SZAKDOLGOZAT

Majoros Gergely

Debrecen 2010

Debreceni Egyetem Informatika

Nonogram
és
automatikus tételbizonyítás

Témavezető:
Aszalós László
PhD

Készítette:
Majoros Gergely
programtervező
informatikus

Debrecen 2010

Tartalomjegyzék

Bevezetés 4
1. A Nonogram 5
2. Az ítéletkalkulus 8
2.1 A szemantika 9
2.2 A kielégíthetőségi tulajdonságok 10
2.3 Formulák ekvivalenciája 11
2.4 Logikai törvények 11
3. A SAT-probléma 13
3.1 Konjunktív normálforma 13
3.2 A DPLL algoritmus 14
4. A backtrack 16
5. A rejtvény implementálása 18
6. A megoldást kereső algoritmus főbb lépései 21
7. Az átfedések meghatározása és a felderített mezők felhasználása 26
7.1 A variációk előállítása 27
7.2 Az átfedések meghatározása 30
7.3 A felderített mezők felhasználása 34
7.4 További tevékenységek 37
8. Új állapot létrehozása 38
8.1 Új állapot létrehozása a programban 39
9. Futási eredmények 42
10. Továbbfejlesztési lehetőségek 46
Melléklet 48
Irodalomjegyzék 52

Bevezetés

A szabatos tételbizonyító módszerek és eljárások létrehozásának igénye lényegében egyidős a matematikával. A modern formájú tételbizonyítás elméletét Herbrand munkássága alapozta meg. Az 1930-ban publikált eredményeket követő három évtizedben még elméleti kutatás folyt ezen a területen, a 60-as évek elején azonban megszületett a számítógépre alkalmas tételbizonyító eljárás, a J. A. Robinson nevéhez fűződő rezolúció. Az automatikus tételbizonyítás, automatikus következtetés a Mesterséges Intelligencia igen aktív területévé vált. A kutatók nagyszámú rezolúciós tételbizonyító programot írtak, mégpedig lényegében két célt követve: a rezolúció hatékonyságát növelő stratégiák kidolgozása, illetve a matematikai tételek gépi bizonyítása. Ami a stratégiák kérdését illeti, az lezártnak tekinthető, a matematikai tételek bizonyítása és általában új, az egyes témakörökhöz illeszkedő bizonyítási módszerek keresése ma is aktívan művelt terület.

A tételbizonyításhoz a logika nyelvét hívták segítségül, melyről idővel kiderült, hogy nem csak a matematika, hanem a Mesterséges Intelligencia gyakorlatból vett problémái is formalizálhatók. Amellett, hogy ez a nyelv, vagyis a predikátum kalkulus nyelve kellően rugalmas a bonyolult állítások leírására, pontos szintaxissal is rendelkezik. Az általunk használt bizonyító eljárás pedig helyes és teljes, azaz minden formalizálható feladat megoldható vele feltéve, hogy a feladat valóban megoldható. Tovább növelte a logika fontosságát az ismeretek meghatározó szerepének felismerése a problémamegoldásban. Így a logika és a logikai reprezentáció több elméleti irányzatnak is alapját képezi.

Ebben a szakdolgozatban a logikai alapú reprezentáció és a Nonogram nevű rejtvény megoldását végző algoritmus és program kerül bemutatásra, mely az automatikus tételbizonyítás egyik módszerét, a DPLL-t használja. A program forráskódja Java nyelven íródott.

1. A Nonogram

A Nonogram vagy, ahogy itthon ismert, grafilogika egy logikai rejtvényfajta, amely egy japán dizájner, Non Ishida ötletei nyomán alakult ki. 1986-ban azzal az ötletével nyert meg egy grafikai versenyt, hogy egy felhőkarcoló bizonyos ablakait kivilágított, másokat pedig elsötétített, így távolról nézve az épületet, egy kép rajzolódott ki. Ezen ötletén alapul a később grafilogikának nevezett rejtvény is. 1990-ben James Dalgety adta ennek a játéknak a "*Nonogram*" nevet és ekkor kezdte el a The Sunday Telegraph megjelentetni ezeket a rejtvényeket.

1993-ban megjelent az első könyv a témában, a *Book of Nonograms,* amelyet Non Ishida írt. Ekkor már többek között Svédországban, az Egyesült Államokban és Dél-Afrikában is ismert volt a játék.

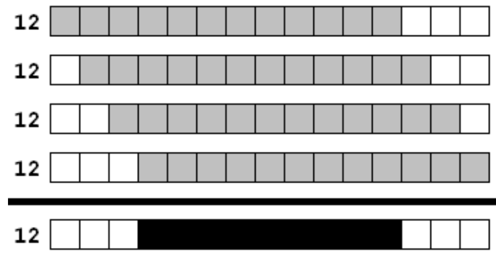
1995-ben a grafilogika megjelent különböző játékkonzolokra, például a Game Boyra is. A növekvő népszerűség hatására Japánban egyre többen kezdtek új feladványokat gyártani, és ekkor már önálló havilapok is jelentek meg.

Mára már sok országban jelenik meg ilyen folyóirat, többek között az USA-ban, Angliában, Németországban, Hollandiában, Olaszországban, Magyarországon és Finnországban.

1. ábra, Egy kitöltött Nonogram rejtvény

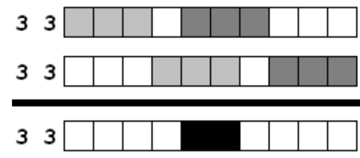
A rejtvény egy téglalap alakú négyzetrácsos hálóból áll, amelynek az egyik vízszintes, és az egyik függőleges oldala mellett számok állnak, amelyek azt jelzik, hogy az adott sorban, vagy oszlopban mekkora méretű sötét blokkok követik egymást. Minden beszínezett blokkot tetszőleges számú, de legalább egy, üres hely választ el egymástól. A játék célja, hogy eldöntsük minden négyzetrácsról, hogy be van-e színezve, vagy sem.

A megoldás során a rejtvényre tekintsünk úgy, mint egy kétdimenziós tömbre. Minden négyzetrácsot két index segítségével azonosítunk, az első a sort, a második pedig az oszlopot adja meg, a tömböt magát, pedig jelöljük *R*-rel. Az indexelés történjen a rejtvény bal felső sarkából, továbbá a Java sajátosságai miatt 0-tól induljon. Tehát egy n×m-es rejtvénynél a bal felső sarokban lévő négyzetrács *R[0, 0]*, míg a jobb alsó *R[n-1, m-1]*.



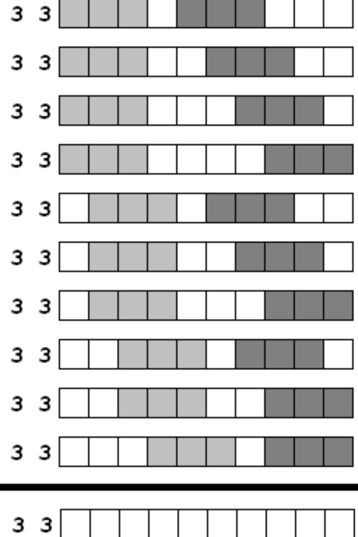
2. ábra, Átfedések megkeresése

A rejtvény megoldásának megkezdése az átfedések megkeresésével történik, egyszerűbb rejtvények pusztán ezzel az egy módszerrel megoldhatóak. Ha egy sor (oszlop) elején egy szám áll, és ez a szám nagyobb, mint a hálózat szélességének (magasságának) fele, akkor átfedés keletkezik. A 2. ábra egy olyan részletét mutatja, egy rejtvénynek melyben 12 egység hosszú sötét blokk van egy sorban, és ez hosszabb, mint a hálózat szélességének a fele. Ha végig vesszük azt, hogy hogyan tudjuk a 12 hosszúságú sötét blokkot elhelyezni, ez látható az ábra felső részén, akkor lesznek olyan négyzetek, amelyek minden esetben be lesznek színezve. Ezek a rajzon az utolsó sorban fekete színűek, és biztosan része lesznek a rejtvény megfejtésének.



3. ábra, különböző blokkok átfedése

Akkor is keletkezhetnek átfedések, ha több szám áll a sor (oszlop) elején. Hogy ezeket megtaláljuk, el kell képzelni a két szélsőséges helyzetet, azaz azt az esetet, amikor a blokkok a baloldalon, illetve amikor a jobb oldalon vannak. Minden átfedés azonban nem érvényes ebben a helyzetben, csak akkor számíthat, ha ugyanazon blokkon belül van. Például egy 3 3-as páros egy tízes blokkon belül nem képez átfedést, noha látszólag létrejön 2 a 3. ábra szerint. Ez azonban nem érvényes, mivel két különböző blokk találkozásánál jön csak létre (könnyű ellenőrizni ezt, képzeljük csak el a lehetséges variációkat).



4. ábra, a két 3 3-as blokk esetén az összes lehetséges variáció

A rejtvény újabb változataiban az egyes négyzetrácsok már nem csak két színnel rendelkezhetnek, de mi csak a Fekete - Fehér változattal foglalkozunk, továbbá kikötjük, hogy **a rejtvénynek mindig van egy és csak egy megoldása**, ugyanis csak ezek számítanak korrekt Nonogram rejtvénynek.

2. Az ítéletkalkulus

Az automatikus tételbizonyítás eszközeinek kihasználásához elengedhetetlen az ítéletkalkulus ismerete, ezért a következőkben a megoldást kereső algoritmus ismertetéséhez szükséges, az ítéletkalkulusban használt fogalmak kerülnek bemutatásra.

Az ítéletkalkulus, más néven nulladrendű predikátumkalkulus, egy egyszerű formális nyelv. Olyan kijelentő mondatok reprezentálására alkalmas, amelyek kötőszavakkal egyszerű részmondatokból épülnek fel, és köznapi értelemben igazak vagy hamisak.

A megoldott rejtvényben a négyzetrácsoknak két állapotuk lehetséges: be van színezve, nincs beszínezve. Azt hogy az egyes négyzetrácsok milyen állapotúak lehetnek az adott sor illetve oszlop előtt álló számoktól függ. Tehát tulajdonképpen az egyes sorokról és oszlopokról kell nyilatkoznunk, úgy hogy megadjuk a blokkok előfordulásának összes esetét, és e sorok és oszlopok összekapcsolásával kapjuk meg magát a rejtvény leírását.

2.1 A szintaxis

Az ítélet kalkulus formuláit az alábbi jelkészlet elemeiből építjük fel.

1. Elválasztó jelek: a (és a) zárójelek, a könnyebb olvashatóság érdekében a *[* és *]* jeleket is használjuk.
2. Logikai műveletek:

¬	a negáció jele	(„nem”)
Λ	a konjunkció jele	(„és”)
V	a diszjunkció jele	(„vagy”)
≡	az ekvivalencia jele	(„akkor és csak akkor”)

3. Ítéletváltozók (logikai változók):

   Jelen esetben ezeket jelöljük $R_{i,j}$–vel, melyek nem összekeverendők a korábban említett *R[i, j]* -kel.

Az ítéletkalkulus formuláit a szintaxis szabályai szerint alakítjuk ki:

- Minden ítéletváltozó egyben formula is.
- Ha *A* és *B* formulák, akkor $\neg A$, $(A \land B)$, $(A \lor B)$, $(A \equiv B)$ kifejezések is formulák.

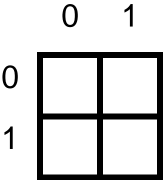
Minden formula e két szabály véges sokszori alkalmazásával előállítható.

A zárójelek gyakran könnyítik a formula megértését, de a teljes zárójelezettség zavaró lehet. A túl sok zárójelet a műveletek precedenciájának figyelembevételével kerülhetjük meg. A precedencia sorrend:

$$\neg, \land, \lor, \equiv$$

Ahol a negáció a legerősebb, az ekvivalencia pedig a leggyengébb művelet.

Ezek után írjuk fel a következő 2×2-es rejtvényt az ítéletkalkulus nyelvén:



5. ábra, egy 2×2-es rejtvény

$$(\neg R_{0,0} \land \neg R_{0,1}) \land [(R_{1,0} \land \neg R_{1,1}) \lor (\neg R_{1,0} \land R_{1,1})] \land$$
$$(\neg R_{0,0} \land \neg R_{1,0}) \land [(R_{0,1} \land \neg R_{1,1}) \lor (\neg R_{0,1} \land R_{1,1})]$$

Ahol az $R_{0,0}$, $R_{0,1}$… azt jelöli, hogy a négyzetrács be van színezve.

2.1 A szemantika

Az ítéletkalkulus egy formulájának a szemantika szabályai szerint adhatunk jelentést, mely két lépésben történik: először **interpretáljuk** a formulát, ezután pedig **kiértékeljük**.

