Logikai feladványok megoldása korlátprogramozással

Szerzők: Papp Ádám, Sós Nikolett

Mérnök informatikus BSc., I. évfolyam

Témavezető: Ősz Olivér, doktorandusz

Széchenyi István Egyetem, GIVK, Informatika Tanszék

2018

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

[1. Bevezetés 1](#_Toc507753148)

[**1.1.** **Korlátprogramozás bemutatása** 1](#_Toc507753149)

[**1.2.** **Logikai feladványok bemutatása** 3](#_Toc507753150)

[2. Feladatok modellezése korlátprogramozással 5](#_Toc507753151)

[**2.1.** **Gardens feladatmegoldás** 5](#_Toc507753152)

[2.1.1. Az első verziónk 5](#_Toc507753153)

[2.1.2. Feladatunk második verziója 8](#_Toc507753154)

[2.1.3. Harmadik megoldás 8](#_Toc507753155)

[2.1.4. Negyedik változat 8](#_Toc507753156)

[2.1.5. Ötödik ötlet 9](#_Toc507753157)

[2.1.6. Verzióktól függetlenül 10](#_Toc507753158)

[**2.2.** **Zebra feladatok** 10](#_Toc507753159)

[**2.3.** **Teszteredmények** 18](#_Toc507753160)

[3. Redundáns megkötések megkeresése 21](#_Toc507753161)

[4. Összefoglalás 24](#_Toc507753162)

[Irodalomjegyzék 25](#_Toc507753163)

# 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 példákon szemléltetjük a feladatok néhány lehetséges modellezését és ismertetjük az összehasonlító tesztek eredményeit. A 3. fejezetben pedig a redundáns megkötések kiszűrésére vonatkozó eredmények találhatók.

## **Korlátprogramozás bemutatása**

A feladatunkat úgynevezett korlátprogramozással oldottuk meg. Ennek a módszernek a legkorábbi verziója – a Sketchpad – 1963-ra tehető és [Ivan Sutherland](https://en.wikipedia.org/wiki/Ivan_Sutherland) nevéhez fűződik[1]. 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[2]. 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 „négyszín 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 hány színt szeretnénk használni és külön megjegyezzük, hogy a megoldás során is ezeket vegyék fel az egységek. A szomszédsági mátrixot – ami egyben a feltételek listája is itt esetünkben – kikötésekben adjuk meg. A legprimitívebb mód erre az, hogy leírjuk páronként a szomszédokat, 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;**

**1. ábra:** Térkép színezése

**%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;**

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 felhasználtunk, 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 színt felcseréljük 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.

## **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.[3]

A összes vizsgált feladatunk a logikára épül, ezért is kapták a „logikai feladvány” nevet. Mi részletesebben az „Einstein-féle” esetekkel[4] 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 lehetőségből. 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[5].

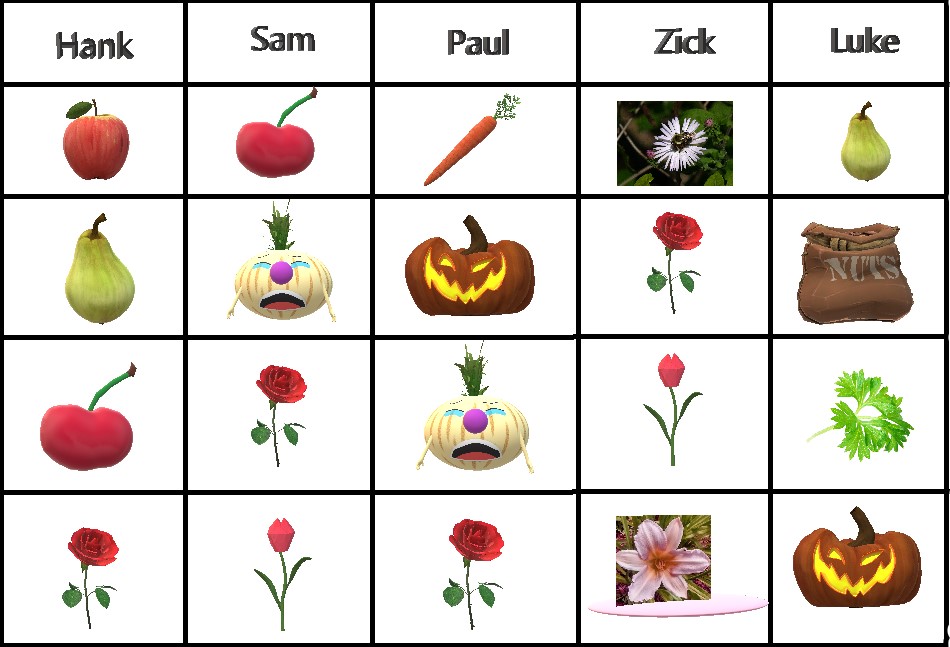
Ilyen feladatra példa a gép-hozzárendelési feladat. Adott meghatározott számú gép és ugyanannyi független munka. Bármelyik gép bármelyik munkát képes elvégezni. Ismertek a gépek adott munkákra vonatkozó költségei. A feladat az, hogy minden géphez rendeljünk pontosan egy munkát úgy, hogy az összköltség minimális legyen.

