DIPLOMAMUNKA

Bóta András

Debrecen

2009

Debreceni Egyetem

Informatikai Kar

Korlát-logikai programozás és alkalmazásai

Témavezető:

Dr. Aszalós László

Egyetemi adjunktus

Készítette:

Bóta András

PTI MSc MI

Debrecen

2009

Tartalomjegyzék

Köszönetnyilvánítás

Szeretném megköszönni konzulensemnek, Dr. Aszalós László Tanár Úrnak, hogy mindvégig támogatott és ösztönzött a munka magas színvonalú elkészítésében.
Szeretném megköszönni a Debreceni Egyetem Informatikai Kar tanárainak tudásuk önzetlen átadását, szeretném kiemelni Dr. Várterész Magda munkáját.
Köszönettel tartozok továbbá szüleimnek és barátaimnak a támogatásért.

0. Bevezető

A korlát-logikai programozás a hagyományos logikai programozás kibővítése a korlát-kielégítési problémák témaköréből vett elemekkel. A deklaratív jellege miatt a problémák megfogalmazására teszi a hangsúlyt, a kapott kód pedig elegáns és tömör. Ehhez egy erős következtetési mechanizmus társul, ami azonban a módszer alkalmazását problémák egy bizonyos részhalmazára szűkíti.

Ezek a kényszer kielégítési problémák, melyekben a feladatot változók, és a hozzájuk kapcsolt korlátok formájában adjuk meg, és célunk egy olyan értékkombináció megtalálása, mely az összes megfogalmazott megszorítást kielégíti.

A dolgozat célja a korlát-logikai programozás ismertetése, valamint alkalmazásának szemléltetése egy gyakorlati példán. Ez a példa a népszerű „kukac” játék egy változata, amiben a játékosnak egy játéktérben kell mozognia, miközben bizonyos mezőkön áthaladva az irányított „kukac” mérete megnő.

A program fejlesztése során a feladat több esetben megváltozott, bővült, így végül négy különböző feladathoz négy részben különböző megoldó algoritmus készült. Ezek a megoldások, ahogy a feladatok is, egymásra épülnek, ismertetésük sorban történik a legegyszerűbbtől a legbonyolultabb felé.

A dolgozat felépítése a következő: az első fejezet a logikai programozással foglalkozik, szerepet kap benne a módszer története, kapcsolata a többi programozási nyelvvel. Szó lesz a logikai programok felépítéséről, valamint ezek elméleti alapjairól. A fejezetet egy gyakorlati példa zárja.

A második fejezet a kényszer kielégítési problémákkal foglalkozik, ezek definíciója után bővebben is ismerteti a problémakört, majd megad egy megoldási módszert is.

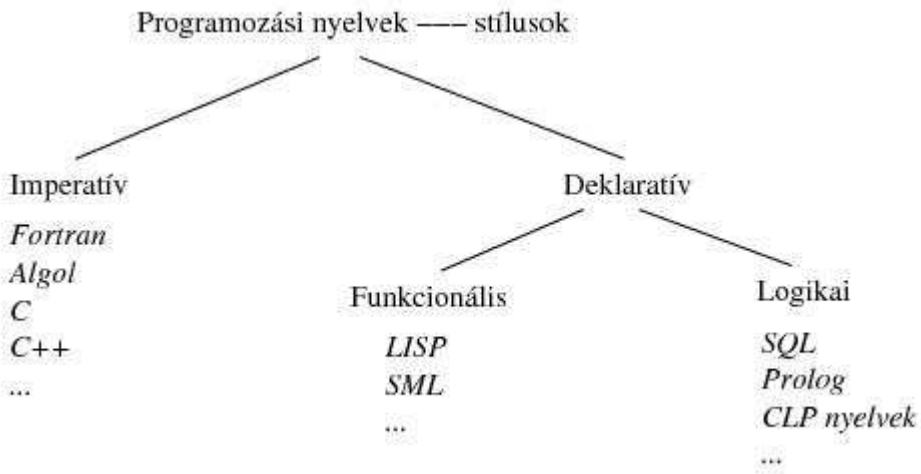
A harmadik fejezet a korlát-logikai programozással foglalkozik, ismerteti ezek felépítését, az alkalmazható alaphalmazokat, valamint a következtetési mechanizmus egy részét. Végül pedig megadja egy konkrét Prolog nyelv korlát-logikai könyvtárát is.

A negyedik rész a feladat rövid leírása után ennek négy fő változatát ismerteti, az ötödik pedig az ezek megoldására szolgáló algoritmusokat, és végül a kiíró modult.

1. A logikai programozás és a Prolog

1.1. Programozási nyelvek

A programozási nyelveket alapvetően két csoportba sorolhatjuk. A legtöbb nyelv az imperatív nyelvek családjába tartozik. Ezek egyik jellemzője, hogy a hangsúlyt az elvégzendő feladat megoldásának pontos leírására fekteti. Parancsok, függvények segítségével határozzuk meg a feladat elvégzésének menetét. Az imperatív nyelvek tipikus példája az assembly, de ilyen a Pascal, C, C++, Java vagy akár a Perl, PHP és a Python nyelv is.
Ezzel szemben a deklaratív nyelvek esetében a probléma megfogalmazásán van a hangsúly. Tényállításokat, szabályokat fogalmazunk meg, és ezek segítségével következtetünk a megoldásra. Deklaratív nyelv például a Prolog, vagy a LISP.



1.1.1. ábra: A Programozási nyelvek osztályozása.[1]

Míg egy imperatív szemléletű program az algoritmusra, a feladat megoldásának módjára helyezi a hangsúlyt, egy deklaratív program magát a feladatot fogalmazza meg.
Ebből következik az, hogy egy imperatív program esetén viszonylag pontosan át tudjuk látni, hogy a program végrehajtása milyen lépésekből, milyen állapotváltozásokból áll. Ezzel szemben egy deklaratív program esetén az „elemi” lépések sokkal összetettebbek: pl. egy egyenletrendszer megoldása, egy következtetési lépés elvégzése stb., ezért a végrehajtás pontos menetét esetleg nem tudjuk követni. De ez nem is szükséges, hiszen a cél a feladat megfogalmazása volt. A végrehajtás menetét rábízhatjuk a nyelv fordítóprogramjára.

A másik fontos különbség az imperatív és a deklaratív nyelvek között a változó-fogalomban mutatkozik meg. Az imperatív nyelvekben a változó egy adott memóriahelyen tárolt aktuális értéket jelent, az algoritmus írása során pedig a változónak ismételten új és új értéket adunk.
Ezzel szemben a deklaratív programozási nyelvek változói a matematika változó-fogalmának felelnek meg: egyetlen konkrét, bár a programozás idején még ismeretlen értéket jelölnek.
A deklaratív nyelvekben ezért nem létezik az imperatív nyelvekben megszokott ismételt értékadás (X = X + 1), hiszen nem létezik olyan x szám, amelyre ez az egyenlet teljesülne. A deklaratív nyelvek az egyszeres értékadású nyelvek sorába tartoznak. Az egyszeres értékadás tulajdonságát a párhuzamos programozás kutatói vezették be, mivel úgy találták, hogy az ilyen tulajdonságú nyelvek párhuzamos végrehajtása sokkal könnyebben megvalósítható, mint a hagyományos, imperatív nyelvek esetén.
Annak eldöntése, hogy imperatív, vagy deklaratív programozási nyelvet használjunk egy konkrét feladat megoldására, nem mindig egyértelmű. Bizonyos típusú problémák jobban illeszkednek a deklaratív, míg mások az imperatív szemlélethez.
Érdemes még megemlíteni, hogy mivel a számítógépek gépi nyelve szinte kivétel nélkül imperatív, ezért a deklaratív nyelvek implementációt gyakran imperatív nyelven írják.
A deklaratív programozási nyelvek általában valamilyen matematikai formalizmusra épülnek. A függvényfogalomra építő közlésmódot funkcionális programozásnak, míg a reláció fogalomra épülőket logikai programozásnak nevezzük.
Az első funkcionális nyelv a LISP volt, amelyet az 1960-as évek elején alkottak meg. Ezt később más nyelv követte, köztük az SML nyelv. A legegyszerűbb logikai nyelvnek a relációs adatbázisok lekérdező nyelve az SQL tekinthető. A „valódi” logikai nyelvek között a Prolog nyelv a legelterjedtebb, de újabban egyre nagyobb jelentőséggel bírnak a Prolog korlát alapú kiterjesztései, a CLP rendszerek (Constraint Logical Programming).
Míg a funkcionális nyelvek a matematikai függvényfogalomra, addig a logikai nyelvek a reláció fogalmára építenek. A legismertebb logikai programozási nyelv a Prolog.[1]

1.2. A logikai programozás története

A logikai programozás alapgondolatai, alapelvei Robert Kowalskitól származnak. Ezeket Alain Colmerauer csoportja vitte át a gyakorlatba a Marseille-i egyetemen 1972-ben, így

született meg az első Prolog megvalósítás. A Prolog első alkalmazása is itt készült, és a természetes nyelvek fordításával volt kapcsolatos.

A másik nagy központ Edinburgh lett, itt dolgozott Kowalski is, valamint itt kezdett el a logikai programozás témájával foglalkozni David Warren is, aki később nagymértékben hozzájárult a Prolog fejlődéséhez.

Itthon Németi István vezetésével foglalkozott egy csoport logikai programozással, melynek alapján Szeredi Péter elkészített egy saját Prolog megvalósítást. Ennek felhasználásával több tucat kísérleti jellegű Prolog alkalmazás készült Magyarországon.

A Prolog hatékony megvalósítási módszereinek kidolgozása David Warren nevéhez fűződik, aki 1977-ben elkészítette a nyelv első fordítóprogramját.

Amerikában viszont nagyon barátságtalanul viselkedtek a logikai programozás iránt, így a Prolog nyelv fejlesztésében főleg európai kutatók vettek részt, ezen belül is Magyarország volt a legnagyobb alkalmazó. 1980-ban tartották az első nagyszabású logikai programozással foglalkozó konferenciát Debrecenben.

1981-ben a japán kormány egy nagyratörő számítástechnikai fejlesztési munkát indított el, az ún. „ötödik generációs számítógéprendszerek” projektet, amelynek alapjául a logikai programozást választották. Ez nagy lökést adott a terület kutató-fejlesztő munkáinak, és megjelentek a kereskedelmi Prolog megvalósítások is. Funkcionális szempontból, a 80-as években a Prolog nyelv alapdefiníciójához eszközrendszer készült. Már az alapdefiníció is tartalmazta azonban, a DCG (Definite Clause Grammars) környezetfüggetlen attribútum-nyelvtan elemzőjét.

Bár a japán ötödik generációs projektben nem sikerült elérni a túlzottan ambiciózus célokat, és ez a 90-es évek elején a logikai programozás presztízsét is némileg megtépázta, mára a Prolog nyelv érett és világszerte elfogadott nyelvvé vált, 1995-ben megjelent a Prolog ISO szabványa is, és egyre több ipari alkalmazással találkozhatunk.

Szélesebb körű elterjedésének legfőbb gátja, hogy más jellegű gondolkodást igényel a programozóktól, mint a hagyományos nyelvek. Éppen ezért egy újfajta rálátást is ad a problémára, és mára a legtöbb egyetemen az informatikusképzésben tananyag a logikai programozás.

A 80-as évek végén jelent meg a logikai programozás egy új irányzata, a CLP. Ennek alapgondolata az, hogy egy szűkebb problémakörre koncentrálva egy sokkal erősebb következtetési mechanizmust lehet adni. Ez a következtető-gép különböző tudomány-

területekről jöhet, mint például a mesterséges intelligencia vagy az operációkutatás. A CLP egy közös keretet ad arra, hogy a problémánkat deklaratív módon írhassuk le, a megoldásra viszont a megfelelő tudomány-területekről alkalmazhatunk módszereket.[1]

1.3. A logikai programozás

A logikai programozás alapgondolata, hogy egy, a matematikai logikán alapuló nyelvet használjunk programozási nyelvként, végrehajtási módszerként pedig logikai következtetési és tételbizonyítási módszereket alkalmazzunk. Ez utóbbi már nem a programozó, hanem az adott logikai nyelvet megvalósító rendszer feladata.

A Prolog nyelv esetében az elsőrendű logika nyelvét az ún. Horn klózokra szűkítjük le, és a programok futásához egy nagyon egyszerű tételbizonyítási módszert használunk, az ún. SLD rezolúciót.

Ezek miatt az egyszerűsítések miatt a tételbizonyítási folyamat értelmezető úgy is, mint a logikai értéket adó eljáráshívások végrehajtása, ahol a paraméterek átadása mintaillesztésen alapul, és az eljárások meghiúsulása ún. visszalépést eredményez. Ezt a fajta értelmezést hívjuk eljárásos értelmezésnek.

A Horn klózok, mint eljárások több különleges vonással rendelkeznek. Az eljáráshívások mindig egy logikai értéket adnak vissza (tehát valójában Boole-értékű függvények). Az igazi értékekkel visszatérő eljárást sikeresnek, a hamissal visszatérőt meghiúsulónak nevezzük. Ha egy eljárás meghiúsul, akkor az adott eljárástörzs további eljárásait nem hajtjuk végre, ehelyett visszalépünk a legutoljára sikeresen lefutott eljáráshoz, és megpróbáljuk azt más módon (más változó-behelyettesítéssel) sikeresen lefuttatni. Ennek sikere esetén az előremenő végrehajtás folytatódik, meghiúsulás esetén újabb visszalépés történik. Ezt hívjuk visszalépéses keresésnek.

