



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK
TANSZÉK

Aknakereső játék megoldó algoritmus

Témavezető:

Fülöp Endre

doktorandusz, MSc

Szerző:

Zsákai Bálint, ruphyy

programtervező informatikus BSc

Budapest, 2024

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	6
2.1. Telepítés	6
2.2. Rendszerkövetelmények	7
2.3. Használati segédlet	7
2.4. A program törlése	10
2.5. Hibák kezelése	10
3. Fejlesztői dokumentáció	13
3.1. Specifikáció	13
3.1.1. Funkcionális követelmények	13
3.1.2. Nem funkcionális követelmények	14
3.1.3. Felhasználói történetek	15
3.1.4. Elérhető megoldások összehasonlítása	17
3.2. Tervezés	19
3.2.1. Felhasználói felület terve	19
3.2.2. Architektúrális tervezés	20
3.2.3. Algoritmus	21
3.3. Implementáció	36
3.3.1. Osztályok kommunikációja	36
3.3.2. Eseménykezelés	37
3.4. Tesztelési terv	39
3.4.1. Unit tesztek	40
3.4.2. End-to-end tesztek	59
4. Összegzés	60
4.1. Összefoglalás	60

4.2. Továbbfejlesztési lehetőségek	61
Köszönetnyilvánítás	62
Irodalomjegyzék	62
Ábrajegyzék	64
Táblázatjegyzék	65
Algoritmusjegyzék	66
Forráskódjegyzék	67

1. fejezet






Bevezetés

Szakdolgozatomban témája egy program, amely algoritmikusan megoldja az aknakereső játékot. "Az aknakereső (Minesweeper) egyszemélyes számítógépes játék, melynek célja a táblán lévő összes akna megtalálása, illetve azok elkerülése. "Az aknakereső alapvetően logikai játék, de bármely játékmenetben előfordulhat olyan szituáció is, amelyben a helyes megoldás a szerencsén múlik. Egy egyforma cellákra osztott játéktáblával indul a játék, ezek alatt rejtőzködnek az aknák." [1]

A tábla mérete és az aknák száma a nehézségi szinttől függően változik:

- Kezdő: 10 akna véletlenszerűen szétszórva egy 9x9-es pályán.
- Haladó: 40 akna véletlenszerűen szétszórva egy 16x16-os pályán.
- Mester: 99 akna véletlenszerűen szétszórva egy 16x30-as pályán.

A következő ábra 1.1 a játéktábla celláinak lehetséges állapotait veszi sorra:

-  lefedett (alaphelyzet),
-  feltárt, szomszédos aknával,
-  feltárt aknamentes,
-  zászlós (véleményünk szerint akna van alatta),
-  kérdőjeles (lehetséges, hogy akna van alatta),
- feltárt, robbanó aknával (ha egy mező ilyen állapotba kerül, a játék véget ér, a játékos a menetet elvesztette).

1.1. ábra. A cellák lehetséges állapotai [1]

"A felsorolás szerinti negyedik és ötödik állapot az egér jobb gombjával érhető el, csupán segítséget nyújt a játékhoz. A játékot teljesíteni lehet anélkül is, hogy akár csak egy mezőt is megjelölnénk zászlóval vagy kérdőjellel, ez azonban a játékban szerzett jelentős gyakorlatot és az egész játékmenet során mindvégig komoly figyelmet igényel.

Egy mezőt feltárni kattintással lehet. Ha egy mező feltárult, és mellette akna található, akkor annak darabszámát egy számmal fogja jelezni (egy mező mellett értelemszerűen maximum 8 akna lehet). Ha a játékos aknamentes környezetű mezőre kattint, akkor az adott mezőhöz oldal- és sarokhatárosan csatlakozó (aknamentes) mezők mindegyike feltárul, valamint az így feltáruló aknamentes "szigettel" szomszédos mezők is feltárulnak.

Az első kattintáskor (a játék indításakor) minden esetben aknamentes mezőt tárunk fel, a következő lépéstől kezdve azonban a feltáruló számok ismeretében, logikai úton, illetve időnként szerencsével tudunk továbbhaladni. A program folyamatosan jelzi a még megjelöletlen akna számát, illetve az eltelt időt. A játék célja: teljesíteni a táblát a lehető legrövidebb idő alatt. Ha aknára kattintunk, az adott mező "felrobban", tehát a játék véget ér, s az adott menetet elvesztettük. Győzelemmel kizárólag abban az esetben fejeződik be a játék, ha felfedtünk minden olyan mezőt, amely alatt nincs akna. A győzelem elérése nem függ attól, hogy hány aknát jelöltünk meg zászlóval vagy kérdőjellel, illetve hogy használtuk-e egyáltalán ezeket a segítségeket." [1]

Az aknakereső játék algoritmikus megoldása számos előnnyel jár, melyek logikai, valószínűségi és számítástechnikai kihívásokat ölelnek fel.

1. Logikai kihívások: Az aknakereső logikája magában foglalja a szomszédos mezőkben található akna számának logikai következtetését. Az algoritmusoknak logikai szabályokat kell követniük annak érdekében, hogy helyesen azonosítsák az aknákat és azok hiányát a mezőkben.
2. Valószínűségi kihívások: Mivel az akna elhelyezése véletlenszerű, az algoritmusoknak fel kell készülniük arra, hogy a mezők egy részében csak valószínűségi alapú következtetéseket tudjanak levonni az akna helyéről. Ez a valószínűségi számítások alkalmazását és az információ hiányos területeken történő stratégiai cselekvéseket jelenti.
3. Számítástechnikai kihívások: Az aknakereső játék számos számítástechnikai kihívást is felvet. Ezek közé tartozik az adatstruktúrák hatékony kezelése a mezők és azok állapotainak tárolásához, a sebesség és hatékonyság optimalizálása, valamint az algoritmusok skálázhatósága nagyobb játékmezők esetén.

Az algoritmikus megoldás segítségével a játékot hatékonyan lehet kezelni és megoldani a logikai szabályok, valószínűségi alapok és hatékony számítástechnikai módszerek alkalmazásával. Ezek az algoritmusok lehetővé teszik a játék optimalizált játszását és segíthetnek a játékosnak abban, hogy logikájuk és valószínűségi érzékük segítségével a legjobb döntéseket hozzák a lehető leggyorsabban és legbiztonságosabban.

Korábban sokat játszottam a játék mobilos változatával. Felkeltette az érdeklődésem, hogy a játékban vannak ismétlődő mintázatok. A minta számok olyan elrendezését jelenti a játéktéren, amelyeknek csak egy megoldása van. Ezeknek a mintáknak a megjegyzésével csökkenthető a játékidő. Szerettem volna egy olyan programot készíteni, amely ilyen mintákat keres a játéktéren, logikai úton. A program a MinesweeperX nevű aknakereső verziót oldja meg. Azért erre a verzióra esett a választásom, mert egyszerű felhasználói felülettel rendelkezik és reklámmentes. A MinesweeperX játék ingyenesen letölthető a következő linkről: <https://minesweepergame.com/download/minesweeper-x.php>

Összehasonlítva a Microsoft Minesweeper verzió több töltési képernyőt is használ. Reszponzív felhasználói felülettel rendelkezik, amit szintén automatizálni kell. Reklámokkal zavarja meg a program működését. A játéknak több webes verziója is létezik, ezek közül kiemelném a Minesweeper Online nevű webhelyet. Ennek kinézete nagyban hasonlít a klasszikus játékéra. Ennél a verziónál a sokszínű webes környezet jelentheti a kihívást az automatizálásra nézve.

