelműen elkülöníthető lépésre bontható fel, amelyek az alábbiak:

1. Komplementálás
2. Növelés eggyel
3. Visszalépegetés a szó elejére
4. Komplementálás

A „Növelés eggyel” lépés végrehajtása azért nem okozott gondot, mert már korábban megterveztük az a gépet, amely ezt megcsinálja, és az ott alkalmazott ötletet ültettük át a második gépre. Ezen a ponton azonban felmerül bennünk egy másik lehetőség a megvalósításra: mi lenne, ha a második gépre nem programoznánk meg az inkrementálást, hanem — mivel az azt végrehajtó gép már úgyis létezik —, akkor, amikor ezt meg kell csinálni, második gépünk egyszerűen átadná a vezérlést az első gépnek, vagyis „meghívná”,

mint egy alprogramot vagy szubrutint, hogy elvégezze helyette a munkát, és ha kész, akkor az inkrementáló gép visszadja a vezérlést, a második gép pedig folyatja a futását.

Ezzel megtettük az első lépést a felé, hogy a Turing–gépek programozására alkamazzuk a magasszintű programozásból már ismert procedurális– és moduláris absztrakció alapelvét. A procedurális absztrakció ott azt jelentette, hogy egy komplex feladatot több, kisebb részfeladatra bontunk szét, és az egyes részfeladatok megoldására külön alprogramokat írunk. A főprogram feladata ezek után arra korlátozódik, hogy a megfelelő sorrendben meghívja, vagyis lefuttatja ezeket az alprogramokat, és így jut el a teljes feladat megoldásához. Ez arra volt jó a magasszintű programnyelvek esetében, hogy egyrészt a részfeladatok megoldására szolgáló kódrészek sokkal áttekinthetőbbek és egyszerűbbek, mint mondjuk egy 1000 soros főprogram, ami csökkenti a hibalehetőséget, másrész pedig ha egy–egy részfeladatot többször kell végrehajtani, akkor elegendő hívni az alprogramot, és a megvalósító kód is csak egyszer szerepel a programban, ami azt eredményezi, hogy a program rövidebb, tehát takarékoskodik a memóriával és egyéb erőforrásokkal.

Ehhez szorosan kapcsolódva a moduláris absztrakció azt jelentette, hogy azokat a már elkészült eljárásokat, vagyis alprogramokat, amelyekre gyakran, a legkülönfélébb programokban is szükség van, összegyűjtjük egy modulba, vagy más szóhasználattal könyvtárba (library), és ha szükséges, akkor ezeket a könyvtárakat egyszerűen hozzákapcsoljuk a programunkhoz.

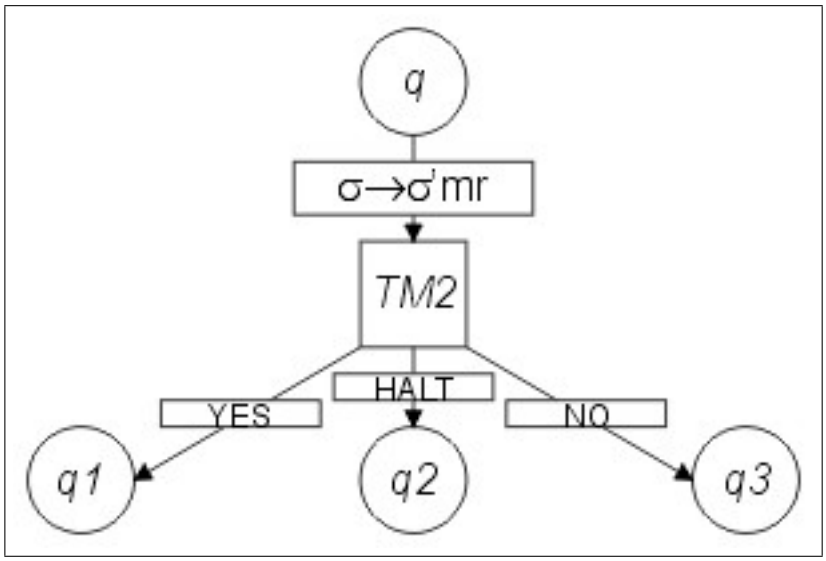
Nem látszik akadálya annak, hogy ezt a Turing–programok esetében is alkalmazzuk. Ha van egy olyan programunk, ami Turing–teljes, tehát tetszőleges Turing–gép működését képes szimulálni (az *Automaton* program ilyen), akkor a gyakori feladatok végrehajtására külön Turing–gépeket definiálunk, és valamilyen formában megőrizzük azokat a későbbi használat esetére, és ha szükséges, akkor csak hivatkozunk rájuk.

Azonban ezt valamilyen módon a Turing–gráfok szintjén is meg kell jelenítenünk, ezért szükséges, hogy kissé módosítsuk az eddigi ábrázolást. ***A gráf csomópontjaiban eddig csak körök szerepeltek, most vezessünk be egy új komponenst, a négyzetet. Ez jelöli, hogy az adott csomópont nem egy állapotot jelent, hanem egy másik Turing–gépre való hivatkozást, vagyis egy végrehajtandó alprogramot.*** A 4.6 ábra mutatja, hogy miként jelennek meg a négyzetek, vagyis a más gépekre való hivatkozások a gráfban. ***Az ábra értelmezése fentről lefele haladva a következő: ha a gép*** q ***állapotban van és a*** σ ***szimbólumot olvassa, akkor*** σ'***–t ír vissza,*** m ***irányú mozgást végez a fejjel, és — itt jön a lényeg — nem vesz fel közvetlenül egy másik belső állapotot,***

mint korábban, hanem elindítja a* TM2 *nevű gépet, méghozzá r ***állapotban, ezt jelenti az élre írt szabály utolsó komponense. Ezzel azonban még nincs vége a folyamatnak. A* TM2 *gép korábbi megállapodásunk értelmében három lehetséges állapotban fejezheti be a működését. Ha* HALT *állapotban áll meg, akkor a hívó gép új belső állapota* q_2 *lesz, ha* YES *állapotban áll meg, akkor a hívó gép* q_1 *állapotot vesz fel, és végül, ha* NO *állapotban fejezi be futását, akkor a hívó gép* q_3 *állapotba kerül.***

Természetesen a TM2 gépnek is van egy gráfja, amely szintén tartalmazhat további gépekre való hivatkozást, annak hasonlatosságára, mint amikor a magasszintű nyelvek esetében egy–egy alprogram további alprogramok „szolgáltatását" veszi igénybe. Az is megoldható, hogy miközben az egyik gép a másik gépre hivatkozik, ugyanakkor a másik gép is tartalmazza a rá hivatkozó gépet a gráfjában. Ebben az esetben kölcsönös rekurziót valósítanak meg, és természetesen annak sincs akadálya, hogy egy gép önmagát tartalmazza a saját gráfjában, vagyis önmagát hívja és önrekurziót hajtson végre.

Az iménti jelölés bevezetésével tulajdonképpen nem tettünk egyebet, mint egy gráfot részgráfokra bontottunk fel, és az egyes részgráfokat önálló elnevezéssel illettük és külön Turing–gépekként kezeltük őket. Ezzel elértük azt, hogy egy-egy gráf már sokkal egyszerűbb és áttekinthetőbb szerkezettel rendelkezzen, mint korábban. Természetesen ez az eljárás fordított irányban is végig játszható. Ha van egy olyan gépünk, amely más gépekre hivatkozik,



4.6. ábra. Hivatkozás más gépekre a Turing–gráfban

akkor a gépnek a gráfját, és a hivatkozott gépek gráfját egyesíteni tudjuk. Ha visszatérünk a 4.6 ábrához, akkor ott például meg tudjuk tenni azt, hogy a TM2 gép gráfját bekepcsoljuk a hívó gép gráfjába oly módon, hogy ugyanazt az élet, amely most a TM2 gép csomópontjára mutat átirányítjuk az újonnan beillesztett gráf r csomópontjára, a TM2 gép HALT, YES és NO állapotait pedig egyszerűen elhagyjuk, és megfeleltetjük nekik a q_1, q_2 és q_3 állapotokat. Ez utóbbi úgy értendő, hogy ha az egyesítés előtt a TM2 gép mondjuk a YES állapotba lépett volna, most minden olyan élet, amely korábban ebbe a csomópontba mutatott, átirányítjuk a q_1 állapotnak megfelelő csomópontra. Ezzel elértük, hogy a TM2 gép korábban önálló gráfja a hívó gép gráfjának részgráfja legyen. Természetesen felmerülhetnek olyan problémák, amelyekkel érdemes foglalkoznunk. Ha egyesítünk két, vagy több gépet, akkor az egyesítés után keletkezett gép külső ábécéje és állapothalmaza a korábbi gépek külső ábécéinek és állapothalmazainak uniójaként áll elő. De mi van például abban az esetben ha a beillesztendő gép tartalmaz olyan nevű belső állapotot, mint amilyen már eredetileg is szerepel a gép belső állapotai között. Ez azért probléma, mert megsérti azt a kikötést, hogy a gráfban minden csomópont csak egyszer szerepelhet, vagyis sérül az egyértelműség. A problémát úgy oldhatjuk meg, hogy megkeressük az azonos belsőállapotokat, és valamelyik halmazban átnevezzük őket, oly módon, hogy az elnevezések egyértelműek és különbözőek legyenek. Tehetjük például azt, hogy ha mondjuk van két gépünk, legyenek ezek TM1 és TM2, és mindegyiküknek van q nevű belső állapota, akkor erre az állapotra az első gép esetében TM1.q formában hivatkozunk, a másik gép esetében pedig TM2.q formában, ezzel is jelezve, hogy az első gép q állapota semmi esetre sem azonos a második gép q állapotával. (Ez a konvenció ahhoz hasonló, mint amikor az objektum–orientált progrmozásban egy objektumon belül deklarált változóra hivatkozunk. Ott is megengdett, hogy két különböző objektumnak azonos nevű mezői legyenek, a fordító program mégis meg tudja különböztetni a szintaktikai szabályok alapján, hogy két különböző változóról van szó.)