Az eljárások paraméter-átvétele kétirányú mintaillesztéssel (egyesítéssel) történik. Ennek folyamán a bemenő és kimenő paraméterek nincsenek megkülönböztetve, azaz ugyanaz az eljárás többféleképpen is használható.

A logikai programozás egyik új irányzata a korlát logikai programozás (CLP), amelynek egyes megvalósításai az újabb Prolog implementációkban könyvtárak formájában hozzáférhetők.[1]

1.4. A Horn klózok és a rezolúció

A Horn klózok

$$a \leftarrow (b_1 \wedge b_2 \wedge \ldots \wedge b_n)$$

alakú implikációk, ahol a, b_1, .., b_n elemi állítások és az összes előforduló változót univerzális kvantorral lekötöttnek tekintjük.
Egy klózt szabálynak nevezzük, ha n > 0, tehát ha az implikáció jobboldala nem üres, ellenkező esetben, ha n=0, tényállításról beszélünk, és a klózt egységesen igaznak tekintjük.
Végül megengedjük, hogy egy Horn klóz baloldala üres, azaz azonosan hamis legyen, ezt célsorozatnak hívjuk. Ez tehát egy negatív állítás, a keresett információ meglétét tagadja. A Prolog végrehajtási mechanizmusát képező tételbizonyítási rendszer ebből a negatív állításból kiindulva, következtetési lépések felhasználásával próbál meg ellentmondásra jutni.
Minden egyes következtetési lépés eredménye egy újabb célsorozat. Az így előálló célsorozatokat rezolvensnek hívjuk. A kezdő rezolvens az eredeti célsorozat. A következtés egy lépése a következő tevékenységekből áll:

- Kiválasztunk a rezolvensből egy literált.
- Keresünk egy olyan klózt, melynek feje és a kiválasztott literál egyesíthető (változó-behelyettesítésekkel azonos alakra hozható)
- Elvégezzük a szükséges változó-behelyettesítéseket, és a kiválasztott literál helyére a klóz törzsét írjuk.

Így megkapjuk az új célsorozatot. A Prolog nyelvben a kiválasztási függvény mindig az első literált választja, és az egyesíthető klózokat is a felírás sorrendjében veszi figyelembe.
A rezolúciós lépéseket addig ismételjük, amíg a célsorozat literáljai el nem fogynak, ekkor a bizonyítás sikeres a megfelelő változó-behelyettesítésekkel. Ha több rezolúciós lépés nem lehetséges, de a célsorozatban még mindig szerepelnek literálok, akkor a Prolog végrehajtási algoritmus visszamegy az előző rezolúciós lépéshez és megpróbál egy másik rezolúciós lépést elvégezni. Ez a visszalépéses keresés.[3]

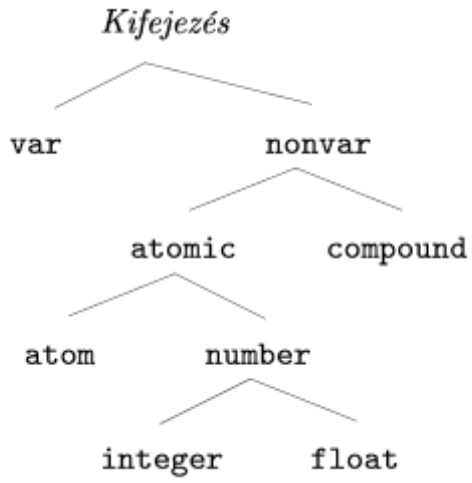
1.5. A Prolog nyelv szintaxisa

A logikai nyelvek, és így a Prolog is, a predikátumok fogalmára építenek. Minden Prolog program egy vagy több predikátumból áll, ezek egy relációt határoznak meg az argumentumukban lévő kifejezések között. Egy predikátum egy vagy több azonos funktorú állításból épül fel, ezeket klózoknak is nevezzük, és együtt definiálják az adott predikátumot. Egy klóz lehet tényállítás, vagy szabály. A tényállítások feltétel nélkül igaz állítások. Egy tényállítás csak egy fejből áll, míg a szabályok fej- és törzs-részből állnak, amelyeket a ':-' jel választ el egymástól. Egy törzs célok vesszővel ellátott sorozata, ahol a vessző „és" kapcsolatot jelent a célok között. A célokat, bizonyos feltételek között pontosvesszővel is elválaszthatjuk ez „vagy" kapcsolatot jelent. Mind a szabályok fej részei, mind a törzs céljai tetszőleges Prolog kifejezések lehetnek. Mindkét esetben a klózt ponttal kell lezárni, ami után legalább egy nem látható karakternek kell következnie.
Tényállítás esetén is beszélhetünk a klóz törzséről, amelyet üresnek tekintünk. Bizonyos helyzetekben viszont a tényállítás törzsének a true azonosan igaz beépített eljárást tekintjük.

⟨Prolog program⟩ ::= ⟨predikátum⟩ ...
⟨predikátum⟩ ::= ⟨klóz⟩ ...
⟨klóz⟩ ::= ⟨tényállítás⟩.␣ |
⟨szabály⟩.␣
⟨tényállítás⟩ ::= ⟨fej⟩
⟨szabály⟩ ::= ⟨fej⟩ :- ⟨törzs⟩
⟨törzs⟩ ::= ⟨cél⟩, ...
⟨cél⟩ ::= ⟨kifejezés⟩
⟨fej⟩ ::= ⟨kifejezés⟩

1.5.1. ábra: A Prolog nyelv közelítő szintaxisa.[1]

A Prolog kifejezések osztályozása során egy meglehetősen hierarchikus szerkezetet kapunk. Egy Prolog kifejezés lehet változó, összetett kifejezés, vagy konstans (atomic). A Prolog változók mindig nagybetűvel kezdődnek. Egy konstans lehet névkonstans (atom), vagy számkonstans (number). Egy számkonstans lehet egész (integer), vagy lebegőpontos szám (float).



1.5.2. ábra: A Prolog kifejezések osztályozása.[1]

Egy összetett kifejezés egy struktúranévből és egy zárójelbe tett argumentumlistából áll. Az argumentumlista egy vagy több, egymástól vesszővel ellátott argumentumból épül fel. A struktúranév egy névkonstans, míg az összes argumentum, rekurzív módon, egy tetszőleges Prolog kifejezés lehet.

Egy összetett kifejezés funktorán a struktúranév/argumentumok száma kifejezést értjük. Az argumentumok számát aritásnak is nevezzük. A konstansokat tekinthetjük 0 argumentumú kifejezésnek. Klózok funktora nem más, mint a klóz fejének, mint Prolog kifejezésnek a funktora. Amennyiben egy névkonstans nagybetűvel kezdődik, aposztrófok közé kell tenni az egész konstans kifejezést azért, hogy meg lehessen különböztetni egy Prolog változótól.[1]

1.6. Prolog példa

Ezek után nézzük meg egy Prolog program felépítését!

```
start :- write('Írja be az elemezni kívánt számot!'),
      nl, readln([A|_]), fakt(A, F), write('A szám faktoriálisa:'),
      nl, write(F), nl, nl, alakit(F, L), length(L, C),
      write('A számjegyek száma: '), write(C), nl,
      keres(L, B), write('Az utolsó nullától különböző számjegy: '),
      write(B), nl.

alakit(F, L) :- number_chars(F, L0), reverse(L0, L).

fakt(0, F) :- !, F = 1.
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N * F1.

keres([A|_], B) :- A \= '0', !, B = A.
keres([A|B], C) :- A = '0', keres(B, C).
```

1.6.1. Példa: Prolog program, mely egy szám faktoriálisának utolsó számjegyét számolja ki.

A program predikátumokból épül fel, ezek a start, az alakít, a fakt és a keres. Az első kivételével mindegyiknek két argumentuma van, ezen kívül mindegyik szabály. A program tényállításokat nem tartalmaz. A program egészekkel és listákkal dolgozik, ugyanakkor a kiíratáshoz használ stringeket is. Az alakit eljárás egy beépített eljárást használ arra, hogy az első argumentumában meghatározott számot a második argumentumában meghatározott listává alakítsa. A megfogalmazásnak és a Prolog végrehajtás módjának köszönhetően az argumentumok bármelyike lehet bemenő paraméter. A Prolog típustalan nyelv, így elméletileg a predikátumok paramétereiként bármilyen Prolog kifejezést megadhatnánk, ugyanakkor a legtöbb predikátum csak a programozó által tervezett típusú változókra működik. A Prolog rengeteg beépített eljárással segíti a programozó munkáját. Egy ilyen eljárás a write (és változatai) melynek segítségével kiíratást végezhetünk, az nl eljárás pedig soremelést végez. Az alakít predikátumban listakezelő és konvertáló beépített eljárásokkal találkozhatunk. Aritmetikai műveleteket végző eljárásokat is találhatunk. A Prolog különböző értékadási és egyenlőségi relációkat értelmez a lehetséges esetekre. A Prolog matematikai logikai nyelv, így nem tartalmaz iterációt. Erre a célra a programozó rekurziót használhat.

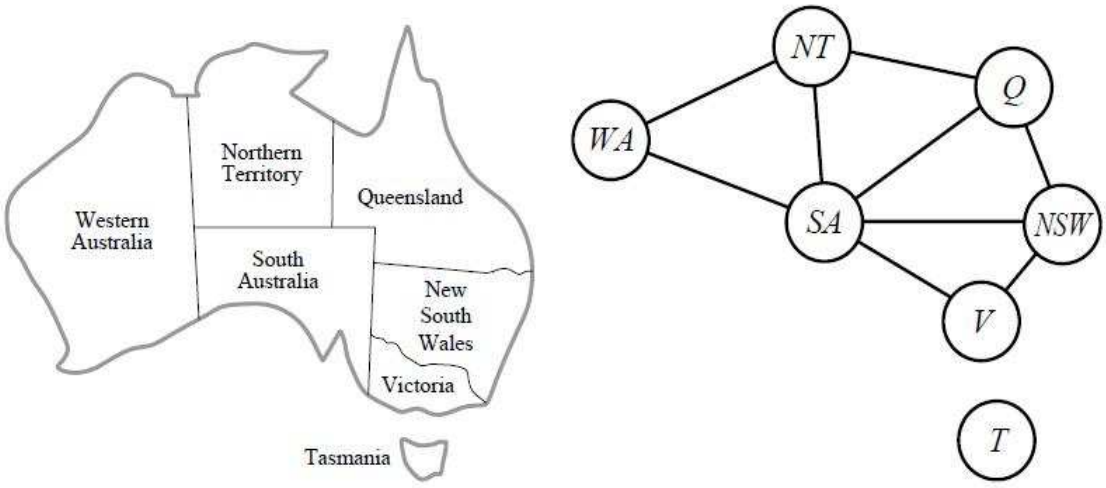
2. Kényszer kielégítési problémák

2.1. Definíció

A kényszer kielégítési problémák formális definícióját x_1, …, x_n változók és c_1, …, c_m kényszerek halmazaival adhatjuk meg. Minden egyes x_i változó esetén adott a lehetséges értékek egy nem üres D_i tartománya. Minden egyes C_i kényszer a változók valamely részhalmazára vonatkozik, és meghatározza a részhalmaz megengedett értékkombinációit. A megoldás során néhány, vagy mindegyik változóhoz értékeket rendelünk hozzá. Egy hozzárendelést konzisztensnek nevezünk, ha egyetlen korlátot sem sért meg. Teljes az a hozzárendelés, amelyben mindegyik változó szerepel, és egy teljes hozzárendelés a kényszer kielégítési problémának megoldása, ha mindegyik kényszert kielégíti. Néhány kényszer kielégítési probléma azt is igényli, hogy a megoldás egy célfüggvényt maximalizáljon.[6]

2.2. Kényszer kielégítési problémák

A kényszer kielégítési problémák egyik tipikus példája a térképszínezési probléma. Legyen adott egy elkülöníthető területeket (országokat vagy államokat) ábrázoló térkép, a feladatunk pedig az, hogy minden egyes részt színezzünk ki úgy, hogy a szomszédos részeknek ne legyen azonos színe. A probléma fentiek szerinti megfogalmazásához szükségünk lesz változókra, ezek legyenek az államok neveinek rövidítései. A térkép mezői a három szín egyikét vehetik fel, tehát a változók ebből a három értéktartományból kaphatnak értéket. A kényszerek segítségével pedig megfogalmazhatjuk azt a kikötésünket, hogy a szomszédos területek csak különböző színűek lehessenek.
Gyakran hasznos lehet, ha felrajzoljuk a probléma kényszergráfját. A gráf csomópontjai a probléma változóinak, élei pedig a korlátoknak felelnek meg.



