

# Logikai feladványok megoldása korlátprogramozással

Papp Ádám, Sós Nikolett  
Mérnök Informatikus BSc, I. évfolyam

Témavezető: Ősz Olivér, doktorandusz  
Széchenyi István Egyetem, GIVK, Informatika Tanszék

2018. április 1.



# Kivonat

A vizsgált feladatok az „Einstein fejtörője” néven ismert logikai feladvány, és annak különböző változatai. Ezekben a feladatokban adottak bizonyos személyek vagy objektumok, ezeknek néhány tulajdonságuk, melyek adott értékeket vehetnek fel. A feladatok hasonlítanak a szakirodalomban hozzárendelési feladatként ismert feladatosztályhoz, azzal a különbséggel, hogy nem az optimális hozzárendelést keressük egy adott szempont szerint, hanem speciális korlátozások vannak megadva a hozzárendelésre, és olyan megoldást keresünk, ami ezeket a feltételeket kielégíti. Míg a hozzárendelési feladatok megoldására a szakirodalomban léteznek hatékony algoritmusok, a logikai feladványok, különösen a korlátozások leírása valamilyen általánosabb modellezési módszert igényel.

Az általunk választott módszer a korlátprogramozás (constraint programming), ami egy modellezési és egy megoldási módszertan is egyben. Az utóbbi néhány évben egyre elterjedtebbé vált a korlátprogramozás használata különböző optimalizálási és kielégíthetőségi feladatok megoldásában.

Munkánk során megvizsgáltuk a logikai feladványok szerkezetét, összegyűjtöttük a korlátozások fajtáit. Különböző módokon modelleztük a korlátozásokat, és összehasonlítottuk őket megoldási teljesítmény szempontjából. Többféle megoldó szoftver teljesítményét is összevetettük egymással, és azonosítottuk, hogy az egyes megoldók hatékonyságát hogyan befolyásolta a használt modellezési módszer.

**Kulcsszavak:** logikai fejtörők, korlátprogramozás, hozzárendelési feladat



# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>7</b>
1.1. Korlátprogramozás bemutatása . . . . .	7
1.2. Logikai feladványok bemutatása . . . . .	9
<b>2. Feladatok modellezése korlátprogramozással</b>	<b>11</b>
2.1. Gardens feladatmegoldás . . . . .	11
2.1.1. Bináris mátrix . . . . .	12
2.1.2. Where helyett implikáció . . . . .	13
2.1.3. Tulaj hozzárendelési mátrix . . . . .	14
2.1.4. Termény halmazok . . . . .	14
2.1.5. Integer mátrix . . . . .	15
2.1.6. Kimenet formázása . . . . .	15
2.2. Zebra feladatok . . . . .	16
2.2.1. Movies példa bináris mátrixszal . . . . .	16
2.2.2. Fundraising Dinner példa változó tömbökkel . . . . .	19
2.3. Teszteredmények . . . . .	22
<b>3. Redundáns megkötések megkeresése</b>	<b>25</b>
<b>4. Összefoglalás</b>	<b>29</b>
<b>Irodalomjegyzék</b>	<b>29</b>



# 1. fejezet

## Bevezetés

Dolgozatunkban az „Einstein-féle” fejtörőkként ismertté vált logikai feladványok megoldásával foglalkozunk. A feladványok modellezéséhez használt korlátprogramozási módszert és a MiniZinc nyelvet az 1.1. alfejezetben mutatjuk be. A vizsgált feladatok bemutatása az 1.2. alfejezetben olvasható. A 2. fejezetben példákön szemléltetjük a feladatok néhány lehetséges modellezését és ismertetjük az összehasonlító tesztek eredményeit. A 3. fejezetben pedig a redundáns megkötések kiszűrésére vonatkozó eredmények találhatók.

### 1.1. Korlátprogramozás bemutatása

A feladatunkat úgynevezett korlátprogramozással oldottuk meg. Ennek a módszernek a legkorábbi verziója – a Sketchpad – 1963-ra tehető és Ivan Sutherland nevéhez fűződik[7]. Az 1980-as évektől egyre keresettebb lett, és mivel a logikai programozást szerették volna kiterjeszteni, így sok helyen korlát-logikai programozásként hivatkoznak rá. Az első praktikus verziókat – amiket üzleti célokra alkottak és már eladásra is bocsájtottak – az 1990-es években készítették.

A projektünkben a MiniZinc nevű grafikus szerkesztő programot használtuk – magát a nyelvet is így hívják –, aminek a fordítója az mzn2fzn, amely FlatZinc-re fordítja a MiniZinc modellt. Ezt a legtöbb megoldó által támogatott formátumú fájlt adja tovább a megoldónak, ami végül kiadja a megoldást. Az egyik legnépszerűbb megoldó a Gecode, melynek fő alkotója Christian Schulte[6]. A munkát 2002-ben kezdték meg, 2005 decemberében adták ki az első verziót, és onnantól kezdve több évben is aranyérmes lett az Association for Constraint Programming által rendezett versenyeken. A Gecode-on kívül más megoldókkal is képes együttműködni a MiniZinc, ilyen például a Gurobi, a Chuffed és a CBC. A megoldók általános működését legkönnyebben egy feladaton keresztül lehet szemléltetni. A feladványt a szakirodalomban „négy szín tételként” szokták említeni. Adottak bizonyos országok és ezeket úgy kell kiszínezni adott számú színnel, hogy a szomszédos területek ne legyenek azonosak. A modell elején megadjuk a használt színek számát és külön ki van emelve, hogy a megoldás során is ezeket vegyék fel az egységek. A szomszédossági mátrixot – ami egyben a feltételek listája is itt esetünkben – kikötések van megadva. A legprimitívebb mód erre az,

hogy a szomszédok meg vannak adva páronként, amiknek más és más értéket kell felvenniük, azaz két egymás mellett lévő rész nem lehet egyenlő tulajdonságú.



MiniZinc modell

```
%adatok
int: maxSzin = 4;

%megadjuk milyen értékeket vehetnek fel
var 1..maxSzin: Belgium;
var 1..maxSzin: Dánia;
var 1..maxSzin: Franciaország;
var 1..maxSzin: Németország;
var 1..maxSzin: Hollandia;
var 1..maxSzin: Luxemburg;

%kikötések
constraint Belgium != Franciaország;
constraint Belgium != Németország;
constraint Belgium != Hollandia;
constraint Belgium != Luxemburg;
constraint Dánia != Németország;
constraint Franciaország != Németország;
constraint Franciaország != Luxemburg;
constraint Németország != Hollandia;
constraint Németország != Luxemburg;

solve satisfy;
```

1.1. ábra.

