Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Nonogram megoldó algoritmus és vizualizálása

Témavezető:

Nagy Sára

mestertanár

Szerző:

Solymosi Patrícia

programtervező informatikus BSc

Budapest, 2019

Tartalomjegyzék

1. Bevezetés

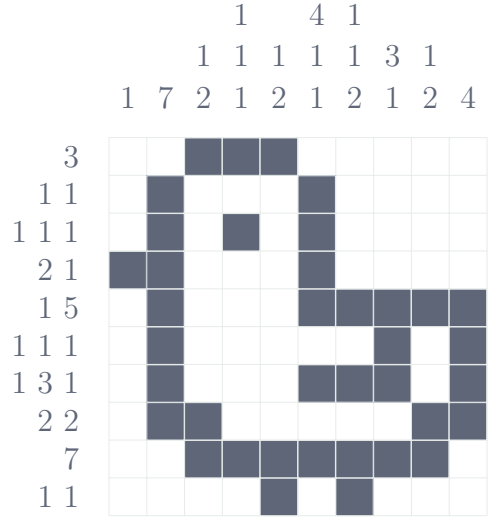
1.1. A Nonogramról röviden

1.1.1. Történelmi áttekintés

A *Nonogram* (más néven: *grafilogika*) egy japán eredetű logikai rejtvényfajta. Története 1987-re nyúlik vissza, amikor egy japán dizájner, Non Ishida egy felhőkarcoló egyes ablakait kivilágítva, másokat elsötétítve képeket rajzolt ki, míg ezzel egy időben Tetsuya Nishido hasonló elven rejtvényt fejlesztett. 1988-ban kezdtek el megjelenni ezek a feladványok japán újságokban, de ekkor még *Window Art Puzzles* néven. A *Nonogram* elnevezést, amely a tervező családnevéből származik, először 1990-ben a *The Sunday Telegraph* használta. Elektronikus formában 1995-től kezdve jelentek meg különböző játékkonzolokon. Hamarosan a világ egyre több pontján találkozhattunk grafilogika rejtvényekkel nyomtatásban, köztük Magyarországon is.

1.1.2. A rejtvények szerkezete, alapvető fogalmak

A *Nonogram* egy téglalap alakú négyzetrácsos hálóból áll, melynek minden sorához és oszlopához tartozik egy számsor. Ezen számsorok határozzák meg, hogy az adott sorban vagy oszlopban milyen sorrendben mekkora hosszúságú sötét blokkokat kell keresnünk. A blokkok között legalább egy üres helyet hagynunk kell. Fontos, hogy a számsorban szereplő értékek sorrendtartóak, tehát balról jobbra, illetve fentről lefelé az adott sorrendben tekintjük a számokat a megoldás során.



1. ábra. Egy egyszerű *Nonogram* rejtvény és megoldása [9]

A dolgozatban a blokkokra *szegmensként*, a számsor elemeire *clue*-ként hivatkozok. Az egyszerűség kedvéért a sorok és oszlopok nem kerülnek megkülönböztetésre, egyszerűen a *sor* elnevezést használjuk.

> **Megjegyzés**: nem minden rejtvény rendelkezik egyértelmű megoldással. A dolgozat azonban csak olyanokkal foglalkozik, amelyek a logikai módszerek használatával egyértelműen megoldhatók (tehát egyetlen megoldásuk létezik). Ez a programban található fájlok mindegyikére teljesül.

1.2. A témaválasztás indoklása

A dolgozatom elsődleges célja a rejtvények megoldásához szükséges logikai módszerek megfogalmazása algoritmusok formájában, majd azok működési folyamatának látványos bemutatása. Az emberi gondolkodás, logika átültetése programkódra érdekes kihívásnak bizonyult.

Tekintve, hogy a téma alapja egy játék, ezért természetesen célom volt, hogy lehetőséget biztosítsak a felhasználónak feladványok megoldására, illetve készítésére.

2. Felhasználói dokumentáció

2.1. Rendszerkövetelmények

A program a következő rendszerkonfiguráció mellett lett tesztelve:

- **operációs rendszer**: Windows 10,
- **processzor**: Intel Core i5-5200U 2.20GHz,
- **memória**: 8 GB,
- **képernyőfelbontás**: 1920 × 1080.

Ettől eltérő konfiguráció esetén a kellően gyors működés nem minden esetben biztosítható.

A `Nonogram.exe` nevű programot elindítva érhetjük el az alábbiakban részletezésre kerülő funkciókat. További fájlokra a használathoz nincsen szükség. Megnyitás után a program automatikusan létrehozza a számára szükséges mappákat a fájl gyökérkönyvtárában.

2.2. A program felépítése, használata

A program nézete négy fő részből épül fel:

(1) **menüsáv**: a fő funkciókhoz (szimuláció, tervező, játék) irányító menüpontokat, az ablakvezérlő gombokat, illetve a súgó megnyitására szolgáló gombot találjuk;

(2) **almenüsáv**: az adott funkción belüli almenüpontokat (pl. fájl megnyitása) tartalmazza;

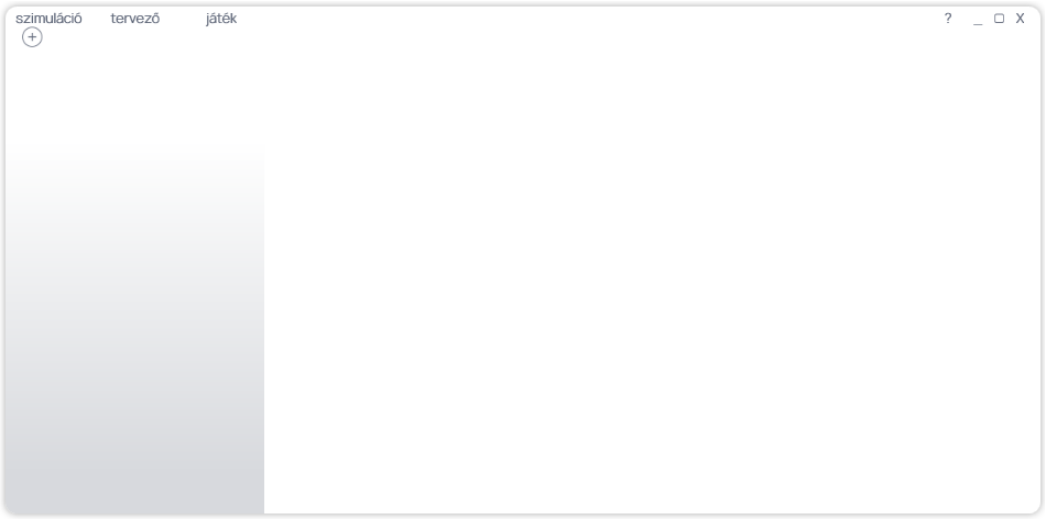
(3) **információ- és irányítósáv**: funkciótól függően az adott rejtvény információit, illetve a vezérlőgombokat (pl. szimuláció indítása) tartalmazza;

(4) **rejtvény helye**.

A következőkben a három fő funkció használatának részletezésére kerül sor.

2.2.1. A szimuláció használata

A szimuláció funkciót választva megjelenik az almenüsávon a rejtvény kiválasztására szolgáló menüpont.



2. ábra. Szimuláció nézete kezdetben

Erre kattintva felugrik egy ablak, ahol kiválasztható a méret, majd egy tetszőleges rejtvény a listából. Az egyes sorokban a következő információkat találjuk:

(1) rejtvény **azonosítója**: a program működése szempontjából lényegtelen, forrásmegjelölésként szolgál;

(2) rejtvény **neve**;

(3) rejtvény **bonyolultsága** 1–5-ig tartó skálán.

> **Megjegyzés**: a rejtvények bonyolultsága a forrásoldalon (`http://www.nonograms.org`) elérhető adatok alapján került meghatározásra.

3. ábra. A rejtvény kiválasztására szolgáló ablak: a nagyméretű rejtvények listájának egy részlete

Egy rejtvény kiválasztását követően az üres tábla mellett, az információsávon megjelennek a részletek:

(1) a **rejtvény adatai**: név, méret, bonyolultság;

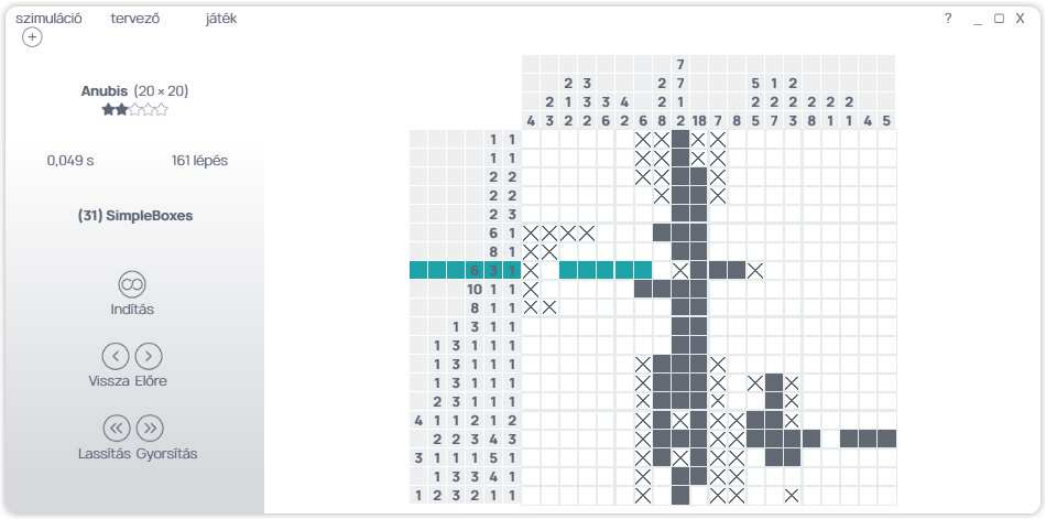
(2) a **szimuláció adatai**:

- **futási idő**: a megoldási folyamat ideje másodpercekben mérve,
- **lépésszám**: hányszor kellett alkalmazni az egyes algoritmusokat, hogy a rejtvény megoldása elkészüljön;

(3) az utoljára alkalmazott **algoritmus sorszáma és neve** (kezdetben ez a sáv természetesen üres);

(4) **vezérlőgombok**:

- **indítás** vagy **szünet**: ezen gomb lenyomásával indíthatjuk el a szimulációt, ekkor a lépések egymás után, folyamatosan kerülnek megjelenítésre;
- **vissza/előre**: szüneteltetett állapotban egyesével léptetve nézhetjük végig a megoldási folyamatot;
- **lassítás/gyorsítás**: ha a szimuláció folyik, akkor ezen gombok segítségével lassíthatjuk vagy gyorsíthatjuk a lépések megjelenítését.



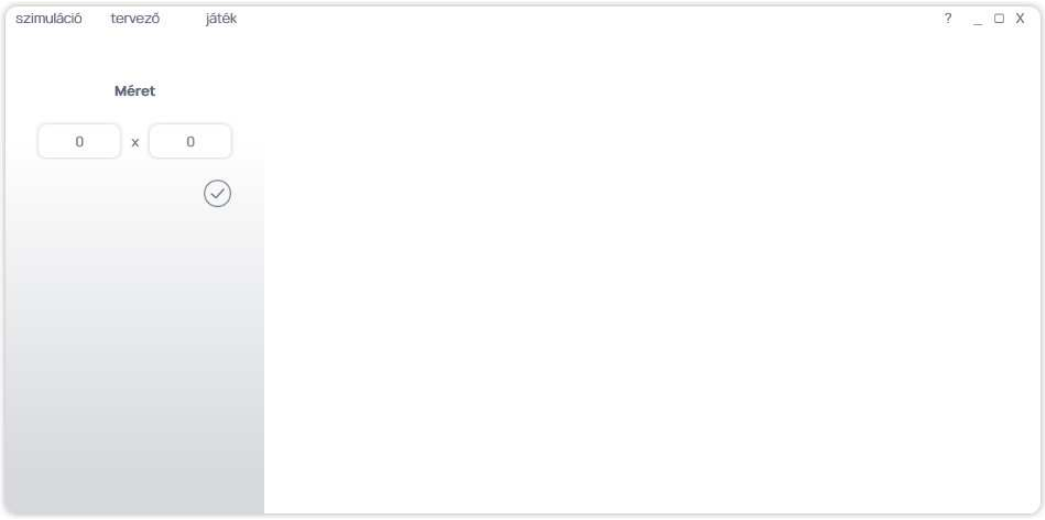
4. ábra. Aktív szimuláció nézete

A(z) 4. ábrán a megoldás 31. lépésének állapotát láthatjuk. Az aktuális lépéshez tartozó algoritmust a kékkel jelölt soron végezzük, amelynek eredményét az ugyanolyan színnel kiemelt mezők jelentik.

A szimuláció végén egy dialógusablakban információt kapunk a megoldás eredményéről (helyes vagy helytelen).

2.2.2. A tervező használata

A tervezőben grafikus felületen létrehozhatjuk saját rejtvényünket. Ehhez a megjelenő beviteli mezők segítségével meg kell adnunk a sorok és az oszlopok számát: mindkét értéknek 1 és 45 közé kell esnie.