Einstein feladatát egyes források szerint az emberiség csupán 2%-a 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 belefektetni a munkába. Éppen ezért vizsgáltuk azt, hogy hogyan lehet az ilyen feladványok megoldását számítógépes segítséggel meghatározni.

# Feladatok modellezése korlátprogramozással

## **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[6].

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.

**2. ábra:** Gardens megoldás

### Az első verziónk

Előszőr 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 tulajokat egy külön tömbben tároljuk, melyeket összekötöttünk a kertekkel.

**1. táblázat:** Első verzió

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **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 |
| **liliom** | 0 | 0 | 0 | **1** | 0 |

A legfontosabb, hogy minden utasítást „constraint”-el kell kezdeni. A szó maga is nagyon jól szemlélteti, hogy amit mögé írunk, azt úgymond „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;

**3. ábra:** Adatok létrehozása

A képeken is jól látszik, hogy Zick például csak virágokat termel, míg Hank-nél vagy Sam-nél mind három fajta termény megtalálható, tehát nagyon különbözőek a kertek. Direkt ezekre az esetekre használtuk programunk során az „alldifferent”-et, ami kifejezi,

hogy mint a kertek, mint a tulajok csak és kizárólag egyszer szerepelnek és eltérnek egymástól.

**4. ábra:** Kikötés: egy embernek pontosan 1 kertje van(azaz minden tulaj különböző)

constraint alldifferent(tulaj);

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

**5. ábra:** Kikötés: Paul kertje a középső, liliom nélkül.

constraint termeszt[12,3]=0;

constraint tulaj[3]=4;

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.

constraint( sum(k in Kertek) (termeszt[1,k])=1 );

**6. ábra:** Kikötés: Az alma csak egy kertben szerepel.

A harmadik pedig nagyon hasonlít a másodikhoz, de a szövegben valamilyen formában még jobban hangsúlyozza az állítást. Ilyen esetekben az „exactly”-t használtuk, aminek konkrét felépítési kritériuma van.

**7. ábra:** Kikötés: Csak egy kertben található egy fajta növényből mind a négy darab.

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);

Végül, de nem utolsó sorban van még egy kikötési formánk, de ez nem volt elég hatékony megoldás a többi megoldásunknál már nem ezt használtuk.

constraint forall(t in Tulaj where tulaj[t]=4) (sum(z in Zoldsegek) (termeszt[z,t])=3);

**8. ábra:** Kikötés: Paul pontosan három fajta zöldséget termeszt.

Ezt a „where”-es megoldást implikációra cseréltük le a projektünk további részeiben.

### Feladatunk második verziója

constraint forall(t in Tulaj)(tulaj[t]=4 -> sum(z in Zoldsegek)(termeszt[z,t])=3);

**9. ábra:** Implikáció.

A programunk második verziója csupán a kikötések felépítésében tér el, de mégis sokkal hatékonyabb lett azzal, hogy implikációkat raktunk bele a kódba.

### Harmadik megoldás

A harmadik verzióban már a meglévő mátrix mellé felvettünk még egyet a tulaj-kert kapcsolatokhoz. Az új mátrixot szintén egyesekkel és nullákkal töltöttük fel.

