DIPLOMAMUNKA

Kádek Tamás

Debrecen

2007

Debreceni Egyetem
Informatikai Kar

Induktív logikai programozás

Témavezető:
Dr. Várterész Magda
egyetemi docens

Készítette:
Kádek Tamás
programtervező matematikus

Debrecen
2007

Tartalomjegyzék

Bevezetés

Az információ technológia fejlődése során egyre inkább halad az emberközeli megoldások felé. Ez a tendencia nem csak felhasználói termékek, mindennapinak, sőt nélkülözhetetlennek számító alkalmazói megoldásainak vonatkozásában figyelhető meg (gondolva itt például akár egy mai szövegszerkesztő alkalmazás képességeire, vagy az egyszerű Web-böngészők által nyújtott szolgáltatások sokféleségére). A mára már mindannyiunk asztalán elérhető számítási teljesítmény lehetővé teszi, hogy a programozó munkáját új megközelítésben szemlélhessük. E teljesítménytöbblet alkalmat ad arra, hogy az informatika időről időre képes legyen új paradigmák mentén megújítani önmagát. Ez a képesség szüntelenül szélesíti a programozó eszköztárát, mely újabb és újabb rétegeket képezve távolodik a konkrét számítógép architektúrától, és biztos lépésekkel közelít a humán gondolkodásmód egyes elemei felé. Így jutunk el a gépi kóddal definiált jól behatárolható utasításkészlettel megvalósított algoritmusoktól a tanulni képes eljárásokig.

A hardvergyártás látványos eredményeinek és az ebből következő teljesítménynövekedésnek természetes következménye a szoftvergyártás és programozói feladatkör átalakulása. A hangsúly azonban mindvégig a modellalkotáson maradt. A cél tehát az adott problémacsalád reprezentálására alkalmas modell létrehozása, mely lehetővé teszi olyan következtetések levonását, melyek hatékonyan segítik a feladatkörbe tartozó problémák megoldását, így vezetve el minket az eredményig. Előrelépni ekkor a modellek finomításával, az absztrakció szintjének növelésével lehet. A fejlődés ezért a modellező eszközök kifejezőerejében rejlik, a programozói feladatkör változása pedig ezek használatában érhető tetten.

A programozó fegyvertárában bekövetkező leglátványosabb változások egyike az objektumorientált programfejlesztés lehetősége volt. Az alprogramtól (pl.: Basic, Fortran) a metódusig (pl.: Delphi, Java) vezető út jól mutatja a szemléletben jelentkező különbségeket. Elég csak a polimorf metódusok által biztosított előnyökre gondolni, figyelembe véve, hogy a megvalósításhoz szükséges erőforrásigény – például egy virtuális metódustáblában történő keresés szükségessége minden metódushívás előtt – szinte teljesen elhanyagolható. A kérdés csupán az marad, mely modellező eszközt érdemes használni, illetve mely paradigma mentén reprezentálhatjuk leghatékonyabban a konkrét feladatot.

Jelen dolgozat témája, hogy bevezessen minket az induktív logikai programozásba, mely a matematikai logika eredményeit alapul véve terjeszti ki a kifejezőerőt. Ezzel lehetővé válik az algoritmus megadás jóval tömörebb formája, illetve a mesterséges intelligencia egyes eredményeinek felhasználása a programozásban. A cél az, hogy a mindebben rejlő lehetőségeket fokozatosan, a hagyományos megközelítéstől indulva vezessünk be, közben sorra véve a logika térhódításának területeit. Mindezt egyetlen következetesen végigvitt mintapélda segítségével fogjuk megtekinteni. A példában megadott problémákban felmerülő kérdések, illetve azok megválaszolási módja fogja kirajzolni az induktív logikai programozáshoz vezető utat, és egyben a demonstrálásra is ez szolgál.

1. Az alapprobléma

Az induktív logikai programozás lehetőségeinek, illetve eszközkészletének bemutatása nem lenne teljes anélkül, hogy részletesen végig ne járnánk a hozzá vezető utat. Ehhez egy olyan mintapéldára van szükség, melynek segítségével be tudjuk mutatni az adat – majd később a tudás – reprezentálásának folyamatát, valamint a problémaleírás logikai eszközeit. Mindemellett össze is kell tudnunk hasonlítani a fenti eszközöket a jól ismert programozói fogásokkal.

Legyen a mintafeladatunk az egyes egyetemi hallgatók megtámogatása egy tanulmányi előrehaladást követő rendszerrel! Ennek feladata, hogy nyilvántartsa, hogy a hallgató (a saját tanterve tekintetében) milyen követelményeket teljesített, valamint képes legyen segítséget nyújtani félévkezdéskor a felvenni kívánt tanegységek kiválasztásában. Alapkövetelmény, hogy a rendszer legyen tekintettel a korábbi teljesítésekre, illetve konfigurálható legyen maga a tanterv. Így az első megoldandó kérdés:

„Hogyan reprezentálhatjuk az adatokat?”

Elvárjuk, hogy a program végezzen ellenőrzést a teljesíthető tantárgyak vonatkozásában. Legyen képes ellenőrizni a tantárgyi előfeltételeket, és figyeljen oda arra, hogy mely esetekben teljesítettük valamennyit. Ezen információk alapján szeretnénk olyan listákat készíteni, melyek a különböző sávokat értékelik aszerint, hogy van-e még kötelezettségünk vele kapcsolatban. Innen már természetes módon adódik a kérdés:

„Milyen tantárgyakat vehetnénk fel?”

Ennél azonban jóval tovább szeretnénk jutni. Mivel a hallgató által a korábbi félévekben meghozott döntések ismertek, fel szeretnénk őket használni az ajánlás megtétele során. A kérdés tehát átfogalmazható a következőképpen:

„Mely tantárgyakat érdemes felvenni?”

Ekkor a hallgató viselkedésének kiértékelésére lesz szükség, és ezt modelleznünk kell a program futása során.

Az első két kérdésre a hagyományos megközelítéssel (melyek alatt most a különböző magasszintű, illetve objektum orientált programozási nyelvek kínálta lehetőségeket értjük) is elég sokféleképp biztosíthatunk választ. Az adatreprezentáció során így az egyszerű struktúrákra, vagy az objektumokra illetve osztályokra történő leképezés volna választható. Míg magát az algoritmust az ezek manipulációját végző utasítássorozatként adnánk meg. A harmadik kérdés ellenben már okoz némi fejtörést. Ez ugyanis felvet két újabb kérdést:

„Hogyan reprezentálhatjuk a tudást?”
„Hogyan tanulhatunk?”

A következőkben az iménti technikákat jórészt mellőzve fokozatosan fogunk eljutni mindezen kérdések megválaszolásához. Ennek során pedig folyamatosan a matematikai logikán alapuló eszközöket tartjuk szem előtt, hogy segítségükkel – ugyancsak fokozatos léptekkel – egészen az induktív logikai programozásig jussunk.

2. Deklaratív megközelítés

A deklaratív megközelítés lehetőséget ad arra, hogy a programozó elszakadjon az objektumorientált algoritmust leíró eszközöktől. Az imperatív nyelvek algoritmus leírása változókon – változtatható memórián – és ezek állapotváltozásait megadó utasításokon alapul, ahol a középpontban a megoldás előállító lépéssorozat áll. A problémamegoldáshoz tehát a „*hogyan*" kérdésre kell választ találni. Ezzel szemben a deklaratív nyelvek fókuszpontjában a „*mit*" meghatározása áll. Nem egy adott eredmény előállításához vezető utat illetve számítássorozatot kell megadnunk, hanem azt kell megfogalmazni, milyen feltételeknek kell eleget tegyen a válasz.

Tekintsünk példának egy szokványos relációs adatbázis-kezelő rendszert. Miután megadtuk az adatmodellt (definiáltuk a táblák szerkezetét és a köztük lévő összefüggéseket illetve megszorításokat), SQL (*Structured Query Language*) utasítások révén férünk hozzá a tárolt adatokhoz. SQL utasításban – eltekintve néhány teljesítményfokozást szolgáló nyelvi kiegészítéstől, mint például a hint-ek – nem írjuk elő, hogy egy több táblát összekötő lekérdezés eredményének meghatározásához először mely adattáblákon iteráljon végig a rendszer, csupán az összefüggéseket kell megadni. Könnyen elképzelhető az is, hogy ugyanaz a lekérdezés két különböző időpillanatban eltérő végrehajtási terv szerint kerül feldolgozásra. A fejlett optimalizáló eszközök a korábbi futtatások tapasztalatait, illetve a táblastatisztikákat figyelembe véve transzparens módon javíthatnak a végrehajtás mikéntjén. Ezen esetekben is működhet a háttérben tanulni képes algoritmus. A megközelítés erejét az adja, hogy mindezen részletek rejtve maradnak a programozó elöl. (Természetesen az adatbázis megfelelő hangolásához a háttérismeretek megléte elengedhetetlen marad.)

Jelen deklaratív megközelítés bemutatására számos lehetőség kínálkozik, ám legcsábítóbbnak az egyidejűleg alkalmazott technikák sokszínűsége miatt, az XML (*Extensible Markup Language*) szabványokon alapú megvalósítás látszik. A következőkben lépésről lépésre haladva tekintjük át, hogy egyszerű technikák egymásra építésével hogyan oldható meg a mintafeladat.

Az XML specifikációja 1996-ban került ki a World Wide Web Consortium (W3C) szárnyai alól, és vált ajánlássá. A cél egy SGML (*Standard Generalized Markup Language*) kompatibilis, programmal egyszerűen feldolgozható, és széles körben támogatott nyelv létrehozása volt, mely lehetővé teszi az adatok struktúrált leírását, olyan formában, hogy az emberi szemlélő szá-

mára is értelmezhető maradjon. Az eddig eltelt idő lehetővé tette a megfelelő feldolgozóprogramok implementálását és a XML-en alapuló technológia széleskörű megjelenítését elsősorban web-es alkalmazásokban, de számos más területen is. A mai adatbázis-kezelő rendszerekben is alapkövetelmény az XML dokumentumok feldolgozásának megléte és terjedőben vannak az XML alapú adatbázisok.

Az XML dokumentum az adatokat fa struktúrában leíró eszköz, ahol már csak a meta adatok megadására is számos módszer terjedt el. XML Schema segítségével a dokumentum definíciója maga is XML dokumentum lehet, vagy használható akár a külön erre a célra kialakított – bár az összefüggések és adattípusok definiálása tekintetében valamelyest szűkebb lehetőségeket adó – DTD (*Document Type Definition*) nyelv. Élve az utóbbi lehetősséggel, az adatstruktúra megadása után a mintaadatokkal feltöltött dokumentum egyszerű és letisztult.

A feldolgozásnak is létezik szabványa. A jelenlegi adatszerkezet akár egy Java program bemeneteként is szolgálhat. A DOM (*Document Object Model*) megoldás a teljes dokumentumot képezi le a memóriába, de ha az költségesnek hangzik, SAX (*Simple API for XML*) feldolgozó segítségével eseményvezérelt beolvasás is elképzelhető. Ezzel azonban még egyetlen lépést sem tettünk a deklarativitás irányába. Jelen vizsgálatunk tárgya így Java kódrészlet helyett legyen az XSL (*Extensible Stylesheet Language*) ajánláscsalád része, mely a következőket foglalja magában:

- XSL Transformations (XSLT)
- the XML Path Language (XPath)
- XSL Formatting Objects (XSL-FO)

A formázó objektumok (*formatting objects*) használatának bemutatásától eltekintek, de példaként felhozhatók jelen dokumentum ábrái, melyek mint vektorgrafikák FO fájlokban kerültek definiálásra. Ezen XML fájlok szerkezetében sok, a HTML lapok megjelenítésének kapcsán már jól ismert szerkezet szolgálja a formázott kimenet előállítását.

