

Zebra típusú logikai rejtvények megoldása evolúciós algoritmussal

Szili Dániel, Schöffer Fruzsina, Tóth Sándor Balázs, Varga Máté

Témavezető: Dr. Hegyháti Máté

2018. március 2.

Tartalomjegyzék

1. Bevezetés	2
2. Zebra rejtvények	4
2.1. Történet és szerkezet	4
2.2. Megoldhatóság, egyértelműség	4
2.3. Megoldó módszerek	7
3. Evolúciós algoritmusok	8
4. Evolúciós algoritmus Zebra rejtvények megfejtésére	12
4.1. Kódszerkezet	12
4.2. Egyedreprezentáció és segédfüggvények	14
4.3. Evolúciós mechanizmusok	16
4.3.1. Véletlenszerű új egyed generálás	17
4.3.2. Mutálás	18
4.3.3. Keresztezés	20
4.3.4. Megold függvény	22
4.4. Egyedek kiértékelése	23
5. Empírikus vizsgálatok	24
6. Kód automatikus generálása	26
7. Összefoglalás	28
Irodalomjegyzék	29
A. Kódgeneráló táblázat	31

1. fejezet

Bevezetés

A logikai rejtvény olyan feladvány, mely a matematika levezetési, bizonyítási területéről származik. Az ilyen jellegű fejtörők már a sumér társadalom emberét is foglalkoztatták. Logikai rejtvényt először Charles Lutwidge Dodgson¹ angol író publikált. A logika játéka (The Game of Logic) című 1886-ban megjelent könyvében egy olyan játékot mutatott be, melyben kijelentések alapján kellett megerősíteni a következtetést[2]. Néhány, napjainkban is népszerű non-verbális rejtvény:

Sudoku melyben megadott szabályok szerint számjegyeket kell elhelyezni egy táblázatban

Nonogram vagy grafilogika, melyben helyes dedukcióval egy rácsot kell fekete-fehér négyzetekkel kitölteni, ami egy képet alkot

Logikai labirintus melyben a dedukció felhasználásával a labirintus szabályait kell kitálni.

Egy igen közkedvelt típus továbbá az úgynevezett zebra feladvány, mely Albert Einstein híres rejtvényéről kapta nevét, mivel állítólag ő találta fel kisfiúként. Számos különböző változata létezik, akad köztük olyan, melyet már az '60-as években publikáltak. Gyakran állítják, hogy az emberiség csupán két százaléka képes megoldani az Einstein-rejtvényt. Ezek megoldása kutatott területnek számít, több különböző matematikai és informatikai megoldás is került már publikálásra, mint például a korlátprogramozás, visszalépéses keresések, vagy a feladvány logikai programozás alapon történő megközelítése.

A modern mérnöki alkalmazásokban, gépi tanulásban is egyre nagyobb népszerűségnek örvendő evolúciós algoritmusok is alkalmasak többek között ilyen jellegű kielégíthetőségi feladatok megoldására. Az evolúciós algoritmus olyan számítógépes problémamegoldó rendszer, amely a biológiai evolúció mechanizmusára épül. A genetikus algoritmus optimalizációs eljárás, ahol a lehetséges hipotéziseket az egyedek, a hipotézis tér valamely részét pedig a populációk reprezentálják. Az algoritmus megegyezik a számítástechnikában és a mesterséges intelligenciában alkalmazott heurisztikus keresési technikákkal, ami kiegészül még az

¹Írói álnevéen Lewis Carroll

evolúciós biológiában ihletett technikák alkalmazásával: mutáció, szelekció, reprodukció és rekombináció.

Fontosabb felhasználási területei:

- autóipari és mérnöki tervezés
- robotika
- biomimetikus találmányok
- útvonalválasztás utazás vagy szállítmányozás során
- titkosítás és kódtörés
- értékesítés

Munkánk során egy genetikus algoritmust dolgoztunk ki, és teszteltük több példán. A futási idő csökkentése érdekében az algoritmus különböző paraméterei finomhangolásra kerültek több tesztet futtatásának tapasztalatai alapján.

Az empirikus vizsgálatok automatizálásának érdekében több rejtvényt elemezve az ezekben előforduló megkötések rendszerezve lettek. Egy olyan módszer került kidolgozásra, mely a feladat egy formális leírásából megoldó kód generálására képes. Az így keletkezett rendszerrel könnyedén vizsgálhatóvá váltak további feladatok, melyeken futtatott tesztek segítségével az algoritmus működése tovább javítható.

Dolgozatunkat a Zebra rejtvények részletesebb bemutatásával kezdjük. Szó esik a történetéről, szerkezetéről, és hogy milyen részekből épül fel. Említésre kerülnek még a megoldhatóság eltérő lehetőségei, valamint az irodalomban megtalálható megoldó módszerek. Ezt követően bemutatjuk az evolúciós algoritmusokat, azok történetét, főbb működési elveit. Folytatásként betekintést nyújtunk arról, miképp alkalmaztunk evolúciós algoritmust Zebra rejtvények megfejtésére. Ismertetjük a kódszerkezetet, az egyedreprezentációt és segédfüggvényeket, bemutatjuk az evolúciós mechanizmusokat, majd végül az egyedek kiértékeléséről esik szó. Ezt követően futtatási eredményeket mutatunk be, valamint demonstráljuk kódunk automatikus generálásának elgondolását és kivitelezését. Végezetül összefoglaljuk munkánk eredményeit.

2. fejezet

Zebra rejtvények

A logikai feladványok egyik legnépszerűbb fajtáját képezik az úgynevezett zebra típusú rejtvények, melyek különböző logikai következtetések segítségével oldhatók meg. A 2.1 alfejezetben a történetük és szerkezetük kerül bemutatásra, a 2.2 alfejezetben a megoldhatóságukat, egyértelműségüket vizsgáljuk. A 2.3 alfejezetben pedig különböző megoldó módszerek kerülnek bemutatásra.

2.1. Történet és szerkezet

A leghíresebb zebra típusú rejtvény Albert Einstein nevéhez fűződik, melyről a logikai feladványoknak ez a fajtája a nevét is kapta, hiszen az Einstein-példa egyik változatában a zebra tulajdonosának kiléte a fő kérdés. Úgy tartják, hogy Einstein ezeknek a feladványok az atyja, aki állítólag gyerekként találta ki a rejtvényt és azt vallotta róla, hogy az embereknek csupán 2 százaléka képes azt megoldani.

Ez a példa kiválóan alkalmas arra, hogy reprezentálja az ehhez hasonló feladványok felépítését. Adott 5 ház egy sorban. Minden háznak 5 sajátossága ismert: a ház színe, a tulajdonos nemzetisége, foglalkozása, valamint az általa kedvelt ital és a háziállat, amit tart. Azonban egyik ház sajátosságait sem ismerjük kezdetben. Einstein állításokat adott meg a feladványához, melyek segítségével egyértelműen meghatározható mind az 5 ház összes tulajdonsága. Ehhez csupán logikus következtetésekre van szükségünk, és lépésről lépésre megoldható a rejtvény.

2.2. Megoldhatóság, egyértelműség

A zebra típusú rejtvények esetében rendkívül fontos az, hogy kellő mennyiségű állítás, megkötés álljon rendelkezésünkre, illetve az is elengedhetetlen, hogy ezek pontosak legyenek. Az alábbi rövid példák ezek fontosságára hívják fel a figyelmünket, valamint egy rövid, egyszerű feladat is megoldásra kerül. Példánkban adott 3 tanuló. Tudni szeretnénk, hogy az adott sorszámú tanulónak mi a neve, valamint az életkora.

Egy utcában öt különböző színű ház van egymás mellett. A házakban különböző nemzetiségű és foglalkozású emberek laknak. Mindenki különböző háziállatot tart és más-más a kedvenc italuk is. A következőket tudjuk.

Az angol a piros házban lakik.

A festő japán.

A norvég a balszélső házban lakik.

A zöld ház a fehérnek jobboldali szomszédja.

A diplomata a sárga házban lakik.

A hegedűművész gyümölcslevet iszik.

Az orvos szomszédja rókát tart.

A spanyol kutyát tart.

Az olasz a teát kedveli.

A zöld házban lakó kávét iszik.

A szobrász csigát tart.

A tejet a középső házban kedvelik.

A norvég a kék ház mellett lakik.

A diplomata melletti házban lovat tartanak.

Kérdés: Kinek a háziállata a zebra?

2.1. ábra. Az Einstein-példa

A megoldás a már korábban megismert állítások segítségével fejthető meg. Fontos azonban, hogy ezek pontos állítások legyenek és ne ütközzünk ellentmondásba, hiszen akkor nem tudunk jó megoldást találni. Például:

- Az 1. pozícióban lévő tanuló neve Peti.
- A 2. pozícióban lévő tanulótól balra lévő tanuló neve Réka.

Mint ahogy a 2.2 ábrán látható, ebben az esetben a két állításnak nem tudunk egyszerre eleget tenni, így nem kapunk megoldást.

	1.	2.	3.
Név	Peti-Réka		
Életkor			

2.2. ábra.

Az sem jó eset, ha túl kevés megkötést adunk meg, ekkor ugyanis több lehetséges megoldás is létezik:

- Az 1. pozícióban lévő tanuló neve Peti.
- A 20 éves tanuló a 25 és a 30 éves tanulók között helyezkedik el, de nem biztos, hogy ebben a sorrendben.
- A 20 éves tanulót Rékának hívják.

Ekkor a 2.3 és a 2.4 ábrák szerinti megoldások mindegyike helyes lenne.

	1.	2.	3.
Név	Peti	Réka	Jani
Életkor	30	20	25

2.3. ábra.