Egy formula interpretációját úgy kapjuk, hogy minden ítéletváltozójához hozzárendeljük a *logikai igaz* és *logikai hamis* értékek egyikét. Ez a hozzárendelés tetszőleges lehet. Az igaz értéket *I*-vel, a hamis értéket *H*-val jelöljük, melyek nem eleme a szintaxis jelkészletének, így nem szerepelhetnek a formulákban.

Az *R[i, j]* -k és az $R_{i,j}$ –k közötti kapcsolatot ezek után definiáljuk a következő módon:

$$R[i, j] = \text{ÜRES} \leftrightarrow |\, R_{i,j} \,| = H$$

$$R[i, j] = \text{FOGLALT} \leftrightarrow |\, R_{i,j} \,| = I$$

Ahol $|\, R_{i,j} \,|$ jelölés az $R_{i,j}$ igazságértékét jelöli.

Az interpretált formula kiértékelését, igazságértékelését, a műveleti jelek szemantikája alapján végezzük:

¬	A	A	Λ	B	A	V	B	A	≡	B
H	I	I	I	I	I	I	I	I	I	I
I	H	H	H	I	H	I	I	H	H	I
		I	H	H	I	I	H	I	H	H
		H	H	H	H	H	H	H	I	H

Jó ötletnek tűnhet, hogy a rejtvényt megpróbáljuk kiértékelni, azonban ha az túl sok mezőből áll, akkor az interpretációk száma nagyon sok lesz, egészen pontosan $2^{\text{mezők száma}}$ darab.

2.2 A kielégíthetőségi tulajdonságok

Egy formula jelentését nyilván akkor ismerjük teljesen, ha minden lehetséges interpretációban kiértékeljük.

Kielégíthetőnek nevezünk egy formulát, ha van olyan interpretáció, amelyben igaz az értéke, ezt az interpretációt a formula egy **modelljének** nevezik. Az általunk vizsgált Nonogram rejtvényekhez felírt formulák mind ilyen tulajdonságúak, tehát minden általunk vizsgált formulához létezik egy modell, ami a rejtvény megoldását adja.

Érvényesnek mondjuk azt a formulát, amelyik minden interpretációban igaz és az érvényes formulát **tautológiának** nevezik.

Kielégíthetetlennek nevezzük az olyan formulát, amelynek minden interpretációban hamis az értéke.

2.3 Formulák ekvivalenciája

Az aritmetikai számításokhoz és az algebrai levezetésekhez hasonlóan a logikában is szükség lehet arra, hogy egy kifejezést egy vele ekvivalens, általában egyszerűbb, kifejezéssel helyettesítsünk.

Két formulát ekvivalensnek nevezünk, ha logikai értékük minden interpretációban megegyezik. Az A és B formulák ekvivalenciájára az $A \sim B$ jelölést használjuk.

Az „ekvivalencia” szó két külön fogalom megnevezése: ez egy részt az egyik logikai művelet (≡) neve, másrészt azt jelenti, hogy két formula logikai jelentése megegyezik. A szövegkörnyezetből mindig kiderül, hogy melyik értelemben használjuk a szót. A szóegyezés különben nem véletlen, mert az A és B formulák akkor és csak akkor ekvivalensek, ha az $A \equiv B$ formula érvényes.

2.4 Logikai törvények

A nevezetesebb $A \sim B$ alakú összefüggéseket logikai törvényeknek nevezzük, melyek közül a számunkra legfontosabbak:

1. $A \wedge B \sim B \wedge A$ (kommutatív törvények)
   $A \vee B \sim B \vee A$
2. $(A \wedge B) \wedge C \sim A \wedge (B \wedge C)$ (asszociatív törvények)
   $(A \vee B) \vee C \sim A \vee (B \vee C)$
3. $A \wedge (B \vee C) \sim (A \wedge B) \vee (A \wedge C)$ (disztributív törvények)
   $A \vee (B \wedge C) \sim (A \vee B) \wedge (A \vee C)$
4. $A \wedge \top \sim A$, ha $\top$ tautológia
   $A \vee \bot \sim A$, ha $\bot$ kielégíthetetlen
5. $A \wedge \bot \sim \bot$
   $A \vee \top \sim \top$
6. $A \wedge \neg A \sim \bot$
   $A \vee \neg A \sim \top$
7. $\neg\neg A \sim A$ (a kettős tagadás törvénye)

8. $A \wedge (A \vee B) \sim A$ (elimináció, elnyelés)

$A \vee (A \wedge B) \sim A$

9. $A \wedge A \sim A$ (idempotencia)

A felsorolt ekvivalenciák könnyen igazolhatók például az összes interpretációt tartalmazó igazságtáblák segítségével. Az is észrevehető, hogy a logikai törvények tulajdonképpen ekvivalencia sémák, hiszen mindegyikből végtelen sok ekvivalencia származtatható konkrét formulák behelyettesítésével, továbbá az is feltűnhet, hogy bizonyos törvények másokból levezethetők.

3. A SAT-probléma

A SAT-probléma a mesterséges intelligencia kutatások egyik kiemelkedő területe, olyan esetekben találkozunk vele, mint az automatikus tételbizonyítás, VLSI áramkörök helyességének biztosítása, tudásalapú ellenőrzés és érvényesítés. Már 1971-ben bizonyította róla Stephan Cook, hogy ez egy *NP-teljes* bonyolultságú feladat. Maga a probléma nem más, mint egy ítéletlogikai állítás kielégíthetőségének a vizsgálata. Mai ismereteink szerint nem létezik algoritmus, mely a legrosszabb esetre, például, ha kielégíthetetlen a formulánk, polinomiális időben megoldaná a feladatot.

3.1 Konjunktív normálforma

Egy SAT probléma megoldásához az algoritmusunk konjunktív normálformát használ. Ez az eredeti formulának olyan speciális alakja, amely az eredetivel ekvivalens.

A konjunktív normálforma olyan kifejezés, amely speciális részformulák, klózok konjunkciója. Egyetlen klóz is konjunktív normálformát alkot.

Egy **klóz** literálok diszjunkciója, de lehet egyetlen literál is. **Literálnak** egy ítéletváltozót vagy egy ítéletváltozó negáltját nevezzük.

Bármely rejtvényhez felírt formula ilyen alakra hozható. Az átalakítást megfelelő oldali disztributív törvények alkalmazásával lehet elvégezni.

A konjunktív normálforma a képzési módja következtében ekvivalens az eredeti formulával, így annak kielégíthetőségi tulajdonsága sem változik. A SAT-probléma megoldását végző algoritmust a formula klóz formájára alkalmazzuk. Ehhez úgy jutunk, hogy a konjunktív normálformából elhagyjuk a konjunkció műveleteket és a formulát klózok halmazának tekintjük. Ezt azért tehetjük meg, mert a klózok sorrendje közömbös. Természetesen nem felejtjük el azt, hogy a klóz halmaz elemeit a konjunkció művelete kapcsolja össze.

3.2 A DPLL algoritmus

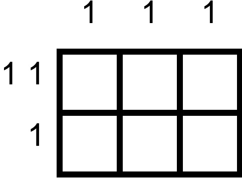
Az automatikus tételbizonyítás egyik fontos megközelítése J. Herbrand nevéhez fűződik, aki algoritmust dolgozott ki olyan igazságértékelés előállítására, mely hamissá tehet egy formulát. Ezt az algoritmust később továbbfejlesztették, és *Davis-Putnam* (DP) néven terjedt el. A módszer időtállóságáról tanúskodik, hogy napjaink vezető SAT-fejtői, mint például a Chaff, még mindig rendelkeznek DP eljárásra épülő modullal.

A DPLL az alábbi egyszerű szabályokból áll:

1. Ha a konjunktív normálforma kielégíthető, akkor a formula kielégíthető.
2. Ha az egyik klóz kielégíthetetlen, akkor a formula kielégíthetetlen.
3. Ha egy klóz egyetlen változót tartalmaz, akkor ez a klóz meghatározza a formula értékét.
4. Valamely heurisztika segítségével válasszunk egy x változót, és állítsuk értékét c-re. Ha az így kapott formula kielégíthető, akkor az eredeti is kielégíthető, egyébként pedig állítsuk az x értékét $\neg c$-re.

A Nonogram rejtvényünk esetében nem az a kérdés, hogy a formula kielégíthető-e, hanem, hogy melyik az az interpretáció, amelyben kielégíthető. Tehát mi az $R_{i,j}$-k értéke, ezért lesz kiemelt fontosságú az algoritmus 3. szabálya, ugyanis ha tudjuk, hogy a formula kielégíthető, így nyilatkozni tudunk azokról a változókról, amelyek egyedül állnak a klózokban. A feladatunk kezdetben tehát ezen egyedül álló változók meghatározása.

Az algoritmus működésére nézzük a következő példát:



6. ábra, példa a DPLL-re

A rejtvényt reprezentáló formula:

$(R_{0,0} \land \neg R_{0,1} \land R_{0,2}) \land [(R_{1,0} \land \neg R_{1,1} \land \neg R_{1,2}) \lor (\neg R_{1,0} \land R_{1,1} \land \neg R_{1,2}) \lor (\neg R_{1,0} \land \neg R_{1,1} \land R_{1,2})] \land [(R_{0,0} \land \neg R_{1,0}) \lor (\neg R_{0,0} \land R_{1,0})] \land [(R_{0,1} \land \neg R_{1,1}) \lor (\neg R_{0,1} \land R_{1,1})] \land [(R_{0,2} \land \neg R_{1,2}) \lor (\neg R_{0,2} \land R_{1,2})] \land [(R_{0,3} \land \neg R_{1,3}) \lor (\neg R_{0,3} \land R_{1,3})]$

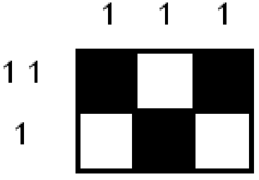
Ha ezt KNF-re hozzuk és alkalmazzuk az idempotencia szabályát a klózokon belül, akkor a következő formulát kapjuk:

$\mathbf{R_{0,0}} \land \mathbf{\neg R_{0,1}} \land \mathbf{R_{0,2}} \land (R_{1,0} \lor \neg R_{1,0}) \land (R_{1,0} \lor \neg R_{1,0} \lor \neg R_{1,1}) \land (R_{1,0} \lor \neg R_{1,0} \lor R_{1,2}) \land (R_{1,0} \lor R_{1,1} \lor \neg R_{1,0}) \land (R_{1,0} \lor R_{1,1} \lor \neg R_{1,1}) \land (R_{1,0} \lor R_{1,1} \lor R_{1,2}) \land (R_{1,0} \lor \neg R_{1,2} \lor \neg R_{1,0}) \land (R_{1,0} \lor \neg R_{1,2} \lor \neg R_{1,1}) \land (R_{1,0} \lor \neg R_{1,2} \lor R_{1,2}) \land (R_{0,0} \lor \neg R_{0,0}) \land (R_{0,0} \lor R_{1,0}) \land (\neg R_{1,0} \lor \neg R_{0,0}) \land (\neg R_{1,0} \lor R_{1,0}) \land (R_{0,1} \lor \neg R_{0,1}) \land (R_{0,1} \lor R_{1,1}) \land (\neg R_{1,1} \lor \neg R_{0,1}) \land (\neg R_{1,1} \lor R_{1,1}) \land (R_{0,2} \lor \neg R_{0,2}) \land (R_{0,2} \lor R_{1,2}) \land (\neg R_{1,2} \lor \neg R_{0,2}) \land (\neg R_{1,2} \lor R_{1,2})$

Ha a rejtvényünknek van egy és csak egy megoldása, akkor az eredeti formula kielégíthető, tehát a KNF is kielégíthető. A DPLL 3. szabálya szerint pedig, az egyetlen literált tartalmazó klózok meghatározzák az általuk tartalmazott ítéletváltozók értékét, tehát az első három klózból következik, hogy $|R_{0,0}|=I$, $|R_{0,1}|=H$ és $|R_{0,2}|=I$. Ezután a KNF-re alkalmazzuk a 2.4.4 és 2.4.5-ös logikaitörvényt.

$(R_{1,0} \lor \neg R_{1,0}) \land (R_{1,0} \lor \neg R_{1,0} \lor \neg R_{1,1}) \land (R_{1,0} \lor \neg R_{1,0} \lor R_{1,2}) \land (R_{1,0} \lor R_{1,1} \lor \neg R_{1,0}) \land (R_{1,0} \lor R_{1,1} \lor \neg R_{1,1}) \land (R_{1,0} \lor R_{1,1} \lor R_{1,2}) \land (R_{1,0} \lor \neg R_{1,2} \lor \neg R_{1,0}) \land (R_{1,0} \lor \neg R_{1,2} \lor \neg R_{1,1}) \land (R_{1,0} \lor \neg R_{1,2} \lor R_{1,2}) \land \mathbf{R_{1,0}} \land (\neg R_{1,0} \lor R_{1,0}) \land \mathbf{\neg R_{1,1}} \land (\neg R_{1,1} \lor R_{1,1}) \land \mathbf{R_{1,2}} \land (\neg R_{1,2} \lor \neg R_{0,2}) \land (\neg R_{1,2} \lor R_{1,2})$

Az így kapott KNF-ből következik, hogy $|R_{1,0}|=I$, $|R_{1,1}|=H$, $|R_{1,2}|=I$ és ezzel minden ítéletváltozó értékét kiderítettük.