2.2.1. ábra: Baloldalon: Ausztrália államai. Ezen térkép kiszínezése felfogható kényszer kielégítési problémaként. Jobboldalon: A probléma kényszergráfja.[6]

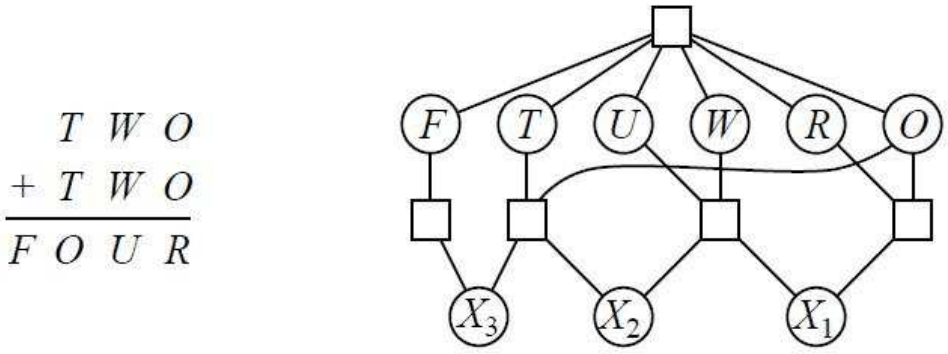
A kényszer kielégítési problémák legegyszerűbb esetében a változók diszkrétek és véges tartományúak. A térképszínezési problémák ide tartoznak, vagy akár az n-királynő problémát is felfoghatjuk véges tartományú kényszer kielégítési problémaként. A véges tartományú esetbe tartoznak a Boole-problémák is, ahol a változók értéke igaz vagy hamis lehet. A Boole kényszer kielégítési problémák speciális esetként tartalmaznak néhány NP-teljes problémát, például a 3SAT-ot.

A diszkrét változók lehetnek végtelen tartományúak is: például ilyen az egész számok halmaza, vagy a füzérek halmaza. Végtelen tartományok esetén már nem lehet a kényszereket a megengedett értékkombinációk felsorolásával felírni, ehelyett egy kényszernyelvet kell használni. Továbbá az ilyen kényszereket nem lehet megoldani az összes lehetséges kombináció felsorolásával, mert végtelen sok van belőlük. Különleges megoldó algoritmusok léteznek egész értékű változók lineáris korlátjaira, azaz olyan korlátokra, ahol mindegyik változó csak lineáris műveletekben jelenik meg.

A folytonos tartományú kényszer kielégítési problémák elég gyakoriak a valós alkalmazásokban, és az operációkutatáson belül is tanulmányozták ezeket a problémákat. A folytonos tartományú kényszer kielégítési problémák legismertebb válfaja a lineáris programozási példák csoportja, ahol a kényszerek konvex tartományt alkotó lineáris egyenlőtlenségek.

Azon túl, hogy megvizsgáljuk a kényszer kielégítési problémákban megjelenő változók típusait, hasznos lehet a kényszerek fajtáit is tanulmányozni. A legegyszerűbb fajta az unáris

kényszer, amely egy változó értékére tesz megkötést. A bináris kényszer két változót köt össze. A csak bináris kényszerekkel rendelkező kényszer kielégítési problémát binárisnak nevezzük.



2.2.2. ábra: Baloldalon: Betűrejtvény. Minden betű más számjegyet helyettesít. A cél olyan behelyettesítés(ek) megtalálása, melyekre a művelet teljesül. Jobboldalon: A probléma kényszer hipergráfja.[6]

A magasabb rendű kényszerek három vagy több változóra vonatkoznak, ezeket egy kényszer hipergráffal lehet ábrázolni. Ennek egy tipikus példája a betűrejtvény. A betűrejtvényben mindegyik betű különböző számot jelöl. Ez a megkötés megadható egy sokváltozós kényszerrel, vagy bináris kényszerek egy gyűjteményével is. Bizonyítható, hogy elegendő segédváltozó bevezetésével minden magasabb rendű véges tartományú kényszer átírható bináris kényszerek halmazává.[6]

```
O + O = R + 10 * X1

X1 + W + W = U + 10 * X2

X2 + T + T = O + 10 * X3

X3 = F
```

2.2.1. Példa: Korlátok a fenti betűrejtvényhez.

2.3. Kényszer kielégítési problémák megoldási struktúrája

Ezek után egy kényszer kielégítési probléma megoldásának főbb lépései a következők:

- A probléma leképezése a kényszer kielégítési problémák világára. A problémának egy olyan modelljét kell megadnunk, melyben a probléma egyes elemeit a megfelelő értéktartományokra képezhetjük le.

- Változók és korlátok felvétele. Be kell vezetnünk a feladatban szereplő változókat és fel kell vennünk a változók között fennálló korlát-relációkat.
- Címkézés. Ha a problémáknak a korlátok alapján nincs egyértelmű megoldása, vagy ezt a rendszer nem tudja kikövetkeztetni, akkor a változókat el kell kezdenünk szisztematikusan ez értéktartományaik egy-egy lehetséges értékéhez kötni, így meg fogjuk kapni a probléma összes lehetséges megoldását. Ha a problémának a korlátok felvétele után már egyértelmű a megoldása, akkor a címkézési fázis elmarad.[2]

3. Korlát-logikai programozás

3. 1. A korlát-programozási paradigma és a korlát-logikai programozás

A korlát-programozás egy deklaratív programozási paradigma, melyben a változók közti relációkat korlátok formájában adjuk meg. A deklaratív szemléletmódból adódóan nem a megoldáshoz elvezető lépések sorozatát adja meg, hanem a megoldás tulajdonságait írja le. A korlátok megfogalmazása különböző tudományterületekből és megoldási módszerekből származhat. Ezek lehetnek logikában felírt korlátok, kényszer kielégítési problémák vagy a szimplex módszer által megoldható problémák.

A korlát-programozási szemléletet valamilyen más programozási nyelvbe szokás ágyazni. Az egyik legelső ilyen nyelv a Prolog volt, így a területet korlát-logikai programozásnak kezdték hívni. A két paradigmának sok közös tulajdonsága van, többek között a változó fogalom, a visszalépéses keresés valamint a probléma felírásának szemlélete.

A két paradigma közötti különbség a modellezett világhoz való viszonyukban van. Bizonyos problémák a logikai programozás szemléletmódjához állnak közelebb, bizonyos problémák pedig a korlát-programozás szemléletmódjához.

A korlát-programozási szemléletmód a világ állapotainak egy halmazában keres, miközben nagyszámú korlátnak kell egyszerre teljesülnie.[4]

3. 2. Korlát-logikai programozás

A korlát-logikai programozás a hagyományos logikai programozás kibővítése a korlát-kielégítési problémák témaköréből vett elemekkel. Alapvető tulajdonsága, hogy a program tartalmazhat a változók értékeire való megszorításokat. Ezek egy bizonyos alaphalmazhoz tartozó megszorítások. Ebből az alaphalmazból (vagy ennek részhalmazából) vehetik fel a változók az értékeiket, és a változókra megfogalmazható relációkat, függvényeket is ez az alaphalmaz definiálja.

A függvényeket és a relációkat CLP-ben nem szintaktikusan, hanem szemantikusan kezeljük. A hagyományos Prolog rendszerekben a program dolga, hogy jelentést tulajdonítson egy függvénykifejezésnek. A végrehajtási algoritmus csak a szintaktikus egyesítés műveletét ismeri (az =/2 műveletet), ennek következtében pl. az X + 2 = X + 1 + 1 hívás hamis értékkel tér vissza, holott matematikailag ez egy helyes állítás. Ahhoz hogy ezt az eredményt megkapjuk, szükségünk van egy úgynevezett korlátmegoldóra, ami a változókból, konstansokból, függvényekből és relációkból álló korlátokat ki tudja értékelni. Fontos probléma a konzisztencia-vizsgálat, azaz az ellentmondások kiszűrése. Ha egy új korlát hozzátételével egy változó a hozzárendelt értéktartományából már semmilyen értéket sem vehet fel úgy, hogy a hozzárendelt korlátok teljesüljenek, ezt a korlátmegoldó észreveszi és az aktuális végrehajtási ág meghiúsul.

Formálisan leírva egy CLP rendszer egy <D, F, R, S> struktúrával írható le, ahol:

- D az alaphalmaz.
- F a fenti halmazon értelmezett függvények halmaza.
- R a fenti halmazon értelmezett relációk halmaza.
- S egy korlátmegoldó algoritmus a fentiekből felépített korlátokra.[2]

3.3. Alaphalmazok

Minden CLP séma egy adattartományon értelmezett korlátokra vonatkozó hatékony következtetési mechanizmus. Az adattartomány különféle megválasztásaiból más-más CLP sémák adódnak.

3.3.1. Logikai korlátok

A legtöbb CLP rendszer rendelkezik logikai korlátokat leíró könyvtárral. Ebben az esetben az alaphalmaz mindössze két értékből, az igaz és hamis értékekből áll. A két értelmezett konstans a 0 és az 1 a szokásos interpretációval. A nyelv ezen kívül még tartalmazza a logikában megszokott operátorokat és relációkat.
Ezek a korlátok sokféleképpen felhasználhatóak, például a digitális áramkör tervezésben is.[2]

3.3.2. Numerikus korlátok: Racionális számok

Ebben az esetben az alaphalmaz a racionális vagy valós számok halmaza, a korlátok pedig az ezek közt fennálló lineáris egyenlőségek vagy egyenlőtlenségek. Ezek a megoldások csak akkor támogatnak nemlineáris korlátokat, ha azok visszavezethetőek lineáris korlátokra. Az alkalmazott következtetési módszerek pedig a Gauss-elimináció és a szimplex módszer. A két megvalósítás közti különbség abban nyilvánul meg, hogy a racionális számokkal foglalkozó könyvtár matematikai értelemben vett racionális számokkal foglalkozik, a másik csak lebegőpontos formában ábrázolt valós számokkal.[2]

3.3.3. Numerikus korlátok: Egész számok

Ez a séma egész számok véges tartományain alapuló rendszert valósít meg. A felhasználható függvények az aritmetikában megszokott műveletek, a moduló, az abszolút érték, valamint a halmazok minimumát és maximumát visszaadó függvények. A felhasználható relációk pedig megint csak az aritmetikában megszokottak, valamint különféle halmazokkal kapcsolatos relációk. Következtetési mechanizmusnak pedig a mesterséges intelligencia kutatásokból ismert CSP módszereket használ.[2]

3.4. CLP procedurális szemantika

Mint láthattuk, egy korlát-logikai program abban különbözik egy hagyományos logikai programtól, hogy kijelentéseinket egy bizonyos alaphalmazon megfogalmazott korlátokkal

tehetjük pontosabbá, és ezek megoldásához, egyszerűsítéséhez külön végrehajtó algoritmus adódik. Tehát egy CLP program két részből áll: megfogalmazhatunk hagyományos állításokat és korlátokat. Ezek után egy korlát-logikai program klózok egy véges halmaza:

$$A_0 \leftarrow C_1, \ldots, C_m, A_1, \ldots, A_n \qquad (m, n \geq 0)$$

Ahol C_i az adott alaphalmaz változóiból, konstansaiból, függvényeiből és relációiból felépíttet kifejezések, A_i-k pedig hagyományos Prolog kifejezések.
A levezetés során pedig figyelembe kell venni nemcsak a végrehajtás során éppen vizsgált korlátot, hanem az eddig vizsgált összes korlátot is.
Ezek után a levezetés egy állapota a <G, s> párral jellemezhető, ahol:

- G a megoldandó célok és korlátok konjunkciója
- s egy korlát-tár, ami az eddig felhalmozott korlátokat tartalmazza.

Fontos elvárásunk lehet a korlát-tárral szemben, hogy konzisztens és kielégíthető legyen, tehát ne tartalmazzon ellentmondást.
A legtöbb CLP megvalósítás korlát-tára csak korlátok egy bizonyos fajtáját tárolja, ezeket egyszerű korlátoknak nevezzük, minden más korlátot pedig összetett korlátnak. Például valós vagy racionális esetben ezek a változókra vonatkozó egyenlőtlenségek, egész esetben pedig legtöbbször csupán az „eleme” reláció, tehát egy változó egy bizonyos halmazból vehet fel értéket. Az összetett korlátok felfüggesztve várják, hogy egyszerűsítés után a korlát-tárba kerülhessenek. A korlátok és a korlát-tár egyszerűsítésére sokféle ok miatt szükség lehet. Az egyik szempont a rendszer által adott válasz egyszerűsége. Nem túl informatív az a válasz, ami változatlan alakban felsorolja az összes érintett korlátot. A másik fontos szempont a konzisztencia ellenőrzés bonyolultsága. Nyilvánvaló, hogy bizonyos aritmetikai egyenlőségek, egyenlőtlenségek sok esetben összevonhatók. Logikai esetben is lehet hasonló helyzetet találni. Evvel a korlát-tár mérete csökkenthető, egy kisebb és egyszerűbb korlát-tár konzisztencia vizsgálata pedig sokkal kevesebb időt vesz igénybe.

A végrehajtás során a korlát-tárral 3 művelet végezhető:

- A korlát-tár bővítése: a hagyományos Prolog végrehajtás során feldolgozott korlátok hozzá vétele a korlát-tárhoz.
- A korlát-tár egyszerűsítése.
- Konzisztencia ellenőrzés a korlát-táron.