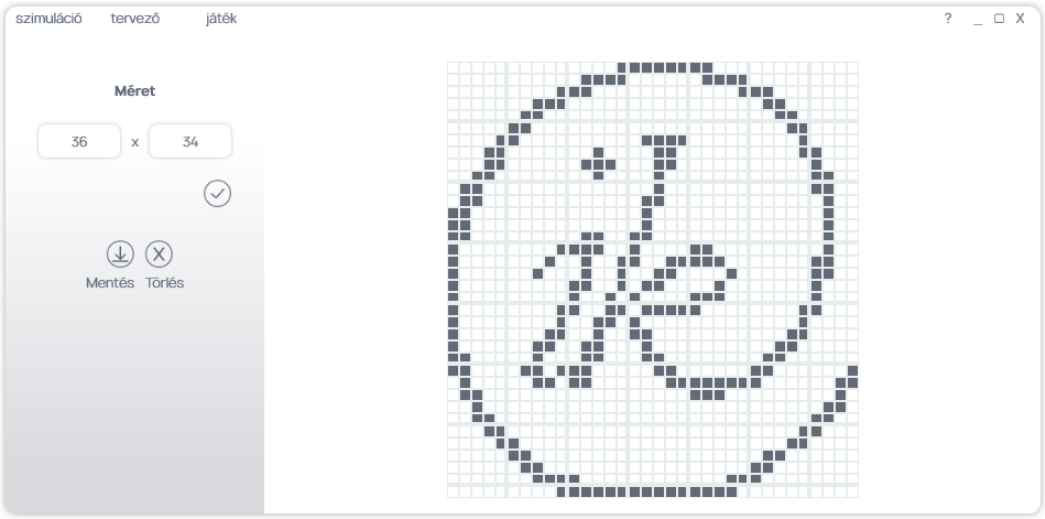


5. ábra. Tervező nézete kezdetben

Ha a beírt értékek helyesek (vagyis a fenti intervallumba esnek), akkor a mezők alatt található gombra kattintva, a számoknak megfelelően megjelenik az adott sorból és oszlopból álló üres tábla, valamint az információsávon két további funkció:

(1) **mentés**: a kitöltött tábla elmentése egyszerű szöveges állományként a felugró ablakban megadott néven (max. 20 karakter hosszú, csak számot, betűt és szóközt tartalmazó szöveg) a program mellett található `Created` nevű mappába;

(2) **törlés**: a tábla aktuális állapotának törlése úgy, hogy minden mezőjét üresre állítjuk.



6. ábra. Példa egy rejtvény készítésére

> **Megjegyzés:** a dolgozatnak nem célja és feladata, hogy tetszőleges feladványról eldöntse, hogy (egyértelműen) megoldható-e, ezért a tervezőben létrehozott rejtvények a szimulációban nem használhatók.

A tábla kitöltésének módja a következő:

(1) **sötétre színezés**: a bal egérgomb lenyomásával a kijelölt mező kitöltöttre állítható;

(2) **áthúzott mező létrehozása**: a jobb egérgomb lenyomásával a kijelölt mező áthúzott (üres) lesz;

(3) **mező visszaállítása üresre**: az aktuális állapottól függően lenyomva ugyanazt az egérgombot a mező ismét üresre állítható.

A **gyors kitöltéssel** az előzőleg sötétre (áthúzottra) állított mező és a következő, azonos sorban vagy oszlopban lévő mező között minden mező sötétre (áthúzottra) állítódik. Ehhez a `SHIFT + F` billentyűkombinációt szükséges egyszer lenyomni az új mező bejelölése előtt.

> **Megjegyzés**: az üres és az áthúzott mezők csupán megjelenítési szempontból különböznek tervezés esetén. Mentéskor minden üresen hagyott mező áthúzottnak minősül.

2.2.3. A játék használata

A programban nehézségi fokozat szerint háromféle kategóriában találhatunk rejtvényeket megoldásra. Az új játék menüpont aktiválására felugró ablakban a nehézség kiválasztását követően megjelenik az adott kategóriához tartozó rejtvények listája.

7. ábra. Új játék megnyitására szolgáló ablak

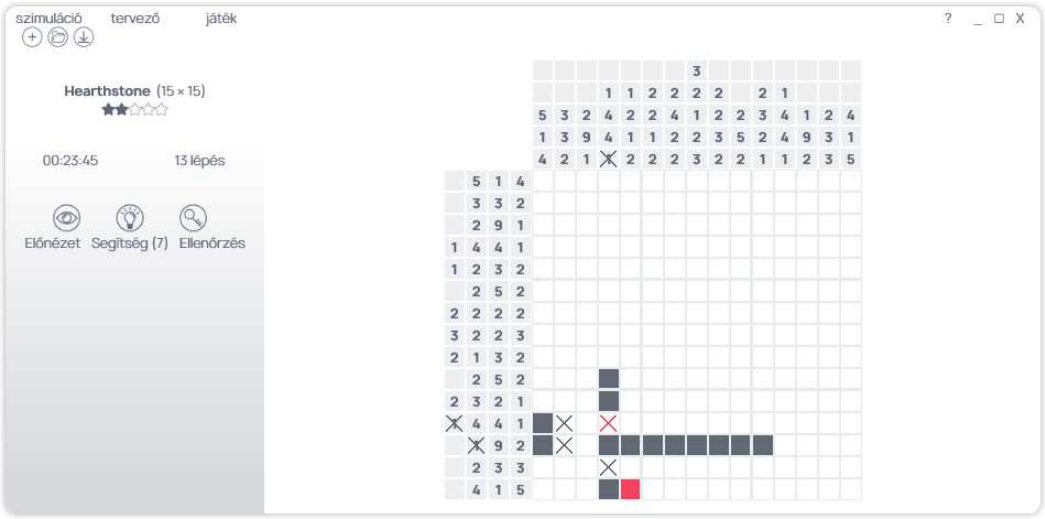
A megoldást könnyítő, vagy éppen kihívások elé állító beállítások közül választhatunk, úgymint:

- **előnézet**: ezen beállítás kiválasztásával az előnézet funkció aktívvá válik, amely használatával a tábla bal felső sarkában megjelenik a teljes megoldás egy pillanatra, ezáltal képet kaphatunk a végeredményről;
- **segítség**: a segítség funkció hatására, ha lehetséges, akkor a táblán felfedésre kerül véletlenszerűen egy helyes szegmens. A segítségek száma a játék bonyolultságától függően eltérő: rendre maximum 10, 7, illetve 5 segítség használható fel;
- **időmérő**: kiválasztásával a játék megoldására adott idő áll rendelkezésre, az idő lejártával a tábla zárolódik, további műveletek nem végezhetők;
- **szakaszjelölés**: a teljesen kitöltött és helyes szegmensek üressel (tehát áthúzott mezővel) történő elválasztása esetén a megfelelő számsor megfelelő elemét jelöli a program a tábla szélén.

Betöltést követően az üres tábla mellett az információsávon megjelennek a játék adatai, illetve a beállításoktól függően aktív vagy nem aktív funkciók:

- **visszaszámláló**: ha az időmérő beállítás aktív, akkor bonyolultságtól függően 25, 35 és 45 perc áll rendelkezésre, hogy megoldjuk a rejtvényt – ez esetben visszaszámlálót találunk, különben egy ∞ szimbólumot;
- **lépésszám**: egy mező átállítása egy lépésnek számít;
- **előnézet** gomb;
- **segítség** gomb: a maradék segítségek számát zárójelben találjuk;
- **ellenőrzés** gomb: mindig aktív funkció, a tábla aktuális állapotát ellenőrzi, majd a helytelen mezőket piros színnel jelöli.

Ha a tábla minden mezője helyesen kitöltött, akkor a program felugró ablakban értesítést küld a megoldás sikerességéről.



8. ábra. Játék felülete: ellenőrzés hatására a program pirossal színezi a helytelenül jelölt mezőket

A megkezdett játék nem zárolt állapotban bármikor menthető, majd újból betölthető a megfelelő menüpont kiválasztása után megjelenő ablakban (9. ábra).

9. ábra. Mentett játékok listája

A rejtvény kitöltési módja a tervezőben ismertetettel azonos.

3. Fejlesztői dokumentáció

3.1. Feladatok meghatározása

Három logikai részből álló grafikus felületű C# WPF (*Windows Presentation Foundation*) alkalmazás megvalósítása.

3.1.1. A szimulációtól elvárt feladatok

- A program által biztosított fájlok listázása kategóriánként. A kiválasztott fájl megnyitása.
- A fájl alapján a rejtvény felépítése, majd grafikus felületen való megjelenítése.
- A későbbiekben részletezett logikai módszerek megvalósítása és alkalmazása az adott rejtvényre.
- Képes legyen az algoritmusok eredményét megjeleníteni grafikus felületen lépésenként vagy folyamként.

3.1.2. A tervezőtől elvárt feladatok

- A felhasználó által megadott számú sorból és oszlopból álló, kezdetben üres tábla létrehozása, majd kirajzolása grafikus felületen.
- A felhasználó előre meghatározott módon a tábla mezőit képes legyen módosítani, a program ezt észlelje és tárolja az aktuális állapotot.
- A tábla mentése `txt` formátumban a program mellett található mappába a felhasználó által megadott néven. Gondoskodjon arról, hogy a név speciális karaktereket (?, !, :, stb.) ne tartalmazhasson.
- A tábla megtisztítása, vagyis a kezdeti, üres állapot újbóli előállítása.

3.1.3. A játéktól elvárt feladatok

- A program által biztosított fájlok, valamint a felhasználó által a tervezőben mentett rejtvények listázása.
- Lehetőség biztosítása különböző segítségek és beállítások kiválasztására. Ezen plusz funkciók kezelése.

  - *Előnézet funkció* esetén biztosítsa a rejtvény megoldásának grafikus megjelenítését a tábla bal felső sarkában 2 másodpercre.
  - *Segítség funkció* esetén rejtvénytől függően adott számú helyes szegmens felfedését engedélyezze. Kezelje azt az esetet, amikor a tábla aktuális állapota miatt nem fedhetőek fel további szegmensek.
  - *Időmérés funkció* esetén a rejtvény bonyolultságától függően hozzon létre visszaszámlálót. Az idő letelte esetén értesítse a felhasználót, majd minden további műveletet zároljon.
  - *Szakaszjelölés funkció* esetén jelölje a tábla szélein a megfelelő szám(ok) áthúzásával a helyesen kitöltött és elválasztott szegmenseket.

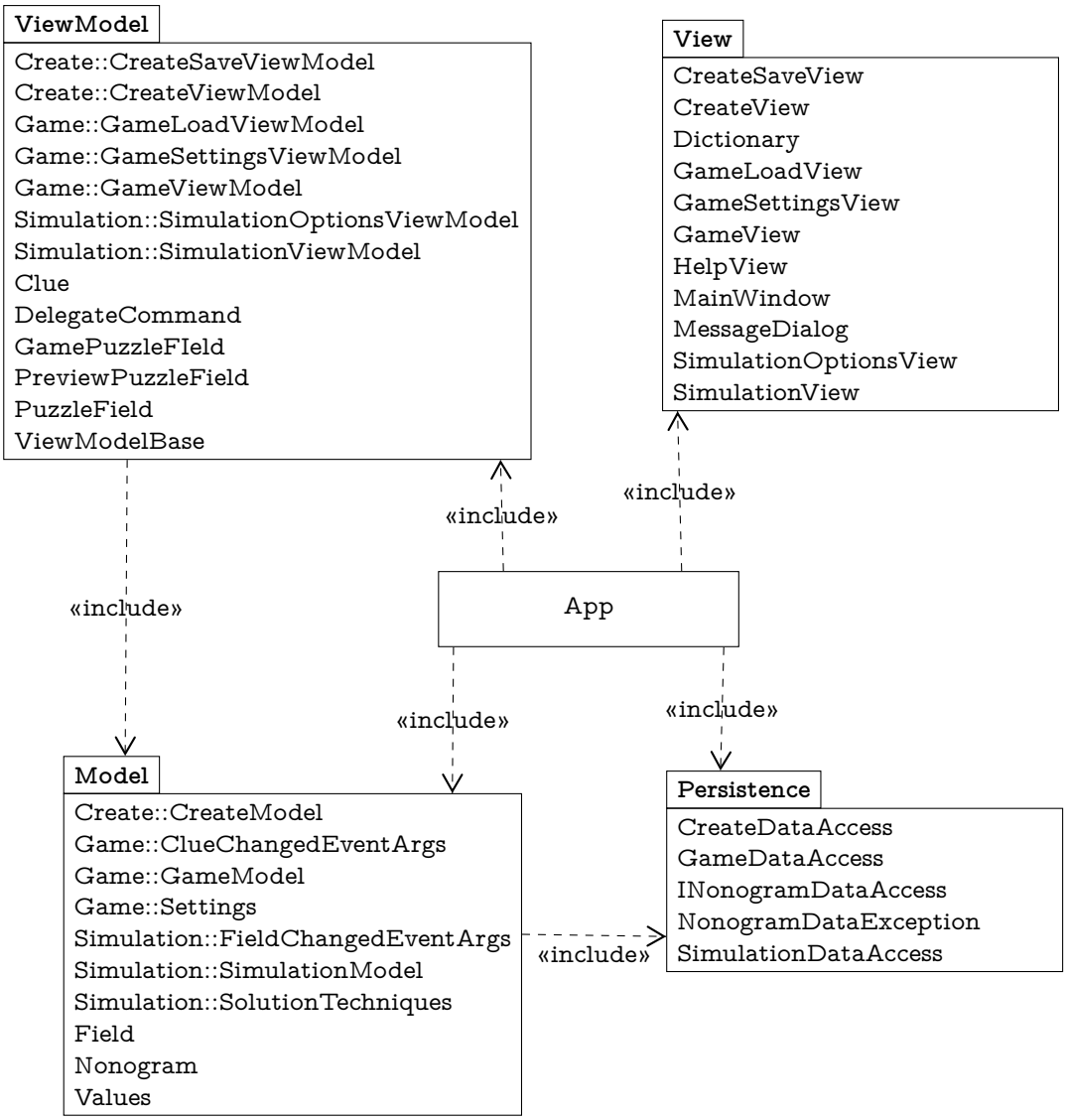
- Adott rejtvény betöltése, majd megjelenítése. A rejtvény megoldásának és a felhasználó által kitöltött táblának külön való kezelése.
- *Ellenőrzés funkció*: képes legyen a tábla és a megoldás összehasonlítására, az eltérő mezőkről értesítse a felhasználót a felületen.
- Számolja a megtett lépéseket.
- Értesítse a felhasználót a helyes kitöltés eredményéről.
- Biztosítson lehetőséget a rejtvény mentésére.
- Képes legyen a mentett rejtvények megnyitására és folytatására.