XML feldolgozás lehetőségei

A teljesség kedvéért ne feledkezzünk meg arról sem, hogy XML dokumentumban akár egy programnyelv forráskódjait is ábrázolhatjuk. Gondoljunk csak a JSP (*Java Servlet Pages*) állományokra, ahol Java kódrészletek és XML elemek ötvözete segítségével generálható egy Java

osztály definíciója. Vagy tekintsünk egy nagyobb lélegzetvételű Java projekt fordításának menetét leíró ANT build script-et, mely a C projektek esetében kedvelt make fájlok megfelelője.

Ezen a ponton ismét hasznunkra válik az ajánlás népszerűsége, ugyanis a legelterjettebb mai böngészőkben elérhető XSLT illetve XPath feldolgozó. A létrehozott mintaprogramok megtekintéséhez így elegendő egy Internet Explorer vagy egy Firefox alkalmazás megléte, bár a szokásos kompatibilitási problémák elkerülése végett csak az 1.0-as ajánlás elemeire hagyatkoztam. A feladat tehát egy XSLT dokumentum (mely természetesen szintén XML) létrehozása, mely a korábban bemutatott formátumban definiált dokumentumokat képes átalakítani HTML dokumentummá. Lehetséges kimenetként szóba jöhetne újabb XML dokumentum vagy egyszerű szöveges állomány is.

Az XSLT rejti magában a deklaratív kódot. A dokumentumot csomópontról csomópontra haladva dolgozhatjuk fel a fában, ahol a különböző csomópontokhoz egyedi feldolgozást rendelhetünk, melyen belül lehetőség van elágazások, illetve iterációk megadására.

A feldolgozás erejét az XPath kifejezések adják. Példaként tekintsük a mintafeladat adatintegritásának ellenőrzését. Az XSLT feldolgozás megáll, ha a tantárgyak leírásában ismeretlen előfeltétel szerepel. A terminálás feltétele a következő:

```
not (count(/subjects/group/subject/required[count(id(@id))=1]) -
count(/subjects/group/subject/required) = 0)
```

Az ellenőrzés különbségképzésen alapul. A jobb oldali összeg az összes előfeltétel számát adja meg. A `count` függvény az argumentumában megadott útra illeszkedő elemeket számlálja meg: az XML által leírt fa struktúrában azon csomópontok számát kapjuk, melyekhez vezező élsorozat csúcsai a fenti címkékkel vannak ellátva. A bal oldalon plusz feltételként jelentkezik, hogy csak olyan előfeltételek vehetők számításba, melyek `id` attribútumában szereplő azonosítóval létezik csomópont. Ha a bal és jobb oldal nem egyezik meg, az csak inkonzisztencia esetén lehetséges. Természetesen az ellenőrzés elkerülhető lett volna, ha a `required` elemben `IDREF` típusú attribútum szerepelne `NMTOKEN` helyett. Ekkor ugyanis a dokumentum nem létező előfeltételre történő hivatkozás esetén érvénytelen lenne, ami azonban azt eredményezné, hogy a feldolgozás meg sem kezdődhet. Jelen megoldás viszont lehetővé teszi, hogy hiba esetén másként folytassuk a feldolgozást. A mintaprogram ekkor kigyűjti az ismeretlen hivatkozásokat, majd hibaüzenettel terminál.

A féléves tantárgyi átlag kiszámítása rekurzív megvalósítása miatt emelhető ki a többi közül. A rekurzív algoritmusok jelentősége a szemléletből következik. Ugyan lehetőség van iteráció

és szelekció kialakítására, de a változók csak egyszer kaphatnak értéket, imperatív algoritmus nem képzelhető el. Az `_average` template egy rekurziós lépésben egy tantárgyból szerzett érdemjegy alapján növeli meg a megszerzett krediteket. Az új kreditérték:

```
$credits + id($nodeset/exam[position()=$index]/@subject)/@credit *
$nodeset/exam[position()=$index]/@result
```

Ahol a rekurziós lépések paraméterei a következők:

- `index`: a következő feldolgozandó elem sorszáma,
- `nodeset`: a feldolgozandó elemek,
- `credits`: a megelőző kreditérték.

Ha a csomópontok elfogytak, a végeredményt két tizedesjegy pontosan jelenik meg. Ehhez persze szükség van az összes megszerezhető kreditek számára, melynek kiszámítására `sum` függvény áll rendelkezésre.

```
<xsl:variable name="all" select="sum(id($nodeset/exam/@subject)/@credit)"/>
<xsl:value-of select="round(($credits div $all)*100) div 100"/>
```

A súlyozott tantárgyi átlag meghatározása a template megfelelő paraméterezésével érhető el. Amennyiben az aktuális csomópont vizsgákat tartalmaz, az eredmény a következőképp adódik:

```
<xsl:call-template name="_average">
    <xsl:with-param name="index" select="1"/>
    <xsl:with-param name="credits" select="0"/>
    <xsl:with-param name="nodeset" select="."/>
</xsl:call-template>
```

A fában történő navigáció kettős. XSLT template elemek segítségével járjuk végig a szerkezetet, ahol az egyes csomópontokat elérve eseményvezérelten változik a feldolgozás. Az esemény az aktuális feldolgozás tárgyát képező csomópont illeszkedése a template-ben megadott mintára. Másfelől, a feldolgozás során a XPATH kifejezések segítségével újabb csomópontlisták állíthatók elő és indítható el a feldolgozásuk.

A folyamat végén egy HTML lap áll elő. A kigenerált tartalom pillanatképe a tanulmányi előrehaladásnak. Segítségével egy egyszerű, de jól strukturált adattömegből meghatároztuk,

hogy mely tárgyak felvétele vált lehetővé, mely sávok illetve tanegységek teljesítésére kell még figyelmet fordítani, és milyen eredménnyel zárultak a korábbi félévek. De a cél – hogy legalább egy Java alkalmazás erejét elérjük – még messze van. Feltűnő azonban, hogy a deklaratív megfogalmazás nem csak sokkal tömörebb, hanem a kifejezések reprezentatívabbak is, hiszen a részeredményeket és nem azok számításának miként jét definiálják.

A generált HTML lap nem a folyamat vége. Az oldalak dinamizálására több script-nyelv is rendelkezésre áll. A szükséges technológia két – szintén W3C által kidolgozott – ajánláson alapul. A weblapokat az őket leíró DOM objektummodellen keresztül mozgathatjuk meg, Ecmascript nyelv segítségével. Ezek többnyire javascript kódok formájában jelennek meg a lap forrásában. Ismét objektumorientált nyelv van a kezünkben, mely prototípusfogalomra épül.

A Java megközelítés főbb elemei átültethetők, osztálydefiníciók mintájára prototípusokat gyárthatunk. A polimorfizmus függvény értékű attribútumok segítségével modellezhető. A prototípusban definiált attribútumok ugyanis öröklődnek, felüldefiniálásuk pedig akár a konstruktorban is megtörténhet. A bezárás eszközrendszere és a szigorú típusosság előnyei elvesznek, de ez inkább csak kényelmetlen, mivel nagyobb odafigyelést igényel a programozótól. Sajnos a nyelv rugalmassága a teljesítmény rovására is megy, és a böngészőprogram által interpretált script futási sebességét össze sem érdemes hasonlítani egy JVM (*Java Virtual Machine*) képességeivel.

Ontológia és szemantika

Az XML dokumentumokba rendezett információkezelés korántsem áll meg a strukturált adatábrázolásnál. Joggal merül fel a kérdés, hogy lehetséges volna-e egy teljes szakterület minden tudását reprezentálni? Ekkor azonban jóval összetettebb feladathoz jutunk. Egy tudományterület leírásához ugyanis fogalmainak tárolására van szükség. Az összefüggések tárolása viszont magával vonja olyan alkalmazások igényét, melyek ezen összefüggések alapján következtetések levonására képesek. Szükségessé válik a fogalomalkotás specifikációja, ontológiák létrehozása, valamint a következtetési láncok helyességét biztosító leíró logikák alkalmazása.

A leíró logikák (*Description Logic*) jelentik a fogalmi rendszer megadásának eszközét. Az egyes leíró logikák közti különbség abban mutatkozik meg, hogy milyen lehetőségeket kínálnak a rendszer definiálásához. Az azonban minden esetben közös jellemzőjük, hogy a megfogalmazott információt – a tudásbázist – az információ jellege szerint két részre bontjuk:

T-box: a fogalmakat bevezető terminológiai (*Terminological*) állításokra, és

A-box: az egyedekre vonatkozó kijelentésekre (*Assertion*).

Alapjában véve fogalmakból és szerepekből építkezhetünk:

- A fogalmak egy valamilyen szempont vagy közös tulajdonság által ugyanazon csoportba sorolt egyedek halmaza. A halmazképzés módjai jellemzik az adott leíró logikai nyelvet.
- A szerep az egyedek közti kapcsolat megadására szolgál, binér reláció. A relációk – ismét az adott logika függvényében – más relációk segítségével is definiálhatók.

Ezzel egy elsőrendű logikához jutottunk, ahol az egyedek összessége megfelel az univerzumhalmaznak. A fogalmak olyan predikátumszimbólumok, melyek egy paraméterrel bírnak. Pontosan akkor igaz egy interpretációban egy fogalmat reprezentáló predikátum értéke, amennyiben az argumentum term olyan univerzumelem, melynek megfeleltetett egyed eleme a fogalmat definiáló halmaznak. A szerepek helyébe pedig olyan predikátumszimbólumot képzelhetünk, mely egymással relációban álló egyedpárok esetén bizonyul igaznak. Leíró logikák esetében elsőrendű logikánál többre nem is lesz szükség.

Leíró logikák

A leíró logikák alapeszközei a fogalmakból illetve szerepekből gyártható kifejezések, melyek a következők:

fogalomkifejezés: melyet fogalom-konstruktorok segítségével hozható létre. Illetve a legegyszerűbb fogalom-kifejezés az atomi-, az univerzális- ($\top$ minden univerzumelemet tartalmazó), és az üres ($\bot$ egyetlen univerzumelemet sem jellemző) fogalom,

szerepkifejezések: melyek építésére jóval kevesebb lehetőség kínálkozik. Az atomi szerepek inverze definiálható az inverz szerep-konstruktor segítségével,

terminológiai axiómák: melyek segítségével az egyes fogalmakról illetve szerepekről adhatunk meg állításokat, például tartalmazási vagy ekvivalencia reláció formájában.

