FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Kővári Bence

Diplomaterv Útmutató

Konzulens

BUDAPEST, 2015

Tartalom

[Összefoglaló 5](#_Toc119572586)

[Abstract 6](#_Toc119572587)

[1 Bevezetés 7](#_Toc119572588)

[1.1 Témaválasztás oka 7](#_Toc119572589)

[1.2 Sudoku és Killer sudoku bemutatása 7](#_Toc119572590)

[1.3 Probléma nehézsége 7](#_Toc119572591)

[2 Irodalomkutatás 8](#_Toc119572592)

[3 Használt technológiák, algoritmusok 9](#_Toc119572593)

[3.1 ASP .NET 9](#_Toc119572594)

[3.2 Blazor 9](#_Toc119572595)

[3.3 Genetikus algoritmus 9](#_Toc119572596)

[4 Megoldó genetikus algoritmussal 10](#_Toc119572597)

[4.1 Tervezés 10](#_Toc119572598)

[4.1.1 Kromoszóma 10](#_Toc119572599)

[4.1.2 Fitnesz függvény 10](#_Toc119572600)

[4.1.3 Szelekció 10](#_Toc119572601)

[4.1.4 Keresztezés 10](#_Toc119572602)

[4.1.5 Mutáció 10](#_Toc119572603)

[4.1.6 Killer sudoku különbségek 10](#_Toc119572604)

[4.2 Genetikus keretrendszer implementáció 10](#_Toc119572605)

[4.2.1 Osztálydiagram 10](#_Toc119572606)

[4.3 Tesztelési tapasztalatok 10](#_Toc119572607)

[5 Sudoku rekurzív algoritmussal 11](#_Toc119572608)

[5.1.1 Megoldó algoritmus 11](#_Toc119572609)

[5.1.2 Generáló algoritmus 11](#_Toc119572610)

[5.1.3 Tesztelési tapasztalatok 11](#_Toc119572611)

[6 Killer sudoku rekurzív algoritmussal 12](#_Toc119572612)

[6.1.1 Megoldó algoritmus 12](#_Toc119572613)

[6.1.2 Generáló algoritmus 12](#_Toc119572614)

[6.1.3 Tesztelési tapasztalatok 12](#_Toc119572615)

[7 Sudoku emberi algoritmussal 13](#_Toc119572616)

[7.1.1 Osztálydiagram 13](#_Toc119572617)

[7.1.2 Megoldó algoritmus 13](#_Toc119572618)

[7.1.3 Generáló algoritmus 13](#_Toc119572619)

[7.1.4 Tesztelési tapasztalatok 13](#_Toc119572620)

[8 Killer sudoku emberi algoritmussal 14](#_Toc119572621)

[8.1.1 Megoldó algoritmus 14](#_Toc119572622)

[8.1.2 Generáló algoritmus 14](#_Toc119572623)

[8.1.3 Tesztelési tapasztalatok 14](#_Toc119572624)

[9 Frontend 15](#_Toc119572625)

[9.1 Sudoku 15](#_Toc119572626)

[9.2 Killer Sudoku 15](#_Toc119572627)

[10 Irodalomjegyzék 16](#_Toc119572628)

[Függelék 17](#_Toc119572629)

Hallgatói nyilatkozat

Alulírott **Kővári Bence**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 10. 19

...…………………………………………….

Kővári Bence

Összefoglaló

A szakdolgozat, vagy diplomaterv elkészítése minden egyetemi hallgató életében egy fontos mérföldkő. Lehetőséget ad arra, hogy az egyetemi évei során megtanultakat kamatoztassa és eredményeit szélesebb közönség előtt bemutassa, s mérnöki rátermettségét bizonyítsa. Fontos azonban, hogy a dolgozat elkészítésének folyamata számos csapdát is rejt magában. Rossz időgazdálkodás, hiányos szövegszerkesztési ismeretek, illetve a dolgozat készítéséhez nélkülözhetetlen „műfaji” szabályok ismeretének hiánya könnyen oda vezethetnek, hogy egy egyébként jelentős időbefektetéssel készült kiemelkedő szoftver is csak gyengébb minősítést kapjon a gyenge minőségű dolgozat miatt.

E dokumentum – amellett, hogy egy általános szerkesztési keretet ad a dolgozatodnak – összefoglalja a szakdolgozat/diplomaterv írás írott és íratlan szabályait. Összeszedjük a Word kezelésének legfontosabb részeit (címsorok, ábrák, irodalomjegyzék stb.), a dolgozat felépítésének általános tartalmi és szerkezeti irányelveit. Bár mindenkire igazítható sablon természetesen nem létezik, megadjuk azokat az általános arányokat, oldalszámokat, amelyek betartásával jó eséllyel készíthetsz egy színvonalas dolgozatot. A részletes és pontokba szedett elvárás-lista nem csupán a dolgozat írásakor, de akár más dolgozatok értékelésekor is kiváló támpontként szolgálhat.

Az itt átadott ismeretek és szemléletmód nem csupán az aktuális feladatod leküzdésében segíthet, de hosszútávon is számos praktikus fogással bővítheti a szövegszerkesztési és dokumentumkészítési eszköztáradat.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül. Ez a magyar nyelvű összefoglaló angolra fordított változata.

# Bevezetés

## Témaválasztás oka

## Sudoku és Killer sudoku bemutatása

## Pontosított specifikáció

### Probléma nehézsége

## Dolgozat felépítése

# Irodalomkutatás

# Használt technológiák, algoritmusok

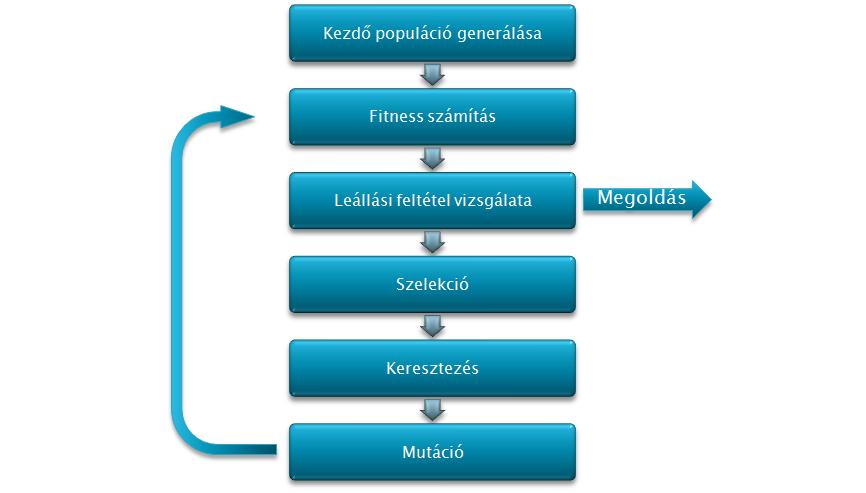
## ASP .NET

## Blazor

## Genetikus algoritmus

Egyik megoldó programom genetikus algoritmussal készítettem. Itt most az algoritmust mutatnám be röviden.

A genetikus algoritmus egy olyan keresési technika, mellyel egy optimumot lehet megkeresni. A genetikus algoritmus az evolúciós algoritmusok egy speciális változata.



1. ábra Genetikus algoritmus működése

A genetikus algoritmus általános működése a fenti ábrán látható. Az algoritmus egy kezdő populációból indul. Ez a populáció egyedek, más néven kromoszómák halmaza. Minden kromoszóma génekből épül fel. Az algoritmus célja, hogy megtalálja a „legjobb” egyedet, vagy azok egy halmazát.

Minden kromoszóma rendelkezik egy fitnesz értékkel, ami alapján lehet őket egymáshoz hasonlítani. Ezt a fitneszt az algoritmus előbb a populáció minden egyedére kiszámolja, majd eszerint rendezi az őket. Ekkor egy előre megadott leállási feltételt kiértékel. Amennyiben ez a feltétel teljesül, az azt jelenti, hogy az algoritmus megtalálta a keresett egyedet, vagy egyedeket. Nem teljesülés esetén az algoritmus előállítja az egyedek következő populációját.

A populáció előállításához az első a szelekciós lépés. Ekkor valamilyen meghatározott szabály alapján a fitnesz által rendezett egyedek közül az algoritmus kiválasztja azokat, melyek az úgynevezett „szülők” lesznek. Ezeket a „szülőket” keresztezi, ami azt jelenti, hogy a szülők génjeiből előállítódnak a gyerekek génjei valamilyen meghatározott szabály alapján. Minden így keletkezett „gyerekre” az algoritmus még végrehajtja a mutációs lépést, ami a „gyerek” génjei közül néhányat (lehet, hogy egyet sem) valamilyen valószínűség, és szabály szerint megváltoztat.