A levezetés lépéseiben az éppen aktuális állapoton végezzük el a fenti műveletek valamelyikét.
Arra, hogy ezen lépések milyen sorrendben kövessék egymást, sokféle stratégia létezik. Egy nézőpont lehet, hogy addig bővítjük a korlát-tárat ameddig lehetséges, és aztán egyszerűsítünk, majd ellenőrizzük a konzisztenciát. Ez a „lusta" módszer azonban kockázatos, hiszen a végrehajtási algoritmus könnyen végtelen ciklusba eshet egy kielégíthetetlen korlát-tár miatt.
Egy másik megközelítési mód, hogy minden bővítés után egyszerűsítünk és ellenőrizzük a konzisztenciát. Ezzel a „mohó" stratégiával elkerülhetjük az előző hibát. Hatékonysági okokból viszont fontos hogy a korlát-tár konzisztencia ellenőrzését hatékonyan tehessük meg, különben komoly számítási költséggel kell számolnunk. A legtöbb mai Prolog rendszer, köztük a SWI Prolog is, ezt a stratégiát valósítja meg.[2]

3.5. A SWI Prolog korlát-logikai könyvtára

A következőkben egy egész számok alaphalmazán értelmezett korlát-logikai könyvtár eszközrendszerét ismerhetjük meg.
A clp/bounds egy egyszerű egész korlát-megoldó, a SICStus clp(fd) szintaxisának egy részét valósítja meg. A következő függvényeket és relációkat támogatja:
Változókra és változók listájára adhatunk meg értékkorlátozást az „in" korlát segítségével. A baloldalon szerepelnek a változó(k), a jobboldalon pedig egy A..F kifejezés, ahol A a tartomány legkisebb felvehető értéke, F pedig a legnagyobb. Ez a korlát tehát a változókhoz egy intervallumot rendel.
Az egyenlőség/egyenlőtlenség relációk írásmódja abban különbözik a hagyományos Prolog írásmódtól, hogy egy # jel kerül az operátorok elé.
A sum/3 segítségével változók egy csoportjára is meg tudunk adni összetett korlátokat.

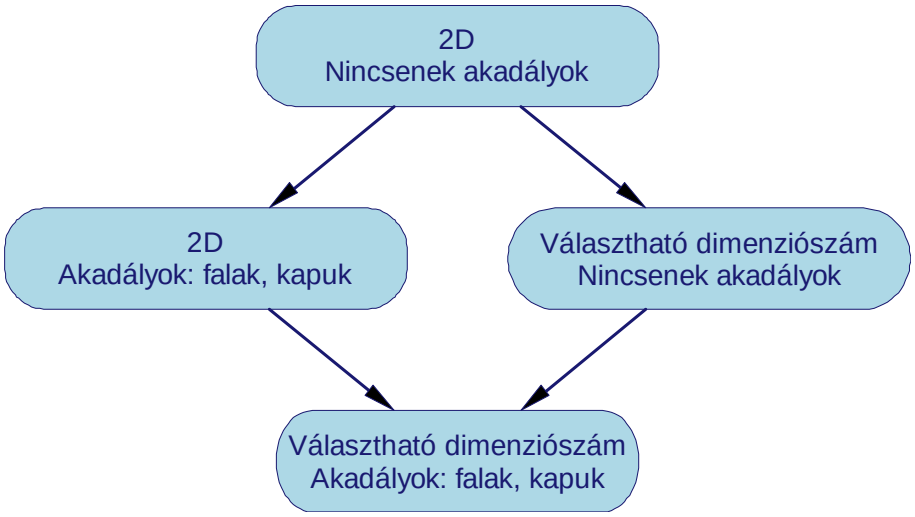
Ki tudjuk jelenteni változók egy csoportjáról, hogy mind különböző értékeket takarnak az alldiff/1 segítségével.
Talán az egyik legfontosabb korlát az indomain/1, ami az argumentumában megadott változóhoz hozzárendel egy értéket a változó ismert lehetséges értékei közül, valamint visszalépés esetén szisztematikusan más-más értékekkel próbálkozik.
A másik legfontosabb korlát a label/1, változók listájához szisztematikusan rendel egy-egy elemet a változók értéktartományaiból.
Ezen kívül a könyvtár támogatja a klasszikus aritmetikai műveleteket, valamint a minimum, a maximum, a moduló és az abszolút érték függvényeket is.
A clp/distinct könyvtár ehhez egy új predikátumot ad hozzá, ez a vars_in/2, aminek első paramétere változók egy listája, második paramétere pedig az ezen változók által felvehető értékek listája. Ez korlát tehát a változóihoz értékek egy listáját rendeli.[8]

4. A feladat

4.1. Feladatváltozatok

Az eredetileg célul kitűzött feladat a népszerű kukac játék egy régebbi változata volt. A játék legismertebb változatában a kukac egy kétdimenziós játéktéren belül folyamatosan mozog miközben különböző tárgyakat kell felvennie (megennie) a játéktér bizonyos pontjain és ezekért pontokat kap. Amikor ez sikeres, a kukac „megnő”. Ahogy a játékos egyre több pontot szerez, a kukac egyre nagyobb lesz, és egyre kevésbé fér el a játéktérben. A játék különböző változatai akadályokat is értelmezhetnek a játéktéren belül. Ezek lehetnek falak, lyukak, stb. A játék közben a játékosnak ezeket el kell kerülnie. Ha nincsenek akadályok a pályán, a játékosnak csak arra kell vigyáznia, hogy a kukac ne ütközzön saját magába, valamint ne próbáljon kimenni a játéktérből, a másik esetben viszont a feladat nehezül egy kicsit.
A játéknak abban a változatában, amellyel én foglalkoztam a kukacnak nem kell tárgyakat felszednie, viszont minden egyes lépés után automatikusan növekszik. Ezek után a játék a következőképpen fogalmazható meg: a játékosnak a játékteret úgy kell bejárnia, hogy minden

mezőt érint és minden mezőt csak egyszer érint. Az eredeti feladat kétdimenziós játékteret értelmez különösebb akadályok nélkül. A program különböző változatai különböző módon bővítik a pályák leírására szolgáló lehetőségeket.



4.1.1. ábra: Programváltozatok és tulajdonságaik.

Összesen négy valamelyest különböző program készült a megfelelő bonyolultságú pályák leírására.

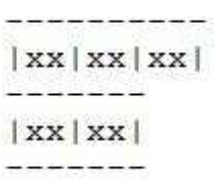
- A legelső, legegyszerűbb változat ez eredeti feladat kétdimenziós játékterét értelmezi és bonyolultabb akadályokat nem lehet megfogalmazni benne.
- A második is kétdimenziós játékteret értelmez, és két különböző akadályt lehet bele tenni: az egyik egy kapu, amelyen keresztül a játéktér egyik pontjából egy másik pontjába juthatunk, ezen kívül rakhatunk falakat két mező közé.
- A harmadik változatban már egy tetszőleges számú elemből álló tömb ábrázolja a kukac jelenlegi koordinátáit. Ez azt teszi lehetővé, hogy a kukac egy tetszőleges (nem 0) dimenziójú teret derítsen föl.
- A negyedik, végső változat pedig kombinálja a második és a harmadik változat tulajdonságait: egy n dimenziós térben lehet egyrészt kapukat megfogalmazni, melyeken keresztül a játéktér egyik pontjából a másik pontba lehet jutni. Másrészt egy adott mezőből letilthatjuk a mozgást valamilyen irány(ok)ba.

A programhoz tartozik egy egyszerű kiíratót algoritmus, amellyel két- és háromdimenziós pályákat és megoldásaikat írathatunk ki. A többi esetben a grafikus megjelenítésnek nem sok értelme lenne. Ezekben az esetekben a program kiírja a megoldásként kapott lépéssorozatot.

4.2. Játékterek

Ebben a fejezetben sorban átnézzük a különböző változatok által megfogalmazható játéktereket, és megvizsgálunk néhány, a megoldásukkal kapcsolatos érdekességet. Ezen kívül megadjuk a leírásukhoz szükséges formalizmust.

4.2.1. Kétdimenziós, akadálymentes játéktér



4.2.1.1. ábra: Egyszerű pálya.

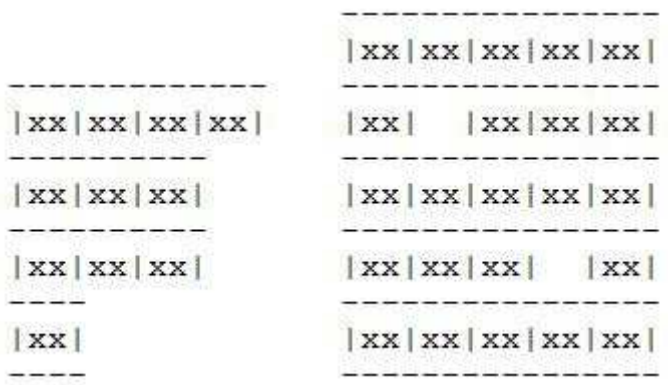
A feladat legegyszerűbb változatában a játéktér kétdimenziós, és a mezőkre semmilyen különleges korlátozást nem fogalmazhatunk meg. Az ábrán egy egyszerű pálya látható. A bal felső sarokból indulva kell bejárni a játékteret. Látható, hogy a feladatnak csak egy megoldása létezik. Ennél a leírásmódnál minden mezőt a mező/2 predikátum egy klóza ír le, melynek argumentumai a mező kétdimenziós koordinátái. Ez a leírási mód több szempontból is rugalmatlan. Nem tesz lehetővé semmilyen további megkötést a mezőkre vonatkozóan, valamint a lehetséges többdimenziós pályák esetében az argumentumszámot növelni kellene, ami roppant nehézzé teszi egy általános dimenziós esetet megoldó algoritmus elkészítését.

```
mező(0, 0).
mező(1, 0).
mező(1, 1).
mező(0, 1).
mező(0, 2).
```

```
4.2.1.1. Példa:
Egyszerű 2D pálya.
```

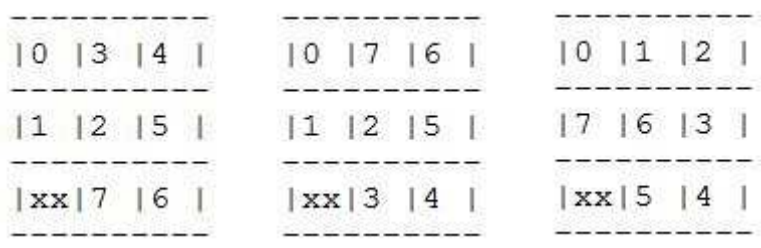
Az alábbi két pálya közül egyik sem megoldható. Az első esetben a probléma nyilvánvaló: a bal felső sarokból kiindulva nem tudjuk úgy bejárni a játékteret, hogy érintsük mindkét „kiálló” mezőt. Olyan mezőből, amibe csakis egy másik mezőből lehet eljutni, csak kettő lehet a játéktérben, és ezeknek kell lenniük a kezdő- és végpontoknak. A másik esetben az 5x5-ös játéktérből 2 mező hiányzik, ez önmagából nem jelent problémát. Mivel a bal felső sarokból indulunk, a hiányzó négyzet miatt az utat a kiinduló mezővel szomszédos mezők valamelyikében kell befejeznünk. A játéktér ezek alapján a kezdőpont környékére, a kezdőponttól távolabbi sarokra, és két 3x3-as alakzatra bontható. A problémát a játéktér két

3x3-as része jelenti, pontosabban szólva ezek önmagukban nem jelentenének problémát, ha nem fednék át egymást. A két rész között a pálya kezdőponttól távolabbi végén lévő sarok jelenti az átjárást. Az átfedés miatt azonban a második 3x3-as pályaszeletből az egyik sarok hiányzik. Ez az alakzat önmagában megoldható, viszont nem járható be úgy, hogy a meghatározott belépési pontból, az ugyancsak megkötött kilépési pontba jussunk. Mivel a pálya szimmetrikus, bármelyik irányba indulva ugyanahhoz a problémához érünk.



4.2.1.2. ábra: Megoldhatatlan pályák.

Az alakzat megoldásai a következő ábrákon láthatóak. A fent említett példával kapcsolatban az elvárásunk az lenne, hogy az ábrán 0-val jelölt kezdőpontból a jobb alsó sarokba jussunk el. A három megoldás közül azonban ez egyre sem teljesül.



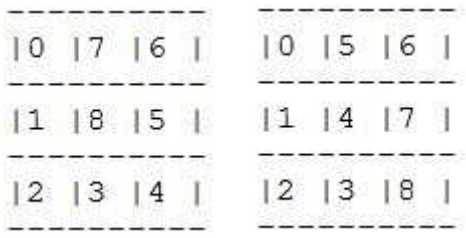
4.2.1.3. ábra: 3x3-as pálya, melyből egy sarok hiányzik. A 0-val jelölt kezdőpontból induló megoldások.