A fenti eszközök valamely részhalmazát képezve különböző leíró logikai nyelvekhez illetve nyelvcsaládokhoz jutunk. A különböző nyelvek létjogosultságát két tulajdonságuk adja. Egyfelől bármelyik reprezentálható lenne az elsőrendű logika eszközkészletével is, mivel azonban az nem eldönthető, így annak csak egy részét használjuk fel. Másfelől már néhány fogalom- és

Formalizmus	Elsőrendű megfeleltetés
$\neg C$	$\mathrm{P}_{\neg C}(x) \rightleftharpoons \neg \mathrm{P}_C(x)$
$C_1 \sqcap C_2$	$\mathrm{P}_{C_1 \sqcap C_2}(x) \rightleftharpoons \mathrm{P}_{C_1}(x) \wedge \mathrm{P}_{C_2}(x)$
$C_1 \sqcup C_1$	$\mathrm{P}_{C_1 \sqcup C_1}(x) \rightleftharpoons \mathrm{P}_{C_1}(x) \vee \mathrm{P}_{C_2}(x)$
$\forall R.C$	$\mathrm{P}_{\forall R.C}(x) \rightleftharpoons \forall y\,(\mathrm{P}_R(x,y) \supset \mathrm{P}_C(y))$
$\exists R.C$	$\mathrm{P}_{\exists R.C}(x) \rightleftharpoons \exists y\,(\mathrm{P}_R(x,y) \wedge \mathrm{P}_C(y))$
$\geqslant nR.C$	$\mathrm{P}_{\geqslant nR.C}(x) \rightleftharpoons \exists y_1 \ldots y_n \left(\bigwedge_{i=1}^{n} \mathrm{P}_R(x,y_i) \wedge \bigwedge_{i=1}^{n} \mathrm{P}_C(y_i) \wedge \bigwedge_{i<j} y_i \neq y_j \right)$
$\leqslant nR.C$	$\mathrm{P}_{\leqslant nR.C}(x) \rightleftharpoons \exists y_1 \ldots y_{n+1} \left(\bigwedge_{i=1}^{n+1} \mathrm{P}_R(x,y_i) \wedge \bigwedge_{i=1}^{n+1} \mathrm{P}_C(y_i) \supset \bigvee_{i<j} y_i = y_j \right)$

2.1. táblázat. Fogalomkonstruktorok ([8] alapján)

Megnevezés	Formalizmus	Elsőrendű megfeleltetés
atomi szerep	R_A	$\mathrm{P}_{R_A}(x,y) \rightleftharpoons R_A(x,y)$
inverz szerep	R_A^-	$\mathrm{P}_{R_A^-}(x,y) \rightleftharpoons R_A(y,x)$

2.2. táblázat. Szerepkifejezések ([8] alapján)

szerep konstruktor elegendő lehet adott célprobléma megoldására, ahol a nyelv szűkössége természetesen a feldolgozási sebesség javára válik. Épp úgy, ahogy elsőrendű logikában is gyakran hívunk segítségül különféle normál formákat az alkalmazásokban, vagy épp a bizonyításokhoz.

Kézenfekvőnek tűnik adatbázisok felhasználása az egyedekre vonatkozó kijelentések tárolása során. Ekkor viszont figyelembe kell venni, hogy a klasszikus relációs adatbázis-kezelő rendszerek lekérdezéseiben, ha valamely egyedhez nem társítunk hozzá egy tulajdonságot, az azt jelenti, hogy nem is rendelkezik azzal. A leíró logikák esetében – a nyílt világ szemantikának köszönhetően – ez nem teljesül, hiszen nem elég egyetlen interpretációt tekinteni ahhoz, hogy következtetést vonjuk le. A feldolgozás módja továbbra is kritikus.

Alapvető leíró logikai nyelvek eszköztárának szemléletére gyakran használják az $\mathcal{AL}$ nyelvcsalád tagjait [8], ahol alapesetben a fogalomkonstruktorok közül a metszet ($C_1 \sqcap C_2$), az értékkorlátozás ($\forall R.C$), az atomi negáció és az egyszerű létezési korátozás áll rendelkezésünkre, valamint a fogalomtartalmazási axióma ($C_1 \sqsubseteq C_2$) használható fel. Például egy tanulmányi előrehaladást követő rendszerben a felvehető tantárgyak fogalma egybeesik azon tantárgyakkal, melyek minden előfeltétele sikeresen teljesített.

$$\mathsf{felveheto} \sqsubseteq \mathsf{tantargy} \sqcap \forall \mathsf{elofeltetele.sikeres}$$

Formalizmus	Elsőrendű megfeleltetés
$C_1 \equiv C_2$	$\mathrm{P}_{C_1 \equiv C_2} \rightleftharpoons \forall x(\mathrm{P}_{C_1}(x) \equiv \mathrm{P}_{C_2}(x))$
$C_1 \sqsubseteq C_2$	$\mathrm{P}_{C_1 \sqsubseteq C_2} \rightleftharpoons \forall x(\mathrm{P}_{C_1}(x) \supset \mathrm{P}_{C_2}(x))$
$R_1 \equiv R_2$	$\mathrm{P}_{R_1 \equiv R_2} \rightleftharpoons \forall xy(\mathrm{P}_{R_1}(x,y) \equiv \mathrm{P}_{R_2}(x,y))$
$R_1 \sqsubseteq R_2$	$\mathrm{P}_{R_1 \sqsubseteq R_2} \rightleftharpoons \forall xy(\mathrm{P}_{R_1}(x,y) \supset \mathrm{P}_{R_2}(x,y))$
$\mathsf{Trans}(R)$	$\mathrm{P}_{\mathsf{Trans}(R)} \rightleftharpoons \forall xyz(\mathrm{P}_R(x,y) \wedge \mathrm{P}_R(y,z) \supset \mathrm{P}_R(x,z))$
$\mathsf{Nominal}(I)$	$\mathrm{P}_{\mathsf{Nominal}(I)} \rightleftharpoons \exists i(\mathrm{P}_I(i)) \wedge \forall xy((\mathrm{P}_I(x) \wedge \mathrm{P}_I(y)) \supset x = y)$
	$\mathrm{P}_{C(a)} \rightleftharpoons \mathrm{P}_C(a)$
	$\mathrm{P}_{R(a_1,a_2)} \rightleftharpoons \mathrm{P}_R(a_1,a_2)$

2.3. táblázat. Axiómák ([8] alapján)

$$\mathsf{felveheto} \sqsupseteq \mathsf{tantargy} \sqcap \forall \mathsf{elofeltetele.sikeres}$$

Természetesen a fogalomtartalmazási axióma segítségével fogalomegyenlősség ($C_1 \equiv C_2$) is leírható, így egyszerűsíthetnénk a fenti kifejezést. Elsőrendű logikával a fogalmakat és szerepeket kezdőbetűikkel helyettesítve:

$$\forall x\,(F(x) \equiv (T(x) \wedge \forall y\,(E(x,y) \supset S(y))))$$

Az alapnyelvet bővítve az unió ($C_1 \sqcup C_1$), a létezési korlátozás ($\exists R.C$) és a teljes negáció ($\neg C$) fogalomkonstruktorok közül eggyel vagy többel, azonos kifejezőerejű nyelvekhez jutunk. A gyakorlati megoldások ennél is tovább lépnek.

Web Ontology Language

A szemantikus világháló gyakorlatába a W3C 2004. február 10-én vezette be az ontológiákat. Ekkor vált ugyanis ajánlássá három nyelv: az OWL (*Web Ontology Language*), az RDF (*Resource Description Framework*) és az RDF Schema. Az OWL – mint ontológianyelv – az RDF sémához hasonlóan, olyan erőforrásokat tartalmaz, melyek lehetővé teszik egy fogalmi rendszer elkészítését. Maga az OWL is tulajdonképpen egy RDF szintaxissal rendelkező nyelv, mivel az OWL eszközök is RDF sémában vannak definiálva.

A szemantikus világháló koncepcióját a web-es adatábrázolás megközelítésének újragondolása adja. A korábbi alkalmazások kimente – tipikusan valamilyen HTML ajánlás mentén formázott, de gondolhatunk akár SVG vagy XSLT FO tartalomra is – a felhasználónak készült, és ennek megfelelően strukturált. Az adatot formátuminformáció veszi körül, ahol a számítógép

egyetlen feladata a megjelenítési szabályok követése. Az alkalmazás ilyenkor nem tudja elemezni vagy értelmezni a tartalmat. A szemantikus megközelítés a gépi adatfeldolgozást segíti. Gondoljunk csak egy RSS (*Really Simple Syndication*) hírcsatornára. A böngésző arról kap információt, hogy a cikket ki, mikor, milyen témában tette közzé, illetve eltárolja, hogy melyiket olvasta már el a felhasználó. Így az ábrázolás lehetővé teszi, hogy kedvenc témáink legfrissebb híreit böngészhessük, kereshessük. Ez persze még csak egy igen apró lépés, de jól mutatja, mire lehet képes a világháló, ha ontológiák segítségével feldolgozhatóvá válik a tartalma.

Maga az OWL eszközrendszer is – hasonlóan a leíró logikákhoz – a felhasznált építőelemek tekintetében több csoportba sorolható, melyekhez más-más feldolgozás társítható:

OWL Full: melyben az RDF és OWL teljes eszköztára szabadon felhasználható, azonban nincs olyan leíró logikai nyelv, melynek megfeleltethető volna,

OWL DL: melyben az OWL Full konstrukciók csak bizonyos megszorításokkal használhatók fel, cserébe viszont létezik megegyező kifejezőerejű leíró logika,

OWL Lite: mely eszköztára valódi részhalmaza az OWL DL-nek, vagyis egyes konstrukciók egyáltalán nem találhatók meg benne, másokat pedig csak megszorításokkal használhatunk.

Az OWL Lite a már ismertetett $\mathcal{ALC}$ nyelv – az $\mathcal{AL}$ nyelvcsalád teljes negációs kiegészítése – elemein túl megengedi a tranzitív- és inverz szerepek, szerephierarchiák ($R_1 \sqsubseteq R_2$) valamint a funkcionális korlátozások (a $\geqslant nR.C$ és $\leqslant nR.C$ számosságkorlátozások $\geqslant 2R$ és $\leqslant 1R$ változata) és adattípusok használatát. Ezt nevezik $\mathcal{SHIF}(\mathcal{D})$ nyelvnek. A felvehető tárgyakra vonatkozó kijelentés alakja ekkor a következő:

```
<owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Tantargy"/>
    <owl:Restriction>
        <owl:allValuesFrom rdf:resource="#Sikeres"/>
        <owl:onProperty>
            <owl:ObjectProperty rdf:ID="elofeltetel"/>
        </owl:onProperty>
    </owl:Restriction>
</owl:intersectionOf>
```

Az OWL DL mindezt a számosságkorlátozások teljes skálájával, és egyedfogalmakkal toldja meg ($\mathsf{Nominal}(I)$), mellyel a $\mathcal{SHOIN}(\mathcal{D})$ nyelvhez jutunk. A leíró logikák emlegetett elemeiről, illetve a nyelvcsaládokról [8, 4. fejezete] ad részletes ismertetést.

Az OWL állományokat érthető okokból nehéz volna XML szerkesztővel kezelni. A konstrukciók kialakítására mára számos fejlesztőeszköz áll rendelkezésre. Népszerű alkalmazás a Protégé, melyben a fenti kódrészlet a

```
Tantargy and (elofeltetel only Sikeres)
```

kifejezés, mint a `felveheto` osztályba tartozás szükséges és elégséges feltétel (fogalomegyenlőségi axióma) kerül megadásra.

3. Szakértő rendszer

Az ontológiák lehetővé teszik az egyes tudományterületek fogalmainak leírását. Adott tehát egy feldolgozásra váró tudásbázis, ahol a továbblépéshez következtetési módszerekre van szükség. A mesterséges intelligencia programokra hárul a feladat, hogy hatékony keresési stratégiák mentén végezzék a problémamegoldást. A követelmény velük szemben, hogy olyan eredményt hozzanak létre, melynek minősége nem magától a stratégiától, hanem az ábrázolt ismeret mennyiségétől illetve pontosságától függ. Tudás alapú rendszerről (*Knowledge Based System*) akkor beszélhetünk, ha a feladatmegoldást egy következtető gép végzi, mely számára az ismereteket külön tudásbázisba szervezetük. A következtető gép nem előre bedrótozott algoritmikus lépések segítségével, hanem a tudásbázis transzformációinak sorozatával dolgozik. Ez a folyamat az adatbáziskezelők működéséhez hasonlóan ez esetben is deklaratív, de a feldolgozás középpontjában nem pusztán adat (passzív, nem értelmezett jelsorozat), hanem tudás[1] áll.

