Ez egy órarend készítő program, ami általános és középiskolák számára készít órarendeket. A program működéséhez szükséges egy tantárgyfelosztás (melyik tanár, melyik osztályban, milyen tantárgyat, hány órában tanít), amit jelenleg egy txt fájlból olvas be. Ezek után a program genetikus algoritmus segítségével előállít egy órarendet. A genetikus algoritmus egy kezdeti véletlen populációból mutációk és kereszteződések segítségével állít elő jobb és jobb órarendeket, amíg végül megszületik a feltételeknek már teljesen megfelelő órarend.

Főbb metódusok leírása röviden

scanData() **(***Datascan* osztály**)**
Beolvassa az adatokat txt-ből, és létrehozza az összes *Lesson* és *Classes* példányt.

initPopulation(List<Classes> allTheClasses, int[] allClassesGrades)
(*GeneticAlgorithm* osztály)
Létrehozza a megadott méretű populációt (véletlen szerűen).

TimeTable.createRandomTimeTable(List<Classes> allClasses)(*Timetable* osztály)
Létrehoz egy órarendet véletlenszerűen. A létrejövő órarend osztály szinten helyes, de globálisan nézve tele van (tanár és tantárgy) ütközéssel.

calcFitness() *(Individual* osztály*)*
Kiszámolja egy órarend esetében, hogy mennyire jó az adott órarend. Az értéke 0 és 1 között van. Ha egy órarend eléri az 1-et akkor megtaláltuk a feltételeknek megfelelő órarendet.

getFittest(int offset) (*Population* osztály)
rendezi az Individual[] tömböt a fitness érték szerint csökkenő sorrendbe, és visszaadja a legjobb elemét az Individual[] tömbnek.

crossoverPopulation(Population population) (*GeneticAlgorithm)* osztály
Ez a metódus felelős a kereszteződésért, vagyis azért, hogy két szülő órarendből kikeverjünk egy gyerek órarendet. előbb szülő társat választunk egy órarendhez, majd véletlenszerűen vesszük az egyik évfolyam óráit az egyik szülőtől és a másik évfolyam óráit a másik szülőtől. És így előállítunk egy „gyerek” órarendet.

mutatePopulation(Population population) (*GeneticAlgorithm)* osztály
Ez a metódus felel a mutációért, vagyis azért, hogy egy órarenden belül apró változás jöjjön létre. A mutációnak három változatát használjuk. Két ütköző órát kicserélünk egymással az osztály órarendben, vagy egy ütköző órát kicserélünk egy véletlenszerű másik órával az osztály órarendben, vagy két teljesen véletlenszerű órát cserélünk ki egymással az osztály órarendben.

A program indítása után:
A program minden iterációnál kiírja a legjobb órarend fittness értékét. Ezért a program indítása után egy ideig csak a következő kiírásokat látjuk pl:

Best solution in 14 generations:
0.011111111

Ez azt jelenti hogy a 14. generáció után a legjobb órarend fittness értéke 0,011. A legjobb órarend fittness értéke folyamatosan növekszik és a program addig fut, amíg el nem éri az 1-et

valamelyik órarend. Ez körülbelül 100 iteráció alatt van meg….Előfordulhat, hogy már 50 iteráció alatt is lesz nyertes, de olyan is volt már hogy 300 iteráció felett állt le. Azt javasoljuk, hogy ha 200 iteráció alatt nem talál megoldást akkor állítsa le és indítsa újra. A genetikus algoritmus sajátossága, hogy két egymás utáni futtatás nem adja ki ugyanazt az órarendet.

Program részletes leírása

Főbb műveletek:

1. Genetikus algoritmus beállításai
2. Adat beolvasás
3. Populáció létrehozása
4. Genetikus algoritmus működtetése (mutáció, kereszteződés, fitness függvény számolás) addig amíg létre nem jön a feltételeknek megfelelő órarend.
5. A nyertes órarend kiírása

1. Genetikus algoritmus beállításai

Létrehozunk egy példányt a genetikus algoritmusból, és beállítódnak az alap paraméterek.
***populationSize*:** hány darabból álljon a populáció
***mutationRate*:** egy adott példány mekkora valószínűséggel mutálódjon egy iterációban
***crossoverRate*:** egy adott példány mekkora valószínűséggel kereszteződjön egy iterációban
***elitismCount*:** A legjobb valahány példányt nem engedi mutálódni. Ha ez a szám nulla, akkor konkrétan a legjobbat nem engedi. Ha pl 3, akkor a legjobb 4 nem mutálódik.
tournamentSize: mintavétel mérete majd a kereszteződéshez.