3.2. A program logikai és fizikai szerkezetének leírása

3.2.1. A projekt szerkezete

A program háromrétegű, MVVM (*Model–View–View Model*) architektúrában került megvalósításra. A modell így elkülönül a nézettől, illetve a nézethez tartozó nézetmodelltől. A háromrétegű architektúra biztosítja az egyes részek egymástól való függetlenségét, elősegíti a projekt áttekinthetőségét és továbbfejlesztését.

A modell az üzleti logikát tartalmazza, a nézetmodell a nézethez szükséges adatokat határozza meg és felel az eseménykezelésért, a nézet feladata pedig a grafikus megjelenítés.



10. ábra. A projekt csomagdiagramja

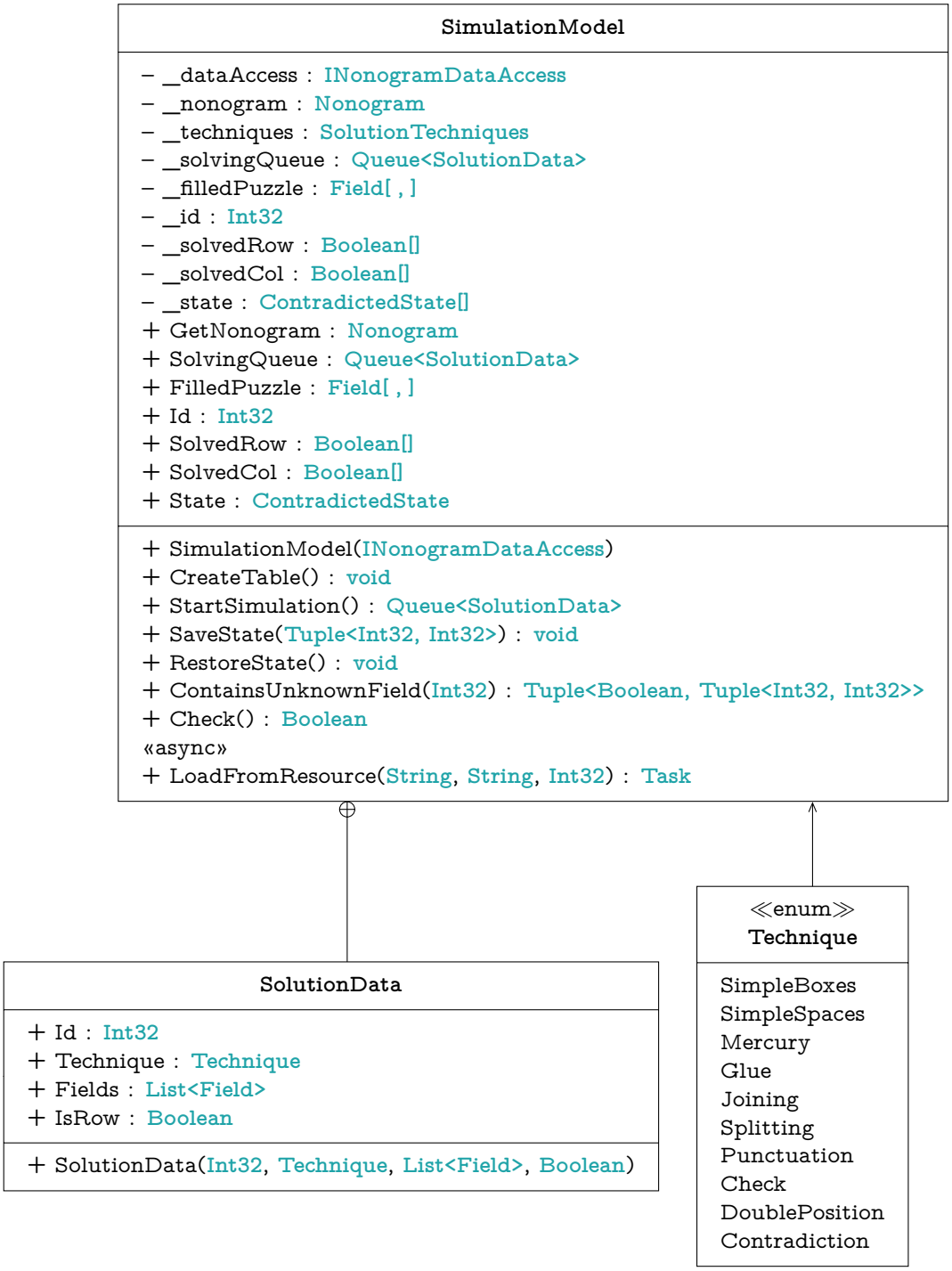
A program három fő funkcióból tevődik össze: a szimulációból, a tervezőből és a játékból. Ezen részek külön névtérben kerültek implementálásra, egymástól függetlenek, saját modellel, nézettel és nézetmodellel rendelkeznek.

A nézetmodellek ősosztálya minden esetben a `ViewModelBase` [1] nevű osztály.

3.2.2. Szimuláció

Modell.

A Nonogram.Model.Simulation névtérben találjuk a szimulációhoz tartozó modellt, amely az algoritmusok implementációját, az aktuális rejtvényhez szükséges információkat és a megoldáshoz szükséges metódusokat tartalmazó osztályokat foglalja magába.



11. ábra. A szimulációért felelős osztály

12. ábra. A logikai módszerek implementációját tartalmazó osztály

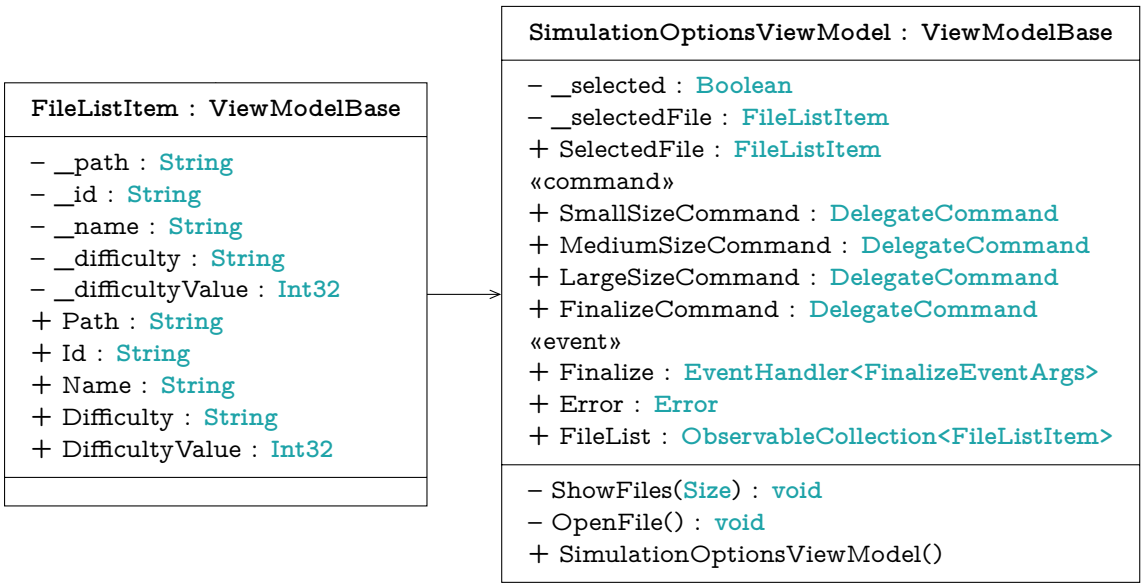
Nézet.

`SimulationView`: a szimuláció kiválasztásával megjelenő ablak implementációját tartalmazza. Felel a kiválasztott rejtvény, az ahhoz kapcsolódó információk, és a funkciógombok megjelenítéséért. A `SimulationViewModel` nézetmodell biztosítja a szükséges adat- és eseménykezelést.

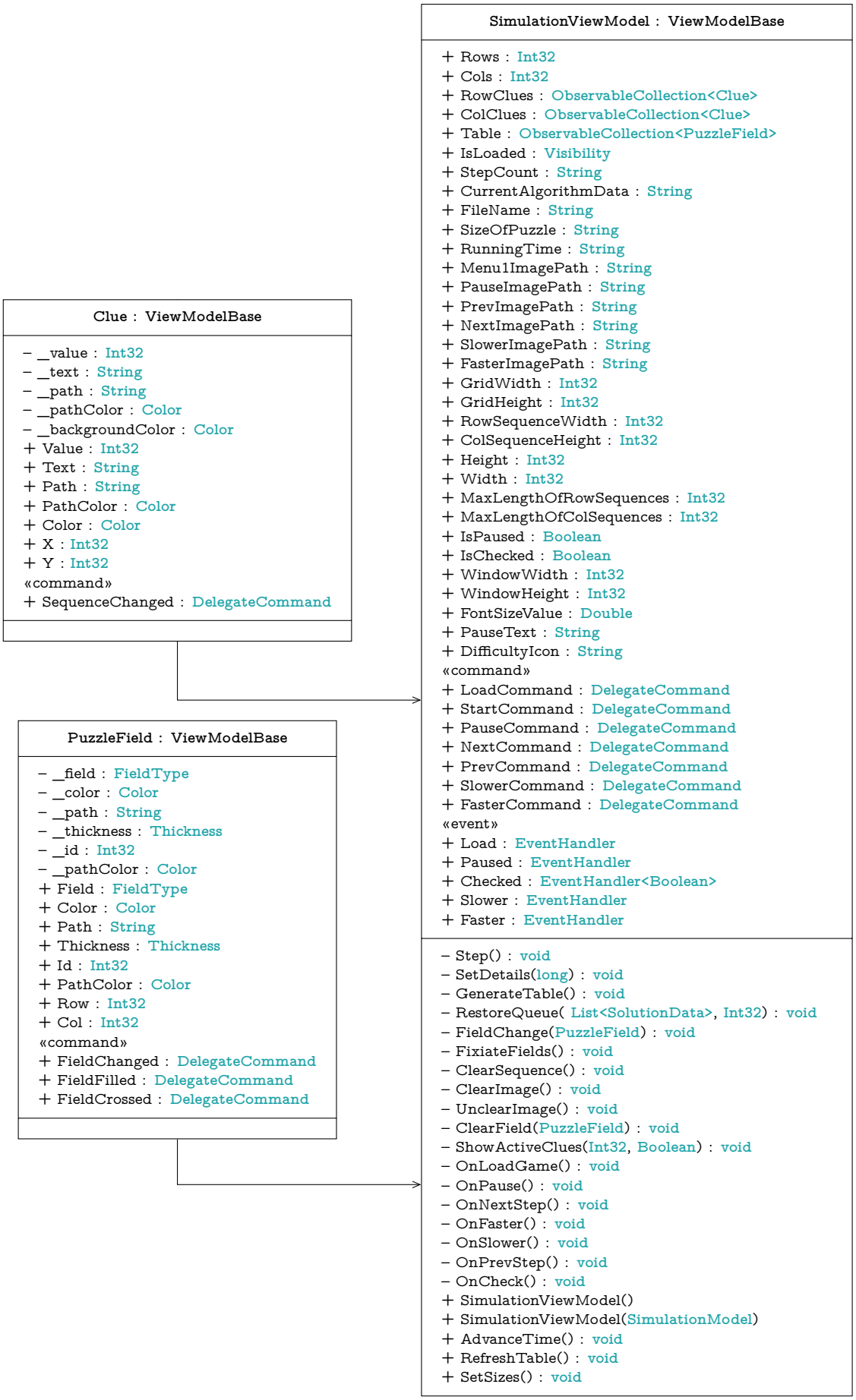
SimulationOptionsView: a programban található példarejtvények kilistázására és kiválasztására szolgáló nézet. A SimulationOptionsViewModel nézetmodell biztosítja a szükséges adat- és eseménykezelést.