Az így kapott „gyerekek” halmaza az új populáció. Ekkor ezt a populációt tekinthetjük ismét a „kezdő” populációnak, és így az algoritmus a leírtak szerint tud tovább futni.

# Megoldó genetikus algoritmussal

A sudoku és killer sudoku feladványokra először egy genetikus algoritmussal működő megoldó programot készítettem. Ezt az algoritmust mutatom most be.

## Tervezés

Az előző fejezetben bemutattam, hogyan működik a genetikus algoritmus általánosságban. Ebben a fejezetben bemutatom, hogy az algoritmus egyes lépéseinek milyen szabályokat, algoritmusokat választottam, hogy azzal egy sudoku feladványt meg lehessen oldani. Fontos szempont volt a tervezésnél, hogy a killer sudoku megoldóhoz ezek a szabályok felhasználhatóak legyenek.

### Kromoszóma

Genetikus algoritmus tervezésénél mindig a legelső lépés a kromoszóma, és annak génjeinek meghatározása. Ebben a feladatban értelemszerű volt egy megoldási kísérletet kromoszómának választani, vagyis egy, az algoritmus által kitöltött (nem feltétlenül jó) megoldást. Géneknek pedig ennek a megoldási kísérletnek egy olyan halmazát, amelyben minden szám egyszer szerepel.

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

2. ábra Gén

Az ábrán látható a kromoszóma egy génje, és annak indexelése. Génnek a sudoku egy négyzetét választottam, az indexelés pedig a bal felső saroktól indítva a jobb alsó sarokig történik. Így egy 9x9 normál méretű sudoku esetén minden kromoszómának kilenc génje van, amiket a génekkel azonos logika mentén indexelünk.

Azért választottam a négyzeteket géneknek, és nem például a sorokat, mivel így a fitnesz függvényben bemutatott számolásnál nem kellett a négyzeteken belül ellenőrzést végrehajtani. Ugyanezt el lehet érni a sorokra, vagy az oszlopokra is, azonban azok ellenőrzése könnyebben megtörténhet, mint a négyzeteké.

### Fitnesz függvény

A fitnesz függvény felelős az aktuális populációban levő minden egyed fitnesz értékének kiszámításáért. Ezek a fitnesz értékek lehetnek jutalompontok különböző feltételek teljesülése, illetve büntetőpontok nem teljesülés esetén. Én a büntetőpontos megközelítést választottam a függvény tervezésekor.

A függvény végignézi minden egyed sorait és oszlopait, majd kétféle esetben oszt büntetést. Minden hiányzó számért jár +1 büntetőpont, illetve minden olyan számért, amelyik többször szerepel az adott sorban, vagy oszlopban annyi büntetőpont jár, amennyiszer előfordul ott. Miután egy egyed összes sorát és oszlopát megvizsgálta a függvény, a kapott fitneszt visszaadja az egyednek. A büntetőpontok miatt értelemszerűen az számít a „fittebb” egyednek, amelyik minél kevesebb fitnesz értékkel rendelkezik. Illetve akkor kapjuk meg a megoldást, ha egy nulla fitneszű egyedet találunk.

Büntetőpontos fitnesz számítás esetén gyakori még az „öregedés” bevezetése. Ez azt jelenti, hogy minden olyan egyed, amely „túlél” egy populációt, és a következő populációban is szerepel, büntetőpontot kap. Én a programomban az ilyen egyedeket plusz 1 ponttal „jutalmaztam”.

### Szelekció

A szelekció, vagyis a „szülők” kiválasztása előtt az algoritmusom még két lépést csinál. Először rendezi a populáció egyedeit a fitneszük alapján csökkenő sorrendben. Ezt követi egy úgynevezett „elitism” lépés. Ekkor a populáció legfittebb, vagyis jelen esetben legkevesebb fitnesz értékkel rendelkező egyedei közül néhányat beleteszünk a következő populációba is.

A tényleges szelekció ezt követően történik. Két fajtája lehet egy szelekciónak: az utód egy vagy két szülőből állítódik-e elő. Én az utóbbit választottam.

for (int i = Population.Count - 1; i >= Elitism; i--)

A szelekció egy ciklusban történik, ahogy a fentebbi kódrészleten látszik. A populáció egyedeit 0 és populációszám-1 közötti indexekkel lehet elérni. A ciklust ezért indítom (populációszám-1) – től. A ciklusváltozó egyesével csökken, és mivel az *Elitism* változó értékéig tart, ezzel biztosítva van, hogy a következő populáció is ugyanannyi elemből fog állni, mint a mostani, hiszen a szelekció előtt pontosan *Elitism* számú egyed került a következő populációba.

private Chromosome Selection(int i)

{

int x = (int) (i \* random.NextDouble());

return Population[x];

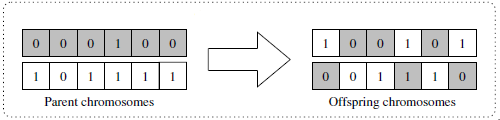
}

A fenti kódrészletben látható szelekciós függvény minden ciklusban kétszer hívódik meg, így kapjuk meg a két „szülőt”. A függvény paraméterként megkapja a ciklusváltozót. Ennek értéke alapján generál egy random számot, ami azt fogja jelölni, hogy a populáció annyiadik elemét választotta.

Mivel a ciklusváltozó folyamatosan csökken, ez azt jelenti, hogy a szelekcióban azok az egyedek, amelyek fitnesz értéke kisebb, gyakrabban lesznek kiválasztva, hiszen a sorszámuk a ciklus több lépésében kerül be a generálható sorszám közé. A ciklus első lépésében a szelekció minden egyed közül generálhat „szülőt”, az utolsó lépésben azonban, már csak a legfittebb egyedek közül választhat. Így elérjük, hogy a következő populációban már valószínűleg fittebb egyedeink lesznek.

### Keresztezés

A keresztezés, vagy angolul crossover az a művelet, amikor a „szülők” génjeiből előállítjuk az „utódok” génjeit. Ebből a műveletből is sokféle variáció létezik, én az algoritmusomhoz a legnépszerűbb Uniform crossovert választottam.

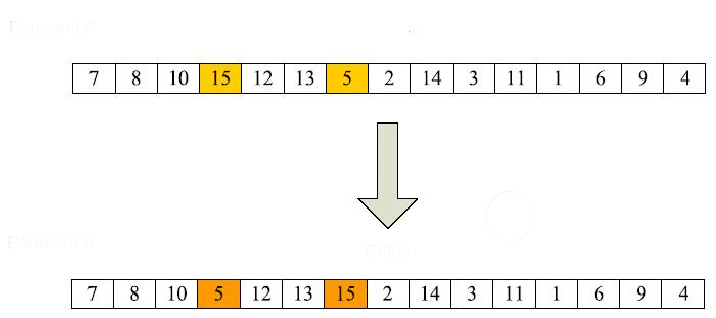


3. ábra Uniform crossover

Az ábrán látható módon az „utód” kromoszómát úgy állítjuk elő, hogy a „szülők” génjei közül felváltva választunk. Így kettő gyerek génjeit is elő lehet állítani, én a programomban az így keletkező utódok közül csak az egyiket tartottam meg.

### Mutáció

A következő populáció előállításának utolsó lépése a mutáció. Ennek a lépésnek a lényege, hogy a már előállított „gyerek” elemek génjei közül néhányat megváltoztatunk. Így olyan gének is létrejöhetnek, amelyek csak a korábbi lépésekből nem jöhettek volna létre.



4. ábra Swap Mutation

A mutációs algoritmusnak „swap mutation”-t választottam. Ez az algoritmus egy lépésben a génen belüli elemeket megcseréli. Más alkalmazása lehet ugyanennek az algoritmusnak, ha a kromoszóma génjeit cseréli meg, én ezt a programomban nem alkalmaztam.

