Ülőhelykiosztási probléma vizsgálata optimalizálási és mesterséges intelligencia módszerek segítségével

**Hallgatók:**

* Egyed Vince (KSWOV1)
* Kertész Krisztián Levente (GDJK9Q)
* Puskás András (FO2GGY)

**Konzulens oktatók:**

* Dr. Fogarassyné Dr. Vathy Ágnes
* Dr. Süle Zoltán

Tartalomjegyzék

[Feladat leírás 1](#_Toc72934221)

[1. Tervezés és előkészítés 2](#_Toc72934222)

[1.1. COVID-19 egészségügyi szabályok 2](#_Toc72934223)

[1.2. Választott fejlesztési nyelv és környezet 2](#_Toc72934224)

[1.3. Felhasználó felület 3](#_Toc72934225)

[1.4. Adatok kezelése 4](#_Toc72934226)

[1.5. Adatok tárolása 5](#_Toc72934227)

[1.5.1. Adatbázis kiválasztása 5](#_Toc72934228)

[1.5.2. Adatstruktúra tervezése 6](#_Toc72934229)

[1.6. Verzió követés 8](#_Toc72934230)

[2. Implementáció 9](#_Toc72934231)

[2.1. Felhasználó felület 9](#_Toc72934232)

[2.1.1. Táblázat 10](#_Toc72934233)

[2.2. Adatok kezelése 12](#_Toc72934234)

[2.2.1. Adatstruktúra 12](#_Toc72934235)

[2.2.2. Utils 14](#_Toc72934236)

[2.3. Adatok tárolása 16](#_Toc72934237)

[3. Gépi megoldó algoritmusok 19](#_Toc72934238)

[3.1. Mohó algoritmus 19](#_Toc72934239)

[3.2. Genetikus algoritmus 21](#_Toc72934240)

[3.3. LP megoldó 24](#_Toc72934241)

[3.4. Gépi megoldók összehasonlítása 27](#_Toc72934242)

Feladat leírás

A feladat egy olyan grafikus alkalmazás és hozzá tartozó módszertan kidolgozása, amely lehetővé teszi különféle eseményekre történő helyfoglalási folyamat optimalizálását a COVID-19 egészségügyi szabályok betartása mellett.

Az alkalmazásban lehetőségünk van dinamikusan definiálni a termet, illetve a székek elrendezését is. A terembe a felhasználó tudja elhelyezni az ülőhelyeket megfelelő előre definiált egészségügyi szabályoknak megfelelően. Az alkalmazás továbbá optimalizálás vagy mesterséges intelligencia segítségével is be tudja rendezni a termet a maximális kihasználtság érdekében.

**Részfeladatok:**

* Programozási nyelv, platform kiválasztása
* Felhasználói felület elkészítése
* Dinamikus terem elkészítése
* Egészségügyi szabályoknak megfelelő kritériumok programozása
* Felhasználói interakciók mögöttes logikájának programozása
* Optimalizálás vagy mesterséges intelligencia segítségével a feladatmegoldó elkészítése
* Alkalmazás dokumentálása

# Tervezés és előkészítés

## COVID-19 egészségügyi szabályok

A járványhelyzet miatt kialakult egészségügyi szabályok a színházakra és mozi termekre is egyaránt vonatkoznak, ennek okán szükséges volt megismernünk a járvány miatt életbe hozott szabályokat. A termekre vonatkozó szabályozás úgy korlátozza az ülésrendet, hogy egymás mellett csak az egy csoportba tartózkodó személyek foglalhatnak helyet. Két csoport között legalább egy széknek üresen kell maradni.

Ezeket a szabályozásokat figyelembe véve a program elkészítése során is figyelembe kellett venni ezeket a szabályozásokat és ennek megfelelően szabályrendszert kell definiálni az alkalmazásba.

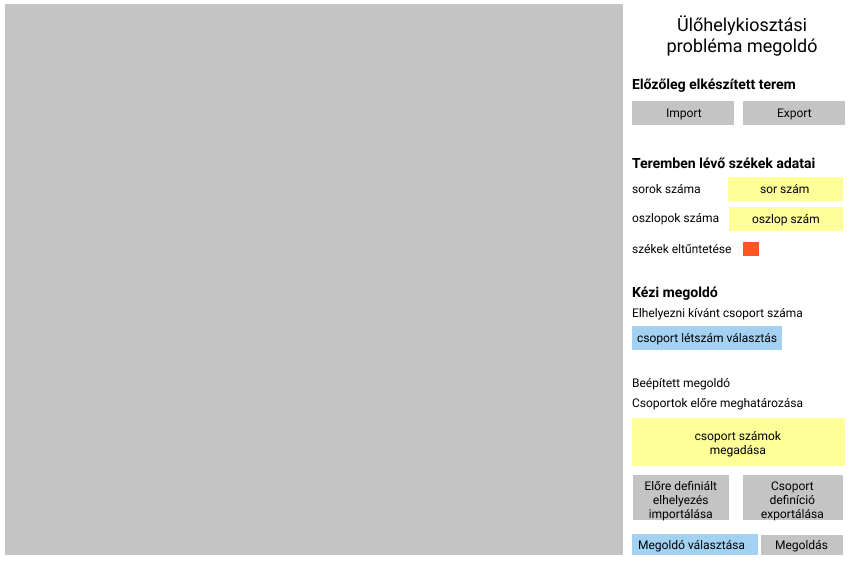
## Választott fejlesztési nyelv és környezet

Fejlesztési nyelvként java-t választottunk, mert komfortosan tudunk benne programozni. A fejlesztést IntelliJ IDEA-ban végeztük. A fejlesztési környezetben könnyedén tudtuk elkészíteni a Maven projektünket, amelyhez több könyvtárat is csatoltunk.

A csapatmunka miatt GitHub-on készítettük el a projektet. Az IntelliJ IDEA könnyű és letisztult beépített verzió követője miatt gördülékenyen tudtunk egymás mellett dolgozni a projekten.