A szakértő rendszerek (*Expert Systems*) olyan tudás alapú rendszerek, melyek egyetlen szakterület ismereteit összpontosítják. Szűk körből származó, de mélyreható információkkal rendelkeznek. Természetesen a reprezentált tudás hitelessége kritikus, a programozónak kiemelt felelőssége van a szakértőkkel történő kommunikációban, illetve a terület feltérképezése és modellezése során. Cserébe, ha a modell elegendően aprólékos és helytálló, a jó szakértő rendszer nagyságrendekkel képes csökkenteni az adott területen vett munkavégzési időt, így hamar megtérülhet.

A klasszikus szakértő rendszer – a logikai megközelítésből fakadóan – szabályelvű. Működésének szemléltetésére kiválóan használható a Clips (*C Language Integrated Production System*) keretrendszer (hiszen jól dokumentált, számos operációs rendszeren elérhető, public domain szoftver). Maga a keretrendszer üres ismeretbázissal rendelkező burok, mely magas szintű szolgáltatásokat biztosít a feldolgozáshoz. A feladat ez által a modellalkotásra koncentrálódhat.

A tény az információábrázolás alapvető formája, adatabsztrakciós eszköz – Clips rendszer-

[1] Szokás különbséget tenni tudás és ismeret közt is, ahol az előbbi csak az emberi tudatban, míg utóbbi szimbolikus leírásként is megjelenhet [4]. A továbbiakban ezen megkülönböztetéstől eltekintek.

ben épp úgy, mint az objektumok és globális változók –, mely statikus kijelentést reprezentál. A rendszer ténylistában tárolja az aktuális tudást, melyet működése során a megfelelő parancsokra új tényekkel egészít ki (`assert`), illetve tényeket távolít el onnan (`retract`). A ténylista nem feltétlenül halmaz, választható olyan – az alapértelmezéstől eltérő – üzemmód, melyben ugyanaz a tény több példányban is bekerülhet a listába. Példaként tekintsük a tantárgyakat, valamint a teljesítésüket megadó tények egy lehetséges ábrázolását:

```
(deffacts subjects "Tantargy es elofeltetellista"
    (subject "M1724" 4 "Diszkrét matematika 1")
    (subject "M1734" 4 "Kalkulus 1")
    (subject "M1725" 4 "Diszkrét matematika 2" requires "M1724"))

(deffacts passes "Teljesitett tantargyak"
    (passed "M1724")
    (passed "M1734"))
```

A Lisp szerű szintaxis a kezdeti tények ábrázolását mutatja. Ezek lesznek azon tényállítások, melyek az alkalmazás betöltése után vagy (`reset`) parancs hatására automatikusan a ténylistába kerülnek. Egyben rendezett tények, melyek egy szimbólumból és az utána következő tetszőlegesen sok mezőből állnak. Nem rendezett tények esetében az egyes mezők nevesítettek és a felvett értékek vonatkozásában megszorítások írhatók elő (például típus), így az imperatív programozási nyelvekből jól ismert struktúra (vagy rekord) fogalmához hasonlítható.

Míg a tény információt, addig a szabály tudást reprezentál. Általa valósul meg a heurisztikus jellegű ismeretek számítógépes ábrázolása, melyek megmutatják, hogy milyen tevékenységekre van szükség egy adott szituációban. Ennek megfelelően a szabálydefiníciók két részből, előzményből és következményből állnak:

- Az előzmény azon feltételek halmaza, melyek mellett alkalmazni kell a szabályt. Maguk a feltételek a ténylistát tesztelik. Kiemelt szerepet játszik a minta – melyet a következtető rendszer mintaillesztő algoritmus segítségével automatikusan értékel ki –, hiszen a feltétel többnyire illeszkedő tények keresését jelenti.

- A következmény a végrehajtás hatását definiáló műveletek halmaza. Egy művelet többnyire a ténylistát manipulálja, például a már emlegetett (`assert`) illetve (`retract`) utasítások segítségével.

Vegyük észre, hogy az imperatív nyelvek feltételes utasítása, és a szabálydefiniálás formailag hasonló kijelentés – „***ha*** *teljesül a feltétel,* ***akkor*** *hajtsuk végre a következőket*” –, mégis

lényegesen különböző eszközökről van szó. A feltételes utasításra a program egy jól meghatározott pontján kerül a vezérlés, a szabálydefiníció ellenben a program teljes élettartama alatt érvényben lehet. Formálisan ezt úgy szokás szemléltetni, hogy a „*ha*” szócska helyett a „*valahányszor*” kifejezést használjuk szabályelvűség esetén.

A következő példában olyan szabályokat adok meg, melyek az előbbi tantárgylistából a teljesített előfeltételek alapján eltávolítják azon tantárgyakat, melyek nem teljesített előfeltétellel rendelkeznek. A szabályokban szemléltetési célból csak az egyszerű mintaillesztésre és a ténylista manipulációra építettem, a további eszközök természetesen tömörebb megfogalmazást is lehetővé tennének.

```
(defrule eliminate_require "A teljesitett elofeltetel eliminalasa"
    ?original <- (subject ?sid ?krd ?nev requires ?req $?others)
    (passed ?req)
  =>
    (assert (subject ?sid ?krd ?nev requires $?others))
    (retract ?original))

(defrule drop_impossible "A teljesithetelten targyak eldobasa"
    ?original <- (subject ?sid ?krd ?nev requires ?req $?others)
    (not (passed ?req))
  =>
    (retract ?original))
```

A szabályok definíciójában lokális változókat is találunk. Ezeket a Clips „`?`” illetve „`$?`” prefix-el látja el. Egy szabályalkalmazásban a váltózók egyszeri értékadással kapnak értéket (mely előbbi esetben egyetlen, míg utóbbiban tetszőleges sok mező lehet). Az értékadás végbemehet a mintaillesztés folyamán – a mintában az aktuálisan illeszkedő tény megfelelő mezőjében illetve mezőiben megjelenő értékkel –, vagy a hozzárendelt érték (ahogyan a $<-$ operátor esetében is megfigyelhető) a sikeresen illesztett tény egésze is lehet. A változók aktuális értéke eztán a hátralévő feltételekben vagy a szabály műveleteiben használható fel.

Stratégiák

A következtető gép feladata, hogy alkalmazza a tudást az információra. A mintaként tekintett keretrendszer mindezt adatvezérelt elven végzi. A feldolgozás a kezdeti tények alapján indul. Meghatározásra kerülnek azon szabályok, melyek alkalmazhatók az aktuális ténylistára

(vagyis az előzményeiket alkotó összes feltétel kielégíthető volt). Ezt követi a kiválasztott szabály következmény részének végrehajtása, mely befolyásolhatja az aktuális tudást. A folyamat mindaddig ismételhető, míg el nem fogynak az illeszkedő szabályok.

Természetesen amennyiben egy szituációban több szabály is alkalmazható, a rendszernek döntést kell hoznia arról, melyiket részesítse előnyben. Erre a forrásszövegtől nagyrészt független módon kapunk választ, a programozónak kevés befolyása van a döntésre (prioritást rendelhet egy-egy szabályhoz). A következtető gép ugyanis implicit, az aktuális konfliktuskezelési stratégia (*current conflict resolution srategy*) alapján dönt a folytatásról. A Clips hét különböző lehetőséget támogat [9, 5.4 alfejezet]:

Szélességi stratégia: (*breath*) a szabályokat aszerint osztályozza, hogy mikortól kerültek fel az alkalmazhatók közé. Az újonnan látókörbe került szabályok alkalmazása hátrébb sorolódik.

Mélységi stratégia: (*depth*) a szélességi stratégia ellentéte. Az új szabályok élveznek előnyt a korábbiakkal szemben.

Komplexitási stratégia: (*complexity*) a szabályokat komplexitásuk szerint osztályozza, és az egyszerűbbeket részesíti előnyben. Azonos komplexitás esetén mélységi stratégiát követ. A komplexitást csak a szabály feltételeket tartalmazó részének összetettsége befolyásolja.

Egyszerűségi stratégia: (*simplicity*) a komplexitási stratégia ellentéte, annyi eltéréssel, hogy azonos komplexitás esetén szintén a mélységi stratégia alapján jár el.

LEX stratégia: (*lexicographic ordering*) eredetileg az OPS5-ben (*Official Production System*) jelent meg először. A tényekhez időcímkét rendel, mellyel szabályokat aktiváló tények időcímkéit csökkenő sorrendbe rendezve időcímke listát kapunk. A szabályokhoz tartozó listát (elemenként balról jobbra haladva) kell összehasonlítani és ez alapján csökkenőleg rendezni őket.

MEA stratégia: (*means end analysis*) – szintén OPS5-ből örökölt – a LEX-el megegyezőképp működik kivéve, hogy a szabályokhoz tartozó időcímke lista elemei nem rendezettek, hanem előfordulásuk sorrendjében adottak. Ha nem sikerülne döntést hozni, LEX stratégia szerint folytatódik a rendezés.

Véletlenszerű stratégia: (*random*) minden szabályalkalmazáshoz véletlen számot rendelve alakít ki sorrendet.

Amennyiben nem használtunk prioritást vagy prioritásegyezőség alakult ki, és a konfliktuskezelési stratégia sem tudta eldönteni a szabályalkalmazás sorrendjét, a Clips következtető gépe önkényesen választja meg a következő szabályt. Érezhető, hogy a választott stratégia nagyban befolyásolja a rendszer teljesítményét, és hatással van az eredményre is.

Interaktivitás

A szakértő rendszerrel szemben támasztott követelmények nem merülnek ki az emberi szakértőhöz hasonló válasz megkonstruálásában. Elvárjuk, hogy interaktív partnere legyen a felhasználónak, kérdéseket lehessen feltenni, melyeket nem csupán megválaszol, hanem meg is indokolja a végeredményt.

A szabályalkalmazások során a létrejött út, valamint annak egyes állomásai adják az indoklást. Mintaalkalmazásunk ajánlattétele során a teljesítésre felkínált tárgyak listájának előállításakor fontos szerepet játszik, hogy mely sávok teljesítése várat még magára. Ezen részeredmény megjelenítését szolgálja a következő szabálydefiníció:

```
(defrule group_finish
    ?original1 <- (group_done ?gnev ?dsum)
    ?original2 <- (group ?gnev ?gsum $?other)
    ( test (>= ?dsum ?gsum))
  =>
    (printout t "Befejezett sáv: " ?gnev " " ?dsum "/" ?gsum crlf)
    (retract ?original1)
    (retract ?original2))
```

A felhasználóval történő interakció természetesen nem ilyen egyoldalú. Ahogy az iménti példában találunk megfelelő konstrukciót szöveges kimenet megjelenítésére `(printout)`, úgy a hagyományos bemeneti utasítások megfelelője sem hiányzik a Clips-ből. Egy `(read)` utasítás eredményét `(bind)` segítségével valamely változóhoz köthetjük, vagy akár tényt is gyárthatunk belőle. Ennél jóval izgalmasabb azonban, hogy a futtatást (szabályalkalmazást) megelőzően, vagy akár a futás után, lehetőség van a ténylista manipulációjára. Ellenkező esetben az aktuális ténylista változatlanul megmarad új szabályok definiálása, illetve betöltése esetén is. Így az egyes szakértő rendszerek felhasználhatják az előző futás végeredményét, ami lehetővé teszi összefűzésüket.

