Zebra típusú logikai rejtvények megoldása evolúciós algoritmussal

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

Témavezető: Dr. Hegyháti Máté

2018. február 27.

Tartalomjegyzék

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 gyökérből 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 genetikus algoritmus került kidolgozásra, és tesztelésre több példán. A genetikus 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 teszteset futtatásának tapasztalai 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 egyedreprezentációkat 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 bevezeto mondat, + hogy melyik alfejezetben mi lesz

2.1. Történet és szerkezet

TODO: Miert zebra, mikbol all a rejtveny, Einstein peldajabol reszlet akar

2.2. Megoldhatóság, egyértelműség

TODO: Pici peldakon bemutatni, hogy ha rosszak a szabalyok, akkor lehet nincs megoldas, vagy ha keves a szabaly, akkor lehet tobb megoldas is van. Egy nagyon apro (3 szek, 2 tulajdonsag mondjuk) pelda kitalalasa es megoldasa par lepesben.

2.3. Megoldó módszerek

TODO: Irodalomban talalhato modszerek, 2 mondat roluk, hivatkozasok

3. fejezet

Evolúciós algoritmusok

TODO: Tortenetuk, hivatkozasok TODO: Altalanos felepitesuk TODO: Akar par szo arrol, mi mindenre alkalmaztak oket, hivatkozasok

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

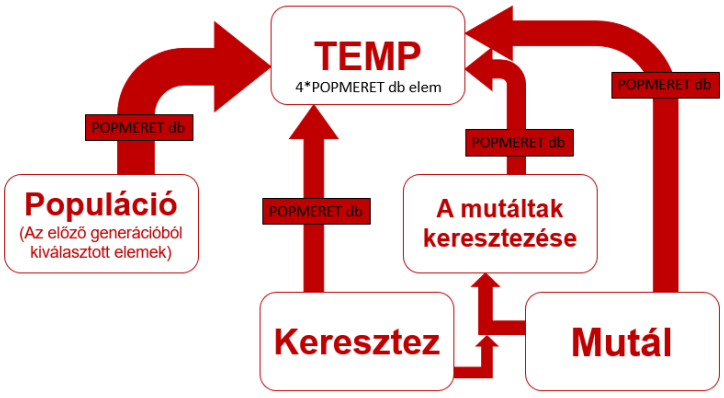
A kód alapvetően 2 logikai egységre szedhető szét. Van egy egységes, genetikus kód rész és egy az adott feladatra vonatkozó specifikációkat tartalmazó rész. Ez a genetikus algoritmusban három különböző részre jól elkülönítve látszik. A genetikus rész 2 .c fájlban található egy main és egy a genetikus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

4.1. ábra. A program lefutása

`hanyatSert`nek nevezzük. A program célja az, hogy a fitnesz értéke 0 legyen, ami akkor valósül 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 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 feladatspecifikus részek jól elkülöníthetők a programban. Itt generálható a feladatban szereplő adatokból a neki megfelelő makrók, hogy későbbiekben egyszerűen egy számként hivatkozhassunk az egyes egyedekre. A programban létrehozott egyedek kritériumoknak való megfelelését a fitnesz 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.

4.2. Egyedreprezentáció és segédfüggvények

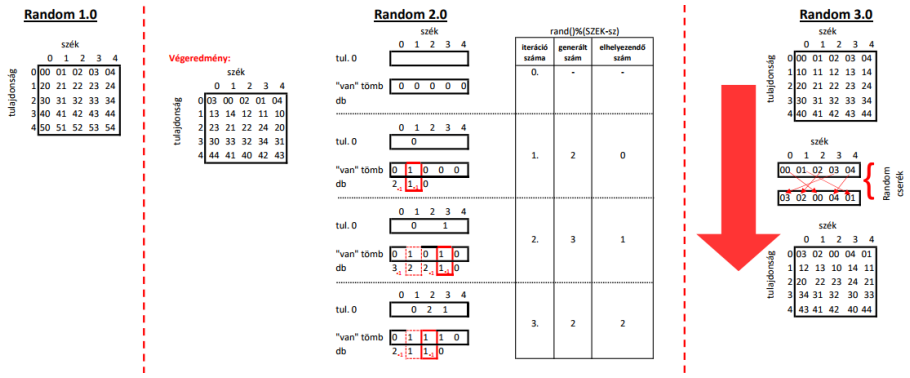
TODO: Hogy reprezentaljuk az egyedet TODO: egyedkiir, sorbarendez, ...