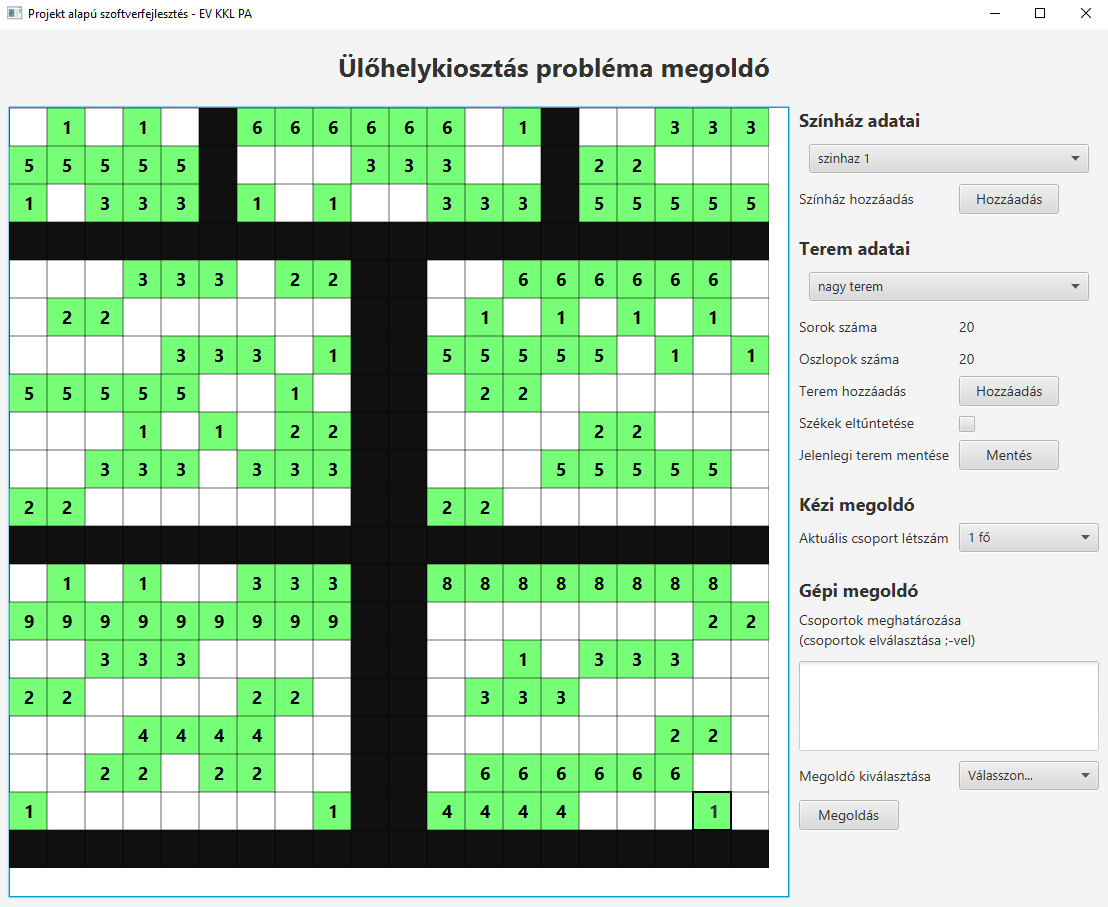
## Felhasználó felület

A felhasználó felület tervezése során törekedtünk, hogy érthető és könnyen átlátható alkalmazást készítsünk. A felhasználói felület megtervezésére egy online tervező alkalmazást használtunk. A tervezőben készült felületet az **1. ábra** szemlélteti. Az ábrán bal oldalt lévő szürke rész a teremet szemlélteti.



**1. ábra:** felhasználói felület terv (<https://www.figma.com>)

A fejlesztés közben az egyeztetések folyamán a felhasználó felület megváltozott, azonban az **1. ábrán** látható elrendezés megmaradt. Az elkészült felhasználó felületet a **2. ábra** szemlélteti működés közben. Az ábrán látható, hogy a jobb oldali sávban történtek változások a tervezethez képest. Ezek a változásokra amiatt volt szükség, mert az eltárolt termek is könnyedén tudja ezen a módon kezelni az alkalmazás.



**2. ábra:** elkészült felhasználói felület

## Adatok kezelése

A model osztályaiban implementáltuk az egyeztetett adatstruktúrát. Az alkalmazásban alapvetően Színházakkal (mint épületekkel), Termekkel és azokon belül Ülésekkel dolgozunk. Ezekhez később hozzávettük a Rendeléseket, melyek kezdetben csoportokat hivatottak reprezentálni, de elképzelhető, hogy a fejlesztés további fázisaiban másképp oldjuk meg ezt a problémát.

Továbbá robosztus kód alkotásának reményében több enum-ot és utils funkciót, valamint egy egységes adatfelhasználást lehetővé tevő interface-t is létrehoztunk. Utóbbit még nem egészen sikerült integrálni a rendszerünkbe, de dolgozunk rajta.

Ez azt a célunkat szolgálná, hogy ne csupán adatbázisból, vagy az alkalmazás segítségével tudjunk újabb adatokat generálni és tárolni, hanem import-export eljárások segítségével akár több forrással és formátummal is dolgozhassunk.

## Adatok tárolása

Az alkalmazás tervezése során felmerült az igény a már elkészített színháztermek és színházak mentésére és betöltésére ezért szükségessé vált valamilyen adatbázis technológia felhasználása. A megbeszélések során felmerült, hogy az adatokat nem csak az adatbázisba kellene elmenteni hanem azokat ki is lehessen exportálni és be is lehessen tölteni. Mivel az alkalmazás json fájlba menti ki az adatokat ezért abban egyeztünk meg, hogy akkor az adatbázis is json fájlokkal dolgozzon. Ezenfelül szükségesnek éreztünk egy a hagyományosnál rugalmasabban kezelhető adatszerkezetet is. Emiatt a két dolog miatt választottuk a hagyományos SQL adatbázisok helyet a rugalmasabb NoSQL adatbázisokat.

### Adatbázis kiválasztása

Az első felmerülő NoSQL adatbázis a Firebase volt mivel ezzel az adatbázissal már voltak előzetes ismereteink. Az elképzelés az volt, hogy ha android alkalmazások fejlesztésére használható volt, akkor nyilván sima asztali java alkalmazást is lehet vele fejleszteni, viszont miután áttekintettünk különböző fórumokat és a Firebase hivatalos dokumentációját világossá vált, hogy a Firebase nem támogatja az asztali java alkalmazás fejlesztést.

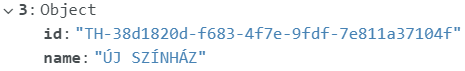
A második felmerülő NoSQL technológia a MongoDB volt. Ennek az adatbázisnak az volt a nagy hátránya, hogy a csapatból eddig még nem volt senkinek előzetes ismerete vele. Viszont mivel a MongoDB egy nagyon népszerű technológia a NoSQL adatbázisok között, ezért viszonylag sok oktató anyag és példakódállt rendelkezésre, amik segítségével a betanulási időszak viszonylag gyors volt.

### Adatstruktúra tervezése

Az adatbázis struktúra tervezése azzal kezdődött, hogy megvitattuk milyen adatokat kell eltárolni az adatbázisba. Az első megbeszélés után megállapítottuk, hogy kettő részbe lehet szedni az eltárolandó adatokat. Ez a kettő egység a színház adatai és a termek adatai külön-külön. Először a színházak adatainak tervezését kezdtük el mivel az lényegesen egyszerűbbnek tűnt, mint a termek adatainak összegyűjtése.