A génen belüli cserének több előnye is volt. Ahogy a korábbi lépéseket (szelekció, keresztezés) ismertettem, fontos megjegyezni, hogy mindegyiket gének között alkalmaztam. A mutáció az egyetlen olyan lépés, amely géneken belül változtat. Azonban minden változtatás egy csere, vagyis ha a kiinduló gének generálásánál figyelünk arra, hogy minden génben a 9 elem 1-től 9-ig különböző számok legyenek, akkor elérjük, hogy bármennyi mutációs lépés után is biztosan ezek az elemek maradnak a génekben. Így biztosítva van, hogy a minden négyzeten belül különböző számok lesznek. Emiatt nem kell a fitnesz függvényben külön ellenőrzést végezni a négyzetekre.

Egy sudoku feladványban mindig vannak megadva kezdő számok, amelyeket aztán nem lehet megváltoztatni. Mivel géneken belül csak a mutációs függvény változtathat, ezért elég ennél a függvénynél elérni, hogy ilyen elemeket ne lehessen megcserélni. Ehhez a kiinduló helyzetet külön el kell tárolni, és ha olyan számot akarna a mutációs függvény a cseréhez kiválasztani, amelyiket nem lehetne, akkor a függvény meghiúsul.

A „swap mutation”-nek, mint már említettem több változata is van. Például lehet egyszerre több számot is megcserélni, nem csak kettőt. Én több fajta cserét is kipróbáltam, de mivel a bonyolultabb változatok nem adtak semmivel sem jobb eredményt, ezért maradtam az egyszerűbbnél.

### Killer sudoku különbségek

Fontos célkitűzés volt a sudoku tervezésénél, hogy az ott megvalósított algoritmusokat lehessen killer sudokura is felhasználni. Az imént bemutatott algoritmus minden eleme használható a killer sudoku megoldására is. Az egyetlen függvény, amit bővíteni kell a fitnesz függvény.

Killer sudokunál a mezőket kisebb csoportokra osztják, és azok összegének egy előre meghatározott számmal kell egyenlőnek lenniük. Ezt a szabályt kell még megvizsgálni a fitnesz függvényen belül.

A sudokuhoz megírt fitnesz függvényt szintén meg kell hívni, hiszen minden sudokuban érvényes szabály a killer sudokuban is betartandó. Ezután a függvény minden „killer csoporton” végigmegy. Amennyiben az összeg nem egyezik meg a várt összeggel, akkor annyi büntetőpontot kap a kromoszóma, amennyivel a kapott összeg eltért az elvárttól. Képlettel leírva: *Büntetés += |elvárt összeg – kapott összeg|.*

Ezzel a változtatással egy killer sudoku megoldót kapunk.

## Genetikus keretrendszer implementáció

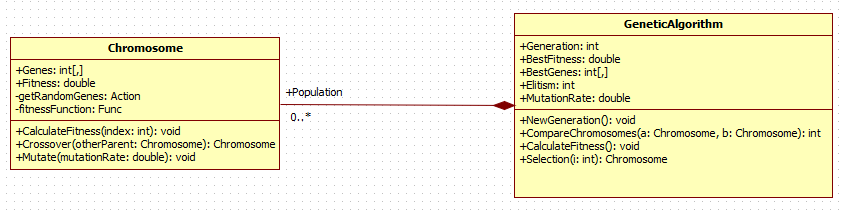
Ebben a fejezetben a genetikus algoritmus implementálásáról írnék röviden.

A genetikus algoritmusra léteznek már megírt keretrendszerek. Ilyen például a GeneticSharp [1], amit kipróbáltam az alkalmazáshoz. Ez a keretrendszer kiválóan működik, ha egy már kész algoritmust kell leprogramozni. Azonban nekem nem volt ilyen, a fentebb ismertetett algoritmust is rengeteg próbálkozás után állítottam elő. A sok próbálkozáshoz viszont nem volt kényelmes használni, ezért végül a saját megvalósítás mellett döntöttem.

### Osztálydiagram

A célom egy olyan keretrendszer létrehozása volt, amiben csak a genetikus algoritmushoz elengedhetetlenül szükséges lépések jelennek meg, hogy minél egyszerűbben lehessen használni, és ami még fontosabb, könnyen lehessen az egyes lépéseken változtatni benne.

A megvalósítást a következő osztálydiagram mutatja be:



5. ábra Genetikus osztálydiagram

Mindössze két osztályra van szükség: a *Chromosome* osztály a kromoszómát reprezentálja, és minden olyan műveletet elvégez, amit egyetlen kromoszómán kell az algoritmusnak végrehajtania.

Az attribútumai között tárolja a géneket, illetve a fitnesz értéket. Ezenkívül két függvényt is eltárol, a fitneszfüggvényt, illetve a random gén generáló függvényt. Ezen metódusokat a konstruktorban kapja meg az osztály a hívótól. A függvények függetlenek az osztálytól, a kromoszómától, ezért nem itt valósítottam meg őket. Így lehetett például elérni, hogy a killer sudoku megoldó program írásakor ehhez az osztályhoz ne kelljen hozzányúlni, csak az átadott függvényt kellett kicserélni.

A konstruktor hívásakor a kapott *getRandomGenes* metódus lefut, ami a kromoszóma kiinduló génjeit generálja. Ebben a függvényben figyelni kell, hogy a generált gének a tervezés részben bemutatott szabályoknak megfeleljenek. Vagyis minden génben a kiinduló számok a megfelelő helyeken legyenek, illetve egy génen belül minden szám egyszer szerepeljen.

Az osztály metódusai a kromoszómán elvégzendő műveleteket valósítják meg. A *CalculateFitness* az eltárolt fitnesz függvényt hívja meg. A *Crossover* és *Mutate* függvények a már ismertetett keresztezés és mutációs algoritmusokat valósítják meg.

A *GeneticAlgorithm* osztály felelős az algoritmus futtatásáért. Ezt a *NewGeneration* metódusban hajtja végre, amiben előállítja a következő „generációt”/populációt. Először a *CalculateFitness* függvénysegítségével kiszámolja az aktuális populáció összes elemének a fitnesz értékét, majd a *CompareChromosomes* metódus alapján rendezi őket. Az eltárolt *Elitism* változó alapján a legjobb egyedeket beválogatja a következő populációba. A többi egyedhez előbb a szülőket kiválasztja a S*election* függvény segítségével, majd ezekből az egyiken a *Crossover* metódust meghívva keletkezik a „gyerek” egyed. Ezen az egyeden még a *Mutate* függvény kerül meghívásra,majd hozzá lehet adni a következő populációhoz. Annyi új „gyerek” egyedet kell létrehozni, hogy a következő populációban szereplő kromoszómák száma ugyanannyi legyen az aktuálissal.

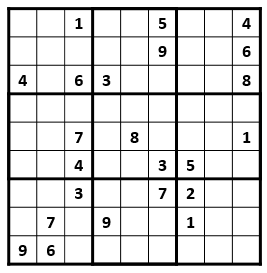
Az osztály ezenkívül még tartalmaz egy *BestFitness* és *BestGenes* változókat. Ezek értékét a *CalculateFitness* metódusban állítja be. Ezen változók segítségül szolgáltak debugolás közben az egyes lépések nyomon követésére, illetve, hogy az algoritmus mennyire halad a 0 fitnesz elérésének irányába.

A genetikus algoritmus futtatásához a *GeneticAlgorithm* osztályt kell példányosítani, át kell adni neki a populációszámot, a génszámot, az *elitism* változó értéket, illetve a megírt fitnesz és géngeneráló függvényeket.

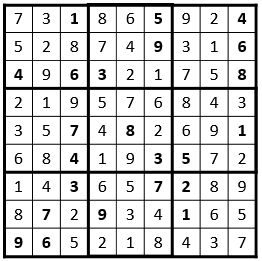
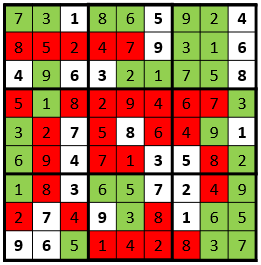
## Tesztelési tapasztalatok

Az programot különböző oldalak különböző nehézségű feladványaival teszteltem. Az általános tapasztalat mind sudoku, mind killer sudoku esetében, hogy a könnyű, és közepes nehézségű feladványokat elég gyorsan (1-2mp) meg tudja oldani. Azonban minél nehezebb feladványt adunk be neki, annál lassabb lesz.

A nehezebb feladványok többnyire azért nehezebbek, mert kevesebb kiinduló szám van megadva az elején. A program pedig azért lesz sokkal lassabb ezekben az esetekben, mivel ilyenkor nagyobb eséllyel kerül egy úgynevezett lokális minimum állapotba.