A játék nehézsége abban rejlik, hogy nem mindig rendelkezünk elegendő információval egy biztos lépés megjátszásához. Az általam készített hibrid algoritmus ötvözi Andrew Adamatzky sejtautomatájának ötletét [2] és Chris Studholme [3] megszorítás-kielégítési problémáját. Ez azért előnyös, mert az esetek többségében biztos információnk van arról, hogy egy játékcella tartalmaz-e aknát, vagy akna-mentes. Erre az esetre a sejtautomata szolgál könnyen implementálható megoldást. Amikor nem tudunk biztos állítást megfogalmazni egy celláról, megszorításokkal statisztikát készíthetünk a cellák akna-valószínűségéről. Míg a sejtautomata csak az adott cellát veszi számításba a következő lépés kiszámolásakor, addig a megszorítások a játéktér adott részeiről fogalmaznak meg megállapítást.

2. fejezet

Felhasználói dokumentáció

A programnak meg kell oldania a játékot mind a három nehézségi szinten. Ehhez képernyőképeket készít és ezeket a képeket elemzi. Az adatok elemzése után az egér mozgásával kattint a játékban. Mintha egy játékos játszana az aknakeresővel. A program nem csak a játékot oldja meg, hanem részletes leírást ad az algoritmus működéséről. Játékmenet közben a program megmutatja az intuitív lépéseket, amelyek egy kezdő játékos hasznára válhatnak. A nem intuitív -valószínűségyszámítást igénylő- döntéseket a program színes cellákkal magyarázza. Ahol nagyobb az aknák valószínűsége ott piros-, ahol kisebb az aknák valószínűsége ott zöld színűre festi a valószínűségi mátrixot. Ez a funkció a haladó játékosoknak nyújt segítséget. A program önálló gondolkodásra buzdít. Ha a játékos elakadt és segítségre van szüksége, akkor kiszámoltathatja a következő lépést és folytathatja tovább az önálló megoldást.

2.1. Telepítés

A program letölthető erről a webhelyről: <https://zsakbalint.web.elte.hu/>
A program nem igényel telepítést. Az alkalmazás futtatásához csomagoljuk ki a .zip fájlt, majd egyszerűen nyissuk meg dupla kattintással a futtatható állományt. Ez a bináris fájl a következő függőségeket csomagolja: win32 api, GDI+ api, googletest keretrendszer. ¹

¹api = Application Programming Interface. Egy program vagy operációs rendszer azon eljárásainak (szolgáltatásainak) és azok használatának dokumentációja.

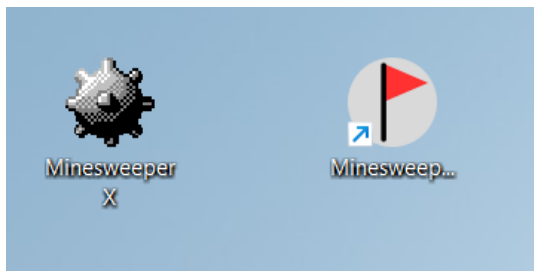
2.2. Rendszerkövetelmények

- operációs rendszer (OS) - Windows 11
- processzor (CPU) - A szoftvernek nincsenek speciális CPU-követelményei. A modern számítógépekben általában megtalálható processzorok széles skálájával kompatibilis.
- memória (RAM) - minimum 4GB
- tárhely (SSD) - 1GB
- grafikus kártya (GPU) - A szoftvernek nincsenek speciális GPU-követelményei. A modern számítógépekben általában megtalálható grafikus kártyák széles skálájával kompatibilis.
- felbontás (RES) - 1080p

A program nem indul el olyan számítógépen, amely nem Windows operációs rendszert használ. 4GB-nál alacsonyabb memóriájú rendszeren a program futása lassulhat, azon belül is az akna valószínűségi mátrix kirajzolása okozhat problémát. 1080p-nél kisebb felbontású monitorokon a program nem tudja megfelelően beolvasni a játéktér celláit és nem tud rájuk kattintani sem. A program csak a MinesweeperX játékprogram futtatását követően használható, más játékverziókkal nem kompatibilis. A program nem igényel internetkapcsolatot.

2.3. Használati segédlet

Nyissuk meg először a MinesweeperX játékot, majd nyissuk meg a megoldó programot a 2.1-es ábra szerint.

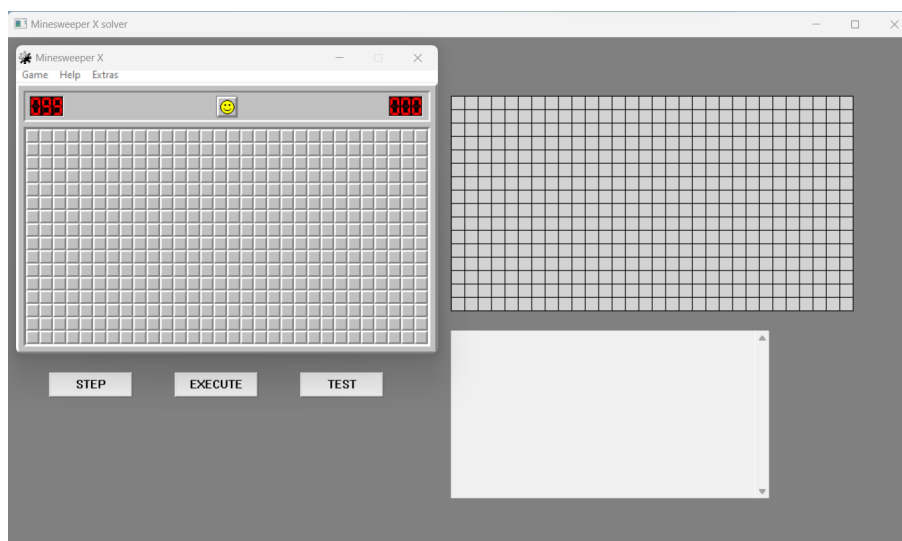


2.1. ábra. Az játék ikonja (bal) és a megoldó program ikonja (jobb)

Az alkalmazás ablaka három gombot tartalmaz:

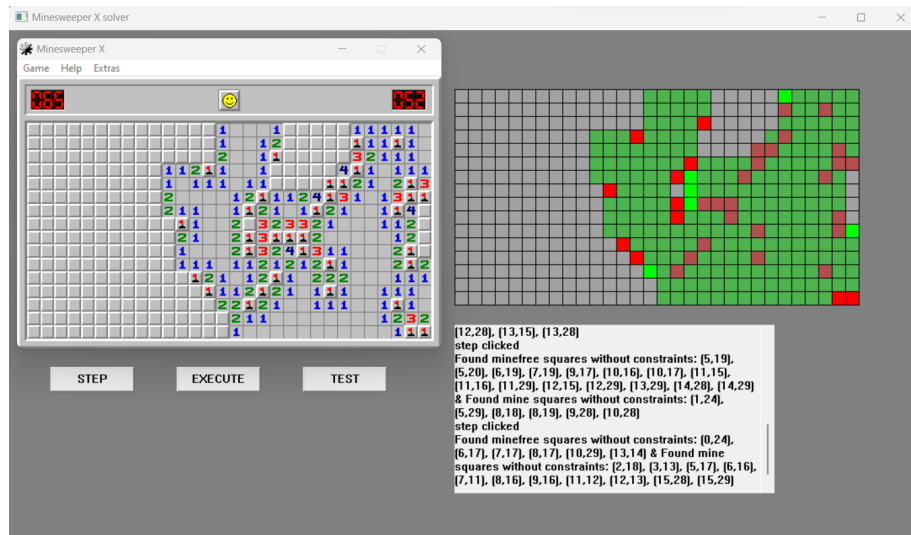
1. "STEP" - a következő lépést hajtja végre a játék megoldása felé.
2. "EXECUTE" - a jelenlegi játékot a játék végéig játssza.
3. "TEST" - megoldja a játékot 100 alkalommal a jelenlegi nehézségi szinten és az eredményeket fájlba írja. A fájl neve és kiterjesztése: "tests.csv". A fájl az alkalmazással megegyező mappába kerül elhelyezésre teszteléskor.