**2. táblázat:** Tulaj-Kert kapcsolatok

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **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 |

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

### Negyedik változat

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

**10. ábra** Halmaz definiálása

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

A tulajokat csak egy egyszerű tömbben tároltuk itt el.

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

Ehhez a „card”-hoz használtunk még helyenként az „intersect” függvényt is, ami kifejezi, hogy az éppen vizsgát halmaz részünk milyen másik halmazzal legyen egy metszetben.

**11. ábra:** Kikötés: Mindenki négy különböző növényt termeszt.

constraint forall(k in Kertek) (card(termeszt[k])=4);

**12. ábra:** Kikötés: Paul pontosan három fajta zöldséget termeszt.

constraint forall(t in Tulaj)(tulaj[t]=4 -> card (Zoldsegek intersect termeszt[t])=3);

A másik újdonság az „array\_union”, ami a tömbben tárolt halmazok unióját képezi.

**13. ábra:** Kikötés: Minden egyes variáció minimum egy kertben megtalálható.

constraint card(array\_union(termeszt))=termenyek;

### Ötödik ötlet

A Kertek utolsó verziójában ismét visszatértünk a mátrixos megoldáshoz, ám itt már nem bináris értékekkel töltöttük fel, hanem konkrétan a terményekkel. A leghasznosabb változtatás itt az volt, hogy a tömböket „enum” típusú adatokkal töltöttük fel, így akár maga az objektum nevét, akár csak a sorszámát írtuk le a kikötésben, mind kettő variációt felismerte a rendszer. 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);

**14. ábra:** Kikötés: Hank nem termel őszirózsát.

Egy plusz kikötést kellett még írnunk ehhez a változathoz az egy megoldás megtartása érdekében. Le kellett fixálnunk a termények sorrendjét, hogy ne adjon ki több lehetőséget is a megoldó, így a sorszámuk alapján növekvő sorrendben rakosgattuk be a növényeket a helyükre. Ennek a módszernek pontos szemléltetése a 2. ábrán látható.

**15. ábra:** Rendezés.

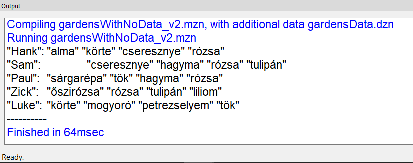
constraint forall(k in Kertek, n in 1..noveny-1)(termeszt[k,n] < termeszt[k,n+1]);

### Verzióktól függetlenül

A kimeneteket egységesen minden verziónál formáztuk a jobb olvashatóság érdekében.

output [show(tulajNev[fix(tulaj[1])]) ++ ":\t"]++[show(termenyNev[t]) ++ " " | t in Termenyek where fix(termeszt[t,1])=1]++["\n"];

**16. ábra:** Output formázás.



**17. ábra:** Output.

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

## **Zebra feladatok**

A bevezetőben bemutatott Einstein-féle úgynevezett Zebra Puzzle-típusú fejtörők[7] modellezése és megoldása tette ki kutatási tevékenységünk nagy részét. Célunk az volt, hogy minél több féle adatszerkezettel és esetleg egy adatszerkezethez is minél több féle módon leírható korlátozásokat készítsünk. A könnyebb kezelhetőség illetve az átláthatóság érdekében a korlátozásokat felépítésük alapján kategorizáltuk, így az ugyan más tartalmú, de hasonló logikai felépítésű kifejezések megírásakor jelentős mennyiségű időt takaríthattunk meg.

A következőkben részletesen bemutatjuk ezen példák megoldására készített modelleket. 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 megvalósítást, illetve ezeken belül további kétféle korlátozás-leírási módot használtunk.

Lássuk is az első 4x5-ös nagyságú példát: adott 4 barát, akik moziba mennek egyik este. A moziban egymás mellé ülnek 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 ül és milyen tulajdonságokkal rendelkezik, a rendelkezésünkre áll 13 állítás, amelyeket modellezésünk során korlátozásokként(constraint) kezeltünk.

Az első verziójú modell esetében a 4 fiú adatainak tárolására tulajdonságtípusonként egy-egy bináris értékeket tartalmazó 2 dimenziós mátrixot deklaráltunk(18. ábra).

