Debreceni Egyetem

Informatikai Kar

Aknakereső rejtvény megoldása kényszer kielégítési módszerek használatával

Témavezető:
Dr. Aszalós László
egyetemi adjunktus

Készítette:
Szabó István
GI Bsc

Debrecen

2010

Tartalomjegyzék

0. *Bevezetés*

Szakdolgozatom témája a kényszer kielégítés módszerének bemutatása a népszerű Aknakereső játék különböző példáin keresztül. A kényszer kielégítés fogalma alatt azt a megoldási folyamatot értjük, amelyben a különböző feltételeknek (kényszereknek) eleget tévő megoldást keressük. A megoldást az a vektor fogja jelenteni, amely minden megadott feltételnek eleget tesz. A kényszer kielégítés magával hordozza a korlát-logikai programozás fogalmát, amely tulajdonképpen a hagyományos logikai programozás kibővítése a korlát kielégítési problémák témaköréből vett elemekkel. A programozási környezet a Prolog és annak CLPFD könyvtára lesz. A dolgozat célja a kényszer kielégítési módszerek alkalmazása az Aknakereső játék különböző alternatíváira. A dolgozatban három különböző játéktáblán, különböző kezdőállapotokból a CLPFD segítségével jutunk el a végállapotokba úgy, hogy nekünk csak a feladatot kell megfogalmaznunk. A végrehajtást rábízzuk a Prologra.

A dolgozat a következőképpen épül fel: Az első fejezet általánosságban mutatja be a Prolog programozási nyelvet. Foglalkozik annak történetével, szintaxisával és fő sajátosságaival. A fejezetet egy példaprogrammal zárjuk.

A második fejezet a kényszer kielégítési problémákkal foglalkozik, a definíció után megismerünk pár példát, feladatot, és azok megoldási folyamatát kényszer kielégítés használatával, majd a végén egy probléma megoldásának főbb lépéseit vizsgáljuk.

A harmadik fejezet a korlát-logikai programozást mutatja be. Foglalkozik a különböző korlátokkal, a CLP szemantikájával, és a CLPFD alkönyvtár is bemutatásra kerül.

A negyedik fejezet az Aknakereső rövid ismertetésével kezdődik. Ebben a fejezetben három különböző kezdőállapotból kiindulva lépésről lépésre haladva oldjuk meg a játékot kényszer kielégítési módszerek és a CLPFD segítségével.

1. A Prolog programozási nyelv

1.1 Programozási nyelvek

A programozási nyelv a számítástechnikában használt olyan, az ember által olvasható és értelmezhető utasítások sorozata, amivel közvetlenül, vagy közvetve, gépi kódra való fordítás után közölhetjük a számítógéppel egy adott feladat elvégzésének módját.
A programozási nyelveknek többféle kategorizálása is van, de alapvetően két csoportot különböztethetünk meg: imperatív, valamint deklaratív nyelveket.
Az imperatív nyelvek csoportjába sorolhatjuk a programozási nyelvek jelentős részét. Talán legfontosabb jellemzőjük, hogy a megoldandó probléma megoldásának lépéseit kell pontosan rögzíteni a kódban. Értjük ezalatt azt, hogy különböző parancsok, eljárások és függvények segítségével adjuk meg a feladat teljesítésének menetét. Tipikus imperatív nyelv az assembly, Java, C és C++, vagy például a Python és Matlab.

Az imperatív stílussal ellentétben a deklaratív stílusban programozónak – elvileg – csak azt kell megmondania, hogy mit akar; az algoritmust az értelmező– vagy fordítóprogram állítja elő. A deklaratív programozás legtöbb esetben valamilyen matematikai formalizmusra épül. Két válfaját szokás megkülönböztetni: a logikai és a funkcionális programozást.
Míg egy imperatív nyelvben megírt program futásának minden lépését átlátjuk, addig egy deklaratív nyelven írt kód esetén, mivel egyes lépések sokszor összetettek, nem mindig tudjuk a program futásának minden elemi lépését követni. Ez általában nem szokott problémát okozni, mivel a programozó célja a feladat megfogalmazása.

A reláció fogalmára épülő paradigmákat logikai programozásnak, míg a függvényfogalomra épülőket funkcionális programozásnak nevezzük.
A funkcionális nyelvek között az első a LISP (List Processing) programozási nyelvcsalád volt, amelyet 1958-ban alkották meg. Ezt később számos hasonló nyelv követte.
A legelterjedtebb logikai nyelv a Prolog, valamint az egyre nagyobb tért hódító Prolog korlát alapú bővítményei, a CLP (Constraint Logical Programming) rendszerek.

1.2 A Prolog nyelv története

A Prolog kifejlesztése Alain Colmerauer és Bob Kowalski számítógépes nyelvészek nevéhez fűződik. 1972-ben, Marseille-ben, az ő munkásságukra támaszkodva elkészítettek egy interpretert, amely az ún. Horn-klózokra épülő, rezolúciós tételbizonyítót tartalmazó programozási nyelv megvalósítása volt és amelyet PROLOG-nak (PROgramming in LOGic) neveztek el. Ez az első magas szintű logikai programozási nyelvnek tekinthető. 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. Az ő nevéhez fűződik a Prolog hatékony megvalósítási módszereinek kidolgozása, 1977-ben elkészítette a nyelv első fordítóprogramját.

Magyarországon 1974-ben, egy angliai tanulmányút egyik eredményeképpen jelent meg a nyelv. A NIM IGÜSZI-be behozták a PROLOG leírását és *Ecsedi Tóth Péter* erről egy szemináriumot is tartott, még ebben az évben elindult az első Prolog tanfolyam, és elkészült az első alkalmazás is. Jelentős nemzetközi visszhangot váltott ki az ekkor megírt magyar lakástervező, vagy egy, a gyógyszerek kölcsönhatásait kimutató program, és egyéb szakértői rendszerek, ugyanis ezek voltak az első gyakorlati alkalmazásai a nyelvnek. Az igazi matematikusok számára a PROLOG tartalmazott eretnek elemeket is, nevezetesen az úgynevezett beépített eljárásokat, melyek a programozás gyakorlati oldalait voltak hivatottak támogatni, mint például beolvasás, kiírás, vezérlés, numerikus eljárások. Ezek az eljárások azért nem illeszkedtek a logika szemléletébe, mert végrehajtásuk sorrendfüggő. Nem lehet pl. egy összeadást végző eljárást akkor meghívni, amikor a megfelelő argumentumai még nem tartalmaznak értéket. A PROLOG-szemináriumot követően, *Szeredi Péter* 1975 első felében CDL-ben megírt egy PROLOG-interpretert, amely gyakorlatilag megfelelt a Marseille-i Egyetemen készült PROLOG-interpreternek. Az interpreter 20 logikai következtetést végzett másodpercenként az ICL 1903A-n, amely akkor Magyarország egyik legnagyobb teljesítményû számítógépe volt. [3]

Az ezt követő húsz évben a hazai PROLOG- fejlesztések két nagy ágra bomlanak: az MProlog moduláris PROLOG rendszerre és alkalmazásaira, valamint a későbbi sokprocesszoros PROLOG rendszerre és alkalmazásaira. Utóbbinak alapvetően két új tulajdonsága van: lehetővé tette több virtuális Prolog alkalmazás egyidejű futását, és

üzeneteken keresztüli kommunikációját, másrészt bevezették a szimulációs időt, mint fogalmat, ami lehetővé tette a szimulált időben történő előre és visszalépkedést. Így a szimulált időben lehetővé vált az időutazás, melynek segítségével a szimulált jövőből a modell információt tárolt, melyet a jelenben is fel tudott használni a visszalépés után.
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. [3]

A mai napig számos fejlesztés és alkalmazás készült a különböző Prolog környezetekben, az eredmények ma már ritkábban jelennek meg önállóan, inkább beépülnek más gyakran használt eszközkészletekbe és programnyelvekbe.
A Prologot gyakran használják mesterséges intelligencia-alkalmazások megvalósítására, illetve a számítógépes nyelvészet eszközeként (különös tekintettel a természetes nyelvfeldolgozásra, melyre eredetileg tervezték). [3]

1.3 A Prolog fő sajátosságai és szintaxisa

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 Horn-klózokra szűkítjük le, és a programok futásához egy nagyon egyszerű tételbizonyítási módszert használunk, az ún. SLD rezolúciót.

A Horn klózok

$$A \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

alakú implikációk, ahol a, C1, .., Cn elemi állítások és az összes előforduló változót univerzális kvantorral lekötöttnek tekintjük.

A Prolog fő sajátosságai a következők:

- A Prologban tények halmaza van, amelyek a valós világ objektumaira vonatkoznak, valamint szabályok, amelyek ezen objektumokra vonatkoznak.
- A program végrehajtását a részproblémák megoldásához szükséges információk befolyásolják, és nem a program utasításainak sorrendje.
- A Prolog program egy kérdéssel indul, amely az információszerzés elsődleges kiinduló pontja.