	1.	2.	3.
Név	Peti	Réka	Jani
Életkor	25	20	30

2.4. ábra.

Egy helyes példa és annak megoldása pár lépésben a következőképp néz ki:

- Az 1. pozícióban lévő tanuló neve Peti.

- A 20 éves tanuló a 25 és a 30 éves tanulók között helyezkedik el, de nem biztos, hogy ebben a sorrendben.
- A 20 éves tanulót Rékának hívják.
- A 30 éves tanuló neve Jani.

Az első 3 állításból egyértelműen látszik, hogy hol helyezkedik el Peti és a 20 éves tanuló, illetve kiderül, hogy a 20 éves diák Réka. Ezt követően pedig könnyedén kikövetkeztethető a feladat megoldása, ugyanis Jani már csak a 3. pozícióban foglalhat helyet. Az ő életkorának a 4. állítás alapján történő beírása után pedig látszik, hogy Peti a 25 éves tanuló, így a táblázatot hiánytalanul kitölthetjük. Ehhez csupán kellő mennyiségű egymásnak ellent nem mondó, helyes állítás szükséges. A megoldást a 2.5 ábra mutatja.

	1.	2.	3.
Név	Peti	Réka	Jani
Életkor	25	20	30

2.5. ábra.

2.3. Megoldó módszerek

Logikai feladványok matematikai eszközökkel történő automatizált megoldását már többen vizsgálták az irodalomban. Merelo-Guervós, Castillo, Esparcia-Alcázar és García 2013-mas értekezésükben heurisztikus megoldási módszereket vizsgáltak az ismert Mastermind logikai feladaton[9]. T W Jerardi 1987-ben megjelent cikkjében egy Smullyan feladvány megoldását a prolog nyelvvel vizsgálja[6]. Deon Garrett , Joseph Vannucci, Rodrigo Silva, Dipankar Dasgupta és James Simien 2005-ben foglalkoztak az úgynevezett „Sailor assignment” feladvány megoldásával, evolúciós algoritmus segítségével[3].

A zebra rejtvények megoldására is már több különböző módszer is született. Az egyik ilyen az Einstein-feladvány megoldása Prolog nyelven kényszerfeltétel kielégítési modul segítségével. Ekkor a program számára a feladat ismertetéséhez 25 változó kerül bevezetésre, melyek a házakhoz tartozó tulajdonságokat jelölik. A rendszer ennek segítségével találja meg a megoldást. A zebra feladványok megoldhatók visszalépéses keresés segítségével is, amikor a keresési teret úgy kell elképzelni, mint egy fastruktúrát, ahol a gyökértől kiindulva egy csúcsot keresünk. További megoldási lehetőség az úgynevezett korlát logikai programozás, ahol a feladvány, mint korlátkielégítési probléma jelenik meg. Ez a megoldó módszerek közül az egyik leggyorsabb, mivel minden alkalommal a korlátozásnak leginkább eleget tevő változó kerül kiválasztásra, így jelentősen csökkenthető a keresési idő. Salvati és társai ilyen megoldások hatékonyságát vizsgálta[11].

3. fejezet

Evolúciós algoritmusok

Az evolúciós algoritmus olyan számítógépes problémamegoldó rendszer, amely az evolúció mechanizmusára épül. Az evolúciós algoritmusokat John Holland dolgozta ki tanítványaival a Michigani Egyetemen az 1960as- 70es években. Céljuk a különböző evolúciós jelenségek (szelekció, adaptáció) formális felírása, elemzése, illetve ezeknek a jelenségeknek a számítástechnikai környezetben történő alkalmazása volt. Később ezeket az algoritmusokat mások optimalizációs célokra kezdték el használni. Goodman 2012-es értekezése a genetikus algoritmusokba való bevezetést és azok egyszerű megértését segíti[4]

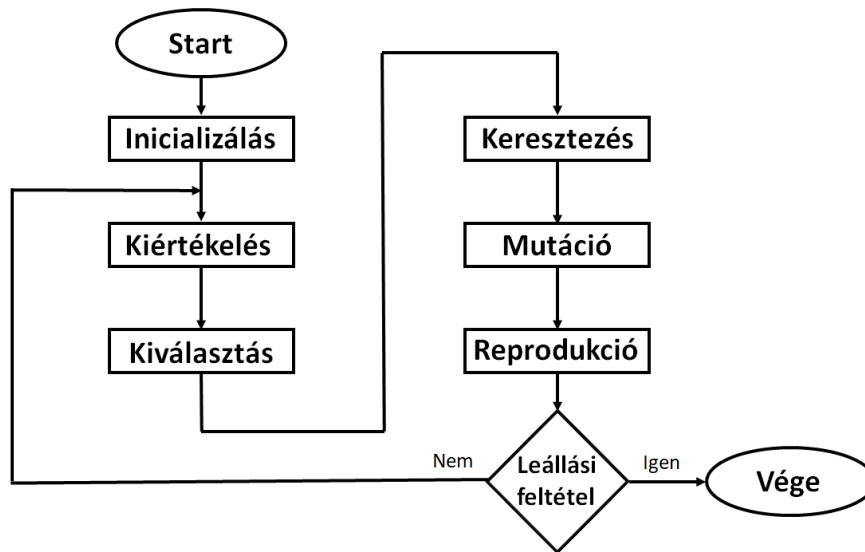
Az evolúciós, vagy más néven genetikus algoritmusok működési elvének alapja az evolúció darwini elméletén nyugszik, mely során a természetben a folyamatosan változó életkörülményekhez való alkalmazkodás a fő szempont. Az evolúciós algoritmusok esetében az optimalizáláskor mindvégig azonos feladat megoldását keressük.

Az egyes egyedek reprezentációja során a kódolásához használt struktúra a kromoszóma vagy genotípus. Génkészletünket a genomok alkotják. Egy genom egy vagy két kromoszómából áll, a kromoszómák pedig génekből épülnek fel. A gén lehet egy bitsztring, alkothatják egész vagy valós számok vagy akár szimbólumok. A gén értékét az allél vagy génváltozat határozza meg.

Az evolúciós algoritmusok esetében az egyedek, vagyis a különböző megoldások készletét populációnak nevezzük. Egy populáción belül az ismétlés megengedett, bár nem feltétlenül ajánlott. Az elemek száma, vagyis a populáció mérete a feladat sajátosságaitól függ, általában 50-100 egyedet tartalmaz.

Mint ahogyan a 3.1 ábra is mutatja, az indítást követően az első fő lépés az inicializáció, mely során feltöltjük a populációkat egyedekkel, melyeket előállíthatunk véletlenszerűen vagy heurisztikus módszerekkel. Múltbéli tapasztalatainkra hagyatkozva készíthetünk egyre jobb egyedeket. A rossz megoldásokon később viszonylag könnyen lehet javítani.

Az inicializációt követi a kiértékelés, mely során az evolúciós algoritmus a rátermettségi függvény alapján eldönti, hogy melyik megoldás mennyire jó. Ennek a függvénynek az értékét az úgynevezett célfüggvény segítségével állítjuk elő, amely az optimális probléma megoldásának értékét jelenti. Ritkább esetekben előfordulhat, hogy a két függvény értéke megegyezik.

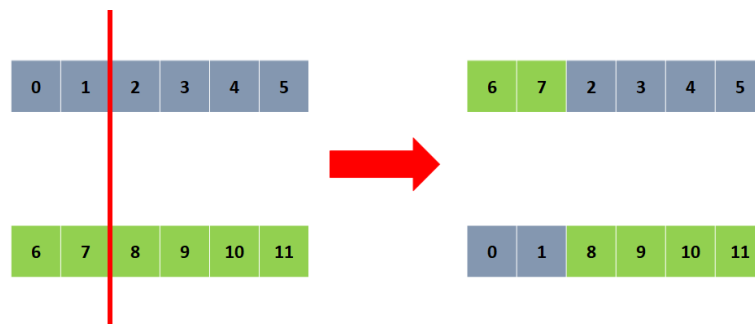


3.1. ábra. Az evolúciós algoritmus általános felépítése

Az aránytalanságok elkerülése érdekében fontos szerepe van a rangsorolásnak, mivel nem előnyös az, ha például néhány „túl jó” egyed hoz létre sok utódot, ugyanis hasonló egyedek keresztezésével nem jutunk előre.

A 3. fő lépés a kiválasztás, mely során az egyedek rátermettségi értéke alapján válogatunk közöttük. Ilyen módszerek például a rulettkerék kiválasztás, versengő kiválasztás, valamint a levágó kiválasztás.

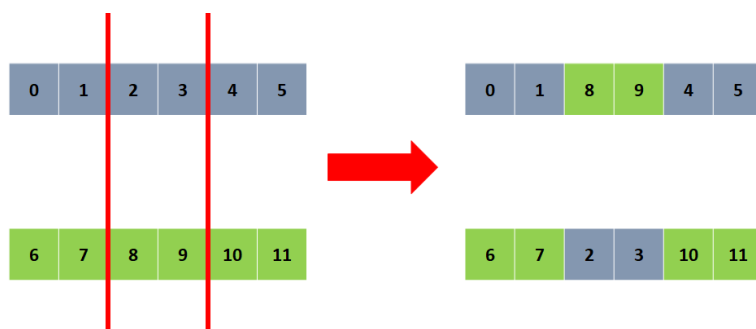
A kiválasztást követi a keresztezés, melynek két elterjedt fajtája például az egy- illetve kétpontos keresztezés. Az előbbi esetében a kromoszómákat egy véletlenszerűen választott helyen kettévágjuk, majd a felcserélt fél-kromoszómákból új egyedeket hozunk létre. Kétpontos keresztezés esetében hasonló módon járunk el, ekkor azonban egy helyett két vágási pont is kiválasztásra kerül.



