

# Zebra típusú logikai rejtvények megoldása evolúciós algoritmussal

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

Témavezető: Dr. Hegyháti Máté

2018. március 1.

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>2</b>
<b>2. Zebra rejtvények</b>	<b>4</b>
2.1. Történet és szerkezet . . . . .	4
2.2. Megoldhatóság, egyértelműség . . . . .	5
2.3. Megoldó módszerek . . . . .	6
<b>3. Evolúciós algoritmusok</b>	<b>7</b>
<b>4. Evolúciós algoritmus Zebra rejtvények megfejtésére</b>	<b>10</b>
4.1. Kódszerkezet . . . . .	10
4.2. Egyedreprezentáció és segédfüggvények . . . . .	12
4.3. Evolúciós mechanizmusok . . . . .	13
4.3.1. Random új egyed generálás . . . . .	13
4.3.2. Mutálás . . . . .	14
4.3.3. Keresztezés . . . . .	16
4.3.4. Megold függvény . . . . .	17
4.4. Egyedek kiértékelése . . . . .	17
<b>5. Tesztek</b>	<b>19</b>
<b>6. Kód automatikus generálása</b>	<b>20</b>
<b>7. Összefoglalás</b>	<b>22</b>
<b>Hivatkozások</b>	<b>23</b>
<b>A. Mintafejezet</b>	<b>24</b>
A.1. Kep betöltése . . . . .	24
A.2. Tablázatok . . . . .	24
A.3. Forraskodok beemelese . . . . .	25
A.4. Hivatkozások . . . . .	25

# 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 Dodgson angol író hozott létre. A logika játéka (The Game of Logic) című 1886-ban megjelent könyvében egy olyan játékot mutatott be, melyben kijelentések alapján kellett megerősíteni a következtetést. Az ilyesféle feladványokat nevezzük szillogizmusnak. Ezenkívül léteznek még teljesen non-verbális rejtvények:

- Sudoku, melyben megadott szabályok szerint számjegyeket kell elhelyezni egy táblázatban,
- Nonogram vagy grafilogika, mely úgy foglalja magába a dedukció használatát, hogy egy rács fekete-fehér négyzetekkel történő helyes kitöltése egy képet alkot,
- Logikai labirintus, melyben a dedukció felhasználásával a labirintus szabályai kell kitalálni.

Továbbá egy igen közkedvelt típus 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:

- Visszalépéses keresés, amikor egy fastruktúrát elképzelve a gyöktől kiindulva egy csúcsot keresünk
- Korlátozási programozás, ahol a változók közötti kapcsolatokat korlátok formájában adjuk meg
- A feladvány Prolog nyelven való megírása, ahol egy megadott logikai formuláról képes eldönteni, hogy logikai következménye-e formulák egy adott halmazának.

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. Munkánk során egy genetikai algoritmus került kidolgozásra, és tesztelésre több példán. A genetikai algoritmus optimalizációs eljárás, ahol a lehetséges hipotéziseket az egyedek a hipotézis tér valamely részét pedig a populációk reprezentálják. A számítástechnikában és a mesterséges intelligenciában alkalmazott heurisztikus keresési technika az evolúciós biológiában ihletett technikák alkalmazásával: mutáció, szelekció, reprodukció és rekombináció.

Fontosabb felhasználási területek:

- autóipari és mérnöki tervezés
- robotika
- biomimetikus találmányok

- útvonalválasztás utazás vagy szállítmányozás során
- titkosítás és kódtörés
- értékesítés