7. ábra, a megfejtés

4. A backtrack

A backtrack egy módosítható megoldáskereső módszer, mely állapottér-reprezentált problémák megoldását keresi.

A backtrack három fő összetevőből épül fel: adatbázis, műveletek, vezérlő. Az adatbázisban tároljuk azokat az állapotokat, amelyeket már elértünk, de előfordulhat, hogy a visszalépésnél ismét szükség lesz rájuk.

A műveletek segítségével módosítjuk az adatbázist, a müveleteknek két fő típusuk van. Az operátorok új állapotok létrehozásában segédkeznek, minden operátornak van alkalmazhatósági előfeltétele. A visszalépés pedig egy előző állapotot tölt be újra.

A vezérlő irányítja a keresést, megmondja, hogy az adatbázis mely részén mikor melyik műveletet kell végrehajtani, ő dönti el, hogy melyik állapot végállapot illetve, hogy a probléma megoldható-e. A keresés során az egyes állapotokat egy gráfban tárolja, a gráf csúcsai az állapotok, élei pedig az operátorok, két csúcs között akkor van él, ha az első csúcshoz tartozó állapotból valamilyen operátor segítségével el tudunk jutni a második csúcshoz tartozó állapotba.

A backtrack egyik továbbfejlesztett változatában körfigyelés is van, ez azt jelenti, hogy ha egy olyan állapot állt elő, ami az adatbázisban már szerepelt, akkor azonnal visszalépés történik és az aktuális állapoton egy másik műveletet próbál elvégezni a vezérlő.

Az alap backtrack vezérlőjének működése:

1. inicializálás. Az adatbázis egyetlen csúcsból, a start csúcsból áll.
2. tesztelés. Az aktuális csúcs a végállapotot tartalmazza-e, tehát terminális csúcs-e?

    ha igen: készen vagyunk

    ha nem: tovább tudunk-e lépni, azaz van-e alkalmazható operátor?

        ha igen: akkor választunk egyet és létrehozunk egy új aktuális csúcsot és erre megismételjük a 2. pontot

        ha nem: visszalépés

A visszalépés során egy még nem alkalmazott operátort kell használnunk, ha ilyen nincs akkor újabb visszalépés következik.

A vezérlő működése akkor fejeződik be, ha:

- az aktuális csúcs terminális
- a start csúcsban teljesül a visszalépés feltétele, ekkor nincs megoldás.

A következőkben ismertetett algoritmus a DPLL és a backtrack alapjaira épül, és ezek segítségével végzi a Nonogram rejtvény megoldás keresését.

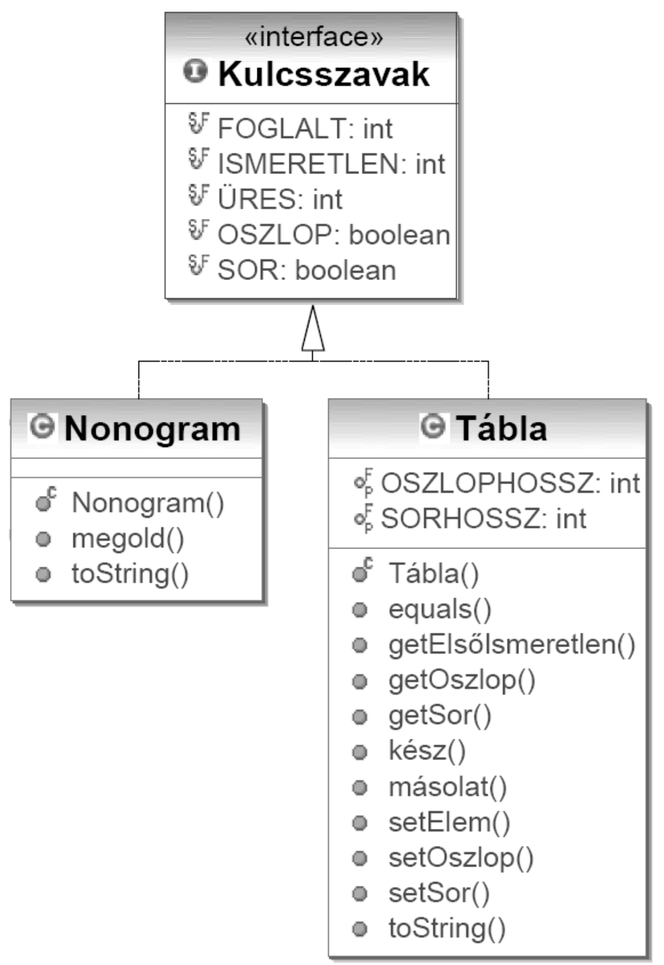
5. A rejtvény implementálása

A programunk feladata egy Nonogram rejtvény megoldása az automatikus tételbizonyítás adta eszközökkel. De mielőtt hozzá kezdenénk azon algoritmus megtervezéséhez, amely a rejtvény megfejtését végzi, el kell gondolnunk, hogy magát a rejtvényt, mint adatok összességét milyen módon kódoljuk le, a válasz egy erre a feladatra szánt osztály megírása lesz.

Az összefüggő sötét blokkok hosszát adó számokat a megfejtést végző osztály, melynek neve `Nonogram,` használja, melyeket majd a konstruktor fog megkapni. Mivel az algoritmus nem egy lépésben fogja a rejtvény összes mezőjéről eldönteni, hogy milyen színű, ezért fontos, hogy a már meghatározott mezők színét tároljuk, amihez a `Tábla` osztályt használjuk.

Az osztály legfontosabb adattagja egy kétdimenziós tömb, melyet csak az osztályban definiált metódusokkal lehet elérni. Az elemei **`int`** típusúak lesznek azért, mert a tábla mezőinak csak a megoldást végző algoritmus utolsó fázisában lesz két fajta értéke (be van színezve tehát `FOGLALT`, nincs beszínezve tehát `ÜRES`), korábban szükség lesz egy harmadikra is, ami azt jelöli, hogy a mezőről még nem tudunk semmit. Kezdetben a tábla összes eleme ilyen értékű lesz.

Azt, hogy melyik **`int`** érték a mező melyik állapotát jelöli, érdemes konstansokkal rögzíteni. Ezeket a konstansokat a megoldás megkeresése során is használjuk, ezért egy interfészben tároljuk őket, aminek a neve `Kulcsszavak.`



8. ábra, a rejtvényt implementáló Tábla, a Kulcsszavak interfész és a megoldás felderítését végző Nonogram

A `Tábla` osztály konstruktorában adjuk meg, hogy a tábla milyen méretű legyen, és itt állítjuk az összes mezőt `ISMERTELEN` értékre is. A `setElem()`, `setSor()`, `setOszlop()`, a tábla módosítását hivatott megkönnyíteni és paraméterül meg kell adni a változtatás helyét a táblában, illetve az új értéket. Ugyancsak ezért vannak a `getSor()` és `getOszlop()` lekérdező metódusok, ahol az adott sor illetve oszlop indexét kell megadnunk. Az `getElsőIsmeretlen()` a rejtvény bal felső sarkából indulva, balról jobbra haladva a legelső sor legelső olyan elemének az indexeit adja meg mely értéke `ISMERETLEN`.

A `toString()`-gel a rejtvényt írhatjuk ki és nyomkövetésre is használhatjuk, amihez jól jön a `kész()` metódus, ami a táblán már megfejtett mezők számát adja meg.

A `Tábla` osztályt a rejtvény megoldását végző `Nonogram` nem csak az algoritmus közben használhatja, hanem a megoldást is ennek segítségével adhatjuk meg, a `megold()` metódussal.

6. A megoldást kereső algoritmus főbb lépései

A megoldást kereső program fejlesztéséhez a korábbi fejezetekben tárgyalt ismeretek birtokában hozzá kezdhetünk. Az első gondolat valószínűleg az lenne, hogy a blokkméretek ismeretében létrehozunk egy műveleti fát, mely a rejtvényt reprezentáló formulát tárolja, és ezen végzünk el különböző műveleteket, az így kapott elemeket pedig letároljuk a `Tábla` osztály segítségével egy kétdimenziós tömbben, ami majd a megfejtett rejtvényt adja vissza. Csakhogy a fát nagyon sokszor kellene átalakítanunk és sokszor kellene benne keresgélnünk, törölnünk, ami idő és erőforrás igényes. Ehelyett egy olyan módszer kerül ismertetésre, amely csak a korábban említett kétdimenziós tömböt használja, és ezt adja eredményül is. Az algoritmus rekurziót használ, paraméterei a kétdimenziós tömb, melynek elemei az első hívás esetén ismeretlenek, és a blokkméretek. Az algoritmus lépései:

1. A sorokhoz és oszlopokhoz tartozó blokkméretek alapján meghatározzuk csak a sorokat és oszlopokat önállóan tekintve az átfedéseket, és a már biztosan ismert mezőket eltároljuk egy olyan kétdimenziós tömbben, mely akkora méretű, mint maga a rejtvény. Ha ebben a tömbben az adott sorra, vagy oszlopra vonatkozóan már van információ (például tudjuk, hogy az adott sor első eleme biztos, hogy be van színezve), akkor ezt is felhasználjuk az átfedések megkereséséhez.

2. Az 1. lépést addig ismételjük, míg a végrehajtás előtt felderített elemek száma meg nem egyezik a végrehajtás után felderített elemek számával, vagy az átfedés során egy elemről olyan állítás kerül napvilágra, ami a korábban letárolt tömb béli értékével nem egyezik meg. Utóbbi esetben a kétdimenziós tömb azt a változatát adja vissza az algoritmus, amellyel meghívtuk, és az algoritmus véget is ér.

3. Ha a kétdimenziós tömb minden eleme be van színezve, akkor ez a tömb adja a megfejtett rejtvényt és készen vagyunk.

4. Ha nem, akkor a tömbről készítünk egy másolatot és ebben a másolatban kiválasztunk egy olyan elemet, amelynek nem ismerjük az értékét és beállítjuk beszínezettre ezután ismét meghívjuk az algoritmust a másolattal.

5. Ami ha olyan tömböt ad vissza, amelynek minden eleme ismert, akkor ez a megoldás.

6. Ha nem, akkor az eredeti tömb ugyan ezen eleméről úgy határozunk, hogy nincs beszínezve és az így kapott tömbbel ismét meghívjuk az algoritmust, melynek eredménye biztos, hogy a kész rejtvényt fogja megadni, tehát ezt adja vissza a jelenlegi algoritmus is.

Ha egy backtrack keresőként nézzük az algoritmust, akkor a probléma egy állapota egy kétdimenziós tömb.

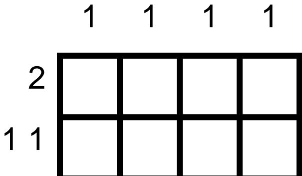
Három operátor áll rendelkezésünkre:

- átfedések meghatározása, melynek előfeltétele, hogy a tömbben minden sorra és oszlopra létezik legalább egy variáció
- egy elem beállítása FOGLALT-ra, előfeltétele, hogy az előző operátor alkalmazása esetén a felderített elemek száma nem változik, továbbá, hogy legyen legalább egy ISMERETLEN értékű elem, több ilyen esetén alapértelmezettként a beszínezendő legyen felülről lefele és balról jobbra haladva az első
- egy elem beálítása ÜRES-re, előfeltétele megegyezik az előző operátoréval

A vezérlő először mindig az első operátort próbálja alkalmazni, azután a másodikat és a harmadikat.

A vezérlőben van körfigyelés is, hiszen az átfedések meghatározása során előfordulhat, hogy nem derítünk fel újabb elemet. Hogy egy állapot végállapot-e, azt úgy ellenőrzi, hogy minden elem ismert-e és, hogy alkalmazható-e rá az átfedések meghatározása, tehát a felderített elemek megfelelnek-e valamelyik variációnak.

Az algoritmus működését szemlélteti a következő egyszerű példa. Legyen a rejtvény a következő:



9.a ábra, példa a megoldást kereső algoritmusra

Ehhez hajtsuk végre az algoritmust a megfelelő blokkméretek és kétdimenziós tömb megadásával, utóbbi 2×4-es méretű és minden eleme ismeretlen, a tömb elemei kezdetben tehát:

?	?	?	?
?	?	?	?