A megoldó úgy dolgozik, hogy kiválaszt egy változót – a példában egy országot – és beállítja egy lehetséges értékre – színre. Ezután a korlátozások alapján következtetve csökkenti a többi változó lehetséges értékkészletét. Ezt a műveletet propagációnak nevezzük. A példában ez azt jelenti, hogy a szomszédos területeknél kizárja azt a lehetőséget, amit már felhasznált, így már csak a megmaradt színekből választhat. Ha a propagáció során egy változó lehetséges értékkészlete üressé válik, akkor a megoldó visszavonja a legutolsó értékadást, és az adott értéket kizárja a változó értékkészletéből, mert ellentmondáshoz vezet. Ezután egy másik értéket ad neki, vagy egy új változót választ ki. Amikor minden változónak sikerült értéket adni, akkor az egy lehetséges megoldása a feladatnak. Természetesen



nem csak egy megoldás létezhet, ha például a piros és a fekete szín fel van cserélve egymással, akkor az már másik megoldásnak fog számítani. Ha több megoldásra vagyunk kíváncsiak, folytathatjuk a keresést az utolsó értékadás visszavonásával. A korlátprogramozásnak ez egy hatalmas előnye más módszerekhez képest, hogy az összes lehetséges megoldást kiadja nekünk, nem pedig csak egyet. A módszer optimalizálásra is használható, ahol a megoldások értékét egy célfüggvény adja meg. A keresés során egy további korlátozást kell figyelembe venni: hogy a megoldás értéke az eddig megtalált megoldásoknál jobb legyen.

## 1.2. Logikai feladványok bemutatása

A címben lévő logika szóról az első érdemleges információnk Arisztotelésztől származik. Fontos megjegyezni, hogy nem tartotta külön tudománynak, csupán egy eszközként tekintett rá más szakirányokhoz. Szerinte ennek az elsajátítása ugyanannyira nem „tudatos”, mint az anyanyelv megtanulása.[5] A összes vizsgált feladatunk a logikára épül, ezért is kapták a „logikai feladvány” nevet. Mi részletesebben az „Einstein-féle” esetekkel[3] dolgoztunk. Ezekben a feladatokban adottak személyek és hozzájuk több személyes tulajdonság vagy tárgy, melyek mindegyike csak egy bizonyos emberhez tartozik, de hogy melyik kihez, annak a meghatározása maga a feladat. A feladvány megad néhány állítást a személyekről és tulajdonságaikról, melyekből levezethető a helyes hozzárendelés. A leghíresebb feladata a következő. Adva van öt különböző színű ház egymás mellett, melyekben más és más nemzetiségű lakó él. Mind az öt ház tulajdonosa egy bizonyos italt iszik, egy bizonyos márkájú cigarettát vesz, és egy adott háziállata van. Mindegyik tulajdonos más háziállatot tart, más márkájú cigarettát szív, és más italt fogyaszt. A mi dolgunk, hogy kitaláljuk a kikötések alapján, hogy mi a helyes megoldás.

A hozzárendelési feladatok nagyon hasonlítanak az általunk vizsgált feladatokhoz, de sok dologban el is térnek tőlük. Az egyik talán legszembetűnőbb különbség, hogy ezek a modellek optimalizálásra lettek kitalálva, ahol valamilyen célfüggvény segítségével a legjobb megoldást keressük a sok lehetőség közül. A kikötések itt egyáltalán nem szerepelnek, minden hozzárendelés egy lehetőség, maximum csak azt szabják meg, hogy nem lehet mindent mindenhez hozzárendelni. A legismertebb ilyen modell a magyar módszer[4]. Ilyen feladatra példa a gép-hozzárendelési feladat. Adott meghatározott számú gép és ugyanannyi független munka. Bármelyik gép bármelyik munkát képes elvégezni. Ismertek a gépek adott munkákra vonatkozó költségei. A feladat az, hogy minden géphez rendeljünk pontosan egy munkát úgy, hogy az összköltség minimális legyen.

Einstein feladatát egyes források szerint az emberiség csupán 2 százaléka képes megoldani, ami nem tudományosan alátámasztott adat, de azt mindenképpen kifejezi, hogy bizony jó logikára van szükség hozzá. Ez is bizonyítja, hogy egyes feladatoknál ahhoz, hogy megkapjuk az eredményt sok időt és energiát kell befektetni a munkába. Éppen ezért vizsgáltuk azt, hogy hogyan lehet az ilyen feladványok megoldását számítógépes segítséggel meghatározni.























## 2. fejezet

# Feladatok modellezése korlátprogramozással

### 2.1. Gardens feladatmegoldás

A munkánk során részletesen bizonyos előre „legyártott” feladatokat vizsgáltunk. Az első ilyen neve „Gardens”, azaz „Kertek”, amit az Einstein-féle logikai feladványokhoz sorolnak[1]. Adott 5 barát, akiknek a kertjei egymás mellett helyezkednek el. Ezeken a területeken 12-féle növényből termesztenek 4-et fejenként. Azt is tudjuk, hogy ezekből a terményekből 4 gyümölcs, 4 zöldség és 4 virág van. Azt, hogy ki melyik kertben dolgozik és azt, hogy mit tartalmaznak ezek, azt kikötések sora után tudjuk csak meg, aminek a végeredményét az alábbi kép szemlélteti.

Hank	Sam	Paul	Zick	Luke
				
				
				
				

2.1. ábra.

2.1. táblázat.

	Hank	Sam	Paul	Zick	Luke
alma	1	0	0	0	0
körte	1	0	0	0	1
mogyoró	0	0	0	0	1
cseresznye	1	1	0	0	0
sárgarépa	0	0	1	0	0
petrezselyem	0	0	0	0	1
tök	0	0	1	0	1
hagyma	0	1	1	0	0
őszirózsa	0	0	0	1	0
rózsa	1	1	1	1	0
tulipán	0	1	0	1	0
lilium	0	0	0	1	0

### 2.1.1. Bináris mátrix

Először is az általunk használt parancsokat mutatnám be a kikötések fajtái alapján a megoldásunk első verzióján. Itt bináris mátrix segítségével dolgoztunk, azaz, ha megtalálható valami a kertben, akkor a helyére 1-est, ha nem akkor 0-át raktunk a táblázatba. A tulajok egy külön tömbben vannak tárolva, melyeket összekötöttünk a kertekkel.

A legfontosabb, hogy minden utasítást „constraint”-el kell kezdeni. A szó maga is nagyon jól szemlélteti, hogy ami mögé van írva, azt úgy mond „kikényszerítjük”, hogy tartsa be a munka során. Első dolgunk a változók deklarálása, méretének megadása és a tömbök létrehozása volt.

Első kettőt az „int” és „set of int”, utóbbit az „array” kulcsszóval tettük meg, ami után megadtuk a tömb méretét, és azt, hogy milyen értékeket vegyenek fel az elemei.

```
int: kertek;

set of int: Kertek = 1..kertek;

array[Termenyek, Kertek] of var {0, 1}: termeszt;
```

2.2. ábra.