2. Adat beolvasás

scanData()
Az adatok beolvasása txt-ből történik. Először az osztályok nevei olvasódnak be a *classes.txt*-ből. Ebben a fájlben minden sor egy osztály nevet tartalmaz. Ebből létre is jönnek az osztályok példányai, ami kezdőértékként megkapja az osztály nevét és évfolyamát.

Majd következik a *classes_summary_wGroups.txt* beolvasása. Ez a fájl tartalmazza az egész iskola tantárgyfelosztását. Minden sor egy osztály egy tantárgyának az adatait tartalmazza. Nézzünk meg pár ilyen sort:

9a/0112,Matematika,4,Kovács Elemér,KE
9a/0223,Német nyelv,3,Maros Vanda,MV
9a,Biológia,3,Madaras Janka,MJ

A különböző adatok vesszővel vannak elválasztva egymástól. Az első adat tartalmazza, hogy melyik osztályhoz tartozik, és hogy mi az adott tanórának a groupId-je. A groupID a csoportbontáshoz kell, ha egy sor nem tartalmaz groupID-t akkor abban az órában az osztály egyben van. Második adat tartalmazza a tantárgynevet, Harmadik adat tartalmazza a tantárgy heti óraszámát. Negyedik adat a tanár nevét, Ötödik adat pedig a tanár monogramját.

GROUPID
A groupID az osztálynév és törtvonal után következő első 3 számjegy.
Első számjegy: Ha nem nulla akkor ez a tantárgy egy évfolyamközi bontású tantárgy. Ez esetben a második számjeggyel együtt kódolja, hogy pontosan melyik évfolyamóráról van szó.
Második számjegy: Ha nem nulla akkor ez egy csoportbontás fajtát kódol. Például a fenti példában a matematika az 1. csoportbontás szerint a német pedig egy második csoportbontás szerint van megtartva.
Harmadik számjegy: Bármilyen csoportbontás esetén azt jelöli, hogy a csoportbontáson belül hanyadik csoportról van szó. A fenti példában a matematika tárgynál Kovács Elemér a 1. csoportot tanítja, míg Maros Vanda németből a 2. csoportot tanítja.
Negyedik számjegy: Azt mutatja, hogy az adott óránál hány részre van bontva a csoport. Később ebből a számból tudja a program hogy egy csoport mellé hány másik csoportot kell keresnie. Ez a szám nem tartozik a groupID-hez, hanem rögtön beolvasás után leválasztásra kerül és egy külön változóban (*howManyPart*) tárolódik.

A beolvasás során minden sorból létrehozunk egy - egy Lesson példányt. És mindegyik nem évfolyam szintű órát hozzáadjuk a megfelelő osztály *Lessons* nevű listájához annyiszor ahány heti óra van a tantárgyból.
Az évfolyam szintű órákat a *GradeLessons* nevű listához adjuk hozzá, de csak egyszer.
Továbbá elmenjük az évfolyam óra első két számjegyéből álló kódját a *Relationships* osztályban a *classesofGradeLesson* nevű mapbe. A kód a kulcs, az érték az osztály indexe (az *allClasses* listából) Továbbá a *Relationships* osztály *gradeLessonsPerWeek* map-be is. Itt a kulcs továbbra is a kód, az érték pedig a tantárgyheti óraszáma

3. Populáció létrehozása (*initPopulation*)

Az első lépésben megadott populáció méret alapján legyárt annyi *Individual*-t, és eltárolja őket egy *Indivdual[]* tömbben. Az Individual egyik adattagja az órarend (*List<Lesson>[][] timetable*), továbbá itt számolódik ki minden iterációban az adott órarend fitness értéke, itt valósul meg konkrétan a mutáció.

TimeTable.createRandomTimeTable(List<Classes> allClasses)
Ezzela metódussal jön létre a véletlen órarend. több apró metódusra van bontva.

1. ***randomHoursPerDay(Classes classes)***
   Meghatározza egy osztálynak véletlenszerűen, hogy melyik nap hány órája legyen. A logika a következő: Kiszámolja a heti össz óraszámból a napi átlagot és ennél egyel kevesebbet illetve egyel többet enged meg. Például ha napi óraszám egy osztály esetében 6,4 akkor az osztálynak lehet olyan anpja hogy 5 órja van vagy hat vagy 7, de nem lehet 8 órája vagy 4 órájaegyik nap sem. Ez biztosítja azt hogy viszonylag egyenletes legyen az órák elolszlása az órarendben.