A genetikus algoritmusok minden egyes lépésben heurisztikák segítségével módosítják, vagy bővítik az aktuális populációt. Minden egyes lépésben képesek adni egy közelítést a megoldásra, amelynek a pontossága az idő teltével nő. A futási idő csökkentése érdekében az algoritmus különböző paraméterei finomhangolásra kerültek nagy számú tesztet futtatásának tapasztalati 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 elsősorban a Zebra rejtvények részletesebb bemutatásával kezdjük. Itt szó esik a történetéről, szerkezetéről, hogy miből állnak a fejtörők, illetve néhány részlet. Említésre kerül még a megoldhatóság eltérő lehetőségei, valamint az irodalomban megtalálható megoldó módszerek. Ezt követően az evolúciós algoritmusokat kívánjuk szemléltetni. Bemutatjuk történetüket és felépítésüket. Folytatásként betekintést nyújtunk arról, miképp lehet felhasználni az evolúciós algoritmusokat Zebra rejtvények megfejtésére. Ismertetjük a kódszerkezetet, a különféle egyedrepresentációkat és segédfüggvényeket, bemutatjuk az evolúciós mechanizmusokat, majd végül az egyedek kiértékeléséről esik szó. Ezt követően szemléltetjük a teszteket, azok megoldásait és futtatási eredményeit, valamint demonstráljuk kódunk automatikus generálásának elgondolását és kivitelezését. Végezetül összefoglaljuk, mit csináltunk célunk elérése érdekében.

## 2. fejezet

# Zebra rejtvények

**TODO: Egy bevezető mondat, + hogy melyik alfejezetben mi lesz**

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

### 2.1. Történet és szerkezet






**TODO: Miért zebra, mikből áll a rejtvény, Einstein példájából részlet akar**

A leghíresebb zebra típusú rejtvény Albert Einstein nevéhez fűződik, melyről a logikai feladványoknak ez a típusa a nevét is kapta, hiszen az Einstein-példa egyik változatában a zebra tulajdonosának kiléte a fő kérdés. Úgy tartják, hogy Einstein ezeknek a feladványoknak 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, valamint az általa kedvelt ital, cigaretta márkája és a háziállat, amit tart. Azonban egyik ház sajátosságait sem ismerjük kezdetben. Ekkor jönnek képbe az állítások. Einstein 15 állítást 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.

1. A brit a piros házban lakik.
2. A svéd kutyákat tart háziállatként.
3. A dán teát iszik.
4. A zöld ház közvetlenül a fehér ház bal oldalán van.
5. A zöld ház tulajdonosa kávét iszik.
6. Az a tulajdonos, aki Pall Mall sziv, madarakat tart.
7. A sárga ház tulajdonosa Dunhill sziv.
8. A középső ház tulajdonosa tejet iszik.
9. A norvég az első házban lakik.
10. A Blends márkát szívó tulajdonos a macskatartó mellett lakik.
11. A lovakat tartó tulajdonos a Dunhill szivó tulajdonos mellett lakik.
12. A Bluemaster szivó tulajdonos sör iszik.
13. A német Prince márkájú cigarettát szív.
14. A norvég a kék ház mellett lakik.
15. A Blends márkájú cigarettát szívó a vizet ivó tulajdonos mellett lakik.

A kérdés: **Ki tart halakat?**

					
Származás					
Szín					
Cigaretta márkája					
Italféle					
Állatok					

2.1. ábra. Az Einstein-példa egyik változata

## 2.2. Megoldhatóság, egyértelműség

TODO: Pici példakon bemutatni, hogy ha rosszak a szabályok, akkor lehet nincs megoldás, vagy ha keves a szabály, akkor lehet több megoldás is van. Egy nagyon apro (3 szek, 2 tulajdonsag mondjuk) pelda kitalalasa es megoldasa par lepesben.

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.

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

**TODO: Irodalomban található módszerek, 2 mondat róluk, hivatkozások**

A zebra rejtvények megoldására több különböző módszer is született. Az egyik módszer a feladvány megoldása Prolog nyelven kényszerfeltétel kielégítési modul segítségével. 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.

### 3. fejezet

## Evolúciós algoritmusok

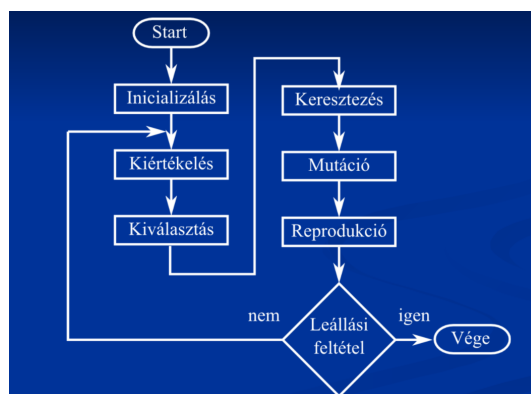
TODO: Történetük, hivatkozások TODO: Általános felépítésük TODO: Akar pár szót arról, mi mindenre alkalmazták őket, hivatkozások

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.