3.2. ábra. Az egyponos keresztezés

Keresztezés végezhető még például egyenletes keresztezés, direkt kódolás, illetve sorrendi kódolás segítségével is.

A következő lépés a mutáció, melynek lényege, hogy bizonyos gének értékeit megvál-



3.3. ábra. A kétpontos keresztezés

toztatjuk. A direkt kódolással történő mutáció esetében a véletlenszerűen választott gén értékét véletlenszerűen módosítjuk. Ennek során csak egy gén értéke változik meg, ezért kis valószínűséggel mutálnak. A mutáció végrehajtható sorrendi kódolással is, mely során két véletlenszerűen választott gén kerül felcserélésre. De miért is van szükségünk mutációra? Egyrészt segít az eddig az egyedekben elő nem forduló, jó allélok felfedezésében, lehetővé teszi a populációból a generációk során kikerült, de valójában jó génértékek visszaállítását. Szerepe van továbbá a globális optimum környezetének megtalálásában, illetve a genetikai változatosság növelésében is.

A leállási feltétel megvizsgálása előtt már csak egy fő lépés van hátra: a reprodukció. Az új gének a már eleve meglévő, illetve a keresztezés és mutáció során keletkezett átmeneti gének elemeiből jönnek létre. Léteznek úgynevezett kanonikus genetikai algoritmusok is, melyek teljesen leváltják az előző gént. Az újonnan keletkezett utódok számának alakulása befolyásolható, mégpedig úgy, hogy a rátermettség alapján eldöntjük, hogy a korábbi generáció utódai közül melyek íródjanak felül. Fontos szerepe van az elitizmusnak is, melynek lényege, hogy az előző generáció legrátermettebb egyedeit megtartjuk.

A kilépési feltétel többféleképpen megadható. Kilépés történhet a generációk száma alapján vagy az időkorlát megadásával. Megadható továbbá a konvergencia vizsgálatával, például akkor, ha az utóbbi meghatározott számú generáció alatt nem sikerült javítani a legjobb egyedeken. Bizonyos esetekben egyértelműen eldönthető, hogy megtaláltuk-e a megoldást, például akkor, ha 0 hibapontos egyedet kaptunk.

Az evolúciós algoritmusok rendkívül széles körben felhasználhatóak és egyre nagyobb népszerűségnek örvendenek. Alkalmazzák például az üzleti szférában cégek a beszerzési és értékesítésre vonatkozó döntések meghozatalakor, az autóiparban az autók festésének koordinációjára, vagy mezőgazdasági termelés optimalizálásában is. Sami Khuri, Tim Walters és Yanti Sugono már 2000-ben kutatta genetikai algoritmus segítségével a gráfok, és gráfok színeinek színezését[7], míg Gizem Sungu és Betul Boz 2015-ben adtak ki tanulmányt gráf-színezési problémákat vizsgáló evolúciós algoritmusok módszeréről[14]. Raghavjee és Pillay 2010-es publikációjukban foglalkoztak a genetikai algoritmus mindennapi problémamegoldó képességével, egy iskolai órarend beosztása formájában[10] Adrian Brezulianu Monica Fira és Lucian Fira 2009-es tanulmányukban ugyancsak a genetikai algoritmusok mindennapos

alkalmazását tesztelték a munkahelyi beosztások elkészítésében[1]

4. fejezet

Evolúciós algoritmus Zebra rejtvények megfejtésére

Célunk az volt, hogy a Zebra típusú feladványokat olyan algoritmus felhasználásával oldjuk meg, mely nem csupán egy konkrét példa megoldására képes, hanem bármilyen, mi általunk megszabott feladatot meg tud oldani. Ezen céloknak az evolúciós algoritmusok tökéletesen megfeleltek.

Az általunk kidolgozott módszert C nyelven implementáltuk, GitHub-on verziókezeltük, és a Travis-CI eszköz segítségével ellenőriztük a kód szintaktikai helyességét automatikusan.

A 4.1. fejezetben az általunk elkészített és implementált módszer kódszerkezete kerül bemutatásra. Ezt követően a 4.2. fejezetben mutatjuk be az egyedreprezentációt és a kapcsolódó segédfüggvényeket. Ezek után a 4.3. fejezetben részletezésre kerülnek az evolúciós mechanizmusok:

- a véletlenszerű új egyed generálás
- a mutálás
- a keresztezés
- valamint a populációk menedzselése

Végül a 4.4 fejezetben az egyedek kiértékeléséről lesz szó.

4.1. Kódszerkezet

A kód alapvetően 2 logikai egységre szedhető szét. Van egy egységes, genetikus kód rész és egy, az adott feladatra vonatkozó specifikációkat tartalmazó rész. Ez a genetikus algoritmusban három különböző részre jól elkülönítve látszik. A genetikus rész kettő .c fájlban található egy main és egy a genetikus függvényeket tartalmazó állományban. Így egy újabb példára való futtatás alkalmával elég csak a tesztek tartalmazó részt átírni.

A genetikus rész legfontosabb függvényeiről a későbbiekben még lesz szó, ezek a mutálás, random kezdőegyedek beállítása, valamint a keresztezés. Azonban emellett más függvények is fontos szerepet játszanak az algoritmus lefutásában. A 4.1 ábrán látható megjelenítésért az `egyedKiir` függvény a felelős, amely az aktuálisan létrehozott populáció legjobb egyedét hivatott bemutatni a programot futtató felhasználó számára. Majd végül kiírja a jó megoldást is. A kódba bele van építve egy `joMegoldasTeszt` nevű függvény, ami nincs használatban és a main függvényben is ki van kommentezve. Ez arra szolgál, hogy amikor újabb feladatot akarunk megoldani a programmal, és ismerjük a megoldását, akkor esetleges futási hiba esetén ezzel le tudjuk ellenőrizni, hogy a megkötések implementálásában van-e hiba és ha igen akkor melyikben.

```

C:\WINDOWS\system32\cmd.exe
Generacio 0:
red      black  green  blue
Joshua   Ryan    Nicholas Daniel
comedy   thriller action horror
popcorn  cookies crackers chips
13       14       11       12
10 (3)(4)(5)(6)(7)(8)(9)(10)(11)(13)

Generacio 1:
green    blue    black  red
Joshua   Ryan    Nicholas Daniel
comedy   action  horror thriller
cookies  chips   popcorn crackers
11       12       14       13
5 (2)(3)(5)(9)(10)

Generacio 2:
green    red      black  blue
Joshua   Ryan    Nicholas Daniel
horror   comedy  action thriller
cookies  popcorn crackers chips
13       11       14       12
4 (1)(2)(3)(10)

Generacio 3:
black    red      green  blue
Joshua   Ryan    Nicholas Daniel
horror   comedy  action thriller
cookies  popcorn crackers chips
13       11       14       12
3 (8)(10)(13)

Generacio 7:
green    red      black  blue
Joshua   Ryan    Nicholas Daniel
horror   comedy  action thriller
cookies  popcorn chips   crackers
13       12       14       11
2 (9)(10)

Generacio 8:
green    red      black  blue
Joshua   Ryan    Nicholas Daniel
horror   comedy  action thriller
chips    popcorn crackers cookies
13       12       14       11
1 (10)

*****

green    red      black  blue
Joshua   Ryan    Nicholas Daniel
horror   comedy  action thriller
popcorn  chips   crackers cookies
13       12       14       11
0

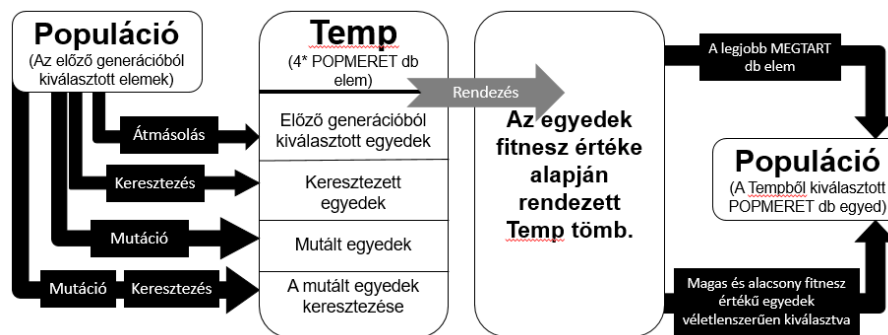
Press any key to continue . . .

```

4.1. ábra. A program lefutása

A main függvényben `joMegoldasTeszt` függvény mellett jelen van a `Megold` függvény is. Ez a függvény felelős a feladatok megoldásáért. A tesztek alapján beállítottunk egy fitnessz függvényt, mi esetünkben ezt `hanyatSertnek` nevezzük. A program célja az, hogy a fitnessz értéke 0 legyen, ami akkor valósul meg, ha minden teszt feltételének eleget tesz a program. Ez a program leállási feltétele. A `Megold` függvény szerkezete igen egyszerű. A `POPMERET` makróval a program elején beállíthatjuk mennyi egyedet szeretnénk létrehozni az alap populációba és a `MEGTART` makróval pedig beállíthatjuk mennyi legyen az elit egyedek száma, azaz mennyit tartsunk meg a legjobbak közül. A ciklusba belépés előtt egy kezdeti populációt kell beállítani a programnak. Ezt biztosítja nekünk a `kezdetiRandom` nevű függvény. Majd ezután léphetünk be a ciklusba, ami a jó megoldásig ismétlődik. A ciklus lényege, hogy feltöltsünk egy temp elnevezésű tömböt, ami minden egyes műveletet `POPMERET` darabszor hajt végre, majd ebből a már `POPMERET*4` elemszámú tömbből egy sorba rendezést követően kiválasztunk `MEGTART` darab elit egyedet a rendezett tömb elejéről. Ezt követően a továbbiakat úgy választjuk ki, hogy nagy eséllyel a jókat válasszunk, de adott legyen az esély a rosszabb egyedek beválogatására is, hiszen az evolúció során is

mindig maradnak fent rosszabb egyedek is. Így áll végül össze a populáció elnevezésű tömb ami ezt a POPMERET darab egyedet tartalmazza, amit később visszatöltünk a ciklusba. A 4.2 ábra is ezt a folyamatot hivatott egyszerűen szemléltetni. Jól látszik az ábrán a 4 művelet amiből a ciklus áll. Fontos azonban megjegyezni, hogy a populáció tömböt az első lefutásnál még a `kezdetiRandom`ból vesszük át teljes egészében és csak a ciklus későbbi lefutásánál töltjük fel a kiválasztott egyedekkel.