6. ábra Nehéz sudoku kiinduló helyzet

 A fentebb található három ábrán lehet látni egy nehéz sudoku feladványt, annak a megoldását, és egy talált „megoldást”. Ezt a talált „megoldást” a programom viszonylag gyorsan (2-3mp alatt) találja meg. Azok a számok vannak az ábrán bekarikázva, amelyek miatt a fitnesz függvény büntetőpontot oszt. Látható, hogy mindössze két olyan oszlop van (a 2. és a 3.) amelyben nem érvényesülnek a sudoku szabályai.

7. ábra Megoldás

8. ábra Talált "megoldás"

A program azonban ennek ellenére nincsen közel a megoldáshoz. Azok a számok, amik a bal oldali ábrán látható megoldáshoz hasonlóan ugyanott helyezkednek el, zölddel, míg azok, amelyek aszerint rossz helyen vannak, pirossal lettek kiszínezve. Látható, hogy a talált megoldásban szereplő számok közel fele rossz helyen van a tényleges megoldáshoz képest.

Ez a genetikus algoritmusoknál előforduló lokális minimum problémája. Az algoritmus optimalizál a megadott szabályoknak megfelelően, és amint egy olyan helyzetet talál, amiből már nem lehet optimalizálni, akkor onnan magától nagyon nehezen fog tudni kijutni. A lokális minimumból való kijutáshoz sok mindent kipróbáltam. A legtöbb helyen a mutáció változtatását javasolták. A mutációhoz éppen ezért sokféle változatot is kipróbáltam, az itt bemutatott adta a legjobb eredményeket. Próbálkoztam még a mutációs ráta (mutáció végrehajtódásának valószínűsége) növelésével bizonyos generáció szám elérése után, de ez sem vezetett semmilyen javuláshoz. A legjobb eredményt az hozta, ha az algoritmust egy idő után teljesen elölről újraindítom, ha elért egy bizonyos generációszámot. Értelemszerűen így viszont a megoldó program rendkívül lassú lett. Egy generáló programhoz pedig úgy gondolom elengedhetetlen egy gyors megoldó program, mivel anélkül nem lehet generálni a feladványt, már csak azért sem, mert a generált feladványt meg is kellene oldani, hogy a felhasználó megoldását ellenőrizhessük.

Mivel megoldást nem találtam, így úgy döntöttem, hogy más, egyszerűbb algoritmusokkal is megpróbálkozom megoldani a problémát.

# Sudoku rekurzív algoritmussal

Ebben a fejezetben egy nagyon egyszerű, rekurzívan működő algoritmust mutatok be sudoku program megoldására.

## Megoldó program

A rekurzió egy programozási módszer, melyben az algoritmust megvalósító függvény végrehajtása során egy ponton saját magát hívja meg. Azért nevezem a most bemutatott megoldó algoritmust rekurzív algoritmusnak, mivel annak egésze egy rekurzív függvényben van megvalósítva. Bár a programot igyekeztem úgy megvalósítani, hogy bármely méretű feladvány megoldására alkalmas legyen, a következőkben az egyszerűség kedvéért feltételezzük, hogy a hagyományos 9x9 méretű feladványról beszélek.

### Az algoritmus

Az algoritmus bemenete a megoldandó sudoku feladvány. Az algoritmus csak egyetlen mező kitöltését valósítja meg, így legalább annyiszor kell futtatni, ahány kitöltendő mező van. Az algoritmus lépései a következők:

* Megkeresi a legelső olyan mezőt, amelynek nincsen még értéke. A legelső mezőt a bal felső saroktól indulva kapjuk meg.
* Ebbe a mezőbe megpróbálja az összes lehetséges számot beírni. 9x9-es sudoku esetén 1-től 9-ig minden számot.
* Minden számnál ellenőrzi, hogy ha a sudoku addigi állapotához hozzávesszük ezt a mezőt ezzel a számmal, úgy is egy szabályos helyzet keletkezik-e. Ez azt jelenti, hogy meg kell vizsgálni, hogy a kérdéses mezőhöz tartozó sorban, oszlopban, illetve négyzetben szerepel-e a választani kívánt szám. Ha szabálytalan helyzet keletkezne, akkor ezt a számot elveti, és a következőt próbálja meg.
* Szabályos helyzet előállásánál a számot beírja a mezőbe, és ugyanezt az algoritmust futtatja az így keletkezett sudoku feladványra.
* Ha már nem talál olyan mezőt, amelynek ne lenne értéke, az azt jelenti, hogy egy szabályos megoldást kaptunk (hiszen minden helyzetben csak azokat a lépéseket engedte az algoritmus végrehajtani, ami egy szabályos állapotba vezetett), így ezt jelzi az algoritmus a korábbinak.

Egyetlen algoritmus futása másképpen is véget érhet. Amennyiben talál még üres mezőt, és a számok végigpróbálgatása során kiderül, hogy egyiket sem lehet választani, mert mindegyik szabálytalan helyzetet eredményezne, akkor az algoritmus jelzi, hogy ezt a feladványt nem lehet megoldani. (Ha egy mezőnek egyetlen szám sem lehet az értéke, akkor értelemszerűen az egész feladványnak nincsen megoldása.) Ez azt jelenti, hogy valamelyik korábbi mező kitöltésénél az algoritmus rossz számot. Az algoritmust, amelyik hibát jelzett az eggyel korábbi mezőt kitöltő algoritmus hívta meg, így a hibajelzést is ez fogja megkapni. Ekkor tudni lehet, hogy a szám, amit ez az algoritmus választott rossz választás, hiszen az így keletkezett újabb sudoku feladványnak nem volt megoldása. Így az általa választott számot kitörli, és a következőt fogja megpróbálni a fentebb ismertetett lépések szerint. Amennyiben már az összes számot végigpróbálta, Ő is hibát jelez.

Ily módon kétféleképpen érhet véget a legelső mező kitöltéséért felelős algoritmus futása. (Ami ugye a teljes megoldó eredményét fogja adni.) Amennyiben hibát jelez, az azt jelenti, hogy az eredeti feladványnak nem volt szabályos megoldása. A másik esetben az utolsó kitöltendő mező azt jelzi, hogy talált jó megoldást. Ekkor ez lesz az eredeti feladványnak is a megoldása.

### Megvalósítás

Az algoritmus megvalósításához mindössze egyetlen osztályt hoztam létre *SudokuBoard* néven. Ennek oka, hogy a sudoku feladványban szereplő mezőknek és azok értékeinek tárolásához tökéletesen alkalmas a C#-ban elérhető mátrix típus. Ennek a típusnak a segítségével az egyes mezőket könnyen el lehet érni, hiszen az indexelés eléggé intuitív, illetve a mezőknek elég csak azt az értéket eltárolni, amilyen szám szerepel rajtuk, és ha semmilyen, akkor azt lehet egy érvénytelen (például 0) értékkel jelezni.

Az osztály publikus interfésze a következő függvényekből, és tulajdonságokból áll:

public int Size { get; }

public int[,] Board { get; }

public void Init(int[,] board);

public void Print();

public bool Solve();

A Size és Board tulajdonságok a feladványt, és annak méretét tárolják. Az Init függvényben kell megadni a megoldandó rejtvényt. A Print függvény konzolra írja ki a táblában szereplő számokat, ezt főképp debugolásnál használtam. A Solve függvény pedig a feladványt megoldó eljárás, aminek a működése az előző fejezetben leírtak szerint történik, annyi különbséggel, hogy ez a függvény nem ellenőrzi, hogy a feladványnak több megoldása van-e, hanem csak az elsőt visszaadja, feltételezve, hogy a feladvány, amit kapott az helyesnek tekinthető. Az osztály még két publikus metódussal rendelkezik, de mivel azok a generáláshoz kellenek, ezért azokat abban a fejezetben fogom részletezni.

### Hatékonyság

A programot többféle nehézségű feladvánnyal teszteltem. Azonban a hagyományos, 9x9-es méretű sudoku esetén nem számított a nehézségi szint, mindenképpen egy másodperc alatt oldotta meg a programom. Erre nem igazán számítottam, mivel az algoritmus kiindulópontja, hogy lényegében végigpróbálja az összes lehetőséget. Ezek szerint viszont azzal, hogy azonnal visszalép amint talált egy rossz mezőt annyit lehet gyorsítani, hogy 9x9-es sudoku esetén már egy kiváló megoldót kapjunk, amit aztán fel lehet használni generálásra is.