4.2.2. Kétdimenziós korlátozható játéktér

A feladat második változatában a játéktér még mindig kétdimenziós, ugyanakkor különböző korlátozásokat fogalmazhatunk meg a mezőkre vonatkozóan. Ennek megfelelően a mezők leírási módja is megváltozott. A mezőket még mindig a mező/2 predikátum klózai írják le. Ezeknek az első argumentuma egy kételemű lista, ami az adott mező koordinátáit ábrázolja. A második argumentum a mezőre vonatkozó korlátozásokat jelenti. Ezen korlátozások egyikét nevezhetjük falnak. Ez alatt azt értjük, hogy két, egyébként szomszédos mező között megtiltjuk az átjárást. Egy mezőhöz egy, kettő vagy három ilyen falat rendelhetünk kétdimenziós esetben. A programnak ez a változata egy vagy két falat értelmezhet a mezők között. Ezekből összesen tíz van, négy egyfalas korlát, amelyek a mezők négy oldalát lefedik, valamint hatféle kétfalas korlát. A baloldali példában egy olyan pályát látunk, amelyben két fal szerepel. Ennek a feladatnak csak két megoldása van, ezek az alábbi ábrákon láthatók. A kezdőpontból csak egy irányba lehet kiindulni úgy, hogy teljesítsük a feladatot, ezek után pedig a pálya közepén szereplő „hídon” kétféle irányban lehet átmenni, a két megoldás ezeknek az irányoknak felel meg.

```
mező([0, 0], 0).
mező([0, 1], 0).
mező([0, 2], 0).
mező([1, 0], b2).
mező([1, 1], t1m1).
mező([1, 2], bm2).
mező([2, 0], 0).
mező([2, 1], 0).
mező([2, 2], 0).
```

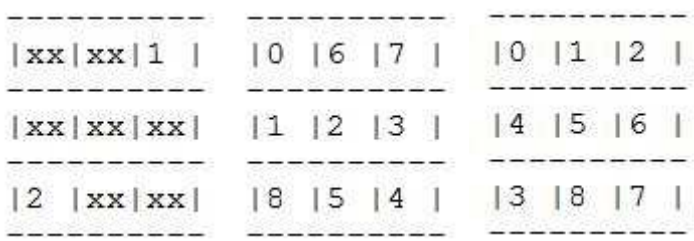
4.2.2.1. Példa: Egy 3x3-as játéktér, melyben két fal létezik: a [0, 1] és az [1, 1] mező között, valamint az [1, 1] és az [1, 2] mező között.



4.2.2.1. ábra: A 4.2.2.1. példa megoldásai.

A másik fajta korlátozás lehetővé teszi, hogy a pálya egyik pontjából egy másik, távolabbi pontjába jussunk el. Egy ilyen korlátozással ellátott mezőről az egyetlen út a „kapu” másik oldalán lévő mezőre vezet. A kapukat hasonló módon adhatjuk meg, mint a falakat. Azonban kapuk esetében külön meg kell adnunk azt a szabályt, ami alapján a kapu működni fog. Ebben a programváltozatban ezt a szabályt egy gate/5 predikátum írja le, amelyben meg kell adni azt

a két mezőt, amelyeket a kapu összeköt. Az alábbi pályák közül az elsőnek nincs megoldása, mivel a benne szereplő két kapu egymásra mutat. Ez azt eredményezi, hogy az egyik kapuba belépve a másik kapu azonnal visszaküldené a játékost az első kapuba. Ez a lépés csak akkor lenne elfogadható, ha az utolsó lenne a játéktér bejárása során. Azonban ha a két kapura nem szabad lépnünk, a maradék játéktér bejárhatatlan.



4.2.2.2. ábra: A baloldali esetben a két kapu egymásra mutat, emiatt a feladat megoldhatatlan. A második esetben a kapuk egymás mellé mutatnak, az ábrákon a probléma két megoldása látható

A második esetben a két kapu nem egymásra mutat, hanem egymás mellé, pontosabban szólva a [0, 2] mezőről a [2, 0] mezőre, illetve a [2, 1] mezőről a [0, 1] mezőre. Ennek a feladatnak két megoldása van, amelyek a fenti ábrán láthatók. Az alábbi példánál látható a kapuk megfogalmazási módja. Ez a forma különösebb alakítás nélkül hozzáadható az algoritmus szabálykészletéhez.

```
mező([0, 0], 0).
mező([0, 1], 0).
mező([0, 2], g1).
mező([1, 0], 0).
mező([1, 1], 0).
mező([1, 2], 0).
mező([2, 0], 0).
mező([2, 1], g2).
mező([2, 2], 0).

gate(g1, _, [0, 2], [2, 0], 0).
gate(g2, _, [2, 1], [0, 1], 0).
```

4.2.2.2. Példa: Kapuk megadási módja

Abban az esetben, amikor nem teszünk korlátozást egy mezőre, az aktuális klóz második argumentuma 0 értéket kap. Ez jelzi az algoritmusnak, hogy az adott mezőre módosítatlan

szabályrendszer vonatkozik. Falak esetében a megfelelő korlátokat az algoritmusban előre rögzített kódok jelölik, ezeket kell az adott mezőkhöz rendelni. Kapuk esetében a megadási mód miatt a felhasználó szabadon használhat bármely, a szabályrendszerben elő nem forduló jelölést.

4.2.3. Általános dimenziós játékterek

Ebben a változatban a mező/2 predikátum első argumentumában szereplő koordinátalistára csak egy kikötést teszünk: nem lehet üres lista. Ugyanakkor ez a változat nem értelmezi az előző fejezetben tárgyalt akadályokat. Ez abban nyilvánul meg, hogy a második argumentum minden esetben 0 értéket kap.

Egydimenziós esetekben egy irányban haladhatunk a játéktérben, ezek az esetek nem túl érdekesek.

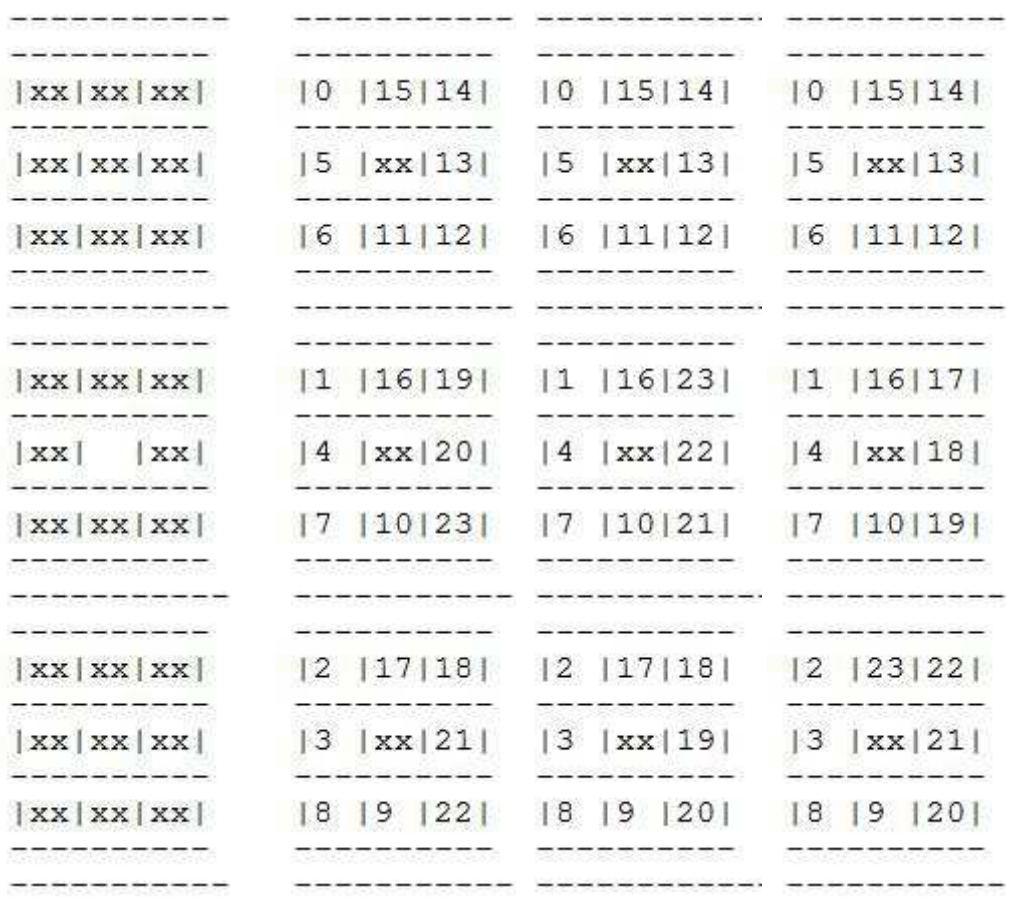
A két- és háromdimenziós játékterek a legismertebbek. Ezek megfelelő kihívást jelentenek önmagukban is, különösebb bonyolítások nélkül, és az ember számára könnyen elképzelhetőek.

A magasabb dimenziós játékterek azonban nehezebb feladatot jelentenek. Ezek elképzelése, legalábbis a szerző számára, lehetetlen feladat. A probléma azonban megfogalmazható. Adott egy n hosszúságú listákat ábrázoló klózokból álló predikátum. A listák elemeinek sorszáma ábrázolja a tengelyeket, ezek mentén mozoghatunk. A listák között az adott lista egyik elemének 1-gyel való növelésével vagy csökkentésével juthatunk egy másik listába, feltéve hogy az a lista létezik. A célunk pedig még mindig az, hogy bejárjuk ezeket a listákat úgy, hogy minden listát csak egyszer érintünk.

A pályák kiíratásával kapcsolatban is lehetnek problémák. Kétdimenziós játékterek ábrázolása egyszerű, és a háromdimenziós eset is szemléltethető. Utóbbi esetben a játékteret „szeletelve”, szintenként ábrázoljuk. Egydimenziós esetben a megoldásul kapott lépéssorozat is elegendő a szemléltetéshez. Magasabb dimenziós esetek az ábrázolására a szerző nem vállalkozik, ebben az esetben a megoldásul kapott lépéssorozatnak elegendőnek kell lennie.

Az alábbi ábrákon két háromdimenziós feladat szerepel. Mindkét változat alapja egy 3x3x3-as kocka, amelyből az első esetben csak a középső mezőt távolítjuk el, a második esetben pedig a kocka közepén lévő, három mezőből álló oszlopot.

Meglepő módon az első eset megoldhatatlan: az egyik sarokból indulva lehetetlenség úgy bejárni a játékteret, hogy minden mezőt csak egyszer érintünk. A második esetben a probléma megoldható, néhány útvonalat az alábbi ábrán láthatunk.



4.2.3.1. ábra: Háromdimenziós problémák ábrázolása és megoldása. A baloldali eset megoldhatatlan. A jobboldali esetek a fentebb vázolt feladat néhány megoldásai.

A probléma ennél is izgalmasabbá válik, ha háromnál több dimenziós feladatokat vizsgálunk. Az alábbi példák közül az első egy ötdimenziós 2 élhosszúságú hiperkocka. Ennek megadási módja a koordinátákat ábrázoló lista hosszán kívül semmiben nem különbözik a kettő- vagy háromdimenziós esettől. Ez a hiperkocka 32 „mezőből” áll. A második példa egy 10 mezőből álló négydimenziós játéktér, aminek öt megoldása van, ezek közül egyet láthatunk az alábbi ábrán.

```
mező([0, 1, 0, 0, 0], 0).
mező([0, 1, 0, 0, 1], 0).
mező([0, 1, 0, 1, 0], 0).
mező([0, 1, 0, 1, 1], 0).
mező([0, 1, 1, 0, 0], 0).
mező([0, 1, 1, 0, 1], 0).
mező([0, 1, 1, 1, 0], 0).
mező([0, 1, 1, 1, 1], 0).
mező([1, 0, 0, 0, 0], 0).
mező([1, 0, 0, 0, 1], 0).
mező([1, 0, 0, 1, 0], 0).
mező([1, 0, 0, 1, 1], 0).
mező([1, 0, 1, 0, 0], 0).
mező([1, 0, 1, 0, 1], 0).
```

```
0   [0, 0, 0, 0]
1   [0, 0, 1, 0]
2   [0, 0, 1, 1]
3   [0, 0, 0, 1]
4   [1, 0, 0, 1]
5   [1, 0, 0, 0]
6   [1, 0, 1, 0]
7   [1, 1, 1, 0]
8   [0, 1, 1, 0]
9   [0, 1, 0, 0]
```

4.2.3.2. ábra: Baloldalon: egy ötdimenziós hiperkocka néhány mezőjének megadási módja. Jobboldalon: egy négydimenziós feladat egy megoldása

4.2.4. Általános dimenziós eset bővített szabályrendszerrel

A program ezen változata az előző változathoz hasonló rugalmasságot ad a probléma megfogalmazására a dimenziók számát tekintve, ugyanakkor lehetővé teszi a 4.2.2-es fejezet által taglalt szabályok alkalmazását egy megváltozott formában. A falak értelmezése a különböző dimenziós esetekben változhat, ezért a megfogalmazás a következőképpen módosul. Az általunk kiválasztott mezőket megjelölhetjük, és az így megjelölt mezőkre a következő korlátozásokat tehetjük: letilthatjuk a belőle egy vagy több általunk kiválasztott tengely mentén az elmozdulást. Ez a jelölésmód azt is maga után vonja, hogy kénytelenek vagyunk az összes, a korlátozás által érintett mezőhöz megadni ezt a megszorítást, ha a hagyományos értelemben vett falakat akarunk a pályához adni. Ugyanakkor ennek segítségével értelmezhetünk „egyirányú” falakat is, amelyeken csak bizonyos irányból lehet áthaladni. Az alábbi példában egy 2x2x2-es kockába illesztünk egy síkot, mely két részre osztja a kockát, de van rajta egy átjáró, a [0, 0, 1] és a [1, 0, 1] mező között. A példában jól látható a mezők jelölési módja. A mező/2 predikátum megjelölt klózai egy tetszőlegesen választható kódot kapnak, majd a block/2 predikátum klózai ezekhez egy vagy több tengelyt rendelnek, melyek irányában tiltjuk az elmozdulást. Ezeket a megkötéseket az