A színházak tervezésénél csak olyan adatokat tároltunk el, amik relevánsak az alkalmazás szempontjából, tehát például a színház címét nem tároljuk, viszont a színház nevét igen. Emiatt a színház gyűjteményben (SQL-ben tábla) csupán két adat kerül eltárolásra minden színház esetén. Ezek a következőek:

* id: itt tárolódik el a színház egyedi azonosítója. Minden azonosító kötelezően a „TH-” karakterekkel kezdődik. ezzel jelezve, hogy színház azonosítóról van szó.
* name: itt tárolódik el a színház neve. Ez a mező tartalmazhat betűket, számokat, illetve whitespace karaktereket is.



**3. ábra:** Példa egy színházra

A második egység a színházterek tervezése volt. Ebben az esetben is csak a releváns adatokat tároltuk el. Ezen felül nem kerülnek mentésre az elkészített ülőhelykiosztások sem mivel azokat az alkalmazás menet közben generálja le. Ha a fejlesztés későbbi szakaszában szükségessé válik az ülőhelykiosztás mentése is akkor valószínűleg egy új gyűjtemény hozunk hozzá létre, ami tartalmaz majd egy idegen kulcsot, ami a szoba azonosító lesz, amivel akár egynél több kiosztás eltárolására is lehetőség lehet. Viszont a fejlesztés jelen állapotában ezzel még nem kellet számolni. Az szobákat tároló gyűjtemény neve „Room” lett. Amin belül a következő adatokat tároljuk:

* id: A színházi terem egyedi azonosítóját tárolja. Az azonosító kötelezően az „RO-” karakterekkel kezdődik, ami a szobára (room) utal.
* theater\_id: A színház azonosítója, amiben a terem megtalálható.
* name: A terem nevét tárolja. Tartalmazhat: betűket, számokat és whitespace karaktereket.
* column\_num: A teremben lévő egy sorban lévő székek maximális száma.
* row\_num: A teremben lévő széksorok maximális száma.
* rows: tömb, ami a székek állapotát tárolja soronként. A székek kapcsoszárójellel vannak elválasztva egymástól. A lehetséges állapotok:
  + 0: Üres szék
  + 1: Foglalt szék
  + 2: Nem használható vagy nem létező szék.

A képen szöveg látható

Automatikusan generált leírás

**4. ábra:** Példa egy színházteremre

## Verzió követés

A csapatban való munkához git verzió követőt használtunk. GitHub-on keresztül hoztuk létre a projektünket és minden egyes csapattagunkat hozzáadtuk a projekthez.

A projektünket könnyedén tudtuk ezen keresztül menedzselni és az egymással párhuzamos munka sem okozott gondot.

Github repository: <https://github.com/asdsajt/seat_allocation_problem>

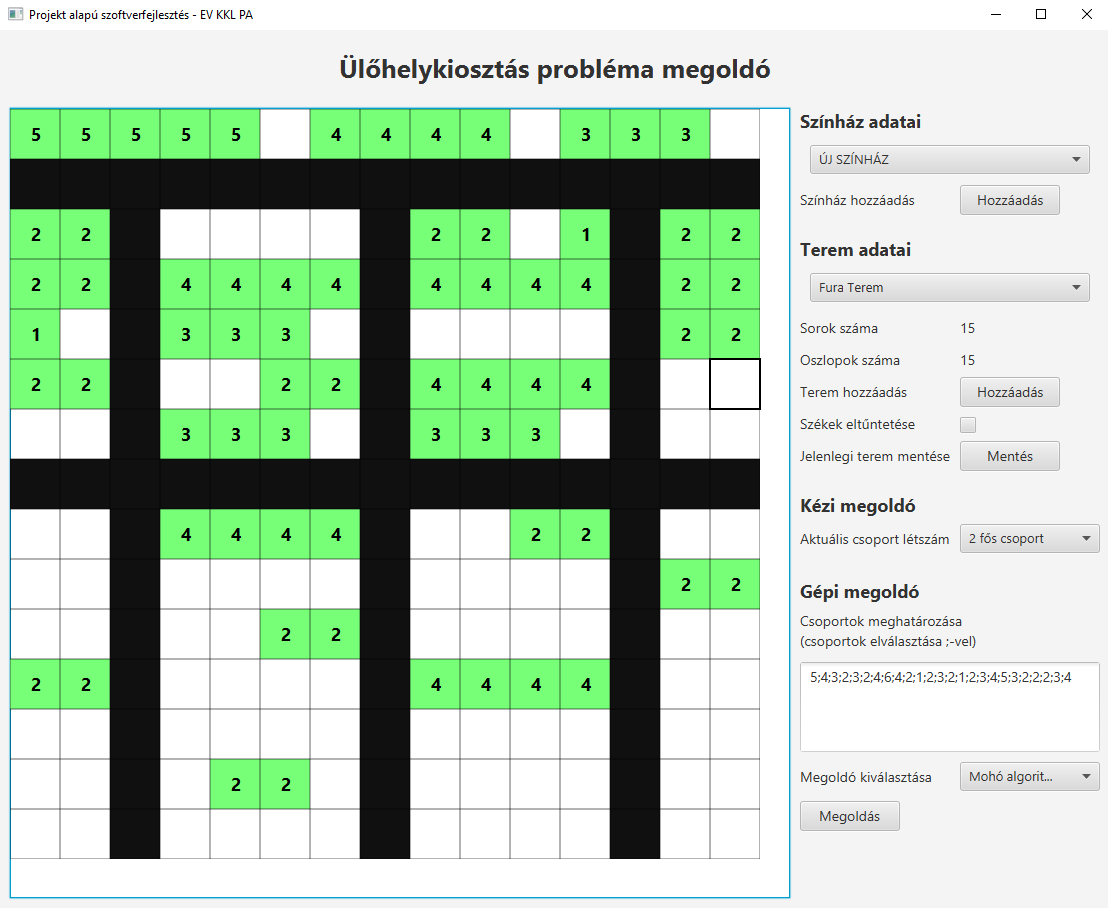
# Implementáció

## Felhasználó felület

A projekt egy nagy megjelenítő felületre épül, amelyen a felhasználó tudja végezni a teremmel kapcsolatos beállításokat, ülőhelyfoglalásokat. Ezen a felületen kívül még az alkalmazás tartalmaz három felugró ablakot, amely a mentésre és az új terem, színház felvételére szolgál.

A felhasználói felület elkészítésénél törekedtünk arra, hogy egy könnyen átlátható, intuitív felületet készítsünk el.

A projekt felhasználói felülete JavaFX segítségével készült el, ezen kívül egy Maven könyvtárat is használtunk a ControlsFX-t. A ControlsFX-ben található SpreadSheetView-val készült el a bal oldalon található táblázat. Nem használtuk a JavaFX-ben lévő beépített táblázatot, mert a SpreadSheetView-t könnyebben lehet kezelni és dinamikusabb, mint a JavaFX-ben lévő TableView.



**5. ábra:** felhasználói felület

### Táblázat

A felhasználó felület bal oldalán található táblázat képzi a program legnagyobb részét. Ezzel a táblázattal történnek a felhasználói interakciók és a beállítások is ezt a táblázatot módosítják. Ennek okán a táblázat sok eseményt, függvényhívást dolgoz fel.