## Generáló program

Az eredeti cél sudoku feladványok generálása volt, amit aztán a felhasználóknak meg kellene oldani. Azonban ahhoz, hogy legyen egy jó generáló program, elengedhetetlen egy gyors megoldó is. Ennek oka, hogy bármilyen feladványt is generálna a generálónk, csak akkor tudjuk biztosan megállapítani, hogy az így kapott feladványnak pontosan egy megoldása van, ha utána azt meg is oldja a megoldó programunk. A genetikus megoldóból ezért nem kezdtem el generáló programot csinálni, mivel az annyira lassú volt, aminek már a felhasználók biztosan nem örülnének.

### Az algoritmus

Az algoritmusnak nincsen bemenete, kimenete pedig a generált sudoku feladvány. Az algoritmusom a következő lépésekből áll:

* Generál egy úgynevezett „bástyaelrendezést”. Ez azt jelenti, hogy véletlenszerűen kiválaszt kilenc mezőt, amelyek közül egyik sincsen a másikkal azonos sorban, oszlopban és négyzetben. Ezeknek véletlenszerű értéket ad a lehetséges értékek közül. (9x9-es sudoku esetén 1 és 9 közöttii számokból választ.)
* Ezt a „bástyaelrendezést”, ami 9 kiinduló számból álló sudoku feladványnak is felfogható, odaadja a megoldó programnak. Természetesen ennek így több megoldása lenne, azonban a megoldó program *Solve* függvényét úgy készítettem el, hogy az első megoldást adja csak vissza.
* Az így kapott „megoldás” a kiinduló helyzet. Ez egy teljesen kitöltött, szabályos elrendezése 1-től 9-ig a számoknak egy 9x9-es négyzetben úgy, hogy minden sorban, oszlopban és kisebb négyzetben minden szám csak egyszer szerepel.
* Ebből a négyzetből aztán véletlenszerűen az algoritmus elhagy néhány számot, attól függően, hogy milyen nehézségű feladványt kell generálni. Több sudoku honlap áttanulmányozása után azt vettem észre, hogy a nehézségi szintek az alábbiak szerint vannak meghatározva a kezdetben megadott számok mennyiségének függvényében: könnyű 32-36, közepes 28-32, nehéz 24-28. Az algoritmus ezek szerint alapján dönti el, mennyi kiinduló számot hagyjon meg.
* Az így keletkezett feladványt ismét átadja a megoldó programnak, azonban most az a megoldó függvény fogja megvizsgálni, amelyik nem áll meg az első megoldásnál. Amennyiben a feladványnak nem pontosan egy megoldása volt, a függvény hibát jelez, és a generáló teljesen elölről fogja kezdeni a működését.

Az így keletkezett sudoku feladvány tehát biztosan helyes lesz, hiszen pontosan egy megoldása lesz. Azonban a generáló nem veszi figyelembe, hogy a keletkezett feladványhoz milyen lépéseket kell tenni a felhasználónak, ahhoz, hogy megoldást találjon. A program csak annyit garantál, hogy lesz egy megoldás, és a kiinduló helyzetben a beállított nehézségi szintnek megfelelő mennyiségű számot fog megadni. Elméletben azonban elképzelhető, hogy egy könnyű feladvány generálása esetén nehezebb feladványt talál, mint egy nehéz feladvány generálásánál.

### Megvalósítás

A generáló algoritmus megvalósításához egy *SudokuGenerator* osztályt hoztam létre. Az osztály a konstruktorában kap egy *SudokuBoard* osztályt, amit el is tárol. Egyetlen publikus függvénye a *Generate*, ami visszaadja a tárolt sudokut, miután végrehajtotta a már ismertetett algoritmust.

do

{

\_sudokuBoard.Init(GenerateRookLayout());

\_sudokuBoard.Solve();

\_sudokuBoard.Generate(SudokuTypes.HARD);

} while (\_sudokuBoard.CheckGeneratedPuzzle() == 1);

return \_sudokuBoard;

Az algoritmus a bástyaelrendezés generálásával kezdődik, ezt a *GenerateRookLayout* privát metódus valósítja meg. Soronként megy végig a generálandó sudoku méretével megegyező nagyságú mátrixon, és minden sorban választ egy mezőt. Ahhoz, hogy ezek a mezők különböző oszlopokba kerüljenek, a még üres oszlopok indexeit eltárolja, és azok közül választ. Így a validáló függvénynek elég csak a kisebb négyzeteket megvizsgálni, hogy a mező, amit a választunk szabályos-e, ellentétben a *SudokuBoard*-ban szereplő validáló függvénnyel.

A kapott bástyaelrendezéssel a program inicializálja a feladványt, annak veszi az első megoldását, majd elhagy belőle számokat a nehézségi szint figyelembe vételével. Az így kapott sudokut a *SudokuBoard* *CheckGeneratedPuzzle* függvényével ellenőrzi, amely visszaadja, hogy több megoldása van-e a generált feladványnak, és ha igen, akkor az egész folyamat kezdődik elölről.

## Tesztelési tapasztalatok

Ez a generáló algoritmus felhasználja, hogy a megoldó megfelelően gyorsan működik. Így ennek a futási ideje is megfelelően gyors, a felhasználó átlagosan 1-2 másodperc alatt kap egy feladványt. Természetesen a sok újragenerálás miatt előfordulhat lassabb futási idő, ez főképp a nehéz feladatok generálásánál történhet meg. Tapasztalataim szerint 28 kezdési számmal még közel 100%-ig tartható, hogy a generálás két másodperc alatt megtörténik. 24-26 kezdőszám generálása esetén azonban már előfordulhat, hogy hosszabb ideig kell várakozni.

Eredetileg az algoritmusomat másképpen terveztem meg. Az ötlet az volt, hogy a bástyaelrendezés generálása, majd annak egy megoldása után mindig abból a keletkezett feladványból történjen a generálás próbálgatása. Azonban számomra érthetetlen okokból nehezebb feladatok generálása esetén rosszabb eredményeket kaptam így, mint amikor az egész folyamatot elölről kezdtem a bástyaelrendezés generálásától. Ezért választottam végül azt a megoldási folyamatot.

# Killer sudoku megoldó program rekurzív algoritmussal

Ebben a fejezetben a rekurzív algoritmussal működő sudoku megoldó programból kiinduló killer sudoku megoldót ismetetem.

Két megoldó algoritmust készítettem, amelyek a sudoku megoldóhoz hasonlóan rekurzívan próbálják megoldani a kapott feladványt. Mindkettő algoritmus megvalósítása egy osztályban, a *KillerSudokuBoard*-ban található. Ez az osztály a sudoku megoldó osztályból származik le, így rendelkezik annak publikus és protected metódusaival.

Azért került mindkettő algoritmus egy osztályba, és származtattam le ezt az osztályt a sudoku megoldó osztályból, mivel mindkettő algoritmus fel tudott így használni olyan függvényeket, amelyeket már implementáltam a sudoku megoldó készítésénél.

## Sudoku megoldó bővítése

Az első megoldó algoritmusomat úgy próbáltam elkészíteni, hogy a már megírt sudoku megoldó programot felhasználjam. Az a bal felső saroktól indulva tölti ki a sudoku táblát, és minden lépésben egyetlen számot próbál meg beírni egy üres mezőbe, miközben ellenőrzi, hogy az így kapott újabb tábla szabályos marad-e. Ezt a *Solve* függvényben valósítja meg a *SudokuBoard* osztály.

Az algoritmus megvalósítása előtt még meg kellett oldanom a killer szabályok, vagyis az összetartozó mezők, és azoknak a várt összegének az eltárolását. Ezt a *Dictionary* típussal oldottam meg. A *Dictionary* egy C# specifikus osztály, amelyben kulcs-érték párokat lehet tárolni. Kulcsnak az összetartozó mezők listáját választottam, értéknek pedig az elvárt összegüket. Mivel a kulcs egy lista, így a Dictionaryben nehezebb keresni, ennek megkönnyítésére bevezettem a *Coordinate* típust, ami egy mező koordinátáit tárolja. ’X’ a sor, ’Y’ az oszlop koordinátát jelöli. Így a *Dictionary* kulcsa egy *Coordinate* típusokat tároló lista lesz. Ennek a *Coordinate* osztálynak egy függvénye van, az Equals, amely el tudja dönteni, hogy a paraméterként megadott koordináta megegyezik-e a tárolt koordinátákkal. A Dictionaryben így könnyen lehet majd keresni, hiszen csak végig kell nézni a kulcsok között szereplő listákat, és amelyiknek az egyik elemének koordinátája megegyezik az éppen vizsgált mező koordinátájával, akkor megkapjuk azt a killer szabályt, amihez a mező tartozik.