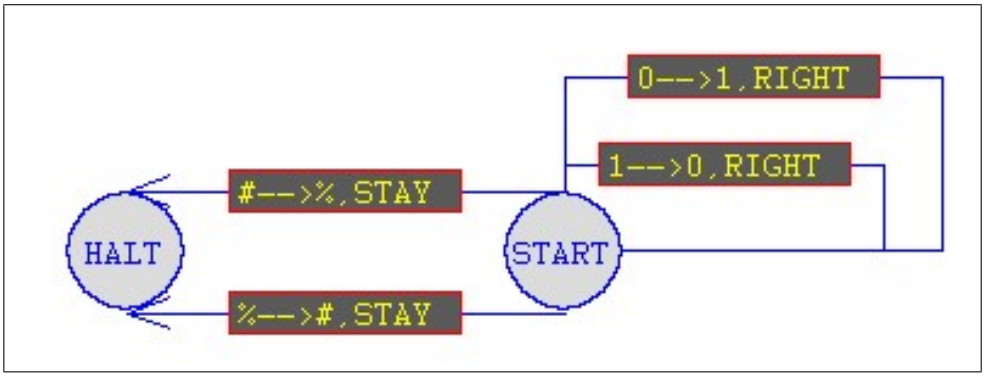
Még egy utolsó megjegyzést kell tennünk az elmondottakkal kapcsolatban. Az 4.6 ábrán látható séma esetében nem teszünk kikötést arra vonatkozólag, hogy az r állapot, amelyben a TM1 gép megkezdi a működését, szükségképpen a START állapot legyen. Ezzel némileg ellentmondunk a korábbiaknak, mert ott azt vettük alapul, hogy a START állapot egy kitüntetett állapot, és a gép mindig ebben az állapotban kezdi meg a működését. A továbbiakban ezt nem követeljük meg, hanem ha egy gépet alprogramként hívunk, akkor abba a gépbe tetszőleges állapotban „beleugorhatunk”. Ez csupán egy egyszerűsítő feltevés, ahhoz hasonlóan, ahogyan azt sem követeljük meg, hogy az író–olvasó fej mindig az input első celláján álljon, mert láttuk, hogy ügye-

sen kitalált szabályokkal bárhova mozgathatjuk a fejet a szalag tartalmának megváltoztatása nélkül. Ehhez hasonlóan, ha megkövetelnénk, hogy egy gép mindig START állapotban induljon, tudnánk megadni olyan szabályokat, hogy a gép bármely más tetszőleges r állapotot felvegyen anélkül, hogy ez szalag tartalmának illetve a fej pozíciójának megváltozását idézné elő. Éppen ezért egyszerűbb, ha közvetlenül az r állapotba „ugrunk", azonban ez az engedmény csak arra jó, hogy a mi dolgunkat megkönnyítse.

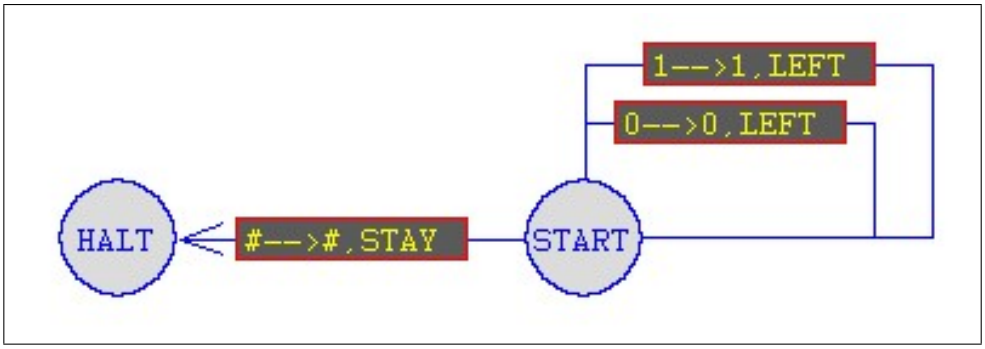
Az elmondottakat figyelembevéve és alkalmazva most már sokkal áttekinthetőbb folyamatábrákat tudunk megadni a Turing–programok algoritmusainak leírására. Az ismertetett ábrázolási formában kell megadni az *Automaton* program számára egy-egy Turing–gép leírását, és a program a megrajzolt gráfot bejárva generálja a gép programját, tehát a δ függvényt. A program használatát a következő fejezetben ismertetjük, azonban még előtte álljon itt a korábban már elkészített bináris dekrementálást megvalósító gép módosított változata, amelyet az imént ismertetett szabályok szerint építünk fel.

4.1.5. Moduláris Turing–gép bináris csökkentésre

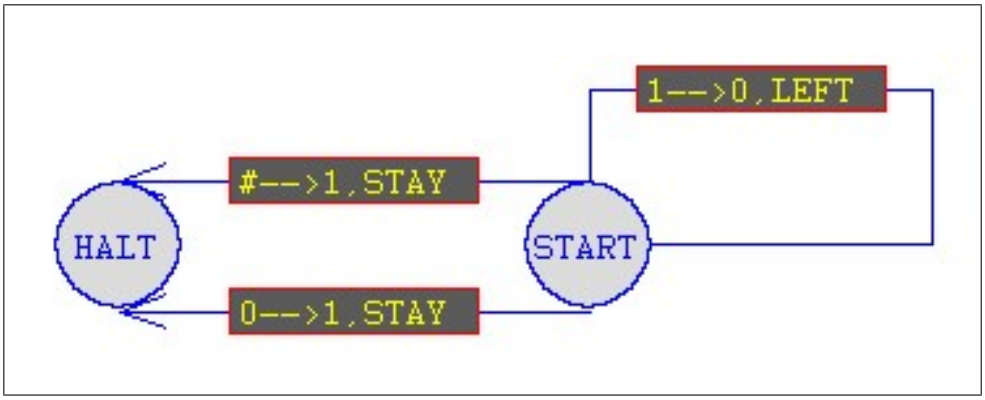
Először is készítsünk önálló Turing–gépeket, amelyek a komplementálást, a balra lépegetést és az egyel való növelést hajtják végre. Legyen a neve ezeknek a gépeknek rendre KOMPLEMENS (4.7 ábra), BALRAMEGY (4.8 ábra) és BININC (4.9 ábra). A legszembetűnőbb dolog, hogy ezeknek a gépeknek a szerkezete meglehetősen egyszerű, ami nem véletlen, hiszen az egyes feladatok, amit megvalósítanak, az sem túlzottan bonyolult. Nem elemezzük részletesen ezen gépek működését, az eddig elmondottak alapján könnyű értelmezni a gráfokat. Viszont annál érdekesebb az a gráf, amely ezeket a gépeket egységbe fogja és amely tulajdonképpen a főprogram szerepét betölti. Nevezzük ezt a gépet MAIN–nek (4.10 ábra). A START állapotból kiindulva kövessük végig a gép működését a gráfban, a nyilak mentén haladva! Ez a gép is START állapotban kezdi meg a működést, és a fej az inputszó legbaloldalibb szimbólumán áll kezdetben. Függetlenül attól, hogy ez a szimbólum 1 vagy 0, a MAIN nem változtatja meg ezeket, hanem változatlanul visszaírja, és a fejet is helyben hagyja. Rögtön átadja a vezérlést a KOMPLEMENS gépnek, méghozzá START állapotban indítja azt el. Ha vetünk egypillantást ennek a gépnek a gráfjára, azonnal látható, hogy ugyanazt a tevékenységsort valósítja meg, amelyet korábban már leírtunk, tehát komplementálja az inputot és egy „%" szimbólumot ír a végére. Ezután HALT állapotban megáll. Viszont ez utóbbi automatikusan maga után vonja, hogy a MAIN gép belső állpota t_1–re változik. Eddig tehát addig jutottunk, hogy a fej az input vége utáni cellán áll, amely a „%" jelet tartalmazza. és a főprogramot megvalósító



4.7. ábra. A Komplemens gép gráfja

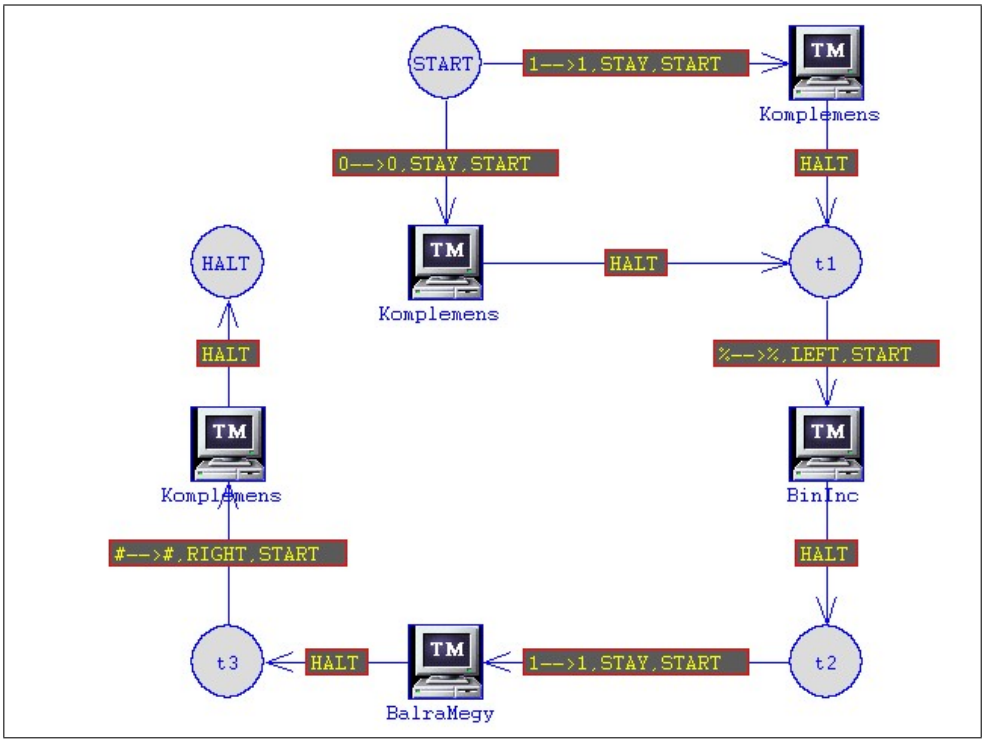


4.8. ábra. A BalraMegy gép gráfja



4.9. ábra. A BinInc gép gráfja

gép felvette a t_1 állapotot. Ebben az állapotban a „%”–ot nem változtatja meg, csupán a fejet mozdítja eggyel balra, és máris átadja az irányítást a BinInc gépnek, amely végrehajtja az eggyel való növelést. Ha ezen utóbbi gép HALT állapotban terminál, akkor a Main a t_2 állapotot veszi fel. Ha



4.10. ábra. A Main gép gráfja

megnézzük a BinInc gépet, akkor látható, hogy megállása pillanatában az aktuális cella mindenképpen 1–et fog tartalmazni. Vagyis amikor a főgép a t_2 állapotba lép át, és folytatja a tevékenységét, teljesen biztos, hogy a fej 1–et fog olvasni a szalagról a következő lépésben. Azonban ha a Main állapota t_2, akkor az 1–et nem változtatja meg, és a fejet sem mozdítja el, hanem belép a BalraMegy gépbe, amely a fejet az input első szimbólumát megelőző üres cellára mozgatja, és utána megáll. Ennek hatására a főgép t_3 állapotba kerül, és ebben az állpotban eggyel jobbra lépteti a fejet, a legnagyobb helyiértékű bitre. Legvégül pedig a főgép a Komplemens gép újra futtatásával komplementálja a biteket, és ha ezen utóbbi futás HALT állpotban ér véget, akkor maga is terminál és ezzel a folyamat befejeződött.