4.2. ábra. A megoldás lefutásának szemléltetése.

A feladat-specifikus részek jól elkülöníthetők a programban. Itt generálható a feladatban szereplő adatokból a neki megfelelő makrók, hogy későbbiekben egyszerűen egy számként hivatkozhatunk az egyes egyedekre. A programban létrehozott egyedek kritériumoknak való megfelelését a fitness függvény értékeli ki. A mi esetünkben ez a `hanyatSert` elnevezést kapta. Minden Zebra típusú feladvány tartalmaz kritériumokat, amik egy egységes szabályrendszer segítségével leírhatók így akár egy Excel tábla segítségével is könnyen teszt generálható belőlük. Ezeket a teszteket (kritériumokat) fűzi össze a `hanyatSert` függvény és értékeli ki az egyedek rátermettségét.

Robert E. Smith, B. A. Dike és S. A. Stegmann 1995-ös tanulmányukban elemezték a fitness függvényt és az öröklődést a genetikai algoritmusokban[12]

Lee K. Graham, Steffen Christensen illetve Franz Oppacher 2007-ben foglalkoztak az algoritmus fitness függvényének dinamikusságával, és annak a megoldásra való hatásával[5]

4.2. Egyedreprezentáció és segédfüggvények

Az algoritmusban az egyedeket szék és tulajdonságok reprezentálják, és tekintve, hogy egy tulajdonság többféle értéket is felvehet, így egy egyed több állélból áll. Az egyed felírásához szükséges függvény az alábbi módon néz ki:

```

struct gen{
    int  allel [TULAJDONSAG][SZEK];
    int  megsert;
}
  
```

N darab tulajdonságnál egy tulajdonság 0 és $n-1$ érték között vehet fel értéket, ahol n a tulajdonság számát jelenti, és az allélok pedig 0 és $m-1$ között vehetnek fel értéket, ahol m az allél számát jelenti. Ez a fajta felírás az algoritmusban alkalmazva az alábbi módon néz ki:

```
egyed . allél [4][3] == 0
```

A fenti kódrészlet alapján lehet visszakódolni, hogy az allél pontosan mit jelent, melyik tulajdonságra mutat. Az említett kódrészlet egy kezdetleges megoldást reprezentál, mely egy kisebb példánál nem, de egy nagyobb feladványnál annál inkább kellemetlenséget tud okozni, és visszafejtése időigényes. Egy olyan lehetőséget próbáltunk még ki, ami végül nem került megvalósításra, viszont úgy tette egyszerűbbé az egyedek megadását, hogy a makrókban megadott értékekről nem kellett tudnunk, azok pontosan melyik tulajdonsághoz tartoznak. A teljes kód tehát a tulajdonság, és az előző verzióban is használatos számokból tevődik össze. A BLACK kódja így például 00.

```
#define SHIRT 0
#define BLACK 00
#define BLUE 01
#define GREEN 02
#define RED 03
```

Mivel több tulajdonság is adott egy feladatban, így létrejöttek olyan kódok mint a 20; 33; 41 stb, melyek értékének visszafejtése időigényesnek bizonyult. Az interpretáció könnyítésére elkészült egy segédfüggvény, amely a kódok értékét elosztotta tízzel, így például a 41-es értékből az algoritmus visszakapta a 4 értéket, melyet a makróból kiolvassva azonosított. Ezt használva, tehát nem nincs szükség a fent bemutatott leírásra, hanem elég az alábbi módon megadni egy allét:

```
egyed . allél [ type ( Joshua ) ][0] == Joshua
```

A fent taglalt tesztíráshoz szükséges módszer a végleges változatba nem került bele, viszont ugyanezen makrókat felhasználva egy új módszer került bevezetésre. Itt tudván, hogy egyes allélok melyik tulajdonsághoz tartoznak, vagyis, hogy a Black a Color tulajdonsághoz tartozik és nem a Snackhez, egyszerűen fel lehet írni az allélokat. Ez a verzió végül egyetlen feladvány megoldásánál lett alkalmazva.

```
egyed . allél [ SHIRT ][ sz ] == BLACK
```

A fent bemutatott megoldás alkalmasnak bizonyult a továbbfejlesztésre, így egy olyan verzió készült el, ami a későbbiekben automatizálva is lett. Egy **HanyadikSzek** nevű segédfüggvény létrehozásával fejlesztettük további feladványokhoz. Ebben a megoldásban nem volt szükség a teszteknél minden egyes egyednél ellenőrizni, hogy a megadott értéket veszi-e fel, mint az előző verzióknál, elég volt csupán megadni a feladvány által előírt tulajdonságot és annak értékét, majd a tesztől függően az egyedek között fennálló összefüggést ellenőrizni.


```

int egyed1=HanyadikSzek(egyed ,DONATION,HARMINCK);
int egyed2= HanyadikSzek(egyed ,AGE,NEGYVEN);

return !(egyed1==egyed2-1);

```

Az algoritmus finomhangolásához, és az ellenőrzések leegyszerűsítéséhez elkészült egy **TULNEVEK** nevezetű kétdimenziós, sztringekből álló tömb. Ezt a tömböt használja az **egyedKiir** függvény, mely, mint neve is indikálja, az egyedek kiírásáért volt felelős. Az ellenőrzésekhez elengedhetetlen volt egy olyan függvény létrehozása, mely vizsgálta, hogy egy egyed milyen kikötéseket sért meg. Ez a függvény a **hanyatSert**, melyet az **egyedKiir** használ.

Az **egyedKiir** függvénybe beágyazva található a **hanyatSert** függvény. Az előbbi funkciója a generált egyedek közül legjobb, vagyis a legkevesebb tesztet megsértő egyed megjelenítése, a **hanyatSert** függvény által adott adatok alapján. Ezek a függvények, bár funkcionalitásukban alapvetően az algoritmusok részét képezik, mégis feladatspecifikusak a feladatonként változó tesztek és gének miatt.

Ezek használata természetesen egy olyan feladatnál, ahol fennáll annak a lehetősége, hogy nincs megoldás nem tud mindig segíteni, viszont az algoritmus fejlesztésénél elkészült egy olyan funkció is, amely ezt ellenőrzi és kiszűri. Vizsgált feladataink mindegyike rendelkezik megoldással, így a **hanyatSert** függvény segítségével ellenőrizni lehet, hogy valóban egy kikötés nem teljesül-e, vagy az algoritmus megfelelő működéséhez szükséges paraméterezés nem megfelelő.

4.3. Evolúciós mechanizmusok

A genetikus algoritmusok az evolúció mintája alapján az abban megtalálható mechanizmusokat modellezzik, melyek segítségével különböző, a feladatok megoldásához elengedhetetlen egyedek hozhatók létre. Ilyen mechanizmus a mutáció és a keresztezés, illetve itt kap jelentőséget az egyedek fitness értéke, amely gyakorlatilag a természetes kiválasztódást jelenti, így a szelekció alapját képezi. Ezek segítségével válik a populáció fejlődőképpessé.

A mutáció és a keresztezés az alappopuláció egyedein alkalmazva új egyedeket hoznak létre. Ezek futása előtt fontos, hogy rendelkezésükre álljon egy alappopuláció, melyen egy ciklus segítségével lefuthatnak. Ezt az alappopulációt a **akezdetiRandom** függvény képezi le a **POPMERET** makróban szereplő érték alapján, mely az egyedek számát adja meg. A mutáció, keresztezés, és a mutáltak keresztezése a 4.2 ábrán látható módon egyenként **POPMERET** egyedet hoz létre, melyek egy **temp** nevű tömbbe kerülnek betöltésre az alappopulációval együtt. Ez mindösszesen 400 egyedet jelent, melyek ezután a fitnessértékük alapján rendezésre kerülnek. A tömbből a **MEGTART** makró által megadott darabszámú egyed, vagyis a legjobb egyedek kerülnek kiválasztásra determinisztikus módon, így megvalósul az elitizmus. Ennek segítségével a biztosan jó gének nem kerülnek kizárásra a keresési térből.

Az egyedek fitness értékét vizsgáló függvény azok sikerességét figyeli, vagyis azt nézi,

hogy hány kikötést nem sért meg az adott egyed futás során. Tehát minél többet megsért, annál rosszabb lesz a fitness értéke, és rendezés esetén annál hátrébb kerül, végső soron pedig kihull, hiszen a futáson nem tud javítani.

Arra való tekintettel viszont, hogy vannak olyan esetek amikor az algoritmus megragadhat egy lokális szélsőértékben, fontos a valószínűségi szabályok szerinti kiválasztás. Ezt a mutálás és a keresztezés biztosítja. A genetikus algoritmusokban nagy szerepet játszik mind a MEGTART és a POPMERET makró is, hiszen ezek értéke jelentősen befolyásolhatja egy algoritmus futási idejét. Ezek értéke természetesen az adott feladványtól, feladványtípustól függ, így a minél nagyobb, vagy minél kisebb érték nem feltétlen jelent gyorsabb futásidőt.