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;

[...]

**18. ábra:** Bináris mátrixok deklarációja.

A MiniZinc nyelv sajátosságait kihasználva a sorok indexelésére egy-egy felsorol(enum) típusú tömböt használtunk. Ennek a megoldás kiíratásánál illetve a korlátozások megfogalmazásánál lesz különös szerepe. Ebből következik, hogy a mátrix sorai az egyes tulajdonságok konkrét értékeit az oszlopai pedig sorrendben a fiúk moziban elfoglalt helyét jelölik. Amelynek indexelésére pedig szintén a leíró nyelv sajátossága miatt egy {1..4} értékkészletű segédhalmazt használtunk, ahogy az a fenti 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.

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

constraint forall(p in PEOPLE)(sum(n in NAMES)(name[n, p])=1);

constraint forall(n in NAMES)(sum(p in PEOPLE)(name[n, p])=1);

[...]

**19. ábra:** Állandó sor- illetve oszlopösszeg biztosítása.

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 következhetett a 13 korlátozás megfogalmazása. Ezen példa esetében 6 különböző típusú korlátozást azonosítottunk, 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(20. ábra).

constraint age[fourteen,3]=1;

**20. ábra:** Kikötés: A 14 éves fiú a harmadik helyen ül.

A II. típus hasonló az elsőhöz, de itt a széksor valamelyik széléről nyilatkozunk(21. ábra). Itt kihasználjuk, hogy bináris mátrixszal dolgozunk, ezért két elem összege 1 kell, hogy legyen, miszerint vagy az egyik szélen vagy a másikon ülhet Joshua.

constraint name[Joshua,1]+name[Joshua,db]=1;

**21. ábra:** Kikötés: Joshua valamelyik szélen ül.

A III. típusú korlátozás két tulajdonságot kapcsol össze egymással(22. ábra).

constraint forall(szek in PEOPLE)(name[Joshua,szek] = movie[horror, szek]);

**22. ábra:** Kikötés: Joshua a horrorfilmet szereti.

A IV. típus valamely tulajdonságú személy mellett közvetlenül balra(illetve jobbra) ülő emberről állít valamit(23. ábra). Itt viszont ki kell kötnünk, hogy az első helyen(tehát a bal szélen) nem ülhet a thriller kedvelő, mivel az ő bal oldalán már nem ülhet senki. A korlátozás megfogalmazásánál ez esetben az implikációs operátort(->) 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 következik). Ennek kiváltására használtunk egy továbbfejlesztett korlátozás-leírási mód esetén tisztán relációs operátorokat(24. ábra).

constraint movie[thriller,1]=0;

constraint forall(szek in 2..db)(movie[thriller,szek]=1 -> shirt[black, szek-1]=1);

**23. ábra:** Kikötés – A változat: A fekete inget viselő fiú közvetlenül a thrillert szerető ember bal oldalán ül.

constraint forall(szek in 2..db)(movie[thriller,szek] = shirt[black, szek-1]);

**24. ábra:** Kikötés – B változat: A fekete inget viselő fiú közvetlenül a thrillert szerető ember bal oldalán ül.

Az V. típusnál adott ember valamely tulajdonságú ember mellett balra(ill. jobbra) tetszőleges pozícióban foglal helyet(25. ábra). A második verzióban szintén relációs operátort(kisebb-egyenlő) használtunk az implikáció kiváltására(26. ábra).

constraint forall(x in PEOPLE)(age[eleven,x]=1 -> sum(szek in 1..x-1)(shirt[black, szek])=1);

**25. ábra:** Kikötés - A változat: A fekete inges fiú a legfiatalabbtól valamelyik balra eső helyen ül.

constraint forall(x in PEOPLE)(age[eleven, x] <= sum(szek in 1..x-1)(shirt[black, szek]));

**26. ábra:** Kikötés - B változat: A fekete inges fiú a legfiatalabbtól valamelyik balra eső helyen ül.

A VI. típusnál egy ember valamely két másik között tetszőleges pozícióban foglal helyet(27. ábra). Ebben az esetben viszont a logikai operátor relációsra cseréléséhez két részre kellett bontanunk az állítást(28. ábra).