A képeken is jól látszik, hogy Zick például csak virágokat termel, míg Hank-nél vagy Sam-nél mind három fajta termés megtalálható, tehát nagyon különbözőek a kertek.

Direkt ezekre az esetekre áll rendelkezésünkre az „alldifferent”, ami kifejezi, hogy mint a kertek, mint a tulajok csak és kizárólag egyszer szerepelnek és eltérnek egymástól.

A leírásban sok olyan kikötés szerepel, ami kijelenti, hogy mi hányszor szerepel. Ennek három speciális esetét különböztettük meg. Az első mikor konkrétan megadják, hogy melyik

```
constraint alldifferent(tulaj);
```

2.3. ábra.

kertben vagy melyik tulaj termeszt/nem termeszt az adott dolgot, de eme módon akár a tulajt is megadhatják. Ilyenkor egyszerűen csak megadjuk az adott cella értékét.

$$termeszt_{12,3} = 0 \quad (1)$$

A második mikor csak annyit tudunk, hogy az adott növényt hányszor termeljük. Ilyen esetekben azt az előnyt használtuk ki, hogy mivel a táblázatunkban számok vannak, így lehet sima összeadás műveletet használni, így ilyenkor a sorösszegeket adtuk meg neki.

$$\sum_{k \in Kertek} termeszt_{1,k} = 1 \quad (2)$$

A harmadik pedig nagyon hasonlít a másodikhoz, de a szövegben valamilyen formában még jobban hangsúlyozza az állítást. Ilyen esetekre van az „exactly” a kulcsszó a gyűjteményben, aminek konkrét felépítési kritériuma van.

```
constraint exactly(1, [bool2int(sum(z in Zoldsegek)
(termeszt[z, k])=4 \ / sum(gy in Gyumolcsok) (termeszt[gy, k])=4
\ / sum(v in Viragok) (termeszt [v,k])=4) | k in Kertek], 1);
```

2.4. ábra.

Végül, de nem utolsó sorban van még egy kikötési forma amit használtunk, de ez nem volt elég hatékony megoldás, melyről a következő alfejezetben lesz szó.

$$\sum_{z \in Zoldsegek} termeszt_{z,t} = 3 \quad \forall t \in Tulaj : tulaj_t = 4 \quad (3)$$

### 2.1.2. Where helyett implikáció

Ha a „forall”-on belül olyan „where” feltétellel szűrünk, ami a változók értéke alapján szűri a korlátozást, akkor amíg nincs meghatározva ezen változók értéke, addig ezek a korlátozások nem vesznek részt a propagációban sem. Az előbbi kikötést, miszerint csak a Paulhoz tartozó kertre kell teljesülnie, hogy a zöldségek száma 3, implikációra cseréltük.

$$tulaj_t = 4 \rightarrow \sum_{z \in Zoldsegek} J_{z,t} = 3 \quad \forall t \in Tulaj \quad (4)$$

2.2. táblázat.

	Luke	Sam	Hank	Paul	Zick
Kert1	0	0	1	0	0
Kert2	0	1	0	0	0
Kert3	0	0	0	1	0
Kert4	0	0	0	0	1
Kert5	1	0	0	0	0

Ez a leírás javította a megoldó hatékonyságát, ahogy az a 3. fejezetben bemutatott futási eredményekből is látszik, így a további modellekben már csak ezt a fajta megfogalmazást használtuk.

### 2.1.3. Tulaj hozzárendelési mátrix

A harmadik verzióban már a meglévő termény mátrix mellé felvettünk még egyet a tulaj-kert kapcsolatokhoz. Az új mátrix szintén egyesekkel és nullákkal lett feltöltve.

A kikötések felépítésében itt nincs változás, tehát nem sok különbség van a két változat között.

### 2.1.4. Termény halmazok

A következő ötletünk az volt, hogy a kétdimenziós tömb helyett halmazokkal reprezentáljuk a megoldást.

```
array[Kertek] of var set of Termenyek: termeszt;
```

2.5. ábra.

A tulajokat csak egy egyszerű tömbben tároltuk itt el, mint az első két változatban.

Két fontos új kifejezés is megjelent a kikötésekben ennél a munkánál. Az első a „card” nevű függvény, ami megadja a részhalmaz elemszámát.

$$\|Termeszt_k\| = 3 \quad \forall k \in Kertek \quad (5)$$

Ehhez a „card”-hoz használtunk még helyenként az „intersect” függvényt is, ami két halmaz metszetét állítja elő.

$$I_t = 4 \rightarrow \|Zoldsegek \cap termeszt_t\| = 3 \quad \forall t \in Tulaj \quad (6)$$

A másik újdonság az "array union", ami a tömbben tárolt halmazok unióját képezi.

$$\| \$termeszt = \$termenyek \| \quad (7)$$

### 2.1.5. Integer mátrix

A Kertek utolsó verziójában ismét visszatértünk a mátrixos megoldáshoz, ám itt már nem bináris értékekkel töltöttük fel, hanem konkrétan a terményekkel. Az első dimenzió maradt a kertek halmaza, de a második dimenzió már nem az összes terményen megy végig, hanem a 4 termesztett növényen, és nem bináris értékekkel történik a hozzárendelés, hanem 4 növénytípus „enum”-ként való megadásával. Egy másik fontos változtatás volt, hogy itt mivel nem tudtunk sorösszegeket számolni, így a „count”-ot kellett használnunk.

```
constraint forall(k in Kertek)(tulaj[k]=Hank ->  
count([termeszt[k, n] | n in Noveny], oszirozsa)=0);
```

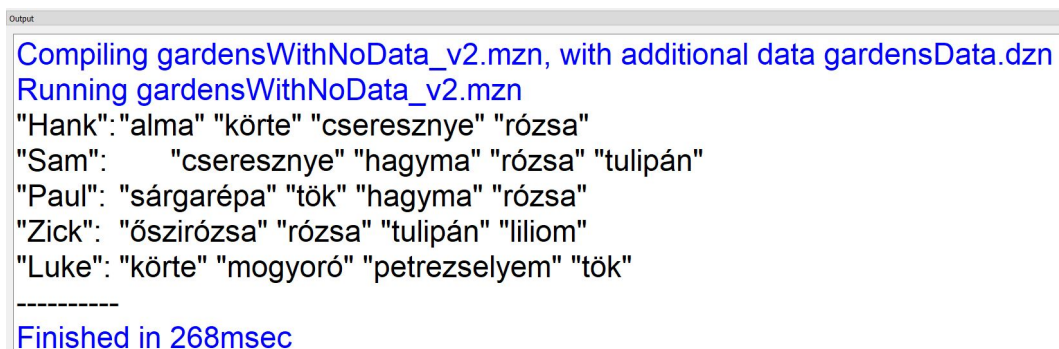
2.6. ábra.