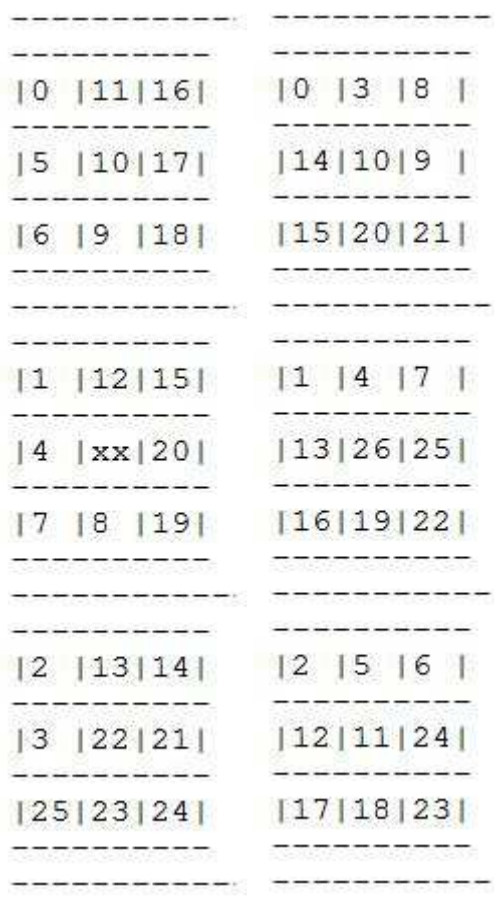
```
mező([0, 0, 0], b0).
mező([0, 0, 1], 0).
mező([0, 1, 0], b1).
mező([0, 1, 1], b2).
mező([1, 0, 0], b3).
mező([1, 0, 1], 0).
mező([1, 1, 0], b4).
mező([1, 1, 1], b5).

block(b0, 1).
block(b1, 1).
block(b2, 1).
block(b3, -1).
block(b4, -1).
block(b5, -1).
```

4.2.4.1. Példa: Egy háromdimenziós feladat

algoritmus a feladat végrehajtása előtt konvertálja a szabályrendszerébe. A fent említett mezőkhöz nem rendeltünk ilyen megszorítást, ez értelmezi az átjárót a kocka két fele között. A feladat megoldása nem feltétlenül egyszerű, sőt az átjáró helyétül függően akár megoldhatatlan is lehet. Ha ez a bizonyos átjáró a kocka kiindulópont felöli szeletében, a kiindulóponttól számított átellenes sarokban van, a feladat megoldhatatlan. A másik két esetben a feladatnak két megoldása van.

A kapuk megadási módja viszont nem változott meg. Mindkét alábbi feladat alapja egy 3x3x3-as kocka. Az első ezek közül az előző fejezetben szereplő megoldhatatlan feladat. Itt a kocka közepén lévő mező hiányzik. Egy kapu beillesztésével azonban a feladat megoldhatóvá válik.



4.2.4.1. ábra: Háromdimenziós feladatok kapukkal

A másik példában három kaput illesztünk egy hiánytalan kockába a következő módon: a [0, 1, 1] mezőből vezet egy a [2, 1, 1] mezőbe, a [2, 0, 0] mezőből vezet egy a [0, 0, 1] mezőbe, illetve az [1, 1, 1] mezőből a [0, 0, 0]-ba. A legutolsó megszorítás azt is maga után

vonja, hogy az utat a kocka közepén kell befejezni, hiszen innen már csak az általunk már látogatott kezdőpontba tudunk visszatérni. Ennek a feladatnak egy megoldását a jobboldali ábra mutatja.
A legutolsó ábra egy hétdimenziós feladat két megoldását ábrázolja. A feladatban szerepel az összes értelmezett korlátozás és a leírási mód dimenziókkal kapcsolatos rugalmasságát is tükrözi, tehát a program által nyújtott teljes eszközkészletet használja.

```
0   [0, 0, 0, 0, 0, 0, 0]
1   [0, 1, 0, 0, 0, 0, 0]
2   [0, 1, 1, 0, 0, 0, 0]
3   [1, 1, 1, 0, 0, 0, 0]
4   [1, 0, 1, 0, 0, 0, 0]
5   [1, 0, 0, 0, 0, 0, 0]
6   [1, 0, 0, 1, 0, 0, 0]
7   [0, 0, 0, 1, 0, 0, 0]
8   [0, 0, 1, 1, 0, 0, 0]
9   [0, 0, 1, 0, 0, 0, 0]
10   [0, 0, 0, 0, 1, 1, 1]
11   [0, 0, 0, 0, 1, 0, 1]
12   [0, 0, 0, 0, 1, 0, 0]
13   [0, 0, 0, 0, 1, 1, 0]
```

```
0   [0, 0, 0, 0, 0, 0, 0]
1   [0, 1, 0, 0, 0, 0, 0]
2   [0, 1, 1, 0, 0, 0, 0]
3   [1, 1, 1, 0, 0, 0, 0]
4   [1, 0, 1, 0, 0, 0, 0]
5   [1, 0, 0, 0, 0, 0, 0]
6   [1, 0, 0, 1, 0, 0, 0]
7   [0, 0, 0, 1, 0, 0, 0]
8   [0, 0, 1, 1, 0, 0, 0]
9   [0, 0, 1, 0, 0, 0, 0]
10   [0, 0, 0, 0, 1, 1, 1]
11   [0, 0, 0, 0, 1, 1, 0]
12   [0, 0, 0, 0, 1, 0, 0]
13   [0, 0, 0, 0, 1, 0, 1]
```

4.2.4.2. ábra: Hétdimenziós feladat megoldása

A feladat két részre bontható. Az első része az egyik előző példában vizsgált négydimenziós játéktér egy változata. Ebben a változatban azonban szerepelnek mind egyirányú, mind kétirányú falak. A [0, 0, 0, 0, 0, 0, 0] mezőt az [1, 0, 0, 0, 0, 0, 0] mezőtől egy kétirányú fal választja el. Az utóbbi mezőre viszont sem az [1, 0, 1, 0, 0, 0, 0] mezőről, sem az [1, 0, 0, 1, 0, 0, 0] mezőről nem lehet lépni, fordítva viszont a mozgás lehetséges. Ezek a korlátozások a részfeladat megoldásainak számát egyre csökkentik.
Az ötödik tengely afféle elválasztóként funkcionál, élesen elkülönítve a játék két részét. A játék második fele egy egyszerű 2x2-es kétdimenziós négyzet, különösebb korlátozások nélkül, aminek két megoldása van. A két játékrész között egy kapu jelent kapcsolatot, amely a [0, 0, 1, 0, 0, 0, 0] mezőt a [0, 0, 0, 0, 1, 1, 1] mezővel köti össze. Végezetül ennek a feladatnak a megfogalmazását láthatjuk, a program által megkövetelt módon.

```
mező([0, 0, 0, 0, 0, 0, 0], b1).
mező([0, 1, 0, 0, 0, 0, 0], 0).
mező([0, 1, 1, 0, 0, 0, 0], 0).
mező([1, 1, 1, 0, 0, 0, 0], 0).
mező([1, 0, 1, 0, 0, 0, 0], b3).
mező([1, 0, 0, 0, 0, 0, 0], b2).
mező([1, 0, 0, 1, 0, 0, 0], b4).
mező([0, 0, 0, 1, 0, 0, 0], 0).
mező([0, 0, 1, 1, 0, 0, 0], 0).
mező([0, 0, 1, 0, 0, 0, 0], g1).
mező([0, 0, 0, 0, 1, 1, 1], 0).
mező([0, 0, 0, 0, 1, 1, 0], 0).
mező([0, 0, 0, 0, 1, 0, 1], 0).
mező([0, 0, 0, 0, 1, 0, 0], 0).

block(b1, 1).
block(b1, 3).
block(b2, -1).
block(b3, -2).
block(b4, -4).

gate(g1, _, [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 1], 0).
```

4.2.4.2. Példa: Egy hétdimenziós feladat megfogalmazása. Ez a feladat tartalmaz egyirányú és kétirányú falakat, valamint egy kaput is.

5. Az algoritmus

5.1. Az algoritmusváltozatok kapcsolata

A program fejlesztése során mind az algoritmus, mind maga feladat sokat változott, bővült. Az eredeti program egy kétdimenziós pályán kereste a megfelelő útvonalat, a végső változat pedig egy általános dimenziós esetű akadályokkal teletűzdelt játéktérben keresi a megfelelő megoldást. A feladatok ábrázolására szolgáló formalizmus ugyan keveset változott, de az ezek megoldására szolgáló algoritmus sokat módosult a fellépő elvárásoknak megfelelően. Mivel a dolgozat célja a korlát-logikai programozás alkalmazási lehetőségeinek vizsgálata, az algoritmusok ismertetése során külön hangsúlyt kap ezeknek a programrészleteknek a kiemelése.

Az előző fejezetben ismertetett négy, egymásra épülő feladattípus megoldására négy, ugyancsak egymásra épülő algoritmus készült. Pontosabban szólva, ahogy a megoldandó

feladat egyre bővült, a megoldására szolgáló algoritmusnak is egyre bonyolultabb kihívásokkal kellett szembenéznie.

A korlát-logikai alkalmazások a szabályrendszerek megvalósításában kapták a legnagyobb szerepet, és ahogy a program bonyolódott, ezek egyre nagyobb hangsúlyt kapnak.

A megoldandó feladatok különbözősége miatt roppant nehéz megfelelő heurisztikát találni. A program fejlesztése során lokális keresési technikák is próbára kerültek, de végül egyszerűségi okokból a Prolog beépített keresési módszerén maradt a választás. Ez egy teljes keresési algoritmus, a visszalépéses keresés. Előnye, hogy az összes megoldást megtalálja, hátránya a lassú futási idő, még közepes méretű feladatok esetében is.

Az algoritmusok közötti különbség nem csak a leírási mód feldolgozásában változott meg, ahogy a program bővült, a programrészletek egyre inkább elkülönültek egymástól egy rugalmas struktúrát alkotva.

A programok vizsgálat során, helytakarékossági okokból, az algoritmus négy végleges változatát elemezzük csak, a köztes változatok ugyan érdekesek lehetnek, de az algoritmus fejlesztése során okkal bizonyultak vakvágánynak.

A program legelső változatában egy fő predikátum végezte a kereséssel kapcsolatos műveleteket, idővel először a szabályképzés, majd az elmozdulást végző programrészek különültek el egyre egymástól.

A kiíratási algoritmus az algoritmus legutolsó változatához készül, ugyanakkor visszafelé kompatibilis a programokkal, a legelső kivételével.

5.2. Kétdimenziós megoldó algoritmus

A program legelső változata a 4.2.1-es fejezetben vázolt kétdimenziós problémát oldja meg. A Prolog megvalósításból adódóan a kód rendkívül rövid, tömör. Abból adódóan, hogy az összes mezőt be kell járnunk, a megoldásul várható út hosszát ismerjük. Ezek alapján a megoldás menete a következő: a játéktérben a szabályoknak megfelelően folyamatosan mozgunk, miközben elkerüljük azokat a mezőket, ahol már jártunk.

A megfelelő korlát-logikai könyvtárak és a feladat betöltésén kívül mindössze négy predikátumból áll. A megoldási útvonal hosszának megállapítása a legelső feladat. Erre az útvonalhossz predikátum szolgál. Ez egy beépített eljárást használ a mező predikátum

klózainak számának megállapítására. Ebből az útvonalhossz számolási módja miatt egyet le kell vonnunk.

A keres predikátumnak nincsenek argumentumai, ez az algoritmus belépési pontja. Erre a célra az összes programváltozat ezt a predikátumot használja. Ennek első célja az útvonalhossz megállapítására szolgáló eljárást hívja, majd az algoritmus fő eljárását, az eredményül kapott listát pedig kiíratja. A programnak ez az egyetlen változata, amely nem kompatibilis a kiíró modullal.

Az újkoord predikátum számolja ki az adott koordináták és elmozdulás-tengely ismeretében az új koordinátákat. A koordinátatengelyek jelölési módja a következő: 1 és 2 jelöli a megfelelő tengelyeket, az ezeken való elmozdulás irányát pedig ezek előjele adja meg. A predikátum négy klóza ezek ismeretében számolja ki az új koordinátákat. 1 és -1 esetében az első koordinátát csökkenti vagy növeli, 2 és -2 esetben pedig a másodikat.