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.



3.1. ábra. Az evolúciós algoritmus általános felépítése

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



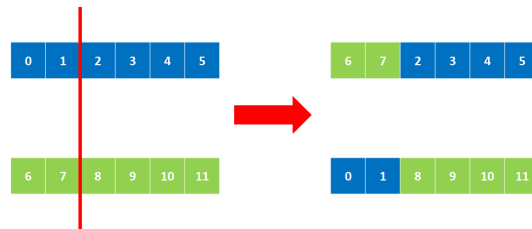
viszonylag könnyen lehet javítani.

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.

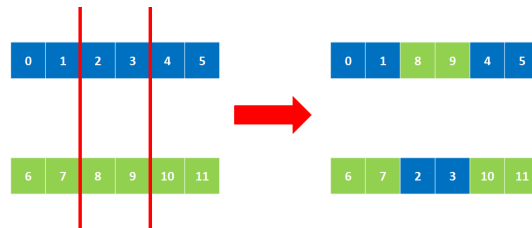
Az aránytalanságok elkerülése érdekében fontos szerepe van a rangsorolásnak, mivel nem optimális az, ha például néhány „túl jó” egyed hoz létre sok utódot, ugyanis hasonló egyedek keresztezésével nem jutunk előre.

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 fő fajtája az egy- illetve kétpontos keresztezés. Az előbbi esetben 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



3.3. ábra. A kétpontos keresztezés

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

A következő lépés a mutáció, melynek lényege, hogy bizonyos gének értékeit megváltoztatjuk. 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 a számítógépes környezetben egyre nagyobb népszerűségnek örvendenek, de az üzleti szférában és az autógyártásban is alkalmazzák őket. Egyes cégek a beszerzési és értékesítésre vonatkozó döntések meghozatalakor is támaszkodnak evolúciós algoritmusokra, de az autók festésének koordinációja is elvégezhető a segítségükkel. Genetikus algoritmusokat használnak a mezőgazdasági termelés optimalizálásában is.

## 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. Erre legalkalmasabbnak az evolúciós algoritmusok feleltek meg.

A 4.1 fejezetben a kódszerkezet kerül bemutatásra, hogy miképp van szervezve a kód, illetve a benne megtalálható függvényeknek mi a feladata.

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 `Megold` függvény.

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 2 .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 tesztekkel van-e a gond és ha igen akkor melyikkel.

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

```

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

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

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

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

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

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

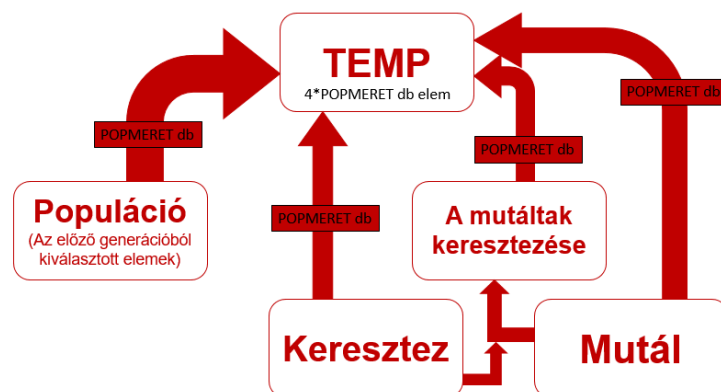
*****
green    red     black  blue
Joshua   Ryan   Nicholas Daniel
comedy   comedy action thriller
popcorn  popcorn chips  crackers
0

Press any key to continue . . .

```

4.1. ábra. A program lefutása

**hanyatSertnek** nevezzük. A program célja az, hogy a fitness 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 tartunk 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 válasszuk, 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 tesztek (kritériumokat) fűzi össze a `hanyatSert` függvény és értékeli ki az egyedek rátermettségét.