9.b ábra, a tömb kezdeti paraméterei

A tömb beszínezett elemeit jelölje 1, a nem beszinezetteket pedig 0.

Az algoritmus 1. lépése szerint meg kell keresnünk az átfedéseket, azonban miután végignézzük a sorokhoz és az oszlopokhoz tartozó blokkméreteket, nem találunk egyet sem. A 2. lépés szerint az átvizsgálást addig kell folytatnunk, amíg ellentmondásba nem botlunk, vagy az átvizsgálás előtt felderített elemek száma, meg nem egyezik az átvizsgálás után felderített elemek számával. A 3. lépést kihagyhatjuk, mert nem derítettünk fel egy elemet sem.

A 4. lépés szerint egy tetszőleges ismeretlen elemet kiválasztunk, és az értékét beállítjuk beszínezettre, ezután ezzel ismét meghívjuk az algoritmust. Válasszuk a legelső elemét a tömbnek, amit beszínezünk. A tömb tehát, amivel ismét meghívjuk az algoritmust:

1	?	?	?
?	?	?	?

9.c ábra, a tömb elemei a 4. lépés után

Az új algoritmust is az 1. lépésnél kezdjük, a sorokhoz tartozó blokkméretek átvizsgálása után a tömb:

1	1	0	0
?	?	?	?

9.d ábra, az új algoritmus hatása a tömbre

Miután végig néztük az oszlopokat is:

1	1	0	0
0	0	1	1

9.e ábra, az új algoritmus hatása a tömbre

Most minden mezőt kitöltöttünk, de a 2. lépés szerint még egyszer el kell végeznünk az 1. lépést, ami során ki derül, hogy a második sorra nem tudunk olyan variációt találni, ami ne okozna ellentmondást a korábban letárolt tömb béli elemekkel, tehát az új algoritmus véget ér és visszaadja a kapott tömböt.

Az első algoritmus így a 6. lépés szerint a tömb legelső elemét nem beszínezettre állítja és egy másik algoritmust hív, a tömb alakja tehát:

0	?	?	?
?	?	?	?

9.f ábra, a tömb elemei a 6. lépés után

A másik algoritmus az 1. lépésben a sorok átvizsgálása után:

0	?	1	?
?	?	?	?

9.g ábra, az új algoritmus hatása a tömbre

Az oszlopok átvizsgálása után:

0	?	1	?
1	?	0	?

9.h ábra, az új algoritmus hatása a tömbre

A 2. lépés szerint az elsőt megismételjük. Az első ismétlés eredménye:

0	1	1	0
1	0	0	1

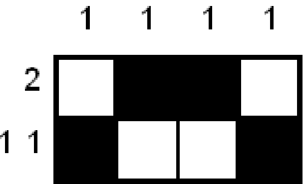
9.i ábra, az 1. ismétlés hatása a tömbre

A másodiké:

0	1	1	0
1	0	0	1

9.j ábra, a 2. ismétlés hatása a tömbre

A 3. lépés szerint ezt a tömböt adja vissza ez az algoritmus és az első algoritmus is, így készen vagyunk. A rejtvény a tömb alapján tehát:

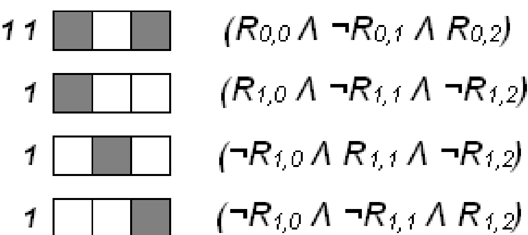


9.k ábra, a rejtvény megoldása

7. Az átfedések meghatározása és a felderített mezők felhasználása

Mivel egy Nonogram rejtvénynél az egyes sorokra és oszlopokra vonatkozó információkkal dolgozunk így érdemes nekünk is csak sorokban és oszlopokban gondolkodni az egész rejtvény helyett.

Világos, hogy egyetlen sor vagy oszlop egyetlen változatát reprezentáló formula önmagában konjunktív normálforma, de nem mindig adja meg a sorhoz/oszlophoz tartozó összes változó összes eshetőségét, ehhez szükségünk van a blokk méretek figyelembevételével előállított összes variációra, melyeket diszjunkcióval kapcsolunk össze. Például egy olyan rejtvénynél ahol egy sor 3 mező hosszú és két 1 hosszúságú blokkot tartalmaz, akkor a $(R_{0,0} \land \neg R_{0,1} \land R_{0,2})$ részformula a sorhoz tartozó változók összes lehetséges előfordulását azért adja meg, mert a két 1 hosszúságú blokkot csak egy féle képen lehet elhelyezni. Ellenben ha csak egy darab 1 hosszúságú blokk szerepelne, akkor több változat is elképzelhető. Ezeket az $(R_{1,0} \land \neg R_{1,1} \land \neg R_{1,2})$, $(\neg R_{1,0} \land R_{1,1} \land \neg R_{1,2})$ és a $(\neg R_{1,0} \land \neg R_{1,1} \land R_{1,2})$ formulák reprezentálják és mivel ezek között „vagy” kapcsolat áll fenn, ezért kell diszjunkcióval összekötni őket.



10. ábra, a fent említett egyszerű rejtvények, és az őket reprezentáló részformulák

A sorokhoz/oszlopokhoz tartozó variációk összességét ismét csak konjunkcióval kapcsoljuk össze, ami magát a rejtvényt adja, és mivel a rejtvény biztos, hogy kielégíthető, ezért a variációk összessége, önmagukban is kielégítető, hiszen a konjunkció művelete csak akkor ad igaz értéket, ha a művelet tagjai mind

igazak. Ezért a *($R_{0,0}$ Λ ¬$R_{0,1}$ Λ $R_{0,2}$)* részformula és a *($R_{1,0}$ Λ ¬$R_{1,1}$ Λ ¬$R_{1,2}$) V (¬$R_{1,0}$ Λ $R_{1,1}$ Λ ¬$R_{1,2}$) V (¬$R_{1,0}$ Λ ¬$R_{1,1}$ Λ $R_{1,2}$)* részformula is biztos, hogy kielégíthető. Tehát elég a variációk összességét megadó formulákat KNF-re hozni és az itt egyedül álló változókról már nyilatkozni tudunk, ezért elegendő, ha a fentebb említett algoritmus első lépésében a sorokat és az oszlopokat önállóan, tehát a többi sortól és oszloptól külön vizsgáljuk. Az első feladat a megoldást kereső algoritmus fejlesztésénél ezért egy olyan kódrészlet megírása, amely adott blokkméretekhez és sor vagy oszlop hosszhoz elkészíti az összes változatot.

7.1 A variációk előállítása

A megoldást a `Nonogram` osztály végzi, ezért a kódrészletet is ide kell megírni, ami egy metódus lesz. A sorokhoz/oszlopokhoz tartozó formulák tárolására egy vektort használunk, ami az osztály adattagja nem pedig a metódusé, ennek neve legyen `átfedések`. A vektor annyi elemből áll ahány változóból a formula, az elemeknek három féle értékük lehet: `ISMERETLEN`, `FOGLALT` (beszínezett), `ÜRES` (nem beszínezett). Egy vektor elemei között alapértelmezetten a konjunkció művelete áll.

A feladatunk tehát az egyes sorokhoz/oszlopokhoz tartozó variációkat jelölő vektorok elkészítése. A metódus neve legyen **`variáció()`** paraméterei:

- egy logikai változó, amely eldönti, hogy egy sornak, vagy egy oszlopnak a variációira van szükség.
- annak a sornak vagy oszlopnak az indexe, amelynek a variációit elő akarjuk állítani.
- egy vektor, melyben a már beállított változó értékeket tároljuk, ennek minden eleme kezdetben `ÜRES`
- a fent említett vektornál meg kell adnunk, hogy hányadik elemig vannak már lekötve az értékek
- meg kell adnunk, hogy most hányadik blokknak a variációit akarjuk a fenti vektorban letárolni.

Az egyes sorokhoz/oszlopokhoz tartozó blokkméretek két összetett listában vannak tárolva, melyek szintén az osztály adattagjai. Ezen változók feltöltése a

megoldást végző osztály konstruktorának feladata, vagyis a konstruktor paraméterei a blokkméretek lesznek. A vektor és a lista elemeit 0-tól indexeljük.

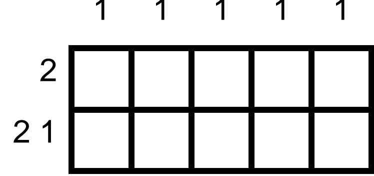
A variációk előállítását végző metódus rekurziót fog használni, a metódus főbb lépéseit a következő algoritmus mutatja be:

1. Meghatározzuk a logikai változó és az index alapján, hogy mely blokkméretekkel fogunk dolgozni.
2. Ha az adott sorhoz/oszlophoz tartozó blokkméretek darabszámával megegyező index van megadva paraméterül, akkor készen vagyunk és az osztályban szereplő vektor értékeit be kell állítanunk a paraméterül kapott vektor értékeire.
3. Ha nem, akkor a következő képlet alapján meghatározzuk az adott blokkmérethez, hogy a paraméterül kapott vektor másolataiban hányadik indexig helyezhetjük majd el a jelenlegi blokkot:

$$hátralévő = vektor\ hossza - \sum_{i=aktuális\ blokkméret\ indexe+1}^{blokkméretek\ darabszáma-1}(blokkméret_i + 1) + 1$$

4. Ezután egy ciklusban a vektorról másolatot készítünk és az aktuális blokk változataival kiegészítjük, figyelembe véve a vektor utolsó lekötött elem indexét és a $hátralévő$ által meghatározott indexet, utóbbi azt jelenti, hogy az aktuális blokk utolsó elemének indexe mindig kisebb kell, hogy legyen, mint $hátralévő\text{-}1$. Majd még szintén a ciklusban újból meghívjuk az algoritmust az éppen aktuális másolattal, a másolatban utoljára lekötött elem indexének és a most aktuális blokkméret indexének egyel megnövelt értékével.

Az algoritmus működésére nézzünk egy példát, állítsuk elő a következő rejtvény második sorának variációit:



11. ábra, példa a variációk előállítására

Az algoritmust tehát egy sorra hívjuk meg, a sor indexe, mivel 0-tól indexelünk így 1 lesz, a vektor 5 elemű lesz minden értéke ÜRES, amit jelöljön 0, a FOGLALT elemeket jelölje 1. Még egy elemről sem döntöttünk így az erre vonatkozó paraméter értéke 0, és az első blokk variációit akarjuk előállítani, aminek az indexe szintén 0.

Az 1. lépés szerint meghatározzuk a megfelelő blokkméreteket melyek a 2, 1 számok. A 2. lépés szerint mivel 2 a sorhoz tartozó blokkméretek száma és mi még csak a 0-dikat vizsgáljuk így tovább kell haladnunk. A 3. lépésben a képlet alapján meghatározzuk *hátralévő* értékét:

$$hátralévő = 5 - (1 + 1) + 1 = 4$$

A 4. lépésben 0 és 4 között előállítjuk a paraméterül kapott vektor másolataiban a blokk variációit és meghívjuk újból az algoritmust, a következő blokkra. A másolatok a módosítás után:

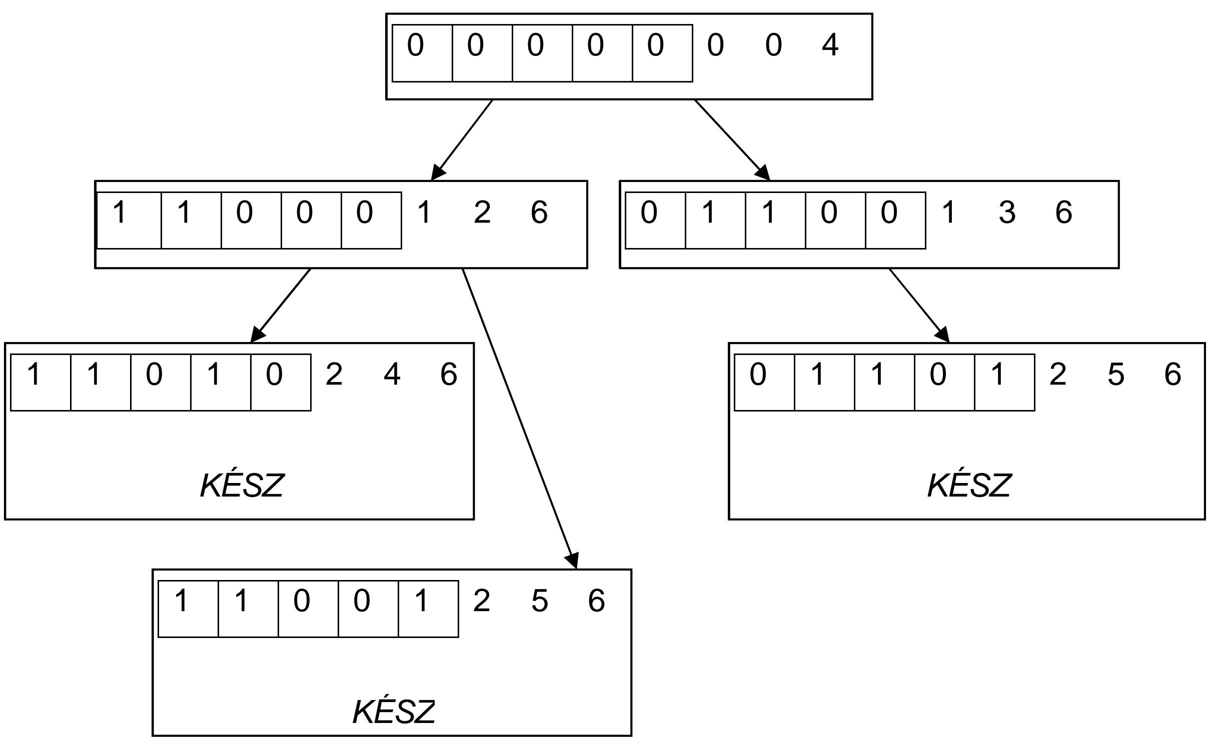
1	1	0	0	0

az utoljára lekötött blokk indexe 2

0	1	1	0	0

az utoljára lekötött blokk indexe 3

Az utoljára lekötött blokk indexe azért nagyobb egyel a letárolt blokk utolsó elemének indexénél, mert két blokk között legalább egy üres mezőnek is állnia kell. Az új algoritmushívások paraméter vektorait a következő fa szerkezet mutatja be, feltüntetve még az aktuális blokkméret indexét, az utoljára lekötött blokk indexét és *hátralévő* értékét:



12. ábra az algoritmus során a variációk alakulása

7.2 Az átfedések meghatározása

Miután előállítottuk az összes lehetséges variációt, következő lépés az átfedések meghatározása. Az első esetben tegyük fel, hogy a vizsgált sorban/oszlopban egyetlen mező sincs felderítve, tehát nincsenek korábbról információink.

Az átfedések megkereséséhez a variációkból alkotott formulákat hozzuk KNF-re. Tekintsük egy sor/oszlop variációinak összességét:

$$(A_1^1 \wedge \ldots \wedge A_n^1) \vee (A_1^2 \wedge \ldots \wedge A_n^2) \vee \ldots \vee (A_1^m \wedge \ldots \wedge A_n^m)$$

Ahol az *A*-k olyan literálok, melyeknél az alsó index adja meg, hogy az adott sorban/oszlopban hányadik elemet jelölik, a felső index pedig annak a variációnak a számát, amelyikben előfordulnak. Bár a formula első látásra bonyolultnak tűnik, de ha jobban megfigyeljük, észrevesszük, hogy a konjunkció művelete csak a zárójeleken belül van, diszjunkció pedig csak két zárójel között, mi pedig azt

szeretnénk elérni, hogy ez fordítva legyen. A KNF-re alakításhoz a disztributivitási törvényeket használjuk, melyek nem csak a diszjunkció és konjunkció között állnak fenn, hanem korábbi ismereteink szerint a szorzás és összeadás között is:

$A \vee (B \wedge C) \sim (A \vee B) \wedge (A \vee C)$ $\qquad$ $a * (b + c) = (a * b) + (a * c)$

Ha ez alapján írjuk át a formulát:

$$(A_1^1 + \ldots + A_n^1) * (A_1^2 + \ldots + A_n^2) * \ldots * (A_1^m + \ldots + A_n^m)$$

Így a formula átalakítása sokkal közérthetőbb, hiszen lényegében csak annyi a feladatunk, hogy minden tagot minden taggal „összeszorozzunk”:

$$(A_1^1 + \ldots + A_n^1) * (A_1^2 + \ldots + A_n^2) * \ldots * (A_1^m + \ldots + A_n^m) =$$
$$= (A_1^1 * \ldots * A_1^n) + \ldots + (A_n^1 * \ldots * A_n^m)$$

Tehát a formula KNF-ben:

$$(A_1^1 \vee \ldots \vee A_1^n) \wedge \ldots \wedge (A_n^1 \vee \ldots \vee A_n^m)$$

Világos, hogy az így kapott klózokból azok fognak egyetlen ítéletváltozót tartalmazni, amelyek csak egy mezőhöz tartozó variációkat tartalmaznak, és ha ezek a változók azonos értékű, ami azt jelenti, hogy vagy minden variációban szerepel előttük a negáció művelete, vagy semelyikben sem, ekkor ugyanis használhatjuk az $A \vee A \sim A$ szabályt a klózban.

Ha egy műveletet egy változó áthelyezésének tekintünk, az átalakítás n^m műveletet igényel, mivel a második variáció első változóját n helyre kell áthelyezni, és a variáció is n elemű, ezért n*n áthelyezés történik itt. A harmadik variáció elemeit viszont n*n helyre kell áthelyezni így itt n*n*n művelet történik és így tovább. Viszont ha csak az átalakítás azon részét végezzük el amikor a variációkon az adott indexű változókat ugyan ahhoz az indexű másik változóhoz helyezzük át, akkor n*m művelet szükséges.

Az átfedések meghatározásánál feladatunk tehát a variációk segítségével ezen klózok meghatározása. Mivel a variációkat vektorban tároljuk így igen könnyű dolgunk van, hiszen csak a megfelelő indexű elemeket kell összevetnünk, és az eredmény egy olyan vektor lesz, amelynek egy bizonyos indexű eleme csak akkor lesz különböző `ISMERETLEN`-től, ha az összes vektorban az adott indexhez tartozó elem azonos értéket vesz fel. Azonban nem szabad elfelejtenünk, hogy csak azonos blokkok közötti átfedést kell figyelembe vennünk, tehát a vektorokban a blokkokat meg kell különböztetnünk egymástól, ezért ha a vektor egy eleme azt jelöli, hogy a variációban az adott mező be van színezve, akkor az elem értéke legyen a blokk sorsszáma egyel megnövelve (mivel 0-tól indexelünk és a 0 már a nem beszínezett mezőket jelöli).

Az összehasonlítást a következő algoritmus mutatja be, mely két vektort kap paraméterül, melyek közül első a `Nonogram` osztályban már korábban definiált `átfedések`, a második pedig egy variáció, feltételezzük, hogy a vektorok egyforma hosszúak, az algoritmushoz tartozó metódus neve legyen **`átfedés()`**:

1. Ha az első vektor `null`, akkor a második vektor elemeire állítjuk az `átfedések` elemeit.
2. Ellenkező esetben készítünk egy új vektort melynek kezdetben minden eleme `ISMERETLEN`, és hossza ugyan annyi, mint a paramétereké.
3. A két paraméter elemeit sorra vesszük és összehasonlítjuk, ha egyezés van, akkor az új vektor elemét az adott indexen erre állítjuk be.
4. az `átfedések` elemeit beállítjuk az új vektor elemeire.

A variációk előállításánál is az `átfedések` vektorban tároltuk az éppen aktuális variációt, ezért használjuk fel itt is. Így nem kell minden egyes variációt külön letárolni, hanem ha előállítottunk kettőt, akkor azonnal megnézhetjük az átfedéseket, csak arra kell vigyázni, hogy mielőtt egy sor/oszlop variációinak elkészítéséhez hozzá kezdenénk, a vektor legyen `null`.

Az algoritmus működésének bemutatásához nézzük, hogyan állítja elő az előző példa variációiból az átfedéseket tartalmazó vektort. Az egyes variációk:

1	1	0	2	0

1	1	0	0	2

0	1	1	0	2

13. ábra, az előző példa variációi

Minden egyes alkalommal, ha előállt egy variáció, az algoritmus lefut. Az első variáció után az `átfedések` értéke még `null` ezért az algoritmus 1. lépése szerint az első variáció elemeinek értékére állítjuk.

A második variáció előállítása esetén így az `átfedések` már nem `null` értékű ezért létrehozunk egy új vektort melynek minden eleme `ISMERETLEN`, majd elvégezzük az összehasonlításokat:

`átfedések`:	1	1	0	2	0
a második variáció:	1	1	0	0	2
az új vektor:	1	1	0	?	?

A 3. lépés szerint az új vektor elemeire állítjuk az `átfedések`-et.

A harmadik variáció után végrehajtva az algoritmust:

`átfedések`:	1	1	0	?	?
a harmadik variáció:	0	1	1	0	2
az új vektor:	?	1	?	?	?

Tehát az `átfedések` értéke a variációk összehasonlítása után:

?	1	?	?	?

Ami azt jelenti, hogy csak a második mezőről tudtuk meg, hogy be van színezve. Észrevehető, hogy az átfedések megkeresése közben, magában az `átfedések` vektorban az `ISMERETLEN` mezők száma minden egyes variáció után csak nőhet, és soha sem csökkenhet.

Az **`átfedés()`** metódus kódját az **1. melléklet** tartalmazza.

7.3 A felderített mezők felhasználása

Tekintsük a következő rejtvényrészletet:

1		

14. ábra, példa a felderített mezők felhasználására

Tegyük fel, hogy az első mezőről tudjuk, hogy nincs beszínezve. A variációkat reprezentáló formula:

$$(R_{0,0} \land \neg R_{0,1}) \lor (\neg R_{0,0} \land R_{0,1})$$

És ha ehhez még hozzá vesszük azt is, hogy az első mező biztos nincs beszínezve:

$$[(R_{0,0} \land \neg R_{0,1}) \lor (\neg R_{0,0} \land R_{0,1})] \land \neg R_{0,0}$$

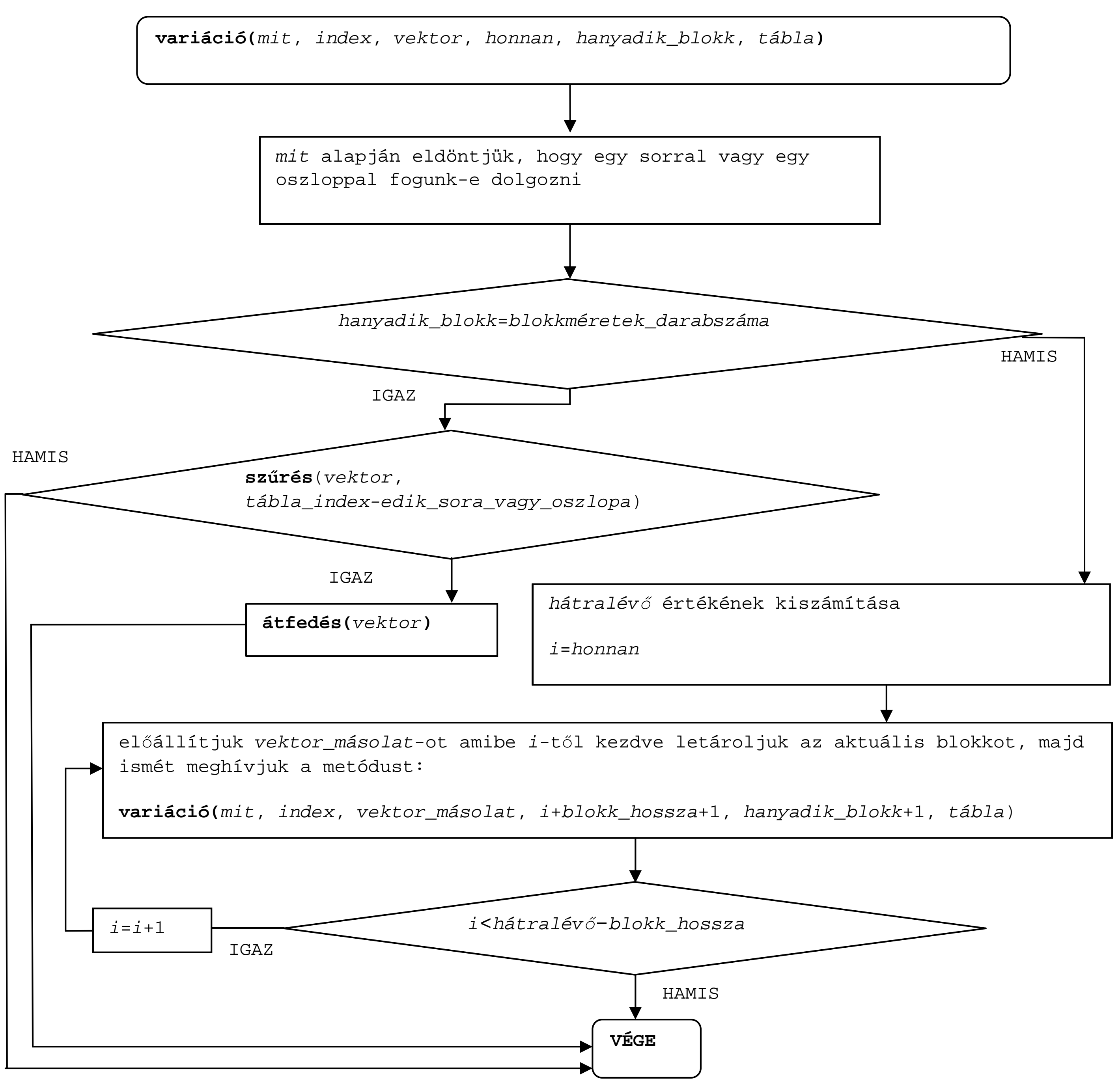
Tudjuk, hogy a variációk összessége kielégíthető, de azt is, hogy csak egy kivételével az összes variáció kielégíthetetlen. Az a tény hogy tudjuk, hogy az első

elem biztos nincs beszínezve, abban segít nekünk, hogy eldöntsük, mely variációk kielégíthetetlenek, hiszen azonnal észrevehető, hogy ha $|\neg R_{0,0}| = I$ akkor $|(R_{0,0} \wedge \neg R_{0,1})| = H$ tehát ez a variáció biztos hamis ezért az átfedések megkeresésénél kihagyhatjuk.