Egy plusz kikötést kellett még írunk ehhez a változathoz, az egy megoldás megtartása érdekében. Le kellett fixálni a termények sorrendjét, hogy ne adjon ugyanazt a megoldást különböző sorrendekkel a megoldó, így megkötöttük, hogy a 4 növény sorszám alapján növekvő sorrendben szerepeljen. Ennek a módszernek pontos szemléltetése a 15.ábrán látható.

$$termeszt_{k,n} < tulaj_{k,n+1} \quad \forall k \in Kertek, n \in 1, \dots, Noveny \quad (8)$$

### 2.1.6. Kimenet formázása

A kimeneteket egységesen minden verziónál formáztuk a jobb olvashatóság érdekében.



```
Output  
Compiling gardensWithNoData_v2.mzn, with additional data gardensData.dzn  
Running gardensWithNoData_v2.mzn  
"Hank": "alma" "körte" "cseresznye" "rózsa"  
"Sam":   "cseresznye" "hagyma" "rózsa" "tulipán"  
"Paul":  "sárgarépa" "tök" "hagyma" "rózsa"  
"Zick":  "ószirózsa" "rózsa" "tulipán" "liliom"  
"Luke":  "körte" "mogyoró" "petrezselyem" "tök"  
-----  
Finished in 268msec
```

2.7. ábra.

Ötből három változatban az adatokat külső fájlban vannak tárolva, így például, ha csak a növények neveit szeretnénk módosítani, nem kell az egész programot átírni hanem csak az adatfájlt. Ez a módszer is egy jó megoldás, ha munkát szeretnénk spórolni később.

## 2.2. Zebra feladatok

A bevezetőben bemutatott Einstein-féle úgynevezett Zebra Puzzle-típusú fejtörők[2] modellezése és megoldása tette ki kutatási tevékenységünk nagy részét. A cél az volt, a minél többféle adatszerkezettel és esetleg egy adatszerkezethez is minél többféle módon leírható korlátozások készítése. A könnyebb kezelhetőség illetve az átláthatóság érdekében a korlátozások felépítésük alapján kategorizálásra kerültek, így az ugyan más tartalmú, de hasonló logikai felépítésű kifejezések megírásakor jelentős mennyiségű időt lehetett megtakarítani. A következőkben részletesen bemutatásra kerülnek az ezen példák megoldására készített modellek. A két példa, amelyekkel részletesen foglalkoztunk a „Movies Night” névre hallgató „könnyű” nehézségi szintű és a „Fundraising Dinner” nevű „nagyon nehéz” nehézségű voltak. Továbbá mindegyikhez két különböző adatszerkezeti megvalósítás, illetve ezeken belül további kétféle korlátozás-leírási mód készült.

### 2.2.1. Movies példa bináris mátrixszal

Az első 4x5-ös nagyságú példa: adott 4 barát, akik egy este moziba mentek, ahol egymás mellé ültek le. Mindegyikük azon felül, hogy hányadik széken ül, 5 tulajdonsággal jellemezhető:

1. Milyen színű inget visel: fekete, kék, zöld, piros
2. Keresztnév: Daniel, Joshua, Nicholas, Ryan
3. Milyen típusú film a kedvence: akció, vígjáték, horror, thriller
4. Nassolnivaló: chips, süti, cracker, popcorn
5. Életkor: 11, 12, 13 és 14 éves

Ahhoz, hogy megtudjuk, melyik széken ki ült és milyen tulajdonságokkal rendelkezik, rendelkezésre áll 13 állítás, amelyeket a modellezés során korlátozásokként(constraint) kezeltünk. Az első verziójú modell esetében a 4 fiú adatainak tárolására tulajdonságtípusonként egy-egy bináris értékeket(0, 1) tartalmazó 2 dimenziós mátrix került deklarálásra(2.8. ábra).

A mátrixok sorai az egyes tulajdonságok konkrét értékeit az oszlopai pedig sorrendben a fiúk moziban elfoglalt helyét jelölik. A MiniZinc nyelv sajátosságait kihasználva, a sorokat egy-egy felsorol(enum) típusú tömb indexeli. Ennek a megoldás kiíratásánál illetve a korlátozások megfogalmazásánál lesz különös szerepe. Az oszlopok indexelésére pedig szintén a leíró nyelv sajátossága miatt egy 1..4 értékkészletű segédhalmazt(PEOPLE) használtunk, ahogy az a 2.8. ábrán is látható. Vegyünk egy példát: amennyiben a movie mátrix 3. sorának és 1. oszlopának metszetében az 1-es érték szerepel(movie[horror, 1]=1) azt jelenti, hogy az első helyen ülő fiúnak a horrorfilm a kedvence. A db változóra a teljeskörű paraméterezhetőség miatt van szükség, értéke jelen esetben a fiúk(és egyben kategóriánként a különböző tulajdonságok) számával egyenlő. A későbbiekben fogjuk sűrűn használni.



```

int: db = 4;
set of int: PEOPLE = 1..db;

enum NAMES = {Daniel, Joshua, Nicholas, Ryan};
enum MOVIES = {action, comedy, horror, thriller};
[...]
array[NAMES, PEOPLE] of var {0, 1}: name;
array[MOVIES, PEOPLE] of var {0, 1}: movie;
[...]

```

2.8. ábra.

Belátható, hogy mivel egy tulajdonság kategóriából minden fiúhoz(egész pontosan ülőhelyhez, mivel még nem tudjuk, ki hol ül) pontosan egy értéket rendelhetünk hozzá, ezért az összes bináris mátrix sor- és oszlopösszegeinek egyenlőnek kell lenniük 1-gyel. Ezt hivatottak biztosítani az alábbi korlátok(9).

$$\begin{aligned}
\sum_{n \in N} name_{n,p} &= 1 & \forall p \in People \\
\sum_{p \in P} name_{n,p} &= 1 & \forall n \in Names
\end{aligned} \tag{9}$$

A forall biztosítja, hogy minden sorra(ill. oszlopra) teljesüljön az, hogy a sum függvény visszatérési értéke(amely az oszlopot ill. sort összegzi) 1. Ezek után következik a 13 korlátozás megfogalmazása. Ezen példa esetében 6 különböző típusú korlátozás volt azonosítható, amelyek esetében a kifejezések szerkezete változatlan csak az adatok(egész pontosan az indexelés) változik. Az I. típusnál konkrétan megmondják, hogy mely pozícióban milyen tulajdonságú ember ül(10).

$$age_{fourteen,3} = 1 \tag{10}$$

A II. típus hasonló az elsőhöz, de itt a széksor valamelyik széléről nyilatkozunk(11). Itt kihasználjuk, hogy bináris mátrixszal dolgozunk, ezért két elem összege 1 kell, hogy legyen, miszerint vagy az egyik szélen vagy a másikon ülhet Joshua.