#### Kattintás figyelők

A táblázathoz két kattintás figyelő esemény lett implementálva, amelyek elvégzik a táblázattal kapcsolatos felhasználói interakciók kezelését.

Az egyik kattintás figyelő segítségével a táblázatban lévő székeket lehet törölni, illetve visszaállítani. Erre amiatt volt szükség, hogy a felhasználó testre tudja szabni a termet, ki tudja jelölni, hogy hol nincsenek székek, azaz a lépcsők és a bejáratok helyét.

A másik kattintás figyelő komplex, mert az kezeli a felhasználó általi ültetéseket. A kattintás figyelő több kritérium alapján dönti el, hogy az aktuális cellába be lehet-e helyezni a felhasználó által választott csoportot, vagy sem. Az algoritmus tartalmazza, hogy két különböző csoport között minimum egy ülőhelyet ki kell hagyni a COVID-19 egészségügyi szabályainak megfelelően. Továbbá figyeli, hogy az aktuális helyre befér-e a csoport vagy sem, ebben az esetben az algoritmus figyeli, hogy jobbra, illetve balra mekkora távolság van. Figyelembe veszi, hogy egy másik csoport vagy törölt ülőhely van-e a legközelebb, mert ez is befolyásolja az algoritmus számolását. Amennyiben az algoritmus nem megfelelőnek találja a kiválasztott ülőhelyet a csoport letételéhez úgy hibaüzenet ad a felhasználó számára.

#### Színház adatai

A felhasználói felület jobb oldalán lévő sávja több kisebb részre van osztva, így a felhasználó az egyes alcímeknek köszönhetően könnyedén tud navigálni.

Az első ilyen rész a színház adatait tartalmazza. Ebben a egységben ki lehet választani már meglévő színházakat, illetve hozzá is lehet adni új színházat is a programhoz.

#### Terem adatai

A következő jobb oldali rész egységben a terem adatai szerepelnek. Ebben a részben ki lehet választani a kiválasztott színházhoz lévő termek közül azt, amelyet szeretnénk betölteni a programba. Amikor egy terem kiválasztásra kerül, akkor a program frissíti a jobb oldalon lévő táblázatot a terem méretei alapján, továbbá megjeleníti a törölt ülőhelyeket is.

A terem kiválasztása után a jobb oldalon is megjelennek a terem méretei szolgáló adatok is. Ezeken a felhasználó nem tud módosítani. Kivéve, ha új termet ad hozzá a felhasználó, mert a felugró ablakon be tudja állítani a terem méreteit.

A székek eltűntetésével lehet törölni az ülőhelyeket, így lehet a teremben lévő közlekedésre alkalmas részeket kijelölni. A törölt ülőhelyekre nem lehet már ülőhelyet foglalni.

A rész végén lévő mentés gombbal a felhasználó a terem jelenlegi állapotát tudja menteni. Mentésre kerülnek a törölt székek is, viszont a kiválasztott ülőhelyek nem.

#### Kézi megoldó

A kézi megoldó részen tudja a felhasználó kiválasztani, hogy hány főből álló csoportot szeretne elhelyezni a teremben. A táblázatba való elhelyezésnél a csoport létszáma is elhelyezésre kerül a kiválasztott cellán.

#### Gépi megoldó

A jobb oldal utolsó részén a gépi megoldó szerepel. A gépi megoldó első részén meg tudja adni a felhasználó, hogy milyen csoportokat szeretne elhelyezni a teremben.

A felhasználónak lehetősége van kiválasztani, hogy milyen megoldóval szeretné elhelyeztetni az adatokat a teremben. A megoldó kiválasztása után a „Megoldás” gomb megnyomása után megjelenik a táblázaton a megoldó által készített megoldás. Továbbá egy felugró ablak jelzi, hogy a megoldó sikeresen elhelyezte-e az összes előre megadott csoportot a táblázatban vagy nem sikerült neki.

## Adatok kezelése

### Adatstruktúra

A modelt reprezentáló adatok ismertetésével kezdem, de akár többször is előfordulhat a utils package-re és osztályaira történő hivatkozás. Ezekről később ejtek szót.

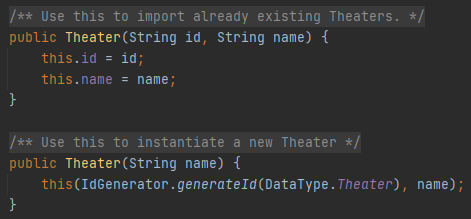
#### Theater

A színház osztálynak kettő adattagja van, az **azonosító** és a **név**.

Futás során két konstruktora közül a másodikat használjuk példányosításhoz – a többi id-vel ellátott osztályhoz hasonlóan. Ennek oka, hogy így a utils IdGenerator osztálya egységes és unique azonosítóval inicializálhatja az új objektumot.

Importálás és adatbázisból történő előhívás esetén viszont az első konstruktort alkalmazzuk, mivel itt biztosított a korábban eltárolt azonosító validitása.

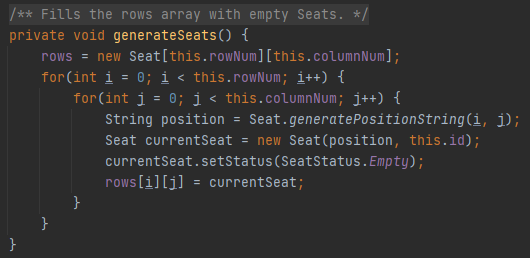
Az alkalmazás futása során az Id-t nem, de a nevet továbbra is módosíthatjuk.



**6. ábra:** A Theater osztály konstruktorai

#### Room

A szobákat **id** mezőjükkel azonosítjuk és **theaterId**-jükkel kötjük (lazán) a színházhoz amiben található. Továbbá elláttuk egy **név**vel és méreteit is számontartjuk **sorszám** (rN) és **oszlopszám** (cN) formában. Ezen méretparaméterek segítségével példányosításkor létrehoz egy rN\*cN-es tömböt amit feltölt üres ülésekkel. Az **üléstöm**böt nem, de az ülések státuszát minden további nélkül módosíthatjuk az applikáció futása során.

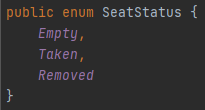


**7. ábra:** A szobák sorainak feltöltése üres ülésekkel

#### Seat

Az ülés konstruktorában meghatározuk, hogy melyik szobában (**roomId**) hol (**position**) – a megadott szoba tömbjének [row][column] koordinátái – találjuk.

Ezen felül (jelenlegi elképzelés szerint), ha valaki az alkalmazáson belül egy ülőhelyre kattint, **orderId** és **status** mezők módosítására nyílik lehetősége. Míg a redelés azonosító egy egyszerű String referencia (az Order object-re), a státusz-t a SeatStatus enum-ból választhatja ki a felhasználó. Ez a szabad (Free), a foglalt (Taken) és a használaton kívüli (Removed) értékeket veheti fel.