A program érzékeli a játék nehézségi szintjét, valamint azt, hogy vége van-e a játéknak. Miután új játékot kezdtünk vagy módosítottuk a játék nehézségi szintjét, az alkalmazásban nem kell semmit beállítanunk. Ha nem merült fel semmilyen probléma az indítás során, akkor a következő 2.2 ablakkonstrukciót láthatjuk:



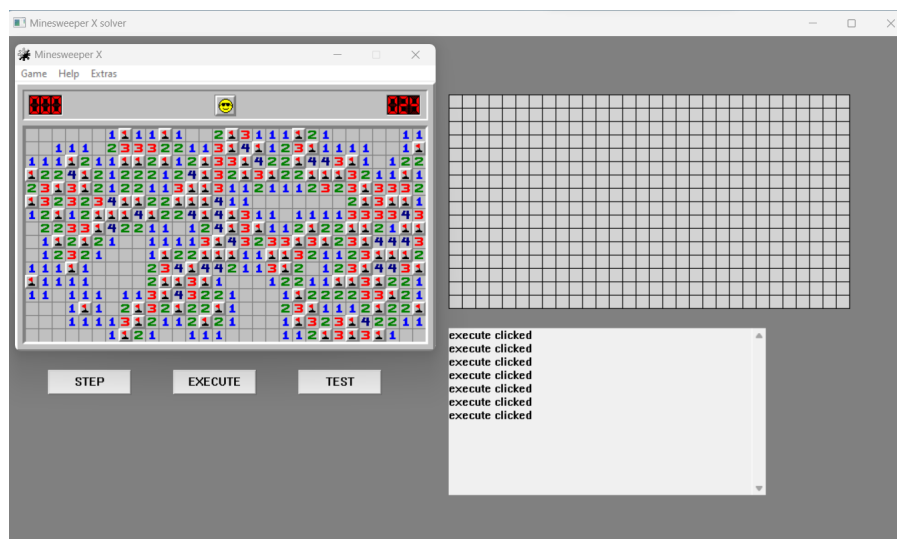
2.2. ábra. Felhasználói felület iniciális állapotban

A játéklaktól jobbra a valószínűségi mátrix látható. A program itt pirossal jeleníti meg az aknát tartalmazó cellákat és zölddel az aknamentes cellákat. Világosabb színnel jelöli azokat a cellákat, amelyeket a jelenlegi lépés során számolt ki és sötétebbel, amiket már korábban felfedezett. Amikor tippelnie kell, akkor sárga színátmenettel jelzi, hogy az adott cella mekkora eséllyel akna. Ennek a mátrixnak a mérete változik amikor változtatjuk a játék nehézségét. A valószínűségi mátrix alatt található a log. Ide írja ki a program szövegesen, hogy mi történt az adott lépést követően. A következő képen 2.3 ennek a két komponensnek a változását láthatjuk, miután a "profi" nehézségi szinten játsszva megnyomtuk többször "STEP" gombot:



2.3. ábra. Felhasználói felület a program használata közben

Az "EXECUTE" gombot megnyomva a program végigjátssza a jelenlegi játékot. A megoldás megáll, ha nyertünk, vagy veszítettünk. Ezt a parancsot kiadva 2.4 a valószínűségi mátrixon és a log felületen nem jelennek meg a megoldás lépései, mert túl gyorsan hajtja végre az utasításokat a program, nem követhető egy ember számára.



2.4. ábra. Felhasználói felület az "EXECUTE" parancs végrehajtását követően

Végül a "TEST" gombot megnyomva a program megoldja 100 alkalommal a jelenlegi nehézségi szinten és az eredményeket egy fileba írja (tests.csv). Ezt a file-t megnyithatjuk például excelben. A következő ábrán 2.5 a "kezdő" nehézségi szinten elvégzett teszt eredményei látszódnak.

	A	B	C
1	DIFFICULTY	NUMBER OF GUESSES	IS WON
2	BEGINNER	1	TRUE
3	BEGINNER	4	FALSE
4	BEGINNER	1	TRUE
5	BEGINNER	1	TRUE
6	BEGINNER	2	TRUE
7	BEGINNER	2	TRUE
8	BEGINNER	4	FALSE
9	BEGINNER	2	TRUE
10	BEGINNER	1	TRUE
11	BEGINNER	1	TRUE
12	BEGINNER	2	TRUE
13	BEGINNER	2	FALSE
14	BEGINNER	3	TRUE
15	BEGINNER	2	TRUE
16	BEGINNER	2	FALSE
17	BEGINNER	2	TRUE
18	BEGINNER	4	TRUE
19	BEGINNER	1	TRUE
20	BEGINNER	3	TRUE
21	BEGINNER	1	TRUE
22	BEGINNER	1	TRUE
23	BEGINNER	2	FALSE
24	BEGINNER	3	FALSE
25	BEGINNER	3	TRUE
26	BEGINNER	9	TRUE
27	BEGINNER	1	TRUE

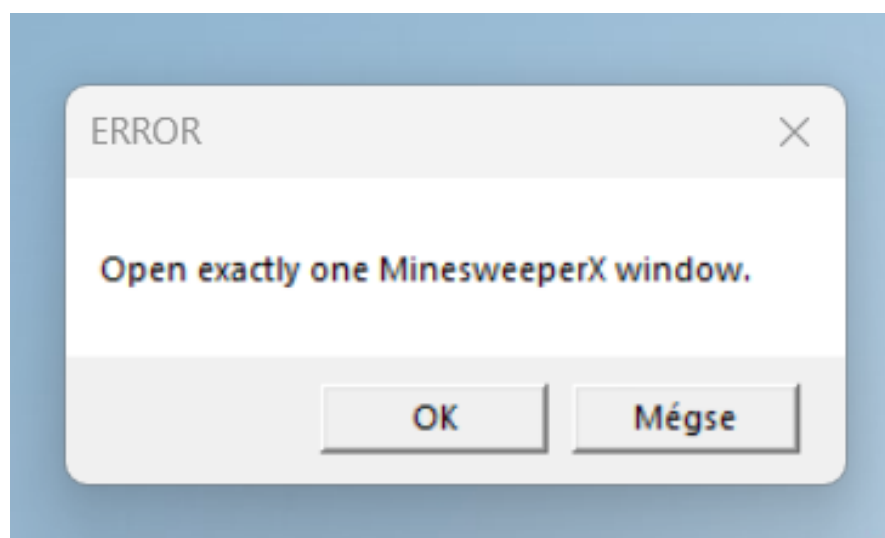
2.5. ábra. A tesztek eredményei táblázatos formában

2.4. A program törlése

Ha szeretnénk törölni a programot, egyszerűen töröljük az összes kicsomagolt állományt. Más teendőnk nincs.

2.5. Hibák kezelése

Ha megnyitjuk a programot, de még nem nyitottuk meg a MinesweeperX játékot 2.6, vagy többször is meg van nyitva, akkor a következő hibaüzenettel találkozunk:

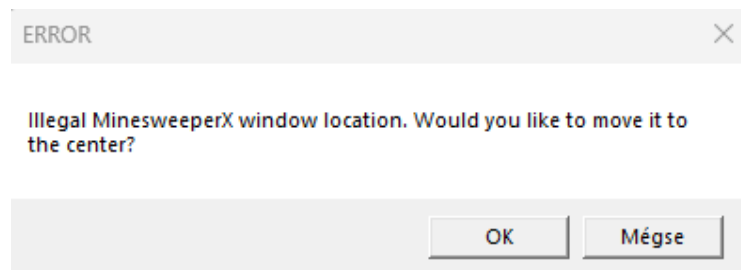


2.6. ábra. Hibaüzenet a program megnyitásánál

A "Mégse" gombot megnyomva vagy bezárva az ablakot a program bezárul. Ha megnyitjuk pontosan egyszer a játékot és rákattintunk az "OK" gombra, akkor elindul az alkalmazás. Ha az "OK" gomb megnyomása után még mindig fennáll a probléma, akkor a hibaüzenet megmarad.

Előfordulhat az az eset, hogy az alkalmazást sikeresen használtuk, majd bezárjuk a programot. A programot újra megnyitva hiába van egy példányban megnyitva a játék, a programunk több megnyitott játékként érzékeli. Ebben az esetben zárjuk be a játékot, majd nyissuk meg újra. Ekkor már működni fog az alkalmazás.