4. Rezolúciós logika

Eddig példánkban a következtető rendszer a szabályalkalmazásokon keresztül, oly módon konstruálta meg a tényállítások halmazából a megoldást, hogy egy kiinduló állapotból következtetett előre, mindaddig míg ez lehetséges volt. Ily módon a következtetés, a kiválasztott stratégia alapján végbemenő szekvenciális szabályalkalmazás sorozat, mint adatvezérelt megoldáskeresés jellemezhető. Ám a gyakorlatban az előrekövetkeztetés nem mindig célravezető, ugyanis sok olyan állapot előállhat, mely nem a kérdésünkre ad választ, hanem az esetleg hibásan megválasztott stratégia melléktermeke. Tipikus példa erre a matematikai tételek bizonyításainak megkonstruálása, hiszen ekkor a bizonyítandó állításnak kitüntetett szerepe van a folyamat során. A következtetési lánc feladata ekkor ugyanis az aktuális tétel igazolásában merül ki, tehát egy konkrét cél felé halad. (Kellemetlen volna, ha más-más stratégia esetén más-más állítást bizonyítanánk be.) Nem előrekövetkeztető adatvezérelt, hanem célvezérelt megoldásra van szükség.

A bizonyításokban található következtetéssorozat – következtetési lánc – folytonosságának, hézagmentességének biztosítója a matematikai logika, mely két összetevőből áll:

$$Logika = Nyelv + Kalkulus$$

Nyelv: olyan szimbólumok és nyelvtani szabályok összessége, melyek lehetővé teszik állítások, kijelentések reprezentálását (gondoljunk csak az aritmetikai állításokat megfogalmazni engedő Ar nyelvre, vagy akár a korábban bemutatott leíró logikai nyelvekre, illetve nyelvcsaládokra), tehát szintaktikát ad meg.

Kalkulus: mint következtető eszközrendszer, mely szintaktikai szabályok (mint például Gerhard Gentzen kalkulusai) segítségével valósítja meg a szemantikus következményrelációt, és így a tételbizonyítás alapja. Mindez akkor érhető el, ha maga a kalkulus helyes, hiszen ekkor a következtetési lánc helyessége is biztosított. Ugyancsak elvárnánk a teljességét is (mi szerint bármely tételt bizonyítására alkalmazható), de ezen a téren a gyakorlatban – mint azt a későbbiekben látni fogjuk – olykor kompromisszumra kényszerülünk.

Maga a logika a tartalmazott eszközrendszer függvényében különböző kifejezőerejű lehet. A gyakorlati alkalmazások ma már a fuzzy, illetve a többértékű logikák elemeire is támasz-

kodhatnak, de jelen példánkban megmaradok a klasszikus elsőrendű (*first order*) logika kínálta lehetőségek mellett.

Elsőrendű rezolúciós kalkulus

Az elsőrendű rezolúciós kalkulus az elsőrendű klózhalmazok (*set of clause*) kielégíthetetlenségének bizonyítási eszköze. Az elsőrendű klóz egy speciális (Skolem normál formájú) zárt formula (avagy mondat), melynek minden változója kötött, univerzálisan kvantált, illetve magja (avagy mátrixa) elsőrendű literálok diszjunkciója. Tetszőleges formula felírható klózhalmazként, hiszen minden formula (a kvantorkiemelési szabályok segítségével) prenexizálható, a prenex formulák magja (implikációs, illetve de-Morgan törvények alkalmazásával) konjunktív normál formára hozható, így a konjunkciós tagokból képzett Skolem normálformák magja csak elemi diszjunkciókat tartalmaz. (Természetesen maga a Skolemizáció is minden elsőrendű formulára elvégezhető, de az eltávolítandó egzisztenciális kvantorok függvényében esetleg új Skolem konstansok illetve Skolem függvények megjelenését eredményezi.) Az ily módon képzett klózhalmaz pedig pontosan akkor kielégíthetetlen, ha maga az eredeti formulahalmaz is az.

A rezolúciós kalkulus alapötlete, hogy az iménti alakban felírt klózhalmaz (most csak nulladrendben vagy ítéletlogikában gondolkodva, ahol a klóz egy elemi diszjunkció) minden elemét kielégítő interpretációk akkor is kielégítik a klózhalmazt, ha abban pontosan egy komplemens literálpárt tartalmazó két klózt helyettesítünk azzal, melyet az iménti kettő diszjunkciójaként kaptunk a komplemens literálpár elhagyása után. Ez a művelet a rezolvensképzés, illetve a kapott klóz az eredeti klózok rezolvense (*resolvent*). Az eljárás elsőrendben is hasonló, de a rezolválhatóság eldöntésére illetve a rezolvensképzéshez a változók és termek miatt illesztő helyettesítés megkonstruálására van szükség. A rezolvensképzés addig tart, míg el nem jutunk az üres (egyetlen literált sem tartalmazó) klózig, vagy elfogynak a rezolválható formulák. Az üres klóz kielégíthetetlen, hiszen két olyan formulából kaptuk, melyek literálok és egymás komplemensei, olyan interpretáció pedig, mely mindkettőt kielégítené, nem létezik.

Az elsőrendű rezolúciós kalkulus helyességét illetve teljességét belátni természetesen jóval komplikáltabb ennél, de mára már jól bejárt út vezet hozzá, melyet többek közt Jacques Herbrand, Thoralf Skolem és Jaakko Hintikka munkássága szegélyez. A precíz bizonyítás megtalálható: [7].

A formulahalmazok kielégíthetetlenségének bizonyítása könnyűszerrel felhasználható a tételbizonyításban. Egy tétel igazolásához ugyanis nem kell mást tenni, mint formulahalmazt

képezni a tudásunkból és a tétel negáltjából. Amennyiben a tétel negáltjával kiegészített formulahalmaz kielégíthetetlen, úgy a tételünk valóban következmény.

Az elsőrendű rezolúció ugyan önmagában helyes és teljes, de a hatékony megvalósításhoz rezolúciós stratégiák bevezetése is szükséges (akár még a teljesség elvesztésének árán is).

Az SLD rezolúció (*Linear input resolution for Definite clauses with Selection function*) olyan elsőrendű rezolúciós, mely a lehetséges klózoknak csupán részhalmazával dolgozik: Horn klózokat használ, ahol a Horn klóz legfeljebb egy nem negált literált tartalmazhat. A következő Horn klózokat különböztetjük meg:

Tény: egyetlen pozitív literálból álló Horn klóz.

Szabály: pontosan egy pozitív és legalább egy negatív literálból álló Horn klóz.

Cél: negatív literálokból álló klóz, a tétel negáltja.

Magát a programot a tények és szabályok – azaz pontosan egy pozitív literált tartalmazó klózok – együttese alkotja. Ezek a definit klózok (*definite clause*). Vegyük észre, hogy a célklóznak rezolvense valamely klózzal, csak a definit klózok egyetlen pozitív literáljából adódhat! (A kiszámítás lépései viszont még nem determinisztikusak, hiszen a célklóz bármely literálja kiválasztható, és ahhoz is több illeszthető definit klóz létezhet.)

Azt, hogy a feldolgozó algoritmus a célklóz mely literáljával próbáljon rezolválni, a kiválasztási függvény (*selection function*) adja meg. A választás eredménye nem szükségszerűen csak a célklóztól függ. A rezolválás után új célklóz jön létre, mely ismételten csak negatív literálokat tartalmaz. Ezzel lineáris levezetéshez jutunk, hiszen a rezolvens mindig a megelőző lépésben előállt klózból keletkezik. Továbbá az aktuális célklóz (mint központi vagy centrális klóz) mellé szükségszerűen a definit klózok közül választunk mellékklózt a rezolváláshoz, így lineáris inputrezolúciónk van. Itt kell megjegyezni, hogy a lineáris inputrezolúció csak Horn logikában teljes, a kifejezőerő csökkenése azonban nem von le az eljárás népszerűségéből.

Adódik a kérdés, miszerint lehetséges volna-e negációt csempészni az SLD rezolúcióba? A triviális válasz a zárt világ feltevés (*closed world assumption*) volna, ahol a negatív következtetés a pozitív állítás hiánya esetén vonható le. A probléma ezzel csupán az, hogy a bizonyíthatatlanság eldönthetetlen, a célklóz a tételnek mint pozitív literálnak a negáltja. Kicsit finomít az elképzelésen, ha a végesen kudarcos levezetést tekintjük sikeres negációnak (*negation as finite failure*), azonban az eljárás helyességét így is elveszítjük. Ugyanis, ha valami nem következménye a definit klózoknak, még létezhet olyan interpretáció, melyben igaz. K. Clark 1978-ban

dolgozott ki egy program kiegészítő eljárást a negáció végesen kudarcos levezetésének megalapozására, mely nyomán SLDNF (*SLD resolution with negation as finite failure*) rezolúcióban tetszőleges számú pozitív vagy negatív literált tartalmazó célklóz kezelhető [2]. Azonban a végtelen kudarcos levezetések tekintetében továbbra sem teljes az eljárás.

Prolog

A Prolog (*programmation en logique*) 1995-ben lett ISO (*International Organization for Standardization*) szabvány, melynek a létrehozását az eredetileg 1972-ben fejlesztett nyelv számos változatának megjelenése indokolta. Lineáris input rezolúcióval dolgozik. Bár a kiválasztási függvény választható lehet, alapműködés szerint visszalépéses (*backtracking*) algoritmussal keresi a célklóz rezolúciós cáfolatát, oly módon, hogy annak első literálját dolgozza fel először. Ekkor a megfelelő partícióba tartozó definit klózokat definiálásuk sorrendjében a legáltalánosabb illesztő helyettesítés mellett értékeli ki, ahol a partíció a célklóz választott literáljának pozitív megfelelőjét tartalmazó klózok sorozatát jelöli. A visszalépéses kereső, a definit klózok természetes nyelvi értelmezése, valamint a rezolúciós tételbizonyító algoritmus lehetővé teszi, hogy egy Prolog programra különböző paradigmák mentén tekinthessünk:

- A Horn logika automatikus tételbizonyító eszköze, melyben a programot az elsőrendű logika építőköveiből rakjuk össze. Definiálhatunk termeket és predikátumszimbólumokat, valamint Horn klózok segítségével összetett formulákat, melyekből klózhalmaz épül, és mely alapján igazolható egy megadott tételformula. A program futása a tétel negáltjával kiegészített klózhalmaz kielégíthetetlenségét bizonyítja.

- A definit klózok tényeket (negálatlan literált) és szabályokat (implikációt) írnak le. A szabályok szerkezete, feje (egyetlen negálatlan literálja) illetve törzse (a negált literálok) következmény illetve feltétel részként értelmezhető. Így tulajdonképpen szakértő rendszert adtunk meg. A program végrehajtása a szakértő rendszer célvezérelt megvalósítását jelenti. Mindezt erősítendő, az `assert` és a `retract` beépített predikátumok segítségével klózok definiálhatók vagy távolíthatók el dinamikusan, épp úgy, ahogy ezt korábban Clips esetén láttuk.

- A szabályok segítségével eljárásokat definiálunk, ahol az eljárás deklarációja a szabály feje, a definíciója pedig a szabály szekvenciaként értelmezett törzse. A program végrehajtása a célklózzal indul, ez lesz a program belépési pontja. Az eljáráshívás nem minden

esetben sikeres. Sikertelen hívás hatására visszalépés következik be, és a végrehajtás egy újabb irányban folytatódik.