**8. ábra:** A SeatStatus enum lehetséges értékei

#### Order

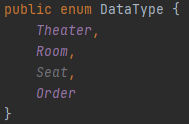
Egy kísérleti osztály – lehetséges, hogy teljesen kivesszük a programból – ami a csoportosításokat próbálja megvalósítani úgy, hogy egy (egyszeres vagy többszörös) kijelöléshez egy **id**-t rendel. Az Orderben tároljuk az szobát amire vonatkozik (**roomId**), a szobán belül pedig a kijelölt szék(ek) orderId mezőjét tesszük egyenlővé az Order id-jével.

### Utils

A model.utils több package és osztály együttese, melyet a Model programozhatóságának és olvashatóságának egyszerűsítése érdekében hoztam létre. Továbbá itt találhatóak az importerek, exporterek és az adatok belső tárolására használt InputData osztály. Ezeket a további fejlesztés során saját mappaszerkezetekbe fogom rendezni.

#### Utils.enums

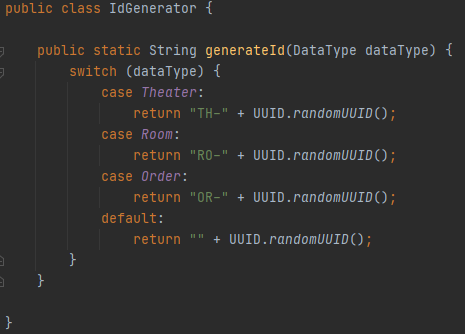
Ide tartozik az ülőhelyeknél említett SeatStatus, valamint a – jelenleg csupán Id generálásra szolgáló – DataType enum, melynek elemei megegyeznek az adatstruktúra osztályainak elnevezéseivel.



**9. ábra:** DataType enum

#### Utils.general

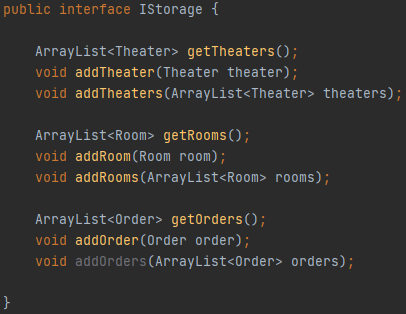
Általános felhasználású osztályok, függvények. Ide tartozik az IdGenerator.



**10. ábra:** IdGenerator, ahogy DataType szerinti unique ID-t generál

#### Utils.interfaces

Jelenleg csak az IStorage interface-t hoztam létre, ezt annak érdekében, hogy egységesítve lehessen hozzáférni a beolvasott adatokhoz, eltekintve attól, hogy éppen milyen formában tároljuk őket (adatbázis, InputData, stb.).



**11. ábra:** IStorage interface

## Adatok tárolása

Ahhoz, hogy az adatbázis json formátumú fájljaiból az alkalmazás számára értelmezhető osztályok legyenek szükség volt egy adatbázis kezelő és átalakító osztályra. Az említett osztály a DatabaseHandler nevet kapta. Az osztály feladata az adatbázisból lekérni a szükséges adatot tartalmazó json fájlt majd a json fájlt átalakítani az alkalmazás által értelmezhető osztállyá. Mindezt lehetőleg úgy, hogy az alkalmazás bármely pontjáról bármilyen adat lekéréséhez elég legyen egy gyors függvényhívás.

Az előzőleg egyeztetett szükséges funkciók a hagyományos lekérés, módosítás, törlés voltak mind a termek mind a színházak esetében. Az első lépés a függvények megírásához az volt, hogy az eddigre már elkészített üres adatbázist feltöltsem minta adatokkal. Ez kettő színházat és minden színházhoz egy-egy termet jelentet. Az első komolyabb feladat az első lekérés megírása volt, ami annyit csinált, hogy lekérte az összes termet az adatbázisból és kimenetként visszaadott egy tömböt, amiben a lekért termek szerepeltek már osztállyá alakítva.

A képen szöveg látható

Automatikusan generált leírás

**12. ábra:** Szobák lekérése az adatbázisból

A getRoom függvény nagyvonalakban két részre osztható az első részben megtörténik a lekérés míg a másodikban megtörténik a json 🡪 Room átalakítás. A MongoDB-ből való lekéréshez szükség van egy conection string-re csakúgy, mint a hagyományos SQL rendszerek esetén. Ezt követően ezt a string-et oda kell adni a MongoClient-nek, ami létrehozza a kapcsolatot az adatbázis szerverrel. Ezután már csak pontosítani kell, hogy melyik adatbázist használja és hogy melyik gyűjteményből kell kiolvasni az adatokat. A MongoDB esetében egy gyűjteményben akármennyi json fájl lehet, viszont ebben az esetben csupán egy json fájlt használunk, amit a fájl azonosítójának megadásával lehet beállítani.

A függvény második részében a lekért jsonObject-et felhasználva végigmegyünk az összes szobán és minden szoba esetében átmásoljuk a json fájlból az adatokat a Room osztályba. Miután az általános adatok kiolvasásra kerültek elkezdődik a székek feldolgozása is, ami során a széksorok string-jeit szétbontjuk önálló székekké, amiknek beállítjuk az állapotát a SeatStatus enum használatával, majd hozzáadom az aktuális szobát a listához. Miután az összes szoba feldolgozásra került átalakítom a listát tömbé majd visszatérek az értékével.

Ennek a függvénynek a mintájára létrehoztam egy másik függvényt, ami a színházakat kéri le. A két függvény kódja nagyban megegyezik csak a változók nevei, illetve az adatbázis csatlakozásnál megadott értékek térnek el.

Ennek a folyamatnak a másik oldala amikor egy osztály kellet visszaalakítani az adatbázis által értelmezhető json formátumba. Ennek a szemléltetésére talán a convertRoomToObject függvény a legalkalmasabb.