## 4.2. Egyedreprezentáció és segédfüggvények

Az algoritmusban az egyedeket székek és tulajdonságok reprezentálják, és tekintve, hogy egy tulajdonság többféle értéket is felvehet, így egy egyed több allélból 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 jelentette, és az allélok pedig 0 és m-1 között vehettek fel értéket, ahol az m az allél számát jelentette. Ez a fajta felírás az algoritmusban alkalmazva az alábbi módon nézett ki:

```
egyed.allel[4][3]==0
```

A fenti kódrészlet alapján lehetett 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 nehézkes. 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 kiolvastva azonosított. Ezt használva, tehát nem volt szükség a fent bemutatott leírásra, hanem elég volt az alábbi módon megadni egy allélt:

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

Egy következő megoldás, melyet a legkönnyebb feladvány megoldásánál is használtunk, ugyanezen makrók igénybevétele. Itt tudván, hogy egyes allélok melyik tulajdonsághoz tartoznak, vagyis, hogy a Black a Color tulajdonsághoz tartozik és nem a Snackhez, egyszerűen fel lehetett írni az allélokat.

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

Ezt a megoldást egy `HanyadikSzek` nevű segédfüggvény létrehozásával egyszerűsítettük a nehezebb feladványokhoz. Ebben a megoldásban nem volt szükség a teszteknél minden egyes egyednél ellenőrizni, hogy a megadott értéket veszi-e fel, mint az előző verzióknál. Itt elég volt megadni a feladvány által megadott 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 használta 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** volt, melyet az **egyedKiir** használt fel. Ennek használata természetesen egy olyan feladatnál, ahol fennáll annak a lehetősége, hogy nincs megoldás nem tud mindig segíteni, viszont az algoritmus fejlesztésénél létrejött olyan funkció is, amely ezt ellenőrzi és kiszűri. Vizsgált feladataink mindegyike rendelkezett megoldással, így a **hanyatSert** függvény segítségével ellenőrizni lehetett, 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 fejlődőképessé.

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 **kezdetiRandom** függvény képezi le a **POPMERET** makróban szereplő a 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.

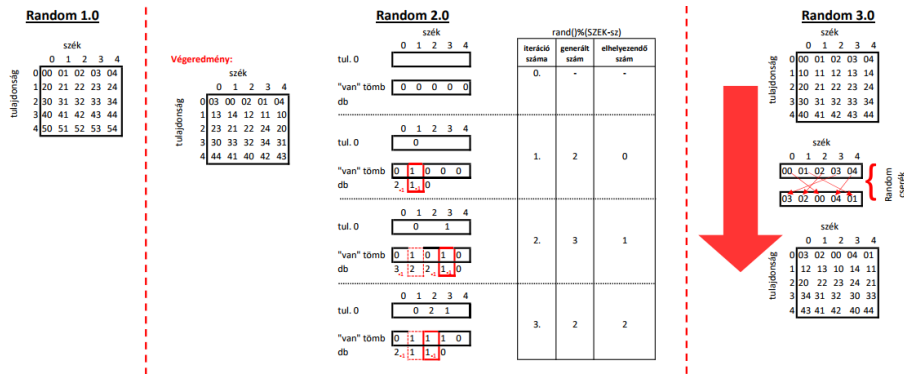
Az egyedek fitness értékét vizsgáló függvény azok sikerességét figyeli, vagyis azt nézi, hogy hány kikötést nem sért meg az adott egyed futás során. Tehát minél többet megsért, annál rosszabb lesz a fitness értéke, és rendezés esetén annál hátrébb kerül, végső soron pedig kihull.

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.

#### 4.3.1. Random 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 ábra bemutatja hogy a kód fejlődése során milyen működési mechanizmusokat használt a függvény. Az ábrán egy 5x5-ös méretű egyed szemlélteti a működésének lényegét.

A 4.3 ábrán 1.0-val jelölt, azaz az első változat még csak feltöltötte az egyedben 0-4-ig a székeket minden tulajdonság esetén. Tulajdonképpen ez is éppen annyira lehetséges megoldás, mintha valóban random gene-



4.3. ábra. Random egyed generálási módszerek.

rálánk. A 2.0-ás és a 3.0-ás ábrák egyaránt megírásra kerültek azonban a 3.0-ás került végül bevezetésre, mivel annak a működése a legáttekinthetőbb. A 3.0 tulajdonképpen a Random 1.0-ból indul ki és azon végez "mutációkat", azaz cserélgeti az elemeit, ezzel egy véletlenszerű sorrendet generálva. Azonban a Random 2.0 működése is említésre méltó. Ez a random sorba megy 0-tól 4-ig a számokon és a helyüket (szék sorszáma) keresi meg tulajdonságoként. A hely keresést a "van" tömb használatával oldja meg az algoritmus. Minden iterációban generál egy random számot, ez a darab. A darab maximális értéke iterációként 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.