A megoldó algoritmus a sudoku *Solve* függvényét használja fel a killer sudoku megoldásához változatlanul. Ezt azért tehetjük meg, mivel ha ugyanazon logika szerinti algoritmust készítünk, akkor csak annyi fog változni a sudokuhoz képest, hogy az ellenőrzésnél még pluszban meg kell vizsgálni, hogy a beírt szám utáni állás megfelel-e a killer sudoku szabályainak is. A validálást a sudoku megoldó az *IsValidNumber* függvényben végezte, amely három ellenőrző függvény segítségével döntötte el, hogy a mezőbe beírt szám szabályos-e, vagyis ugyanilyen szám szerepel-e ebben a sorban, oszlopban vagy négyzetben. Tehát ezt a függvényt kellene felüldefiniálni, úgy hogy ellenőrizze a killer szabályokat is.

Az ellenőrzés úgy indul, hogy az algoritmus megnézi, teljesülnek-e a sudoku szabályai, vagyis meghívja az ősosztály függvényét. Ezután végigmegy a killer szabályokon. Első lépésben a már ismertetett módon megkeresi, a kérdéses mező melyik listába tartozik a killer szabályok közül. Ezt követően már csak ezt az egy killer szabályt kell vizsgálni, hiszen ha egy mezőbe beírunk egy új számot, az csak egyetlen killer szabályra lesz hatással, arra, amelyikbe az tartozik.

A vizsgálás során a szabályba tartozó összes mező értékét megnézi az algoritmus. Amennyiben talál olyan mezőt, amelyben az a szám szerepel, amivel az éppen vizsgált mezőt szerette volna kitölteni, akkor visszatér, jelezve, hogy szabálytalan helyzet keletkezett. Egyébként a már kitöltött mezők értékeit vizsgálja meg, és az alapján próbálja eldönteni, hogy a killer szabály érvényes-e még, illetve az esetlegesen még hiányzó mezők kitöltése esetén is érvényes maradhat-e. Ennek az általános leírása elég bonyolult lenne, így inkább pár példán mutatnám be, mihez hasonló helyzeteket vizsgál meg az algoritmus.



. ábra Első killer példa

Ebben a helyzetben az 5-ös mellé a 6-os számnak kell kerülnie, hogy az összeg az elvárt 11 legyen. Így az algoritmus minden 6-ostól különböző számot el fog utasítani.

******

. ábra Második killer példa

Tegyük fel, hogy éppen a 9-es melletti mezőt vizsgálja az algoritmus. A kérdés, lehet-e oda 3-as értéket írni. Ha abba a mezőbe 3 kerülne, akkor az összeg az üres mező nélkül is már 19 lenne, vagyis az utolsó mezőbe bármilyen számot írnánk 1 és 9 között, akkor az összeg nagyobb lenne, mint 19. Emiatt az algoritmus ebben az esetben nem enged abba a mezőbe 3-ast (vagy annál nagyobb számot) írni.



. ábra Harmadik killer példa

Ugyanez a szabály igaz a másik irányba is. Tegyük fel, hogy a 9-estől baloldali mezőbe 4-est próbál meg írni az algoritmus. Ekkor a jelenlegi összegérték 13, vagyis ahhoz, hogy 22-őt kapjunk, legalább 9-est kellene írni az utolsó üres mezőbe. Ez viszont nem lehetséges, mivel 9-es már szerepel ebben a négyzetben, így azt nem lehet oda írni. Vagyis az elérhető számok közül bármelyiket választva kisebb összeget kapunk, mint az elvárt, így az algoritmus 4-est (és annál kisebb számot) nem enged beírni.

Az ellenőrző függvény felüldefiniálásán kívül nincs szükség egyéb változtatásra. Így egy jól működő megoldó programot sikerült készítenem. Azonban tesztelésnél kiderült, hogy a sudoku megoldóhoz képest ez az algoritmus sokkal lassabb. Ennek fő oka pedig nem a plusz ellenőrzés (nyilván az is számít valamennyit), hanem a killer sudoku azon tulajdonsága, hogy üres tábláról indul a feladvány. Emiatt a program az összes mezőt ki kell, hogy töltse, és az elején ráadásul szinte minden számot végig kell próbálgatnia. Így az algoritmus futásideje exponenciálisan megugrik. Ezen próbáltam meg javítani azzal, hogy jobban a killer tulajdonságokra építem a megoldót.

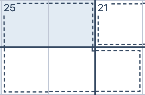
## Megoldó killer tulajdonság alapján

Ez az algoritmusom szintén rekurzív alapokra épít. A fő ötletem az volt, hogy ne a bal felső saroktól kezdve indítsa az algoritmus, hanem az eltárolt killer tulajdonságokon menjen egyesével végig. Ez azt jelenti, hogy az ősosztályban használt *Solve* függvényt így nem lehet használni. Ehelyett egy új *KillerSolve* függvényben valósítottam meg az algoritmust.

A killer szabályok egy *Dictionaryben* vannak eltárolva, így ezen kell végigiterálni. A *Dictionary* kulcsa azon mezők listája, amelyek egy killer szabály alá tartoznak. Vagyis ha ilyen sorrendben fog az algoritmus végigmenni a mezőkön, akkor az egy szabály alá tartozó mezőket egymás után fogja kitölteni. Ez a fő különbség a két megoldó függvény között, az ellenőrzésekre a már megírt függvényeket használom.

Arra számítottam, hogy ezzel a megközelítéssel javulni fog a program futási ideje, mivel a killer szabályok szerinti iterálás miatt hamarabb vissza fog lépni az algoritmus a rekurzióból, ha ezek a szabályok nem teljesülnének. Ezzel a szemben a futási idő még lassabb lett.

A következő javítási kísérletem az algoritmuson az volt, hogy a killer szabályokat rendezem aszerint, hogy hány elemet tartalmaznak.



. ábra két, illetve 5 mezőből álló killer szabályok

Például egy kételemű killer szabály egy mezőjének kitöltése jobban tud korlátozni, mint egy ötelemű szabály, hiszen a kételemű esetén az egyik mező kitöltésével már következik a másiknak az értéke is. Ezzel szemben egy ötelemű szabályból egy mező kitöltése lehet, hogy semmilyen hatással nem lesz a többi mező lehetséges értékeire. Így logikusnak tűnt, hogy a kisebb méretű szabályok legyenek előbb kitöltve. Így kevesebb rekurziós lépés lesz, mivel a nagyobb méretű szabályok kitöltésénél ezeket a mezőket már az értékkel rendelkezők miatti sudoku szabályok fogják korlátozni.

A rendezést egy *SortedList*-ben valósítottam meg, ami a C# egy beépített kulcs-érték párokat tartalmazó típusa. Minden egyes elemet a rendezés szerinti helyére szúrja be. Ezt a rendezést a kulcs szerint végzi el, amit a programozónak kell definiálni. A már leírtak szerint ez a rendezés a killer szabálylisták hossza szerint történik növekvő sorrendben.

A *KillerSolve* függvényen pedig csak annyit kellett változtatni, hogy ezen a listán iteráljon végig.

## Tesztelési tapasztalatok

Az így kapott megoldó program gyorsabb lett, mint a rendezés nélküli. Azonban ennek az algoritmusnak a futási ideje is több másodpercig, akár percig is eltartott, mint a korábban ismertetett megoldó programoké.

Érdekesség, hogy a bal felső saroktól induló rekurzív algoritmus, illetve a killer szabályok szerinti algoritmus különböző feladványoknál teljesen máshogy teljesítettek. Valamikor az egyik, valamikor a másik volt sokkal gyorsabb. Azonban az átlag futási idő 30-60 másodperc között volt mindkettő esetében nehéz killer sudoku feladványokra.

A rendezéses algoritmusnál még kipróbáltam az értékek szerinti rendezést is, vagyis hogy azok a szabályok kerüljenek előrébb, amelyekbe több szám beírása van kizárva.



. ábra killer szabályok összehasonlítása érték szerint

Az ábrán látható kételemű szabály például csak az 1-es számot nem engedi egyik mezőbe sem, míg a mellette levő háromelemű szabály minden 1 és 5 közötti számot. Így ez a szabály bár több mezőt tartalmaz, azok értékét mégis jobban korlátozza, emiatt tűnt logikusnak eszerint rendezni a szabályokat.