A képen szöveg látható

Automatikusan generált leírás

**13. ábra:** Osztály json formátumba alakítása

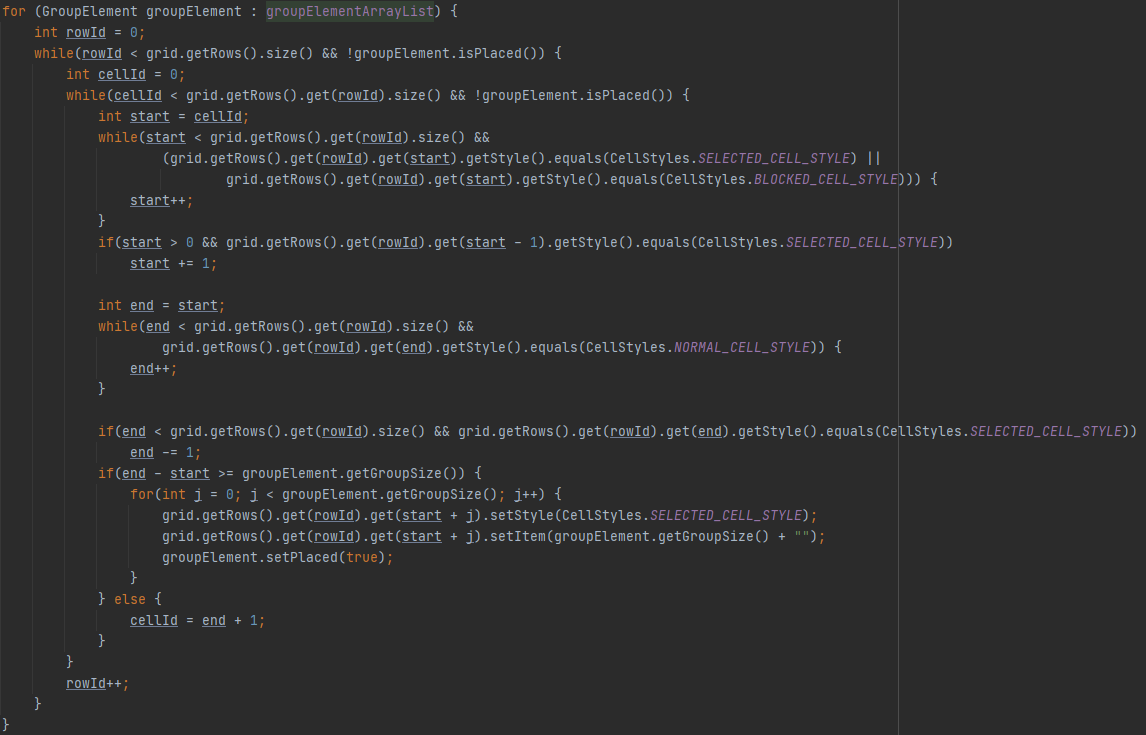
A convertRoomToObject függvény egy Room osztályt vár paraméterként. Kezdésként létrehozok egy BasicDBObject-et, amit a későbbiekben a terem adataival feltöltök. A roomObj létrehozása után hozzáadom a szoba azonosítóját, a színház azonosítóját, a terem nevét, a sorok és az oszlopok számát. Ezután a székeket visszaalakítom széksorokká, amit egy string-ben tárolok, amit a sor feldolgozása után ugyancsak hozzáadok a roomObj-hez. Miután az összes sor feldolgozása megtörtént a függvény visszatér a roomObj-vel, ami tartalmazza a paraméterben kapott osztály json átiratát.

# Gépi megoldó algoritmusok

## Mohó algoritmus

A gépi megoldókhoz implementálásra került egy mohó algoritmus, amely az előre meghatározott csoportokat el tudja helyezni a táblázatban.

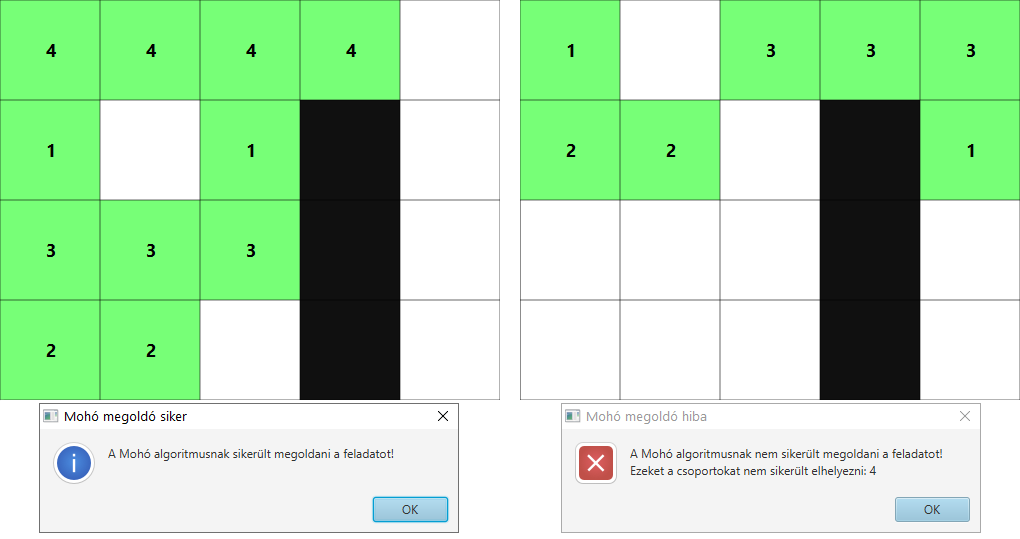
Az algoritmus az első inputként megkapott csoporttól kezdődően végighalad az utolsó csoportig. Minden egyes csoportnál megkeresi az első olyan helyet, ahol el lehet helyezni a csoportot, majd erre a helyre elhelyezi a csoportot és tovább lép a következő csoportra.



**14. ábra:** mohó algoritmus megoldó kódja

A **14. ábra** szemlélteti az algoritmus működését. Látható, hogy a ***GroupElement***-ben eltárolt csoportokon megy végig az algoritmus. Sorfolytonosan végighalad az egész táblázaton a függvény keresve a legelső olyan helyet, ahová el lehet helyezni az aktuális csoportot. Amikor ezt a helyet megtalálja az algoritmus, akkor lefoglalja a csoportnak ezt a helyet és átállítja a ***placed*** adattagját a csoportnak ezzel jelezve, hogy sikerült leültetni a csoportot. A ***placed*** adattag igazra állítása után lép a következő csoportra az algoritmus vagy akkor, ha végig járta az algoritmus az egész táblázatot.

Az algoritmus a lefutást követően egy felugró ablakban jelzi a felhasználó számára az algoritmus sikerességét. Amennyiben nem sikerült az összes elemet elhelyezni a táblázatban, akkor ezen az ablakon megjelennek azok a csoportok, amelyeket nem sikerült elhelyezni.



**15. ábra:** mohó algoritmus példa

Az algoritmus számára nagyon fontos, hogy milyen sorrendben vannak a csoportok megadva neki, mert sorfolytonosan dolgozik. A **15. ábra** mutatja, hogy mennyiben meg tud változni egy elrendezés, ha két csoportot felcserélünk a sorban. A bal oldali képen 4, 1, 3, 2, 1 sorrendet adtunk meg, a jobb oldali képen pedig 1, 4, 3, 2, 1. Továbbá látható, hogy a bal oldali esetben az algoritmusnak sikerült elhelyezni az összes csoportot, viszont a jobb oldali képen pedig már nem.

## Genetikus algoritmus

A gépi megoldókhoz implementálásra került egy genetikus algoritmus is, amely az előre meghatározott csoportokat közel optimálisan el tudja helyezni a színházteremben.