Ha jobban szemügyre vesszük a gráfot, akkor látható, hogy tulajdonképpen a főprogramban szereplő t_1, t_2 és t_3 állapotok csak azt a célt szolgálják,

hogy biztosítsák az összeköttetést az egyes gépek között, ugyanis az nem megengedett, hogy az egyik gépből közvetlenül „átugorjunk" a másikba, tehát az *Automaton* program nem engedi meg, hogy olyan élet rajzoljunk, amelynek forrás– és célcsomópontja is egy–egy alprogramot végrehajtó gép. Továbbá ami még eltér a korábban leírtaktól az az hogy a program nem egy sima négyzettel jelöli az alprogramok csomópontjait, hanem helyette egy számítógépet ábrázoló ikont használ. (A program használatával kapcsolatos ilyen jellegű kikötéseket a következő alfejezetben tárgyaljuk.)

Mindenesetre az bizonyos, hogy némi gyakorlat megszerzése után bárki könnyen tudja ehhez hasonló gráfokkal a Turing–programok algoritmusait leírni, illetve ilyen gráfokat értelmezni. Az eddigiekben tárgyalt feladatok megoldása megtalálható a CD mellékleten az alábbi fájlnevek alatt:

- Bináris növelést végrehajtó gép: `BinInc.aut`
- Bináris csökkentést végrehajtó gép (első változat): `BinDec.aut`
- Bináris csökkentést végrehajtó gép (utóbbi változat): `BinDec_2.aut`

Érdemes ezeket megnyitni a programmal és tanulmányozni őket, valamint kipróbálni, hogy tényleg jól működnek–e. Azonban hogy ezt megtehessük, legalább alapjaiban ismernünk kell az *Automaton* program használatát. Ezt ismertetjük a továbbiakban.

4.2. Az *Automaton* program

A korábbikaban már említettük azt a tételt, hogy létezik olyan Turing–gép, amely tetszőleges másik gép működését képes szimulálni. Azonban mivel a Turing–gépeket — a Church–tézis alapján — azonosítottuk az algoritmusokkal, nyilvánvaló, hogy nem szükséges ezt a gépet a maga fizikai valóságában megépíteni, hanem elvileg tetszőleges programnyelven lehet implementálni azt az algoritmust, amely bármely Turing–gép működését utánozza. Nos az *Automaton* program pontosan ezzel a céllal készült. ***Az ilyen jellegű programokat már régóta használják a Turing–gépek működésének tanulmányozására és szemléltetésére, azonban az Automaton program érdekessége az, hogy nem közvetlenül a szimulálandó Turing–gép leírását kell neki megadni, hanem azt az iménti alfejezetben ismertetett gráfot, amely a Turing–gép működését leírja. Ezek után a program bejárja ennek a gráfnak a csomópontjait és éleit, és az ott található információk alapján automatikusan előállítja a szimulálandó gép leírását, tehát a Σ és K halmazokat, valamint a δ függvényt, ahogyan az a 3.3.1 definícióban szerepel.***

4.2.1. A program használata

Bár a program használata egyáltalán nem bonyolult, és némi gyakorlattal bárki könnyen kitapasztalhatja annak működését, mivel saját súgóval (még) nem rendelkezik, kívánatos, hogy röviden ismertessük a program használatát. Elöljáróban csak annyit jegyzünk meg, hogy a program teljesen ingyenes, szabadon másolható és terjeszthető, felhasználható bármilyen célra, kivéve a kereskedelmi és üzleti jellegű tevékenységet. A CD mellékleten megtalálható a program forráskódja is, ez is bárki számára hozzáférhető és a szabadon módosítható, egészen addig a pontig, amíg az így elkészült programváltozatokat csak egyéni vagy oktatási célra használják fel.[4]

Az *Automaton* program egy WIN32 platformon futó MDI (*Multiple Document Interface*) alkalmazás, amit magyarul általában többdokumentumos alkalmazásnak neveznek. A program úgynevezett projektekkel dolgozik, egy–egy projekt pedig nem más, mint egy feladat megoldására készült Turing–gépek gráfjainak összessége. Ha létrehozunk, vagy megnyitunk egy projektet, akkor az abban lévő gráfok mindegyike egy különálló gyerekablakban jelenik meg, és lehetőség van arra, hogy ebben az ablakban — amit a továbbiakban nevezzünk gráf szerkesztőnek vagy gráf editornak — a gráf szerkezetét módosítsuk, tehát csomópontokat és éleket adjunk hozzá, vagy vegyünk el belőle. Egy projekt maximálisan 50 darab gráfot tartalmazhat, ha ennél többet akarunk létrehozni, akkor a program ezt nem engedi, és hibaüzenetet ad. A program három darab eszköztárral rendelkezik, amelyeken keresztül a program funkciói elérhetőek. Vegyük sorra ezeket az eszköztárakat! ***A 4.11***

4.11. ábra. A főeszköztár

ábrán a program főeszköztárát láthatjuk. Ez az eszköztár tartalmazza azokat az alapvető funkciókat megvalósító gombokat, amelyekkel a legtöbb WINDOWS alatt futó program rendelkezik. A gombok és a hozzájuk tartozó funkciók az alábbiak

 Az eszköztár legelső gombja arra szolgál, hogy új gráfot adjunk hozzá a projekthez. Ha erre a gombra kattintunk, akkor egy dialógus

[4] Az *Automaton* program MICROSOFT VISUAL C++® 6.0 programnyelven íródott, ami a MICROSOFT VISUAL STUDIO® programcsomag része.

ablak jelenik meg, ahol meg kell adnunk az új gráf nevét. Ennek a névnek egyedinek kell lenni, tehát egy projektben nem szerepelhet két azonos nevű gráf. Ha mégis megpróbálunk olyan nevet megadni, amely már létezik akkor a program tájékoztat arról, hogy már van ilyen nevű gráf, és nem enged még egyet létrehozni. Amikor a programot elindítjuk, akkor nem kell külön projektet létrehozni, hanem elegendő a gráf hozzáadása gombra kattintani, és az első gráf hozzáadásakor automatikusan létrejön a hozzátartozó projekt is.

 Az eszköztár második gombja arra szolgál, hogy korábban elmentett projekteket újra megnyissunk. Ha erre kattintunk, akkor az ismert dialógus ablak jelenik meg, amelyben kiválaszthatjuk a megnyitni kívánt fájlt. A program egyszerre csak egy projekttel tud dolgozni, ezért ha már van egy megnyitva, és másikat szerenénk megnyitni, akkor a program előbb megkérdezi, hogy bezárhatja–e az akutális projektet.

 Az eszköztár harmadik gombja a mentést célozza. Ha még nem volt az állomány a korábbiakban elmentve akkor egy dialógus ablak jelenik meg amely alapértelmezésben az `Untitled.aut` fájlnevet kínálja fel. Ezt célszerű egy beszédesebb névre lecserélni, amely többet árul el a projekt tartalmáról. Ha már korábban mentettük a projektet, akkor a dialógus ablak nem jelenik meg, hanem a mentés automatikusan a korábbiakban megadott néven történik. A projektek alapértelmezett kiterjesztése a `.aut` kiterjesztés. Természetesen ettől el lehet térni, de nem ajánlatos, mert a program ez első futtatás alkalmával automatikusan regisztrálja a maga számára ezt a kiterjesztést a rendszerleíró–adatbázisban, és a továbbiakban ha valamelyik fájlkezelőben (például a *Explorer*–ben) a `.aut` kiterjesztésű állományokra kattintunk, akkor azokat a WINDOWS automatikusan az *Automaton* programmal nyitja meg.

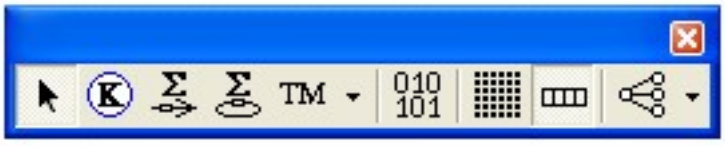
 Az eszköztár 4., 5. és 6. gombja rendre a *Kivágás*, *Másolás* és *Beillesztés* parancsokat implementálja, vagyis a vágólap kezelést valósítják meg. ***Egy gráfban önálló csomópontokat és éleket nem lehet vágólapra másolni, hanem csak részgráfokat.*** Ez azt jelenti, hogy ha kijelöljük valamely csomópontot, akkor automatikusan kijelölésre kerülnek azok a csomópontok is, amely közvetlen szomszédságban vannak az adott csomóponttal, és azok az élek is, amelyeknek az adott csomópont forrás– vagy cél csomópontja. Ehhez hasonlóan ha egy élet jelölünk ki, akkor azok az csomópontok is kijelölésre kerülnek,

amelyekre az adott él illeszkedik. ***A kijelölés oly módon történik, hogy a* Ctrl *gomb nyomvatartása mellett az egérrel arra a csomópontra vagy élre kattintunk, amelyet ki szeretnénk jelölni.*** A kijelölt objektumokat a program az eredeti színekhez képest inverz színekkel jeleníti meg. Ha van kijelölt objektum, akkor automatikusan elérhetővé válnak a *Kivágás* és *Másolás* parancsok. Ha a kijelölést szereténk megszüntetni, akkor egyszerűen kattintsunk a rajzterületen egy olyan üres területre, ahol nem található sem él sem pedig csomópont. ***Itt jegyezzük meg, hogy a program egyszerre csak egy példányban futtatható a* Windows*–ban***, ezért ha már fut a program, és megpróbáljuk mégegyszer elindítani, akkor egy tájékoztatást kapunk arról, hogy a program egy példánya már aktív, és nem fog újra elindulni. Ezt a viselkedést a vágólap kezeléssel kapcsolatos programozás–technikai okok indokolják, ugyanis egy projekten belül sokkal egyszerűbb a vágólapon keresztül részgráfokat másolni és beilleszteni, mint egyik projektből a másikba mozgatni az adatokat, ezért a program úgy van kitalálva, hogy egyszerre csak egy példányban futhasson és az is egyszerre csak egy projektet nyithasson meg.

 Az eszköztár hetedik gombja a nyomtatást indítja el, de a program ezt a funkciót (jelenleg) nem használja, így mindig le van tiltva, és hasonlóan a többi nyomtatással kapcsolatos funkció is a menükben (jelenleg) nem elérhető. (Ezek a opciók majd a program egy későbbi verziójában lesznek implementálva.)

 A főeszköztár nyolcadik gombja a program információkat jeleníti meg egy dialógus ablakban, az utolsó gomb pedig a súgót aktiválja, azonban — mint már említettük —, ez utóbbi is a program egy későbbi verziójában lesz csak elérhető.