A Prolog rekurzív logikai formulákat kezelni képes rendszer. Egy kód mondatok sorozatából áll, amelyeket pont zár le. Ezeket a mondatok állításoknak nevezzük. Minden mondatot egy logikai állításnak vagy formulának tekintünk, amelyhez tartozik egy igazságérték. Ez az érték igaz vagy hamis lehet.
A Prolog alkalmazza a hagyományos informatikai értelemben vett adattípusokat is, mint például az egészek vagy a listák, de nagyobb szerepet kapnak a lexikai elemek. A Prologban mind az adatokat, mind a végrehajtandó programot termek formájában írjuk le. Ez a nyelv egyetlen, összetett adatokat megjeleníteni képes struktúrája. Ezek tekintendőek a Prolog programozási nyelv szavainak.
Egy összetettebb Prolog kifejezés egy névből, és a névhez rendelt adott számú egyszerűbb kifejezésből, argumentumból áll. A Prolog kifejezések osztályozása során egy meglehetősen hierarchikus szerkezetet kapunk. Egy kifejezés lehet változó, konstans, vagy összetett kifejezés. A változók mindig nagy betűvel kezdődnek, a kis betűvel írottak a konstansokat (atomic) jelölik. Konstanst kezdhetünk nagy betűvel is, és használhatunk ékezeteket, ha a nevet idézőjelek közé tesszük. Egy konstans lehet névkonstans (atom), vagy számkonstans (number). Egy számkonstans lehet egész (integer), vagy lebegőpontos szám (float).
Egy Prolog program a következő osztályokból épül fel: tények, szabályok, célállítás. [5]

Tények

A feltétel nélküli Prolog állításokat tényállításoknak, tényeknek nevezzük. A tények olyan egyszerű állítások, amelyek a vizsgált világ dolgai között fennálló kapcsolatokat, összefüggéseket írják le. A tény a predikátum nevével kezdődik, ezt a zárójelbe tett argumentumok követik, bennük azokkal az objektumokkal, amelyek között a kapcsolat fennáll. A tényt pont zárja le.

Tények például a következők:

```
egyenlo(A,A).
```

A egyenlő A-val.

```
etel(kenyer).
```

A kenyér étel.

Szabályok

A logikai következtetés formájú Prolog állításokat szabályoknak nevezzük. Ezek a szabályok kisbetűvel kezdődő állításból, vagy konstansokból, változókból épülnek fel, pont zárja őket. Ezek az állítások objektumai, más néven tagjai. Egy szabályban az első tag viszonyban van a másodikkal, azonban ez fordítva nem teljesül. A tagok, objektumok számát nevezzük a kijelentések argumentumának. A konstansokat 0 argumentumú kifejezésnek tekintjük. A szabályok mindig a predikátum fejjel kezdődnek, teljesülésük a feltételektől függ. A feltételek az `if` kulcsszó vagy a vele megegyező :- jel után állnak, több szabály esetén kötelező a logikai kapcsolat megjelölése. Ilyen jelek még a „`vagy`”(pontosvessző), „`és`”(vessző), „`not`” szavak.

Példa szabályra:

```
auto(A) :- jarmu(A) , negykereku(A).
```

Jelentése: A akkor autó, ha jármű és négy kereke van.

Célállítás

A Prologban egy feladat kitűzése egy célállítás megfogalmazását jelenti. A program végrehajtása tulajdonképpen egy célállítás megfogalmazása. A célállításokat a részcélok igazolásával igazolhatjuk.

Példa célállításra:

```
?- anyja(X,”Judit”) , anyja(X,”Gergő”).
```

Jelentése: van-e olyan X személy, aki Judit és Gergő anyja, tehát Gergő és Judit testvérek-e?

1.4 Prolog példa

Az elméleti ismertetés után nézzük meg egy Prolog forráskód felépítését!

```
futtat :-
     write(’Adja meg a hőmérsékletet Celsius fokban '),
     read(C),
     atvalt(C,F),
     write(’A hőmérséklet '), write(F), write(' Fahrenheit’),
     nl,
     figyelmeztet(F, figyelmeztet),
     write(figyelmeztet).
     atvalt(Cel, Fahr) :-
               Fahr is 9.0 / 5.0 * Cel + 32.
figyelmeztet (T, 'Vigyázz, igazi hőség van’) :- T > 90.
figyelmeztet (T,’Vigyázz, nagyon hideg van!') :- T < 30.
figyelmeztet (T,  ”) :- T >= 30, T =< 90.
```

1.ábra: Prolog program, amely a Celsiusban megadott hőmérsékletet váltja át Fahrenheit fokra.

Ez egy egyszerű példa, de a Prolog felépítésének szemléltetésére megfelelő. A program a `futtat, atvalt, figyelmeztet` predikátumokból épül fel. A futtat predikátum kivételével mindegyik két argumentumból áll. Az `atvalt` predikátum bemenő argumentumként megkapja a `C` számot, mint input változót, és a kiszámítási képlet elvégzése után vissza adja az `F` számot (output változó). A Prolog nem típusos nyelv, így elméletileg a predikátumok paramétereiként bármilyen Prolog kifejezést megadhatnánk, ugyanakkor a legtöbb predikátum csak a programozó által tervezett típusú változókra működik.

A Prolog számtalan beépített eljárással rendelkezik, ilyen például a `read`, ami beolvassa az átváltani kívánt számot, vagy a `write`, amellyel szöveget és az eredményt íratjuk ki a képernyőre. Az `nl` eljárás soremelést végez. A Prolog matematikai logikai nyelv, ezért iterációt nem, ellenben beépített matematikai eljárásokat szép számban tartalmaz, de természetesen az egyszerű matematikai jeleket, képleteket is kezeli. Az iterációt a programozó a rekurzió használatával helyettesíti.

2. A kényszer kielégítés módszere

2.1 Definíció

A kényszer kielégítés, angolul Constraint Satisfaction a mesterséges intelligenciában és az operációkutatásban is használt fogalom. Ez egy olyan megoldási folyamat, amelyben a program különböző feltételeket állít a változókkal szemben. A megoldás tehát egy olyan vektorváltozó, amely eleget tesz az állított feltételeknek. A kényszer kielégítési probléma (Constrait Satisfaction Problem – CSP) egy olyan speciális probléma, ami a problémákkal szemben általánosan megfogalmazott alapkövetelmények mellett néhány strukturális tulajdonságnak is eleget tesz.

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. Teljes az a hozzárendelés, amelyben mindegyik változó szerepel, és egy teljes hozzárendelés a kényszer kielégítési problémának megoldása, ha mindegyik kényszert kielégíti. Egy hozzárendelést konzisztensnek nevezünk, ha egyetlen korlátot sem sért meg.
Néhány kényszer kielégítési probléma azt is igényli, hogy a megoldás egy célfüggvényt maximalizáljon.[6]

2.2 Kényszer kielégítési problémák

Egy kényszer kielégítési problémában az állapotokat egy változók (variables) halmaz értékei határozzák meg, és a célfüggvény a kényszerek (constrains) egy halmazát adja meg, amelyeknek teljesülniük kell. Például a 8-királynő-problémát tekinthetjük egy kényszer kielégítési problémának, amiben a változók a nyolc királynő pozíciói, a változók lehetséges értékei a sakktábla mezői, és a kényszerek azt fogalmazzák meg, hogy két királynő nem lehet azonos sorban, oszlopban, vagy átlóban. A probléma megoldása minden változóhoz úgy rendel értéket, hogy azok kielégítsék a kényszereket. Számos tervezési és ütemezési feladatot is meg lehet fogalmazni CSP-ként, így azok nagyon fontos alosztályt alkotnak. A kényszer kielégítési problémákat meg lehet oldani általános célú problémamegoldó algoritmusokkal is, azonban speciális felépítésükből adódóan a kifejezetten CSP-k megoldására tervezett algoritmusok lényegesebben jobb teljesítményt nyújtanak.

A kényszerek számos formában jelentkezhetnek. Az unáris kényszerek egyetlen változó értékére vezetnek be korlátozásokat. A bináris kényszerek változó párokat kapcsolnak össze. A 8-királynő-probléma összes kényszere bináris kényszer. A magasabb rendű kényszerek három vagy több változót kötnek össze – például egy betűaritmetikai problémában az oszlopok változói összeadási kényszerben állnak, amelyben számos változó szerepelhet.[6]

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

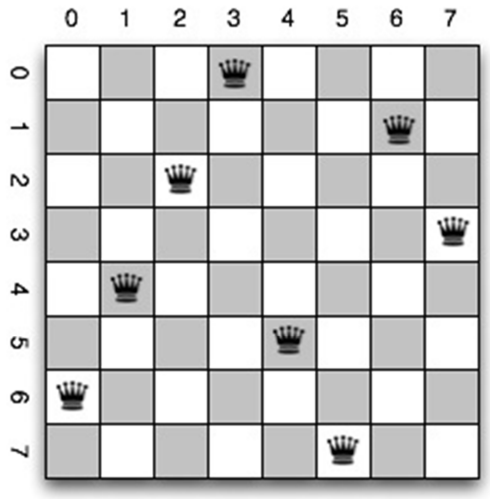
2. ábra: Betűaritmetikai példa: A Send More Money probléma

A magasabb rendű kényszerek legismertebb példája a Send More Money 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. A megszorításokat helyi érték szerint szabjuk ki a

betűkre, és majd a programunk ezen megszorítások alapján számolja ki a megoldást. A megoldó programmal később foglalkozunk.