A genetikus megoldó implementálásához egy egyszerű online elérhető generikus algoritmust használtunk. A használatához meg kell adni egy bemenetet és egy fitness függvény. Az algoritmus szelekciót, mutációt és keresztezést használ. Az algoritmus megállási feltételének a megadott fitness érték elérése. Jelen algoritmus esetében a küszöbérték a nézőcsoportok száma. Ha eléri a küszöbértéket akkor az azt jelenti, hogy minden csoportot le tudott ültetni az algoritmus.

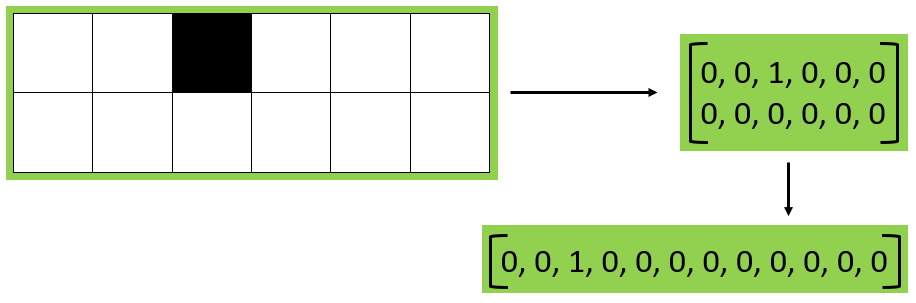
Az algoritmus bemenetként meg kell adni az aktuális színháztermet és a nézőket tartalmazó szöveget. Első lépésben a nézőket tartalmazó string-et átalakítjuk kulcs-érték struktúrává, ahol a kulcs a néző csoport mérete és az érték az adott darabszámú csoportok száma.

A képen asztal látható

Automatikusan generált leírás

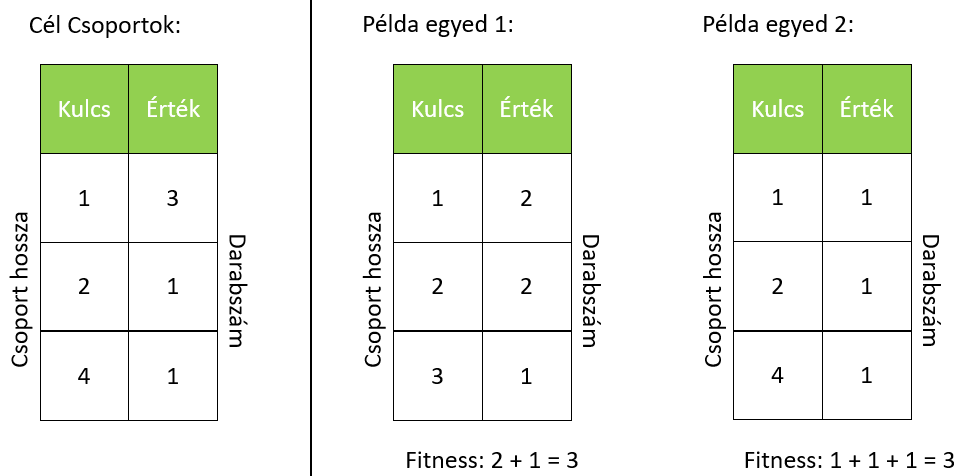
**9. ábra**: Nézők feldolgozása

Ezután átalakításra kerül a terem egy vektorrá. Ennek a menete két lépésben történik. Első lépésben a terem székeiből kinyerve a székek státuszát létrehozzuk egy mátrixot majd második lépésként átalakítjuk a mátrixot egy vektorrá. A létrejött vektor lesz a kiindulópontja az algoritmusban.



**10. ábra**: kiinduló állapot létrehozása

A algoritmust működtető fitness függvény megkapja a gén vektort amit kiértékel és ellátja egy pontszámmal. A fitness függvény az előállított géneket átállítva mátrixos formába és onnan tovább kulcs-érték struktúrába. Az újonnan létrehozott map-et összehasonlítjuk az eredeti map-el és minden csoport után, ami szerepel az eredeti map-ben az aktuális gén kap egy pontot. Viszont, ha az eredeti map-ben az adott méretű csoportból már mindet leültettük akkor azért már nem jár több pont.



**11. ábra:** Példa fitness érték számítására

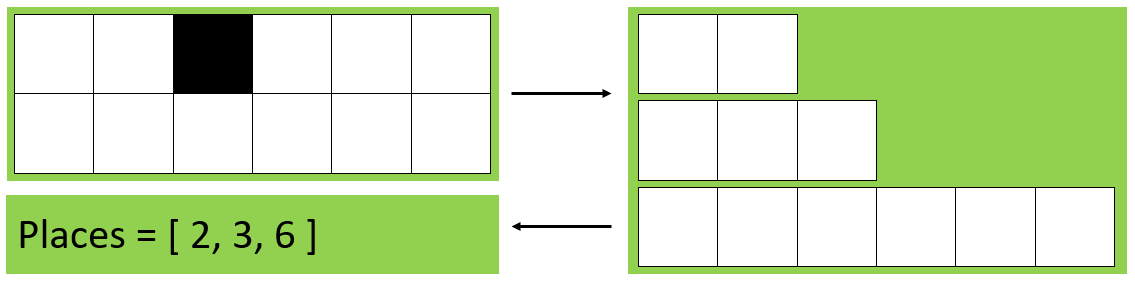
A fenti példában az látszik, hogy az eredeti map, ami a leültetendő nézőket tartalmazza van bal oldalon. A középen látható egyed esetében megnézzük a csoportokat, amik összejöttek és az látszik, hogy egy hosszú csoportból van kettő ezt összehasonlítjuk a célhoz és ha kevesebb a célnál akkor az adott mennyiségű pontot kap az egyed, ha viszont nagyobb, mint a cél akkor a cél mennyiségnek megfelelő pontot kap az egyed. Itt az látszik, hogy egyesekből kap két pontot kettesből kap egy pontot és a hármas csoportért nem kap pontot mert nincs rá szükség.

Ha a fitness érték eléri eléri a cél, ami a leültetendő csoportok száma akkor megtaláltuk az eredmény és elkezdődik az eredmény visszaalakítása. A gén vektort visszaalakítjuk mátrix-á majd a mátrixot vissza szobává. A szobát átadjuk a függvénynek, ami kirajzolja az elhelyezett csoportokat.

## LP megoldó

A gépi megoldókhoz implementálásra került egy LP megoldó is, amely az előre meghatározott csoportokat optimálisan el tudja helyezni a színházteremben.

A megoldóhoz szükséges modellt az általános hozzárendelési feladat átírásával állt elő. A modellnek bemenetként meg kell adni egy ***places*** vektort és egy ***people*** vektort. A ***places*** vektor tartalmazza az egybefüggő helyek hosszát, míg a ***people*** vektor tartalmazza a néző csoportokat pontosabban a csoportokban lévő nézők számát felsorolásszerűen.



**12. ábra:** places vektor elkészítésének menete

Azáltal, hogy nem egyesével rendelünk hozzá nőzőket a székekhez, hanem néző csoportokat rendelünk összefüggő helyekhez kevesebb változó és megkötés felvételével is végrehajtható a számítás. Ezen felül mivel kevesebb a változók száma a számítás is gyorsult, bár ez a feladatok mérete miatt elhanyagolható.