A program második eszköztára alapértelmezésben a főablak baloldalán van dokkolva. Ez az eszköztár a ***gráf szerkesztő eszköztára***, vagyis ezen ke-



4.12. ábra. A gráfeszköztár

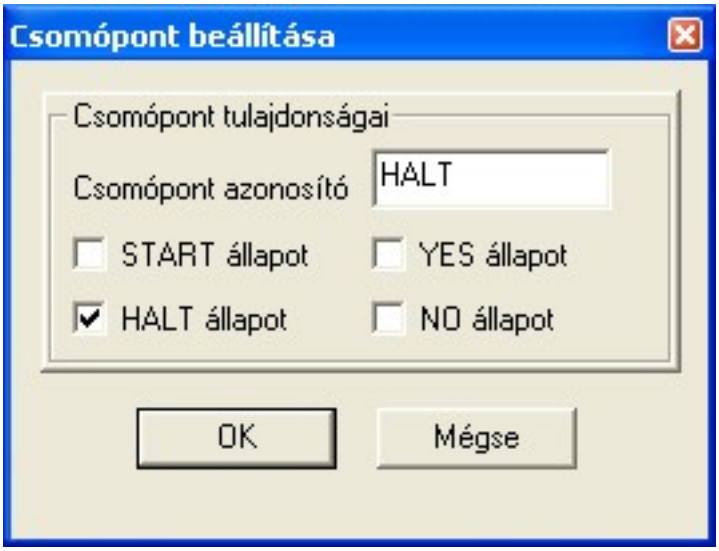
resztül érhetjük el azokat a funkciókat, amelyek egy gráf megrajzolásához

szükségesek. Az eszköztárat a 4.12 ábra mutatja, nézzük meg sorjában az ezen található gombokat is.

 Az első gomb, amely egy egér mutatót ábrázol, arra szolgál, hogy kikapcsoljon minden aktuális műveletet, és az egér „működését" alapértelmezettre visszaállítsa. Ez úgy értendő, hogy ha valamilyen műveletet akarunk végezni — például egy élet vagy csomópontot megrajzolni stb. —, akkor be kell kapcsolni az adott művelet gombját az eszkörtáron. Egészen addig, amíg nem kattintunk az eszköztár első gombjára (vagy egy másik művelet gombjára), az utoljára bekapcsolt művelet lesz az aktív, tehát például ha a csomópont hozzáadást aktiváltuk utoljára, akkor minden egyes kattintáskor a program feltesz egy új csomópontot a gráfra, egészen addig, amíg ezt a műveletet ki nem kapcsoljuk, illetve egy másik műveletre át nem váltunk. Azt, hogy mi az aktuálisan bekapcsolt művelet, onnan tudhatjuk, hogy egyrészt az eszköztáron az adott művelet gombja lenyomott állapotban van, másrészt a gráf editor ablaka felett mozgatva az egeret annak mutatója olyan formát vesz fel, amely tartalmazza az aktuális művelet eszköztáron is látható ikonját. ***Ha minden művelet ki van kapcsolva, tehát az első gomb van lenyomott állapotban, akkor lehetőség van arra, hogy a gráfra eddig felrakott éleket és csomópontokat az egérrel megragadjuk és a „fogd–és–vidd" módszerrel máshova mozgassuk a rajzterületen. Ennek akkor lehet szerepe, ha már nagyon sok komponensből áll egy gráf, és emiatt egyik a másikat eltakarja. Ebben az esetben az egérrel „széthúzhatjuk" és „kibogozhatjuk" a gráfot úgy, hogy az áttekinthető legyen. Azt, hogy mely komponensekre alkalmazható a vonszolás, az mutatja, hogy az adott komponens felett mozgatva az egeret, a mutató egy kéz formát vesz fel. Arra is lehetőség van, hogy egy teljes részgráfot mozgassunk más helyre, ehhez az szükséges, hogy a vágólapkezelésnél ismertetett módszerrel kijelöljük egy részgráfot, és ezek után a kijelölés bármely csomópontját vagy élét megragadva az egész részgráf vonszolhatóvá válik.***

 Az eszköztár második gombjával tudjuk a „Csomópont hozzáadása" műveletet aktiválni. Ha ezt a gombot lenyomjuk, akkor lehetővé válik az, hogy új csomópontokat adjunk hozzá a gráfhoz. A művelet aktiválása után egyszerűen kattintsunk arra a pontra, ahova az új csomópontot el szeretnénk helyezni, és a program automatikusan lerakja azt a kattintás helyére. Egészen addig ezt a viselkedést követi, amíg

ki nem kapcsoljuk az aktuális műveletet, vagy át nem váltunk egy másik műveletre. A korábban elmondottak értelmében teljesen mindegy, hogy hová helyezzük az új csomópontot, hiszen később is bárhova mozgathatjuk. Az újonnan felhelyezett csomópont alapértelmezésben a „?” névvel rendelkezik, ez jelöli azt, hogy új csomópontról van szó, és még nem adtunk meg annak a belső állapotnak a nevét, amelyre hivatkozik. A csomópont nevét kéféle módon tudjuk megváltoztatni. Az egyik az, hogy duplán kattintunk az egérrel a csomóponton, a másik pedig az, hogy az egér jobb gombjával kattintunk rajta és a felbukkanó menüből a „<...> csomópont tulajdonságai” menüpontot választjuk. Minkét



4.13. ábra. A csomópont tulajdonságok dialógusablaka

esetben az lesz a végeredmény, hogy a 4.13 ábrán látható dialógusablak jelenik meg, ahol a csomópont tulajdonságait meg tudjuk változtatni. A „Csomópont azonosító” nevű bevitelimezőbe kell begépelni annak a belső állapotnak a nevét, amelyet a csomópontnak megfeleltetünk. ***A csomópont neve legfeljebb öt karakter hosszúságú lehet, és tetszőleges* ASCII *karaktereket tartalmazhat, valamint a névnek egyedinek kell lennie, vagyis egy belső állapothoz csak egy csomópont rendelhető hozzá***, de ha mégis több csomópontnak adjuk ugyanazt a nevet, akkor ezt a program megengedi, csak akkor fogja a hibát jelezni, amikor a gráf alapján szeretnénk vele előállíttatni a gép programját, a δ függvényt. ***A dialógusablakon szereplő négy darab jelölő négyzettel jelezhetjük a program számára, hogy***

az adott csomópont valamelyik kitüntetett állapotot, tehát a* START, HALT, YES *vagy a* NO *állapotot kódolja. Természetesen egy állaptotnak nem lehet egyszerre több speciális jelentése, hanem az imént felsoroltak egyértelműen meghatározottak, és egy gráfon belül minden speciális jelentéssel bíró állapotból egyszerre csak fordulhat elő. Éppen ezért ***ha már létezik egy olyan csomópont a gráfban, amely mondjuk meg van jelölve* START *állapotként, és létrehozunk egy másik csomópontot és azt is* START *állapotnak deklaráljuk, akkor a program automatikusan törli a korábbi állapot* START *tulajdonságát (vagyis az közönséges belső állapottá válik), és minden esetben az az állapot lesz a* START *tulajdonságú, amelynél utoljára jelöltük be az ennek megfelelő jelölőnégyzetet. Természetesen ez utóbbi megállapítás vonatkozik az egyes termináló állapotokra is.***

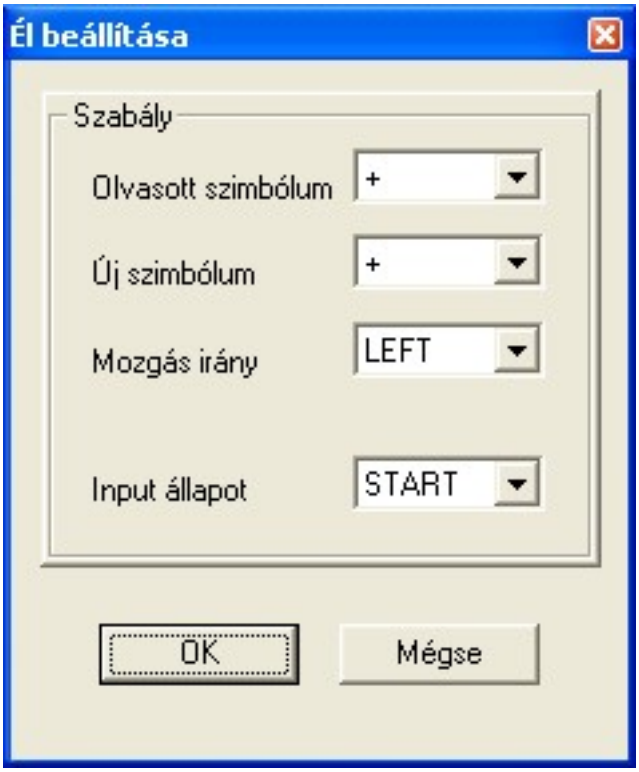
 Az eszköztár harmadik és negyedik gombja segítségével kapcsolhatjuk be azt a funkciót, amely az élek rajzolását teszi lehetővé. A harmadik gomb lenyomásával olyan éleket tudunk rajzolni amelyek forrás– és cél csomópontja különböző, a negyedik gomb pedig a hurokélek rajzolását aktiválja. Az előző alfejezetben ismertettük a gráfok szerkezeti elemeit[5], és az ott elmondottak alapján az éleket az alábbi három típusba sorolhatjuk:

1. *Normál*: forrás– és célcsomópontja is egy belső állapotnak felel meg. (Lehet hurokél is.)
2. *Input*: forrás csomópontja egy belső állapot, cél csomópontja pedig egy olyan csomópont, amely egy másik Turing–gépre hivatkozik.
3. *Output*: forrás csomópontja egy másik Turing–gépre hivatkozik, cél csomópontja pedig egy belső állapot.