Előfordulhat, hogy az összes variáció hamisnak bizonyul, ez nem feltétlenül jelenti azt, hogy az egész rejtvény kielégíthetetlen, csupán azt, hogy a megoldás keresése során, rossz úton indultunk el és vissza kell fordulnunk.

Tehát a korábban megszerzett információkat az adott sorral vagy oszloppal kapcsolatban az után érdemes felhasználni, hogy készen vagyunk egy variációval, és el szeretnénk dönteni, hogy ezt a variációt felhasználjuk-e az átfedések megkeresésekor. Ezt egy olyan metódus végzi el, aminek neve legyen **`szűrés()`**, és két vektort kap paraméterül, az első a kétdimenziós tömbben már korábban letárolt információkat őrzi, értelem szerűen egy mező értéke `ISMERETELEN` lesz, ha nem tudunk róla semmit, a második pedig az éppen aktuális variáció. A metódus egy logikai értéket ad vissza, igazat, ha a második vektor adott indexű eleme minden esetben megegyezik az első vektor adott indexű elemével amennyiben az `ISMERETLEN`-től különböző, más esetben hamisat.

A **`szűrés()`** metódus kódját a **2. melléklet** tartalmazza.



15. ábra, a variáció() metódus folyamat ábrája mely tartalmazza a szűrés() és az átfedés() metódus használatát is

A `variáció()` metódus kódját a **4. melléklet** tartalmazza.

7.4 További tevékenységek

A korábban említett három metódus: a `variáció()`, `átfedés()`, `szűrés()` felhasználásával keressük meg a rejtvényben az átfedéseket. Ezen metódusok szabályos működéséhez azonban további tevékenységeket is el kell végeznünk, hisz például az `átfedések` változót minden sornál, oszlopnál `null` értékre kell állítanunk, és a felderített elemeket is le kell még tárolnunk a kétdimenziós tömbben.

Ezeket a tevékenységeket három metódus végzi el:

`összesVariáció()`: paraméterül kap egy kétdimenziós tömböt, ami a rejtvény aktuális állapotát őrzi, egy logikai változót, ami azt mondja meg, hogy sorban, vagy oszlopban fogunk- e dolgozni, illetve a sor vagy oszlop indexét.

Feladata a kapott tömb manipulálása, ami a metódus egy „mellékhatása”. Legelőször is, azt a kivételes esetet vizsgálja, hogy az adott sorban, oszlopban van-e egyáltalán elhelyezendő blokk, ha nincs, akkor a kétdimenziós tömb megfelelő elemeit `ÜRES`-re állítja be. Ha van, akkor az `átfedések` értékét `null`-ra állítja és készít egy `ÜRES` elemeket tartalmazó vektort, amellyel meghívja a `variáció()` metódust. Természetesen a megfelelő indexel, logikai értékkel stb. Miután a `variáció()` lefutott, megvizsgálja az `átfedések` változót, aminek ha az értéke `null`, akkor dob egy kivételt, jelezve, hogy a megoldás keresés megakadt, ha nem `null` akkor a megfelelő sor, oszlop elemeit `átfedések` elemeire változtatja.

`kitölt()`: paraméterül kap egy kétdimenziós tömböt, és feladata, hogy az `összesVariáció()` metódust minden sorra és oszlopra meghívja.

`dpll()`: paraméterül kap egy kétdimenziós tömböt, és az a feladata, hogy a `kitölt()` metódust addig hívja meg erre a tömbre, amíg az abban felderített elemek száma változik, ez a metódus eredményül adja az új tömböt, nem pedig mellékhatásaként változtat rajta.

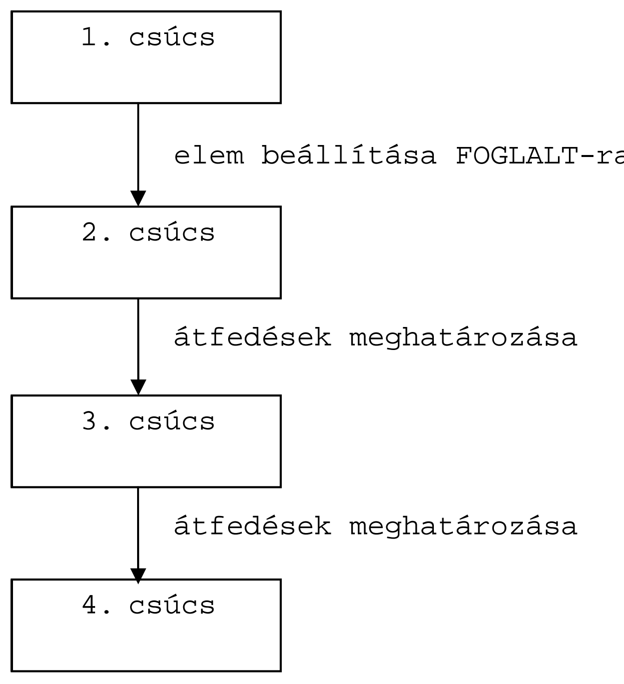
Backtrack keresőként tekintve a programot, a `dpll()` valósítja meg a vezérlőt, ez hívja meg az operátorokat és ez ellenőrzi, hogy a gráfban keletkezett-e kör. A továbbiakban pedig a `dpll()` feladata lesz majd a másik két operátor meghívása és annak eldöntése, hogy mikor érkeztünk el a végállapothoz.

Az `összesVariáció()` metódus kódját a **3. melléklet** tartalmazza.

8. Új állapot létrehozása

Backtrack keresőként tekintve az algoritmust, az átfedések meghatározása operátor alkalmazásának következménye egy új állapot és egy új aktuális csúcs melyet az adatbázisban kell eltárolnunk, mint a keresés során használt gráf egy csúcsát.

A programnak azonban nem kell minden egyes állapotot megjegyezni arra az esetre, ha a visszalépés műveletét választanánk. Csak azokat a csúcsokat kell letárolni, amelyekre a második vagy harmadik operátort alkalmazzuk. Ugyanis ezen operátorok előtt biztos, hogy nem használunk visszalépést csak kör esetén, de a kör mindig csak két ugyan olyan csúcsból áll. Ha ezen operátorok után találunk kört akkor, pedig biztos, hogy addig fogunk visszalépni, míg el nem jutunk egy olyan csúcsba melyre a második vagy a harmadik operátort alkalmaztuk. Ezt szemlélteti a következő ábra és a hozzá tartozó magyarázat:



16. ábra, példa a visszalépésre

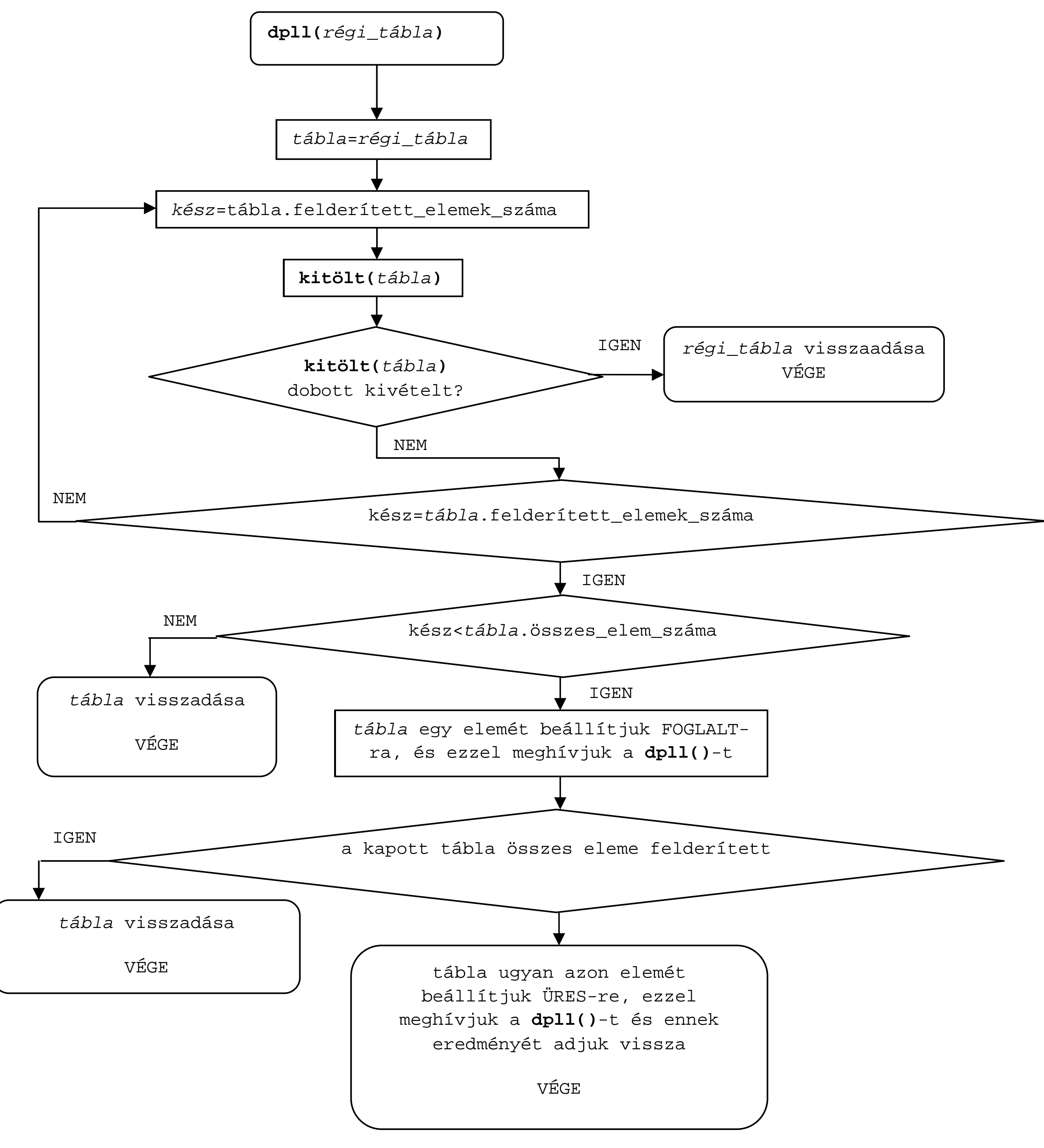
Tegyük fel, hogy az első csúcsban tárolt állapotra az átfedések meghatározása operátor nem derít fel új elemet, ekkor a második operátor alkalmazhatóvá válik, és ennek eredménye a második csúcs.

A második csúcsra alkalmazzuk az első operátort, aminek eredménye a harmadik csúcs amire ismét alkalmazzuk az első operátort aminek eredménye negyedik csúcs, a három állapotban a felderített elemek száma eltérő.

Tegyük fel, hogy a negyedik csúcsban nem tudunk alkalmazni egyetlen operátort sem, tehát vissza kell lépnünk, így a harmadik csúcs lesz az aktuális csúcs. Itt most első lépésként meg kell néznünk, hogy mely operátorokat alkalmaztuk már, és melyeket alkalmazhatjuk még. A harmadik csúcsban az első operátort, az átfedések meghatározását végző operátort használtuk, tehát ezt már nem használhatjuk, mert akkor visszajutunk a negyedik csúcsba. Csakhogy a másik két operátort sem használhatjuk, mert azok alkalmazhatósági feltételében szerepelt, hogy az első operátor használata után a felderített elemek száma nem változik, de ez nem teljesül, ezért a harmadik csúcsból is vissza kell lépnünk a másodikba, ahol ugyan ez a helyzet, így visszajutunk az első csúcsba.