Az így kapott megoldó program volt azonban egyértelműen a leglassabb, ezért úgy döntöttem, tudomásul veszem, hogy a killer sudoku sajátosságai miatt (nincsen megadott szám az elején, illetve több ellenőrzés) a rekurzív megközelítés nem működik.

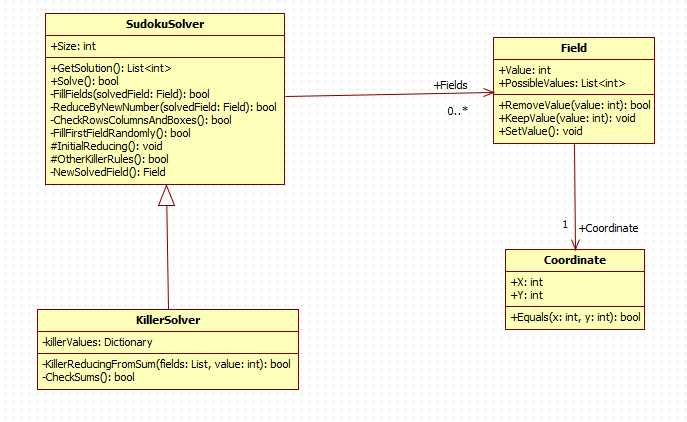
# Emberi gondolkodáshoz hasonló algoritmus

Ebben a fejezetben egy olyan megoldó algoritmust mutatok be, amelynek működése hasonlít ahhoz, ahogyan az emberek szoktak sudoku, vagy killer sudoku feladványokat megoldani. Ez általában úgy szokott történni, hogy olyan mezőket keresnek, ami csak egyetlen értéket vehet fel, és így jutnak el a megoldásig. Ehhez a gondolkodásmódhoz próbáltam meg egy algoritmust készíteni.

Egy olyan mezőt szeretnénk megkeresni, ami csak egyetlen értéket vehet fel, ezért el kell tárolni minden egyes mezőhöz azokat a felvehető értékeket, amik az éppen aktuális állás alapján oda kerülhetnek. Bevezettem egy *Field* osztályt, ami egyetlen mezőt reprezentál. Ennek a tulajdonságai között megtalálható a koordinátái (sor, oszlop), a felvehető értékek listája, illetve az érték, amit felvett. Ha a mezőnek van értéke, akkor a felvehető értékek listájának üresnek kell lennie, és ez fordítva is igaz, vagyis amennyiben ez a lista nem üres, akkor a mező értékének 0-nak kell lennie. A kettő közötti konzisztencia megtartásáért a *SetValue* függvény felel, ami akkor írja át az érték tulajdonságot, ha a lehetséges számok listája egy elemű.

## Osztálydiagram

Ehhez az algoritmushoz új osztályokat hoztam létre, ez látható az alábbi diagramon.



A diagramon az osztályokban csak a megoldó algoritmushoz érdekes függvények vannak feltüntetve. A *SudokuSolver* osztály egy listában tárolja a mező objektumokat. A *Solve* hívás hatására az összes tárolt Field objektum Value attribútumának lesz értéke, így kapjuk meg a megoldást. A KillerSolver leszármazik a SudokuSolverből és annak néhány virtuális függvényét valósítja meg, amelyek biztosítják, hogy a megoldás során a killer szabályokat is figyelembe vegye a program. A két osztály további függvényei az algoritmus egyes részeit implementálják, ezekről az algoritmus ismertetésénél fogok részletesebben írni.

## Sudoku megoldó algoritmus

Az algoritmus magas-szintű működése a következő:

* Különböző, előre definiált szabályok szerint csökkenti a mezők felvehető értékeinek listáját.
* Ha talál egy olyat, amelynek már csak egy értéke lehet, akkor az az érték lesz a mezőbe írt szám.
* Ha már egyik szabály szerint sem történt csökkentés, és nem létezik olyan mező sem, amelynek csak egy felvehető értéke van, akkor kiválasztja a legbaloldalibb elemet, és annak lehetséges értékei közül választ egyet, és azzal próbálja megoldani a feladványt. Ha ez nem sikerül, akkor választja a következőt, stb.
* Mikor minden mezőnek van értéke, az algoritmus megáll, mert megtalálta a megoldást. Ha nincsen megoldás, azt egy bool változóval jelzi.

A fent leírt egyes pontokat szeretném most részletezni. A felvehető értékek csökkentéséhez két szabályt vezettem be.

1. Minden kitöltött mező által felvett érték az azzal egy sorban, oszlopban és négyzetben szereplő mezők lehetséges értékei közül törölhetők. Ennek a szabálynak a megvalósítása a *ReduceByNewNumber* metódusban történik, ami paraméterként megkapja a legutóbb kitöltött mezőt. Felesleges lenne az összes értékkel rendelkező mezőt megvizsgálni minden lépésben, hiszen ugyanazokból a mezőkből próbálna mindig törölni az algoritmus. Amennyiben nem volt ilyen mező, akkor a függvény nem csinál semmit. Az algoritmus futása elején ezt a szabályt minden kezdeti kitöltött értékre lefuttatja.
2. Egy sorban, oszlopban és négyzetben minden számnak pontosan egyszer kell szerepelnie. Vagyis ha talál például egy olyan sort, amelyben egy értéket csak egyetlen mező vehet fel, akkor annak a mezőnek a lehetséges értékei közül törölhető minden más szám. A szabály megvalósítása a *CheckRowsColumnsAndBoxes* metódusban történik meg.

Minden szabály - köztük a később ismertetett killer sudokuhoz tartozók is - egy bool értékkel térnek vissza, attól függően, hogy sikerült-e valamely mező lehetséges érték lista elemeinek csökkentése.

Az algoritmus minden sikeres értéklista csökkentés után megvizsgálja, van-e olyan mező, amelynek már csak egy értéke lehet. Ez a *NewSolvedField* metódusban történik, amely ha talál ilyen mezőt, azon beállítja az értéket annak *SetValue* függvényével. Ezután ezt a mezőt visszaadja, hogy az első szabály ez alapján tudja további mezők lehetséges értékeit csökkenteni.

Abban az esetben, amikor egyik szabállyal sem sikerült semmilyen mező értékeit csökkenteni, a *FillFirstFieldRandomly* metódus hívódik meg. Ez a függvény megkeresi az első értékkel nem rendelkező mezőt (ha ilyet nem talált, az azt jelenti, hogy kész van a feladvány megoldása), majd értéknek beállítja az értéklistájának első elemét. Előtte azonban elmenti az aktuális állapotát a feladványnak. Így ha kiderül, hogy ez a szám mégsem jöhet abba a mezőbe, és a következő elérhető számot kell majd megpróbálni, vissza lehet majd állítani azt az állapotát a táblának, amikor ez a függvény meghívódott. A kitöltött mezőt átadva meghívja a megoldó függvényt, ami így egy rekurzív hívás. Ha ez a megoldó igazzal tér vissza, vagyis talált megoldást, akkor ez a függvény is visszatér. Ha így nem volt megoldás, a metódus visszaállítja az elmentett állapotba a sudoku feladványt, és a kérdéses mező következő lehetséges értékével próbálja meg a fent leírtakat. Ha már minden értéket megpróbált, akkor a függvény visszatér, jelezve, hogy az aktuális sudoku feladvány állapota nem megoldható.

Ez az egész algoritmus a *FillFields* metódusban van megvalósítva, aminek működése a lentebbi kódrészleten van bemutatva. Ezt a függvényt a *Solve* metódus hívja meg a kezdeti inicializálások után.

|  |  |
| --- | --- |
|  | while (true) |
|  | { |
|  | if (WrongAlready()) |
|  | return false; |
|  | if(ReduceByNewNumber(solvedField) || CheckRowsColumnsAndBoxes()) |
|  | { |
|  | solvedField = NewSolvedField(); |
|  | } |
|  | else |
|  | { |
|  | return FillFirstFieldRandomly(); |
|  | } |
|  | } |

Egy „végtelen” ciklusban (ami valójában nem lehet végtelen) történik a szabályok megvalósítása. A ciklus elején viszont van egy ellenőrzés. Itt olyan mezőt keres a program, aminek nincsen értéke, és a felvehető érték listája is üres. Ekkor nyilvánvalóan nincsen megoldás, így a függvény visszatér. Egyébként az algoritmus a leírtak szerint történik. Ha valamely szabálynak sikerült mezők értéklistáit csökkenteni, akkor megnézi, van-e olyan mező, amelynek már egyértelmű, melyik számot kell felvennie. Amikor már egyik szabály sem tudott csökkenteni, akkor hívja meg a *FillFirstFieldRandomly* függvényt, aminek a visszatérési értékével fog ez a függvény is visszatérni.