Végül a kényszerek lehetnek abszolút kényszerek, amelyek kielégítése elengedhetetlen a megoldás létéhez, vagy preferenciális kényszerek, amelyek megmondják, hogy mely megoldásokat részesítünk előnyben.

Egy kényszer kielégítési probléma minden egyes V_i változójához tartozik egy D_i értékkészlet (domain), ami nem más, mint a változó által felvehető értékek halmaza. Az értékkészlet diszkrét vagy folytonos lehet. Egy autó tervezése során például változót rendelhetünk az alkatrészek tömegéhez (folytonos) és az alkatrészek gyártóihoz is (diszkrét). Egy unáris kényszer az értékkészlet megengedett részhalmazát adja meg, míg egy két változón megfogalmazott bináris kényszer a két értékkészlet keresztszorzatának megengedett részhalmazát specifikálja. Diszkrét CSP-k esetén, ahol az értékkészletek végesek, a kényszereket egyszerűen a megengedett értékkombinációk felsorolásával is reprezentálhatjuk.

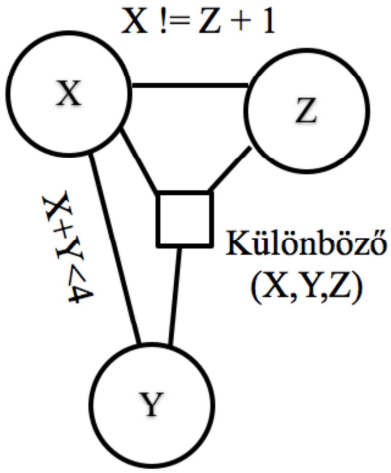


3.ábra: A 8-királynő-probléma egy megoldása

Például a 8-királynő-problémában a V_1 jelölje az első királynő első oszlopban elfoglalt sorát, V_2 jelölje a második királynő második oszlopban elfoglalt sorát. A V_1 és V_2 változók értékkészlete *{0,1, 2, 3, 4, 5, 6, 7}*. A V_1 és V_2 változót összekötő „nem-támad” kényszert le lehet írni V_1 és V_2 megengedett értékpárjaival: *{ (0, 2), (0, 3), (0, 4), … (1, 3), (1, 4), …}* és így tovább. A „nem-támad” kényszer a 64 kombinációból összesen 22 kombinációt töröl. Ezt a

felsorolási megközelítést követve az összes diszkrét kényszer kielégítési problémát át lehet alakítani binárissá. Fontos a változók megfelelő megválasztása, ugyanis ezek érték-hozzárendelési sorrendjének, illetve a megfelelő értékek kiválasztásának körültekintő megválasztásával gyakran sokkal jobb eredményeket lehet elérni.

A következő példa bemutatja, hogyan működik egy, a kényszer kielégítés módszerét használó program. A feladat egyszerű: adott változókhoz, értékkészleteikhez és a változókon értelmezett kényszerfeltételekhez keresünk a kényszerfeltételeket kielégítő változó hozzárendelést.



	X	Y	Z
Kiindulás	{1,2,3,4,5}	{1,2,3,4,5}	{1,2,3,4}
X+Y<4	{1,2}	{1,2}	{1,2,3,4}
Különböző	{1,2}	{1,2}	{3,4}
Választás	{1,2}	{1}	{3,4}
Különböző	{2}	{1}	{3,4}
Választás	{2}	{1}	{3}

4.ábra: Egy egyszerű kényszer kielégítési probléma gráfmegjelenítése

Az ábrán egy egyszerű kényszer kielégítési probléma gráfmegjelenítése látható: a probléma három változóból (*X*, *Y*, *Z*) és három kényszerből (X+Y<4, X!=Z+1 és mindhárom változó értéke különböző) áll. A változók értékkészleteit az ábra jobb oldalán látható táblázat első sora tartalmazza.[8]

A kényszer kielégítési problémák megoldásának gyakori módszere, hogy a változók értékkészleteit a kényszerek figyelembe vételével szűkítjük az összes lehetséges megoldás megőrzésével. Ha a szűkítés során egy változó értékkészlete üres halmaz lesz, akkor a probléma nem kielégíthető. Például az X+Y<4 kényszer miatt nem lehetséges, hogy X vagy Y lehetséges tartományából bármilyen értéket felvehessen, ezt a módosítást a táblázat második sora tartalmazza. A következő kényszer az, hogy a változók között ne legyenek egyformák.

Ez ismét szűkíti a változók értékkészletét. A harmadik kényszer (X!=Z+1) miatt a megoldás következő lépésében szűkíti a még választható lehetséges értékeket.
Szűkítéssel nem mindig lehet végeredményhez jutni – ilyenkor egy visszalépéses keresést szokás alkalmazni: egy kiválasztott változó értékkészletét csökkentve próbálunk megoldást keresni (*Választás* sorok a táblázatban). Sikertelenség esetén visszalépünk, és így az összes lehetőséget megvizsgáljuk. A megoldó folyamat végén megkapunk egy lehetséges megoldást.

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 elemei leképezhetőek a megfelelő értéktartományokra.
- 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.[7]

3. A CLP bemutatása

3.1 A korlát logikai programozás

A korlátozás programozás (Constraint Programming / Constraint Logic Programming, CLP) egy deklaratív programozási paradigma kombinatorikus (optimalizálási) problémák megoldására. Hatékonyságának, rugalmasságának és az egyszerű formalizmusoknak köszönhetően számos gyakorlati alkalmazással büszkélkedhet, pl. az ütemezés, órarendkészítés, grafikai alkalmazások, stb. területén. 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. Ez egy általános séma, amely egy szűkebb tartományon, meghatározott alakú relációk esetén, a Prologhoz képest bonyolultabb következtetéseket enged megvalósítani.

Például a CLP(Q) korlát-nyelv esetén:

- a tartomány: a racionális számok halmaza,
- a megengedett korlátok (constraints): lineáris egyenletek és egyenlőtlenségek,
- a következtetés: a korlátok egyszerűsítése és megoldása, azaz egy lineáris egyenletekre és egyenlőtlenségekre vonatkozó megoldó algoritmus.

```
> Sol=[S,E,N,D,M,O,R,Y],
    domain([E,N,D,O,R,Y],0,9), domain([S,M],1,9),
        1000*S + 100*E + 10*N + D +
        1000*M + 100*O + 10*R + E #=
        10000*M + 1000*O + 100*N + 10*E + Y,
    all_different(Sol),
    labeling([ff],Sol).
> Sol = [9,5,6,7,1,0,8,2]
```

5.ábra: A Send More Money megoldása

A 2.2 fejezetben bemutatott betűrejtvény Prologban írt, kényszer kielégítési módszereket alkalmazó kódja látható a fenti ábrán.

A program felépítése a következő: elsőként megadjuk a változókat, amelyeknek megoldásként értéket kell kapniuk. Következő lépésként megadjuk a változók értékkészletét. Látszik hogy a soronként legnagyobb helyi értékű betűk értékkészlete szűkebb, mivel egy szám nem kezdődhet nullával. A fő feltétel maga az összeadás. A szavakat helyettesítő számok úgy épülnek fel, hogy a megfelelő betűket vesszük a saját helyi értékükön, és összeadjuk őket. Fontos kikötés, hogy a számok különböznek egymástól, ezt köti ki az `all_different(Sol)` korlát. Minden lehetséges megoldásra kíváncsiak vagyunk, ezért használjuk a `labeling([ff],Sol)` predikátumot. Végül a program kiírja a megoldást.

A korlát-logikai programozás általános sémáját CLP(X)-szel jelölik. Itt X-ként valójában három dolgot kell megadni: egy tartományt, amelyből a korlátváltozók az értéküket vehetik, az adott tartományon megengedett korlátokat és egy következtetési módszert. Tehát a CLP valójában egy nyelvcsalád, amelynek példányai különböző tartományokon, különböző korlátfajtákra különböző következtetéseket valósítanak meg. Ami közös a nyelvcsalád elemeiben, az az ún. korlát-tár fogalmán alapuló végrehajtási mechanizmus. A korláttár a program által a végrehajtás egy adott pillanatáig összegyűjtött korlátok halmaza (azaz konjunkciója). A CLP(X) séma előírja, hogy a rendszernek mindenképpen biztosítania kell a korlát tár konzisztens voltát: ha a felhalmozott korlátok együtt ellentmondásosak, ezt a rendszernek észre kell vennie.

CLP(X) nyelvcsalád leggyakrabban használt tagjai a következők. A CLP(Q) séma használatakor megadjuk a tartományt, a korlátokat és használni kívánt következtetési módszert. A CLP(X) séma egy másik speciális esete a CLP(B), amely az igazságértékek kételemű tartományán az ítéletkalkulus állításai, mint korlátok között biztosít egy teljes következtetőt, amely a mesterséges intelligencia SAT (propositional SATisfiability) részterületéről származik. A CLP(FD) változat viszont egy véges halmazt enged meg tartományként (FD = Finite Domain, általában egész értékek egy véges halmaza), és különböző aritmetikai, logikai és kombinatorikai korlátokon következtet, a mesterséges intelligencia egy másik, ún. CSP tudományterületéről származó módszerek segítségével. A Prolog rendszerek esetén a korlát-kezelők általában opcionálisan betölthető könyvtárakként állnak rendelkezésre.