8.1 Új állapot létrehozása a programban

Az operátorok megvalósítása `dpll()`-ben történjen, így nem kell külön ellenőrizni az alkalmazhatósági feltételt, hanem ha kört észlelünk a gráfban, akkor alkalmazhatjuk a második és harmadik operátort. Ez azt jelenti, hogy a metódust ismét meghívjuk a kapott kétdimenziós tömb módosított változatával és az új metódus eredményét vizsgáljuk, így nem kell csúcsonként letárolni, hogy mely operátorokat használtuk már. Itt fontos, hogy a paraméterül kapott tömbről először másolatot készítsünk és csak a másolatot manipuláljuk, mert így elkerüljük, hogy a paraméter tartalma megváltozzon, hiszen ha a paraméter egy objektum, akkor az attribútumainak értékéről a metódus nem készít másolatot, tehát a változtatás hatása meg marad a metódus futása után is.



17. ábra, a dpll() folyamat ábrája

A `dpll()` metódus kódját az **5. melléklet** tartalmazza.

A Nonogram osztálynak most már csak egyetlen metódusát kell implementálni, ami a `megold()` nevet viseli, és feladata a megoldás keresés elindítása és az eredmény szolgáltatása, alakja:

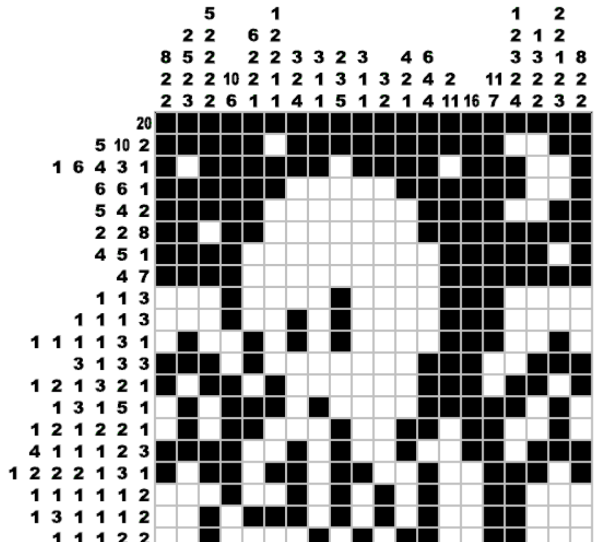
```
public Tábla megold(){
    return dpll(new Tábla(sor.size(),oszlop.size()));
}
```

9. Futási eredmények

Ez a fejezet a `Nonogram` osztály idő és memória használatát mutatja be néhány példán keresztül:

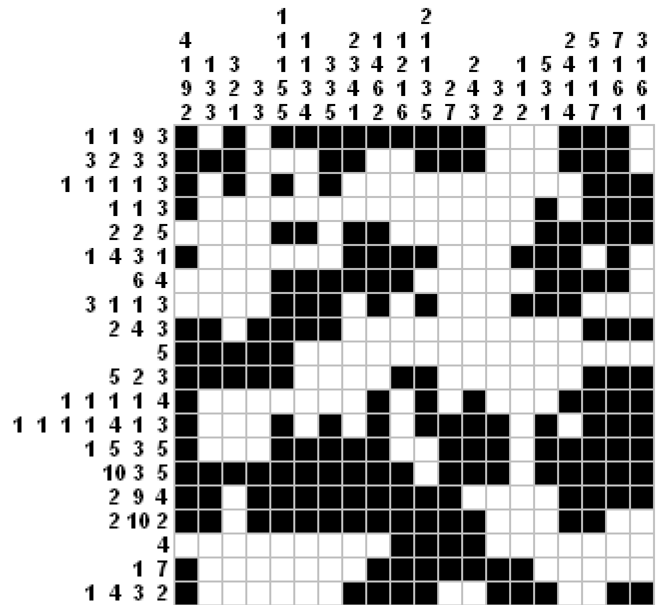
Futási idő(ms):	~0.00
Lefoglalt bájtok száma:	282 688
Használt Tábla objektumok száma:	2

18. ábra, egy 8×8-as rejtvény



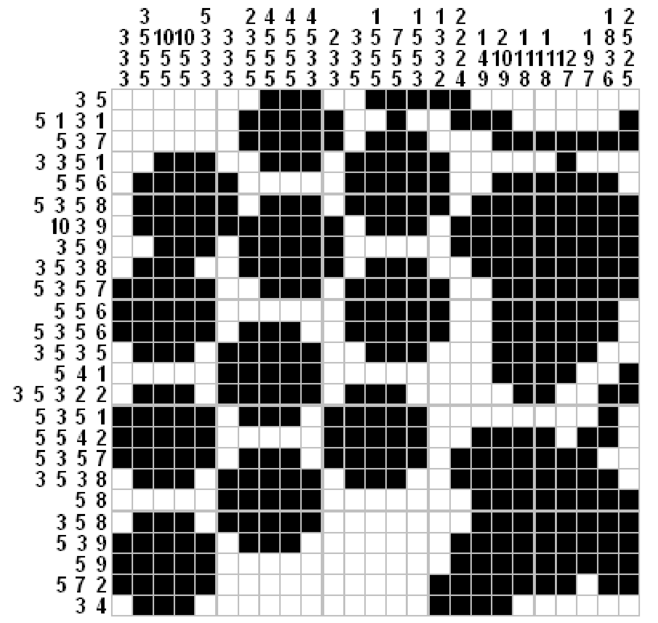
19. ábra, egy 20×20-as rejtvény

Futási idő(ms):	~250.00
Lefoglalt bájtok száma:	70 352 915
Használt Tábla objektumok száma:	2



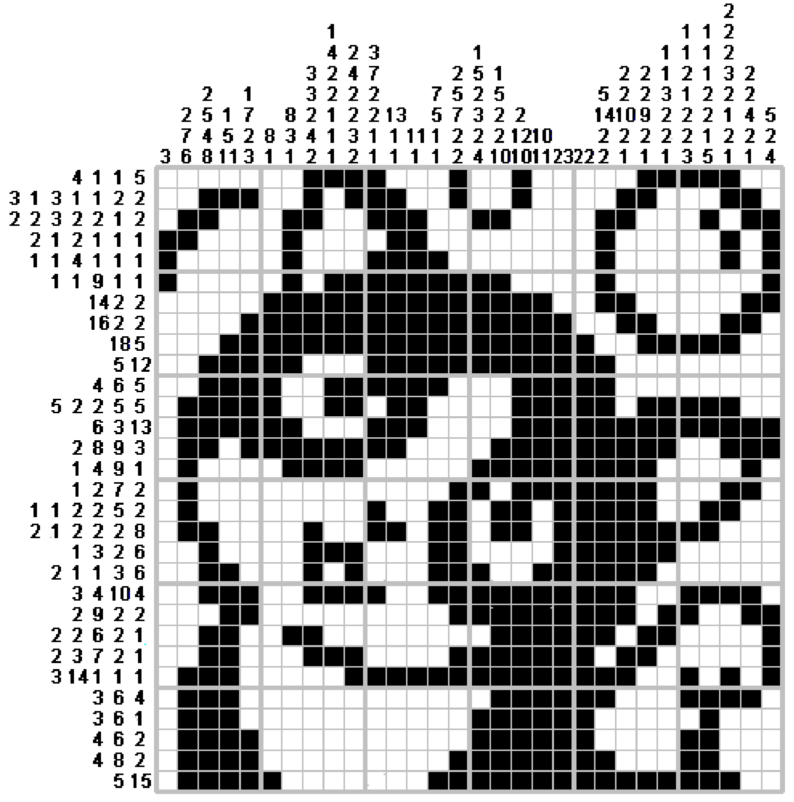
20. ábra, még egy 20×20-as rejtvény

Futási idő(ms):	~65.00
Lefoglalt bájtok száma:	18 186 432
Használt Tábla objektumok száma:	2



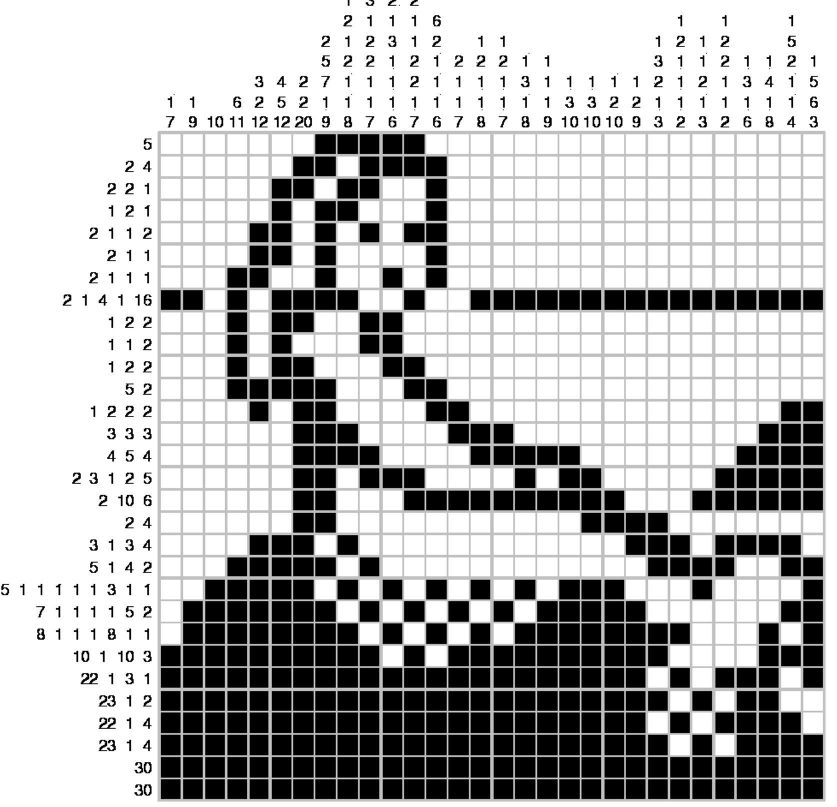
21. ábra, egy 25×25-ös rejtvény

Futási idő(ms):	~120.00
Lefoglalt bájtok száma:	36 981 968
Használt Tábla objektumok száma:	2



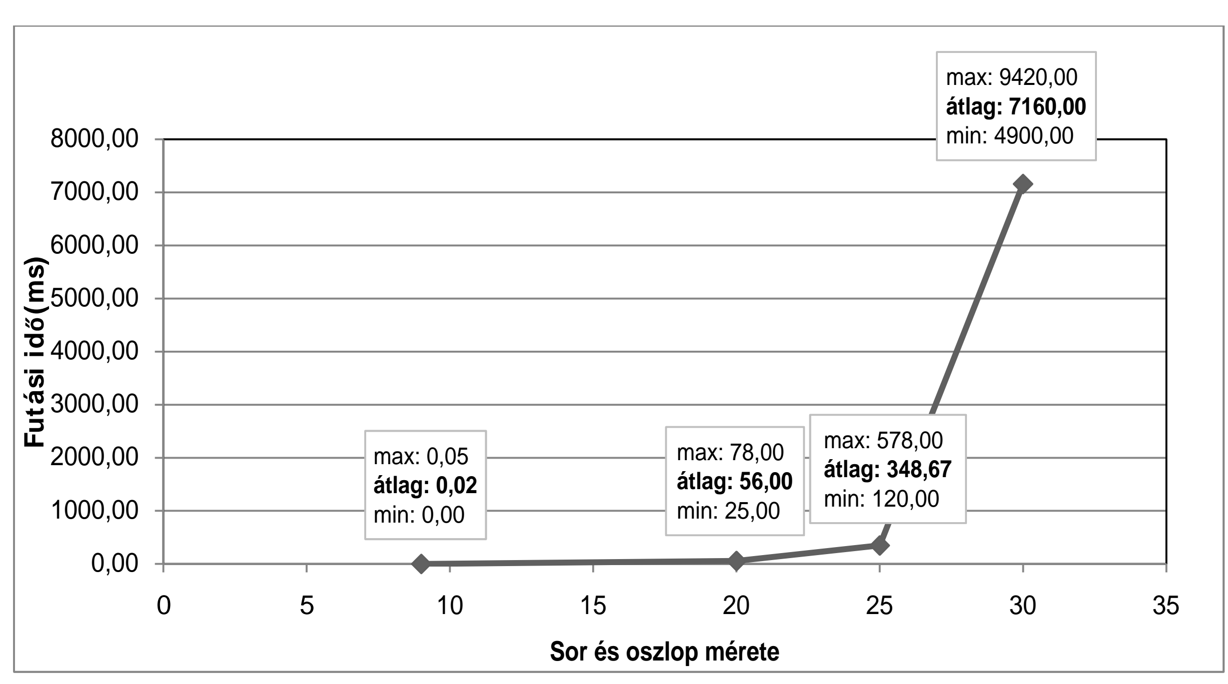
Futási idő(ms):	~9 420.00
Lefoglalt bájtok száma:	3 696 614 736
Használt Tábla objektumok száma:	4