Mindhárom szemlélet fontos szerepet játszik egy Prolog program kialakítása során. Habár kényelmes volna egyszerűen a rezolúciós tételbizonyítás eszközeként tekinteni rá, nem feledkezhetünk meg arról, hogy használati erejét számos procedurális környezetben megszokott szerkezet átvételének is köszöni. Ugyanakkor, ha az eljárásorientált szemlélet mögött nem látjuk a rezolúciós levezetést és a levezetési fát, mely a visszalépéses algoritmus és a rezolvendképzés során kialakul, lehetetlen volna algoritmust megfogalmazni vele.

A mintaproblémánk lehetőséget biztosít arra, hogy általa betekintést nyerjünk a Prolog hármas szemléletvilágába. Tekintsük először a már korábban ismertetett tantárgyak és teljesítések – mint tényállítások – megadását:

```
% Sávkód, Megnevezés, Kötelező kredit
group('f1', 'Elso feleves kotelezo', 15).
...
% Tárgykód, Kredit, Sávkód, Megnevezés, Előfeltétellista
subject('M1724', 4, 'f1', 'Diszkrét matematika 1', [] ).
subject('M1725', 4, 'f2', 'Diszkrét matematika 2', ['M1724']).
subject('I1205', 5, 'f3', 'Programozás 2',         ['I1202','I1203']).
...
% Tantárgykód, Érdemjegy
exam('I1301',5).
```

A tényállítások törzsnélküli szabályok, tehát predikátumok. A predikátumokat neve és argumentumszáma jellemzi, ahol argumentumként tetszőleges term szerepelhet. Term-ként értelmezhetjük ebben az esetben a szöveges literálok illetve számok mellett a listát is. A lista maga egy termsorozat, melynek kezelésére számos beépített eljárás rendelkezésre áll. Adódik a lehetőség, hogy a visszalépéses kereső miatt gyakran használt rekurzív algoritmus segítségével, a listát feldolgozva határozzuk meg egy sávról, hogy befejezetlen-e:

```
countCredit( [], Credits, Credits ).
countCredit( [Subject|Subjects], C1, C2 ) :-
  subject( Subject, Credit, _, _, _),
  Sum is C1+Credit,
  countCredit( Subjects, Sum, C2).

incomplete( Group ) :-
  findall( Subject, accomplished(Subject,Group), Subjects ),
```

```
countCredit( Subjects, 0, AllCredit ),
group( Group, _, MinCredit ),
MinCredit > AllCredit.
```

Ha eljárásként tekintünk a `countCredit` szabályfejű partícióra, azt mondhatjuk, hogy az első két bemeneti paraméteréhez gyártja le rekurzívan a harmadik paraméterben reprezentált kimenetet. Az `incomplete` szabályt ezzel szemben tekinthetjük olyan predikátumnak, mely a paraméterben megadott sáv teljesítettségét határozza meg. A `findall` beépített predikátum és az `is` operátor – melynek segítségével aritmetikai kifejezéseket csempészhetünk a nyelvbe – értelmezése a rezolúciós logikán túl mutat. (Bár előbbi definiálható volna olyan predikátumként, mely az utolsó paraméter (mint lista) tartalmának ekvivalenciáját állítja, az első paraméterének alakjában megjelenő, és a második paraméterben megadott célt teljesítő termek halmazával. A gond csupán az, hogy a második paraméter (ami elsőrendű logikánkban term kéne legyen) egy tételformula.)

Mindez nem lenne teljes, ha magával a levezetési fával kapcsolatos manipulációkra nem lenne lehetőség. Természetes módon merül fel az igény erre minden olyan esetben, mikor többféleképp is előállítható a tétel negáltjának cáfolata (több lehetséges út vezet az üres klózig), ugyanis gyakran (elsősorban a procedurális megközelítés esetén) minden megoldásra kíváncsiak vagyunk. Ekkor ugyanis nem annak a ténynek a megállapítása a cél, hogy a tétel bizonyítást nyert, hanem a probléma megoldását az üres klózig vezető út egyes állomásain kapjuk. Ki kell kényszeríteni, hogy a feldolgozás ne szakadjon meg az első sikeres cáfolatnál. (A teljesség kedvéért persze nem feledkezhetünk meg arról, hogy amennyiben a feltett kérdésre több lehetséges válasz is létezik – a tétel negált segítségével többféleképp is előállhat az üres klóz –, a felhasználó kikényszerítheti, hogy az egyes válaszok megtalálása után ismét visszalépés következzen be. Ez persze kevésnek bizonyul, ha – mint a következő, a negáció kezelését mutató példában – a megoldáskeresés közben szeretnénk kihasználni ezt a tulajdonságot.) A megoldás egy olyan beépített predikátum, mely egyetlen szabállyal sem rezolválható (`fail`), tehát mindenképp visszalépést eredményez.

Ugyancsak szükséges a keresőfa (avagy a keresési tér) csonkítása a felesleges ágak elhagyásának érdekében. Erre szolgál a vágás művelete, mely visszalépés esetén fejti ki hatását. Vágásról visszalépve egész annak közvetlen őséig tekintjük sikertelennek a feldolgozást. Más szavakkal: azt a klózt tekintjük kudarcosnak, mely miatt a vágás a levezetésbe került. Ennek azonban az adott szituáció függvényében két megvalósítása lehet [2, 218. oldal]:

- Amennyiben a vágás olyan részfát távolított el, mely nem vezet új megoldáshoz, a prog-

ram deklaratív jelentése nem változik meg (*green cuts*). Ilyenek lehetnek például az „*if, else-if*” (ismét procedurális igényt kielégítő) szerkezet hatékony megvalósításai, ahol a későbbi feltételek is tartalmazzák a korábbi megszorításokat. A vágás segítségével az aktuálisan teljesülő feltétel utáni illesztéseket spórolhatjuk meg.

- Amennyiben a vágás elhagyása új megoldások megjelenéséhez vezet (*red cuts*), már jóval körültekintőbben kell alkalmazni ezt az eszközt, cserébe viszont nagyobb hatékonyságot nyerünk. Az iménti példánál maradva, egy „*if, else-if*” szerkezet későbbi feltételeiből a vágás helyes használatával, a korábban már vizsgált feltételek megismétlése spórolható meg.

A következő példában a *cut* (`!`) és `fail` használatát, negáció alkalmazása kapcsán tekinthetjük meg. (A példa csak szemléltetés erejű, hiszen a korábban emlegetett negációkezelés a Prologban rendelkezésre áll.)

```
accomplished( Subject ) :-
  exam( Subject, Result ),
  Result > 1.

notpassed( Subject ) :-
  accomplished( Subject ), !, fail.
notpassed( _ ).
```

Érdemes megfigyelni, hogy az `accomplished` predikátumot az iménti példák mindegyikében használtuk, ráadásul nem egyezett meg a paraméterek száma a két esetben. Ez annak köszönhető, hogy a Prolog megengedi a predikátumnevek túltöltését (avagy túlterhelését), és ekkor, mint két különböző predikátumként tekinthetünk rájuk. Ennél valamivel nagyobb problémát jelent a dokumentációs kényszer hiánya. A program meglepően tömör és kifejező, de az egyes szabályok magyarázatra szorulnak. Ritkán teljesül ugyanis, hogy egy predikátum tetszőleges argumentumaként változó szerepelhessen. Általában valamilyen célfeladat elvégzésének tekintetében beszélnünk kell bemeneti-, illetve kimenő paraméterekről. Bár csábító lenne feltenni a `countCredit(X,0,6)` kérdést – azaz, hogy mely tantárgyak teljesítése jelent 6 kreditet – de ezt valószínű a futás során használt verem betelésének jelzésével hálálná meg a redszer.

A *negation as finite failure* módszer megvalósítása a vágásról történő visszalépés tulajdonságainak kihasználásával érhető el. Ekkor – mint már említettük – a vágás őse lesz kudarcos, azaz a `notpassed` szabályfejű partíció további elemei nem kerülnek feldolgozásra. Ha még

a vágás előtti valamely predikátum válik illeszthetetlenné, a feldolgozás a partíció következő elemével folytatódik. Ez pedig nem más, mint egy olyan tény, melynek minden (jelen esetben egyetlen) változója független, így biztosan rezolválható. Tehát, ha a negálandó kifejezés nem igazolható, a szabály rezolválása sikeres lesz. Ellenkező esetben, ha túljutunk a vágás előtti literálokon, a vágást követő `fail` kényszerít ki visszalépést a vágásra, biztosítva ezzel a kudarcot az egész partíció tekintetében. Innen következik, hogy a negálandó kifejezés ebben a formában nem csupán egyetlen predikátum lehet, hanem:

$$\neg A \equiv \neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$$

$$\neg A \equiv \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$$

ahol B_1, B_2, $\ldots B_n$ a vágás előtt sorakozó literálok, esetünkben `accomplished`, valamint $\neg A$ nem más mint `notpassed`.

5. Induktív eljárások

Az induktív logikai programozás (ILP – *Inductive Logical Programming*) a már ismertetett deduktív stratégiák (következtetési sémák egymásra épülő használata) helyett, indukció (egyedi információk általánosítása) segítségével dolgozik. Ez az általánosítás biztosít lehetőséget a tanulás folyamatának algoritmizálására. Hiszen a mesterséges intelligenciától épp azt várjuk, hogy oly módon értelmezze a korábbi tapasztalatait, melyet felhasználhat egy ismertetlen szituációban, úgymond tanulva a múlt eseményeiből legyen képes döntést hozni.

Az ILP feladata a logikai programozás kiterjesztése oly módon, hogy képes legyen közelíteni vagy megjósolni egy ismeretlen predikátum értékeit. Ehhez két eszköz áll rendelkezésére: egyfelől a predikátum értéke bizonyos esetekben (korábbi tapasztalatok[1] nyomán) ismert, másfelől rendelkezésre áll a háttérismereteket (*background knowledge*) reprezentáló olyan – szintén ismert definíciójú – szabályok halmaza, mely elemeit az indukció során a közelítő predikátum értékének meghatározásához felhasználhatjuk.

A FOIL algoritmus

Az egyik lehetséges megoldást a FOIL algoritmus jelenti. Létre kell hozni egy olyan szabálysorozatot, ahol a szabályok feje a tanulni kívánt pozitív literál, és mely minden pozitív példára (avagy tényekre) alkalmazva kielégíti a szabályt, ugyanakkor egyetlen negatív példára sem teljesíti azt. Ehhez a következőket kell tenni:

1. Specializáljuk az üres törzsű szabályt (tényt) mindaddig, míg egyetlen negatív példára sem teljesül a törzse, de valahány pozitív példát lefed. Vegyük ezt hozzá a szabálysorozathoz.

2. Távolítsuk el a pozitív példák közül azokat, melyeket a megelőző szabálysorozat bármely tagja lefed. A hátralevő lépésekben már csak a maradékkal kell foglalkozni. Továbbra is megtartjuk viszont a negatív példák mindegyikét.

[1] Predikátum esetében kétféle tapasztalatunk lehet: pozitív-, illetve negatív példa. Ezt a szakirodalom fogalmi tanulásként emlegeti: [4], függvények esetén pedig felügyelt tanulásról beszélnénk.

3. Ismételjük az első lépést mindaddig, amíg a pozitív példák el nem fogynak.

Az eljárás végén egy definit klózsorozathoz jutunk. Az algoritmus kulcsa a szabály specializációja a háttértudás felhasználásával, mely az aktuális klóz törzsének literállal történő bővítését jelenti. A kiinduló üres törzsű szabályunk feje különböző változókat tartalmaz. A szabály törzsébe a specializáció során bekerülhetnek új változók, de legalább egynek már szerepelnie kell a szabályban. Az eljárás eredményessége azon múlik, hogy mennyire sikerül jól megjósolni a legkedvezőbb literált.