Shane Legg és Marcus Hutter 2005-ös értekezésükben a fitness függvény hatásaként létrejövő monotonitást és annak elkerülését, továbbá az egyedek diverzitásának megtartását vizsgálták[8]

4.3.1. Véletlenszerű új egyed generálás

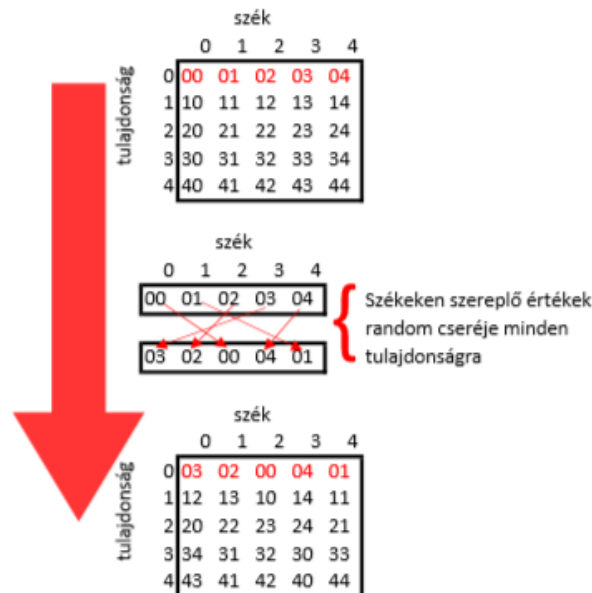
A random új egyed generálása a `kezdetiRandom` függvényben játszott csak szerepet. Itt azonban különböző stádiumokon esett át a függvény. Ahogy a kód fejlődött úgy ez is mindig módosult. A 4.3 és a 4.4 ábra bemutatja hogy a kód fejlődése során milyen működési mechanizmusokat használt a függvény. Az ábrán egy 5x5-ös méretű egyed szemlélteti a működésének lényegét.

	szék	rand()%(SZEK-sz)
	0 1 2 3 4	iteráció száma
tul. 0		generált szám
"van" tömb		elhelyezendő szám
db		
tul. 0	0 1 2 3 4	0.
"van" tömb	0 0 0 0 0	-
db		-
tul. 0	0 1 2 3 4	1.
"van" tömb	0 1 0 0 0	2
db	2 1 0	0
tul. 0	0 1 2 3 4	2.
"van" tömb	0 1 0 1 0	3
db	3 2 2 1 0	1
tul. 0	0 1 2 3 4	3.
"van" tömb	0 2 1	2
db	2 1 1 0	2

4.3. ábra. Random egyed generálása segéd tömbbel.

Az első változat még csak feltöltötte az egyedben 0-4-ig a székeket minden tulajdonság esetén. Ezt a megoldást is randomizálásnak nevezhetjük, mert tulajdonképpen ez is éppen annyira lehetséges megoldás, mintha valóban random generálnánk a számokat. A 4.3 és a 4.4 ábra egyaránt megírásra kerültek azonban a második verzió (a cserélgetéses) került végül

bevezetésre, mivel annak a működése a legáttekinthetőbb. Ez a verzió tulajdonképpen a legelső random generálási módszerből indul ki (amikor csak sorba beírjuk a számokat) és azon végez "mutációkat", azaz cserélgeti az elemeit, ezzel egy véletlenszerű sorrendet generálva.



4.4. ábra. Random egyed generálása cserélgetéssel.

Azonban a kódban szerepet nem játszó (segéd tömbös) random működése is említésre méltó. Ez a random sorba megy 0-tól 4-ig a számokon és a helyüket (szék sorszáma) keresi meg tulajdonságonként. A hely keresést a "van" tömb használatával oldja meg az algoritmus. Minden iterációban generál egy random számot, ez a darab. A darab maximális értéke iterációnként egyre kisebb lesz. A kezdetben csak nullákból álló "van" tömbhöz minden egyes iterációban hozzáfűzünk egy-egy egyest a következőképpen: a darab szám eggyel csökken, ha a van tömbben 0-t találunk és változatlan marad, ha 1-es. Így tulajdonképpen a darab azt mutatja meg, hogy hányadik üres helyre szeretnénk berakni az éppen aktuális számot. A függvény addig megy amíg minden helyre 1-es nem került. Ez egy sor. Ezt minden tulajdonságra megismételve kaphatjuk meg a random egyedet.

4.3.2. Mutálás

A mutáció az egyedek paramétereit véletlenszerűen változtatja meg, így olyan egyedeket is létre tud hozni, melyek a keresztezésnél alapvetően nem biztos, hogy létrejönnek. Így képes lehet az eljárást kimozdítani egy lokális szélsőértékből és tovább tud fejlődni a populáció. Tekintettel viszont arra, hogy így annak a lehetősége is fennáll, hogy rosszabb egyed jön létre, a mutálást kisebb gyakorisággal alkalmazzuk.

A genetikus algoritmusokban alapvetően többféle mutációt alkalmazhatunk. Ilyen például a szomszédsági mutáció a 4.5 ábrán, amikor két pontot kiválasztunk, majd az ezek

között levő pontok sorrendjén módosítunk. Ez a fajta mutáció alapvetően nagyobb feladványoknál hasznosabb, mivel nagyobb módosítást, így nagyobb mutációt is jelent. Az általunk feldolgozott feladványok nem rendelkeznek akkora allélmennyiséggel, ami ennek a mutáció-típusnak a használatát megkövetelné, így az algoritmusban egy egyszerű sorrendi mutációt alkalmaztunk. Ez egyszerűen megfogalmazva véletlenszerű módon választ ki két pontot, melyeket megcserél.

	1.szék	2.szék	3.szék	4.szék
SHIRT	black	blue	green	red
NAME	Daniel	Joshua	Nicholas	Ryan
MOVIE	action	comedy	horror	thriller
SNACK	chips	cookies	crackers	popcorn
AGE	eleven	twelve	thirteen	fourteen

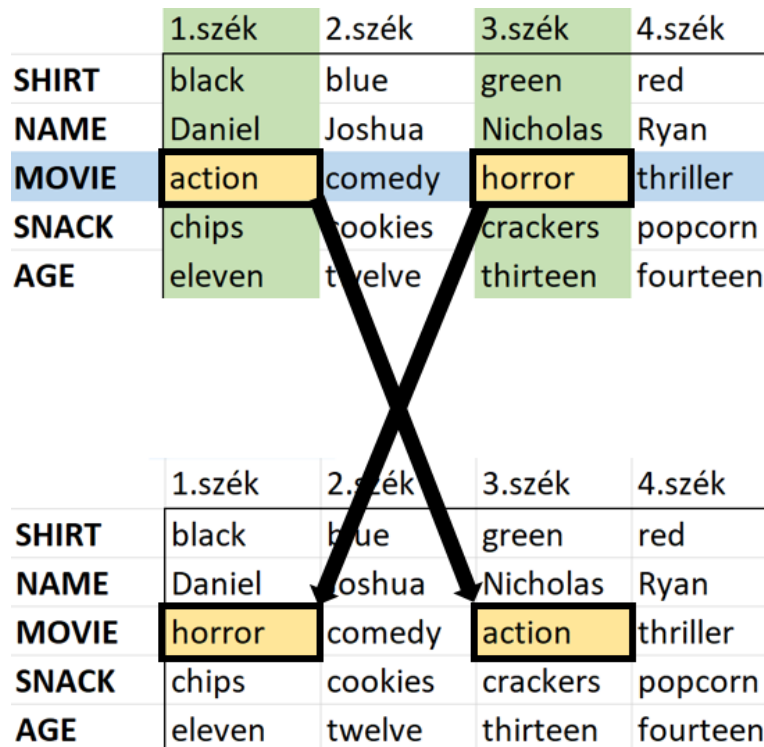
	1.szék	2.szék	3.szék	4.szék
SHIRT	black	blue	green	red
NAME	Daniel	Joshua	Nicholas	Ryan
MOVIE	horror	action	comedy	thriller
SNACK	chips	cookies	crackers	popcorn
AGE	eleven	twelve	thirteen	fourteen

4.5. ábra. A szomszédos mutáció működése egyszerűsítve

Azt figyelembe véve, hogy az egyedek alléljai kétdimenziós tömbből kerülnek kiválasztásra, a mutációt lehet végezni vagy a székek szerint, vagy a tulajdonságok szerint. Az algoritmusban a székek, vagyis oszlopok szerinti mutációt alkalmaztuk. Ebből kifolyólag egy-egy darab széket a `csere1sz` és `csere2sz` változók segítségével kiválasztásra kerül, ahol egy `while` ciklussal ellenőrizzük, hogy ugyanaz a kettő ne kerüljön kiválasztásra. A székek kiválasztása után a `cseret` változó segítségével véletlenszerűen kerül kiválasztásra a módosítandó tulajdonság. A kiválasztott székek és a tulajdonság metszetében kijelölt allélok lesznek felcserélve, mint ahogyan azt a 4.6 ábra mutatja.

Az algoritmus kezdeti státuszában a függvényben az egyed egyszer került mutálásra. Hogy a mutáció és ezzel együtt az egyedek is változatosabbak legyenek egy `hanyatmutal` nevű változó került bevezetésre. A változó 1 és 5 között vesz fel véletlenszerű módon értéket és aszerint fut le az egyed többszöri mutálását végző ciklus.