22. ábra, egy 30×30-as rejtvény

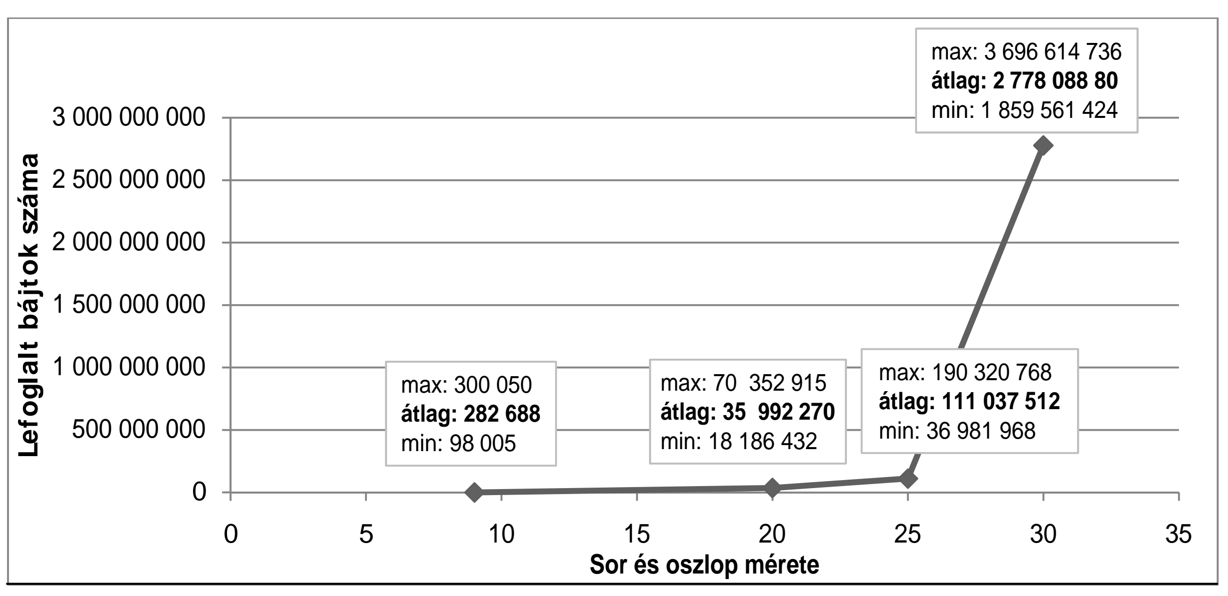


Futási idő(ms):	~4 900.00
Lefoglalt bájtok száma:	1 859 561 424
Használt Tábla objektumok száma:	2

23. ábra, még egy 30×30-as rejtvény



24. ábra, a futási idő és a rejtvény méretének kapcsolata



25. ábra, a lefoglalt bájtok száma és a rejtvény méretének kapcsolata

A program futtatásához használt gép adatai: egy 1GB-os DDRII-es memória, Dual Core 1.6GHz-es processzor 1MB-os cache-el. A futási idő csak a `megold()` metódus működésére vonatkozik. A lefoglalt bájtok és a `Tábla` objektumok számának meghatározásához a NetBeans 6.7.1 Profile szolgáltatását használtam.

10. Továbbfejlesztési lehetőségek

Az előző fejezetben látott futási eredményekből látható, hogy a tábla méretével exponenciális arányban nő a memória használat és a futási idő is, ezért a továbbfejlesztés legfőbb feladata ezek csökkentése.

A programban a variációk és a `Tábla` objektumok használják a legtöbb memóriát ezért ezek számát kell valahogy csökkentenünk. A variációk előállítása determinisztikus folyamat, tehát ugyan ahhoz a sorhoz/oszlophoz mindig ugyan annyi variációt fog előállítani, és a sorrendjük is ugyan az lesz. Ha egy variációról kiderül, hogy felesleges, mert a **`szűrés()`** metódus hamis értéket adott rá, akkor legközelebb is hamis értéket fog rá adni, abban az esetben, ha a `Tábla` elemei csak az átfedések keresése következtében változtak. Ezért célszerű az oszlopokhoz és sorokhoz egy listát csatolni, mely a felesleges variációk indexét tartalmazza, hogy ne állítsuk elő még egyszer őket, ezzel memóriát és időt spórolva. Természetesen, ha új `Tábla` objektumot hozunk létre a heurisztika miatt, akkor ezeket a listákat is másolni kell.

A `Tábla` objektumokat nagy számban a backtrack kereső miatt használunk, ezért ezt az eljárást kell valahogy kiküszöbölni, viszont a DPLL-nél szükségünk van valamilyen heurisztikára, ami megőrzi a korábbi állapotokat. A DPLL helyett használhatunk más SAT-fejtő algoritmust is, például a WalkSat-ot, mely az összes mezőnek véletlen értékeket ad és ezekről próbálja belátni, hogy igazzá teszik-e a formulát. Mivel csak egy interpretáció teszi a formulát igazzá és több ezer nem, ezért tanácsos nem minden elemnek értéket adni, hanem az ismeretleneket az átfedések megkeresésének módszerével feltárni.

A használt memória méretét úgy is csökkenthetjük, hogy egy `Tábla` eleméhez nem 32 bitet használunk, hanem csak 2-t, mivel ennyire van szükség három különböző érték jelölésére, így egy `int` típusú változóban 16 elemet lehet letárolni, tehát a táblák és a variációk mérete az 1/16-ra csökken.

Összefoglalás

Jelen szakdolgozat célja gyakorlati példa bemutatása az automatikus tételbizonyítás használatára.

A szakdolgozat elején a Nonogram rejtvény ismertetése történt meg, ahol egy a megoldás során használt eljárás is bemutatásra került. Ezután fejlesztéshez elengedhetetlen logikai alapfogalmak tisztázása következett, majd az automatikus tételbizonyítás egyik módszere, és egy megoldást kereső algoritmus.

A dolgozat második felében a megvalósítás van előtérben. Bemutatásra került a megfejtést kereső algoritmus nagyvonalakban, ezután az egyes lépések részletes tárgyalása következett, ahol a részlépésekhez tartozó algoritmusok, és ezek együttműködése is szerepelt. Az algoritmusok az elméleti alapokat használják, de nem valósítanak meg minden pontot, mert a gyakorlatban sok dolog felesleges lenne köszönhetően a megszorításainknak és a feladat adottságainak.

Melléklet

A következő kódrészek a Nonogram osztályból származnak és Java nyelven íródtak:

1. **melléklet:**

```
private void átfedés(int[] variáció){
    if(átfedések==null){
        átfedések=variáció.clone();
    }else{
        int[] új=feltölt(ISMERETLEN,variáció.length);
        for(int i=0;i<átfedések.length;i++){
            if(átfedések[i]==variáció[i]){
                új[i]=variáció[i];
            }
        }
        átfedések=új.clone();
    }
}
```

2. **melléklet:**

```
private boolean szűrés(int[] szűrő, int[] mit){
    for(int i=0;i<szűrő.length;i++){
        if((szűrő[i]==FOGLALT && mit[i]==ÜRES) ||
            (szűrő[i]==ÜRES && mit[i]!=ÜRES)){
            return false;
        }
    }
    return true;
}
```

3. melléklet:

```
private void összesVariáció(boolean mit, int hanyadik,Tábla tábla)
            throws InsatiableFormulaException{
        if(mit==OSZLOP && oszlop.get(hanyadik).size()==1 &&      //1.
                oszlop.get(hanyadik).get(0)==0){
            tábla.setOszlop(hanyadik, feltölt(ÜRES,sor.size()));
            return;
        }
        if(mit==SOR && sor.get(hanyadik).size()==1 &&
                sor.get(hanyadik).get(0)==0){
            tábla.setSor(hanyadik, feltölt(ÜRES,oszlop.size()));
            return;
        }
        átfedések=null;
        int[] Bsor=feltölt(ÜRES, mit==OSZLOP ? sor.size() :
            oszlop.size());
        variáció(mit, hanyadik, Bsor, 0, 0, tábla);
        if(átfedések==null){                                      //2.
            throw new InsatiableFormulaException();
        }

        if (mit == OSZLOP) {                                      //3.
            tábla.setOszlop(hanyadik, átfedések);
        } else {
            tábla.setSor(hanyadik, átfedések);
        }
        return;

    }
```

//1. azt az esetet vizsgáljuk amikor nincs blokk az adott sorban

//2. kivétellel jelezzük, hogy ellentmondást fedeztünk fel

//3. a táblához hozzá adjuk az újonnan felderített elemeket

4. melléklet:

```
    private void variáció(boolean mit, int index, int[] vektor, int
honnan, int hanyadik_blokk, Tábla tábla){
        List<List<Integer>> temp1;
        int[] temp2;
        if(mit==SOR){                                           //1.
            temp1=sor;
            temp2=tábla.getSor(index);
        }else{
            temp1=oszlop;
            temp2=tábla.getOszlop(index);
        }
        if(hanyadik_blokk==temp1.get(index).size()){            //2.
            if(szűrés(temp2,vektor)){
                átfedés(vektor);
            }
        }else{
            int hátralévő=0;
            for(int i=hanyadik_blokk+1;                         //3.
                i<temp1.get(index).size();i++){
                hátralévő+=temp1.get(index).get(i)+1;
            }
            hátralévő=vektor.length-hátralévő+1;
            for(int i=honnan;i<hátralévő-                       //4.
                    temp1.get(index).get(hanyadik_blokk);i++){
                int[] temp_vektor=new int[vektor.length];
                for(int j=0;j<vektor.length;j++){
                    temp_vektor[j]=vektor[j];
                }
                for(int j=i;
                 j<i+temp1.get(index).get(hanyadik_blokk);j++){ //5.
                    temp_vektor[j]=hanyadik_blokk+1;
                }
                variáció(mit,index,temp_vektor,
                        i+temp1.get(index).get(hanyadik_blokk)+1,
                        hanyadik_blokk+1,tábla);
            }
        }}
```

```
//1. mit alapján lekérdezzük a megfelelő elemeket a táblából
//2. ha nincs több blokk, akkor ellenőrizzük az átfedéseket
//3. az utolsó lefoglalható index meghatározása
//4. a variációk előállítása
//5. a blokkváltozatok elhelyezése a variációkban
```

5. melléklet:

```
private Tábla dpll(Tábla régi_tábla){
        Tábla tábla=régi_tábla.másolat();
        Tábla temp;
        int kész;
        for(;;){
            kész=tábla.kész();
            try{                                                    //1.
                kitölt(tábla);
            }catch(InsatiableFormulaException e){
                return régi_tábla;
            }
            if(kész==tábla.kész()){
                break;
            }
        }
        if(kész<sor.size()*oszlop.size()){                          //2.
            java.awt.Point p=tábla.getElsőIsmeretlen();
            tábla.setElem(p, ÜRES);
            temp=dpll(tábla);
            if(temp.kész()==sor.size()*oszlop.size()){
                return temp;
            }else{
                tábla.setElem(p, FOGLALT);
                return dpll(tábla);
            }
        }
        return tábla;
    }
//1. új elemek felderítése az átfedések meghatározásával
//2. ha még nem vagyunk kész, akkor megváltoztatunk egy elemet
```

Irodalomjegyzék

[1] Futó Iván, Mesterséges Intelligencia, Aula kiadó Kft., Budapest, 1999

[2] Dragalin Albert, Buzási Szvetlána: Bevezetés a matematikai logikába, Kossuth Egyetemi Kiadó, Debrecen, 1986.

[3] Pásztorné Varga Katalin, Várterész Magda: A matematikai logika alkalmazás-szemléletű tárgyalása, Panem Kiadó, Budapest, 2003.

[4] Szendrei Ágnes: Diszkrét matematika, Polygon Kiadó, Szeged, 1994.

[5] Papadimitriou, Christos H.: *Számítási bonyolultság (Computational complexity).* Egyetemi tankönyv. Novadat Bt., 1999.

[6] Vég Csaba, Instant Java/Java EE/Soa, Logos 2000, 2007

[7] Dr. Várterész Magda: Mesterséges Intelligencia 1 előadás 2006/2007 tanév, http://www.inf.unideb.hu/~varteres/mi1folia/foliafo.pdf

Köszönetnyílvánítás

Szeretném megköszönni témavezetőm, Aszalós László munkáját aki tanácsokkal látott el és minden felmerülő kérdésemre választ adott.