3.2 Korlátok

Egy c x_1 ,…x_k változókra értelmezett korlát formális definíciója szerint egy k változós reláció a $D_1 \times ... \times D_k$ direkt-szorzat felett, amely a változók megengedett kombinációira áll fenn.

Prolog korlátok jelölése eltér a megszokottaktól, például az A>B relációt `A #>B` korlátként írjuk fel. Az `(A#=3 \/ B#> C-2)` korláton az (A=3 vagy B>C-2) feltételt értjük. Ilyen korlát például az `all_different(L)`, amely kiköti, hogy az L lista elemei különbözőek.

A fenti k tetszőleges értéket felvehet, de kiemelünk két esetet:

Unáris korlátok (k = 1) . Egy c_u unáris korlát arra használható, hogy egy x_i változó értékkészletéből eltávolítsuk a nemkívánatos értékeket. Pl. az $x_i \neq 3$ korlát esetén a 3 értéket eltávolíthatjuk D_i-ből. Ettől a pillanattól kezdve a c_u korlát biztosan ki van elégítve, és akár el is távolítható C-ből.[4]

Bináris korlátok (k = 2) . Ha egy CSP minden korlátja bináris, akkor bináris CSP-ről beszélhetünk. Egy bináris CSP ábrázolható egy gráffal, melynek csúcsai a változók, élei a megfelelő változópárok közt fennálló korlátok. Elméleti jelentősége van annak, hogy tetszőleges CSP átalakítható bináris CSP-vé, bár ez új változók bevezetését teheti szükségessé. Ma ezt a technikát már nem használják, hatékonyabb algoritmusok ismertek magasabb fokszámú korlátokat is tartalmazó CSP-k megoldására.

Minden CLP séma egy adattartományon értelmezett korlátokra vonatkozó hatékony következtetési mechanizmus. Az adattartomány különféle megválasztásaiból más-más CLP sémák adódnak.[4]

Logikai korlátok : A legtöbb CLP rendszer rendelkezik logikai korlátokat leíró könyvtárral, ilyen például a CLP(BIN). 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.[7]

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. Ilyen könyvtár például a CLP(Q).[7]

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. Egész számokkal dolgozó CLP könyvtár a CLP(FD), amelyet az Aknakereső feladat megoldásához is alkalmazunk később. A dolgozatban ezzel a könyvtárral részletesen is foglalkozunk.[7]

3.3 CLP szemantikája

A korlát logikai programokat az különbözteti meg a hagyományos logikai programoktól, hogy a logikai kijelentéseket konkrét alaphalmazon megfogalmazott korlátokkal egészítjük ki, és ezeket külön megoldó algoritmusokkal oldjuk meg. Ennek tükrében egy CLP program két részből épül fel: első lépésként megfogalmazhatunk állításokat és korlátokat, majd ezután állítjuk fel a korlát-logikai program klózok egy véges halmazát,

$$X_0 \leftarrow Y_1,\ldots,Y_m , X_1,\ldots,X_n \qquad (m,n\geq 0)$$

Ahol Y_i az adott alaphalmaz változóiból, konstansaiból, függvényeiből és relációiból felépíttet kifejezések, X_i -k pedig hagyományos Prolog kifejezések. A lefutás során figyelembe kell venni az eddig vizsgált összes korlátot.
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 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. Ezzel a korlát-tár mérete csökkenthető, egy kisebb és egyszerűbb korlát-tár konzisztencia vizsgálata pedig sokkal kevesebb időt vesz igénybe.[1]

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 fenti műveletek valamelyikét mindig a levezetés közbeni aktuális állapoton hajtjuk végre. 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 korláttár 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.

3.4 Az SWI-Prolog CLPFD könyvtára

Az SWI-Prolog fejlesztői környezetnek többek között 2 korlát logikai könyvtára is van, ezek a CLPFD és a CLPQR. A CLPFD az egész számok tartományában értelmezhető, míg a CLPQR a racionális számokat is kezeli. Az általunk kitűzött feladat megoldása során az előbbire van szükségünk, így ezt mutatjuk be részletesebben. A CLPFD a Constraint Logic Programming over Finite Domains fogunk megismerkedni. Ezt a könyvtárat előszeretettel használják a különböző kombinatorikai problémák megoldására, például tervezésre, ütemezésre, logikai és

elosztási feladatok megoldására. A könyvtár legtöbb állítmánya véges tartományi kényszer az egész számok tartományában. Támogatja a klasszikus aritmetikai műveleteket, valamint a minimum, a maximum, a moduló és az abszolút érték függvényeket is.

A legfontosabb véges tartományi kényszerek:

`Kif1 #>= Kif2`	*`Kif1` nagyobb, vagy egyenlő, mint `Kif2`*
`Kif1 #=< Kif2`	*`Kif1` kisebb, vagy egyenlő, mint `Kif2`*
`Kif1 #= Kif2`	*`Kif1` egyenlő `Kif2`-vel*
`Kif1 #\= Kif2`	*`Kif1` nem egyenlő `Kif2`-vel*
`Kif1 #> Kif2`	*`Kif1` szigorúan nagyobb, mint `Kif2`*
`Kif1 #< Kif2`	*`Kif1` szigorúan kisebb, mint `Kif2`*

A Prologtól eltér a az egyenlőség és egyenlőtlenség relációk írásmódja annyiban, hogy az operátorok elé # jel kerül CLP könyvtár használata esetén.
A Prolog nyelv esetén a CLP(FD) könyvtárat a következő parancs segítségével használhatjuk:

```
[library(clpfd)].
```

A clp/bounds egy egyszerű egész korlát-megoldó, a CLPFD szintaxisának egy részét valósítja meg. A következő függvényeket és relációkat támogatja:
Értékkorlátozást tudunk megadni a változókra és azok listáira az „in” korlát segítségével. A baloldalon szerepelnek a változók, a jobboldalon pedig egy olyan kifejezés, amelyben szerepel a tartomány legkisebb és legnagyobb felvehető értéke. Ez a korlát tehát a változókhoz egy intervallumot rendel.

Például:

`[A,B] ins 0..5.` Jelentése: A és B értékkészlete : {0,1,2,3,4,5}

Az indomain/1 korlát ahhoz a változóhoz rendel egy értéket a lehetséges értékek közül, amelyet az argumentumában megadtunk, valamint ha visszalépés történt, akkor más értékekkel próbálkozik.

Hasonlóan fontos korlát a label/1, amely listákhoz rendel elemeket a változók értéktartományából, szisztematikusan.
A call_residue_vars/2 és a copy_term/3 arra használható, hogy megvizsgálja a még fennmaradt célokat, és azokat a kényszereket amihez egy változónak köze van.
Az fd_dom/2 vagy az fd_size/2 predikátumok akkor lehetnek hasznosak, ha saját címkéző stratégiánkat szeretnénk megvalósítani.
Erre a stratégiára akkor lesz szükségünk, ha a problémáknak a korlátok alapján nincs egyértelmű megoldása, vagy ezt a rendszer nem tudja kikövetkeztetni. Ekkor a változókat el kell kezdenünk szisztematikusan az 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.[9]

4. A feladat

4.1 Az Aknakereső bemutatása

Az Aknakereső egy számítógépes logikai játék, melynek célja a mezőn lévő összes akna megtalálása, illetve azok elkerülése.
Egy egyforma mezőkre osztott táblával indul a játék, ezek alatt rejtőzködnek az aknák. A tábla mérete és az aknák száma a nehézségi szinttől függően változik. A mezők állapotai a következők lehetnek:

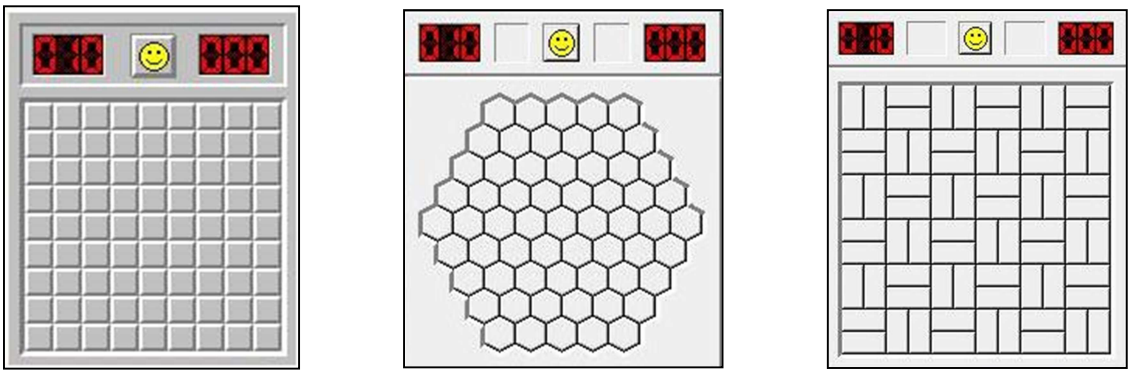
- lefedett (alaphelyzet)
- feltárt, szomszédos aknával
- feltárt aknamentes
- zászlós (véleményünk szerint akna van alatta)
- kérdőjeles (lehetséges, hogy akna van alatta)

Az utolsó két állapot az egér jobb gombjával érhető el, csupán segítséget nyújt a játékhoz. Egy mezőt feltárni kattintással lehet. Ha aknára kattintottunk, a játék véget ér. Ha egy mező feltárult, és mellette akna található, akkor annak darabszámát egy számmal fogja jelezni. Egy

mező mellett a hagyományos játéktéren maximum nyolc akna lehet, hexagonális esetben hat, „parketta” típusú játéktéren pedig hét. A program folyamatosan jelzi a még meglévő aknák számát, illetve az eltelt időt.
A játék célja: teljesíteni a táblát a lehető legrövidebb idő alatt.