Az Automaton program csak a felsorolt típusú élek rajzolását engedi meg, tehát még véletlenül sem fordulhat elő, hogy olyan élet adjunk meg a gráfban, amelynek mindkét csomópontja egy másik Truing–gépre, mint alprogramra hivatkozik. ***Továbbá minden olyan csomópontnak, amely egy másik Turing–gépre hivatkozik, legfeljebb egy bemenő (input) éle, és legfeljebb három kimenő (output) éle lehet, és a kimenő élek között nem lehet***

[5]Lásd a 4.2 és a 4.6 ábrákat az 63. oldalon és a 70. oldalon.

két azonos. Amikor egy élet megrajzolunk, a program automatikusan hozzárendel egy alapértelmezett szabályt az adott élhez. Normál élek esetében ez a szabály a $\# \rightarrow \#, \mathrm{STAY}$, input élek esetében $\# \rightarrow \#, \mathrm{STAY}, \mathrm{START}$, output élek esetében pedig HALT. A szabályt a korábbiakhoz hasonlóan úgy tudjuk módosítani, ha duplán kattintunk az élen, vagy a jobb egérgommbal kattintva a felbukkanó menüből a „Tulajdonságok" menüpontot választjuk. Ekkor a 4.14 ábrán látható dialógus ablak jelenik meg. Ezen keresztül tudjuk az élhez



4.14. ábra. Az él tulajdonságok dialógusablaka

rendelt szabály egyes komponenseit megadni, tehát az olvasott szimbólumot, a visszaírandó szimbólumot és a fej mozgás irányát, valamint input élek esetében az input állapotot, ami azt írja elő, hogy az adott szabály érvényesülése esetén milyen belső állpotban kell elindítani az él cél csomópontjában lévő Turing–gépet. Output élek esetén csak azt kell megadnunk, hogy az adott él melyik termináló állapotnak felel meg. ***Ezen a ponton még fontos megjegyezni, hogy az Automaton programban a modellezett Turing–gépek külső ábécéje nem***

lehet bármi, hanem csak az* ASCII *kódtábla 33–tól 126–ig terjedő kóddal rendelkező karakterei lehetnek benne.[6]


A gráf eszköztár ötödik gombja segítségével tudunk egy gráfhoz olyan csomópontot hozzáadni, amely egy másik gépre hivatkozik. Ha közvetlenül a gombra kattintunk, akkor egy dialógus ablak jelenik meg, amely felsorolja a projektben eddig létrehozott összes gépet. Ha valamelyiket kiválasztjuk, és bezárjuk az ablakot az „OK" gombbal, akkor a gráfhoz hozzá tudjuk adni a kiválasztott gépre hivatkozó csomópontot. Ugyanezt a hatást érhetjük el azzal is, ha a gomb mellet lévő kis nyílra kattintunk, és a felbukkanó menüből kiválasztjuk a megfelelő gép nevét. A csomópont hozzáadásához ebben az esetben is csak annyit kell tenni, hogy arra pontra kattintunk, ahova a csomópontot akarjuk helyezni és a program leteszi azt a kijelölt helyre. Egy lépésben csak egy ilyen csomópontot lehet a gráfba illeszteni, ha többet is akarunk, akkor a fenti lépéssort minden egyes esetben meg kell ismételni. Ha egy másik gépre hivatkozó csomóponton duplán kattintunk, akkor a program automatikusan annak a gráfnak a szerkesztőablakát teszi aktívvá, amelyre a csomópont hivatkozik. Ugyanez a hatása annak, ha a jobbgombbal kattintunk és a felbukkanó menüből a „<...> gráf aktiválása" parancsot választjuk.


A hatodik gomb valósítja meg az eszköztár legfontosabb funkcióját, a ***program generálását***. Ha erre kattintunk, akkor a program bejárja a megadott gráfokat, és azok alapján megpróbálja előállítani az általuk leírt Turing–gépek programját. Ha ez kész, akkor egy felbukkanó üzenet ablak tájékoztat a művelet eredményéről. Ez utóbbi abból áll, hogy a program felsorolja azokat az „észrevételeket", amelyekre a gráfok elemzése során jutott. Ezek az „észrevételek" kétfélék lehetnek: hibák és figyelmeztetések. A hiba egy olyan tényre hívja fel a figyelmet, amely miatt a gráf által megadott Turing–gép működésképtelen, ezzel szemben a figyelmeztetés csak azt jelenti, hogy a gráf által leírt gép működőképes, de potenciális hibaforrást tartalmaz, vagyis olyat, amely nem fog minden esetben előjönni, hanem csak néha, attól függően, hogy milyen inputtal indítjuk a gépet. Az „*A program hibaüzenetei*" című alfejezetben felsoroljuk a leggyakoribb hibaüzeneteket és figyelmeztetéseket és azok lehetséges okát.

[6] Ez nem jelent semmiféle megkötést és nem szűkíti le a programmal modellezhető gépek osztályát, ugyanis például olyan univerzális Turing–gépet is meg lehet adni, ahol $|\Sigma| = 2$, tehát a külső ábécéje csak két elemű. Vö. [RISz] 206. o. Ha egy ilyen univerzális gépet szerkesztünk meg a programmal, bármely másik működését szimulálni tudjuk.

 A gráfeszköztár hetedik gombja a szerkesztőablakok rácspontjainak ki– és bekapcsolására szolgál. Ha ez a rácspontok bekapcsolt állapotban vannak, akkor a program automatikusan ezekhez a rácspontokhoz igazítja a gráfra felhelyezett komponenseket. Ez segít abban, hogy például két csomópontot pontosen egymás mellé vagy fölé helyezzünk.

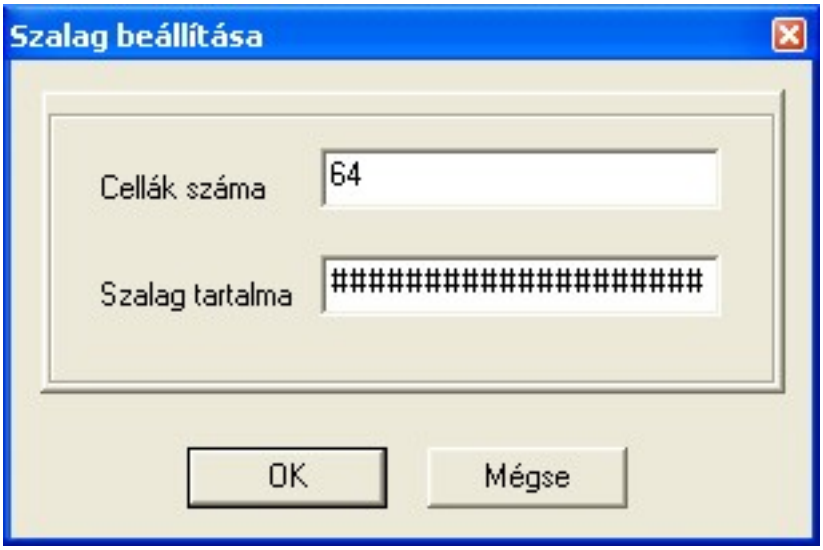
 A nyolcadik gomb az ablak alján lévő szalag ablakának megjelenítésére illetve elrejtésére való. Ha a szalagot elrejtjük, akkor nagyobb lesz a rajzterület, ahol rajzolhatunk, és nem kell görgetni az ablakot, hogy lássuk a gráf minden részletét.

 Az eszköztár legutolsó gombja az egyes gráfok szerkesztő ablakainak megjelenítésére szolgál. Fontos felhívni a figyelmet arra, hogy ***ha egy gráf szerkesztő ablakát bezárjuk, akkor a program eltünteti azt, de a gráf nem kerül törlésre a projektből. Ha szeretnénk újra megjeleníteni egy gráf bezárt szerkesztő ablakát, akkor ennek a gombnak a segítségével aktiválhatjuk azt, illetve tetszőleges másik szerkesztő ablakot.*** A gomb működése teljesen azonos, mint az eszköztár ötödik gombjának esetében. (Lásd feljebb.) ***Ha egy teljes gráfot szeretnénk eltávolítani a projektből, akkor kattintsunk az eltávolítandó gráf szerkesztő ablakának rajzolási területére, és válasszuk a „Gráf törlése" parancsot. Figyelem: a program nem rendelkezik visszavonási funkcióval, ezért ha bármit törlünk, akkor az végleg elveszik, ha vissza szeretnénk állítani, akkor az egészet újra kell rajzolni.*** Ha egy teljes gráfot törlünk a projektből, akkor minden egyéb gráfból is törlődnek a rá hivatkozó csomópontok és élek. ***Ha nem teljes gráfot, hanem csak részgráfot akarunk törölni, jelöljük ki azokat a komponenseket, amit el akarunk távolítani, és nyomjunk meg a Delete billentyűt.***

A program harmadik és egyben legutolsó eszköztára a szalag ablakához tartozik, és ezen keresztül érhetőek el azok a funkciók, amelyek a modellezett Turing–gépek különféle inputokkal való futtatásához és megállításához szükségesek. Ezt az eszköztárat 4.15 ábra mutatja. A rajta található gombokkal az alábbi műveleteket hajthatjuk végre.

 Ha az első gombra kattintunk, akkor egy dialógus ablak jelenik meg, ahol a szalag celláinak számát és a szalag tartalmát adhatjuk meg (4.16 ábra). ***A szalag celláinak száma 1 és 8192 között lehet***, és a második bevitelimezőbe, ahol a szalagtartalmát, vagyis az inputot

4.15. ábra. A szalageszköztár



4.16. ábra. A szalag tartalmának beállításra szolgáló dialógusablak

deklaráljuk, értelem szerűen maximálisan annyi karaktert lehet beírni, ahány az első bevitelimezőben megadott cellák száma. Ha az input hossza kisebb, mint az összes cella száma, akkor a program az üres cellákat automatikusan a „#" jellel tölti fel. ***Ha a futás során a fej bármely irányban túllépi a megadott cellaszámot, akkor a program automatikusan kiterjeszti a szalag hosszát a szükséges irányban, tehát nem fordulhat elő, hogy a fej „leesik" valamely oldalon. Ha csak egyetlen cella tartalmát akarjuk megváltoztatni, akkor kattintsunk az adott cella felett az egér jobb gombjával, és válasszuk a „Cella tartalma" menüpontot.*** Ekkor a felbukkanó ablakban csak annak az egyetlen cellának a szimbólumát tudjuk magváltoztatni, amely felett a menüt előhívtuk. ***Ha ugyanebben a menüben a „Fejpozíció beállítása erre a cellára" parancsot választjuk, akkor a fej rááll az adott cellára, és onnan indul (vagy folytatódik) a futás.***

 Az eszköztár második gombjával lépésenként tudjuk végrehajtani az aktuális Turing–gép programját. Ha erre kattintunk, akkor a gép végrehajtja a következő lépést, amelyet az olvasott szimbólum, és aktuális belső állapota meghatároz, és után megáll. ***Minden esetben annak a Turing–gépnek a programja tekintendő aktívnak, amelynek a szerkesztő ablaka is aktív, tehát amely a többi ablak felett az előtérben látszik.*** Ez arra jó, hogy a projektben szereplő minden egyes gépet önállóan tudjuk futtatni és tesztelni. Bár nem kötelező, de minden projektben célszerű egy gépet deklarálni MAIN (vagy valami hasonló) néven, amely a főprogram szerepét ellátja, és a többi részfeladatot megoldó alprogramot (gépet) egységbe fogja. ***Ha esetleg egy Turing–gép olyan ponthoz érkezik a futása során, amelynél nincs definiálva, hogy milyen cselekvéssort hajtson végre, akkor ezt egy üzenet ablakban közli, és a futás félbeszakad.*** A nagyobb problémát a lehetséges végtelen ciklusok okozzák, mert ezeket lehetetlen felderíteni, de ha a fej sokáig nem mozdul, akkor gyanakodhatunk, és célszerű a futást megszakítani az eszköztár ötödik gombjával. (Lásd alább.)