Az útvonal predikátum végzi a keresés műveletét. Az eljárás paraméterezése a következő: az első a még hátralévő útvonal hossza, ez minden sikeres lépés után csökken eggyel. A következő két koordináta ez éppen aktuális koordinátát jelöli. A negyedik koordináta azt az irányt jelöli, ami felől az adott koordinátájú mezőre érkeztünk. Az utolsó kettő pedig az eddig bejárt útvonalat gyűjti. Ez két szempontból is fontos: egyrészt el akarjuk kerülni a már meglátogatott mezőket, másrész a teljes hosszában bejárt útvonalat megoldásként vissza akarjuk adni. A kiinduló paraméterek a következők: a keres predikátum első céljával megállapított útvonalhossz, hiszen a teljes útvonalat be akarjuk járni. Mindig a [0, 0] koordinátáról indítjuk a keresést, tehát a következő két paraméter ez lesz. Azt is feltesszük, hogy az 1 irányban érkeztünk erre a pozícióra, a következő paraméter tehát ez lesz. Az eddig bejárt mezőket tartalmazó lista üres lista lesz, ez lesz a következő paraméter, az utolsóként megadott változó pedig sikeres futás esetén a teljes útvonalat adja vissza.

```
útvonal(0, _, _, _, L, L).
útvonal(N, X, Y, Et, L, L0):-
     Etm is -Et, T in -2..2, T #\= 0,
     T #\= Etm, indomain(T),
     újkoord(T, X, Y, Z, W), mező(Z, W),
     \+ member([Z, W, _], L), N1 is N-1,
     útvonal(N1, Z, W, T, [[X, Y, T]|L], L0).
```

5.2.1. Példa: A megoldó algoritmus egy részlete.

A predikátum első klóza a teljes út megtétele után, tehát az első argumentum 0-s értéke esetén utolsó paraméterként visszaadja az eddig a pontig megtett utat tartalmazó listát. A második klóz végzi magát a keresést. A következő elmozdulás irányát a T változó jelöli ez kezdetben a -2, 2 intervallumból vehet fel értékeket. Ebből levonjuk a 0-s irányt, mivel ilyet nem értelmeztünk, ezen kívül még a bejövő iránnyal ellentétes irányt is le kell vonnunk belőle, hiszen nem léphetünk vissza arra a mezőre ahonnan ide jutottunk. Mivel ezek a korlátok még nem határozzák meg egyértelműen a továbbhaladási irányt, elvégezzük a címkézés műveletét, így választási pontokat hagyva magunk után szisztematikusan végigjárhatjuk a lehetséges irányokat. Ezek után kiszámoljuk a koordinátákat, és leellenőrizzük, hogy az éppen próbált irányban van-e egyáltalán mező. Ez az egyetlen pont, ahol a mezők létét ellenőrizzük, tehát egy lépésben először elmozdulunk egy új koordinátára, majd meghatározzuk, hogy onnan milyen irányban lépünk tovább. E miatt a rendszer miatt lesz az útvonalhossz a mezők számánál eggyel kevesebb. Ezek után azt is ellenőriznünk kell, hogy ezen a mezőn jártunk-e már.

Utolsó lépésben pedig egy rekurzív hívás segítségével folytatjuk a keresést. Első paraméterként a hátralévő út hosszánál eggyel kevesebbet adunk át, majd az új koordinátákat és irányokat, valamint a bejárt mezők listájához hozzáfűzzük az éppen elhagyott mezőt.

5.3. Kétdimenziós megoldó algoritmus bővített szabályrendszerrel

A 4.2.2-es fejezetben szereplő feladatok megoldására készült a második algoritmus. A szabályrendszer implementálása miatt a kód hosszabb lett, mint az előző esetben. Az algoritmus felépítése is sokat módosult, egyedül az útvonal hosszát számoló eljárás változatlan.

A különbség már a legelső sorokban jelentkezik. A megfelelő könyvtárak és a pálya betöltése mellet szükség lesz a szabályok predikátum dinamikus kezelésére is. Ez a predikátum fogja megvalósítani a feladatban szereplő szabályrendszert. Mivel a játéktérben bárhová elhelyezhetünk kapukat, az ezekre vonatkozó korlátokat lehetetlenség előre kódolni, így szükséges a szabályrendszer dinamikus változtatása.

Ezt a műveletet az init eljárás végzi, ami a fő kereső predikátum meghívása előtt fut le. Ez lényegében beépített eljárások segítségével feldolgozza a feladatban szereplő gate/5 predikátumot, és a meglévő szabályrendszerhez illeszti. Az összes ilyen szabály feldolgozását

most nem a megszokott rekurzió biztosítja. A fail állítás sohasem teljesül, ezzel visszalépésre kényszerítve az algoritmust, amíg az összes szabályt fel nem dolgozza. Ezek után a predikátum második klóza feltétel nélkül igaz értékkel tér vissza.

```
init:-
        predicate_property(gate(_,_,_,_,_), interpreted),
        gate(A, B, C, D, E), assertz(szabalyok(A, B, C, D, E)), fail.
init.
```

5.3.1. Példa: Szabályok előfeldolgozása.

A predicate_property eljárás ebben a formában vizsgálja, hogy létezik-e egyáltalán kapu szabály a megadott feladatban. Ez azért fontos, mert ha nem, az eljárás következő célja nem meghiúsul, hanem futásidejű hibával tér vissza. Miután megtalálta a kaput, az algoritmus átnevezi, és lényegében változatlan formában hozzáfűzi a szabálygyűjteményhez.

Az új koordinátákat kiszámoló eljárás lényegében változatlan. Az egyetlen különbség a koordináták ábrázolásában van. Két külön argumentum helyett egy kételemű lista ábrázolja őket itt és a program többi részében is, előfutárként a program következő változatához.

A keres predikátum a belépési pont ebben a programban is, ez tulajdonképpen csak az init eljárás hívásával bővült.

Az útvonal predikátum irányítja a keresést, az előző változathoz hasonlóan itt is hat argumentuma van. A legelső az útvonalhossz, aminek számítási módja teljesen megegyezik az előzőváltozatban használttal. A második a koordinátákat tartalmazó kételemű lista. Mivel az előző fejezetben ismertetett módszer szerint irányítjuk a lépéseket, meg kell adnunk az adott mezőre érvényes szabályt is, hiszen ezt az előző rekurziós lépésben számoltuk ki. Ez a harmadik paraméter. A többi argumentum változatlan: a negyedik a bejövő irány, az ötödik az eddig megtett lépések listája, a hatodik pedig a keresés eredményét fogja visszaadni.

A predikátum első klóza változatlan módon az útvonal végének felismerését és az eredménylista visszaadását végzi. A második klóz viszont megváltozott. A mezőkre vonatkozó szabályok alkalmazását és az új koordináták kiszámolását a szabályok predikátum végzi. Ezek után következik az új koordinátákon lévő mező létének ellenőrzése, valamint annak vizsgálata, hogy ezt a mezőt látogattuk-e már. Az eljárás a rekurziós lépéssel zárul. A kezdő paraméterei is nagyon hasonlóak az előző változathoz. Egyedüli változások, hogy a

kezdő, [0, 0] koordinátákat egy kételemű lista ábrázolja, valamint ebben a változatban a kezdőpontra nem adhatunk meg semmilyen különleges korlátozást.

A szabályok predikátum egy adott mezőre határozza meg az onnan lehetséges továbbhaladási irányokat. Bemenő paraméterei a következők: szüksége van a jelenlegi mezőre vonatkozó korlátra. Ezeket egy minden mezőhöz hozzárendelt kód adja meg. Falak esetében az előre meghatározott kódokat kell használni, kapuk esetében a felhasználó adja meg ezt, az alapesetet pedig a 0 jelöli.

```
szabalyok(bm1, Etm, [X, Y], [Z, W], T):-
        T in -2..2, T #\= 0, T #\= -1, T #\= Etm,
        indomain(T), újkoord(T, [X, Y], [Z, W]).
szabalyok(bm2, Etm, [X, Y], [Z, W], T):-
        T in -2..2, T #\= 0, T #\= -2, T #\= Etm,
        indomain(T), újkoord(T, [X, Y], [Z, W]).
szabalyok(t1m1, Etm, [X, Y], [Z, W], T):-
        T #\= Etm, vars_in([T], [1, -1]),
        újkoord(T, [X, Y], [Z, W]).
```

5.3.2. Példa: Három faltípust meghatározó szabályrendszer.

A falakat a tíz faltípusnak megfelelően tíz kód jelöli. A tíz lehetséges faltípus: 4 jelöli azt az esetet, amikor a mezőt egy fal határolja a négy lehetséges irány valamelyikében. Két esetben két fal párhuzamos egymással, valamint további négy esetben a mező egyik sarkát határolják.

Második paraméterként szükségünk van a bejövő irányra, hiszen nem akarunk visszamenni oda ahonnan jöttünk, a harmadik paraméter pedig a jelenlegi koordinátákat adja meg. A két kimenő paraméter pedig az új koordinátákat és a továbbhaladás irányát adják meg.

A falak első négy esetében, ahol csak egy irányt korlátoznak, a művelet hasonló az előző fejezetben tárgyalthoz. A továbblépés irányára legelőször a -2 és 2 alsó illetve felső korlátot tesszük, majd ezt szűkítjük. Először a 0-s irányt vonjuk le, majd a bejövő iránnyal ellenkezőt, és végül a falnak megfelelő irányt. Ezek után címkézünk és meghatározzuk a kimenő koordinátákat, valamint visszaadjuk a kimenő irányt.

Amikor viszont két fal határol egy mezőt, a bemenő irány ismeretében egyértelműen meghatározható a kimenő irány.

Kapuk esetében a szabályrendszer az ezeknek megfelelő tényállításokkal bővül. Ekkor a bemenő és a kimenő koordináták adottak, a továbbhaladási irány 0 lesz, hogy a következő lépésben tetszőleges irányba mozoghassunk, valamint a bejövő irány lényegtelen.

5.4. Általános dimenziós megoldó algoritmus

A harmadik algoritmus az általános dimenziós eset megoldására készült, és nem tartalmazza azt a bonyolult szabályrendszert, ami az előző fejezetben szerepelt.
Az előző változathoz képest nincs szükség a szabályok predikátum dinamikussá tételére, valamint a kapuk hiánya miatt az inicializáló eljárás is elmarad. A szabályok eljárás nagy része eltűnik, hiszen csak a 0-val jelölt alapesetet kell értelmezni. Ezen kívül a lehetséges iránytengelyekre vonatkozó korlátozás alapértéke változik meg, erre most már a –n és az n alsó és felső korlátot tesszük, ahol n az iránytengelyek száma (a koordinátalista hossza). A keresést irányító eljárás sem módosul sokat. Ami viszont teljesen megváltozik, az az új koordinátákat számoló predikátum, valamint ezek után a kezdő paraméterek megadása sem lesz annyira egyértelmű.
Ezekre azért van szükség, mert a koordinátákat ábrázoló lista hossza nem előre meghatározott. A listák elemeinek sorszáma jelképezi a koordinátatengelyeket, az elemek maguk pedig a koordinátákat. Ebből következik, hogy n hosszú lista esetében a lehetséges elmozdulások halmaza {-n, …, n} \ {0}.

```
init1(A):-nth_clause(mező(_, _), 1, Ref),
    clause(mező(L, _), _, Ref), length(L, N), init2(N, A).

init2(0, []):-!.
init2(N, [0|A]):- N1 is N-1, init2(N1, A).
```

5.4.1. Példa: A kezdő koordinátalista inicializálását végző eljárás

Ebben a programváltozatban mindig a [0, 0, …, 0] koordinátájú kiindulópontból indul a keresés. A kezdőpont létrehozását az init1 és az init2 eljárás végzi. Három beépített eljárás segítségével megméri a mező predikátum első klózában első paraméterként szereplő lista hosszát. Az első két beépített eljárás visszaadja az előbbi listát, ennek hosszát a length predikátum adja meg. Ezek után az init2 predikátum létrehoz egy megfelelő hosszúságú, 0-val

feltöltött listát egy egyszerű rekurzió segítségével. A keres eljárás hívásakor második paramétereként (a kezdőkoordináta) ezt a listát adjuk meg.

A második változás az új koordináták számolási módjában történt. A megfelelő eljárás közvetlenül az újkoord néven hívható, aminek a három megszokott argumentuma van: az iránytengely és a kimenő és bemenő koordináták. Ez egy újkoord1 nevű eljárást hív meg, ami tovább bővíti az argumentumok számát egy számláló jellegű változóval, aminek 1 a kezdőértéke.

```
újkoord1(_, _, [], []):- !.
újkoord1(N, A, [X|Y], [Z|W]):-
     N #= A, Z is X + 1, N1 is N + 1, !,
     újkoord1(N1, A, Y, W).
újkoord1(N, A, [X|Y], [Z|W]):-
     N #= -A, Z is X - 1, N1 is N + 1, !,
     újkoord1(N1, A, Y, W).
újkoord1(N, A, [X|Y], [X|W]):-
      N1 is N + 1, újkoord1(N1, A, Y, W).
```

5.4.2. Példa: Egy n dimenziós térben az új koordináták kiszámolására szolgáló eljárás.

Az eljárás rekurzívan végigmegy a bemenő koordinátalistán és átmásolja a kimenő koordinátalistába. A számlálót eközben növeli, és minden lépésben megpróbálja összehasonlítani az elmozdulás iránytengelyével. Ha a számláló egyezik a második paraméter abszolút értékével, akkor az előjelétől függően csökkenti vagy növeli az adott koordinátát. Az kimenő koordinátalistába természetesen az új érték kerül. Az algoritmus akkor áll le, ha a bemenő koordinátalista kiürül.

A keresést irányító útvonal eljárás az előző fejezetben vázoltak szerint működik. Mivel már az előző változat is listaként ábrázolta a koordinátákat, módosításra nem volt szükség. A szabályok eljárás segítségével számolja ki az új koordinátákat (aminek most csak egy klóza van), majd ellenőrzi, hogy van-e mező az általuk meghatározott helyen, és vizsgálja, hogy az adott pontban jártunk-e már. A keresés a megszokott rekurzív hívással fejeződik be. Az első klóz abban módosul, hogy az algoritmus felépítése miatt a legutolsó koordinátát itt kell hozzáfűzni az eredménylistához.