4.2 A feladat megfogalmazása

A feladat megfogalmazásakor három különböző játéktér megoldása volt a cél. Az alapértelmezett *10x10* nagyságú négyzet alakú, a hexagonális és az ún. parketta típusú játéktereken jutunk majd el a célállapotokba. Minden esetben tíz bomba van véletlenszerűen elrejtve a játékban, a cél az összes üres mező felfedése.



6.ábra: Négyzet, hatszög és téglalap alapú játékterek

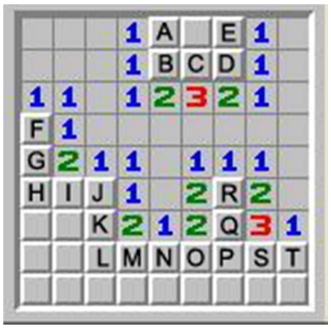
A feladatot a Prolog programozási nyelv és CLPFD könyvtára segítségével fogjuk megoldani. Az Aknakereső játékot tekinthetjük egy kényszer kielégítési problémának. A játék kezdetekor teljesen véletlenszerű az első mező kiválasztása, az állapotokat az első kattintás utáni állapottól kezdjük vizsgálni. A kényszereket a változók és az információ tartalmú mezők relációjából generáljuk. A megtalált bombákat az ábrákon zászlóval jelöljük. A célállapotnak azt az állapotot adjuk meg, mikor mind a tíz bomba helye ismert, és nincs a táblán rejtett mező. Minden egyes állapotot megvizsgálunk, a konkrét feladatot megfogalmazzuk a Prolog nyelvén, és a futtatás után a visszaadott értékeket, amelyek a megjelölt változókhoz vannak rendelve felszabadítjuk.

Egy változó értéke *0* vagy *1* lehet, *0* érték jelöli azt, hogy a mezőn nincs bomba, az *1* érték jelöli a bombát.
Mielőtt hozzálátunk a játék megoldásához, a Prologban meg kell hívnunk a CLPFD alkönyvtárat:

```
?- [library(clpfd)].
```

4.3 A négyzet alapú tábla megoldása

A hagyományos Aknakereső játék játéktere a négyzet alapú tábla. A tábla mérete 10x10 mező, a bombák száma tíz. A kezdőállapot a következő:



7.ábra: Egy lehetséges kezdőállapot

A változókat az ABC nagy betűivel jelöltük, A-tól T-ig. Azokkal a rejtett mezőkkel, amelyekről nincs információnk, tehát nem érintkezik számmal, nem foglalkozunk még. A feladat az, hogy a rendelkezésre álló adatokból megfogalmazzuk a feladatot, amelyet a Prolog végre fog hajtani. A korlátokat a változók és a mezők melletti információk segítségével fogalmazzuk meg. A fenti ábra alapján az első állítás a következő:

```
1 ?- [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T] ins 0..1,
A+B#=1, B#=1, B+C#=2, B+C+D#=3, C+D#=2, D#=1, D+E#=1, F#=1,
F+G+H+I+J#=2, J#=1, J+K+L+M+N#=2, M+N+O#=1, N+O+P+Q+R#=2,
Q+R#=2, P+Q+R+S+T#=3, S+T#=1.
```

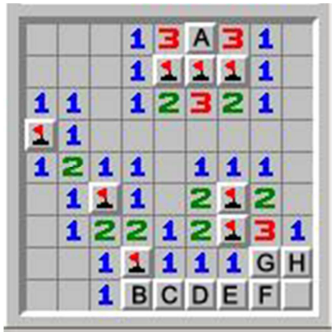
Az `[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T] ins 0..1,` részben felvesszük a változókat `A`-tól `T`-ig, és az `ins` predikátummal megadjuk, hogy az értékük csak *0*, vagy *1* lehet. Csak azt vizsgáljuk hogy az adott mezőn van-e bomba. Az ábrán látszik, hogy az `A` melletti ismert mező értéke *1* és az `A` és `B` mezőkkel érintkezik, tehát `A` vagy `B` mezők közül az egyik bomba. Ha haladunk tovább a következő ismert információs mezőre, akkor láthatjuk, hogy az érték *1*, és csak a `B` mezővel érintkezik, ezért a `B` értéke *1* lesz. A következő információnk az hogy a `B` és `C` mezők összege *2*, valamint a `B, C, D` mezők összege *3*. Mezőről mezőre haladva írjuk fel ezeket a korlátokat, egészen a `T` mezőig. Fontos, hogy a kifejezéseket egymástól vesszővel választjuk el, az utolsó kifejezés végére pedig pontot teszünk, ezzel zárjuk az állítást.

A Prolog által visszadott értékek:

```
A = 0, B = 1, C = 1, D = 1, E = 0, F = 1, G = 0, H = 0,
I = 0, J = 1, K = 0, L = 0, M = 1, N = 0, O = 0, P = 0,
Q = 1, R = 1, S in 0..1, S+T#=1, T in 0..1.
```

Ha helyesen fogalmaztuk meg a feladatot, akkor a Prolog kiszámolja, és kiírja a képernyőre a megoldást. Amelyik változó értékét meg tudta határozni, azokkal kezdi a kiíratást. Azon változóknak, amelyeknek nem lehet egyértelműen meghatározni az értékét, visszaadja az értékkészletét, például: `T in 0..1.` Az *A* értéke *0* lett, ezt a korlátok alapján következtette ki a program, nagyon egyszerű módja volt, mivel `A` és `B` összege *1*, és `B` értéke *1*. Ettől persze sokkal bonyolultabb számításokat is el tud végezni a Prolog a feladat bonyolultságához mérten.

Azokat a mezőket, amelyek biztosan bombát rejtenek, zászlóval jelöltük meg. A *0* értékű, azaz üres mezőket felszabadítottuk. Az ezután kapott állapot a következő:



8.ábra: A játék második állapota

Az így kapott ábrán újraneveztük a mezőket, a meglévő információk és bombák függvényében az alábbi állítást fogalmazhattuk meg:

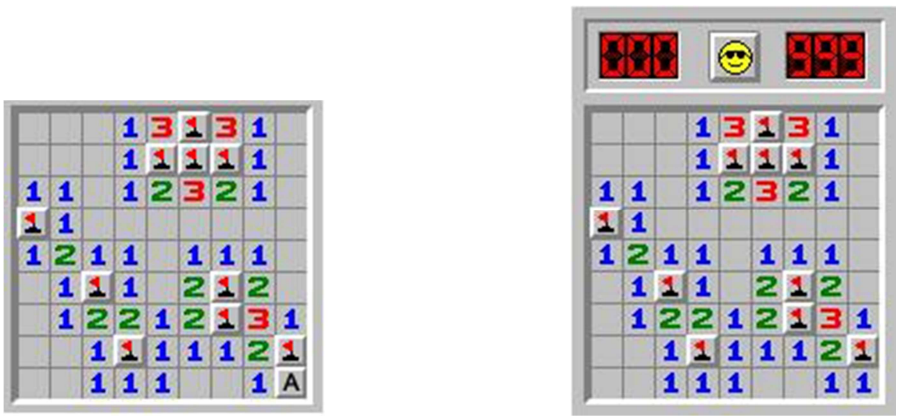
```
2 ?- [B1,B2,B3,B4,B5,B6,B7,B8] ins 1, [A,B,C,D,E,F,G,H] ins 0..1, B1+B2+A#=3, B6+B#=1, B6+B+C+D#=1, B7+C+D+E+F+G#=1, B7+B8+G+H#=3, G+H#=1.
```

Az állításban a bombák helyét `B1`-től `B8`-ig jelöltük, értékük *1*. Mostanra csökkent a mezők száma, kevesebb mezőt kell felfednünk, de több információnk van róluk.

A Prolog által kiszámolt értékek:

```
B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1, B8 = 1, A = 1, B = 0, C = 0, D = 0, E = 0, F = 0, G = 0, H = 1.
```

Azokat a mezőket, amelyeket az előző állapotban még nem tudtunk felfedni elegendő információ híján, most sikerült, mivel szűkítettük a lehetőségeket. A most megadott kifejezésekből a Prolog az összes most keresett változóértéket ki tudta számolni. Az ábrán látható, hogy az utolsó lépésre mindössze egy mező maradt lefedve.