Nézetmodell.

A szimulációhoz tartozó nézetmodellek a Nonogram.ViewModel.Simulation névtérben találhatóak meg.



13. ábra. SimulationOptionsView nézetmodell osztály



14. ábra. SimulationViewModel nézetmodell osztály

A meghatározott feladatok implementációja.

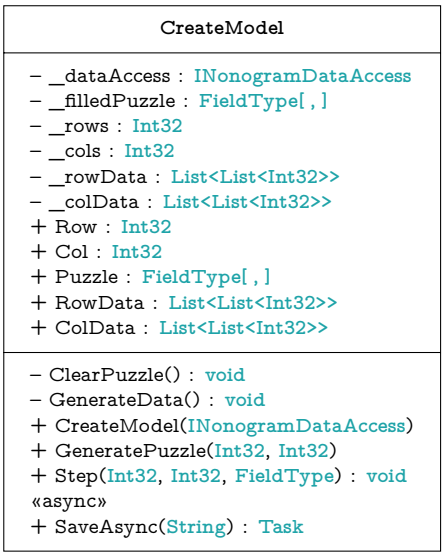
- A rendelkezésre álló fájlok listázását a `SimulationOptionsViewModel` nézetmodell végzi. Egy eseményen keresztül adja át a kiválasztott fájlt (`FileListItem` típus) az `App`-nak, amely ennek hatására meghívja a modell `LoadFromResource()` metódusát.
- A rejtvény felépítése a modellben a `CreateTable()` metódus segítségével történik. A tábla grafikai megjelenítését a nézetmodell `GenerateTable()` metódusa végzi.
- Amint a tábla betöltésre és megjelenítésre került, meghívódik a modell `StartSimulation()` metódusa, amely a logikai módszerek alkalmazásáért felel. A módszerek alkalmazásának a logikája a következő:
  - A megoldási technikák mind egy-egy

    `delegate Boolean Algorithm(List<Values>, Field[ , ], Int32, Boolean, Int32, Queue<SolutionData>, Boolean[])`

    függvényként vannak definiálva. Ezeket egy listába helyezve, majd azon végigiterálva alkalmazzuk őket egyesével először a tábla soraira, majd az oszlopaira.
  - Ha egy ciklusban minden függvény `false` eredménnyel tér vissza, akkor az azt jelenti, hogy egy módszer sem adott megoldást. Ekkor:
    - ha van még olyan mező, amelyről nem tudtuk megállapítani, hogy sötét vagy üres, akkor alkalmaznunk kell a tippelgetés (`Contradiction`) módszerét;
    - ha nincs több ismeretlen állapotú mező, akkor készen vagyunk.
- A `StartSimulation()` metódus visszatérési értéke egy olyan sor, amely `SolutionData` típusú elemeket tartalmaz – ebben tároljuk az egyes módszerek eredményét. A nézetmodell ezt a sort fogja feldolgozni, és adatai alapján megjeleníteni a megoldást a grafikus felületen.

3.2.3. Tervező

Modell.



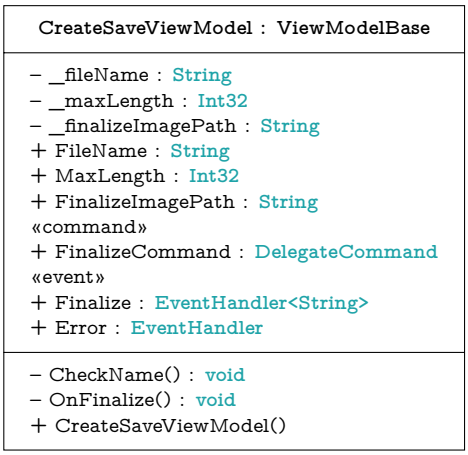
15. ábra. A tervező modelljét tartalmazó osztály

Nézet.

A tervező nézetét a `CreateView`, a mentés ablakának a nézetét `CreateSaveView` valósítja meg. Előbbi tartalmazza az inputmezőket, az egyes funkciógombokat és a rejtvény rácsát.

Nézetmodell.

A tervezőhöz tartozó nézetmodellek a `Nonogram.ViewModel.Create` névtérben találhatóak meg.



16. ábra. `CreateSaveViewModel` nézetmodell osztály

17. ábra. CreateViewModel nézetmodell osztály

A meghatározott feladatok implementációja.

A(z) 3.1.2. alfejezetben ismertetett feladatok megoldása a következő függvények segítségével került implementálásra:

- `GeneratePuzzle(Int32, Int32)`: a nézetmodellből érkező értékeknek megfelelően létrehoz egy új, `FieldType` típusú elemeket tartalmazó kétdimenziós tömböt (`Puzzle`), amelynek kezdetben minden eleme ismeretlen (`FieldType.Unknown`);
- `Step(Int32, Int32, FieldType)`: a `Puzzle` adott sorában és oszlopában lévő érték megváltoztatása a harmadik paraméterben megadott értékre. A nézetmodell hívja meg a felhasználó által végzett műveletek hatására;
- `SaveAsync(String)`: a tábla mentéséért felelős függvény, amely paraméterben megkapja a mentés útvonalát (előre, a programban definiált út), amely már tartalmazza a felhasználó által megadott nevet is (a helyesség ellenőrzése a nézetben történik). Mentés előtt meghívódik a `GenerateData()` metódus, amely a tábla aktuális állapota alapján előállítja a sorokhoz és oszlopokhoz tartozó számsorokat (`RowData`, `ColData`).
- `ClearPuzzle()`: a tábla mezőinek üresre állításáért felel.

3.2.4. Játék

Modell. A játékhoz tartozó modell a `Nonogram.Model.Game` névtérben található.

18. ábra. A játék modellje

Nézet.

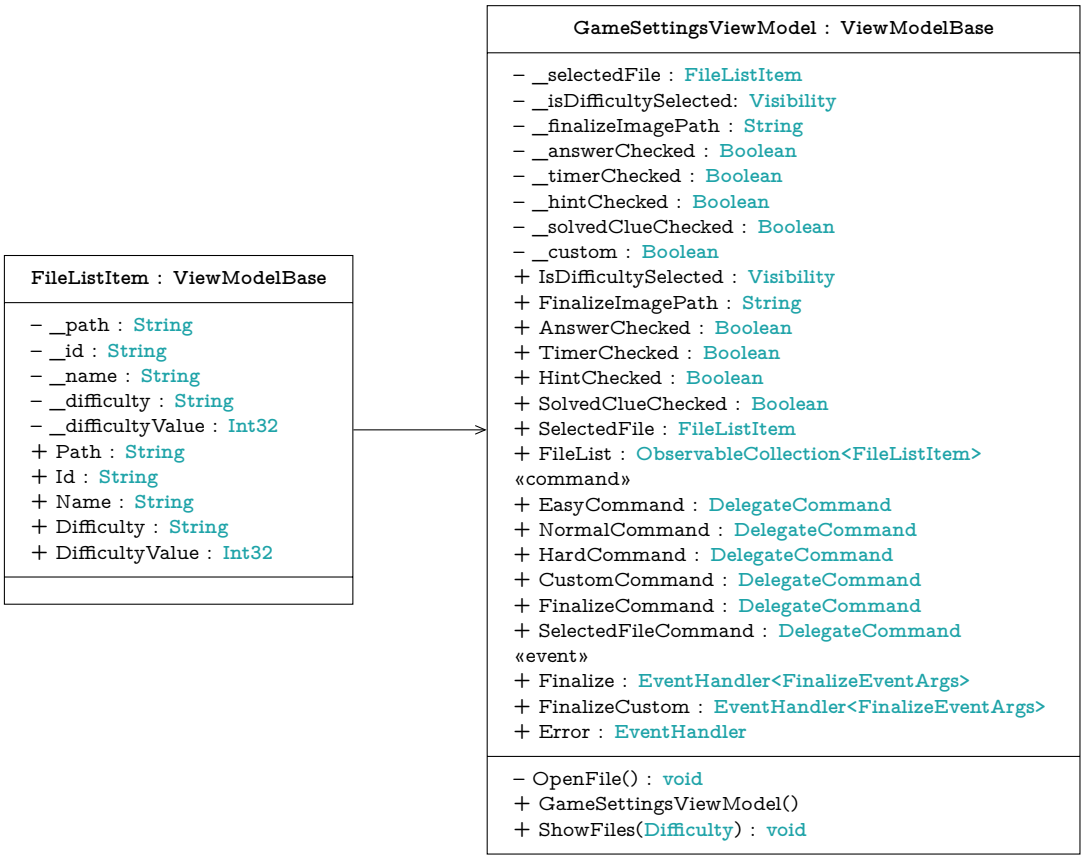
GameView: a funkciógomboknak, a játék információnak és a rejtvény rácsának megjelenítésért felelős nézet.

GameSettingsView: új játék indítása esetén megjelenő ablak, amely felel a programba épített rejtvények, valamint a felhasználó által a tervezőben mentett fájlok kilistázásáért. Továbbá a játékhoz tartozó beállítások választhatóak ki ebben az ablakban.

GameLoadView: már megkezdett és lementett játékok kilistázására szolgáló nézet, amely biztosítja az újbóli megnyitás lehetőségét.

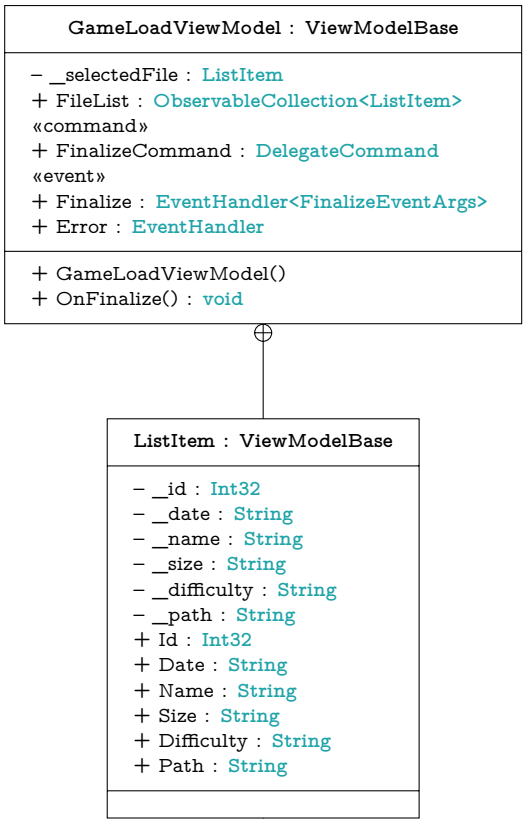
Nézetmodell.

Az előzőekben ismertetett nézetekhez különböző nézetmodellek kerültek implementálásra: a GameViewModel a játék felületének modelljét, a GameSettingsViewModel a fájlok kilistázásáért és a beállításokért felelő nézet modelljét, míg a GameLoadViewModel a mentett játékok megjelenítését végző nézet modelljét tartalmazza. Ezen osztályok diagramjai a következők:



19. ábra. GameSettingsViewModel nézetmodell osztály

20. ábra. A GameViewModel nézetmodell osztály



21. ábra. `GameLoadViewModel` nézetmodell osztály

A meghatározott feladatok implementációja.

A játéktól elvárt feladatok (3.1.3. alfejezet) megoldásai a következő módon lettek implementálva:

- `ShowFiles(Difficulty)`: a nézeten kiválasztott bonyolultságtól függően (paraméterben átadott érték) összegyűjti a `FileList` nevű `ObservableCollection`-be a programhoz csatolt fájlokat, vagy a felhasználó által létrehozott rejtvényeket. A megnyitást a modell `OpenNewGame(Settings)`, illetve `OpenNewGameFromFile(Settings)` metódusai végzik.

- A `GameSettingsViewModel` megfelelő *property*-jei tartalmazzák az egyes funkciók aktív vagy inaktív állapotát. Az `OpenFile()` metódusban ezek alapján létrehozunk egy `Settings` típusú változót, amelyben tároljuk a beállításokat. Esemény hatására (`Finalize` vagy `FinalizeCustom`) átadjuk ezt az `App`-nak, ami kommunikál majd a modellel.

  – Az előnézet megjelenítéséért a `ShowPreview()` metódus felel, amelyben a `PreviewTable` elemeit módosítjuk a játék megoldása alapján. Az `App`-

ban a funkció aktiválásakor létrehozunk egy Timer-t, majd 2 másodperc elteltével leállítjuk azt, és meghívjuk a StopShowingPreview() metódust.