2. ***calculateSiteOfFreePeriod(int[] HoursPerDay)***
   Az előző metódus kimenetét várja bemenetként. A jelenlegi koncepció szerint naponta összesen a nulladik órával együtt egy osztálynak maximum 9 órája lehet egy nap. De ha egy osztálynak csak 6 órája van akkor azt jelenti hogy nincs 0., 7. és 8. órája. Ez a metódus előállítja azoknak az óráknak az indexeit majd a végső órarendben, ahol ezek a lyukak vannak, amikor nincs óra.
3. ***findPlacesOfGradeLessons(Relationships relationships, Set<Integer>[] siteOfReservedPlaces)***
   Végigmegyünk egy iterátorral a *Relationships*-ben letárolt *classesOfGradeLesson* map-en. Kiolvassuk sorban minden kódhoz a hozzá tartozó osztály listát, hogy melyik osztályokban van megvalósítva az az évfolyam óra. A kiolvasott osztálylistához készítünk egy *badTimeSlots* nevű Set-et, amiben összesítjük a kiolvasott osztályok már foglalt időpontjait. (ezek lehetnek lyukas órák vagy akár már egy korábban hozzáadott évfolyam óra is). Majd véletlen számot generálunk a visszamaradt órák közül, annyiszor amennyi az adott évfolyam óra heti óra száma. A heti óraszámot is a *Relationships*-ből kérjük le a *gradeLessonsPerWeek* map-ből a kód alapján. A generált helyet lementjük a *Relationships* – ben a *placeOfGradeLessons* map-be. A helyek itt egy listát alkotnak ami az érték, a kód pedig a kulcs. Végül a helyet hozzáadjuk a foglalt helyk listájához is, hogy következő alkalommal már ne lehessen beválasztani.
4. ***lessonsInTimeTable(List<Lesson> lessons, List<Lesson> gradeLessons, List<Integer> siteOfFreePeriod, Relationships relationships)***
   Ez a metódus felel azért hogy ténylegesen létrejöjjön egy osztály random órarendje. Először létrehoz egy Tömböt, aminek az elemei List<Lesson> -ok. Majd a korábban elmentett lyukasóra helyek alapján beteszi a megfelelő helyekre a lyukasórákat (helyben példányosítja is őket FreePeriod néven)
   Ezután végigmegy a *gradeLessons* listán és mindegyik évfolyamórának kiolvassa a helyét a Relationships placeOfGradeLessons map - ből és beteszi a megfelelő helyre az évfolyamórát.
   Végül a maradék órákon megy végig amit bemenetként megkap listaként (*lessons*). Ezt úgy végzi el, hogy végig megy az eddig félig feltöltött tömbön. És minden helyre ami még null betesz egy órát a listából. Ha a beteendó óra egész osztályos óra akkor ez meg is van és keresi a következő null értéket a tömbben. Ha egy osztályszintű csoportbontásos óra, akkor pedig tesz mellé annyi órát, hogy összesen annyi legyen amennyi részre az osztály ebben a pillanatban bontva van. Természetesen figyelve arra, hogy ugyanolyan csoportbontás szerint órát tegyen bele, de különböző csoportot.

Amint legenerálódott a véletlen órarend el is mentődik az Individualban, és következő lépésként lefut a *calcFitness()* metódus (*Individual osztály*).

calcFitness()

Ez a metódus egy 0 és 1 közötti értékkel tér vissza, és a visszatérési érték meghatározza, hogy az adott órarend mennyire jó. Ha egy órarendre ez az érték 1, Akkor megtaláltuk azt az órarendet ami megfelel a betáplált feltételeknek.
A korábban létrehozott véletlen órarendben két féle hiba fordulhat elő. Osztályszinten az órarendek helyesek. Minden osztálynak annyi órája van amennyinek lennie kell, viszont előfordulhat, hogy egy napra 4 angol óra is kerül. Ez nem szerencsés. Szeretnénk, ha egy napon

belül nem lenne kétszer ugyan az az óra. Ez esetben egy napon belül kétszer is szerepel ugyan az a tantárgynév. A másik előforduló probléma, hogy egy tanár egy idő pillanatban több osztályhoz is bevan írva. Ezeket a problémákat ütközésnek nevezzük. És az órarend annál jobb minél kevesebb ütközés van benne. Ha nincs ütközés akkor megvan a feltételeknek megfelelő órarend. A végeredményként kapott fitness értéket a következőképpen határoztuk meg:

$$fitness\,érték = \frac{1}{1 + \text{ü}tközések\ száma}$$

A *calcFitness()* metódus gyakorlatilag végigmegy az órarenden és megszámolja, hogy hány ütközés található benne. Közben az ütközések indexeit eltárolja, mert később a mutációnál ez még kelleni fog.