9.ábra: A játék harmadik, és végállapota

A harmadik állapoton csak az `A` mező maradt fedett, egyszerűen kiszámítható, hogy az értéke *0* lesz:

```
3 ?- [B1,B2,B3,B4,B5,B6,B7,B8,B9,B10] ins 1, [A] ins 0..1, B9+A#=1.
```

```
B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1, B8 = 1, B9 = 1, B10 = 1, A = 0.
```

Mind a tíz bombát sikerült megtalálnunk, nem maradt lefedett mező a táblán, a feladat megoldása sikeres.

4.4 A hexagonális tábla megoldása

Az Aknakeresőnek számos alternatívája ismert, most egy hexagonális, azaz hatszög alapú játéktér megoldását fogjuk szemléltetni. A rejtett bombák száma tíz, egy mező szomszédjainak a száma jelen esetben hat. A játékszabályok természetesen minden játéktéren megegyeznek. A hexagonális játék kezdőállapota a következő:

10. ábra: A hatszög alapú játék egy lehetséges kezdőállapota

Ennél a példánál is csak azokkal a rejtett mezőkkel kell foglalkoznunk, amelyekről információnk van, tehát olyan ismert mezőkkel szomszédosak, amelyek mellett biztosan van bomba. A kezdőállapotban `A`-tól `J`-ig nevezzük meg az előbbi feltételnek megfelelő mezőket. Az első állapot utáni állításunk a következő:

```
?- [A,B,C,D,E,F,G,H,I,J] ins 0..1, A#=1, A+B+C#=2, C+D#=2,
D#=1, D+E#=1, E+F+G#=1, G+H#=1, H+I#=2, J#=1.
```

Természetesen ennél a feladatnál is kitűzzük, hogy a változók csak *0* vagy *1* értéket vehetnek fel. Rögtön az első kényszer egyértelműen meghatározza az `A` értékét, majd egyértelmű kényszerként megadható, hogy a `D` és a `J` mező értéke is egy. Ezt onnan tudjuk leolvasni, hogy ezekhez a mezőkhöz tartozik olyan információs mező, amelynek értéke *1*, és esetenként csak az adott mezővel (`A,D,J`) állnak szomszédságban. A többi mezőt is egyértelműen kiszámolja a Prolog, mivel a kényszerek száma elegendő a megoldáshoz.

A Prolog által meghatározott értékek:

```
A = 1, B = 0, C = 1, D = 1, E = 0, F = 1, G = 0, H = 1, I = 1,
J = 1.
```

A mezőket a megfelelő módon felszabadítjuk, vagy zászlóval jelöljük meg, attól függően, hogy bombát tartalmaz-e vagy sem. A művelet után kapott állapot a következő:

11.ábra: A hatszög alapú tábla második állapota

Újra elnevezzük azokat a rejtett mezőket, amelyeknek van esélyünk az értékét kikövetkeztetni. Az előző állításunk hét bomba helyét biztosan meghatározta, így már csak három bomba maradt rejtve, csökkent a változók száma is.

```
2 ?- [B1,B2,B3,B4,B5,B6,B7] ins 1, [A,B,C,D,E,F,G] ins 0..1,
B1+A+B+C+B2#=3, B3+D+E+B4#=2, B4+F+G+B5#=2.
```

A bombákat a megszokott módon `B1`-től `B7`-ig jelölik konstansok, értékük *1*. A mostani változókkal csupán 3 olyan mező érintkezik, amely hasznos információt tartalmaz számunkra, ezért 3 kényszert tudunk felírni.

A Prolog a következő értékeket számította ki:

```
B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1,
D = 0, E = 0, F = 0, G = 0,
A in 0..1, A+B+C#=1, B in 0..1, C in 0..1.
```

A lefutás után a Prolog kiírja a bombák már ismert értékét, majd azokat a változókat, amelyeknek értékét egyértelműen meghatározta. Ilyenek a `D,E,F,G` mezők. Az `A,B,C`

mezőknek nem lehetett ennyi kényszerből meghatározni az értéküket. Ezek a területek továbbra is rejtettek maradnak.

A következő állapot:

12.ábra: A hexagonális játéktábla harmadik állapota

Az előző szakaszban nem találtunk bombát, a mezők elnevezése után az alábbi állítást tudjuk felállítani:

```
3 ?- [B1,B2,B3,B4,B5,B6,B7] ins 1, [A,B,C,D,E,F,G,H,I] ins
0..1, A+B+C+B1+B2#=3, C+D+B2+B3#=3, D+E+F+B4#=2, B4+F+G+H#=2,
B5+B6+H+I#=2.
```

A harmadik állapotban sok egyezés van a második állapottal, ezeket áthoztuk a mostani állításunkba. Új kényszereket is felállítottunk, az előbb meghatározott üres mezők most információs mezőként szerepelnek. A Prolog által megfogalmazott értékek a következők:

```
B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1, H = 0,
I = 0, A in 0..1, A+B+C#=1, B in 0..1, C in 0..1,  C+D#=1, D
in 0..1, D+E+F#=1, E in 0..1, F in 0..1, F+G#=1, G in 0..1.
```

A Prolog két változónak tudott értéket meghatározni, a `H` és `I` mezők *0* értékűek, a többi mező ismeretlen maradt. Ha a Prolog az állítás kiértékelésekor nem tud társítani értéket egy változóhoz, akkor kiírja annak értékkészletét, és azt a kényszert, amiben a változó szerepel.
A megfelelő mezők felszabadítása után a *13.ábrán* látható állapotot kapjuk meg. Az ábrán jól látható, hogy az előző állítással felszabadított két mező hordoz számunkra új információt, a

többi mező változatlan maradt. Ez azt jelenti, hogy csak a *13.ábra* `F` és `G` mezőjének értékét tudja majd kiszámolni a Prolog.

13.ábra: A hexagonális tábla negyedik állapota.

A negyedik állapothoz tartozó állításunk, és annak kiértékelése:

```
4 ?- [B1,B2,B3,B4,B5,B6,B7] ins 1, [A,B,C,D,E,F,G] ins 0..1,
A+B+C+B1+B2#=3, C+D+B2+B3#=3, D+E+F+B4#=2, B4+F+G#=2, G#=1.

B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1, F = 0,
G = 1, A in 0..1, A+B+C#=1, B in 0..1, C in 0..1, C+D#=1,
D in 0..1, D+E#=1, E in 0..1.
```

Láthatjuk, hogy az `F` üres mező lesz, a `G` pedig bombát rejt. A többi változó értékét információ hiányában továbbra sem tudjuk megállapítani. Az így kapott állapot a következő:

14.ábra: A hexagonális tábla ötödik állapota.

A *14.ábrán* látható állapot kényszereinek megfogalmazása után az állításunkhoz hozzáírjuk a `label` predikátumot, amely az argumentumaiként megadott változók értékének összes lehetséges kombinációját kiírja megoldásként, természetesen a kényszerek kielégítésével.

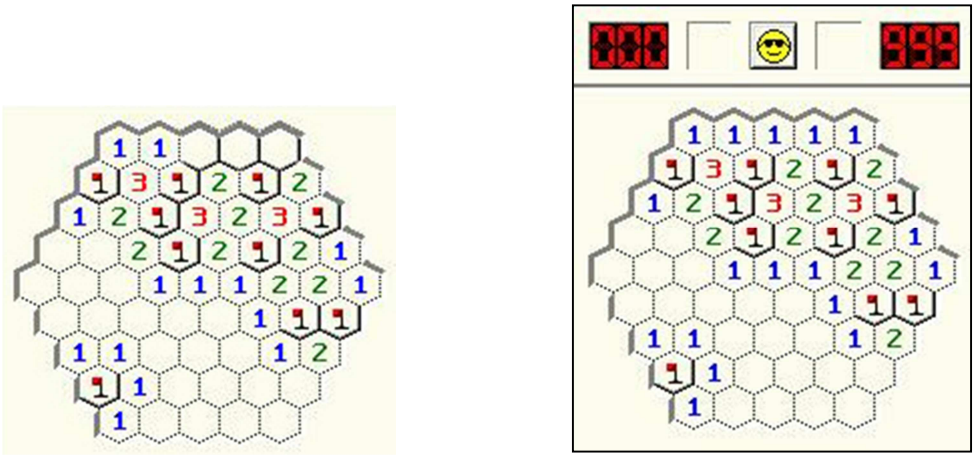
A *14.ábrához* tartozó állítás:

```
5 ?- [B1,B2,B3,B4,B5,B6,B7,B8] ins 1, [A,B,C,D,E,F] ins 0..1,
B1+A+B+C+B2#=3, B2+C+D+B3#=3, D+E+B4#=2, E+F+B4+B8#=3,
label([A,B,C,D,E,F]).
```

A Prolog az összes megfelelő megoldást kiírja, ha létezik több. Természetesen esetünkben csak egy jó megoldás kerülhet szóba, de vannak olyan szituációk az Aknakeresőben, hogy a választást a véletlenre kell bíznunk. A Prolog által javasolt megoldás:

```
B1 = 1, B2 = 1, B3 = 1, B4 = 1, B5 = 1, B6 = 1, B7 = 1, B8 = 1,
A = 0, B = 0, C = 1, D = 0, E = 1, F = 0.
```

A `label` használatával minden változónak megtaláltuk az értékét. A `C` és az `E` mezők rejtenek bombát, ezzel meg is találtuk mind a tízet.