Az algoritmusban a mutáció a `Megold` függvényben kerül meghívásra és `POPMERET` alkalommal mutál egyedet, vagyis 100 db mutált egyed kerül a `TEMP` tömbbe, így negyedét kiteve annak.



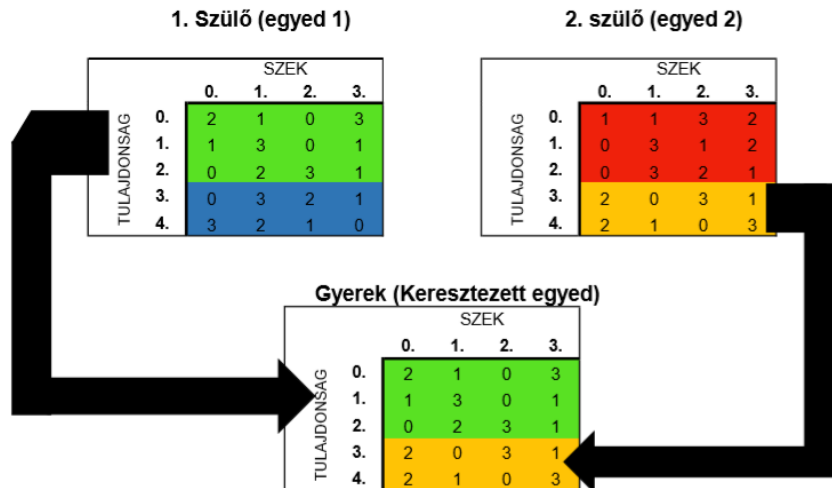
4.6. ábra. A mutáció működése

4.3.3. Keresztezés

Keresztezésből alapvetően két fajtát különböztetünk meg. Az egypontos és a kétpontos keresztezést. Az egypontos keresztezés esetében a kromoszómákat véletlenszerűen választott helyen kettévágjuk, majd a felcserélt fél-kromoszómákból újakat hozunk létre. A kétpontos keresztezés hasonlóan működik, csak ebben az esetben 2 ponton vágjuk el az allélt és a keletkezett 3 darabot fűzzük össze tetszőleges sorrendben.

A mi esetünkben az egyedek allélja egy 2 dimenziós tömb, amelyen egypontos keresztezés került alkalmazásra. Itt 2 fajta választási lehetőség fordul elő. Vagy soronként vágunk vagy a tömb sorait vágjuk ketté. Ebben az esetben az utóbbi eljárás került megírásra. A programok mindegyike egypontos keresztezést használ, ami a sorokat cseréli meg egy bizonyos ponton elvágva a tömböt. A 4.7 ábra szemlélteti a függvény működésének a lényegét. A program során 2 fajta keresztezés került kidolgozásra. Az első verzió egy fix ponton vágta el a tömböt és a 2 felét cserélte meg. A fix pont a számtani közepe a tulajdonságok számainak. Ezzel a módszerrel az a probléma, hogy a folytonos közepén való vágás nem illeszkedik bele a genetikus algoritmus randomitásába. Későbbiekben ez a módszer egy változó segítségével javítva lett azt biztosítva, hogy minden egyedpár különböző helyen legyen elválasztva.

A program írása során két fajta keresztezés került kipróbálásra. A crossover 1.0 is úgy lett kitalálva, hogy minden egyedén végezzen keresztezést, viszont az a hibája, hogy mindig 2 egymás mellett lévön végzi el. Ez viszont nem bizonyult előnyösnek, mivel a populáció tömb rendezve volt így a legjobb egyedek egymás között keresztezve igen nagy eséllyel



4.7. ábra. A keresztezés mechanizmusa

rosszabb egyedet adtak eredményképpen. Ezt igen egyszerűen ki lehetett javítani azzal, hogy a populáció tömbből véletlenszerűen választunk ki 2 egyedet és azokon végezzük a program keresztezést. Ez a módszer lehetővé teszi a programnak azt is, hogy 2 egyforma egyedet válasszon ki, ezzel fenntartva a lehetőséget, hogy egy-egy egyed keresztezés nélkül kerüljön be a temp tömbbe. Az 1.0-ás változathoz képest jelentős eltérés, hogy ez a módszer nagy eséllyel hagy olyan egyedeket amiken nem végezz keresztezést, mivel nem kerül kiválasztásra. Azonban ez a genetikus algoritmus jegyeit jobban mutatja, mivel 2 véletlenszerűen kiválasztott egyeden végezz keresztezést egy véletlenszerűen választott pontban.

```
// Crossover 1.0
temp[k]=Keresztez(populacio[POPMERET-1],populacio[0]);
k++;
for (j=0;j<POPMERET-1;j++){
    temp[k]=Keresztez(populacio[j],populacio[j+1]);
    k++;
}

// Crossover 2.0
for (j=0;j<POPMERET;j++){
    int x=rand()%POPMERET;
    int y=rand()%POPMERET;
    temp[k]=Keresztez(populacio[x],populacio[y]);
    k++;
}
```

A keresztezés POPMÉRET db alkalommal fut le, azonban a többi evolúciós algoritmushoz hasonlóan innen sem maradt ki a keresztezés a mutált egyedeken, ami szintén POPMÉRET db-szor fut le. Mivel a temp tömb $4 \cdot \text{POPMÉRET}$ méretű így a kereszte-

zett egyedek pont a temp felét teszik ki végül, így ebben az algoritmusban is a keresztezés dominál.

Soricone Robert és Neville Melvin a 2004-ben megjelent cikkükben részletesebben vizsgálták a mutáció és keresztezés hatását a genetikus algoritmusokban[13] Jun Zhang, Henry S. H. Chung illetve Jinghui Zhong 2005-ben foglalkoztak a mutáció és keresztezés hatásával áramkörök optimalizálásánál[15]

4.3.4. Megold függvény

A **Megold** függvényben gyakorlatilag a már eddig megismert függvények meghívása valósul meg.

A **Megold** függvényt az algoritmus kezdeti fázisában a **main** függvény helyettesítette, amiben a kezdeti random populáció generálása, és a temp tömb **POPMERET** értékének szerinti feltöltése történt. Utóbbinál a **POPMERET** többszörös értékét volt szükséges megadni a mutáció és keresztezés betöltendő egyedei miatt, melyek függvényhívása a mainben egy megadott ciklusérték szerint történt. A **main**ben került továbbá meghívásra a **rendezés** függvény, mely a temp tömbön végzett rendezést. A legelső változatban még nem szerepelt, viszont a genetikus algoritmusok egyik alapvető elemeként hamar bekerült a rendezésbe az elit egyedek kiválasztására szolgáló kódrészlet, ami a **MEGTART** makró értékét használja.

Az algoritmus fejlesztése során egyre több függvény került a **Megold**, eredetileg **main** függvénybe. Így került bele az **egyedKiir** függvény is, mely igen fontos szerepet játszott az algoritmus működőképességének ellenőrzésekor. A **Megold** függvényben először is létre kellett hozni egy temp tömböt a kódban előforduló **POPMERET** szerinti keresztezés és mutáció alapján. A **kezdetiRandom** függvény segítségével létrehoztunk **POPMERET** egyedet, mely a temp tömbbe bemásolásra került. Ugyanez történt a keresztezés és mutálás, illetve a már mutált egyedek keresztezésével kapott új egyedekkel is.

A temp tömb ezután rendezésre kerül a fitnessérték alapján, és a **MEGTART** makróban megadott számú egyed átmásolásra kerül a populáció tömbbe. A temp tömbből ezután a **MEGTART** makró értékével kevesebb egyedből **POPMERET-MEGTART** egyed kerül kiválasztásra a fitnessérték alapján jobbra értékelt egyedekből és töltődik be a populáció tömbbe.

A **Megold** függvény végén még egy olyan funkció került beépítésre, ami a megtalált megoldást az előtte generált populációktól jól láthatóan elválasztja, így könnyen kiértékelhető az algoritmus által adott eredmény. A **main** függvényből a fejlesztés során kikerültek a fent felsorolt függvények és a **Megold** függvénybe lettek átrakva. A **main** ennek ellenére továbbra is megmaradt, és alapesetben **Megold** függvényt hívja meg. Megtalálható benne még a **joMegoldasTeszt** függvény is, mely meghívva a jó megoldást írja ki. Ennek akkor van nagyobb jelentősége, ha az algoritmus belátható időn belül nem talál megoldást. Amennyiben a feladvány rendelkezik megoldással, az bevitelre és a **joMegoldasTeszt** függvénnyel meghívásra kerül. A függvény a tesztek helyességét a bevitt megoldáson ellenőrzi, ami alapján el lehet dönteni, hogy valóban a tesztekkel van-e a probléma, vagy az algoritmus paraméterezése nem megfelelő.

```

int main(){

    // joMegoldasTeszt();

    Megold();
    return 0;
}

```

4.4. Egyedek kiértékelése

A Zebra típusú feladványokban szereplő állítások több típusra bonthatóak. Ebből kifolyólag a tesztek felépítése nem egységes, de a cél közös: ha egy adott tulajdonsággal rendelkező személy a megszabott feltételek alapján a megfelelő pozícióban van (mi esetünkben: széken ül), akkor a függvény (**Teszt**) 0-át ad vissza értékül, különben pedig 1-et.

Az egyedek kiértékelésében fontos szerepet játszik az általunk beállított fitness függvény, a **hanyatSert** függvény.

```

int hanyatSert(struct gen egyed){
    int sert=0;
    sert+=Teszt1(egyed);
    sert+=Teszt2(egyed);
    sert+=Teszt3(egyed);
    ...
    sert+=Teszt11(egyed);
    sert+=Teszt12(egyed);
    sert+=Teszt13(egyed);
    return sert;
}

```

A cél az, hogy a függvény 0 értéket adjon vissza, ami azonban csak akkor valósul meg, ha a program eleget tesz az összes teszt feltételének. Ellenkező esetben a függvény 0-nál nagyobb értéket ad vissza, attól függően hogy hány darab tesztet sértett meg. Jelen esetben 13 teszt feltételének kell a programunknak szimultán megfelelnie.