A példa kedvéért tegyük fel, hogy olyan predikátumot szeretnénk definiálni, mely megmutatja, hogy mely tantárgyakat érdemes felvenni a következő félévben: Favorizalt(x). Pozitív példa természetesen minden olyan tantárgy, mely már teljesítésre került, negatív példának tekinthető pedig mindaz, melynek teljesítésére már nem kerül sor (ilyen a teljesített sávba tartozó tantárgy, hiszen ezekkel könnyen túlléphetjük a képzésben megadott maximális teljesítendő kreditszámot, esetemben 330-at). Az első feladat, a háttértudást reprezentáló predikátumok megadása. Ezek legyenek most a következők:

- felvehető tantárgy (azaz minden előfeltétele teljesített): Felveheto(x)
- kötelező tantárgy: Kotelezo(x)
- teljesítése a mintatantervben erre a félévre ajánlott: Ajanlott(x)
- közvetlen előfeltétele valamely tárgynak: Elofeltetele(x, y)

A példa az áttekinthetőség kedvéért kissé mesterkélt ugyan, de könnyen életszerűvé tehető, hiszen számos objektív illetve szubjektív tulajdonság volna még definiálható.

Tegyük fel, hogy valamely félévben a következő táblázatban található tárgyak vonatkozásában az alábbi döntéseket hoztuk:

A Favorizalt(x) predikátum értékeit a $\oplus$ (mint pozitív példa) illetve $\ominus$ (mint negatív példa) szimbólumok jelölik. (5.1 táblázat) A táblázat utolsó oszlopának értékein az Elofeltetele(x, y) fogalom alkalmazás során kialakítható maximális lánc hosszát értjük, mely a kérdéses tárggyal indul. Például Elofeltetele(x, y) és Elofeltetele(y, z) esetén ez 2, ami praktikusan annyit jelent, ha ezen a vonalon szeretnénk a tárgyakat teljesíteni (például z-t), akkor ehhez az előfeltételek miatt legalább két félévre van szükség.

Az algoritmus indulólépése a Favorizalt$(x) \leftarrow$ klózból kiindulva (mely mint tény, kezdetben mindegyik esetre teljesül) addig szűkíteni a fogalmat, míg az egyetlen negatív példát sem fed

favorizált	tantárgy	felvehető	kötelező	ajánlott	előfeltétellánc
$\oplus$	T1	igen	igen	nem	2
$\oplus$	T2	igen	nem	igen	1
$\oplus$	T3	igen	nem	nem	3
$\oplus$	T4	igen	nem	igen	3
$\ominus$	T5	nem	igen	nem	1
$\ominus$	T6	nem	igen	igen	1
$\ominus$	T7	nem	nem	igen	3
$\ominus$	T8	igen	nem	nem	2
$\ominus$	T9	igen	nem	igen	0

5.1. táblázat. Pozitív és negatív minták

le. Mivel egyetlen teljesítetlen előfeltétellel rendelkező tárgy sem vehető fel, legyen az első fogalomspecializáció a következő:

$$\text{Felveheto}(x) \supset \text{Favorizalt}(x)$$

Ezzel T5, T6 és T7 negatív példákat sikeresen kizártuk, de az összes többi példát lefedi a szabály. T8-al is végzünk, ha a mintatanterv szerint nem ajánlott tárgyakat is kiszűrjük a következő képpen:

$$\text{Felveheto}(x) \wedge \text{Ajanlott}(x) \supset \text{Favorizalt}(x)$$

Az egyetlen negatív példa, mely esetén az implikációnk továbbra is igaz: T9, de mivel ez egyetlen későbbi tárgynak sem előfeltétele, a következő kiegészítéssel már egy olyan szabályhoz jutunk, mely valóban része lesz a tanult fogalomnak.

$$\text{Felveheto}(x) \wedge \text{Ajanlott}(x) \wedge \text{Elofeltetele}(x, y) \supset \text{Favorizalt}(x)$$

Ezen formula már T2 és T4 tárgyak esetében helyesen interpretálja a fogalmunkat, így ez a két pozitív példa a továbbiakban nem fog szerepet játszani. Megőrizzük azonban negatív példáink mindegyikét, így a táblázatunk 7 eleműre szűkül. Ekkor az algoritmust az 1. lépésnél folytatva (épp úgy mint az imént) szűkítsünk a felvehető tárgyak alapján, majd kötelezők szerint. Ekkor a következő implikációt kapjuk:

$$\text{Felveheto}(x) \wedge \text{Kotelezo}(x) \supset \text{Favorizalt}(x)$$

Ez az implikáció ismételten nem tesz eleget egyetlen negatív példának sem, teljesül azonban T1 tárgyra, így az algoritmus második iterációjával is végeztünk. Az utolsó lépésre maradt egyetlen pozitív példa: T3 tantárgy, és természetesen a táblázat utolsó öt sora. Ha a szokásos módon csak a felvehető tárgyakat tekintjük, azt vehetjük észre, hogy T3 az egyetlen, mely viszonylag sok következménnyel rendelkezik (alapozó tárgy lehet), így az előfeltétellánc alapján különböztethető meg a maradéktól.

$$\text{Felveheto}(x) \wedge \text{Elofeltetele}(x, y) \wedge \text{Elofeltetele}(y, z) \wedge \text{Elofeltetele}(z, v) \supset \text{Favorizalt}(x)$$

Most is sikerült kizárni minden negatív példát, és mivel a pozitívak is elfogytak, az algoritmus végére értünk. A keletkezett három implikáció immár a Prolog szintaktikáját követve a következő partícióval foglalható össze (az eredmény nem szükségszerűen Horn klózok sorozata, de jelen példánkban az átírás lehetséges):

```
favorizalt( X ) :- felveheto( X ), ajanlott( X ), elofeltetele( X, _ ).
favorizalt( X ) :- felveheto( X ), kotelezo( X ).
favorizalt( X ) :- felveheto( X ), elofeltetele( X, Y ),
  elofeltetele( Y, Z ), elofeltetele( Z, _ ).
```

Mint már azt említettük, természetesen az algoritmus kimenete nagyban függ attól, hogy milyen sorrendben válogattuk a literálokat a specializáció során. Bár lehetséges volna szélességi, illetve mélységi keresőalgoritmusok használata a döntéshozatalban, a FOIL algoritmus a literál hozzáadás hasznosságát előnymérték szerint osztályozza [4, 476. oldal]:

$$t \cdot \left(log_2 \frac{p_1}{p_1 + n_1} - log_2 \frac{p_0}{p_0 + n_0} \right)$$

ahol p_0 és n_0 a szabály által lefedett, míg p_1 és n_1 a választott literál hozzáadásával keletkező pozitív illetve negatív példák számát adja, t pedig a régi és új szabály által lefedett példák közti átfedések száma.

A fogalmi tanulás segítségével el tudtuk érni, hogy (valamely korábbi félévben meghozott döntéseink alapján) egy következtető gép osztályozni tudja az újonnan teljesíthetővé vált (vagy korábbról hátramaradt) tantárgyakat. A szabályzatszerűen – jól definiáltan – rendelkezésre álló teljesíthetőségi fogalmat (melyet korábban többféleképp is előállítottunk) egészítettük ki ez által egy olyan eszközzel, mely épp azon adatokat sorolja előre, melyre vélhetően valódi szükségünk van. Ugyanez az eljárás keresőprogramokban is kiválóan alkalmazható. A nagymennyiségű találat közül, a korábbi keresések során alkalmazott viselkedésünk szerint azon eredményeket kapjuk először, mely a kérdésünkre keresett legvalószínűbb válasz, így jó eséllyel lényegesen

hamarabb jutunk el a kielégítő eredményhez. Még csak azt sem kell elvárni, hogy a tanult fogalom feltétlenül illeszkedjen a későbbi szituációra, hisz ilyen esetben nem magát a döntés meghozatalát ruházzuk át az automatizmusra, hanem az eredményes döntéshozatal támogatását tűztük ki célul. Jól látható, hogy erre a feladatra akár egy ilyen végtelenül egyszerű példa is alkalmas lehet.

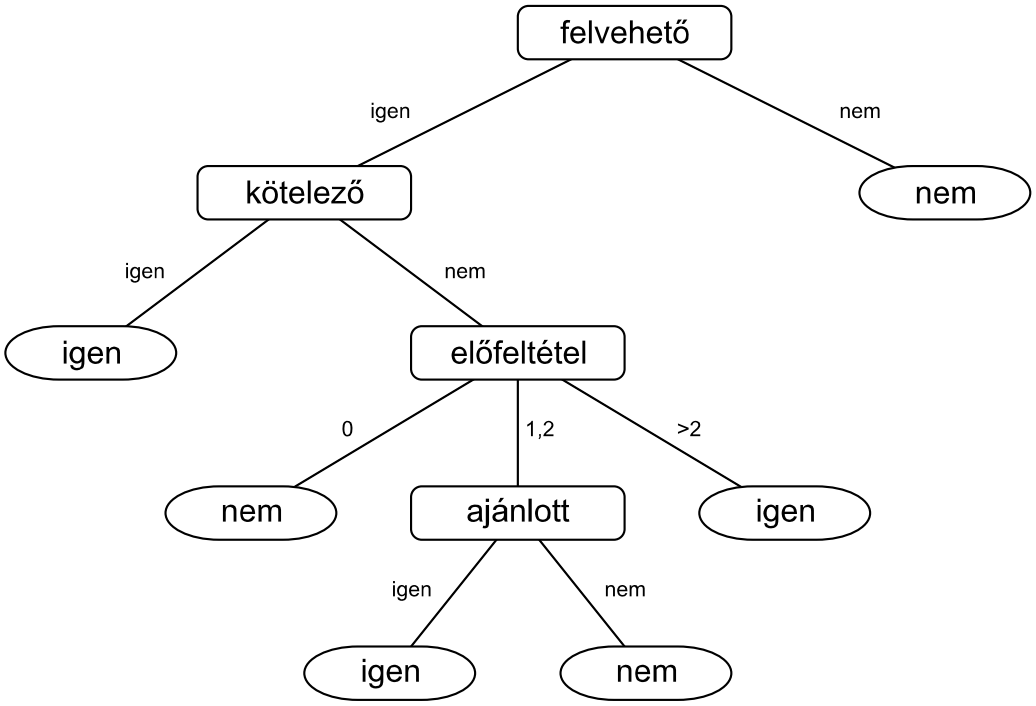
Döntési fák

A gépi tanulás területén születő eredmények felhasználását elsősorban a nagytömegű adatok feldolgozása kapcsán érhetjük tetten. A bankoknak illetve nagyvállalatoknak alapvető gazdasági érdeke fűződik az ügyfelek, illetve partnerek viselkedésének elemzéséhez (kezdve a vásárlási szokások tanulmányozásától a kockázatelemzésig). Ráadásul elegendő apparátussal rendelkeznek az adatgyűjtésre és statisztikák készítésére. A rendelkezésre álló nagytömegű információ értelmezésében kulcsszerepet játszik az adatbányászat (*data mining*) valamint az információban rejlő tudás feltárása (*knowledge discovery*). Ezen feladatok hatékony megoldásához a döntési fák indukcióján keresztül vezet az út.