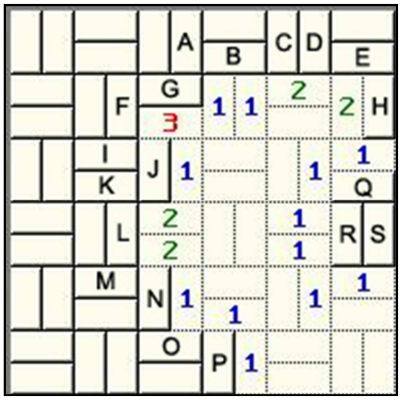
15.ábra: A hexagonális tábla hatodik, és végállapota.

A *15.ábrán*, a játék hatodik állapotán látható, hogy megtaláltuk az összes bombát ugyan, de 3 mező még rejtve maradt. Ezek értékei értelemszerűen *0*, tehát bátran felfedhetjük őket. A

végállapot mutatja, hogy az állításaink, és a megfogalmazott kényszereink helyesek voltak, a Prolog megfelelően dolgozott.

4.5 A „parkettás” tábla megoldása

Végül ismerkedjünk meg egy harmadik típusú játéktérrel és megoldásával. A „parkettás” vagy más néven téglalap alapú tábla jelen esetben szintén tíz bombát rejt, egy mezőnek maximum hét mező lehet a szomszédja. A megoldás menete hasonló lesz a korábban megismertekkel. Kis változtatás, hogy ebben a feladatban a bombák nem lesznek egymástól külön jelölve, hanem mindegyiket egységesen a `BOMBA` konstans fogja jelölni, értéke természetesen *1* marad. A kezdőállapot jelen esetben is az első kattintás utáni véletlenszerű állapot lesz.



16.ábra: A „parkettás” tábla egy lehetséges kezdőállapota.

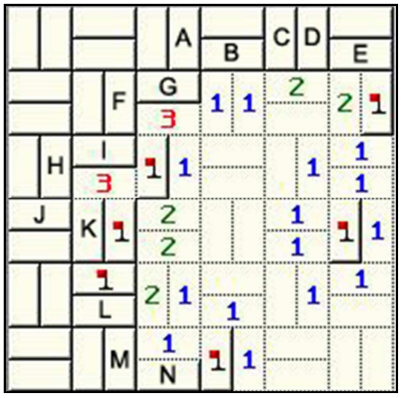
Ennél a kezdőállapotnál sok fedett mezőről van információnk, a mezőket továbbra is az ABC nagy betűivel jelöljük, jelen esetben `A`-tól `S`-ig. Sorra vesszük az ismert információs mezőkkel szomszédos rejtett mezőket, és felírjuk a kényszereket a változók és az információs mezők segítségével. A kezdőállapotra a következő állítást írhatjuk fel:

```
1 ?- [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S] ins 0..1,
A+G+B#=1, B+C#=1, B+C+D+E#=2, D+E+H#=2, H+Q#=1, Q+R#=1, R#=1,
R+S#=1, F+G+I+J+K#=3, J#=1, J+K+L#=2, L+M+N#=2, N+O+P#=1,
O+P#=1, P#=1.
```

Futás után a Prolog a következő értékeket adja vissza:

```
H = 1, J = 1, K = 0, L = 1, M = 1, N = 0, O = 0, P = 1, Q = 0,
R = 1, S = 0,
A in 0..1, A+B+G#=1, B in 0..1, B+C+D+E#=2, B+C#=1, C in 0..1,
D in 0..1, D+E#=1, E in 0..1, G in 0..1, F+G+I+1+0#=3,
F in 0..1, I in 0..1.
```

Látható, hogy nem mindegyik változóhoz sikerült értéket rendelni, `A`-tól `F`-ig, valamint az `I` mezőkhöz nem tartozott olyan szomszédos információs mező, amelyekből ki tudta volna számolni a Prolog a hozzá tartozó értékeket. Hat változó értéke lett *1*, tehát már az első állításunkkal megtaláltunk hat bombát, ezeket a táblán továbbra is zászlóval jelöljük. A *0* értékű mezőket felszabadítjuk, így kapjuk meg a játék második állapotát:



17.ábra: A „parkettás” tábla második állapota.

Az új állapotra tekintve láthatjuk, hogy csökkent a száma a felfedhető mezőknek, ezen az állapoton `A`-tól `N`-ig jelöljük a rejtett téglalapokat. Mivel már bombát is jelöltünk a táblán bevezetjük a `BOMBA` konstanst *1* értékkel az állításunkban. A kényszereket továbbra is az információs mezők és a változók segítségével állítjuk elő. Az információs mező értéke megegyezik a vele szomszédos betűvel, vagy bombával jelölt mezők összegével. A nulla információtartalmú mezőkkel továbbra sem foglalkozunk. A második állapotról a következő állítást tudjuk felírni:

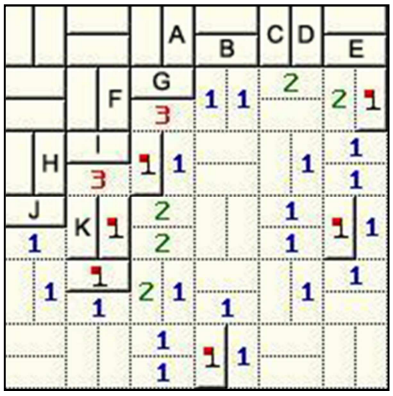
```
2 ?- [A,B,C,D,E,F,G,H,I,J,K,L,M,N] ins 0..1, [BOMBA] ins 1,
A+B+G#=1, B+C#=1, B+C+D+E#=2, D+E+BOMBA#=2, F+G+I+BOMBA#=3,
```

```
I+H+J+K+BOMBA+BOMBA#=3,  L+M+BOMBA+BOMBA#=2,  L+M+N+BOMBA#=1,
BOMBA+N+M#=1.
```

A Prolog által előállított értékek:

```
L = 0, M = 0, N = 0, BOMBA = 1,
A in 0..1, A+B+G#=1, B in 0..1, B+C+D+E#=2, B+C#=1, C in 0..1,
D in 0..1, D+E#=1, E in 0..1, G in 0..1, F+G+I#=2, F in 0..1,
I in 0..1, H+I+J+K#=1, H in 0..1, J in 0..1, K in 0..1.
```

Az állítás megvizsgálása után az `L`, `M` és `N` változókhoz tudott értéket rendelni a Prolog a megadott kényszerek alapján. Mindhárom mező üres, új bombát nem találtunk. A többi változót a megadott korlátokkal nem tudtuk egyértelműen meghatározni. A mezők felszabadítása után a következő állapotot kapjuk:



18.ábra: A „parkettás” játéktér harmadik állapota

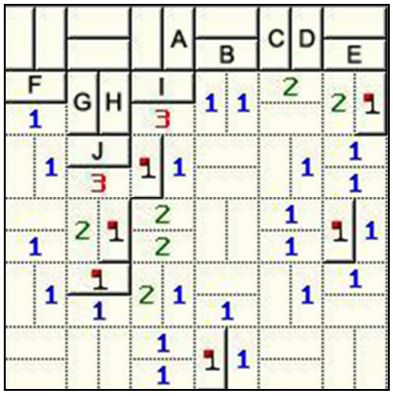
A *17.ábra* `M` mezőjével nagy területet sikerült felszabadítani, mivel ez a mező nem információs mező. Ilyenkor a vele érintkező üres mezőket, és az azokat határoló információs mezőket automatikusan felszabadítja az Aknakereső. A felfedhető rejtett mezők száma csökkent, `A`-tól `K`-ig jelöljük őket. A kényszereket a szokásos módon leírva az alábbi állítást fogalmazzuk meg a Prolog számára:

```
3 ?- [A,B,C,D,E,F,G,H,I,J,K] ins 0..1, [BOMBA] ins 1,
A+B+G#=1, B+C#=1, B+C+D+E#=2, D+E+BOMBA#=2, F+G+I+BOMBA#=3,
H+I+J+K+BOMBA+BOMBA#=3, J+K+BOMBA#=1.
```

A visszatérési értékek a következőek:

```
J = 0, K = 0, BOMBA = 1,
A in 0..1, A+B+G#=1, B in 0..1, B+C+D+E#=2, B+C#=1, C in 0..1,
D in 0..1, D+E#=1, E in 0..1, G in 0..1, F+G+I#=2, F in 0..1,
I in 0..1, H+I#=1, H in 0..1.
```

Ebben az állapotban sem találtunk bombát, de két üres mezőt ismét felfedhetünk, a `J`-t és a `K`-t. Ismét információ nélküli mezőt találtunk, így az Aknakereső program megint több mezőt fed fel automatikusan:



19.ábra: A „parkettás” játéktér negyedik állapota.