5. fejezet

Empírikus vizsgálatok

A program futásakor egymás után születnek újabb és újabb generációk. Fontos megjegyezni, hogy adott populáción belül mindig a legjobb egyedek maradnak meg, melyek majd az aktuális generációt fogják alkotni.

```
Generacio 0:
|      purple      black      white      red      blue|
|      Diana      Jane      Lidia      Carmen      Melissa|
|      40,000 $    20,000 $    50,000 $    10,000 $    30,000 $|
|      cosmopolitan  manhattan  martini  margarita  daiquiri|
|      moonstone    sapphire    turquoise  pearl    emerald|
|      55          60          50          40          45|
16 (1)(2)(3)(6)(8)(9)(12)(13)(14)(15)(16)(17)(18)(19)(20)(21)

Generacio 1:
|      purple      black      red      white      blue|
|      Lidia      Jane      Carmen      Diana      Melissa|
|      50,000 $    20,000 $    10,000 $    30,000 $    40,000 $|
|      manhattan  cosmopolitan  daiquiri  martini  margarita|
|      sapphire    moonstone    pearl    turquoise  emerald|
|      55          45          60          40          50|
8 (1)(3)(8)(12)(13)(15)(20)(21)

Generacio 2:
|      blue      red      white      black      purple|
|      Carmen      Jane      Melissa      Diana      Lidia|
|      20,000 $    10,000 $    30,000 $    50,000 $    40,000 $|
|      martini    margarita  manhattan  cosmopolitan  daiquiri|
|      moonstone    pearl    turquoise  sapphire    emerald|
|      40          45          50          55          60|
6 (1)(2)(3)(11)(12)(14)
```

5.1. ábra. A program a futás kezdetekor

Az 5.2 ábra demonstrálja, hogyan is néz ki egy generáció: a feladványban szereplő tulajdonságok rendezetten fel vannak sorolva. Ez mindig úgy alakul, hogy minél kevesebb feltételt sértsen meg, illetve egyri inkább közeledjen a jó megoldás felé. A tulajdonságok alatt levő számsorozatot azt szimbolizálja, hogy a program futás közben mennyi tesztet sért meg, a zárójelben levők pedig hogy konkrétan melyikeket. A cél akkor teljesül be, ha ez a szám 0-val egyenlő.

```

Generacio 1:
| purple      black      red      white      blue |
| Lidia      Jane      Carmen    Diana      Melissa |
| 50,000 $    20,000 $    10,000 $    30,000 $    40,000 $ |
| manhattan  cosmopolitan  daiquiri    martini     margarita |
| sapphire   moonstone    pearl       turquoise   emerald |
| 55         45         60         40         50 |
| 8 (1)(3)(8)(12)(13)(15)(20)(21)

```

5.2. ábra. Példa egy generációra

A futási sebességet több tényező is befolyásolja:

- feladat nehézsége, még precízebben a feladat mérete, vagyis a tulajdonságok és a betölthető pozíciók száma,
- valamint a populáció és a megtartott egyedek száma

Az 5.3 ábra mutatja be, hogy miként változik meg a futási sebesség, ha esetenként megváltoztatjuk a populáció és a megtartott egyedek számát. Emelett megfigyelhető, hogy hány darab generáció volt szükséges a jó megoldás eljutásához, és hogy összesen mennyi egyed született meg. Az adatok növekvő sorrend szerinti rendezést követnek a program lefutási idejének alapján.

Populáció mérete (darab)	Megtartott egyedek száma	Idő (másodperc)	Generáció (darab)	Egyed (darab)
500	10	0.379	17	34 000
300	10	0.535	42	50 400
200	20	0.818	96	76 800
1 000	10	1.043	19	76 000
1 000	200	1.115	20	80 000
1 000	50	1.349	26	104 000
1 000	100	2.779	58	232 000

5.3. ábra. Futási eredmények különböző paraméterek esetén

6. fejezet

Kód automatikus generálása

Az első program megírása után már felmerült az igény, hogy legyen valami ami legenerál bármilyen feladathoz egy kódot. Ez könnyedén megoldható, hiszen a program genetikus része minden ilyen fajta problémához egységesen kell, hogy működjön, így a generált kódokban a genetikus részt nem is kell módosítani csak a feladathoz tartozó adatokat kell átírni. Mivel a program 3 .c kiterjesztésű állományból áll, így a 2 genetikus vonatkozást tartalmazó program mindig fix maradhat és csak egy 3. a tesztet és a struktúrát tartalmazó .c kiterjesztésű fájlt kell cserélgetni, valamint ezt kell a main-be egyszerűen include művelettel beágyazni. A kód generálása azért egy hasznos dolog, mert így a genetikus algoritmusok bármelyik felhasználóhoz eljuthatnak, hiszen nem kell programozási ismeret hozzá. Csak ki kell tölteni a táblázatot, kiexportálni egy c fájlba, majd lefuttatni a main.c-t, természetesen a main.c-be nem szabad kifelejteni a kapott fájl beágyazását.

TULAJDONSAG	5		1. szék	2. szék	3. szék	4. szék	
SZEK	4	SHIRT	black	blue	green	red	
		NAME	Daniel	Joshua	Nicholas	Ryan	
POPMERET	100	MOVIE	action	comedy	horror	thriller	
MEGTART	10	SNACK	chips	cookies	crackers	popcorn	
		AGE	eleven	twelve	thirteen	fourteen	

6.1. ábra. Egyedek megadása

A kódgenerálás kivitelezésére a Microsoft Office Excel program tűnt a leginkább optimálisnak, hiszen ezzel a programmal a szöveg összefűzési műveletek és az input bevitele viszonylag egyszerűen megoldhatóak. A 6.1 és a 6.2 ábrák a legelső program Excelbe történő bevitelét hivatottak szemléltetni. Azonban a táblázat úgy van megtervezve, hogy akár egy 20x20-as egyedmérettel és akár 250 teszttel is működőképes legyen. Maga a munkafüzet igen felhasználó barát. 4 lapból áll, amiből 2 lapvédelem alatt áll. Az első lap az Adatok. Ide egészen egyszerűen csak be kell írni a tulajdonságokat és a különböző székeken lévő értékeket. Még csak azzal sem kell foglalkozni, hogy a székek vagy a tulajdonságok számát beírjuk, mivel a bevitt adatokhoz képest dinamikusan változnak ezek a számok is. Egyedül a POPMÉRÉTER és a MEGTART értékeit lehet módosítani a makrók közül. Annyi kritérium van azonban a lappal kapcsolatban, hogy a szám adatokat szöveggel kell bevinni a program helyes futása érdekében (a program legenerálódik így is, csak a makrók nem lehetnek

számok, így nem tudjuk futtatni majd).

Teszt	one of the ends	between	exactly to the	somewhere to the	position	likes	egyed1 value	egyed2 value	egyed3 value	position
Joshua is in one of the ends.	X						JOSHUA			
The boy wearing the Black shirt is somewhere to the left of the youngest			X				BLACK	ELEVEN		left
Joshua likes Horror movies.					X		JOSHUA	HORROR		
The 14 years old boy is at 3. position.				X			FOURTEEN			3
The boy wearing the Red shirt is somewhere between the 13 years old boy		X					RED	THIRTEEN	ACTION	
Daniel likes Thriller movies.					X		DANIEL	HORROR		
The boy who is going to eat Cookies is in one of the ends.	X						COOKIES			
The boy wearing the Black shirt is exactly to the left of the one who likes			X				BLACK	THRILLER		left
The boy who is going to eat Crackers is exactly to the right of the boy who			X				CRACKERS	COMEDY		right
The boy wearing the Red shirt is somewhere between the boy who is going		X					RED	POPCORN	NICHOLAS	
In one of the ends is the boy who likes Thriller movies.	X						THRILLER			
Nicholas is somewhere between Joshua and Daniel in that order.		X					NICHOLAS	JOSHUA	DANIEL	
At 1. position is the boy wearing the Green shirt.				X			GREEN			1

6.2. ábra. Tesztek megadása

A második lap a Tesztek nevet kapta. Ide maximum 250 darab tesztet lehet megadni, ami egy 20x20-as egyedhez elegendő is. A tesztek szöveges megadása nem kötelező. Anélkül is le lehet generálni a kódot. Ha valaki meg szeretné adni a teszt szövegét is annak nem kell X-et raknia a teszt típusok megfelelő négyzetébe. Amennyiben valaki eltekint a teszt szöveges megadásától annak be kell X-elnie az egyik oszlopot különben nem generálódik le a teszt. Ezeken felül a teszt típusától függően kötelező megadni 1, 2 vagy 3 egyed értékét. A tulajdonság fajtáját nem kell megnevezni, mivel azt egy függvény a bevitt érték alapján vissza tudja keresni a későbbiekhez. A 6.2 ábrán láthatjuk ennek a felületnek a megjelenését. Azonban a lapon megjelenik még egy position elnevezésű oszlop is, amit csak a tőle ballra/jobbra, illetve konkrét pozíciót leíró függvények esetén kell használni.