5.5. A végső változat

Az algoritmus végleges változatának nem csak az általános dimenziós esetet kell kezelnie, hanem a 4.2.4-es fejezetben szereplő korlátokat is értelmeznie kell. Ez a bővítés az előfeldolgozási részben jelenik meg.

Mivel nem teszünk megkötést a dimenziók számára, a falakra megadható korlátokat lehetetlenség felsorolni, hiszen számuk állandóan változik. Az egyetlen megoldás, hogy a falakra vonatkozó szabályokat dinamikusan, mindig az adott feladathoz igazítva adjuk meg. Ugyanakkor a falakra vonatkozó megszorítások lényegesen bonyolultabbak, mint a kapukra vonatkozók. Míg az utóbbi esetben két mező egyértelműen összekapcsolódik egymással, falak esetében a továbblépés irányát megadó korlátok bővülnek.

Ennek megfelelően az algoritmus két része változik meg alapvetően: az előfeldolgozó rész és a szabályokat megvalósító rész.

Az inicializáló résznek most már nem csak a kapukra vonatkozó szabályokkal kell bővítenie az algoritmust, hanem a falakra vonatkozó megszorításokkal is. Sajnos nem adható meg egy általános séma arra nézve, hogy az utóbbi megszorítások milyen formát öltenek. Ezért ezeknek a szabályoknak az alkalmazását két részre bontjuk. Először egy egységes, csak a paraméterekben különböző szabállyal bővítjük a szabálybázist, ami egy külön eljárást fog meghívni, amely aztán rekurzívan összegyűjti az adott falra vonatkozó korlátokat.

```
init3:- predicate_property(block(_,_), interpreted), block(A, B),
        assertz(':-'(barrier(A, B, T), T #\= B)), fail.
init3:- mező(_, A), A \= 0, predicate_property(block(_,_), interpreted), block(A, _),
        asserta(':-'(barriers(A, T, L), (barrier(A, B, T), \+ memberchk(B, L), !, barriers(A, T, [B|L])))),
        assertz(':-'(barriers(A, T, _), !)), fail.
init3.

init2:- mező(B, A), A \= 0, predicate_property(block(_,_), interpreted), block(A, _),
        assertz(':-'(szabalyok(A, Etm, B, Z, T), (length(B, N), T in -N..N,
        T #\= 0, T #\= Etm, barriers(A, T, []), indomain(T), újkoord(T, B, Z)))), fail.
init2.
```

5.5.1 Példa: A falak előfeldolgozását végző programrész.

Ezt a feladatot végzi el az init2 és az init3 predikátum. Ezek közül az init2 végzi a szabályok eljárás bővítését. A kapukkal ellentétben itt nem elég egy tényállítással bővíteni a predikátum klózait, szabályokat kell alkalmaznunk. Szerencsére a Prolog nyelv felépítése miatt ezt egyszerűen megtehetjük, hiszen egy Fej :- Törzs alakú szabály felírható ':-'(Fej, Törzs) belső alakban is. Először ellenőriznünk kell, hogy létezik-e egyáltalán fal szabály a feladatban, majd meg kell keresnünk azokat a mezőket, amelyekre ilyen korlátok vonatkoznak. Ezek után egy, a 0-s alapesthez nagyon hasonló klózzal bővítjük a predikátumot. Az egyetlen változás a barriers eljárás hívása, ami a falat jelölő kód alapján összegyűjti az arra vonatkozó megszorításokat. Ezt az eljárást is dinamikusan, az előfeldolgozás során adjuk hozzá az algoritmushoz. Az összes ilyen tulajdonságú mező feldolgozását nem rekurzió, hanem a fentebb is vázolt kényszerített visszalépés, a fail állítás biztosítja.

Ez az init3 feladata: a barrier és a barriers predikátum megvalósítása. Ezek közül az első írja le magukat a korlátokat, a második pedig az ezek gyűjtését valósítja meg.

A barrier esetében a feladatban szereplő minden, a block állítás által leírt korlátozáshoz felveszünk egy ehhez teljesen hasonló szabályt, aminek fejéhez hozzáveszünk egy harmadik paramétert: ez lesz az a változó, aminek az értéket korlátozzuk a megadott iránytengellyel. A törzsébe pedig egy korlát-logikai megszorítást helyezünk, ami leírja a megfelelő szabályt. A megszorítást azonosító kódot azért hagyjuk meg, mert biztosítanunk kell, hogy ugyanazt az állítást nem dolgozzuk fel kétszer.

Ezen szabályok feldolgozását a barriers eljárás végzi, amit a szabályokhoz hasonlóan, az egyes korlátozott mezőkhöz rendelünk. Ez rekurzió segítségével végighalad a fenti predikátum azon klózain, amik az első paraméterben adott kóddal rendelkeznek. Közben egy lista segítségével figyeli, hogy az adott korlátot feldolgozta-e már. A második paraméterként kapja a korlátozni kívánt változót, erre hajtja végre sorban a barrier predikátum megszorításait. Az eljárás második klóza a leállást biztosítja, ha már nincs több feldolgozandó szabály.

Ezekhez a műveletekhez mind a szabályok, mind a barrier és barriers eljárásokat dinamikussá kell tenni.

A program többi része az előző fejezetben leírt módon működik. Záró példaként egy futás közben keletkezett szabályrendszert láthatunk: a barriers predikátum legelső klózát, valamint a barriers és szabályok eljárás b1-es kódú falra vonatkozó korlátait, és a g1 kapura vonatkozó szabályt.

```
szabalyok(b1, C, [0, 0, 0, 0, 0, 0, 0], D, A) :-
    length([0, 0, 0, 0, 0, 0, 0], B),
    A in -B..B,
    A#\=0,
    A#\=C,
    barriers(b1, A, []),
    indomain(A),
    újkoord(A, [0, 0, 0, 0, 0, 0, 0], D).
```

```
barriers(b1, A, C) :-
    barrier(b1, B, A),
    \+ memberchk(B, C), !,
    barriers(b1, A, [B|C]).
barriers(b1, _, _) :- !.
```

```
barrier(b1, 1, A) :-
    A#\=1.
barrier(b1, 3, A) :-
    A#\=3.
barrier(b2, -1, A) :-
    A#\= -1.
barrier(b3, -2, A) :-
    A#\= -2.
barrier(b4, -4, A) :-
    A#\= -4.
```

```
szabalyok(g1, _, [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 1], 0).
```

5.5.2 Példa: Fent baloldalt: a b1 falra vonatkozó szabály, középen: a b1 falra vonatkozó korlát gyűjtő eljárás, jobboldalt: a falakra vonatkozó korlátok. Alul: a g1 kapura vonatkozó szabály.

5.6. A kiíratást végző algoritmus

Meglepő módon a kiíratást végző algoritmus a leghosszabb az eddig ismertetett programok közül, ugyanakkor roppant rugalmas, nem csak a kereső algoritmusokkal képes együttműködni, hanem a pályákat önmagukban is képes kiíratni, illetve megfelelő paraméterekkel a mezők megjelölésére is alkalmas.

A hívása rendkívül egyszerű. Az egyetlen paramétere egy listákból álló lista, ami mezők koordinátáit, illetve egy ezekhez rendelt számot tartalmaz. Alapesetben ezek a számok a megoldási útvonal aktuális lépésének a számát jelölik, de kis fantáziával ezzel a módszerrel bármelyik mezőt megjelölhetjük.

A kiíratás módját három esetre osztjuk: kétdimenziós, háromdimenziós, illetve egyéb esetekre. Utóbbi esetekben egyszerűen a lépések szerinti sorrendben kiíratjuk a mezők koordinátáit. Mivel az algoritmus a fordítva számozza a lépéseket, a legelső mező kapja az n-edik számot (ahol az útvonal hossza n), tehát célszerű a számozást megfordítani. Ehhez szükség van az útvonalhossz eljárásra, ami már definiált, ha kiíratást egy kereső algoritmusból hívjuk, de szükség esetén a program maga is tartalmazza ezt.

A kétdimenziós kiíró algoritmus két egymásba ágyazott rekurzió segítségével halad végig a pálya oszlopain és sorain. A kiindulóponttól lefelé és oldalirányban halad, tehát az y tengely fordítva jelenik meg. Az eljárás során fontos a sorok hosszának, illetve az legelső oszlop

hosszának megállapítása. Ezeket a feladatokat a sorhossz eljárások végzik a findall beépített megoldáskereső eljárás különféle módon paraméterezett változataival. További feladat a sorszámok megfelelő átalakítása. Az ábrázolt pályákon két karakternyi hely jut egy mezőre, tehát az egy karakter hosszú számokat konvertálni kell, egy szóközt kell hozzáfűzni a végekre. A paraméterül megadott listában a keres0 eljárás végzi a megfelelő koordinátához tartozó érték keresését. Ha ez nem sikerül, akkor az „xx” karakterekkel jelöljük az aktuális mezőt.

```
sorkiir(A, A, L):-
     sorhossz(A, B), nvonás2(B),
     sorkiir0(0, B, L, A), nvonás2(B), !.
sorkiir(N, A, L):-
     N < A, sorhossz(N, B), nvonás2(B),
     sorkiir0(0, B, L, N), N1 is N+1, sorkiir(N1, A, L).

sorkiir0(B, B, L, Ek):-
     write('|'), keres0([Ek, B], L, A), !,
     write(A), writeln('|').
sorkiir0(N, B, L, Ek):-
     N < B, write('|'), keres0([Ek, N], L, A), write(A),
     N1 is N+1, sorkiir0(N1, B, L, Ek).
```

6.3.1 Példa: A kétdimenziós kiíratást végző eljárás főbb részei.

A háromdimenziós ábrázolást hasonló módon három egymásba ágyazott rekurzió végzi. Ezek paraméterezésének és hívásának részletezése hosszadalmas lenne, és mivel a dolgozat témájának nem képezi szerves részét, tárgyalására nem kerül sor.

Az eljárás paraméterében megadott listától függően a mezőket különféle módon jelölhetjük. Lehetőségünk van számok és karakterek használatára, azt azonban észben kell tartani, hogy minden mezőre két karakternyi hely jut.

A program nem tökéletes, célja a megadott pályák és az eredményül kapott utak praktikus és egyszerű ábrázolása. A felhasználóbarát ábrázolási módszer kialakítása a dolgozat tárgyát nem képezi, a célra ez a módszer is tökéletesen megfelel.

6. Összefoglaló

A dolgozat során megismerkedhettünk a logikai programozás és a kényszer kielégítési problémák alapjaival, valamint korlát-logikai programozással bővebben is. Ezek tárgyalása során sor került a módszerek elméleti és történeti hátterének ismertetésére, valamint felépítésük és működésük vizsgálatára, megértésüket példák és ábrák segítették.

Ezek után következett a gyakorlati feladatok ismertetése, ezek négy, egymásra épülő változatban jelentek meg. A probléma lényege, hogy egy játékteret kell bejárnunk úgy, hogy minden mezőt érintünk, és pontosan egyszer tesszük ezt. A négy különböző változatban más és más megszorításokat tehetünk a mezőkre, valamint a pálya dimenziója is változik. Így végül a feladat leírására egy rugalmas formalizmust adtunk meg, mely képes kezelni többdimenziós játéktereket, melyekben a mezőkre különböző megszorításokat is tehetünk, tehát a lehetséges pályák megfogalmazásának csak a fantázia szabhat határt.

A megoldó algoritmust is négy, egymásra épülő fejezetben ismertettük, és mindegyikben külön kiemelve szerepelt a korlát-logikai programozás alkalmazásának példája. Ezek bonyolultsága a feladatnak megfelelően változott. Míg az alapfeladatot megoldó program rövidnek és egyszerűnek tűnhet, a végső változat képes n-dimenziós koordinátalisták kezelésére, és a felhasznált szabályrendszert dinamikusan, a feladatnak megfelelően módosítja. Az összes algoritmus a Prolog beépített visszalépéses keresését használja, így képes az összes lehetséges megoldás megtalálására.

Irodalomjegyzék

[1] Szeredi Péter, Benkő Tamás: Deklaratív Programozás,
Budapesti Műszaki és Gazdaságtudományi Egyetem, Oktatási segédlet,
Budapest, 2004.

[2] Szeredi Péter, Benkő Tamás: Nagyhatékonyságú logikai programozás,
Budapesti Műszaki és Gazdaságtudományi Egyetem, Kézirat,
Budapest, 2002.

[3] Ulf Nilsson, Jan Maluszynszki: Logic, Programming and Prolog,
Wiley and Sons Ltd, 2000.

[4] Kovács András: A korlátozás programozás alapjai,
Budapesti Műszaki és Gazdaságtudományi Egyetem, Oktatási segédlet.

[5] Szeredi Péter, Gyimóthy Tibor: Logikai programozás és alkalmazásai,
VIII. Országos Neumann Kongresszus, Előadás kivonat.

[6] Stuart Russel, Peter Norvig: Mesterséges intelligencia modern megközelítésben,
Panem, 2005.

[7] Futó Iván: Mesterséges intelligencia,
Aula, 1999.

[8] Jan Wielemaker: SWI-Prolog 5.6 Reference Manual,
University of Amsterdam, 2006.

Ábrajegyzék