Az érdekelt mezőket `A`-tól `J`-ig neveztük meg a *19.ábrán*. A korábban használt kényszerek mellett kibővítjük az állításunkat az `F,G,J` mezőkre vonatkozó új kényszerekkel:

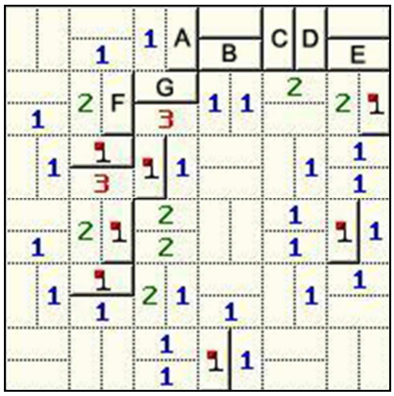
```
4 ?- [A,B,C,D,E,F,G,H,I,J] ins 0..1, [BOMBA] ins 1, A+B+I#=1,
B+C#=1, B+C+D+E#=2, D+E+BOMBA#=2, H+I+J+BOMBA#=3, F+G+J#=1,
G+J#=1, J+BOMBA+BOMBA#=3.
```

A Prolog által kiszámolt értékek a következőek:

```
F = 0, G = 0, J = 1, BOMBA = 1,
A in 0..1, A+B+I#=1, B in 0..1, B+C+D+E#=2, B+C#=1, C in 0..1,
D in 0..1, D+E#=1, E in 0..1, I in 0..1, H+J#=1, H in 0..1.
```

Az állításunkba újonnan bevett mezők értékét a Prolog visszaadta, de a többi keresett értéket még nem tudta kiszámítani információ hiányában. Újabb bombát találtunk a *19.ábra*

J mezője alatt. Ezzel a felfedezett bombák száma hétre nőtt. A J mezőt zászlóval jelöljük, az F és G mezőket felszabadítjuk. Ekkor a következő ábrát kapjuk:



20.ábra: A „parkettás” játéktér ötödik állapota.

A *19.ábra* G mezője információs mező, értéke *2*, szomszédságában két bomba található. Szintén a *19.ábra* F mezője azonban üres, nem rendelkezik információ tartalommal sem, ezért újabb automatikus felszabadítás történik a környezetében. A *20.ábra* az ez utáni állapotot szemlélteti. Közel kerültünk a megoldáshoz, jelen esetben hét változó értékének meghatározására van lehetőségünk. Az állításunk a következő:

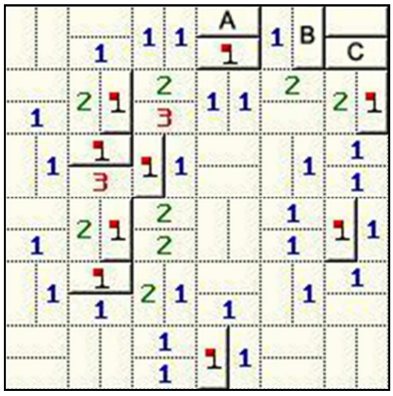
```
5 ?- [A,B,C,D,E,F,G] ins 0..1, [BOMBA] ins 1, A+B+G#=1,
B+C#=1, B+C+D+E#=2, D+E+BOMBA#=2, F+G+BOMBA+BOMBA#=3,
F+BOMBA#=2, F+G#=1, A+F+G#=1.
```

Az állításunkra kapott válasz:

```
A = 0, B = 1, C = 0, F = 1, G = 0, BOMBA = 1, D in 0..1,
D+E#=1, D+E#=1, E in 0..1.
```

A *20.ábrán* látható állapot már eléggé leszűkült ahhoz, hogy a kényszerek felírásával majdnem az összes változóértéket meg tudja adni a Prolog. Ismét megállapítottuk két bomba helyét, ezek a *20.ábra* B és F mezőjén találhatóak és zászlóval jelöljük őket a következő állapotban. Ezek után már csak egy bombát kell felkutatnunk és megoldottuk a feladatot. Az A, C és G mezőket felszabadítjuk.

A *21.ábrán* látható, hogy már csak négy mező maradt ismeretlen, ezek közül hármat esélyünk van felfedni. A kényszerek egyértelműen megfogalmazhatók, egy rövid állítással meg lehet határozni a mezők értékét.



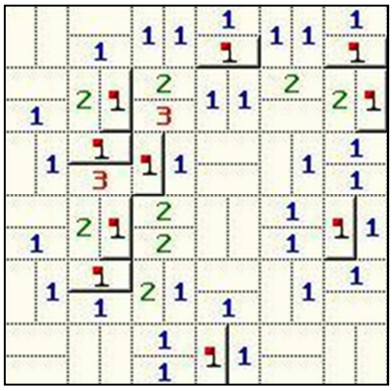
21.ábra: A „parkettás” játéktér hatodik állapota.

```
6 ?- [A,B,C] ins 0..1, [BOMBA] ins 1, A+B+BOMBA#=1,
B+C+BOMBA#=2.
```

A Prolog által kiszámított válasz:

```
A = 0, B = 0, C = 1, BOMBA = 1.
```

A három változó értékének meghatározásához elegendő volt két kényszert megadnunk az állításunkban. Láthatjuk a Prolog válaszában, hogy az utolsó keresett bombát a `C` mező rejti, mivel a változó értéke *1* lett. A `C` mező megjelölése, és a másik két mező felszabadítása után megkapjuk azt az állapotot, amelyben már csak egy mező van fedve előttünk, de mivel megtaláltuk az összes bombát, ez biztosan üres lesz. Az Aknakereső játék „parkettás” játékterének végállapota a következő:



22.ábra: A „parkettás” játéktér végállapota.

5. Összefoglalalás

Szakdolgozatom elkészítésével betekintést szerettem volna nyújtani a kényszer kielégítési problémák világába az Aknakereső logikai játékon keresztül. A kényszer kielégítés (Constraint Satisfaction) egy olyan megoldási folyamat, amelyben a program különböző feltételeket állít a változókkal szemben. A kényszer kielégítési módszerek alkalmazása a Prolog logikai programozási nyelven, és annak korlát-logikai (CLPFD) könyvtárán keresztül került bemutatásra.

A dolgozat során megismerkedhettünk a Prolog nyelv és a korlát-logikai programozás alapjaival, valamint a kényszer kielégítési módszerekről és problémákról részletesebben is szó esett. Ezen témák tárgyalásakor sor került az elméleti és történeti hátterek áttekintésére úgy, mint a felépítésük és működésük vizsgálatára. A megértést példákkal és ábrákkal próbáltam megkönnyíteni.

Ezek után következett maga a feladat ismertetése. A cél az Aknakereső játék három különböző játékterének megoldása volt, kényszer kielégítési módszereket alkalmazva. A feladat lényege az volt, hogy a játék adott kezdőállapotából kiindulva kényszereket tartalmazó állításokat fogalmaztam meg az éppen aktuális ábráról. Ezt az állítást a Prolog korlát-logikai alkönyvtára segítségével feldolgozta, és a változókhoz értékeket rendelt, egy következő állapotba lendítve a játék megoldását.

A játék ilyen típusú megoldásával célom a kényszer kielégítés egyik lehetséges alkalmazási területének bemutatása volt.

6. *Irodalomjegyzék*

[1] Bóta András: A korlát-logikai programozás és alkalmazásai,
Debreceni Egyetem, diplomamunka,
Debrecen, 2009.

[2] Peter Flach: Logikai programozás,
Panem, 2001.

[3] Futó Iván: A logikai programozás hazai története,
http://www.sulinet.hu/termeszetvilaga/archiv/2000/0014/07.html
Letöltve: 2010.szeptember 5.

[4] Kovács András: A korlátozás programozás alapjai,
http://www.sztaki.hu/~akovacs/publications/CLPsegedlet.pdf
Letöltve: 2010. szeptember 13.

[5] Matijevics István: A programozás alapjai, programozási nyelvek,
http://www.cs.bme.hu/~szeredi/oktatas/vflp/vflp04/prologikai_talk.pdf
Letöltve: 2010. szeptember 12.

[6] Stuart Russel, Peter Norvig: Mesterséges intelligencia modern megközelítésben,
Panem, 2000.

[7] Szeredi Péter, Lukácsy Gergely: Nagyhatékonyságú logikai programozás,
Budapesti Műszaki Egyetem, jegyzet,
Budapest, 2010.

[8] Ujhelyi Zoltán: Modelltranszformációk statikus analízise,
Budapesti Műszaki és Gazdaságtudományi Egyetem, publikáció,
Budapest, 2009.

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

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőmnek, Dr. Aszalós Lászlónak a szakdolgozatom megírásához nyújtott hasznos tanácsait, türelmét és segítőkészségét.

Valamint szeretném kifejezni köszönetemet családtagjaimnak, akik szeretetükkel és segítségükkel mindvégig támaszt nyújtottak eddigi egyetemi tanulmányaim során és szakdolgozatom elkészítésének teljes ideje alatt.