$$name_{Joshua,1} + age_{Joshua,db} = 1 \tag{11}$$

A III. típusú korlátozás két tulajdonságot kapcsol össze egymással(12).

$$name_{Joshua,szek} = movie_{horror,szek} \quad \forall szek \in People \tag{12}$$

A IV. típus valamely tulajdonságú személy mellett közvetlenül balra(illetve jobbra) ülő

emberről állít valamit(13). Itt viszont ki kell kötnünk, hogy az első helyen(tehát a bal szélén) nem ülhet a thriller kedvelő, mivel az ő bal oldalán már nem ülhet senki. A korlátozás megfogalmazásánál ez esetben az implikációs operátort( $\rightarrow$ ) használtuk, amely kimondja, hogy ha megtaláltuk a thrillerkedvelőt, akkor fekete inges ül mellette(igaz állításból csak igaz következhet), egyéb esetben nem történik semmi.(hamis állításból bármi következhet). Ennek kiváltására használtunk egy továbbfejlesztett korlátozás-leírási mód esetén tisztán relációs operátorokat(14).

$$\begin{aligned} movie_{thriller,1} &= 0 \\ movie_{thriller,szek} = 1 &\Rightarrow shirt_{black,szek} = 1 \quad \forall szek \in 2..db \end{aligned} \quad (13)$$

$$\begin{aligned} movie_{thriller,1} &= 0 \\ movie_{thriller,szek} &= shirt_{black,szek} \quad \forall szek \in 2..db \end{aligned} \quad (14)$$

Az V. típusnál adott ember valamely tulajdonságú ember mellett balra (ill. jobbra) tetszőleges pozícióban foglal helyet (15). A második verzióban szintén relációs operátort (kisebb-egyenlő) használtunk az implikáció kiváltására (16).

$$(age_{eleven,x} = 1) \rightarrow \left( \sum_{szek \in 1..x-1} shirt_{black,szek} = 1 \right) \quad \forall x \in People \quad (15)$$

$$age_{eleven,x} \leq \sum_{szek \in 1..x-1} shirt_{black,szek} \quad \forall x \in People \quad (16)$$

A VI. típusnál egy ember valamely két másik között tetszőleges pozícióban foglal helyet(17). Ebben az esetben viszont a logikai operátor relációsra cseréléséhez két részre kellett bontanunk az állítást(18).

$$(age_{thirteen,x} = 1 \wedge movie_{action,y} = 1) \rightarrow \left( \sum_{szek \in x+1..y-1} shirt_{red,szek} = 1 \right) \quad \forall x, y \in People \quad (17)$$

$$\begin{aligned} age_{thirteen,x} &\leq \sum_{szek \in x+1..db} shirt_{red,szek} \quad \forall x \in People \\ movie_{action,x} &\leq \sum_{szek \in 1..x-1} shirt_{red,szek} \quad \forall x \in People \end{aligned} \quad (18)$$

A kimenet képzése a MiniZinc nyelvnek megfelelő szintaktikával valósult meg.

```
output [format(width, show(n)) ++ "\t" | t in PEOPLE, n in NAMES where
fix(name[n,t])=1]++["\n"];
```

2.9. ábra.

A format függvény pusztán esztétikai formázást, a show karakterlánccá konvertálást,

a fix függvény pedig a változók értékének kiíratás előtti „stabilizálását” végzi. A where záradékban pedig azt vizsgáljuk, hogy a bináris mátrixban hol található 1-es érték, mert csak azokat az értékeket írjuk ki.

### 2.2.2. Fundraising Dinner példa változó tömbökkel

Az eddigiekben részletezett bináris mátrix mellett egy másik, ettől merőben eltérő adatszerkezettel rendelkező implementációt is készítettünk. Ezt a második, 5x6-os méretű feladaton, a „Fundraising Dinner” címűn keresztül fogjuk bemutatni. Néhány szó magáról a feladatról: a leírás szerint 5 jó módú hölgy egy jótékonyági vacsorán vesz részt. Egy egyenes asztalnál, egymás mellett foglalnak helyet és mindegyikük 6 féle tulajdonsággal jellemezhető:

1. Keresztnév
2. Viselt ruha színe
3. Nyakláncukon található drágakő típusa
4. Életkoruk
5. Fogyasztott koktéljuk
6. Adakozott pénzmennyiség dollárban

Ennél az adatstruktúránál tulajdonságkategóriánként egy-egy felsorol(enum) típusú egydimenziós tömböt készítettünk, amelyet a már említett segédhalmazzal indexeltünk(30. ábra).

```
array[PEOPLE] of var COLORS: dress;  
array[PEOPLE] of var NAMES: name;  
[...]
```

2.10. ábra.

Itt a tárolt értékek már a konkrét tulajdonságok neveit vehetik fel, sorrendjük pedig meghatározza, hogy melyik széken ülő emberre mi jellemző. Egy tömbön belül az értékek egyediségét az alldifferent globális constraint biztosítja(31. ábra).

```
constraint alldifferent(dress);  
constraint alldifferent(name);  
[...]
```

2.11. ábra.

A korábban már részletezett korlátozástípusok jelen adatstruktúra esetében a következőképpen lettek modellezve. Az I. illetve II. típusú kikötés ebben a példában nem fordult ugyan elő, de modellezése hasonló módon történik, mint a „Movies” példa esetén. A III.(19) és IV. típus(20) esetén egyféle leírási mód választható csak.

$$(name_p = Jane) \leftrightarrow (donation_p = twentythousand) \quad \forall p \in People \quad (19)$$

A IV. típusú korlátozás leírásánál is az ekvivalencia operátor( $\leftrightarrow$ ) használata figyelhető meg. Azokat az eseteket, amelyeknél az említett személyek valamely szélén ülnek a széksornak, itt külön kell kezelni, mivel ekkor egyértelmű egymáshoz viszonyított helyzetük. Másrészt pedig így elkerülhető a tömbök bejárása során a kiindexelés.

$$\begin{aligned} cocktail_p = cosmopolitan &\leftrightarrow age_{p+1} = sixty \\ \forall p \in 1..db - 1 \wedge cocktail_{db} \neq cosmopolitan \wedge age_1 \neq sixty \end{aligned} \quad (20)$$

Az V. típus az implikáció operátorral pedig a 2.12. ábrán látható. B változat esetében pedig az indexekkel való bejárásnál fogalmazható meg egy apróbb szabály(21).

```
forall(x in PEOPLE)(cocktail[x]=margarita ->
    count([dress[p] | p in 1..x-1], blue)=1) /\
forall(x in PEOPLE)(dress[x]=blue -> count([cocktail[p] |
    p in x+1..db], margarita)=1);
```

2.12. ábra.

$$\neg(cocktail_a = margarita \wedge dress_b = blue) \quad \forall a, b \in People \wedge a < b \quad (21)$$

A VI. korlátozástípusnál szintén kétféle modell készült, az A változatnál logikai operátorral és a később részletesen taglalt visszafelé következtetés módszerével(2.13. ábra). B verzióánál viszont a count függvény segítségével történik a számlálás(2.14. ábra).