  - A segítségek számának beállítását a modell SetHintCount() metódusa végzi. A funkció aktiválásának hatására először ellenőrizzük, hogy a funkció aktív-e; ha igen, és felfedhető szegmens (CanShowHint()), akkor a modell Hint() függvénye véletlenszerűen keres egy szegmenst, amelyet a nézetmodell felfed.
  - A hátralévő idő tárolására a TimeSpan típusú _timerValue változót használjuk. Attól függően, hogy az időmérő funkció aktív-e, elindítunk egy Timer-t az App-ban, amely másodpercenként csökkenti a rendelkezésre álló időt az AdvanceTime() metódus segítségével. Az idő lejártakor kiváltódik a TimeOut esemény, a tábla elemeit zároljuk (SetInactive()), majd az App egy felugró ablakban (MessageDialog nézet) értesíti a felhasználót.
  - CheckSolvedClues(Int32, Int32, Boolean, Boolean, Boolean): egy mező megváltoztatásának hatására ellenőrizzük, hogy a mező által keresztezett sorban és oszlopban elkészült-e egy szegmens. Ha igen, akkor a megfelelő index(ek) átadásával a nézetmodell jelöli azokat a *clue*-kat a tábla szélein.

- CreateGame(): új rejtvény betöltéséért felel, az OpenNewGame(Settings) függvényből hívódik meg.

  CreateGame(Tuple<FieldType[,], Tuple<Int32, Int32, TimeSpan>>): mentett játék betöltéséért felel, a LoadGame(String, String, Int32) függvényből hívódik meg.

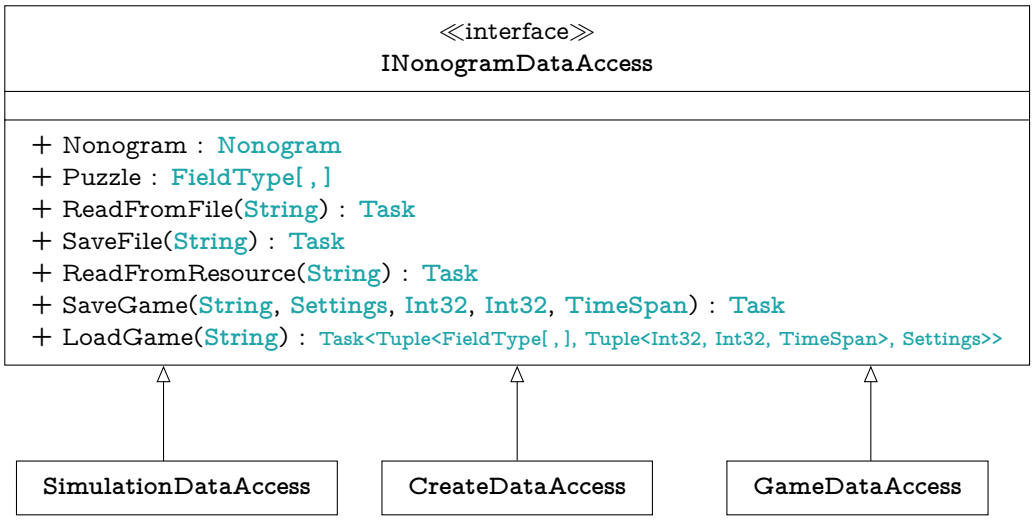
  A nézetmodell az így létrehozott táblák alapján jeleníti meg a rejtvényt a grafikus felületen (GenerateTable()).

- FindWrongFields(): a nézetmodell a ShowWrongFields()-ben lekéri a modell ezen metódusában meghatározott hibásan kitöltött mezőket, majd a kapott koordináta-párokat tartalmazó lista alapján jelöli a megfelelő mezőket a felületen.

- `IncreaseSteps()`: a modell `_stepCount` változóját növeli eggyel minden olyan esetben, amikor a tábla valamely mezőjén módosítás történik.
- `IsSolved()`: a `Solved` esemény akkor váltódik ki, amikor a modell `IsSolved()` függvénye 'igazat' ad eredményül. A tábla zárolását követően az esemény hatására az `App` egy `MessageDialog`-ban értesíti a felhasználót.
- `SaveAsync(String)`: előre meghatározott útvonalra menti a rejtvény aktuális állapotát.
- A `GameLoadViewModel` nézetmodellben listázódnak a már mentett játékok, a betöltést a modell `LoadGame(String, String, Int32)` metódusa végzi.

3.3. Fájlkezelés, fájlok szerkezete

A fájlok megnyitásához, illetve mentéséhez tartozó feladatokat a `Persistence` névtérben található `INonogramDataAccess` interfész osztály végzi. Mivel az egyes részek különböző szerkezetű fájlokat igényelnek, ezért célszerű volt minden rész olvasási és írási műveleteihez saját implementációt írni. Ezen műveletekhez a `StreamReader` és a `StreamWriter` osztályok kerültek felhasználásra.



22. ábra. Az adatkezelésért felelős osztályok szerkezete

A programban felhasznált fájlok szerkezete funkciótól függően változó, hiszen a *Nonogram* alapvető adatain kívül más-más információkra van szükség a szimulációban vagy a játékban.

3.3.1. A szimulációban használt fájlok szerkezete

A szimulációban megtalálható fájlok szerkezete a legegyszerűbb:

- az **első sor** tartalmazza a sorok és oszlopok számát szóközökkel elválasztva (`M N`);
- a következő **`M db sorban`** a sorokhoz tartozó számsorokat találjuk meg, az egyes *clue*-kat szóközökkel elválasztva;
- ezt követi **`N db sor`**, amelyben az oszlopokhoz tartozó számsorokat találjuk.

Az ilyen típusú fájlok `Embedded Resource`-ként kerültek a programba, vagyis mintegy beágyazott fájlokként. Kiolvasásuk a `ReadFromResource()` metódussal történik. A fájlok eléréséhez a programon belül meg kell határozni a címüket, amihez a `System.Reflection` névtérben található `Assembly` nevű osztály van segítségünkre.

3.3.2. A tervezőben használt fájlok szerkezete

A tervezőben létrehozott rejtvény mentésekor az előzőekben ismertetett információkat követően a kitöltött tábla is mentésre kerül. A tábla egy sora a fájl egy sorának feleltethető meg:

- ha a tábla egy mezője áthúzott (vagy üresen lett hagyva), akkor a fájlba 0-t írunk;
- ha a mező sötét, akkor a fájlba 1-es kerül.

Az elérési út, ahová menti a fájlt a program, előre definiált, a felhasználó által nem megváltoztatható.

3.3.3. A játékban használt fájlok szerkezete

Játék kezdésekor kétféle olvasási művelet megkülönböztetése szükséges: a beágyazott fájlok megnyitása `Resource`-ból (`ReadFromResource()`), illetve a tervezőben mentett `txt` fájlok megnyitása (`ReadFromFile()`). Új játék kezdésekor a fájlok szerkezetükben nem különböznek a tervezőben ismertetett szerkezettől.

Játék mentésekor a fájl szerkezete a következőképp módosul:

- A *clue*-k értékei mellett feljegyezzük `boolean` érték formájában (*True* vagy *False*), hogy a megoldás során az adott elem elkészült-e. (Ez a szakaszjelölés funkció esetén lényeges információ.)

- Ha a felhasználó egy mezőt
  - **sötétnek jelölt**, akkor a megfelelő sor megfelelő pozíciójára 1-est írunk;
  - **áthúzottnak jelölt**, akkor a fájlba 0 kerül;
  - **üresen hagyott**, akkor '?' szimbólummal jelöljük a fájlban.
- A tábla aktuális állapotát jelző sorok után a következő információk kerülnek tárolásra:
  - addig megtett lépések száma;
  - hátralévő idő;
  - a négy funkció (előnézet, szakaszjelölés, segítség, időzítő) aktív-e (`boolean` értékek).

3.4. A megoldási módszerek leírása

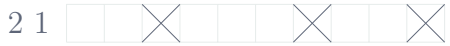
Mivel a dolgozat legösszetettebb feladatának az egyes algoritmusok megalkotása bizonyult, ezért fontosnak tartom, hogy ezen módszereket egy külön fejezetben részletezzem.

A `Nonogram.Model.Simulation` névtérben található a `SolutionTechniques` osztály, amely tartalmazza a logikai módszerek implementációját, valamint a hozzájuk szükséges további metódusokat. Először az ilyen metódusok közül tekintsük át a legfontosabbakat, majd az algoritmusok részletesebb leírására kerül sor.

3.4.1. Szakaszok meghatározása

```
List<Tuple<Int32, Int32>> GetBoundaries(Field[,], Int32, Boolean)
```

Szakasznak nevezzük a tábla széleivel és/vagy áthúzott mezőkkel határolt részt. Ez a függvény visszaadja azon zárt intervallumú koordinátapárok listáját, amelyek az adott sorban vagy oszlopban lévő szakaszok határait jelölik. Ehhez paraméterül megkapja a kétdimenziós táblát, az adott sor vagy oszlop indexét, valamint azt az információt, hogy sorról vagy oszlopról beszélünk-e. Az alábbi ábrát véve alapul a következő listát kapnánk: {(0, 1), (3, 5), (7, 8)}.



23. ábra. Egy példa a szakaszok meghatározásához

3.4.2. Szegmensek meghatározása

`Segments GetSegments(List<Tuple<Int32, Int32>>, Field[,], Int32, Boolean)`

Szegmensnek nevezzük a sor olyan összefüggő részét, amely sötétre színezett mezőkből áll. A visszaadott `Segments` típus valójában egy *alias* a

`Dictionary<Int32, List<Tuple<Int32, Int32>>>`

típusra: a `Dictionary` kulcsa a szakasz indexe, a kulcshoz pedig az adott szakaszon előforduló szegmensek listája tartozik. Az alábbi ábra esetén a függvény eredménye a következő: $\{\langle 1, \{(2, 3)\}\rangle, \langle 2, \{(6, 6), (8, 9)\}\rangle\}$.

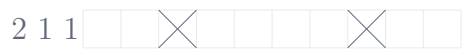
24. ábra. Egy példa a szegmensek meghatározásához

3.4.3. Lehetséges *clue*-k meghatározása szakaszokra

`List<Tuple<List<Int32>, Int32>> GetPossibleClues(List<Values>, Int32, Int32)`

Fontos, hogy minden szakaszra meghatározzuk az oda illeszthető (elférő) *clue*-k indexeit (0-tól kezdve az indexelést). A visszaadott lista olyan párokat tartalmaz, melynek első eleme egy lista, amelyben megadjuk, hogy az adott szakaszra mely indexű *clue*-k férnek el, második eleme pedig a következő szakaszra kerülő következő *clue* indexe (tehát az első olyan index, amely még nem került elhelyezésre a sorban). Paraméterként a *clue*-kat tartalmazó lista, a soron következő *clue* indexe, valamint a vizsgálandó szakasz hossza kerül átadásra (a szakasz indexének ismerete a függvényen belül nem lényeges). Az alábbi ábra alapján például:

$$\texttt{GetPossibleClues(\{2, 1, 1\}, 0, 2)} = \{(\{0\}, 1)\}$$
$$\texttt{GetPossibleClues(\{2, 1, 1\}, 0, 4)} = \{(\{0\}, 1), (\{0, 1\}, 2)\}$$



25. ábra. Egy példa a szakaszonkénti lehetséges *clue*-k meghatározásához

3.4.4. Lehetséges *clue*-k meghatározása szegmensekre

`List<Int32> GetPossibleCluesOnSegments(List<Tuple<Int32, Int32>>, Tuple<Int32, Int32>, Tuple<Int32, Int32>)`

Szükséges, hogy nemcsak szakaszokra, hanem szegmensekre is meghatározzuk a lehetséges *clue*-k indexeit. Ha tudjuk, hogy egy szegmensre csak adott indexű *clue*(-k) illeszthető(ek), akkor abból egyes algoritmusok megállapíthatnak információkat (lsd. *Joining*). A paraméterek: a lehetséges *clue*-k indexei és a hozzájuk tartozó értékek, az adott szakasz határai, valamint a vizsgált szegmens. Az alábbi ábrán látható példa eredménye a következő:

`GetPossibleCluesOnSegments({(0, 1), (1, 2), (2, 3)}, (0, 14), (3, 3))` $= \{0, 1\}$
`GetPossibleCluesOnSegments({(0, 1), (1, 2), (2, 3)}, (0, 14), (7, 8))` $= \{1, 2\}$
`GetPossibleCluesOnSegments({(0, 1), (1, 2), (2, 3)}, (0, 14), (12, 12))` $= \{2\}$.

Az algoritmus kiszámítja a szegmens előtt és után fennmaradó üres helyek hosszát, majd vizsgálja, hogy egy adott *clue* illesztése esetén az előtte szereplő *clue*-k kiférnek-e a szegmens elé, illetve az utána lévők a szegmens után. Ha igen, akkor bekerül a listába.

26. ábra. Egy példa a szegmensekre illeszthető *clue*-k meghatározásához

> **Megjegyzés**: ez az eredmény tovább szűkíthető annak vizsgálatával, hogy két szomszédos szegmens összekapcsolása lehetséges-e. Ha a példa utolsó szegmensét tekintjük, akkor látjuk, hogy arra kizárólag a 2-es indexű *clue* illeszthető minden megoldás esetén. Az előtte lévő szegmens listájából ennek hatására ez a *clue* kikerül, hiszen a két szegmens összekapcsolása nem lehetséges (mivel 6-hosszú szegmenst eredményezne). Ezt a logikát alkalmazva minden egyértelműen meghatározható illeszkedésre (minden, egyelemű listával rendelkező szegmensre) a példánk esetén a listák a következőképp módosulnak: `{0}`, `{1}`, `{2}`.