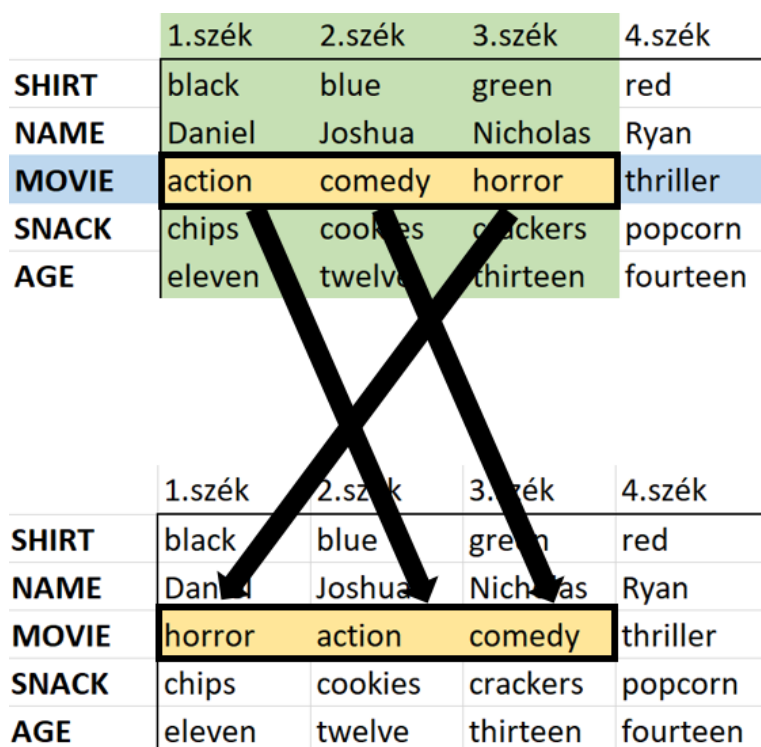
### 4.3.2. Mutálás

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.

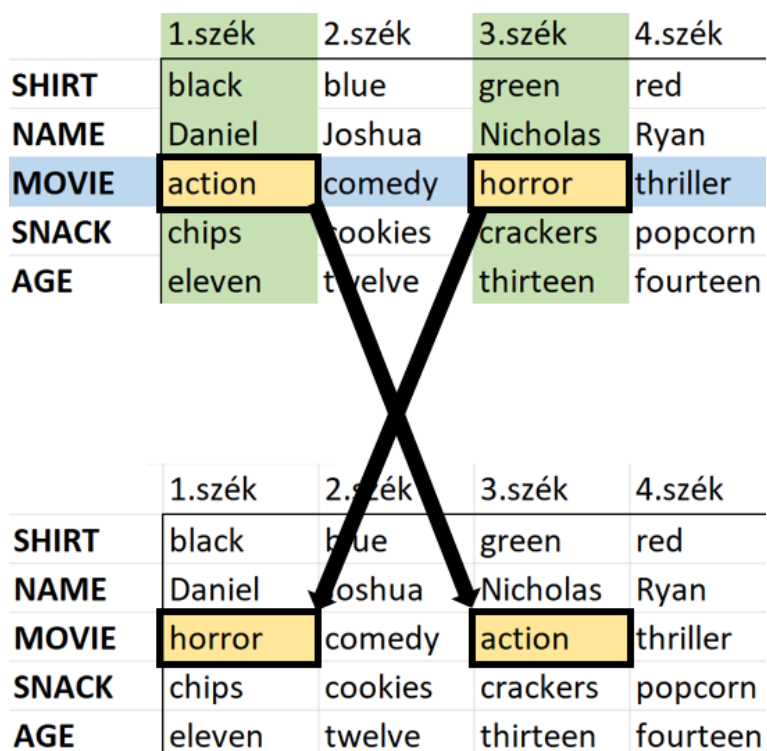
A genetikus algoritmusokban alapvetően többféle mutációt alkalmazhatunk. Ilyen például a szomszédsági mutáció A 4.4 ábrán, amikor két pontot kiválasztunk, majd az ezek között levő pontok sorrendjén módosítunk. Ez a fajta mutáció alapvetően nagyobb feladványoknál hasznosabb, mivel nagyobb módosítást, így nagyobb mutációt is jelent. Az általunk feldolgozott feladványok nem rendelkeznek akkora állélmennyiséggel, ami ennek a mutációtípusnak a használatát megkövetelné, így az algoritmusban egy egyszerű sorrendi mutációt alkalmaztunk. Ez egyszerűen megfogalmazva véletlenszerű módon választ ki két pontot, melyeket megcserél.