Ezen felül ennél a feladatnál azonosítható volt egy újfajta korlátozás, amely egy objektum valamely oldaláról nyilatkozik, viszont nem rögzíti, hogy melyikről. Ezeknél az úgy nevezett visszafelé következtetés is felhasználásra került. Az alábbi példát megvizsgálva észrevehető, hogy abból, hogy Lidia a Cosmopolitant ivó mellett ül, viszont következik az is hogy a Cosmopolitan fogyasztó mellett – üljön ő bárhol is - valamely oldalon kell helyet foglalnia Lidia-nak. Ezt az xor(kizáró vagy) operátor biztosítja. A széksor széleit a IV. típusnál

```
forall(x in PEOPLE)(donation[x]=fourtythousand -> count([dress[p]
    | p in 1..x-1], red)=1) /\
forall(x in PEOPLE)(dress[x]=red -> count([donation[p] | p in
    x+1..db], fourtythousand)=1) /\
forall(x in PEOPLE)(donation[x]=twentythousand -> count([dress[p]
    | p in x+1..db], red)=1) /\
forall(x in PEOPLE)(dress[x]=red -> count([donation[p] | p in
    1..x-1], twentythousand)=1);
```

2.13. ábra.

```
forall(x in PEOPLE)(donation[x]=fourtythousand ->
    count([dress[p] | p in 1..x-1], red)=1) /\
forall(x in PEOPLE)(dress[x]=red -> count([donation[p]
    | p in x+1..db], fourtythousand)=1) /\
forall(x in PEOPLE)(donation[x]=twentythousand ->
    count([dress[p] | p in x+1..db], red)=1) /\
forall(x in PEOPLE)(dress[x]=red -> count([donation[p]
    | p in 1..x-1], twentythousand)=1);
```

2.14. ábra.

részletezett okok miatt ismét külön kellett kezelni.

$$\begin{aligned}
& (name_p = Lidia) \rightarrow (cocktail_{p-1} = cosmopolitan \oplus cocktail_{p+1} = cosmopolitan) \\
& \quad \forall p \in 2..db - 1 \quad \wedge \\
& (cocktail_p = cosmopolitan) \rightarrow (name_{p-1} = Lidia \oplus name_{p+1} = Lidia) \quad \wedge \\
& (name_1 = Lidia \rightarrow cocktail_2 = cosmopolitan) \quad \wedge \\
& (name_{db} = Lidia \rightarrow cocktail_{db-1} = cosmopolitan) \quad \wedge \\
& (cocktail_1 = cosmopolitan \rightarrow name_2 = Lidia) \quad \wedge \\
& (cocktail_{db} = cosmopolitan \rightarrow name_{db-1} = Lidia) \quad \forall p \in 2..db - 1
\end{aligned} \tag{22}$$

Az eredmény kiíratásánál itt kihasználható az, hogy a tömb elemeit nem kell vizsgálni, hanem sorrendben, vizsgálat nélkül kiíratathatóak.

```
output [format(width,show(dress[p])) ++ "\t" | p in PEOPLE]++["\n"];
```

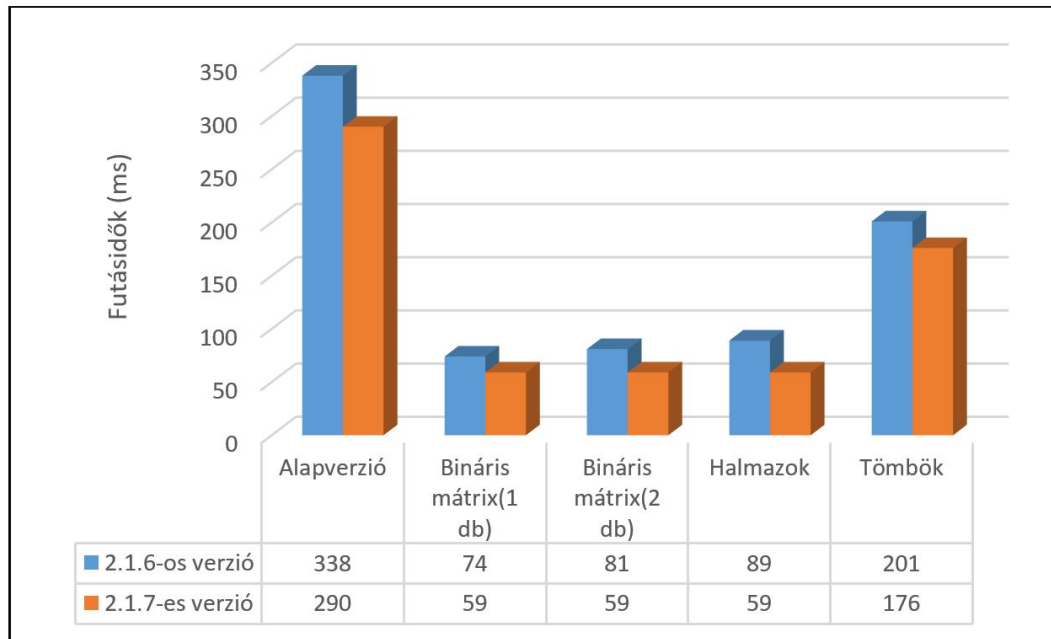
2.15. ábra.

## 2.3. Teszteredmények

A kiválasztott példák különféle módokon történő modellezésének végeztével, az így rendelkezésre álló modellek átfogó teszteknek lettek alávetve. Érdeklődésünk középpontjában az állt, hogy a különböző nehézségű példák, illetve az elérő módon modellezett korlátozások miként befolyásolják a megoldók hatékonyságát. A korábbi fejezetben már vázolt „Gardens” nevű példa elkészülte után, a tesztelés kezdeteként a MiniZinc IDE és a Gecode megoldó két verzióját is összevetettük egymással.

A 2.16. ábra tartalmazza a futási eredményeket milliszekundumban(ms) mérve. Megoldóként ez esetben a Gecode-ot használtuk. Az átlag mindegyik esetben 6 futtatás számtani közepe. Az adatokból több következtetés is levonható. Egyrészt, a 2.1.6-os verzió minden esetben valamivel lassabban oldotta meg a példákat, elmondható, hogy készítői láthatóan javítottak termékükön és optimalizálták a hatékonyságát. A második szembevetendő jelenség, miszerint az készített alapmodell/alapverzió, még meglehetősen nyersnek bizonyult megoldhatóság szempontjából is. A futási eredmények jó okot szolgáltatnak a forall függvény where záradékában döntési alapként felhasznált, értékkel még nem feltétlenül rendelkező változók használatának kerülésére.