3.4.5. A szimuláció „lelke”: az összes lehetséges megoldás meghatározása egy sorra

`Solutions FindAllPossibleSolutions(List<Values>, List<Tuple<Int32, Int32>>, Int32, Int32, Dictionary<Int32, List<Int32>>, Positions, Segments)`

A `Solutions` szerkezete egy fához hasonlítható: szakaszindexenként tárolja az arra a szakaszra illeszthető *clue*-k indexeinek listáját. Mivel egy szakaszhoz (a fa szintjéhez) több megoldás is tartozhat, ezért a következő szint megoldása is többféleképpen alakulhat. Ehhez a függvényt rekurzívan definiáljuk. A paraméterek rendre a következők: *clue*-k listája, szakaszok listája, aktuális szakasz indexe, aktuális (még nem elhelyezett) *clue* indexe, fix helyen már meghatározott *clue*-k listája szakaszonként, szakaszonként lehetetlen *clue*-k listája, szegmensek. Kezdetben az indexek 0-k, a fix és lehetetlen indexek listái `null`-ok. Az algoritmus a következő logikára épül:

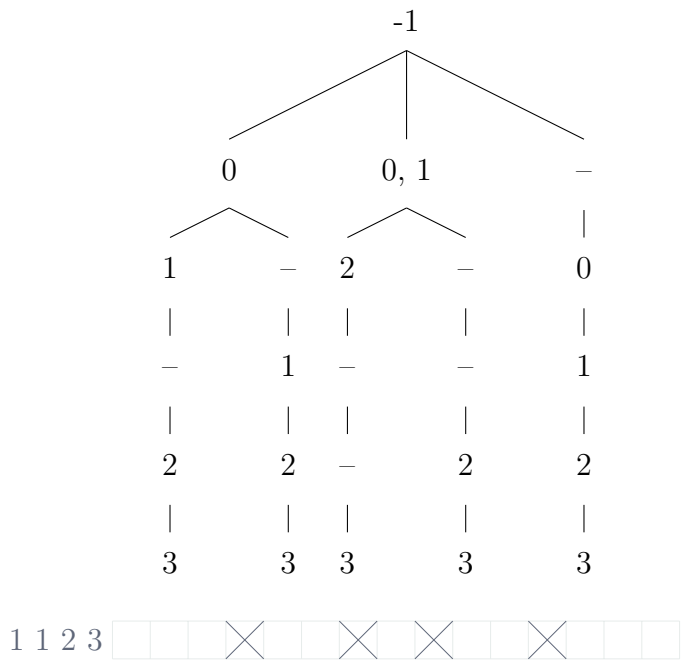
(1) Határozzuk meg az aktuális szakaszra (kezdetben a 0. indexűre) illeszthető *clue*-k indexeit (`GetPossibleClues`).

(2) Menjünk végig az előző pontban meghatározott lehetséges megoldásokon. Ha a vizsgált szakasz nem az utolsó (tehát van még utána szakasz), akkor hívjuk meg a függvényt újból, de most már a következő szakasz indexét, és a soron következő – még el nem helyezett – *clue* indexét átadva paraméterként (a `GetPossibleClues` eredményének 2. része).

(3) Ha elfogytak a *clue*-k (tehát mindegyiket elhelyeztük valahová), akkor a részfát elhelyezzük a `Solutions`-ben; ha nem, de a szakaszok végére értünk, akkor az egy helytelen megoldási út, amivel nem foglalkozunk.

Később meghatározhatjuk különböző feltételek alapján, hogy egy szakaszra már biztosan nem (vagy biztosan igen – ekkor pedig a többi szakaszra már biztosan nem) illeszkedhet egy (vagy több) *clue*. Így folyamatosan szűkíthető a lehetséges megoldások száma, aminek hatására az algoritmusok szűkebb halmazzal dolgozhatnak.

Az alábbi példára a függvény eredménye szemléletesen ábrázolva a következő:



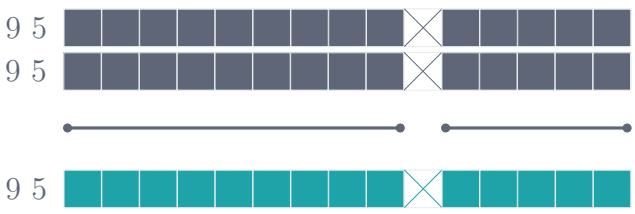
27. ábra. Egy példa a lehetséges megoldások meghatározásához

A gyökér kivételével minden szint egy szakasznak feleltethető meg. A fa egy ága egy lehetséges megoldást reprezentál. Ezeken a képzeletbeli ágakon végezzük a műveleteket, és keressük a helyes megoldási utat.

3.4.6. Átfedések keresése (*Simple Boxes*)

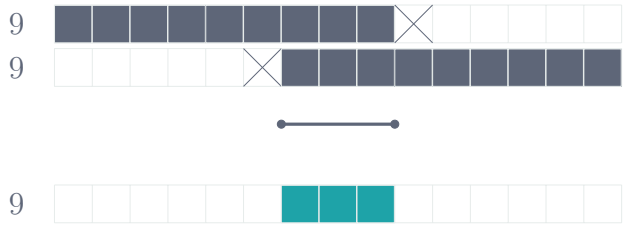
Elmélet. Kezdetben ez az egyetlen módszer, amit alkalmazhatunk. Olyan mezőket keresünk, amelyekről biztosan megállapíthatjuk, hogy sötétre kell őket színezni. Ehhez képzeljük el, hogy a *clue*-kat először balról, majd jobbról helyezzük el egyetlen üres mező kihagyásával közöttük. Így megkapjuk azt a két szélsőséges esetet, amelyek alapján könnyen láthatóvá válik, hogy keletkezik-e átfedés. Tekintsünk néhány példát:

A legkézenfekvőbb példa az olyan számsor, amely pontosan kitölti a rendelkezésre álló teret (28. ábra). Ekkor a két szélsőséges eset ekvivalens és a blokkok készen vannak.



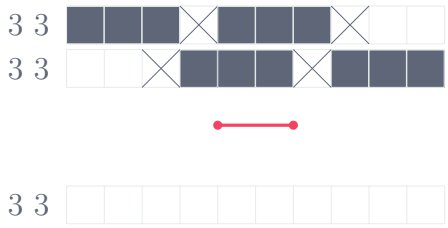
28. ábra. *Simple Boxes* teljes átfedés esetén

Olyan sor esetén is keletkezik átfedés, amelyhez pontosan egyetlen *clue* tartozik és annak értéke nagyobb mint a sor szélességének fele (tehát például 15-hosszú sor esetén a *clue* értéke legalább 8):



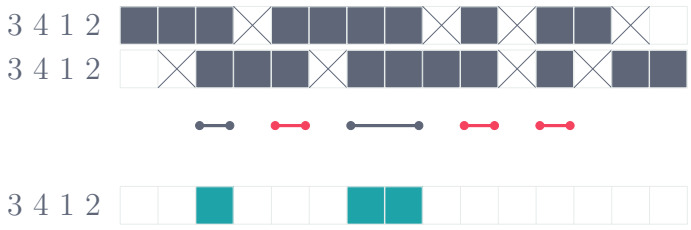
29. ábra. *Simple Boxes* egyetlen *clue* esetén

Az átfedések nem minden esetben adnak „biztos” eredményt. Tekintsük a(z) 30. ábrát: látható, hogy az 5–6. pozíciókon mindkét lehetséges kitöltés esetén sötét blokkokat találunk, de mégsem jelölhetjük őket sötétnek. Ennek oka, hogy az első kitöltésben a második 3-ashoz, a második kitöltésben pedig az első 3-ashoz tartozik az 5-ös és 6-os pozíció.



30. ábra. *Simple Boxes* nem egyértelmű esete

Vegyünk egy olyan példát, ahol a helyes és „helytelen” átfedések egyaránt előfordulnak:



31. ábra. *Simple Boxes* helyes és helytelen átfedésekkel

A módszer implementálása. Az algoritmust szakaszonként alkalmazzuk, méghozzá úgy, hogy a szakasz összes lehetséges megoldásainak metszetét vesszük. A metszet adja a **biztosan** az adott szakaszra kerülő *clue*-k listáját. (Például: a fenti 27. ábra esetén az utolsó szakasz metszete nem lesz üres, hiszen minden megoldási út

esetén az a szint tartalmazza a 3. indexű *clue*-t. Erre már alkalmazható a következő algoritmus.)

Ezen megoldási technikához egyszerűen megfogalmazható matematikai eljárás tartozik, amely könnyedén implementálható. Szemléltetéséhez vegyük példának a(z) 31. ábrán szereplő sort. Továbbá legyen N a *clue*-k száma, illetve *clues* a számsort tartalmazó N-elemű tömb. A módszer lépései a következők:

(1) Vegyük a *clue*-k összegét, adjuk hozzá az üres helyek számát, majd vonjuk ki a szakasz hosszából:

$$d = length - \left(\left(\sum_{i=0}^{N-1} clues[i] \right) + N - 1 \right),$$

itt:

$$d = 15 - ((3 + 4 + 1 + 2) + 4 - 1) = 2.$$

(2) Az előzőleg kapott értéknél hosszabb *clue*-k átfedést fognak adni, méghozzá annyi mezőt, amennyi a két szám különbsége.

$$\forall i \in [0, N) : c_i = clues[i] - d,$$

itt:

$$c_0 = 3 - 2 = 1,$$
$$c_1 = 4 - 2 = 2,$$
$$c_2 = 1 - 2 = -1,$$
$$c_3 = 2 - 2 = 0.$$

Vagyis a 3-as *clue*-ból 1 db mezőt, a 4-es *clue*-ból 2 db mezőt kapunk meg biztosan. A többi *clue*-hoz 1-nél kisebb értéket kaptunk, így azokkal nem foglalkozunk.

(3) Már csak a pontos pozíciókat (zárt intervallum) kell meghatároznunk:

$$\forall i \in [0, N),\ c_i > 0 : p_i = \left(start + \sum_{j=0}^{i} clues[j] + i - c_i,\ start + \sum_{j=0}^{i} clues[j] + i - 1 \right),$$

ahol *start* az aktuálisan vizsgált szakasz kezdőindexe (itt ez 0). Itt:

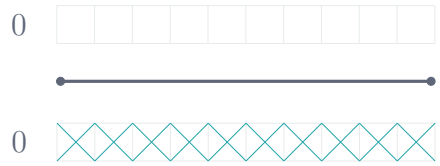
$$p_0 = (0 + 3 + 0 - 1, 0 + 3 + 0 - 1) = (2, 2),$$
$$p_1 = (0 + 7 + 1 - 2, 0 + 7 + 1 - 1) = (6, 7),$$

amely eredmény az ábrán is látható.

3.4.7. Üres helyek keresése (*Simple Spaces*)

Elmélet. Az algoritmus lényege, hogy meghatározzuk a biztosan üres helyeket különböző feltételek alapján.

A legegyszerűbb eset, amikor a számsor csupán egyetlen elemből, a 0-ból áll: ekkor minden mezőt üresre kell állítani (32. ábra).

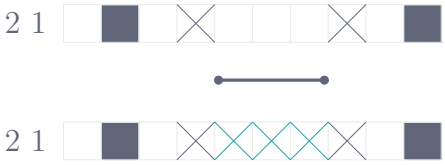


32. ábra. *Simple Spaces* 0-t tartalmazó számsor esetén

Előfordul, hogy egy szakasz hossza minden *clue* értékénél kisebb: ekkor biztosan nem kerülhet oda egyetlen elem sem, így a mezői üresek lesznek:

33. ábra. *Simple Spaces* olyan szakaszra, ahová nem fér egyetlen *clue* sem

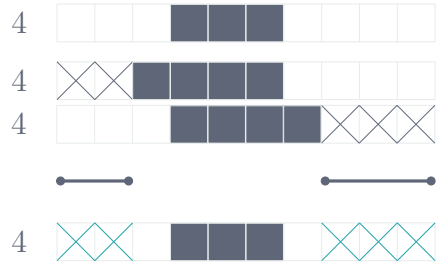
Ha már minden *clue* a „helyére” került, és maradt kitöltetlen szakasz, akkor annak mezői is üresek lesznek:



34. ábra. *Simple Spaces* olyan szakaszra, ahol nem szerepelhet már szegmens

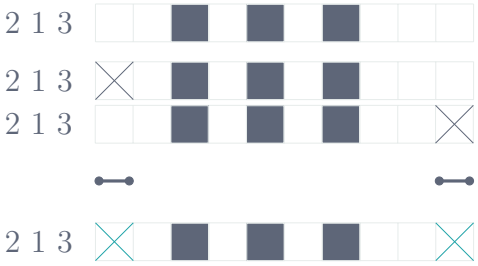
3.4.8. A higany módszer (*Mercury*)

Elmélet. Olyan sort (vagy szakaszt) keresünk, amelyben csak egyetlen blokk fog szerepelni. Ha találunk a szakaszban már kitöltött mezőt, akkor alkalmazható ez a módszer, amelynek lényege, hogy „képzeletben” kiegészítjük a hiányzó mezőkkel mindkét irányban a szegmenst (35. ábra). A fennmaradó mezők így biztosan üresek lesznek.