constraint forall(x in PEOPLE, y in PEOPLE)(age[thirteen,x]=1 /\ movie[action,y]=1 -> sum(szek in x+1..y-1) (shirt[red,szek])=1);

**27. ábra:** Kikötés - A változat: A piros inget viselő fiú valahol a 13 éves és az akció kedvelő között ül, ebben a sorrendben.

constraint forall(x in PEOPLE)(movie[action, x] <= sum(szek in 1..x-1)(shirt[red, szek]));

constraint forall(x in PEOPLE)(age[thirteen, x] <= sum(szek in x+1..db)(shirt[red, szek]));

**28. ábra:** Kikötés – B változat: A piros inget viselő fiú valahol a 13 éves és az akció kedvelő között ül, ebben a sorrendben.

A kimenet képzése a nyelvnek megfelelő szintaktikával valósult meg(29. ábra).

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

[...]

**29. ábra:** Kimenet

A format függvény pusztán esztétikai formázást, a show karakterlánccá konvertálást, a fix függvény pedig a változók értékének „stabilizálását” végzi. A where záradékban pedig azt vizsgáljuk, hogy a bináris mátrixban hol található 1-es érték, mert csak azokat az értékeket íratjuk ki.

Az eddigiekben részletezett bináris mátrix mellett egy másik, ettől merőben eltérő adatszerkezettel rendelkező implementációt is készítettünk. Ezt a második, 5x6-os méretű feladatunkon, a „Fundraising Dinner” címűn keresztül fogjuk bemutatni. Néhány szó magáról a feladatról: a leírás szerint 5 jómódú hölgy egy jótékonysági 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;

[...]

**30. ábra:** Tömbök deklarációja.

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);

[...]

**31. ábra:** Egyediség biztosítása.

A korábban már részletezett korlátozástípusokat a következőképpen modelleztük jelen adatstruktúra esetében. 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.(32 ábra) és IV.(33. ábra) típus esetén egyféle leírási mód választható csak.

forall(p in PEOPLE)(name[p]=Jane <-> donation[p]=twentythousand);

**32. ábra:** Kikötés: Jane 20000 dollárt adományozott.

A IV. típusú korlátozás leírását is az ekvivalencia operátor(<->) használatával valósítottuk 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 elkerülhetjük a tömbök bejárása során a kiindexelést.

forall(p in 1..db-1)(cocktail[p]=cosmopolitan <-> age[p+1]=sixty) /\

(cocktail[db]!=cosmopolitan) /\

(age[1]!=sixty);

**33. ábra:** Kikötés: A legidősebb hölgy közvetlenül a cosmopolitant ivó jobbján ül.

Az V. típust pedig a 34. ábrán látható módon, az implikáció operátor segítségével modelleztük. B változat esetében pedig az indexekkel való bejárásnál fogalmaztunk meg egy apróbb szabályt(35. ábra).

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);

**34. ábra:** Kikötés – A változat: Aki margaritát iszik, az a kék ruhás nőtől balra ül.

forall(a,b in PEOPLE where a<b)(not(cocktail[a]=margarita /\ dress[b]=blue));

**35. ábra:** Kikötés – B változat: Aki margaritát iszik, az a kék ruhás nőtől balra ül.

A VI. korlátozástípusnál szintén kétféle modellt készítettünk, az A változatnál logikai operátorral és a később részletesen taglalt visszafelé következtetés módszerével. B verziónál viszont a count függvény segítségével számoljuk meg

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);

**36. ábra:** Kikötés – A változat: A piros ruhás hölgy valahol a 20000 és a 40000 dollárt adományozó személyek között ül, ebben a sorrendben.

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);

**37. ábra:** Kikötés – B változat: A piros ruhás hölgy valahol a 20000 és a 40000 dollárt adományozó személyek között ül, ebben a sorrendben.