Harmadrészt pedig jól látható, hogy az olyan speciális, nem „Einstein-típusú” logikai feladvány esetében, mint amilyen a „Gardens” is, nem szerencsés a tömböket használó adat-reprezentáció választása. Részben a lassú megoldáskeresés, részben pedig a bonyolult megvalósítás miatt sem. A bináris mátrixok illetve halmazok használata között jelentős futás-

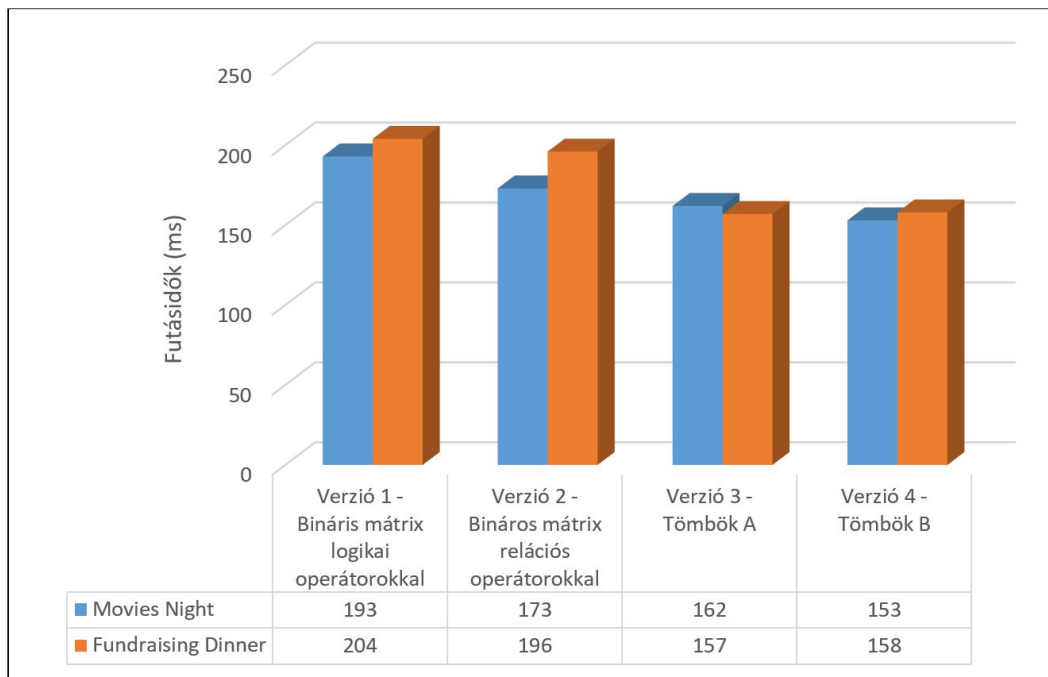


2.16. ábra.

időbeli eltérés nem volt. Főként annak tükrében, hogy az újabb verziónál teljesen azonos volt az átlagos futásidő mindkét modell típus esetében. Ám a régebbi verzió esetében is csak nagyon kis mértékben bizonyultak hatékonyabbnak a bináris mátrixos adatstruktúrák. A következőkben a Zebra vagy Einstein-típusú logikai fejtörők hatékonyságának összehasonlítása következik. Jelen esetben 2 példa állt rendelkezésünkre: A „Movies Night” mint 4x5-ös nagyságú könnyebb és a „Fundraising Dinner”, mint 5x6-os nehezebb feladat. Megoldóként továbbra is a Gecode-ot használtuk, és az átlagszámítási módszer is azonos volt az előzőekben ismertetetthez.

Amint az a 2.17. ábrán látható a „Fundraising Dinner” feladat esetében hosszabb futásidőket tapasztalhatóak, mint a „Movies Night” esetében, amely a példa összetettebb mivoltát tekintve nem meglepő. Az viszont már jóval szembetűnőbb, hogy az eltérés nagyon kicsi, ami jelzi, hogy a megoldó egy nagyobb és nehezebb, több hozzárendeléssel dolgozó feladat esetében is képes a futásidőket abszolút alacsony szinten tartani. Ez komoly érv mellett mikor eszközt választunk a hasonló típusú feladatok megoldásához. Más modellezési technikákhoz képest a korlátprogramozás esetében nem a feladat összetettségének mértékében nőnek a futásidők, hanem annál jelentősen lassabban, és ez a futásidő növekedés is csak a hozzárendelések számának jelentős növekedésekor mutatkozik meg igazán.

Emellett, a jóval gyakoribb Einstein alkotta Zebra Puzzle-ök modellezésekor sokkal célszerűbb egy tömböket használó adatstruktúrát megvalósítani. Mind a modellezés során a könnyebb kezelhetőség, mind pedig a jóval rövidebb futásidők szempontjából sokkal optimálisabb, ha ezt az adatrepresentációt részesítjük előnyben. Összegzésként, a végső konzekvenciákat levonva: a korlátprogramozás a kifejezetten rövid megoldási és futásidőket figyelembe véve, egy borzasztóan hatékony eszköz a vizsgált logikai fejtörők, mint speciális



2.17. ábra.

hozzárendelési feladatok modellezéséhez. A gyakori és sokak által ismert Einstein-féle feladatok esetében a felsorol típussal indexelt tömbök használatát célszerű választani. Viszont az olyan nem hétköznapi példánál – amilyen ez esetben a „Gardens” is volt – ahol nem beszélhetünk a klasszikus egy-az-egyhez hozzárendelésről már más a helyzet. Ez esetben javasolt valamilyen halmazokkal dolgozó adatszerkezet, esetleg bináris mátrixok deklarációja az maximális hatékonyság elérése érdekében.



## 3. fejezet

# Redundáns megkötések megkeresése

A példák korlátozásainak modellezése során felmerült annak lehetősége, hogy redundáns kifejezések fordulhatnak elő a feladatokban, azaz vannak olyan megkötések vagy megkötéshalmazok, amelyek elhagyása esetén továbbra is egyetlen, az eredeti feladatkiírásnak megfelelő megoldást talál a megoldó. Célunk ezek megkeresése, továbbá annak meghatározása volt, hogy elhagyásuk miképpen befolyásolja, vagy befolyásolja-e egyáltalán a különböző megoldók futási illetve megoldási idejét. Mivel a megoldók gyorsasága már a korábbiakban bebizonyosodott, ezért nagy esély mutatkozott arra, hogy egy vagy több korlátozás eltávolítása esetén is rövid idő alatt meg lehet határozni, hogy továbbra is egyértelmű-e az eredmény. A feladatok jellegéből adódóan egyértelműnek az az eset volt tekinthető, amikor a program lefutása után csak egy megoldást talál, amely az eredeti – minden kikötést tartalmazó – feladatkiírásnak is megfelel. Módszerünk a következő volt: minden kikötést egy elágazásban helyeztünk el (3.1. ábra).

```
constraint if kivesszuk[4]=1 then true else
    forall(p in PEOPLE)(necklace[p]=sapphire <->
        age[p]=fiftyfive)
endif;
```

3.1. ábra.