4.3. Evolúciós mechanizmusok

TODO: esetleg par felvezeto szo, a kapcsolodo makrok megemlitese (popmeret, megtart)

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.



4.3. ábra. Random egyed generálási módszerek.

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

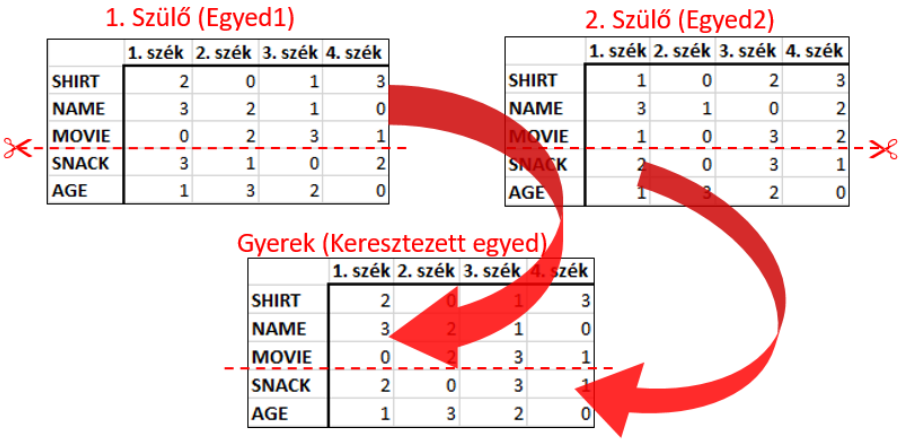
4.3.2. Mutálás

TODO: Ugyanez. Milyen valtozatok voltak, vannak, részletesen bemutatva

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 4.4 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.



4.4. 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 lévőn 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 genetikus 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*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

TODO: Ugyanez. Milyen valtozatok voltak, vannak, részletesen bemutatva

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 fitnesz függvény, a hanyatSert függvény.

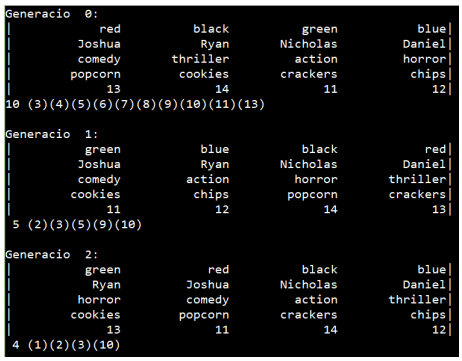
```
int hanyatSert(struct gen egyed){
    int sert=0;
    sert+=Teszt1(egyed);
    sert+=Teszt2(egyed);
    sert+=Teszt3(egyed);
    ...
    sert+=Teszt11(egyed);
    sert+=Teszt12(egyed);
    sert+=Teszt13(egyed);
    return sert;
```

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 értéked ad vissza, attól függően hogy hány darab tesztet sértett meg. Jelen esetben 13 teszt feltételének kell a programunknak szimultán megfelelnie.

5. fejezet

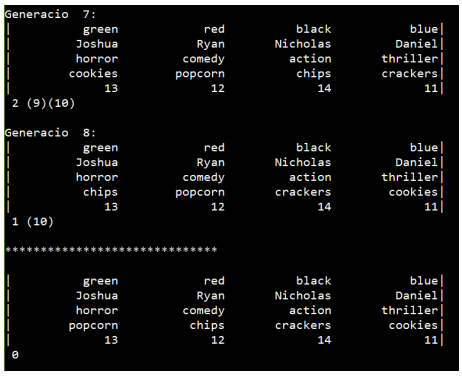
Tesztek

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



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.



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 végeredényhez 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.

			1. szék	2. szék	3. szék	4. szék
TULAJDONSAG	5					
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 POPMÉRET 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 teszttípusok 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 bo		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 goi		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 csinaltunk 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. Tablazatok

Tablazatot a `Tablazat` makroval lehet csinalni az alabbiak szerint. Az elso parameter a cimke, ebbol lesz egy tab_cimke cimke, amit ref-ekhez lehet hasznalni. MAsodik parameter a tablazat cime, a harmadik az oszlopok szerkezete (lasd tabular tutorial), az utolso maga a tablazat tartalma.

```
Ahogy az \aref{tab_cimke} tablazatban lathato, bla bla bla.
\Tablazat{cimke}{Tablazat cime}{r||cc|l}
{
    Elso sor elso cella jobra 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 jobra 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 tovabbi 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}
```

```c
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 {
        tanuljmegindentalni();
    } while (i<5);
\end{lstlisting}
```

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