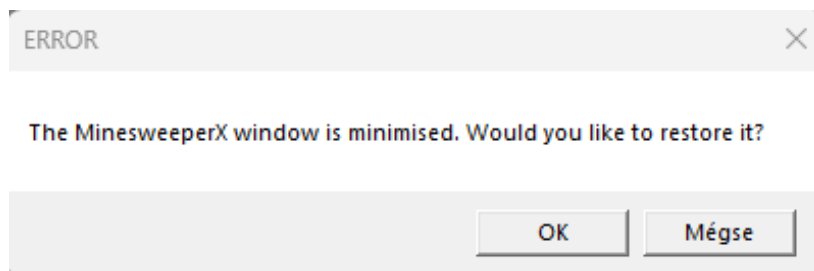
A program képernyőképek alapján gyűjt információt a játékállásról. Ezt csak úgy tudja elérni, ha mindig tudjuk, hogy hol van a játék ablaka (a program ablakához viszonyítva). Emiatt, ha mozgatjuk a program ablakát, akkor a játék ablakát is áthelyezi a megfelelő helyre. Fontos információ: a megfelelő képernyőkép készítése érdekében windows alatt a "Rendszer > Kijelző > Méretezés" beállításnak 100%-on kell lennie. Ennek hiányában a program nem működik megfelelően! A program megakadályozza, hogy a játék ablakát külön áthelyezzük (amíg a program fut). A megfelelő képernyőképek készítéséhez az is szükséges, hogy a játék ablaka ne lógjon ki a képernyő oldalán. Ha a program ablakát úgy mozgatjuk, hogy a játék ablaka nem megfelelő helyen van 2.7, akkor egy hibaüzenetet kapunk:



2.7. ábra. Hibaüzenet a program futása közben I.

Itt az "OK" gombra rákattintva a program ablaka (a játék ablakával együtt jól fotózható helyre kerül. A "Mégse" gombot megnyomva az alkalmazás bezárul.

Ha valamely funkciót úgy akarjuk használni, hogy a játék ablaka kicsinyített nézetben van 2.8, ezt a hibaüzenetet kapjuk:



2.8. ábra. Hibaüzenet a program futása közben II.

Itt az "OK" gombra rákattintva a játék ablakát a program visszaállítja felnagyított állapotba. A "Mégse" gombot megnyomva az alkalmazás bezárul.

3. fejezet

Fejlesztői dokumentáció

3.1. Specifikáció

3.1.1. Funkcionális követelmények

- A funkciók felhasználói felületről érhetők el. A három fő funkció: a megoldás következő lépésének kiszámolása és az adatok ember által értelmezhető megjelenítése; A játék végigjátszása a lehető legpontosabban, a legrövidebb idő alatt; Az eredmények számszerűsítése és statisztikák készítése.
- A funkcióknak működniük kell a játék három fő nehézségi szintjén: Kezdő, Haladó, Profi. Nem elvárt, hogy egy felhasználó által készített egyedi játéktáblán is működjenek (ahol a felhasználó ízlés szerint megadhatja a játéktábla méretét és az aknák számát). A nehézségi szint meghatározása a funkciók meghívása előtt automatikusan történik, azt nem a felhasználónak kell expliciten megadnia.
- Az első funkció működése: A felhasználói felületnek rendelkeznie kell egy színes táblával, amely megmutatja a felhasználónak, hogy melyik cella milyen valószínűséggel tartalmaz aknát. Ezen kívül szükség van egy szövegdobozra a felhasználói felületen, amely szöveges formában is tájékoztatja a felhasználót a jelenlegi lépésekről. Ezeknek az elemeknek elegendő az első funkció meghívásakor frissülniük. Ez azért van, mert a második funkció túl gyorsan hajtja végre a feladatot, hogy egy ember követni tudja. A harmadik funkció pedig pusztán tesztelésre szolgál.
- A második funkció működése: A program nem írhatja át a játék által allokált memóriát. Az adatok olvasását képernyőképek készítésével és azok elemzésével

kell véghezvinnie. A játék navigálásához az egeret kell használnia. A megoldásnak minél nagyobb pontosságúnak kell lennie.

- A harmadik funkció működése: a megoldás eredményeit .csv fájlba kell írni és a következő adatokat kell tartalmaznia egy rekordnak: játék nehézségi szintje; tippelések száma; sikerült-e megnyerni az adott játékot.
- Az indítás előtt meg kell győződni, hogy a játék csak egy példányban van megnyitva, mert a megoldó program az indítást követően csak ezzel az egy játékpéldánnyal dolgozik.
- Az indítást követően minden funkció meghívása előtt ellenőrizni kell, hogy a játék ablaka megfelelő helyen van-e: a képernyőképek készítése és a játéktéren való kattintás nem ütközhet problémába.
- A program bezárása nem eredményezi a játék bezárását, de a játék bezárása eredményezi a program bezárását. Ezt elegendő akkor leellenőrizni, amikor a játék bezárása után a felhasználó meghív egy funkciót.

3.1.2. Nem funkcionális követelmények

- Megbízhatóság: A tesztelési adatok csak lokálisan kerülnek mentésre. A funkciók mindig elérhetőek és végrehajtnak, akkor is, ha a tesztelés által generált adatok sérülnek. A megbízhatóan kezelt adatok mennyisége korlátozott.
- Biztonság: Nincs garancia az adatok biztonságára.
- Hatékonyság: A program válaszideje gyors minden funkcióra.
- Hordozhatóság: A program csak Windows10, vagy újabb operációs rendszeren fut, személyi számítógépen. A program másolással könnyen áttelepíthető.
- Felhasználhatóság: Minden programfunkciónak azonnal áttekinthetőnek kell lennie, kézikönyv nélkül.
- Környezeti: A felhasználás helye egy Windows10-et futtató számítógép. Egyetlen más programmal kell együttműködnie: a MinesweeperX játékkal.
- Működési: A felhasználók a programot néhány óránál nem hosszabb ideig használják, közben a funkcionalitást intenzíven igénybe veszik.

3.1.3. Felhasználói történetek

Felhasználói történetek		
<i>AS A</i>	<i>I WANT TO</i>	<i>SO THAT</i>
<i>Felhasználó</i>	Elindítani az alkalmazást.	-
<i>GIVEN</i>	WHEN	THEN
<i>Korábban elindítottam az aknakereső játékot (egy példányban)</i>	Amikor elindítom az alkalmazást.	A program ablaka hibaüzenet nélkül megjelenik.
<i>Korábban nem indítottam el az aknakereső játékot</i>	Amikor elindítom az alkalmazást.	A program megjelenít egy hibaüzenetet messagebox formában.
<i>Korábban több példányban is elindítottam az aknakereső játékot</i>	Amikor elindítom az alkalmazást.	A program megjelenít egy hibaüzenetet messagebox formában.
<i>AS A</i>	<i>I WANT TO</i>	<i>SO THAT</i>
<i>Felhasználó</i>	Lépésenként akarom megoldani a játékot.	-
<i>GIVEN</i>	WHEN	THEN
<i>Rákattintottam a "STEP" gombra</i>	A játék ablaka megfelelő pozícióban van.	A program végrehajtja a következő lépést és részletes tájékoztatást ad a döntési folyamatról.
<i>Rákattintottam a "STEP" gombra</i>	A játék ablaka kilóg a viewport-ról.	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak áthelyezésére.
<i>Rákattintottam a "STEP" gombra</i>	A játék ablaka kicsinyített nézetben van.	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak visszaállítására.