Azt figyelembe véve, hogy az egyedek álléljai 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 szék 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 4.5 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.



4.4. ábra. A szomszédos mutáció működése egyszerűsítve



4.5. ábra. A mutáció működése



Az algoritmusban a mutáció a Megold függvényben kerül meghívásra POPMERET db alkalommal, vagyis 100 db mutált egyed kerül a TEMP tömbbe, így negyedét kitevé annak. Ugyanezen a 100 egyeden végez a keresztezés függvény keresztezést, újabb 100 egyedet létrehozva, melyek ugyancsak a temp tömbbe kerülnek.

### 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 alléja 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 4.6 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 genetikai algoritmus randomitásába. Későbbiekben ez a módszer egy változó segítségével javítva lett azt biztosítva, hogy minden egyedpár különböző helyen legyen elválasztva.



4.6. ábra. A keresztezés mechanizmusa

A program írása során két fajta keresztezés került kipróbálásra. A crossover 1.0 is úgy lett kitalálva, hogy minden egyeden 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 rosszabb egyedet adtak eredményképpen. Ezt igen egyszerűen ki lehetett javítani azzal, hogy a populáció tömbből véletlenszerűen választunk ki 2 egyedet és azokon végez 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 végez keresztezést, mivel nem kerül kiválasztásra. Azonban ez a genetikai algoritmus jegyeit jobban mutatja, mivel 2 véletlenszerűen kiválasztott egyeden végez 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 keresztezett egyedek pont a temp felét teszik ki végül, így ebben az algoritmusban is a keresztezés dominál.

#### 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 **mainben** 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 100 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 darab 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 a **Megold** függvényt hívja meg.

### 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

# Tesztek

A program futtatását követően figyelemmel lehet követni, hogy hány generáció született meg ahhoz ahhoz, hogy beteljesedjen az összes teszt feltétele.

```
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      Joshua    Nicholas    Daniel|
|  horror      comedy     action      thriller|
|  cookies      popcorn    crackers    chips|
|      13      11      14      12|
|4 (1)(2)(3)(10)|
```

5.1. ábra. A program futás alatt

A precizitás az idő függvényében változik: az idő előrehaladtával egyre inkább közelít a jó megoldás felé, egészen addig, amíg a hanyatSert függvény 0-át nem ad vissza értékül.

```
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|
```

5.2. ábra. Végeredmény

Figyelembe véve, hogy egy egyszerű példáról van szó, amely nem tartalmaz sok tulajdonságot és személyt, a program egy szempillantás alatt megoldja a feladatot. Átlagosan 8 generáció születik meg a végeredményhez érve, valamint átlagosan mindössze 0.005 (!) másodperc alatt fut le hibátlanul.

## 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 POPMÉRÉRET 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).

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

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

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 szűréssel mind eltüntethetőek, szóval a felhasználók csak a programot látják. Ez a legenerált 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

# Hivatkozások



## A. függelék

# Mintafejezet

### A.1. Kep betoltese

Kep betoltese a `\Abra` makroval az alabbiak szerint. Az elso parameter a fajl neve, ebbol lesz egy `fig_fajlnev` cimke, amit ref-ekhez lehet hasznalni. MASodik parameter a kepalairas, a harmadik a meret.

```
\Aref{fig_zebra} abran bla bla.  
\Abra{zebra}{Talalo kepalairas}{width=6cm}
```

Az A.1 abran bla bla.