A „kivesszuk” egy, a korlátozások számával egyező elemszámú bináris értékeket tartalmazó tömb. Amennyiben a tömb adott indexű eleme 1 értéket tartalmaz, akkor a korlátozás a konkrét kifejezés értéke helyett egyszerűen igaz értéket vesz fel, így a kifejezést figyelmen kívül lesz hagyva. Amennyiben viszont az adott sorszámú elem 0 értékű, akkor a különben ágban elhelyezett, a feladatleírásnak megfelelően modellezett kifejezés lesz korlátozásként kezelve. Ezen elvet követve mindegyik korlátozás egy a fenti ábrához hasonló vezérlési szerkezetbe került beágyazásra. A „kivesszuk” tömböt pedig a könnyebb szerkeszthetőség érde-

kében nem a modellen belül lett definiálva, hanem értéke külön fájl(ok)ból lett beolvasva. Mivel egyes példák nagy mennyiségű kikötést tartalmaznak és ebből következően rengetegféle kombinációban lehetne ezeket elhagyni, így igyekeztük ezeket a tesztekét valamilyen szinten automatizálni. Az adatfájlok (a "kivesszük" tömb definícióját tartalmazó) egy C Sharp programmal voltak generálva oly módon, hogy első körben mindegyikben a tömbnek pontosan egy eleme lesz 1 értékű, így értelemszerűen a korlátozások darabszámával egyező adatfájl keletkezett. Ezeket a modellel együtt parancssorból futtattuk és feljegyeztük azokat a sorszámokat, amelyek esetén egyértelmű eredmény született. Második körben az imént említett C Sharp program legenerálta ezen összegyűjtött sorszámok összes létező, 2 tagból álló kombinációját, majd újabb parancssoros futtatás következett. Megint feljegyeztük az eredményeket és ezt ismételtük mindaddig, amíg a sorszámok értékkészlete le nem szűkült teljesen. A kapott eredmények meglepőek voltak. Még egy olyan kicsi és gyenge nehézségű, kevés kikötésből álló példánál is, mint amilyen a „Movies Night”, relatíve jelentős mennyiségű kikötés elhagyása esetén is teljesült, hogy az egyetlen kapott megoldás megegyezett a kiindulási feladat megoldásával. Ez esetben az összes 13-ból önmagában 3 korlátozás is redundánsnak bizonyult. Ezekből pedig 3 olyan különböző 2 tagból álló kombináció volt alkotható, amelyek egy időben is elhagyhatóak. Ezen számok a nagyobb példák esetében még tovább nőttek. A feldolgozott példák esetében a redundáns kikötések eloszlásait a 3.1. táblázat tartalmazza.

3.1. táblázat.

	Movies Night	Fundraising dinner	Gardens
1 tagú kombinációk	3	7	5
2 tagú kombinációk	3	15	5
3 tagú kombinációk	0	13	2
4 tagú kombinációk	0	4	0

Az adatokból kitűnik, hogy azon korlátozások közül, amelyek önmagukban elhagyhatóak, nem választható ki tetszőleges kombinációban 2 vagy több. Azaz már meg kell válogatni, hogy pontosan melyek együttes elhagyása ad egyértelmű eredményt. Tehát például ha két kifejezés külön-külön elhagyható, az korántsem jelenti azt, hogy ezek egyszerre is elhagyhatóak. Vegyünk egy konkrét példát: a „Fundraising dinner” esetében az elhagyható korlátozásokból(7) alkotható összes 2 tagú kombináció száma 21, de ezek közül csak 15 ilyen pár felelt meg a kitételeknek. Jellemzően minél több kifejezést szeretnénk egy időben feleslegessé minősíteni, annál inkább szűkül a választási spektrum, annál kevesebb kombináció jöhet számításba. Egy idő után elértük azt a küszöbértéket, amelynél több kifejezés nem hagyható el egyszerre. Ez jellemzően az összes kikötés számának 15-23%-a volt. A fentiek továbbá azt is jelentik, hogy a feladat megalkotói jelentős redundanciával dolgoztak a kikötések meghatározásánál. Ám ezt megállapítani a klasszikus módszerrel, miszerint egyszerűen kevesebb korlátozással kiírt példát akarunk megoldani papíron, sokszor egy átlagos képességű embernek nem lehetséges vagy túlságosan hosszú ideig tart. Esetünkben viszont egzakt matematikai-logikai modellekkel leírt korlátozásokat a számítógép oldotta meg, amely értelemszerűen az em-

berénél jóval nagyobb számítási kapacitással rendelkezik. Így a nem feltétlenül szükséges kifejezések felderítése viszonylag rövid idő alatt, egyszerű eszközökkel kivitelezhető volt. Ez felhasználható a feladványok készítésekor a minél nehezebb feladatok eléréséhez, melyekben minden információt fel kell használni a megoldás során. Futásidők tekintetében a módosíthatatlan, teljes feladathoz képest a kihagyásoknál jelentős változás sem a parancssoros sem a grafikus felületről történő futtatás során nem történt, bár egyes esetekben kis mértékben a futásidők megnőttek. A megoldók hatékonysága ismét bebizonyosodott, mivel kevesebb korlát, így a megoldáskeresés során kevesebb támpont esetén is komoly gyorsasággal adtak eredményt.



## 4. fejezet

# Összefoglalás

Munkánk során az „Einstein-féle” logikai feladványok szerkezetét és lehetséges megoldásukat vizsgáltuk. A feladatok általános modellezése végett megismerkedtünk a korlátprogramozás módszereivel. A feladatokat többféle módon modelleztük, és megvizsgáltuk az egyes modellek megoldási hatékonyságát. Ez után azt vizsgáltuk, hogy hogyan lehet kiszűrni a redundáns megkötéseket, amik elhagyásával még egyértelműen megoldható marad a feladat.



# Irodalomjegyzék

- [1]
- [2]
- [3] C. Browne. In computational intelligence in games. 2013. Deductive search for logic puzzles.
- [4] H. W. Kuhn. The hungarian method for the assignment problem. pages 83–97, 1955. Naval Research Logistics.
- [5] L. I. Perlovsky. The mind vs. logic: Aristotle and zadeh. pages 30–33, 2007. Critical Review.
- [6] Tack G. Schulte, C. and M. Z. Lagerkvist. Modeling and programming with gecode. 2010. Christian and Tack, Guido and Lagerkvist, Mikael.
- [7] Sutherland. Sketchpad a man-machine graphical communication system. 1964. Transactions of the Society for Computer Simulation.