<i>GIVEN</i>	<i>WHEN</i>	<i>THEN</i>
<i>AS A</i>	<i>I WANT TO</i>	<i>SO THAT</i>
<i>Felhasználó</i>	Végig akarom számoltatni a játék megoldását.	-
<i>GIVEN</i>	<i>WHEN</i>	<i>THEN</i>
<i>Rákattintottam az "EXECUTE" gombra</i>	A játék ablaka megfelelő pozícióban van.	A program végigjátssza a játékot, közben nem jelenít meg adatokat.
<i>Rákattintottam az "EXECUTE" gombra</i>	A játék ablaka kilóg a viewport-ról	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak áthelyezésére.
<i>Rákattintottam az "EXECUTE" gombra</i>	A játék ablaka kicsinyített nézetben van.	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak visszaállítására.
<i>AS A</i>	<i>I WANT TO</i>	<i>SO THAT</i>
<i>Felhasználó</i>	Tesztelni akarom a program hatékonyságát.	-
<i>GIVEN</i>	<i>WHEN</i>	<i>THEN</i>
<i>Rákattintottam a "TEST" gombra</i>	A játék ablaka megfelelő pozícióban van.	A program végigjátssza a játékot a jelenlegi nehézségi szinten 100 alkalommal, közben nem jelenít meg adatokat. Az eredményeket csv fájlba írja.
<i>Rákattintottam a "TEST" gombra</i>	A játék ablaka kilóg a viewport-ról	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak áthelyezésére.

<i>GIVEN</i>	<i>WHEN</i>	<i>THEN</i>
<i>Rákattintottam a "TEST" gombra</i>	A játék ablaka kicsinyített nézetben van.	A program megjelenít egy hibaüzenetet messagebox formában, amely kéri a Felhasználót az ablak visszaállítására.
<i>AS A</i>	I WANT TO	SO THAT
<i>Felhasználó</i>	Bezárom a programot.	-
<i>GIVEN</i>	WHEN	THEN
<i>Megnyomtam a jobb felső sarokban az "X" gombot</i>	Miután a program bezárult	A játék tovább fut.
<i>AS A</i>	I WANT TO	SO THAT
<i>Felhasználó</i>	Bezárom a játékot.	-
<i>GIVEN</i>	WHEN	THEN
<i>Bezárult a játék ablaka</i>	Rákattintok a program egy gombjára	A program bezárul.

3.1. táblázat. felhasználói történetek

3.1.4. Elérhető megoldások összehasonlítása

A követelmények ismertetése után vizsgáljuk meg a piacon elérhető megoldásokat, amelyek alternatívát tudnak nyújtani erre a problémára. Számos hasonló program elérhető az interneten. A következőkben ezek közül néhányat fogok ismertetni, valamint ezeknek az előnyeit és hátrányait fogom tárgyalni a fejlesztendő szoftverhez képest.

- <https://github.com/andrewbae/windows-xp-minesweeper-hack> Ez a program memória hackelés felhasználásával készült. Ennek a programnak két fő funkciója van: letiltja a programot arról az eseményről, amikor aknát tartalmazó cellára kattint a felhasználó. A második funkciója az, hogy visszaállítja a játék időzítőjét 0-ra. Ez a program nem oldja meg automatikusan a játékot, hanem a felhasználó oldja meg könnyedén, ezeknek a csalásoknak a használatával. Ez a megoldás előnyösebb, ha a célunk a játék hibátlan megoldása. Hátránya, hogy a játékos nem tanul a hibáiból, az alkalmazás

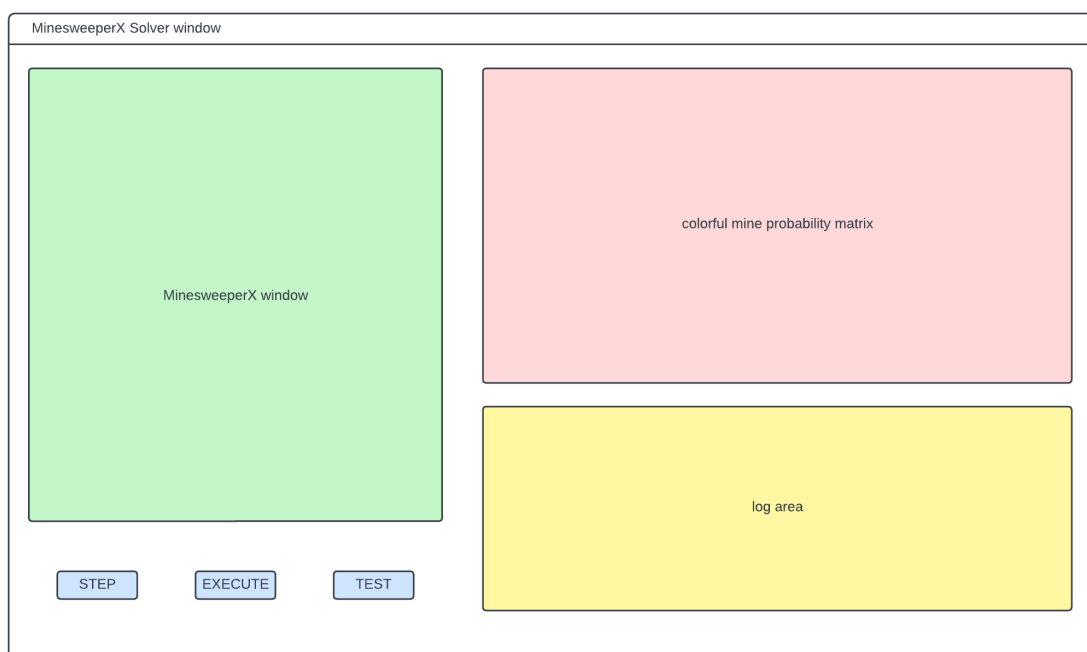
nem nyújt támpontokat a helyes kitöltésre nézve. Az alkalmazás C++ és C nyelven íródott.

- <https://github.com/gamescomputersplay/minesweeper-solver> Ez a program pythonban készült és nagyon hasonlít az én megoldásomhoz. Körülbelül azonos gyorsasággal és pontossággal oldja meg a játékot. Ehhez a programhoz nem tartozik felhasználó-barát felhasználói interface. A játéktér méretét és az aknák számát kézzel kell megadnunk. Ez a metódus megengedi, hogy egyedi játéktéren játszassunk, amelyet az én programom jelenleg nem tesz elérhetővé.
- <https://www.logigames.com/minesweeper/solver> Ezen a weboldalon egy eltérő megoldással találkozhatunk. Itt nekünk kell kézzel beállítani minden információt, amit eddig a játéktéren felfedtünk. A weboldal ezután kiszámolja az összes lehetséges megoldást. Ennek előnye, hogy nem függ a játék felhasználói felületétől. A felhasználó bármilyen aknakereső verziót játszik, használni tudja ezt a segítséget. Hátránya hogy sokszor kézzel bevinni az adatokat körülményes.

3.2. Tervezés

3.2.1. Felhasználói felület terve

Az alkalmazás rendelkezik felhasználói felülettel 3.1, amelynek tervezésére a következő drótváztervet vázoltam fel. A specifikációban tárgyalt három fő funkciónak nevet adtam: az első funkció neve - "STEP", a második funkció neve - "EXECUTE", a harmadik funkció neve - "TEST". Az egyes funkciókat meghívó gombokat felcímkeztem ezekkel a nevekkkel.