A.1. ábra. Talalo kepalairas

### A.2. Tablazarok

Tablazarot a `\Tablazar` makroval lehet csinálni az alabbiak szerint. Az elso parameter a cimke, ebbol lesz egy `tab_cimke` cimke, amit ref-ekhez lehet hasznalni. MASodik parameter a tablazar cime, a harmadik az oszlopok szerkezete (lasd tabular tutorial), az utolso maga a tablazar tartalma.

```
Ahogya az \aref{tab_cimke} tablazarban lathato, bla bla bla.  
\Tablazar{cimke}{Tablazar cime}{r||cc|l}  
{  
  Elso sor elso cella jobbra igazitva & kozepre & kozepre & balra \}
```

```

\hline
Masodik sor & & yay & much wow\\
ize & bize & mize & meh... \\
}

```

Ahogy az az A.1 tablazatban lathato, bla bla bla.

Elso sor	elso cella	jobbra igazitva	kozepre	kozepre	balra
	Masodik sor			yay	much wow
	ize	bize	mize	meh...	

A.1. táblázat. Tablazat cime

### A.3. Forraskodok beemelese

Forraskodot vagy a Forraskod makroval lehet betolteni a fajlbol, ahol az elso parameter a fajl neve, a masodik tovvabbi opciok, pl hogy melyik sortol melyik sorig, stb. A masik lehetoseg, hogy a kodba keruljon be az alabbiak szerint:

```
\Forraskod{ize.c}{frame=single,lastline=4}
```

```

int fuggvenynev(int tralala){
    int a;
    int b=3; // egy nagyon hasznos komment
    return a+3*b-masikfuggveny(tralala);
}

```

```

\begin{lstlisting}[frame=single, language=C]
printf("Hello world\n");
if(whatever){
    return 1;
} else do {
    tanulmegindentalni();
} while (i<5);
\end{lstlisting}

```

```

printf("Hello world\n");
if(whatever){
    return 1;
} else do {
    tanulmegindentalni();
} while (i<5);
}

```

### A.4. Hivatkozások

Mainemelis és Ronson publikáltak egy szuper módszert amivel, sakfjlsdkfjalskjf[?].

Lucas Bordeaux a constraint és a satisfiability programmingról írtak összehasonlító elemzést 2006-ban megjelent publikációjukban[?]

Goodman 2012-es értekezése a genetikus algoritmusokba való bevezetést és azok egyszerű megértését segíti[?]

Raghavjee és Pillay 2010-es publikációjukban foglalkoztak a genetikus algoritmus mindennapi problémamegoldó képességével, egy iskolai órarend beosztása formájában[?]

Adrian Brezulianu Monica Fira és Lucian Fira 2009-es tanulmányukban ugyancsak a genetikus algoritmusok mindennapos alkalmazását tesztelték a munkahelyi beosztások elkészítésében[?]

Soricone Robert és Neville Melvin a 2004-ben megjelent cikkükben részletesebben vizsgálták a mutáció és keresztezés hatását a genetikus algoritmusokban[?]

Merelo-Guervós, Castillo, Esparcia-Alcázar és García 2013-mas értekezésükben heurisztikus megoldási módszereket vizsgáltak a Mastermind logikai feladaton[?]

T W Jerardi 1987-ben megjelent cikkjében egy Smullyan feladvány megoldását a prolog nyelvvel vizsgálja[?]

Bui Thang N. és Nguyen ThanhVu H. 2006-os értekezésükben vizsgálják a gráfszínezési módszereket[?]

Gizem Sungu és Betul Boz által 2015-ben kiadott tanulmány gráfszínezési problémákat vizsgál evolúciós algoritmusok segítségével.[?]

Jun Zhang, Henry S. H. Chung illetve Jinghui Zhong 2005-ben foglalkoztak részletesebben a mutáció és keresztezés hatásával áramkörök optimalizálásánál[?]

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

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

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

Sami Khuri, Tim Walters és Yanti Sugono már 2000-ben kutatta genetikus algoritmus segítségével a gráfok, és gráfok széleinek színezését[?]

Deon Garrett , Joseph Vannucci, Rodrigo Silva, Dipankar Dasgupta és James Simien 2005-ben foglalkoztak az úgynevezett „Sailor assignment” feladvány megoldásával[?]