4. Genetikus algoritmus működtetése (mutáció, kereszteződés, fitness függvény számolás) addig amíg létre nem jön a feltételeknek megfelelő órarend.

getFittest(int offset) metódus (*Population* osztályból)
rendezi az Individual[] tömböt a fitness érték szerint csökkenő sorrendbe, és visszaadja a legjobb elemét az Individual[] tömbnek.

crossoverPopulation(Population population) metódus a *GeneticAlgorithm* osztályból
Ez a metódus felelős a kereszteződésért, vagyis azért, hogy két szülő órarendből kikeverjünk egy gyerek órarendet.
Végigmegyünk a teljes populáción és egyesével eldöntjük hogy az adott órarend kereszteződjön-e vagy ne. (Ezt egy véletlen szám generátorral döntjük el.) Ha kereszteződik, akkor hívjuk ezt a példányt a továbbiakban anyának. Kell egy másik órarend is (apa)

1. ***selectParent(Population population)*** metódus
   véletlenszerűen ki választunk valamennyi (t*ounamentSize* - a genetikus algoritmus elején lehet beállítani) órarendet, és ezekből a legjobbat (*getFittest()*)választju ki apának.

A kiválasztott két szülőből összesen 10 gyereket készítünk el.

2. ***breadOffspring(Individual parent1, Individual parent2, int[] allClassGrades)*** metódus az Individual osztályból:
   Azért felelős, hogy létrehozzon egy gyerek órarendet a szülőkből. A gyerek készítésénél az egyik évfolyam osztályainak órarendjeit vesszük az egyik órarendből és a másik évfolyam osztályainak órarendjeit vesszük a másik órarendből.

A gyerekek közül vesszük a legjobbat és az anya és a gyerek közül azt tartjuk meg amelyik jobb fitness értékkel rendelkezik.

mutatePopulation(Population population) metódus a *GeneticAlgorithm* osztályból
Ez a metódus felel a mutációért, vagyis azért, hogy egy órarenden belül apró változás jöjjön létre. Végigmegyünk a teljes populáción és egyesével A legjobbakat(*ElitismCount*) nem engedjük mutálni. A mutációnak három változata van. És akár mindhárom mutáció

bekövetkezhet egy órarendnél, minden osztálynál, vagy egyik sem, itt is véletlenszám generátor határozza meg, hogy bekövetkezik-e valamelyik mutáció vagy nem.

1. ***mutateTwoCollisions(int classID, int[] allClassGrades*** az Individual osztályból
   Két ütközéses órát kicserélünk az osztály órarendben az osztályórarenden belül. ebben a metódusban kiválasztódik véletlenszerűen a korábban eltárolt ütközések indexeiből az a két óra ami ki fog cserélődni egymással.
2. ***mutateOneCollusion(int classID, int[] allClassGrades)*** az Individual osztályból
   Egy ütközéses órát kicserélünk egy random órával az osztályórarenden belül. ebben a metódusban kiválasztódik véletlenszerűen a korábban eltároltütközések indexeiből az egyik ütközéses óra valamint teljesen véletlenszerűen a másik óra, ami egymással kicserélődik majd.
3. ***mutateRandom(int classID, int[] allClassGrades)*** az Individual osztályból
   Kicserélünk teljesen véletlenszerűen két órát az osztályórarenden belül., vagyis egy random szám generátorral kiválasztjuk azt a két órát ami egymással helyet cserél majd.

A tényleges kicserélést pedig a következő metódusok bonyolítják le:

1. ***mutate (int classID, ind dayHour1, int dayHour2, int[] allClassGrades)***
   Előállhat hogy a kicserélni kívánt órák közül az egyik évfolyam óra. és ilyenkor nem csak ezt a két órát kell kicserélni, hanem az évfolyam órához tartozó többi osztály óráját is.
   4 eset játszódik itt le:
   a) Mind két óra amit cserélni akarunk évfolyam óra,
   b) Az első óra évfolyam óra, a másik nem
   c) Az első óra nem évfolyam óra a másik igen
   d) Egyik óra sem évfolyam óra.
2. ***findOtherGroup(int classID, int[] allClassGrades)***
   Az előző metódus első három esetében azonosítani kell az évfolyam többi osztályait, hogy ott is ki lehessen cserélni az órákat, ez a metódus ezt végzi.
3. ***swap((int classID, ind dayHour1, int dayHour2)***
   Elvégzi a két óra kicserélését az osztály órarenden.

A mutáció után az új és a régi közül a jobbat tartjuk meg.

5. A nyertes órarend kiírása

A nyertes órarend megtalálása után a ciklus leáll és a konzolon megjelenik a nyertes órarend.

Készítette:

Zöld Péter
Kovács Roland
Simon András Péter