3.1. ábra. Mockup

Magyarázat:

- világoszöld: az akna kereső játék ablaka
- pink: színes mátrix, amely azt mutatja meg, hogy az egyes cellák mekkora eséllyel akna, valamint a már feltárt cellákat is megjeleníti. Erre színátmenetet használ: piros szín: biztosan akna -> 0 valószínűség, zöld: biztosan akna-mentes -> 1 valószínűség, sárga: lehetséges, hogy akna -> [0..1] valószínűség.
- világoskék: három gombja van az alkalmazásnak: STEP - az algoritmus által következő lépést teszi meg, EXECUTE - a játék elvesztéséig vagy megnyeréséig ismétli a STEP parancsot, TEST - a jelenlegi nehézségi szinten 100 alkalommal meghívja az EXECUTE parancsot, és közben az eredményeket CSV

fileba írja. egy új bejegyzésnek tartalmaznia kell a következőket: (nehézségi szint, tippelések száma, nyertünk-e)

- világossárga: a program log felülete, amely szöveges információkkal látja el a felhasználót a program lefutása közben

3.2.2. Architektúrális tervezés

Azért választottam a Windows10 operációs rendszert, mert a legtöbb háztartás ezt az operációs rendszert használja, valamint a választott aknakereső verzió is Windows-on fut. A szoftver elkészítéséhez a C++ programozási nyelvet választottam. Egyrészt azért, mert objektum orientáltan szerettem volna megoldani a problémát, másrészt mert ez az egyik kedvenc programozási nyelvem a Java-n kívül. Fejlesztői környezetnek a Visual Studio Community 2022-t választottam (17.6.4-es verzió), hogy a fejlesztés gördülékenyen menjen és ne kelljen bajlódnom a fordítással. Fejlesztés során a g++ 8.3.0 verzióját vettem igénybe. A program elkészítéséhez a következő függőségeket használtam:

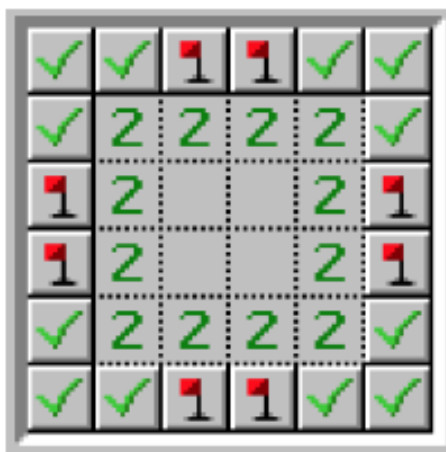
- Az alkalmazásnak kommunikálnia kell a játék ablakával. Ezt a követelményt és az alkalmazás applikációs ablakát meg lehet valósítani a win32 api segítségével. Emelett a döntés mellett több érv is szól. Ez az api elérhető natívan minden Windows operációs rendszeren. Alacsony szintű interfész, ezért a hatékonyság növeléséhez jól használható. Használatával Windows-specifikus technológiák válnak elérhetővé, ezt a következő pontban fogom részletesen kifejteni. Néhány alternatíva Windows rendszeren asztali alkalmazások fejlesztésére: .NET keretrendszer, Qt könyvtár, wxWidgets könyvtár. Azért a win32 api-ra esett a választásom, mert szerettem volna megtanulni hogyan kell Window alatt natívan ablakot készíteni.
- Az ablakra a valószínűségi mátrixot valahogy ki kell rajzolni. Ehhez a GDI+ api-t választottam, mert könnyen elkészíthetők vele 2D-s grafikák, C/C++ programozási nyelvre lett tervezve és natív Windows integrációval rendelkezik. Néhány alternatíva 2D-s grafikák készítésére: GDI, Direct2D, OpenGL.
- A program tesztelésére a googletest framework-öt választottam, mert open source és ingyenesen használható, könnyen integrálható Visual Studio projektbe és jól dokumentált. Néhány alternatíva C++ programok tesztelésére: Catch, UnitTest++, CppUnit.

A program architektúrája háromrétegű: model/nézet/perzisztencia. A nézet réteg felelős a program ablakának megjelenítéséért és az események kezeléséért. A model felelős az adatok -beolvasásáért és feldolgozásáért. A perzisztencia réteg felelős az adatok tárolásáért és mentéséért.

3.2.3. Algoritmus

Ebben az alfejezetben először ismertetem a probléma formális leírását, majd néhány meglévő algoritmust fogok bemutatni és ismertetem, hogy ezek az algoritmusok hogyan kapcsolódnak az én megoldásomhoz.

Definíció - Az adott aknakereső játéktábla konzisztens: "Az Aknakereső játéktábla akkor konzisztens, ha van legalább egy olyan konfigurációja a lehetséges aknáknak és biztonságos mezőknek, amely nem eredményez ellentmondást." [4] A következő ábrákon bemutatok egy konzisztens 3.2a és egy inkonzisztens 3.2b játéktáblát. Az első példa konzisztens, mert meg tudunk adni egy lehetséges megoldást az aknák elhelyezkedésére. A második példába inkonzisztens, mert a 6-os számú cellának csak 3 szomszédja van, valamint a 2-es számú cellának nem lehet 5db akna a szomszédja.



(a) Példa konzisztens játéktáblára [4]



(b) Példa inkonzisztens játéktáblára [4]

3.2. ábra. Példák konzisztens és inkonzisztens játéktáblákra

Az aknakereső játék konzisztenciája NP-teljes. Ezt az állítást Richard Carini [4] látta be úgy, hogy az Aknakereső konzisztenciáját visszavezette az áramkör kielégítési problémára:

1. Az aknakereső konzisztenciája NP komplexitású: "Tegyük fel, hogy kapunk egy lehetséges megoldást egy tetszőleges Aknakereső rácsra (más szavakkal, egy

teljesen felfedett táblát, amelyen nincsenek rejtett cellák). Hogy ellenőrizzük, hogy a megoldás konzisztens-e, az algoritmusnak végig kell iterálnia egyesével a cellákon. Ellenőriznünk kell, hogy minden cella értéke pontosan megegyezik az aknát tartalmazó szomszédos cellák számával. Minden cellának maximum 8 szomszédos celláját kell ellenőriznünk, így ha az adott Aknakereső rács n négyzetből áll, a futási idő legfeljebb $8t * n$ (ahol t az ellenőrzéshez szükséges idő egy cella szomszédjára). Mivel ez az algoritmus egyértelműen polinomiális idejű, az Aknakereső konzisztencia probléma NP-ben van." [4]

2. "Az aknakereső konzisztenciája NP nehéz: Az a tény, hogy az Aknakereső NP-ben van, nem elég; azt is bizonyítanunk kell a probléma NP nehéz. Tegyük fel, hogy kapunk egy tetszőleges logikai áramkört, amelynek bemenetei változóit jelöljük x_1, x_2, \dots, x_n sorozattal és a kimenetét jelöljük y -al. Ez az áramkör ÉS, VAGY és NOT logikai kapukból épül fel, amelyek végül igazat/hamisat rendelnek y -hoz, a bemeneti változók értékétől függően. Szeretnénk egy olyan hozzárendelést találni a bemeneti változókhoz, amely a IGAZ értéket ad kimenetként y -ra (definíció szerint ez az „áramkör kielégítési probléma"). Mivel képesek vagyunk vezetékeket, ÉS-kaput, VAGY-kaput és NOT-kaput létrehozni az Aknakereső celláiból, így az áramkörhöz megfeleltethetünk egy aknakereső rácsot. Az áramkör végén aknát rendelünk y értékéül (IGAZ értéket adunk neki). Feltéve, hogy van egy algoritmusunk, amely meghatározza az Aknakereső konzisztenciáját polinom időben, meghatározhatunk egy logikai hozzárendelést a kezdeti változóinkhoz (akna vagy aknamentes), amely konzisztens marad az általunk megadott y értékkel. Ily módon az áramkör kielégíthetősége, meghatározható az Aknakereső konzisztencia segítségével, feltéve, hogy rendelkezünk egy polinomiális időben lefutó algoritmussal ennek ellenőrzésére. Mivel tudjuk, hogy az áramkör kielégíthetősége NP nehéz (bármilyen probléma megoldására használható NP-ben), ezért az aknakereső konzisztenciának is NP nehéznek kell lennie." [4]
3. "Mivel az Aknakereső konzisztenciáról beláttuk, hogy NP-ben van, és NP nehéz, ezért a probléma NP teljes." [4]

Most hogy ismerjük a probléma komplexitását, ismertetek pár meglévő megoldást.

Az egyik naív algoritmus, amelyet Andrew Adamatzky [2] dolgozott ki, felhasználja a sejtautomata működését az Aknakereső játék megoldására. Ezt a megközelítést úgy lehetne röviden és közérthetően leírni mintha egy petri csészében magára hagynánk sejteket. A sejtek az idő múlásával megváltoztatják a saját állapotukat és kihatással vannak a körülöttük lévő sejtekre.