Ezen felül ennél a feladatnál azonosítottunk egy újfajta korlátozást, amely egy objektum valamely oldaláról nyilatkozik, viszont nem rögzíti, hogy melyikről. Ezeknél egy korábban bevezetett új elemet, az úgy nevezett visszafelé következtetést is használtuk. Az alábbi példát megvizsgálva észrevehetjük, 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. Ehhez az xor azaz a kizáró vagy operátort hívtuk segítségül. A széksor széleit a IV. típusnál részletezett okok miatt ismét külön kezeltük.

forall(p in 2..db-1)(name[p]=Lidia -> (cocktail[p-1] = cosmopolitan xor cocktail[p+1]=cosmopolitan)) /\

forall(p in 2..db-1)(cocktail[p]=cosmopolitan -> (name[p-1] = Lidia xor name[p+1]=Lidia)) /\

(name[1]=Lidia -> cocktail[2]=cosmopolitan) /\

(name[db]=Lidia -> cocktail[db-1]=cosmopolitan) /\

(cocktail[1]=cosmopolitan -> name[2]=Lidia) /\

(cocktail[db]=cosmopolitan -> name[db-1]=Lidia);

**38. ábra:** Kikötés: Lidia a cosmopolitant ívó hölgy mellett foglal helyet.

Az eredmény kiíratásánál itt kihasználtuk azt, hogy a tömb elemeit nem kell vizsgálnunk, hanem sorrendben, vizsgálat nélkül kiírathatóak.

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

**39. ábra:** Kimenet.

## **Teszteredmények**

A kiválasztott példák különféle módokon történő modellezésének végeztével, az így rendelkezésünkre álló modelleket átfogó teszteknek vetettük alá. 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, ezzel kezdtük a tesztelést. Ekkor a MiniZinc IDE és a Gecode megoldó két verzióját is összevetettük egymással.

**40. ábra:** Futásidők összehasonlítása a MiniZinc és a Gardens példa verziói esetén.

A 40. ábra tartalmazza a futási eredményeket milliszekundumban(ms) mérve. Megoldóként ez esetben a Gecode-ot használtuk. Az átlagot mindegyik esetben 6 futtatás eredményéből számoltuk. Az adatokból több következetés is levonható.

Egyrészről, a 2.1.6-os verzió minden esetben valamivel lassabban oldotta meg példáinkat, így kimondhatjuk, hogy készítői láthatóan javítottak termékükön és optimalizálták a hatékonyságát.

A második szembetűnő jelenség, miszerint az első általunk készített alapmodell, 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 adatreprezentáció választása. Részben a lassú megoldáskeresés, részben pedig a bonyolult megvalósítás miatt sem.

A bináris mátrixok illetve halmazok használata között jelentős futásidőbeli eltérés nem volt. Főként annak tükrében, hogy az újabb verziónál teljesen azonos volt az átlagos futásidő mindkét modelltípus 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.

**41. ábra:** Zebra feladatok futásidejeinek(ms) összehasonlítása.

Amint az a 41. ábrán látható a „Fundraising Dinner” feladat esetében hosszabb futásidőket tapasztaltunk, 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 megoldónk egy nagyobb és nehezebb, több hozzárendeléssel dolgozó feladat esetében is képes a futásidőket abszolút alacsony szinten tartani. Ez komoly érv amellett 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 növelésekor mutatkozik meg igazán.

Emellett azt tapasztaltuk, hogy 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 adatreprezentációt részesítjük előnyben.

Összegzésként, a végső konzekvenciákat levonva: a korlátprogramozás a kifejezetten rövid megoldási és futásidőket figyelembe véve, egy borzasztóan hatékony eszköz az általunk vizsgált logikai fejtörők, mint speciális hozzárendelési feladatok modellezéséhez. A gyakori és sokak által ismert Einstein-féle feladatok esetében válasszuk a felsorol típussal indexelt tömbök használatát. Viszont az olyan nem hétköznapi példáknál – amilyen esetünkben 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.

# Redundáns megkötések megkeresése

A példák korlátozásainak modellezése során felmerült bennünk annak lehetősége, hogy redundáns kifejezések fordulhatnak elő a feladatokban, azaz vannak olyan megkötések vagy megkötés-halmazok, amelyek elhagyása esetén továbbra is egyetlen, az eredeti feladatkiírásnak megfelelő megoldást találunk. Kutatásunk utolsó szakaszában célunk ezek megkeresése, továbbá annak meghatározása volt, hogy elhagyásuk miképpen befolyásolja, vagy befolyásolja-e egyáltalán a különböző megoldók futási illetve megoldási idejét.