35. ábra. *Mercury* alapesete

A módszer kiterjeszthető olyan (több *clue*-t tartalmazó) szakaszokra is, amelyekben a balról (vagy jobbról) előforduló első szegmensre csak egyetlen *clue* illeszthető az összes lehetséges megoldás figyelembevételével (36. ábra).



36. ábra. *Mercury* több *clue* esetén

A módszer implementálása. Szakaszonként vesszük az összes lehetséges megoldást. A lehetséges megoldások halmazából gyűjtsük ki a balról és jobbról vett szélső indexeket egy-egy listába. Ha valamely lista csak azonos értékeket tartalmaz és az abból az irányból elsőként előforduló szegmensre csak az az érték illeszthető, akkor vizsgálható, hogy lesz-e üres mező. Legyen

`B` : `List<Tuple<Int32, Int32>>` a szakaszok,

`S` : `Dictionary<Int32, List<Tuple<Int32, Int32>>>` a szegmensek.

Az esetlegesen keletkező üres mezők pozícióit (zárt intervallum) a következőképpen számítjuk az i. szakaszon:

- **balról**: $(B[i]_1,\ S[i][0]_1 - h - 1)$, ahol h a szakasz első szegmensének hiányzó mezőinek száma,
- **jobbról**: $(S[i][l]_2 + h + 1, B[i]_2)$, ahol l az utolsó szegmens indexe és h az utolsó szegmens hiányzó mezőinek száma.

A fenti két példára a következő eredményeket kapjuk (feltesszük, hogy a szakaszok kezdőindexe mindkét esetben 0):

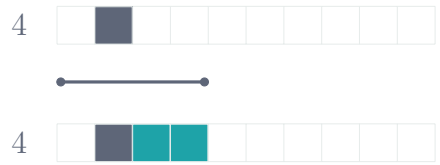
$$\begin{aligned}
\textbf{a(z) 35. ábra esetén:}\quad & (0, 3 - (4 - 3) - 1) = (0, 1) \\
& (5 + (4 - 3) + 1, 9) = (7, 9) \\
\textbf{a(z) 36. ábra esetén:}\quad & (0, 2 - (2 - 1) - 1) = (0, 0) \\
& (6 + (3 - 1) + 1, 9) = (9, 9).
\end{aligned}$$

3.4.9. Ragasztás (*Glue*)

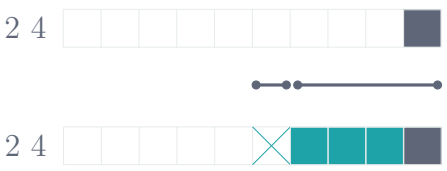
Elmélet. Keressük azokat a szegmenseket, melyek a határhoz vagy a határként viselkedő biztosan üres mezőhöz elég közel vannak. Egy szegmens elég közel van egy határhoz, ha legelső mezője a határtól számítva legfeljebb

$$(\text{rá illeszkedő } \textit{clue} \text{ értéke} - 1)$$

távolságra található. Ha a szakasz legelső (vagy legutolsó) mezőjén találunk kitöltött mezőt, akkor a szegmens készen lesz, elválaszthatjuk üres mezővel (38. ábra).

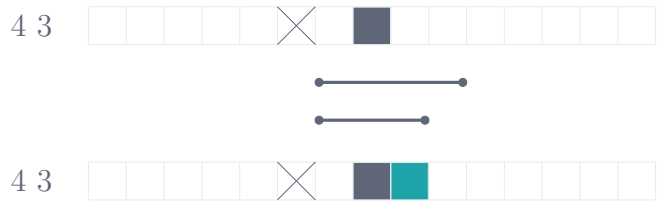


37. ábra. *Glue* egyértelmű alapesete



38. ábra. *Glue* elkészülő szegmens esetén

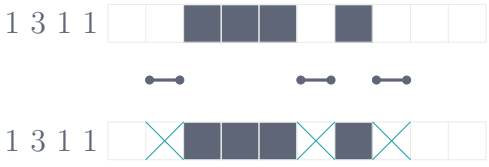
Az algoritmus kiterjeszthető oly módon, hogy nemcsak egyértelmű megoldást tartalmazó szakaszokra alkalmazzuk, hanem több különböző megoldás esetén meghatározzuk a szélső szegmensekre illeszthető összes *clue*-t, majd vesszük azok minimumát (39. ábrán a szegmens 4-es és 3-as egyaránt lehet, $\min(4, 3) = 3$).



39. ábra. *Glue* több megoldást tartalmazó szakaszra

3.4.10. Elválasztás (*Punctuation*)

Elmélet. A *Simple Spaces* egy speciális esete, amikor a kész szegmenst felismerve azonnal lezárjuk azt biztosan üres mezőkkel a szélein.



40. ábra. Példa *Punctuation*-re

A módszer implementálása. Olyan szegmenseket keresünk, amelyre egyetlen (vagy több, de azonos értékű) *clue* illeszkedhet, és annak értékével a szegmens hossza megegyezik. Egy `S : Tuple<Int32, Int32>` szegmens esetén ekkor:

- ha $S_1 - 1 \geq 0$, akkor $(S_1 - 1)$. pozíció biztosan üres;
- ha $S_2 + 1 < l$, akkor $(S_2 + 1)$. pozíció biztosan üres, ahol l a sor hossza.

3.4.11. Összekapcsolás (*Joining*)

Elmélet. Az algoritmus több szegmens esetén alkalmazható, hiszen lényege, hogy két szomszédos szegmensről megállapítsuk, a köztük lévő üres hely(ek) kitöltésével azok összekapcsolhatóak-e. Ennek eldöntésére számos feltételt figyelembe kell vennünk. A legegyszerűbb eset, amikor egy szakaszra egyetlen megoldásunk van, amely egyetlen *clue*-t tartalmaz, a szegmensek száma viszont nagyobb 1-nél, például:

41. ábra. *Joining* legegyszerűbb esete

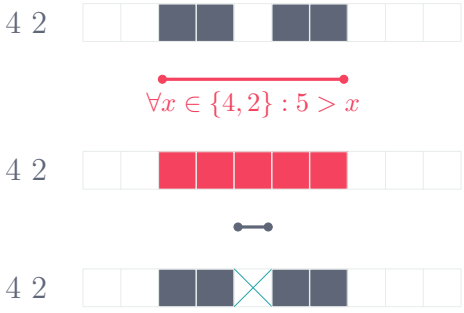
A következő ábrán az első szegmensre az 1-es vagy 7-es, a másodikra és a harmadikra viszont csak a 7-es illeszthető, ezért az utóbbi kettőt biztosan összekapcsolhatjuk. Ha 1 7 helyett a számsor például 3 7 lenne, akkor az algoritmusnak nem lenne eredménye.

42. ábra. *Joining* több *clue* esetén

A módszer implementálása. Az algoritmus csak olyan szakaszokra alkalmazható, amelyeknek egyértelmű megoldása van. A módszer alapja az egyes szegmensekre illeszkedő *clue*-k megállapítása. Ha két szomszédos szegmensre ugyanazon *clue* illeszthető (és csakis az), akkor biztosan összekapcsoljuk azokat (a köztük lévő mezőket kitöltöttre állítjuk).

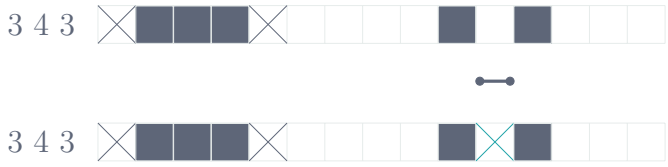
3.4.12. Szétválasztás (*Splitting*)

Elmélet. Az algoritmust olyan szomszédos szegmensek esetén alkalmazzuk, amelyek között pontosan egy kitöltetlen helyet találunk. Ha adott feltételek teljesülnek, akkor azt a mezőt biztosan üresnek jelöljük. Az egyik feltétel, hogy összekapcsolva túl hosszú szegmenst eredményezne (olyat, amely minden *clue*-nál nagyobb), például:



43. ábra. *Splitting* legegyszerűbb esete

A másik feltétel a *Joining* módszer alapjára épül: határozzuk meg a szegmensekre illeszthető *clue*-k listáját. Ha két szomszédos szegmens esetén a listák diszjunktak, akkor biztosan elválasztjuk őket. (Elegendő lenne ezt a feltételt vizsgálnunk, hiszen az előzőleg felvázolt esetben is teljesülne, azonban ha az előző feltétel teljesül, akkor az utóbbival járó „felesleges" számítások elkerülhetőek.)



44. ábra. *Splitting* a második feltétel alapján

A módszer implementálása. Az algoritmus tulajdonképpen a *Joining* módszer inverzének tekinthető, hiszen azt vizsgáljuk, hogy két szomszédos szegmens biztosan nem összekapcsolható. Jelölje

S_b : `Tuple<Int32, Int32>` a bal oldali,

S_j : `Tuple<Int32, Int32>` a jobb oldali szegmenst,

L : az adott szakaszra illeszthető *clue*-kat tartalmazó listát.

Az algoritmus lépései a következők:

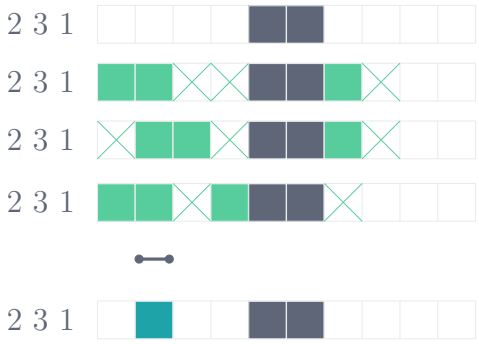
(1) **Első feltétel:** ha $\forall l \in L : (S_{j_2} - S_{b1} + 1) > l$, akkor az $S_{b2} + 1$. pozíción szereplő mező biztosan üres.
A fenti példára (43. ábra): $(6 - 2 + 1) = 5 > 4$ és $5 > 2 \Longrightarrow (3 + 1) = 4$. pozíció üres.

(2) **Második feltétel:** legyen C_b az S_b-re illeszthető *clue*-k listája, C_j pedig az S_j-re illeszthető *clue*-k listája. Határozzuk meg ezeket. Ha $C_b \cap C_j = \emptyset$, akkor az $S_{b2} + 1$. pozíció biztosan üres.
A fenti példára (44. ábra): $C_b = \{1\}$, $C_j = \{2\}$, $\{1\} \cap \{2\} = \emptyset \Longrightarrow (9+1) =$ 10. pozíció üres.

3.4.13. Kettős elhelyezés (*Double Position*)

Elmélet. Tegyük fel, hogy egy szakaszon egynél több szegmens fog szerepelni és már találunk egy (nem kész) szegmenst, amelyről pontosan megállapíthatjuk,

hogy melyik *clue*-hoz fog tartozni. Ekkor a szegmens hiányzó mezőinek figyelembevételével a szakaszt „képzeletben” három részre oszthatjuk: a szegmens előtti, a szegmenst tartalmazó, és a szegmens utáni részre. A cél, hogy az adott szegmens elé, illetve után kerülő *clue*-k illesztésével biztosan üres, vagy biztosan kitöltött mezőket találjunk. Az algoritmus a *Simple Boxes* alapjára épül, vagyis megkeressük a két szélsőséges esetet (itt: 2–4 eset), amiből próbálunk következtetni. Például:



45. ábra. *Double Position*: a szegmens csak a 3-as lehet

Fontos feltétel, hogy a kezdeti szegmens pontosan kétféle pozíciót vehessen fel a hiányzó mezőivel kiegészítve – ha egyetlen mező hiányzik a szegmensből, akkor ez egyértelműen teljesül, hiszen vagy az elejére, vagy a végére illesztjük azt az egy elemet. Több hiányzó mező esetén csak akkor lehet mindössze kétféle pozíció, ha a szegmenshez olyan közel van a határ (vagy egy másik szegmens), hogy abba az irányba nem tudnánk kiegészíteni a hiányzó mezőivel (vagy a másik szegmenssel nem összekapcsolható).

3.4.14. Tippelgetés (*Contradiction*)

Elmélet. A fent ismertetett algoritmusok nem minden esetet fedhetnek le egyes – többnyire bonyolultabb – rejtvények esetében, így előfordul, hogy a megoldás elakad. Ekkor egy tetszőleges, még ismeretlen állapotú mezőt sötétnek (vagy biztosan üresnek) jelölünk, majd folytatjuk a megoldást a már ismert módszerek segítségével. Ha hibás végeredményt kapunk, akkor a tippelt mező állapota helytelen, így biztosan az ellenkezője lesz helyes.

A módszer implementálása. A megoldás akkor akad el, ha nincs már olyan sor vagy oszlop, amelyre egy iterációban valamely algoritmus eredményt adna, de

van még olyan mező, amelynek nem ismerjük az állapotát.

(1) Ebben az esetben tároljuk a rejtvény állapotát (tehát a mezők értékeit, a helyes lépéseket tartalmazó sort, stb.).

(2) Keressünk egy ismeretlen állapotú mezőt: jelöljük kitöltöttnek.