A döntési fában minden út egy konjunkció-sorozat, ahol a nem levélcsomópontból kiinduló él címkéje a csomópont által reprezentált attribútum[2] értékére vonatkoztatott megszorítás. Ezen megszorítások az attribútum lehetséges diszkrét értékeit tartalmazó halmaz diszjunkt részhalmazait definiálják úgy, hogy azt teljesen lefedjék. Az út végén található levélelem a konjunkció-sorozat által definiált formulához rendelt logikai érték. Ha összegyűjtjük azon formulákat, melyek igaz értéket kaptak, a belőlük képzett diszjunkció segítségével fogalom definíciót kapuk. Ekkor a döntési fa tulajdonképpen egy diszjunktív normál formában adott kifejezést reprezentál. Bár mindez ítéletlogikai eszközökön alapul, az iménti fogalomdefiníció is reprezentálható döntési faként (5.1 ábra).

A döntési fa felépítésére több hatékony algoritmus létezik, például a C4.5 illetve elődje, az ID3 (*Iterative Dichotomiser 3*), melyek közül most az utóbbit ismertetjük. Az ID3 olyan rekurzív algoritmus, mely tetszőleges osztályozás szerint képes felépíteni a döntési fát a megadott példák alapján. (Jelen példánkban, a fogalmi tanulás során, – a logikai megközelítésnek köszönhetően – az osztályozás abban merült ki, hogy a levélelemekhez logikai értéket rendeltünk – ezáltal két osztályt létrehozva –, bár maga az ID3 tetszőlegesen sok osztály esetén is

[2] A döntési fát készítő eljárások bemenete általában egy adatbázistáblában kap helyet, melynek sorai a példák. Az attribútumok a tekintett adatbázistábla oszlopaiból adódnak, esetünkben a 5.1 táblázat fejléccímkéi (az elsőt kivéve) nevezik meg őket.



5.1. ábra. Döntési fa

használható.) A fa építésének rekurzívan ismétlődő lépései informálisan a következők (a precíz algoritmus, illetve változatai megtalálhatók: [4], illetve [5]):

- Adott a minták egy halmaza. Amennyiben ez egyetlen osztály elemeinek részhalmaza (példánkban mind pozitív vagy mind negatív minták) levélelemhez jutottunk, mely az osztály nevét viseli.
- Ha ez nem teljesül, de vannak még fel nem dolgozott attribútumok, illetve minták, választani kell egyet az iménti attribútumok közül (például: kötelező), melyből új csomópontot készítünk. A csomópontból kiinduló élek az attribútum értékek szerint címkézésre kerülnek. A csomópont gyermekeinek meghatározására az élek mentén folytatjuk az eljárást a minták élcímkéknek megfelelő részhalmazával.

Az eljárás erőssége az attribútum kiválasztás módjában rejlik (épp úgy, ahogy a FOIL algoritmusnál is kulcskérdés volt a választott literál). Az ID3 úgy választja meg az attribútumot, hogy ezzel a lehető legjobban csökkentse a mintahalmaz entrópiáját, mely a következőképp adódik [5, 292. és 293. oldal]:

$$\underbrace{\sum_{j=1}^{v} \frac{s_{1j}+\ldots+s_{mj}}{|S|}\left(\sum_{i=1}^{m} \frac{s_{ij}}{|S_j|}\log_2\left(\frac{s_{ij}}{|S_j|}\right)\right)}_{\text{az attribútum szerint várható információ}} - \underbrace{\sum_{i=1}^{m} \frac{s_i}{|S|}\log_2\left(\frac{s_i}{|S|}\right)}_{\text{várható információ}}$$

ahol az S-el jelölt mintahalmaz v darab diszjunkt S_i halmazra bontható a tekintett attribútum szerint, ahol S_i halmazban a j-edik osztályba tartozó elemek száma s_{ij}, illetve S halmazban s_j, valamint összesen m darab osztályozás létezik. Ekkor a képlet az entrópia várható csökkenését adja.

A cél az, hogy a lehető legegyszerűbb döntési fát gyártsa le az algoritmus. Az ID3 algoritmus (habár létezik javított változata) hála az attribútum kiválasztási módszerének, mindezt elég jól közelíti. Természetesen minél egyszerűbb maga a fa, az általa végzett osztályozás annál gyorsabban végezhető el.

6. Összefoglalás

Az első lépés, melyet mintaalkalmazásunk implementálása felé tettünk, a támasztott követelmények felsorolása volt, melynek kapcsán öt alapvető kérdés megválaszolását tűztük ki célul. A különféle technikák fokozatos egymásra építésével egytől-egyig minden kérdésre választ találtunk. Ehhez a legtöbb esetben a matematikai logikát hívtuk segítségül, és végezetül a deklaratív megközelítés, a tudásreprezentáció és rezolúciós logika alkalmazásán át eljutottunk egészen az induktív logikai programozás néhány alapeszközének bemutatásához és egyben az utolsó kérdéseink megválaszolásához.

Az egyes lépések folyamán mindvégig sikerült megadni a mintaalkalmazás által támasztott kritériumokat kielégítő kódrészleteket. Láthattuk, hogy ezek az alkalmazott technikában sokkal tömörebben adhatók meg, és főképp a feladatmegfogalmazásra koncentrálhatnak, így érzékelni tudtuk a kifejezőerőben megmutatkozó különbségeket. Mindeközben lehetőségünk nyílt betekinteni a szakértői rendszerek, valamint a logikai programozás szemléletmódjába.

Az induktív technikák olyan – a mesterséges intelligencia területéhez tartozó – kérdések megválaszolását is lehetővé tették, melyek hagyományos eszközökkel történő megoldása jóval komplikáltabb, illetve egyenesen megvalósíthatatlan. Érzékelhettük, hogy milyen esetekben érdemes az induktív eszközökhöz nyúlni. Tehát megállapítható, hogy a kitűzött célt elértük.

Irodalomjegyzék

[1] BATES, C.: *XML Elmélet és gyakorlat*. Budapest: Panem 2004.

[2] DAS, S. K.: *Deductive Databases and Logic Programming*. Cambridge: Addison-Wesley 1992.

[3] DŽEROSKI, S. & LAVRAČ, N. (szerk.): *Relational Data Mining*. Berlin Heidelberg: Springer 2001.

[4] FUTÓ, I. (szerk.): *Mesterséges Intelligencia*. Budapest: Aula 1999.

[5] HAN, J. & KAMBER, M.: *Adatbányászat*. Budapest: Panem 2004.

[6] NILSON, U. & MAŁUSZYŃSKI, J. (1998): „*Logic, programming and Prolog (2ED)*". Online: http://www.ida.liu.se/~ulfni/lpp

[7] PÁSZTORNÉ VARGA, K. & VÁRTERÉSZ, M.: *A matamatikai logika alkalmazásszemléletű tárgyalása*. Budapest: Panem 2003.

[8] SZEREDI, P., LUKÁCSY, G. & BENKŐ, T.: *A szemantikus világháló elmélete és gyakorlata*. Budapest: Typotex 2005.

[9] RILEY, G.: (June 15th 2006): „*CLIPS Reference Manual Volume I Basic Programming Guide*". Online: http://www.ghg.net/clips/download/documentation/bpg.pdf

Mellékletek

A következő mellékletek a korábbi fejezetekben található kódrészletek értelmezéséhez adnak segítséget.

Az XML reprezetnáció DTD-je

```
<!ENTITY        % number        'CDATA'>
<!ELEMENT       group           (subject+)>
<!ATTLIST       group
  description   CDATA           #REQUIRED
  required      %number;        #REQUIRED>
<!ELEMENT       required        EMPTY>
<!ATTLIST       required
  id            NMTOKEN         #REQUIRED>
<!ELEMENT       subject         (required*)>
<!ATTLIST       subject
  id            ID              #REQUIRED
  credit        NMTOKEN         #REQUIRED
  labour        %number;        '0'
  seminar       %number;        '0'
  lesson        %number;        '0'
  name          CDATA           #REQUIRED>
<!ELEMENT       subjects        (group+, index?)>
<!ELEMENT       index           (term*)>
<!ELEMENT       term            (exam*)>
<!ATTLIST       term
  date          CDATA           #REQUIRED>
<!ELEMENT       exam            EMPTY>
<!ATTLIST       exam
  subject       NMTOKEN         #REQUIRED
  result        (1|2|3|4|5)     '1'>
```

Az átlagszámítás teljes XSL kódja

```
<xsl:template name="_average">
    <xsl:param name="nodeset"/>
    <xsl:param name="index"/>
    <xsl:param name="credits"/>
    <xsl:choose>
        <xsl:when test="count($nodeset/exam) &lt; $index">
            <xsl:variable
                name="all"
                select="sum(id($nodeset/exam/@subject)/@credit)"/>
            <xsl:value-of select="round(($credits div $all)*100) div 100"/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:call-template name="_average">
                <xsl:with-param name="nodeset" select="$nodeset"/>
                <xsl:with-param name="index" select="$index + 1"/>
                <xsl:with-param name="credits"
                    select="$credits +
                    id($nodeset/exam[position()=$index]/@subject)/@credit *
                    $nodeset/exam[position()=$index]/@result"/>
            </xsl:call-template>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Interaktivitás a CLIPS-ben

```
(defrule print_groups_in_order "Sorszámozva írja ki a sávokat"
    ?c  <- ( counter ?num )
    ?gr <- ( group ?gnev ?gkrd $?others )
    ( group_done ?gnev ?gcount )
=>
    ( assert ( ugroup (+ ?num 1 ) ?gnev (- ?gkrd ?gcount ) $?others ) )
    ( assert ( counter (+ ?num 1 ) ) )
    ( printout t (+ ?num 1 ) ".  " ?gnev " [" ?gcount "/" ?gkrd "] " crlf)
    ( retract ?gr )
    ( retract ?c )
)

(defrule print_groups_in_order_end "A sorszám beolvasása"
    ?c  <- ( counter ?num )
    ( forall (group $?others) (nonsense) )
=>
    ( printout t "Valassza ki a legerdekesebb savot:  " )
    ( bind ?sorszam (read) )
    ( assert ( rowin ?sorszam ) )
)

(defrule check_input
    ( rowin ?num )
    ( counter ?count )
    ( test ( and ( numberp ?num ) ( > ?num 0 ) ( <= ?num ?count ) ) )
    ( ugroup ?num ?nev $?info )
    ( not ( offer $?any ) )
=>
    ( printout t "Kiválasztva: " ?nev crlf)
    ( assert ( offer 0 ) )
)
```

Felvenni érdemes tárgyak listája Prolog-ban

```
accomplished( Subject ) :-
  exam( Subject, Result ),
  Result > 1.
accomplished( Subject, Group ) :-
  accomplished( Subject ),
  subject( Subject, _, Group, _, _).

notpassed( Subject ) :-
  exam( Subject, Result ),
  Result > 1,
  !,
  fail.
notpassed( Subject ).

passed( [] ).
passed( [Subject|Subjects] ) :-
  accomplished( Subject ),
  passed( Subjects ).

countCredit( [], Credits, Credits ).
countCredit( [Subject|Subjects], C1, C2 ) :-
  subject( Subject, Credit, _, _, _),
  Sum is C1+Credit,
  countCredit( Subjects, Sum, C2).

incomplete( Group ) :-
  findall( Subject, accomplished(Subject,Group), Subjects ),
  countCredit( Subjects, 0, AllCredit ),
  group( Group, _, MinCredit ),
  MinCredit > AllCredit.

offer( Subject ) :-
  subject( Subject, _, Group, Name, Requires ),
  passed( Requires ),
  notpassed( Subject ),
  incomplete( Group ),
  write( Name ), nl.

start() :- offer(_), fail.
```