Ennek matematikai leírása a következő: "Meghatározzunk egy Q cellaállapotok halmazát és egy f átmenet függvényt, amely egy cella állapotát a t időpontban lévő állapotból a $t + 1$ időpontban lévő állapotra képzi." [2][5] Andrew Adamatzky modelljében minden állapot egy tuple-ként ¹ van ábrázolva. Az állapotok halmazát a következő Descartes-szoratként definiáljuk:

$$Q = \{\#, \bullet, \circ\} \times \{0, 1, \dots, 8\}$$

ahol $x \in Q$,

$x^t = \#$ azt jelenti, hogy x cella t időpillanatban aknát tartalmaz.

$x^t = \bullet$ azt jelenti, hogy x cella t időpillanatban feltáratlan.

$x^t = \circ$ azt jelenti, hogy x cella t időpillanatban feltárt.

"Az egyszerűség kedvéért Adamatzky [2] jelölését fogom használni és definiálom a következő jelöléseket: $x^t \in \{\#, \bullet, \circ\}$ és $v(x) = \text{Label}(x) \in \{0, 1, \dots, 8\}$. Így a cella állapota leírható egy $(x^t, v(x))$ rendezett kettessel." [5] "Hogy az aknakereső játék szabályait ne szegjük meg, $v(x)$ -et csak akkor ismerhetjük t időpillanatban, ha $x^t = \circ$. f magába foglalja az AFN ² és AMN ³ eseteket is." [2] Az Adamatzky által definiált átmenetfüggvény a következő:

¹tuple - értékek rendezett sorozata, ebben az esetben rendezett kettessel beszélünk.

²AFM - all free neighbors - minden szomszédos cella aknamentes

³AMN - all mine neighbors - minden szomszédos cella aknát tartalmaz.

$$x^{t+1} = f(x^t) = \begin{cases} \circ, \text{ha } (x^t = \bullet) \wedge (\exists y \in \text{Neighbors}(x) : y^t = \circ \\ \wedge (v(y) = \text{NumberOfMarkedNeighbors}(y^t))) \\ \\ \# , \text{ha } (x^t = \bullet) \wedge (\exists y \in \text{Neighbors}(x) : y^t = \circ \\ \wedge \text{NumberOfAllNeighbors}(y^t) = 1 \\ \wedge v(y) - \text{NumberOfMarkedNeighbors}(y^t) = 1) \\ \\ \bullet , \text{máskülönben} \end{cases} \quad (3.1)$$

Az átmenetfüggvény magyarázata:

- Ha egy fedetlen cellának már megjelöltük zászlóval megfelelő számú szomszédját, akkor a többi szomszédos cellája biztosan aknamentes (AFN eset).
- Ha egy fedetlen cellának a címkéje 1 és csak egy feltáratlan szomszédos cellája van, akkor az biztosan bomba (AMN eset).

"Ez a megközelítés teljesen determinisztikus. Mivel a sejtautomata a játék iniciális állapotában nem tud kinyitni egy cellát sem, ezért már elkezdett játékot tudunk csak további megoldásra odaadni neki. Az automata ezen kívül tippelni sem tud, pedig ez egy nagyon fontos része a játéknak." [5]

Egy másik stratégia, amit Kasper Pederson [6] írt le az úgynevezett "Egy pontos stratégia", amely már nem determinisztikusan működik. "Ez a naív algoritmus egy S halmazt használ, amiben olyan cellák vannak, amelyek determinisztikusan eldönthető, hogy aknát tartalmaznak, vagy nem. Ha S halmaz üres, akkor az algoritmus találmásra választ egy fedett cellát és belerakja a halmazba. Így az első lépést találmásra választja ki. S halmaz elemeit hasonlóan vizsgálja mint az előző algoritmus. Ezután a cella szomszédait megjelöli zászlóval vagy felfedi. Ha a cella effektív címkéje ⁴ nem megfelelő, akkor az algoritmus nem csinál vele semmit. Csak akkor fog vele ismét foglalkozni, ha a szomszédai megváltoznak." [5] Ezt az algoritmust használva már képesek vagyunk végigjátszani a játékot, de a véletlen cellaválasztások miatt nyerési ráta alacsony lesz.

⁴effektív címke - a cella értékéből kivonjuk a már ismert szomszédos aknák számát. Ez a szám azt adja meg, hogy hány darab aknát kell még a az adott cella körül megtalálnunk.

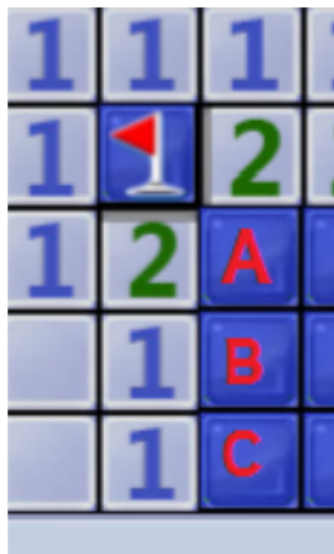
1. algoritmus Algorithm 1 Single Point Strategy

```

1:  $S \leftarrow \{\}$ 
2: while game not lost OR game not won do
3:   if  $S = \{\}$  then
4:      $x \leftarrow \text{Random}(\text{CoveredCells}())$ 
5:   else
6:      $x \leftarrow \text{First}(S)$ 
7:   end if
8:   if  $\text{MinesLeft}(x) = \text{Count}(\text{CoveredNeighbours}(x))$  then
9:      $toFlag \leftarrow \text{CoveredNeighbours}(x)$ 
10:     $\text{FlagCells}(toFlag)$ 
11:     $S \leftarrow S \cup \text{UncoveredNeighbours}(toFlag)$ 
12:   else if  $\text{MinesLeft}(x) = 0$  then
13:      $safeCells \leftarrow \text{CoveredNeighbours}(x)$ 
14:      $newCells \leftarrow \text{Uncover}(safeCells)$   $\triangleright$  returns cells uncovered in this move
15:      $newBorders \leftarrow \text{BorderingCells}() \cap \text{CoveredNeighbours}(newCells)$ 
16:      $S \leftarrow S \cup \text{UncoveredNeighbours}(newBorders)$ 
17:   end if
18:    $S.\text{remove}(x)$ 
19: end while

```

A harmadik megoldás amit bemutatok a Chris Studholme [3] által kifejlesztett "Megszorításkielégítési-algoritmus" (angolul Constraint satisfaction problem). Ennek lényege, hogy az algoritmus nem közvetlenül a cellák alapján dönt, hanem a cellák kapcsolatai alapján. A megszorítások 3.3 fogalmát a következőképp lehet szemléltetni:



3.3. ábra. Példa megszorításokra. [5]

Ebben a konfigurációban három megszorítás látható. Fentről lefelé leolvassva: $A + B = 1$; $A + B + C = 1$; $B + C = 1$; . Ha a második megszorítást redukáljuk az elsővel ezt kapjuk: $C = 0$; . Ha a második megszorítást a harmadikkal redukáljuk, ezt kapjuk: $A = 0$; . A redukálás után tehát: $A = 0$; $C = 0$, amiből $\implies B = 1$; eredményt kapjuk. A program azért megszorításokat használ a számolásokhoz, mert a megszorításhalmazok nem egy-egy celláról, hanem a játéktér adott részeiről állítanak valamit. Így értelmezve a játéktér több információt kapunk, mintha cellánként értelmeznénk a látottakat. A megszorításokat redukálással tudjuk egyszerűsíteni, amit az előző példában szemléltettem. A redukálás azért fontos, mert így új összefüggéseket kapunk a meglévő megszorításokból.