A modell célfüggvényét a változók összege adja. Optimális megoldás esetén minden csoportot le lehet ültetni pontosan egy helyre. Ebben az esetben a célfüggvény értéke megegyezik a néző csoportok számával.

A modellben szereplő változók száma a people vektor elemeinek száma szorozva a places vektor hosszával. A példában látható színházterem és nézők esetében 3 \* 5 azaz 15 változó szerepel.

A modellben összesen két féle megkötést kell alkalmazni az egyik, amikor megadjuk, hogy egy néző csoport pontosan egyszer kell elhelyezni a másik, amikor megmondjuk, hogy egy összefüggő ülőhely csoporton belül ne lehessen leültetni a maximális ülőhelyszámnál többet.

Az első megkötés a gyakorlatban elég egyszerű mivel csak össze kell adni az azonos i indexű változókat, amiknek az értéke egyenlő kell legyen eggyel.

A második megkötés egy kicsit trükkösebb mivel ott már figyelembe kell venni a vírushelyzetben bevezetett megkötéseket is, ami miatt két csoport között ki kell hagyni egy üres helyet. Ezért sorban haladva találtuk ki a megkötést. Először csak azt írtuk fel, hogy ne haladja meg a leültetett emberek száma a maximális ülésszámot az üléshely csoportban.

Ezután hozzáadtuk az egy üres szék rész, úgy hogy minden elhelyezett csoport után egyet hozzáadtunk az egyenlőtlenség bal oldalához majd a végén kivontunk egyet, mivel ha az utolsó csoport után nem szeretnénk másik csoportot leültetni akkor nem szükséges a plusz egy hely.

A modellben a ***places*** vektor hosszának és a ***people*** vektor hosszának összegével egyenlő kikötés szerepel. A példában ez azt jelenti, hogy 3 + 5 azaz nyolc darab kikötés van három a helyekre és őt a néző csoportokra.

A megoldó lefutása után az eredmény leolvasható a változókról. Az eredmény jobban látható, ha mátrixos formában jelenítjük meg. Ahogy az a **13.ábrán** is látható a példa színház és a példa nézővektor (4, 2, 1, 1, 1) megadásával a megoldó a következő megoldást adja.

A képen asztal látható

Automatikusan generált leírás

**13. ábra:** Példa a megoldó kimenetére

Ebből leolvasható, hogy melyik nézőcsoport melyik székcsoporthoz lett rendelve. A példában az látható, hogy az első négy fős nézőcsoport a harmadik székcsoporthoz lett rendelve. Ezen felül még a harmadik egy fős nézőcsoport is a harmadik székcsoporthoz lett rendelve.

Ezeket az étékeket átalakítjuk egy kulcs-érték adatszerkezetbe, ahol a kulcs a nézőcsoport indexe és az érték a székcsoport indexe, amihez hozzá lett rendelve a néző. Az eredmény megjelenítésének megkönnyítése érdekében a nézőket sorba rendezzük a hozzá rendelt székcsoport indexe alapján. Ez azért jó mert így elég egyszer végig iterálni a termen és különösebb ellenőrzések nélkül lehet elhelyezni a nézőket.

Az elkészült és nézőkkel feltöltött kétdimenziós mátrixot átalakítjuk egy vektorrá amit átadunk a kirajzoló függvénynek ami elvégzi a grafikus felületre történő kirajzolást és felcímkézi a csoportokat nézők száma alapján.

## Gépi megoldók összehasonlítása

Az implementálásra került három gépi megoldók mindegyike részben vagy egészben leültetni a nézőket az előírt szabályok betartása mellet. Az algoritmusok és azok működése elveinek különbözősége miatt a különböző megoldók más-más tulajdonságokkal rendelkeznek.

A működési elvében legegyszerűbb és ebből kifolyólag a leggyorsabb megoldónk a mohó algoritmus. Ennek az algoritmusnak a legnagyobb előnye, hogy nagyon gyors és nem befolyásolja a futási idejét a székcsoportok száma. Ebből kifolyólag nem függ a futási ideje a színházak felépítésétől. Ez a tulajdonság önmagában nagyon kecsegtető viszont a megoldó nagy hátránya, hogy a kapott megoldás nem feltétlenül optimális. A mohó tulajdonság miatt mindig lokális optimumra törekszik az algoritmus, ami miatt az eredmény nem feltétlenül optimális. Emellett az algoritmus megoldását befolyásolhatja a nézőcsoportok változatossága is. Ez a megoldó a következő esetekben ajánlott:

* Nagy színházterem.
* Nincs szükség feltétlenül optimális megoldásra.
* Ha nem lesz teljesen feltöltve a színházterem.
* Ha gyors megoldás szükséges

A második megoldónk a genetikus algoritmus, ami már egy olyan megoldó, ami optimális megoldást ad. Viszont nagy hátránya, hogy a futási ideje nagyban változhat mivel az algoritmus a véletlenen alapul ezért szerencsés esetben az algoritmus már az első iteráció alkalmával talál optimális megoldást, de van amikor az 5 milliomodik iteráció alkalmával találja csak meg a megoldást. Az algoritmus miatt abban az esetben amikor teljesen ki van használva a terem akkor általában a megoldó lassan talál megoldást. Ezt a megoldót a következő esetekben érdemes használni:

* Optimális megoldás szükséges.
* A terem előreláthatóan nem lesz tele.
* A nézőcsoportok kicsik.

A harmadik és egyben a legmegbízhatóbb megoldónk az LP algoritmus. Ennek a megoldónak nagy előnye, hogy mindig megtalálja az optimális megoldást, ha lehetséges, illetve a futási ideje az előző megoldóhoz képes nagyjából állandó ugyanolyan bemenetek esetén. Másik hátránya az algoritmusnak, hogy nagy tagolt termek esetében jóval lassabb, mint az első mohó megoldó. Mivel ez a megoldó érzékeny a székcsoportok számát illetően minél több csoport található egy teremben annál több időt igényel az algoritmus lefutása. Viszont előnye, hogy az algoritmust nem befolyásolja a nézőcsoportok mérete. Előnye még hogy a szabályok változása esetén könnyen lehet módosítani az algoritmust. Ez az algoritmus a következő esetekben ajánlott:

* Nagy termek esetén
* Optimális megoldás szükséges.
* Bármilyen kihasználású terem esetén.
* Bármilyen nagy csoportok esetén.

Az megoldok főbb tulajdonságait a következő táblázatban láthatók:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Optimális eredmény** | **Sebesség** | **Ajánlott teremméret** | **Terem kihasználtsága** |
| **Mohó** | Nem | Gyors | Nagy | Közepes |
| **Genetikus** | Igen\* | Változó\* | Közepes | Közepes |
| **LP** | Igen | Gyors | Nagy | Nagy |