(3) Futtassuk tovább az algoritmusokat. Ekkor több eset lehetséges:

   (a) Ha ismét elakad a megoldás, akkor állítsuk vissza az (1)-es pontban leírt állapotra a rejtvényt, majd tippeljünk a (2)-es pontban lévő mezőtől különböző ismeretlen állapotú mezőt kitöltöttnek.

   (b) Ha helytelen végeredményre jutunk, akkor a (2)-es pontban lévő mező biztosan üres.

   (c) Helyes végeredményt kapunk, a megoldás így elkészül.

> **Megjegyzés**: azon rejtvények, ahol tippelés szükséges, sokkal nagyobb futási időt fognak mutatni, hiszen helytelen tippelések esetén számos helytelen megoldási utat fog végigjárni az algoritmus.

3.5. A megoldási módszerek eredményei számokban

A logikai módszerek látványos szemléltetése mellett célom volt megfigyelni, hogy az algoritmus futási ideje, illetve lépésszáma a rejtvények méretétől és bonyolultságától függően hogyan változik. (Ez indokolja a konstans méretű feladványok alkalmazását, habár a program képes kezelni tetszőleges méretű rejtvényeket is.)

> **Megjegyzés**: a félkövér betűkkel kiemelt rejtvényekhez szükség volt a tippelgetés (`Contradiction`) módszerére.

20 × 20	2–3	A Month [10]	0,024 s	115
		Man with pipe [11]	0,033 s	115
		Tree [12]	0,027 s	145
	4–5	Moon [13]	0,114 s	182
		Deer [14]	0,086 s	183
		Daffodil [15]	0,120 s	215
		The first bird [16]	0,122 s	237
		Anubis [17]	0,086 s	161
	6–7	Hedgehog [18]	0,179 s	237
		Cow [19]	0,143 s	233
		Chicken [20]	0,233 s	243
		Fruit fly [21]	0,335 s	190
		Fox [22]	0,719 s	233
		Cat with flower [23]	6,289 s	232
	8–	**Komar (Mosquito)** [24]	0,721 s	225

46. ábra. A 20 × 20-as rejtvények futási ideje és lépésszáma

		2 – 3	4 – 5	6 – 7	8–
Lépések száma:		125	196	228	225
Futási idő	**tippelés nélkül:**	0,028 s	0,106 s	0,185 s	NA
		0,106 s			
	tippeléssel:	0,028 s	0,106 s	1,316 s	0,721 s
		0,615 s			

47. ábra. A 20 × 20-as rejtvények átlagértékei

30 × 30	2–3	Gift [25]	0,110 s	218
		Floppy disk [26]	0,161 s	183
		Romance [27]	0,044 s	167
	4–5	Sherlock [28]	0,073 s	236
		Mountains [29]	0,247 s	287
		Winter Night [30]	0,360 s	429
		Hipster [31]	0,129 s	372
		Bighorn [32]	0,156 s	306
	6–7	Moose [33]	0,371 s	432
		The hippopotamus who... [34]	3,286 s	454
		Howling Wolf [35]	0,828 s	474
		Baby Dragon [36]	0,522 s	452
		Squirrel [37]	0,501 s	431
	8–	**The Statue Of Liberty** [38]	4,751 s	394
		Surfer [39]	1,080 s	489

48. ábra. A 30 × 30-as rejtvények futási ideje és lépésszáma

		2 – 3	4 – 5	6 – 7	8–
Lépések száma:		189	326	449	442
Futási idő	**tippelés nélkül:**	0,105 s	0,193 s	0,465 s	1,080 s
		0,313 s			
	tippeléssel:	0,105 s	0,193 s	1,102 s	2,915 s
		0,841 s			

49. ábra. A 30 × 30-as rejtvények átlagértékei

30×30	2–3	Ship [40]	0,120 s	248
		Lighthouse [41]	0,211 s	256
		Shy girl [42]	0,192 s	365
	4–5	Vampire [43]	0,247 s	435
		Black Queen [44]	0,420 s	633
		Bear [45]	0,274 s	467
		Corgi [46]	0,265 s	482
		Griffin [47]	0,286 s	504
	6–7	Joyful dog [48]	0,761 s	617
		Lone wolf [49]	0,774 s	761
		Kabuki (Japanese dance-drama) [50]	0,418 s	553
		Pharaoh [51]	1,594 s	633
		Japanese woman [52]	0,725 s	708
	8–	The lady in the hat [53]	0,901 s	748
		Mother's Legacy [54]	2,340 s	802

50. ábra. A 40×40-es rejtvények futási ideje és lépésszáma

		$2-3$	$4-5$	$6-7$	$8-$
Lépések száma:		290	504	654	775
Futási idő	**tippelés nélkül:**	0,174 s	0,298 s	0,854 s	0,901 s
		0,513 s			
	tippeléssel:	0,174 s	0,298 s	0,854 s	1,620 s
		0,635 s			

51. ábra. A 40×40-es rejtvények átlagértékei

3.6. Tesztelés

A jól egységtesztelhető részekre a `NonogramTest` projektben egységtesztek kerültek megvalósításra. A további funkciók tesztelése manuálisan történt.

3.6.1. Manuális tesztek

- **Szimuláció**:
  - Fájl betöltésének tesztelése.
  - A megoldási folyamat indításának/szüneteltetésének tesztelése.
  - A megoldási folyamat léptetésének tesztelése. Teljesül, hogy léptetni csak szüneteltetett állapotban lehetséges.
  - A kirajzolás gyorsításának, illetve lassításának tesztelése.
- **Tervező**:
  - A rejtvény mentésének tesztelése: a fájl formátuma megfelelő és az előre meghatározott mappába kerül.
  - A fájlnévre vonatkozó szabályok teljesülésének tesztelése.
- **Játék**:
  - Fájlok listázásának tesztelése: a felhasználó által létrehozott fájlok megjelennek a megfelelő menüpont alatt.
  - Fájl betöltésének tesztelése.
  - A játék mentésének tesztelése: a fájl formátuma megfelelő és az előre meghatározott mappába kerül.
  - Játék betöltésének tesztelése: a kezdetekben kiválasztott funkciók aktívak továbbra is, a mentéskor fennálló állapot a betöltéskor keletkező állapottal megegyezik.

3.6.2. Egységtesztek

- **Tervező**:
  - `NewPuzzleTest()`: új rejtvény létrehozásakor a megadott értékeknek megfelelő sorból és oszlopból épül fel a tábla.
  - `StepTest()`: kattintás hatására az adott mező értéke a kattintásnak megfelelőre változik.
  - `GenerateTest()`: a tábla állapotának megfelelően kerülnek kiszámításra a számsorok.
- **Játék**: A játék egységteszteinek elkészítéséhez szükséges volt egy tesztrejtvény létrehozása – ennek felépítésére szolgál a `TestUtilities` osztály.
  - `StepTest()`: egy adott mező állapota a kattintástól függően a megfelelő értéket veszi fel.
  - `WrongFieldsTest()`: a helytelen mezők megkeresésének tesztelése.
  - `HintTest()`: a beállításoktól függően megfelelő számú segítség áll rendelkezésre.
  - `CanShowHintTest()`: a tábla állapotától függően fedhető fel szegmens.
  - `TimerTest()`: a rejtvény bonyolultságától függően kerül beállításra a megfejtésre rendelkezésre álló idő.

3.7. Továbbfejlesztés lehetőségei

A dolgozatban a kétállapotú rejtvényekre esett a fókusz, azonban a *Nonogramoknak* létezik színes változatuk is, amelyekben egy mező állapota sokféle lehet, nem csupán fekete vagy fehér (üres). A kód szerkezetileg alkalmas arra, hogy könnyedén felkészítsük a színes rejtvények kezelésére, illetve az azokhoz szükséges megoldó algoritmusok implementálására.

Emellett a szimuláció modelljének kibővítésével elérhetővé válhatna a több megoldással rendelkező rejtvények összes lehetséges megoldásának megtalálása és tárolása.

Hivatkozások

[1] Cserép Máté: Eseményvezérelt alkalmazások fejlesztése II., 7. előadás: WPF alkalmazások architektúrája, https://mcserep.web.elte.hu/data/education/2017-2018-1_EVA2/elte_eva2_ea07.pdf, 2019. március

[2] Cserép Máté: Eseményvezérelt alkalmazások fejlesztése II., 8. előadás, Összetett WPF alkalmazások, https://mcserep.web.elte.hu/data/education/2017-2018-1_EVA2/elte_eva2_ea08.pdf, 2019. március

[3] C# Reference, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/, 2019. május

[4] Windows Presentation Foundation Framework, https://docs.microsoft.com/en-us/dotnet/framework/wpf/, 2019. május

[5] Grafilogika, Wikipédia, https://hu.wikipedia.org/wiki/Grafilogika, 2019. május

[6] Nonogram, Wikipédia, https://en.wikipedia.org/wiki/Nonogram, 2019. május

[7] Survey Of Paint-By-Number Puzzle Solvers, https://webpbn.com/survey/, 2019. május

[8] Nonograms, http://www.nonograms.org, 2019. május

---

[9] Chicken, http://www.nonograms.org/nonograms/i/2877, 2019. március

[10] A Month, http://www.nonograms.org/nonograms/i/11591, 2019. március

[11] Man with pipe, http://www.nonograms.org/nonograms/i/384, 2019. március

[12] Tree, http://www.nonograms.org/nonograms/i/1068, 2019. március

[13] Moon, http://www.nonograms.org/nonograms/i/14635, 2019. március

[14] Deer, http://www.nonograms.org/nonograms/i/8543, 2019. március

[15] Daffodil, http://www.nonograms.org/nonograms/i/18841, 2019. március

[16] The first bird, http://www.nonograms.org/nonograms/i/18647, 2019. március

[17] Anubis, http://www.nonograms.org/nonograms/i/8495, 2019. március

[18] Hedgehog, http://www.nonograms.org/nonograms/i/3345, 2019. március

[19] Cow, http://www.nonograms.org/nonograms/i/4194, 2019. március

[20] Chicken, http://www.nonograms.org/nonograms/i/6486, 2019. március

[21] Fruit fly, http://www.nonograms.org/nonograms/i/1946, 2019. március

[22] Fox, http://www.nonograms.org/nonograms/i/7752, 2019. március

[23] Cat with flower, http://www.nonograms.org/nonograms/i/3967, 2019. március

[24] Komar (Mosquito) http://www.nonograms.org/nonograms/i/3166, 2019. március

[25] Gift, http://www.nonograms.org/nonograms/i/15057, 2019. március

[26] Floppy disk, http://www.nonograms.org/nonograms/i/17126, 2019. március

[27] Romance, http://www.nonograms.org/nonograms/i/556, 2019. március

[28] Sherlock, http://www.nonograms.org/nonograms/i/22535, 2019. március

[29] Mountains, http://www.nonograms.org/nonograms/i/527, 2019. március

[30] Winter Night, http://www.nonograms.org/nonograms/i/21916, 2019. március

[31] Hipster, http://www.nonograms.org/nonograms/i/19698, 2019. március

[32] Bighorn, http://www.nonograms.org/nonograms/i/3502, 2019. március

[33] Moose, http://www.nonograms.org/nonograms/i/7731, 2019. március

[34] The hippopotamus who was afraid of vaccinations, http://www.nonograms.org/nonograms/i/11200, 2019. március

[35] Howling Wolf, http://www.nonograms.org/nonograms/i/19524, 2019. március

[36] Baby Dragon, http://www.nonograms.org/nonograms/i/19999, 2019. március

[37] Squirrel, http://www.nonograms.org/nonograms/i/4107, 2019. március

[38] The Statue Of Liberty, http://www.nonograms.org/nonograms/i/2162, 2019. március

[39] Surfer http://www.nonograms.org/nonograms/i/5775, 2019. március

[40] Ship, http://www.nonograms.org/nonograms/i/15391, 2019. április 4.

[41] Lighthouse, http://www.nonograms.org/nonograms/i/15390, 2019. április 4.

[42] Shy girl, http://www.nonograms.org/nonograms/i/16889, 2019. április 4.

[43] Vampire, http://www.nonograms.org/nonograms/i/15453, 2019. április 4.

[44] Black Queen, http://www.nonograms.org/nonograms/i/15925, 2019. április 4.

[45] Bear, http://www.nonograms.org/nonograms/i/15184, 2019. április 4.

[46] Corgi, http://www.nonograms.org/nonograms/i/20868, 2019. április 4.

[47] Griffin, http://www.nonograms.org/nonograms/i/15410, 2019. április 4.

[48] Joyful dog, http://www.nonograms.org/nonograms/i/7989, 2019. április 4.

[49] Lone wolf http://www.nonograms.org/nonograms/i/4261, 2019. április 4.

[50] Kabuki (Japanese dance-drama), http://www.nonograms.org/nonograms/i/1285, 2019. április 4.

[51] Pharaoh, http://www.nonograms.org/nonograms/i/2369, 2019. április 4.

[52] Japanese woman, http://www.nonograms.org/nonograms/i/19283, 2019. április 4.

[53] The lady in the hat, http://www.nonograms.org/nonograms/i/6417, 2019. április 4.

[54] Mother's Legacy http://www.nonograms.org/nonograms/i/4334, 2019. április 4.