## Killer sudoku megoldó algoritmus

A killer sudoku megoldó osztályt, a *KillerSolvert* a *SudokuSolverből* származtattam le. Ennek az osztálynak nincsen új publikus függvénye. A sudoku megoldónál bemutatott *Solve* metódus az, amivel a killer sudokut is meg lehet oldani. Ezt direkt így terveztem, mivel a sudokunál bemutatott függvények és szabályok itt is érvényesek. A *SudokuSolver* osztályon belül azokon a helyeken, ahol a killer szabályok miatt plusz ellenőrzés kell, illetve a killer összegekre építő lista csökkentő szabályokhoz virtuális függvényeket vezettem be, amiket a *KillerSolverben* definiáltam. Most ezeket a plusz függvényeket, algoritmusokat mutatom be.

A killer szabályok tárolására a rekurzív megoldóhoz hasonlóan a *Dictionary* osztályt használtam, ahol a kulcs az összetartozó mezők listája, érték pedig azon mezők elvárt összege.

Az osztály legfontosabb függvénye *KillerReducingFromSum*. Ez paraméterként a dictionary egy elemét kapja, vagyis az összetartozó mezők listáját, és az összeget. A listában szereplő mezők lehetséges értékei közül törli ki azokat, amik a killer szabály miatt nem kerülhetnek egyik mezőbe sem. A 6.1-edik fejezetben már írtam erről részletesebben, több példán bemutatva, ezért itt csak egyet mutatok:

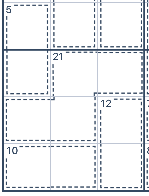


. ábra killer példa

Ezen a példán a két mező összegének ötnek kell lennie, ezért az algoritmus az ötöt, és az annál nagyobb számokat törli a mezők lehetséges értékei közül.

A *KillerSolver* osztály négy virtuális függvényét definiálja az ősosztálynak. Az *InitialReduce* még az algoritmus futásának legelején hívódik meg. Végigmegy a killer szabályokon és mindegyikre meghívja fentebb említett *KillerReducingFromSum* metódust, ezzel csökkentve a mezők lehetséges értékeinek számát. Az OtherKillerRules függvény a sudokunál definiált szabályokkal egy helyen hívódik. Ide lehet a killer osztályon belül plusz szabályokat definiálni a killer sudokura. Én a kezdeti redukció mellett még egy szabályt írtam, a *CheckSums* függvényt.

Ez a metódus végignézi az összes sort, oszlopot és négyzetet, és olyat keres köztük, ahol tudjuk nyolc szám összegét, így a kilencedik számot egyértelműen ki lehet találni.



. ábra Killer sudoku szabály

A fenti ábrán látható egy példa. Itt az alsó négyzet bal felső sarkában levő mezőnek 2-es értéket kell felvennie, mivel a többi mező összege 43, és tudjuk, hogy minden négyzetben minden szám egyszer szerepel, vagyis egy ilyennek az összege 45. Tehát 45-43=2 kapjuk a maradék mezőre.

A *ReduceKillerField* függvény minden egyes mező kitöltésénél hívódik meg. Itt a kitöltött mezővel egy listában szereplő mezők értékeinek esetleges csökkentése történik. Például a fenti ábránál kiderült, hogy a négyzet bal felső sarkában levő szám 2-es. Ezzel a mezővel egy killer szabályon belül egy másik mező van, ennek a lehetséges értékeit pedig a 2-es kitöltése után lehet csökkenteni. (3-ason kívül más nem kerülhet oda)

A *RefreshKillerValues* függvény a *FillFirstFieldRandomly* metódusban van meghívva. Ennél említettem, hogy ha a próbálgatáshoz választott számról kiderül, hogy nem megfelelő, akkor az elmentett állapotot vissza kell állítani. A *RefreshKillerValuesban* a *Dictionaryben* szereplő mezőket kell frissíteni, mivel a visszaállítást úgy oldottam meg, hogy új *Field* objektumokat hoztam létre. Így nem kellett a feleslegesen végrehajtott lépéseket visszafele is megcsinálni, vagyis azon értékeket visszaadni a mezők értékeihez, amik azután lettek törölve, hogy egy rossz számot választott az algoritmus az egyik mezőnek. Ez a megoldás úgy gondolom így lényegesen egyszerűbb volt.

## Tesztelési tapasztalatok

Mind a két megoldó algoritmust ugyanazokkal a feladványokkal próbáltam ki, mint amikkel a rekurzív algoritmusokat teszteltem. A sudoku megoldó hasonlóan gyors volt, szinte azonnal oldotta meg a feladványokat. A killer sudoku megoldó program viszont sokkal gyorsabb lett ezzel a módszerrel, mindegyik feladványt 1-2 másodperc alatt tudott megoldani.

Bár ebben a megközelítésben is alkalmaztam rekurziót, viszont a lehetséges értékek tárolásával, és azok csökkentésével egyértelműen gyorsabb megoldót lehetett készíteni. Természetesen több szabály definiálásával a rekurziós hívások számát még tovább lehetne csökkenteni. Azonban már ezen egyszerű szabályok implementálása sem volt egyszerű, és az így kapott megoldó viszonylag gyorsnak tekinthető, ezért már több szabályt nem kezdtem el kidolgozni.

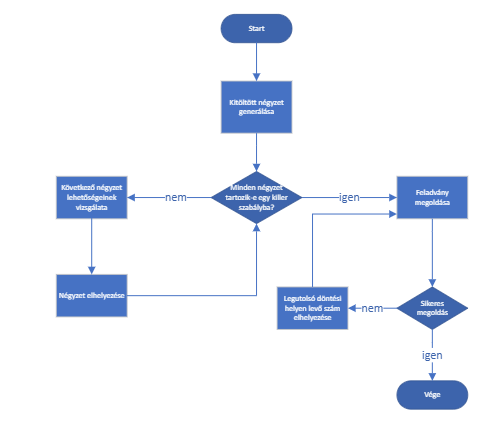
# Killer sudoku generálás

Killer sudoku generálása kicsit eltér a sudoku feladványok generálásától, hiszen itt a mezők értékei, illetve a kezdeti számok mellett szükség van a killer szabályok generálására is. Az algoritmushoz felhasznált megoldó program az előző fejezetben lett bemutatva, azonban ezen egy kicsit változtattam annak érdekében, hogy a generált feladványnak csak egyetlen megoldása lehessen.

Bevezettem a *SudokuSolver* osztályba egy Solutions változót, ami a megoldások tárolására alkalmas. (a kitöltött számok mely helyekre kerültek) Az összes megoldást viszont nem szükséges megkeresni, elég csak azt megtudni, hogy a feladványnak van-e egynél több megoldása. Így abban az esetben, ha egy megoldás megtalálásakor ez a *Solutions* lista már nem üres, tudjuk, hogy az eredeti feladványnak nem lesz egyértelmű megoldása. A *Solve* függvény emiatt hibát fog jelezni.

## Generáló algoritmus

A célom az volt, hogy olyan generálót készítsek, amely egyértelműen megoldható feladványt generál, és lehetőleg gyors is. Ebből a második rész nem teljesen sikerült, de előbb lássuk az algoritmust.



. ábra Generálás folyamata

Az algoritmus első lépése, hogy egy 9x9-es, sudoku szabályoknak megfelelően kitöltött táblát generál. Ez az 5.2-es fejezetben már ismertetett módszer alapján történik. A következő lépésben minden négyzetet elhelyez egy killer szabályban. Az ábrán ez a rész csak vázlatosan szerepel, így ezt a részét az algoritmusnak részletesebben is bemutatnám.

Minden mező három helyre kerülhet a szabályokban: a felette levő mezővel egy csoportba, a tőle balra levővel egy csoportba, vagy egy újabb csoportba. Ezek eldöntése néhány esetben egyértelmű, egyébként pedig random történik.

### Generáló algoritmus

### Tesztelési tapasztalatok

# Frontend

## Sudoku

## Killer Sudoku

# Irodalomjegyzék

1. https://github.com/giacomelli/GeneticSharp

Függelék