 A harmadik gomb a ***folyamatos futás*** elindítására való, ha rá kattintunk, akkor a gép elindul, és folyamatosan, a felhasználó beavatkozása nélkül hajtja végre az egymás utáni lépéseket. Ez a gomb szoros kapcsolatban áll a hatodik  gombbal, amellyel a futás sebességét állíthatjuk be. Ha ez utóbbit választjuk, akkor egy dialógus ablakban

megadhatjuk, hogy a gép mennyi ideig várakozzon az egyes lépések végrehajtása között folyamatos futás esetén. Az időintervallumot miliszekundummban kell megadni, és az alapértelmezett értéke 1500, ami tehát azt jelenti, hogy a gép minden egyes lépés végrehajtása után másfél másodpercig várakozik, mielőtt a következő ciklust megkezdené. (Ez esetenként lassú lehet, ezért érdemes áttállítani.)

 Az eszköztár negyedik gombja az iménti módon elindított ***futás időleges felfüggesztésére*** szolgál. Lehetőség van arra, hogy tetszőleges ponton felfüggesszük az aktuális gép futását, és utána lépésenként haladjunk a programban, majd esetleg ha szükséges, akkor újra a folyamatos futtatásra térjünk vissza. Ezzel szemben a következő  (ötödik) gomb nem csak felfüggeszti, hanem teljesen l***eállítja az aktuális gép futását***. Fontos felhívni a figyelmet arra a tényre, hogy ***amíg egy gép futása folyamatban van, addig a projektet bezárni/menteni, másik projektet megnyitni, a gráfokat szerkeszteni és az egész programot bezárni nem lehet. Ha futás közben ezen műveleteket valamelyikét kezdeményezzük, a program tájékoztat arról, hogy a futás folyamatban van, és előbb állítsuk azt meg. Attól, hogy a futást időlegesen felfüggesztettük, azt még a program úgy tekinti, hogy a futás folyamatban van, ezért a megállításhoz minden esetben az eszköztár ötödik gombját kell igénybe vennünk.***

4.2.2. A program hibaüzenetei

Az alábbiakban áttekintjük azokat a leggyakoribb hibaüzeneteket és figyelmeztetéseket, amelyeket a program generál, miközben elemzi a megadott grágokat.

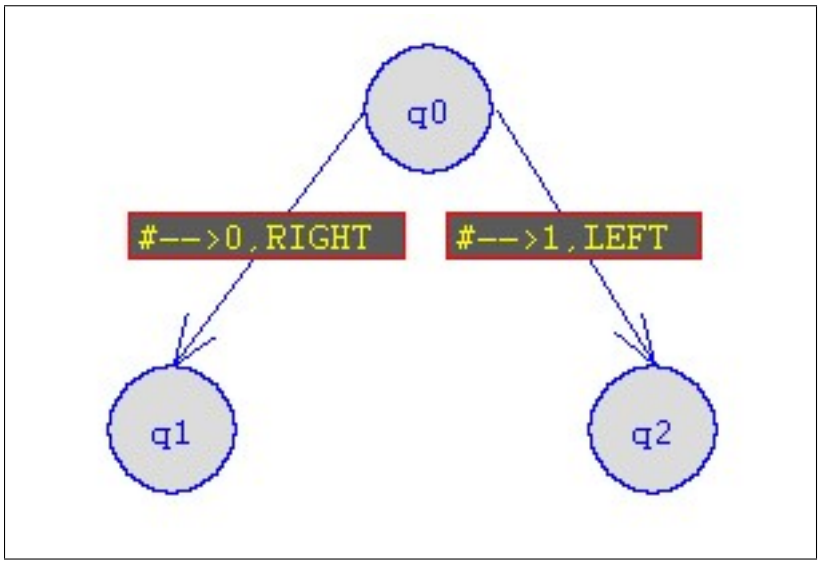
1. Hiba: A(z) <...> belső állapot csomópontja többször szerepel a(z) <...> gráfban.

   Magyarázat Minden csomópont, amely állapotot kódol, csak egyszer fordulhat elő egy gráfon belül, ezért töröljük azokat, amelyekből több is van.

2. Hiba: A(z) <...> gráf <...> csomópontjában a program futása nem egyértelmű.

   Magyarázat A megnevezett csomópont több olyan élnek is forrás csomópontja, amely élekhez hozzárendelt szabály ugyanazt az olvasott

szimbólumot tartalmazza. Példaként tekintsük a 4.17 ábrán lévő részgráfot. Abból egyrészt az következik — a bal oldali élen végig haladva —,



4.17. ábra. Csomópont, ahol a gép futása nem egyértelmű

hogy $\delta(q_0, \#) = (q_1, 0, +1)$, másrészt — ha a jobb oldali élen megyünk végig —, akkor $\delta(q_0, \#) = (q_2, 1, -1)$. Vagyis ha a gép q_0 állpotba kerül és „#" jelet olvas, akkor nem tudja egyértelműen eldönteni, hogy milyen cselekvéssort hajtson végre.

3. Hiba: A(z) <...> gráfban nincs megjelölve kezdő állapot.

   Magyarázat Minden gráfban kell lennie egy olyan csomópontnak, amelyet állapotot START állapotként deklarálunk.

4. Hiba: A(z) <...> gráfban nincs megjelölve vég állapot.

   Magyarázat Minden gráfban kell lennie legalább egy olyan csomópontnak, amely termináló állpotra hivatkozik.

5. Hiba: A(z) <...> gráfban nem létezik a <...> csomópont, amelyet a <...> gráf megkövetel.

   Magyarázat Ezt a hibaüzenetet input él esetében kapjuk, akkor ha mondjuk az élhez hozzárendelt szabály azt írja elő, hogy az él cél csomópontjában lévő gépet valamilyen q állapotban kell elindítani, de az adott gépnek nincs q állapota (például azért, mert korábban volt, de átneveztük, vagy töröltük).

6. Figyelmeztetés: A(z) <...> állapot csomópontjának nincs bemenő éle a(z) <...> gráfban.

   Magyarázat Mivel nincs bemenő él, ez azt jelenti, a gép soha nem fogja ezt az állapotot felvenni, ezért teljesen felesleges.

7. Figyelmeztetés: A(z) <...> állapot csomópontjának nincs kimenő éle a(z) <...> gráfban.

   Magyarázat Mivel nincs kimenő él, ez azt jelenti, hogy a δ függvény ebben a pontban nem lesz definiálva, és ha véletlenül a gép ebbe az állapotba kerül, akkor azt soha nem hagyja el, mert nem tudja majd folytatni a futását. Természetesen ez nem minden esetben fordul majd elő, hanem csak speciális inputok esetén, amelyek olyan utasítás végrehajtását indukálják, amely hatására a gép felveszi ezt az állapotot.

Ezzel áttekintettük azokat a legtipikusabb (valódi és potenciális) hibákat, amelyek egy gráf szerkesztése során felmerülhetnek. ***Minden esetben a hibák és figyelmeztetések számát is közli a program, de a projekt által tartalmazott gépek csak abban az esetben indíthatóak el, ha a hibák száma nulla. Fontos szem előtt tartani továbbá azt, hogy ha bármilyen módosítást hajtunk végre egy gráfon, akkor minden esetben a program generáló gombra kell kattintani, hogy a módosítások érvénybe lépjenek.*** Ha ezt elmulasztjuk, akkor az *Automaton* program továbbra is a régi beállításokkal fog dolgozni, függetlenül attól, hogy milyen módosítást hajtottunk végre a gráfokon legutóbb. Ha a gráfokban nincs hiba, és a program előállítása sikeres volt, akkor a programutasításokat megtekinthetjük, ha a gráf szerkesztő ablakok alján az egér segítségével felfedjük az alapértelmezésben nem látható listát. Ezt úgy is megtehetjük, hogy a program főmenüjéből az „Ablakok" menüpont alatt található „Ablak felosztása" parancsot választjuk.

Az eddig elmondottakkal gyakorlatilag a program működtetéséhez szükséges minden információt megadtunk, úgy is mondhatjuk, hogy ez az alfejezet pótólja a program hiányzó súgóját, vagyis a felhasználói dokumentációt. Még kiegészítésként annyit említünk meg, hogy az itt tárgyalt eszköztárak funkciói a főmenükben vagy az egyes felbukkanó menükben is elérhetőek, illetve a legfontosabb műveleteket billentyű kombinációk lenyomásával is aktiválhatjuk. Itt nem közöljük részletesen, hogy melyek ezek a billentyű kombinációk, de ha az eszköztárak gombjai fölé visszük az egérkurzort, akkor a felbukkanó eszköztippek tájékoztatnak a gomb funkciójáról és a hozzárendelt billentyűkről.

4.3. Iterációk és ciklusok Turing–gépen

A korábbiakban azt a célt deklaráltuk, hogy megpróbáljuk valamilyen formában a magasszintű programozás alapelemeit „lehozni" a Turing–gépek szintjére. A különböző szakirodalmak más és más felsorolását adják ezeknek az alapelemeknek, példaként említhetjük az irodalomjegyzékben szereplő [TL] jegyzetet, amely rövid, de lényegretörő formában tárgyalja azokat. Mi a továbbiakban mindössze három alapelemet különböztetünk meg, és vizsgálunk kicsit részletesebben, méghozzá a következő programozáselméleti tételre támaszkodva.

4.3.1. Tétel (Böhm–Jacopini tétel). *Az alábbi három vezérlőszerkezeti elem segítségével tetszőleges algoritmus felépíthető:*

1. *Szekvencia: egymásután végrehajtandó tevékenységek sorozata.*
2. *Szelekció: választás megadott tevékenységek között.*
3. *Iteráció: megadott tevékenységek ismételt végrehajtása.*

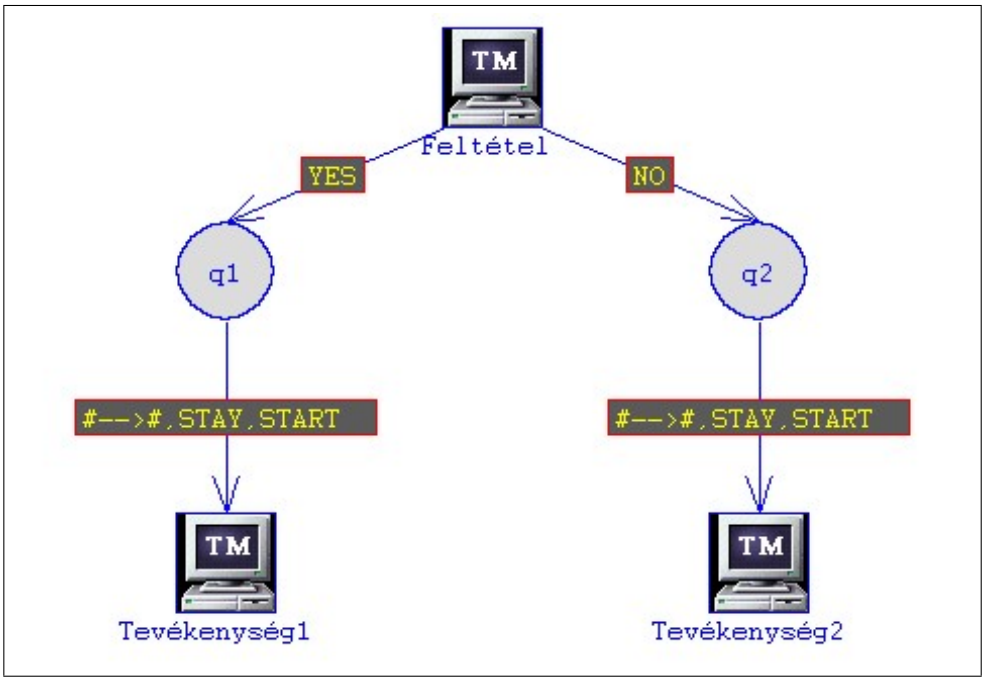
A 4.3.1 tételt eredetileg azzal a céllal bizonyították be, hogy igazolják azt a feltevést, hogy a programokban nincsen szükség a feltétel nélküli vezérlésátadás használatára, amit a magasszintű nyelvekben a „goto" utasítás valósít meg. A programozási nyelvek fejlődésének kezdeti szakaszában gyakran használták a „goto" utasítást, és emiatt az elkészült programok teljesen áttekinthetetlenek lettek, és természetesen az esetleges hibák felderítése is komoly erőfeszítést igényelt, gyakran több munkával járt, mint magának az egész programnak az elkészítése.