A harmadik lapot (Kódok) már lapvédelem védi. Ez a lap kizárólag segéd lapként szerepel a munkafüzetben. Azért van lapvédelem alatt, hogy a felhasználó bele kattintani se tudjon, mivel neki azzal semmi dolga nincs. Az kizárólag a tesztek kóddá való átírására szolgál. Az egész program végül az utolsó Program lapon áll össze. Egy szűrő segítségével a lapon csak a felhasználó szempontjából értékes adatok láthatóak. A program dinamikus változását úgy lehetett csak megoldani, hogy sok üres cellát iktasson közbe. Ezek azonban egy egyszerű szűrővel mind eltüntethetőek, szóval a felhasználók csak a programot látják. Ez a leggenerált kód. Ezzel nincs más dolog már, mint kijelölni az egész látható területet (a felesleg el van rejtve) és beilleszteni egy .c kiterjesztésű fájlba. (Ami fontos a kimásolás előtt, hogy a lap tartalma frissítve legyen. Ez könnyedén elérhető, mert csak rá kell kattintani a szűrő ikonjára és minden módosítás nélkül le kell okézni.) Innen már a main.c-be amikor a felhasználó beírja include paranccsal a létrehozott .c kiterjesztésű fájl nevét akkor észlelni fogja a program és le is fut (ha nem nincs elírás a munkafüzetben).

7. fejezet

Összefoglalás

TODO: Mit csináltunk roviden

Irodalomjegyzék

- [1] Adrian Brezulianu, Monica Fira, and Lucian Fira. A genetic algorithm approach for a constrained employee scheduling problem as applied to employees at mall type shops. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, ICHIT '09, pages 497–501, New York, NY, USA, 2009. ACM.
- [2] Lewis Carroll. *The game of logic*. Macmillan, 1886.
- [3] Deon Garrett, Joseph Vannucci, Rodrigo Silva, Dipankar Dasgupta, and James Simien. Genetic algorithms for the sailor assignment problem. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1921–1928, New York, NY, USA, 2005. ACM.
- [4] Erik D. Goodman. Introduction to genetic algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 671–692, New York, NY, USA, 2012. ACM.
- [5] Lee K. Graham, Steffen Christensen, and Franz Oppacher. A simple genetic algorithm for reducible complexity. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1514–1514, New York, NY, USA, 2007. ACM.
- [6] T W Jerardi. Puzzles, prolog and logic. *SIGPLAN Not.*, 22(4):63–69, April 1987.
- [7] Sami Khuri, Tim Walters, and Yanti Sugono. A grouping genetic algorithm for coloring the edges of graphs. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 1*, SAC '00, pages 422–427, New York, NY, USA, 2000. ACM.
- [8] Shane Legg and Marcus Hutter. Fitness uniform deletion: A simple way to preserve diversity. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1271–1278, New York, NY, USA, 2005. ACM.
- [9] Juan Julián Merelo-Guervós, Pedro Castillo, Antonio M. Mora García, and Anna I. Esparcia-Alcázar. Improving evolutionary solutions to the game of mastermind using an entropy-based scoring method. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 829–836, New York, NY, USA, 2013. ACM.

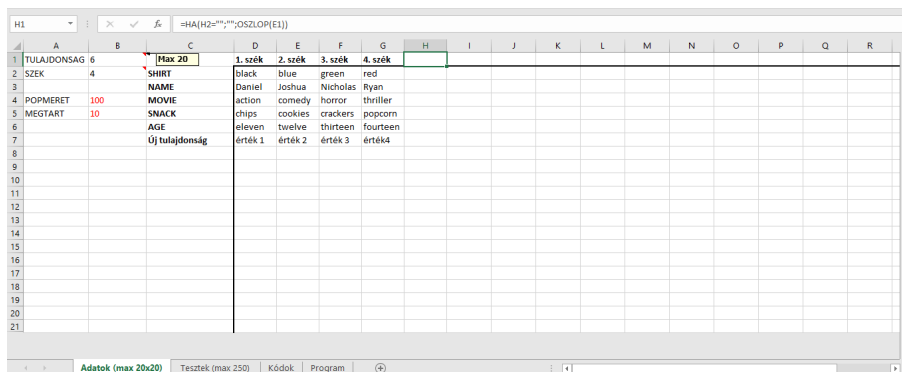
- [10] Rushil Raghavjee and Nelishia Pillay. An informed genetic algorithm for the high school timetabling problem. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 408–412, New York, NY, USA, 2010. ACM.
- [11] Soroor Salavati, Sahar Hajjarzadeh, and Masoud Mazloom. An optimized method for solving zebra puzzle. *Second International Conference on Computer and Electrical Engineering*, pages 448–451, 2009.
- [12] Robert E. Smith, B. A. Dike, and S. A. Stegmann. Fitness inheritance in genetic algorithms. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, SAC '95, pages 345–350, New York, NY, USA, 1995. ACM.
- [13] Robert Soricone and Melvin Neville. Comparative analysis of genetic algorithm implementations. *Ada Lett.*, XXIV(4):35–38, November 2004.
- [14] Gizem Sungu and Betul Boz. An evolutionary algorithm for weighted graph coloring problem. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1233–1236, New York, NY, USA, 2015. ACM.
- [15] Jun Zhang, Henry S. H. Chung, and Jinghui Zhong. Adaptive crossover and mutation in genetic algorithms based on clustering technique. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1577–1578, New York, NY, USA, 2005. ACM.

A. függelék

Kódgeneráló táblázat

Adatok megadása

Az A.1 ábra szemlélteti a megadás formátumát. Az egyedeket egy 20x20-as táblázat segítségével adhatunk meg. Ez ahhoz fontos, hogy a makrók később automatikusan generálódjanak és ne kelljen később hozzányúlni. A székek számozásával valamint a TULAJDONSAG és SZEK makrók beállításával nem kell foglalkozni, hiszen azok kitöltése automatikusan történik. A módosítható makróértékek piros betűszínnel vannak jelezve. Tehát a POPMERET és a MEGTART bármikor módosítható.



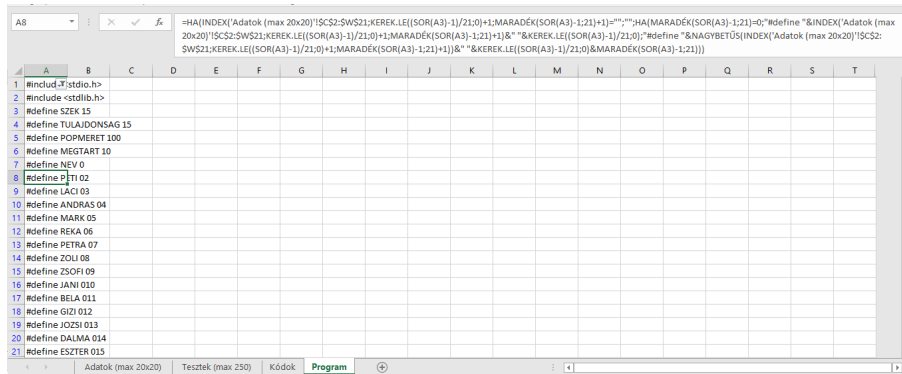
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	TULAJDONSAG	6	[max 20]	1. szék	2. szék	3. szék	4. szék											
2	SZEK	4	SHIRT	black	blue	green	red											
3			NAME	Daniel	Joshua	Nicholas	Ryan											
4	POPMERET	100	MOVIE	action	comedy	horror	thriller											
5	MEGTART	10	SNACK	chips	cookies	crackers	popcorn											
6			AGE	eleven	twelve	thirteen	fourteen											
7			Új tulajdonság	érték 1	érték 2	érték 3	érték 4											
8																		
9																		
10																		
11																		
12																		
13																		
14																		
15																		
16																		
17																		
18																		
19																		
20																		
21																		

A.1. ábra. Egyedek megadása

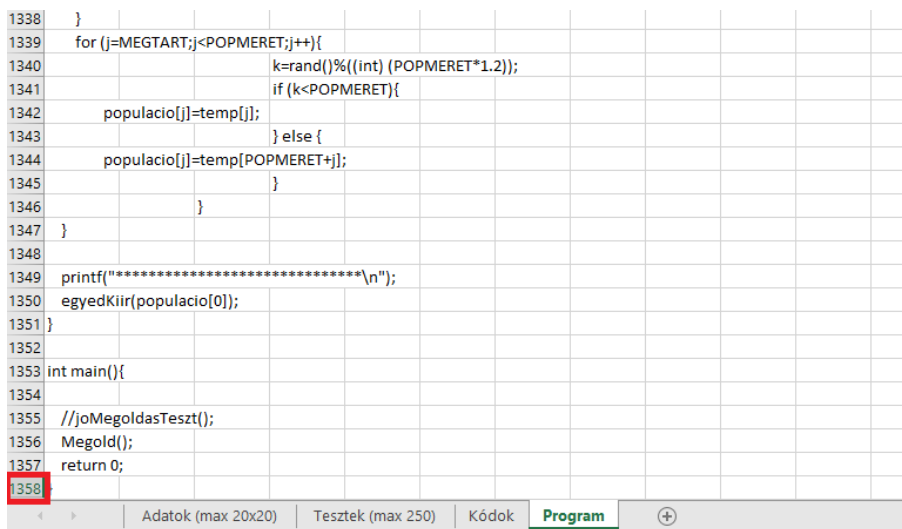
Tesztek kitöltése

Az A.2 ábra szemlélteti a tesztek megadásának menetét. A tesztek megadása során a szöveges megadás elhanyagolható, azonban annak segítségével a program automatikusan ki tudja tölteni a tesztek fajtájára vonatkozó részt. A kényelmesség érdekében csak az értékeket kell megadni a tesztek során, a tulajdonság típusát már a program is vissza tudja keresni így az elhagyható.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------



A.4. ábra. A kód generálásának automatikussága



A.5. ábra. Nagy példák is kezelhetőek