Mivel megoldóink gyorsasága már a korábbiakban bebizonyosodott, ezért jó eséllyel gondolhattunk arra, hogy amennyiben egy vagy több korlátozást eltávolítunk abban az esetben rövid idő alatt meg tudjuk határozni, hogy továbbra is egyértelmű-e az eredmény. A feladatok jellegéből adódóan egyértelműnek azt az esetet tekintettük, amikor a program lefutása után csak egy megoldást találtunk, amely az eredeti – minden kikötést tartalmazó – feladatkiírásnak is megfelel. Módszerünk a következő volt: minden kikötést egy elágazásban helyeztünk el a 42. ábrán látható módon.

constraint if kivesszuk[4]=1 then true else

forall(p in PEOPLE)(necklace[p]=sapphire <-> age[p]=fiftyfive)

endif;

**42. ábra:** Kikötés elágazásba ágyazása.

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 hagyjuk. Amennyiben viszont az adott sorszámú elem 0 értékű, akkor a különben ágban elhelyezett, a feladatleírásnak megfelelően modellezett kifejezést kezeljük korlátozásként.

Ezen elvet követve mindegyik korlátozást egy a x. ábrához hasonló vezérlési szerkezetbe ágyaztunk. A „kivesszuk” tömböt pedig a könnyebb szerkeszthetőség érdekében nem a modellen belül definiáltuk, hanem értékét külön fájl(ok)ból olvastuk be. Mivel egyes példák nagy mennyiségű kikötést tartalmaznak és ebből következően rengeteg féle kombinációban tudnánk ezeket elhagyni, így igyekeztük ezeket a teszteket valamilyen szinten automatizálni. Az adatfájlokat (amelyekben a „kivesszuk” tömböt definiáltuk) egy C# programmal generáltuk oly módon, hogy első körben mindegyikben a tömbnek pontosan egy eleme lesz 1 értékű, így értelemszerűen a korlátozások darabszámával egyező adatfájl keletkezett. Ezeket parancssorból a modellel együtt parancssorból futtattuk és feljegyeztük azokat a sorszámokat, amelyek esetén egyértelmű eredmény született. Második körben az imént említett C# program segítségével legeneráltuk 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. Az általunk feldolgozott példák esetében a redundáns kikötések eloszlásait a 3. táblázat tartalmazza.

**3. táblázat:** Elhagyható korlátozások eloszlása

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Movies Night** | **Fundraising Dinner** | **Gardens** |
| **Önmagában elhagyható korlátozások száma** | 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 választhatunk ki tetszőleges kombinációban 2-t vagy többet. Azaz már meg kellett válogatnunk, hogy pontosan melyek együttes elhagyása ad egyértelmű eredményt. Tehát például ha két kifejezés külön-külön elhagyható, az korántsem jeleni azt, hogy ezek egyszerre is elhagyhatók. Vegyünk egy konkrét példát: a „Fundraising dinner” esetében az elhagyható korlátozásokból(7) alkotható összes 2 tagú kombináció száma, 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 spektrumunk, 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 számítógéppel oldattuk meg, amely értelemszerűen az emberénél 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 módosítatlan, 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 hatékonyságáról ismét megbizonyosodhattunk, mivel kevesebb korlát, így a megoldáskeresés során kevesebb támpont esetén is komoly gyorsasággal adtak eredményt.

# 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. Sutherland, I. E. (1964). Sketchpad a man-machine graphical communication system. *Transactions of the Society for Computer Simulation*, *2*(5), R-3.
2. Schulte, C., Tack, G., & Lagerkvist, M. Z. (2010). Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*
3. Perlovsky, L. I. (2007). The mind vs. logic: Aristotle and Zadeh. *Critical Review*, *1*(1), 30-33.
4. Browne, C. (2013, August). Deductive search for logic puzzles. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on* (pp. 1-8). IEEE.
5. Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, *2*(1‐2), 83-97.
6. http://www.mathsisfun.com/puzzles/gardens.html
7. https://www.brainzilla.com/logic/zebra/