Mi most a tételnek nem ezt az aspektusát használjuk fel, hiszen vizsgálódásunk szempontjából jelenleg nem az a fontos, hogy hatékony programokat valósítsunk meg a Turing–gépeken, hanem azt kutatjuk, hogy egyáltalán lehetséges–e a magasszintű programokat átvinni Turing–gépre.

A fenti felsorolásban a szekvenciákkal van a legkönnyebb dolgunk, hiszen már láttuk azt, hogy ezek alkalmazhatóak a Turing–gépekre, méghozzá két különböző értelemben is. Egyrészt a Turing–programok utasításait tekinthetjük egy–egy szekvenciának, vagyis a $(q, \sigma) \to (q', \sigma', m)$ alakú utasítások egymásutánjából építhetünk fel programokat egy–egy feladat megoldására. Másrészt viszont láttuk azt is, hogy miként tudjuk ez egyik gép kimenetét összekötni a másik gép bemenetével, és ezáltal úgy is tekinthetjük a dolgot, hogy az egyes részfeladatok megoldására alkotott gépek képeznek egy–egy tevékenység sort, vagyis szekvenciát és ezeket megfelelő sorrendben elindítva eljuthatunk a feladat megoldásához. Ehhez mindössze egy olyan gépre van

szükség, amely a főprogramot helyettesíti, és egységbe fogja a többi részfeladatot végrehajtó gépet.

A szelekciókkal és iterációkkal már nem ennyire egyszerű a helyzet. A szelekció nem más, mint közismertebb nevén a feltételes elágazás, vagyis a HA <*feltétel*> AKKOR <*tevékenység1*> KÜLÖNBEN <*tevékenység2*> típusú utasítás sor. Ez azt fogalmazza meg, hogy ha teljesül a kiszabott feltétel, vagyis annak logikai kiértékelése igaz értékkel zárul, akkor az első tevékenység sor hajtandó végre. Ha viszont a kiértékelés után a feltétel hamisnak bizonyul, akkor a második tevékenység sorral kell folytatni a futást. Mivel a feltétel minden esetben valamilyen logikai feltétel, annak kirértékelése két lehetséges eredménnyel zárulhat: IGAZ vagy HAMIS. Ezen a ponton mutatkozik meg, hogy az általunk a 3.3.1 definícióban megadott Turing–gép leírás miért tartalmazza a termináló állapotai közöt az elfogadó YES, és az elutasító NO állapotot. Ezek ugyanis megkönnyítitk azt, hogy a feltételes elágazást magvalósítsuk a Turing–gépen. Méghozzá azon a módon, amelynek sémáját a 4.18 ábrán látható gráf mutatja. Az alapelv az, hogy szerkesztünk

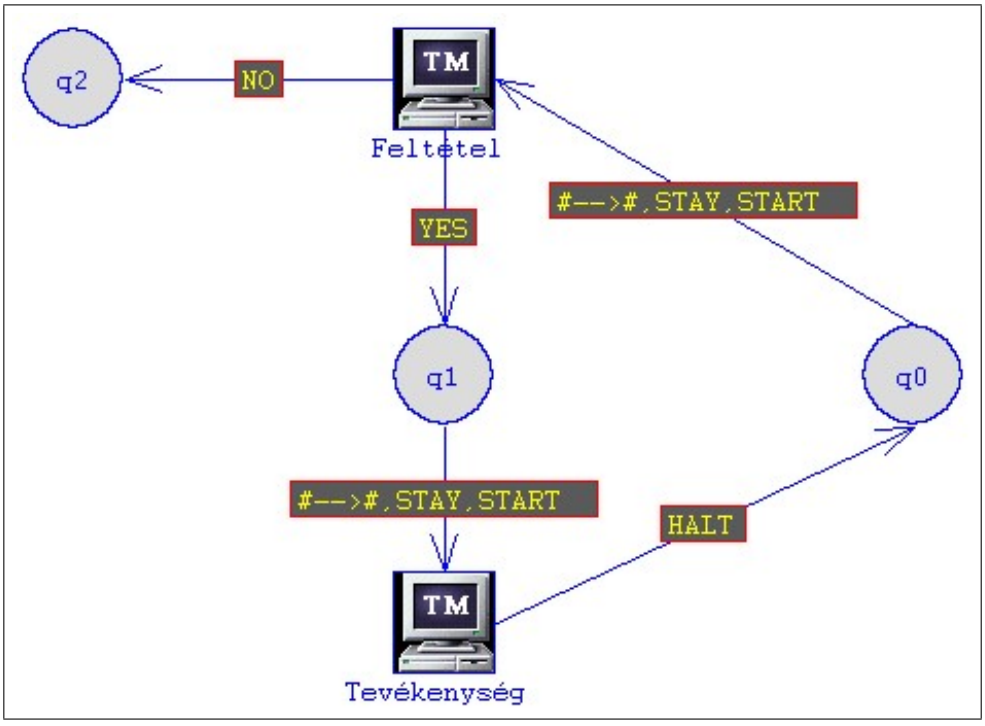


4.18. ábra. A szelekció Turing–gráfja

egy olyan gépet, amely semmi egyebet nem tesz, mint megvizsgálja a feltétel teljesülését. Természetesen a feltétel nagyon sok dolog lehet, egészen ez olyan egyszerű dolgoktól kezdve, mint két szám egyenlőségének vizsgálata, az olyan

összetett feltételekig, mint például annak eldöntése, hogy egy szám prím vagy sem. De ha elvileg tudunk olyan gépet szerkeszteni a feltételben megfogalamzott eldöntendő kérdésre, amely elfogadja a bemenetet, ha a feltétel igaz, és elutasítja azt ha a feltétel hamis, akkor ezek után nem kell mást tenni, mint a végrehajtandó tevékenységekre is külön gépet szerkeszteni, és ha a feltételt eldöntő gép YES állapotban terminált, akkor a TEVÉKENYSÉG1 nevű gépbe átlépni, ha pedig NO állapotban terminált, akkor TEVÉKENYSÉG2 nevű gépbe „ugrani”.

Természetesen összetett szelekciót is meg tudunk valósítani, tehát olyat, mint például HA <*feltétel1*> AKKOR <*tevékenység1*> KÜLÖNBEN HA <*feltétel2*> AKKOR <*tevékenység2*> KÜLÖNBEN <*tevékenység3*>. Nem kell mást tenni, mint a 4.18 ábrán a TEVÉKENYSÉG2 nevű gépen belül megvalósítani a FELTÉTEL2 eldöntését, a folytatás pedig az iméntiek alapján már nyilvánvaló. Ezzel a megoldással tetszőleges sok feltételt is egymásba ágyazhatunk. Látjuk tehát, hogy a feltételes elágazások átvitele Turing–gépre nem



4.19. ábra. Az iteráció Turing–gráfja

különösebben nehéz feladat. Továbbá ezzel már az iterációkkal kapcsolatban is megtettünk egy nagy lépést, ugyanis azok visszavezethetőek a feltételes elágazásokra. Az iterációk általános formája AMÍG <*feltétel*> ISMÉTELD

<*tevékenység*>. A feltételes elágazástól csupán annyiban különbözik, hogy most nem választani kell két tevékenység között, hanem ugyanazt a tevékenységet ismételni, amíg a feltételben megfogalmazott állítás logikai kiértékelése igaznak bizonyul. Természetesen az iterációk több féle ciklussal is megvalósíthatóak, például FOR ciklussal, elöltesztelő– illetve hátultesztelő ciklussal, de ezek közül bármelyik képes szimulálni a másikat[7], ezért az ilyen forma megkülönböztetések csak elvi jellegűek, és elegendő az iterációkat és ciklusokat általánosságban vizsgálni.

A 4.19 ábra mutatja, hogy milyen formában alkalmazhatóak az iterációk a Turing–gépre. Itt is külön gépet készítünk a feltétel eldöntésére és a tevékenységsor végrehajtására. Amíg a feltételt eldöntő gép YES állapotban áll meg, minden esetben átlépünk a tevékenységet végrehajtó gépbe. Ha ez utóbbi befejezte a működését HALT állapotban, akkor — és itt jön a lényeg — egy közvetítő állapoton keresztül visszamegyünk a feltétel vizsgálatát elvégző gépbe. Ezen a ponton pedig zárul a kör, és indul a következő iteráció. Az, hogy zárul a kör, ebben az esetben szó szerint értendő, ugyanis látható, hogy az iterációkat leíró részgráf egy irányított körként jelenik meg a teljes gráfban. Ezzel szemben a feltételes elágazásnál — ha visszatekintünk —, azonnal szembetűnik, hogy az nem tartalmaz kört, ha nem gráfja fa szerkezetű.

Az eddigi megfontolásaink tehát azt valószínűsítik, hogy ha adva van egy magasszintű nyelven megfogalmazott algoritmus, akkor — úgy tűnik — nincs elvi akadálya annak, hogy azt átfogalmazzuk Turing–gépre. Sőt az sem lehetetlen — bár nem triviális feladat —, hogy írjunk egy olyan fordítóprogramot, amely egy magasszintű nyelven, például PASCAL–ban megírt programot lefordítja a Turing–gép kódjára.

Természetesen az eddig elmondottak pusztán elméleti jellegű fejtegetések voltak, és ha arra kérnek bennünket, hogy egy konkrét feladat esetében végezzük is el ezt az átalakítást, egyáltalán nem biztos, hogy ez triviális módon végrehajtható. Azért, hogy ne csupán elméleti szinten tárgyaljuk a kérdést, az alábbiakban mutatunk egy egyszerű példát arra, hogy ez miként valósítható meg. A korábbiakban már elkészítettük azokat a Turing–gépeket, amelyek egy bináris számot eggyel megnöveltek illetve csökkentettek. Most ezek felhasználásával készítsünk egy olyat, amely összead két kettes számrendszerben megadott számot. Természetesen most is több lehetőség kínálkozik a megvalósításra, mint eddig minden esetben. Mi a következő algoritmus szerint járunk el. Kezdetben felírjuk a szalagra a két összadandó számot valamilyen szeparátor jellel elválasztva. Az, hogy mit használunk szeparátor jelnek, teljesen mindegy, de mivel összeadásról van szó legyen ez a jel a „+”

[7]Vö. a 3.3 és a 3.4 ábrákon lévő algoritmusokat a 24. és 24. oldalon

szimbólum. Ez csupán arra szolgál, hogy a gép meg tudja állapítani, hogy hol van a vége az egyik összeadandónak és hol kezdődik a másik. A gép a következők szerint jár el. Megkeresei a szeparátor jelet, és megállapítja, hogy hol kezdődik a második szám. Ezt a számot lecsökkenti eggyel. Utána vissza megy az első számhoz, azt pedig megnöveli eggyel. Ezt a tevékenység

```
while b > 0 do
   begin
      b := b - 1;
      a := a + 1;
   end;
```

4.20. ábra. Az összeadás algoritmusa

sort egészen addig ismétli, amíg a második szám 0–val lesz egyenlő. Ha ez bekövetkezik, akkor megáll. Az elv nagyon egyszerű, tulajdonképpen arról van szó, hogy van két „doboz" az egyikben van a darab dolog, a másikban pedig van b darab dolog. Ezek után elkezdjük átrakosgatni a második doboz tartalmát az elsőbe egyesével. Abban a pillantaban, amikor a második doboz kiürül, az elsőben pontosan $a + b$ darab dolog lesz. A 4.20 ábrán látható, hogy miként valósítanánk ezt meg PASCAL nyelven. Nem több, mint egy while–ciklussal megadott iteráció. Azonban Turing–gépre átültetni csak ezt az egyetlen egy while–ciklust is, meglehetősen munkaigényes feladat. Terjedelmi okok miatt itt nem vállalkozunk arra, hogy részletesen ismertessük, de a CD mellékleten megtalálható a feladat *Automaton* programmal való megoldása a `BinOsszeAd.aut` fájlnév alatt. Ha ezt megnyitjuk, akkor láthatjuk az egyik lehetséges megvalósítást, és kipróbálhatjuk, hogy tényleg működik. A projekt főprogramja a MAIN névre halgató Turing–gép, az inputot a fentebb ismertetett formában kell megadni, és induláskor a fejet az input szó legelső szimbólumára kell állítani. Érdemes tanulmányozni a projektben lévő ISZERO nevű gép működését, amely annak a feltételnek az eldöntésért felelős, hogy a második szám 0–val egyenlő–e.

Befejezés

A fenti oldalakon a mai modern számítógépek szerkezetének és működésének elemzésén keresztül eljutottunk egy olyan egyszerűsített számítóeszköz modelljéhez, amely már eléggé egyszerű ahhoz, hogy az általa végzett számítás sorozatot matematikai fogalmakkal is megragadhassuk és elemezhessük. Első megközelítésben talán meglepő — ha összehasonlítjuk a valódi számítógépek komplexitását a Turing–gépek egyszerűségével —, hogy milyen kevésre van szükségünk ahhoz, hogy mindenünk meglegyen, ha automatizálni akarjuk a matematikai számításokat. Az Alan Turing által kidolgozott számítóeszköz a matematikai és számítástechnikai tudományokban azért jelentett nagy előrelépést, mert hozzájárult olyan nehezen, vagy egyáltalán nem definiálható fogalmak tisztázásához, mint az algoritmus, automata, eldönthetőség, kiszámíthatóság stb.

Láttuk azt is, hogy ez a tudományterület egy olyan területe a matematikai kutatásnak, ahol számos megoldatlan probléma és nyitott kérdés van. Leibniz álma még az volt, hogy az emberi gondolkodást és szellemi tevékenységet le lehet írni formállogikai törvényekkel, és elvileg lehet olyan gépet építeni, amely utánozza az emberi gondolkodást, és tetszőleges matematikai problémát meg tud oldani. Ma már tudjuk azt, hogy matematika teljes automatizálása soha nem lesz megoldható. A XX. század leghíresebb matematikai–logikai eredménye Kurt Gödel nevéhez fűződik, ugyanis Ő volt az első, aki kimutatta, hogy bármilyen axiómarendszerből is építjük fel a matematikát, mindig lesz az egész számoknak legalább egy olyan tulajdonsága, amely a megadott axiómákból nem bizonyítható és nem is cáfolható. Vagyis az igazság erősebb fogalom, mint a bizonyíthatósag, tehát nem minden bizonyítható, ami igaz. Ezek az eredmények azok, amelyek más formában ugyan, de a Turing által felállított elméletben is megjelennek, és azt indukálják, hogy vannak algoritmikusan megoldhatatlan és eldönthetetlen problémák. Ha ehhez még hozzá vesszük a Church–tézist, és elfogadjuk, hogy igaz, akkor el kell fogadnunk azt is, hogy a Turing–gép a matematikai értelmeben vett megismerhetőség végső határát jelenti, tehát bármilyen matematikai kérdést teszünk fel a körülöttünk lévő világ logikai tulajdonságaival kapcsolatban, ha kérdés egyáltalán

megválaszolható, akkor van olyan Turing–gép, amely megadja ezt a választ.

Természetesen azt nem tudjuk megmondani, hogy tényleg igaz–e Church tézise, vagy sem, de az elmondottak alapján úgytűnik, hogy van okunk „hinni" benne és van okunk azt feltételezni, hogy bármilyen számítóeszköz– és algoritmus fogalmat dolgoznak is ki a jövő kutatói, az sem lesz jobb a Turing–gépnél és az asztalunkon álló gépnél.

Ábrák jegyzéke

Irodalomjegyzék

[AKS] Agrawal, M. – Kayal, N. – Saxena, N.: *Primes is in P*, letölthető [URL2]–ről

[BI] Bach Iván: *Formális nyelvek*, TypoTEX Kiadó, Budapest, 2001

[BR] Braithwaite, R. B.: *Introduction to Gödel's Theorem*, letölthető [URL3]–ról

[FG] Farkas Gábor: *Programozás elmélet*, Veszprémi Egyetemi Kiadó, Veszprém, 1994

[GL] Gács Péter – Lovász László: *Algoritmusok*, Tankönyvkiadó, 1987

[GK] Gödel, Kurt: *On formally undecidable propositions of Principia Mathematica and related systems I.*, letölthető [URL4]–ről

[HF] Hartung Ferenc: *Bevezetés a numerikus analízisbe*, Veszprémi Egyetemi Kiadó, Veszprém, 1999

[HR] Hersch, Ruben: *A matematika természete*, TypoTEX Kiadó, Budapest, 2000

[HA] Hraskó András (Szerk.): *Új matematikai mozaik*, TypoTEX Kiadó, Budapest, 2002

[LM] Lavrov, I. A. – Makszimova, L. L.: *Halmazelméleti, matematikai logikai és algoritmuselméleti feladatok*, Műszaki Kiadó, Budapest, 1987

[LL] Lovász László: *Algoritmusok bonyolultsága*, ELTE egyetemi jegyzet, 1994

[M] Manyin, J. I.: *Bevezetés a kiszámíthatóság matematikai elméletébe*, Műszaki Kiadó, Budapest, 1986

[NJ] Ropolyi László (Szerk): *Neumann János válogatott írásai*, TypoTEX Kiadó, Budapest, 2003

[OJ] Oláh Judit (Szerk): *Matematikai zseblexikon*, Akadémiai Kiadó – TypoTEX Kiadó, Budapest, 1992

[P] Papadimitriou, Ch. H.: *Számítási bonyolultság*, Egyetemi Tankönyv, Budapest, 1999

[RISz] Rónyai Lajos – Ivanyos Gábor – Szabó Réka: *Algoritmusok*, TypoTEX Kiadó, Budapest, 1998

[RJ] Rothe, Jörg: *Some facets of complexity theory and cryptography: a five-lectures tutorial*, Heinrich Heine Universität, Düsseldorf, 1999 (letölthető [URL5]–ről)

[SL] Surányi László: *Metaaxiomatikai problémák*, TypoTEX Kiadó, Budapest, 1997

[SZ0] Szalkai István: *Diszkrét matematika és algoritmus elmélet alapjai*, Veszprémi Egyetemi Kiadó, Veszprém, 2001

[SZ1] Szalkai István: *Az algoritmus fogalma, bonyolultsága,NP-teljesség*, Bólyai füzetek 1. (BJMT), Veszprém, 1999

[SZ2] Szalkai István: *NP-Completeness*, oktatási segédlet (kézirat), 1997

[SZ3] Szalkai István: *The Proof of Gödel's 1'st incompleteness theorem*, oktatási segédlet (kézirat), 1997

[TA] Tanenbaum, Andrew. S.: *Számítógép architechtúrák*, Panem Kiadó, Budapest, 2001

[TL] Tímár Lajos.: *A programozás hét alapeleme, a Turbó Pascal és én I.*, Veszprémi Egyetemi Kiadó, Veszprém, 1998

[T] Trahtenbrot, B. A.: *Algoritmusok és absztrakt automaták*, Műszaki Kiadó, Budapest, 1978

[V] Veldhuizen, T. L.: *C++ Templates are Turing Complete*, letölthető [URL6]–ról

[VR] Vollmar, Roland: *Sejtautomata–algoritmusok*, Műszaki Kiadó, Budapest, 1982

Web oldal hivatkozások[8]

[URL1] `http://www.ank.sulinet.hu/TANTARGY/SZAMTECH/alapismeretek/tortenet/szamttor/index.htm`

[URL2] `http://www.cse.iitk.ac.in/news/primality.html`

[URL3] `http://www.ddc.net/ygg/etext/godel/godel2.htm`

[URL4] `http://www.ddc.net/ygg/etext/godel/godel3.htm`

[URL5] `http://www.eccc.uni-trier.de/eccc-reports/2001/TR01-096/Paper.pdf`

[URL6] `http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf`

[8]Ezen dolgozat írásakor az itt felsorolt web helyek még (remélhetőleg) léteznek, de semmi garancia nincs rá, hogy ez hosszabb távon is így van, ezért az ezekről származó anyagok megtalálhatóak a CD mellékleten.

NYILATKOZAT

Alulírott Cziráki Tamás, a Veszprémi Egyetem Matematikai és Számítástechnikai Tanszékének informatika tanár szakos hallgatója kijelentem, hogy a 2004. évben készített „*A számítástudomány alapjainak szemléltetése az oktatásban*” című diplomadolgozat saját szellemi munkám eredménye.

2004. április 26.

.....
Cziráki Tamás
egyetemi hallgató