A Chris Studholme [3] által megfogalmazott CSP⁵ algoritmus dinamikusan karbantart egy megszorításhalmazt, melynek neve C . "Minden lépés után az algoritmus megszorításokat készít a játéktéren lévő cellákat felhasználva. Minden nem triviális megszorítás bekerül a C -halmazba, miközben a degeneratív megszorításokat a rendszer eldobja a megfelelő AFN vagy AMN műveletek végrehajtása után. Továbbá az algoritmus felismeri a megszorítás részhalmazokat, és végrehajtja a redukciót olyan módon, ahogy az előző példában szemléltettem. Ha a C halmaz csak redukált, nem triviális megszorításokból áll, az algoritmus elkészíti visszalépéses keresést használva az összes lehetséges megoldást a megszorítás halmazra. A visszalépéses keresés kiválaszt változókat, hogy azokhoz értéket rendeljen. Az értékek kiválasztási sorrendje nem fontos, mivel a visszalépéses keresés minden lehetséges variációt kiszámol.

A probléma méretének csökkentése érdekében felosztjuk C -t részhalmazokra. Két megszorítás akkor kerül egy részhalmazba, ha legalább egy közös változójuk van. Ennek eredményeként a visszalépéses keresés megoldhat minden részhalmazt külön-külön, mint független részproblémákat. Miután az algoritmus megtalálta az összes lehetséges megoldást, megkeresi azokat a cellákat, amelyek aknát tartalmaznak, vagy aknamentesek minden megoldásban." [5] Az algoritmus ezeket a cellákat vagy zászlóval jelöli (ha minden megoldásban akna), vagy 'rákattint' (ha minden megoldásban aknamentes). Egyes megoldások olyan megszorításokat tarthatnak fel, amelyek találgatást igényelnek. Ezek olyan esetek, amikor az algoritmus kiszámolta az összes lehetséges megoldást a megszorítás halmazra, és egyes cellákra homogén eredményt kapott (pl.: 50-50% vagy 1/3-1/3-1/3 esély hogy akna). Jellemzően az ilyen esetek két ismeretlen mező között állnak fenn. Amikor az algoritmus találga-

⁵CSP - Constraint Satisfaction Problem, a "Megszorításkielégítési-algoritmus".

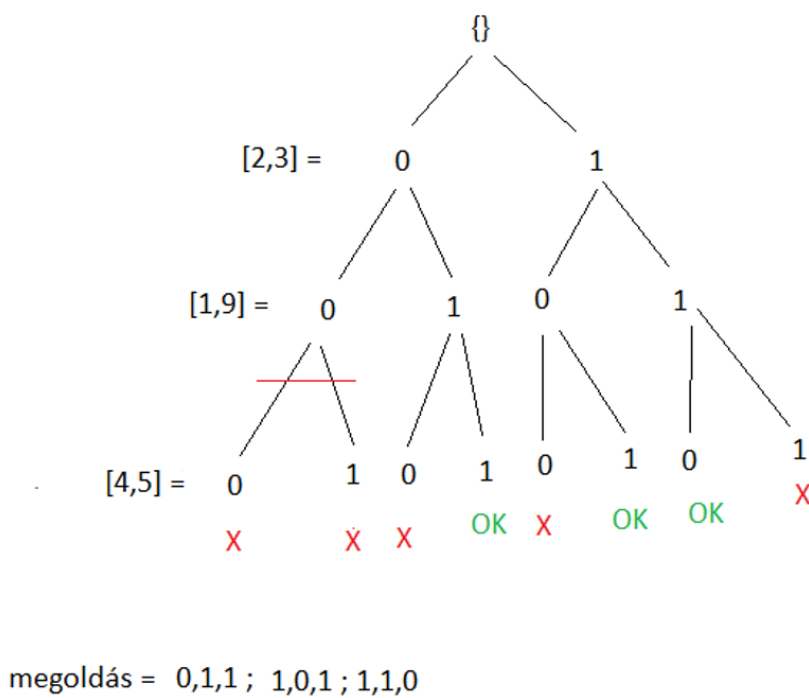
tást igénylő cellákat talál, azonnal kezeli őket, hogy a lehető leggyorsabban új játékot tudjon kezdeni.

"A CSCSP⁶ algoritmus a lehető legtöbb információt használja fel, ha tippelnie kell. Ha az összes lehetséges megoldás kiszámítása után nem fedezett fel biztos lépést, az algoritmus minden cellára kiszámít egy valószínűségi becslést (hogy milyen valószínűséggel akna/aknamentes a cella). Ezt egy egyszerű osztással teszi. Az akna esetek számát leosztjuk az összes esettel akkor megkapjuk, hogy milyen valószínűséggel akna az adott cella. Ebben az esetben a legkisebb valószínűségű cella számít a legjobb tippnek. Ezután az algoritmus összehasonlítja a legjobb tippet az ismeretlen cellák valószínűségeivel (az ismeretlen cellák nem tartoznak egyetlen megszorítás-hoz sem). Ezeknek a valószínűségeknek a kiszámolásához ismernünk kell a kezdeti aknák számát és a fennmaradó aknák számát. Ha ez a valószínűség nagyobb, mint a legjobb tipp valószínűsége, akkor az algoritmus a legjobb tipphez tartozó cellát választja. Ellenkező esetben az algoritmus véletlenszerűen kiválaszt egy ismeretlen sarokcellát, élcellát vagy belső cellát, ebben a sorrendben." [5]

A következőkben Andrew Fowler és Andrew Young által készített dolgozatához [7] hasonlóan ismertetem a megszorítások kiértékelése a lineáris algebra kapcsolatát. A játéktábla celláit a következőképp alakítjuk megszorításokká: A cella címkéje az egyenlet bal oldalán lévő összeg, a cella szomszédai -amelyeket a szomszédok helyének (sor,oszlop) kettesével jelöljük- pedig az egyenlet jobb oldalán lévő ismeretlenek. Az ismeretlenekhez csak 1-es és 0-t adhatunk értékül (akna vagy aknamentes) úgy, hogy a szomszédok értékeinek összege meg kell hogy egyezzen a cella címkéjével. Ha ez nem teljesül, akkor a cella szomszédai nem lehetnek akna/aknamentes állapotban a megadott konfigurációban. Egy példán keresztül szeretném szemléltetni a problémát. Számoljuk ki egy elképzelt megszorításra $2 = [2, 3] + [1, 9] + [4, 5]$ az összes lehetséges, cellákhoz tartozó értékeket. Egy megszorítás két részből áll. Egy összegből (ebben az esetben 2) és változók tömbjéből (ebben az esetben a $\{[2, 3], [1, 9], [4, 5]\}$). A megszorítás annyit jelent, hogy a változók értékeinek összegének meg kell egyezniük a megszorítás összegével. Az aknakereső játékban egy ilyen megszorítást egy feltárt, számmal jelölt celláról olvashatunk le. Ekkor az összeg a cella száma, a változók a cella szomszédos celláinak [sor,oszlop] párpai. A játékban ilyen koordinátákkal azonosítjuk a cellákat, nem pedig betűkkel.

⁶CSCSP - Coupled Subsets Constraint Satisfaction Problem, a "Megszorításkielégítési-algoritmus" kibővítése a megszorítások halmazokba rendezésével.

Az összes lehetséges megoldás:



3.4. ábra. Példa egy megszorítás megoldásaira.

Egy megszorítás megoldását itt egy fagráffal oldunk meg. Az összes lehetséges megoldást leolvashatjuk egy úttal a start csúsból egy adott levélcsúcsba. Minden elágazásnál egy értéket rendelünk a megszorítás változóihoz. A megoldásunk helyes, ha a változókhoz rendelt értékek összege nem nagyobb a megszorítás összegénél. A megoldás teljes, ha minden változóhoz rendeltünk értéket (levélcsúcsok). A gráf bal oldalán látható egy vágás: itt már biztosan nem találunk megoldást, mert a fennmaradó cellák száma + eddigi egyesnek választott cellák száma $<$ összeg (nem helyes). Máshogy fogalmazva: innen akárhogy választjuk meg a cellák értékeit, ez már biztosan nem fog helyes megoldást eredményezni.

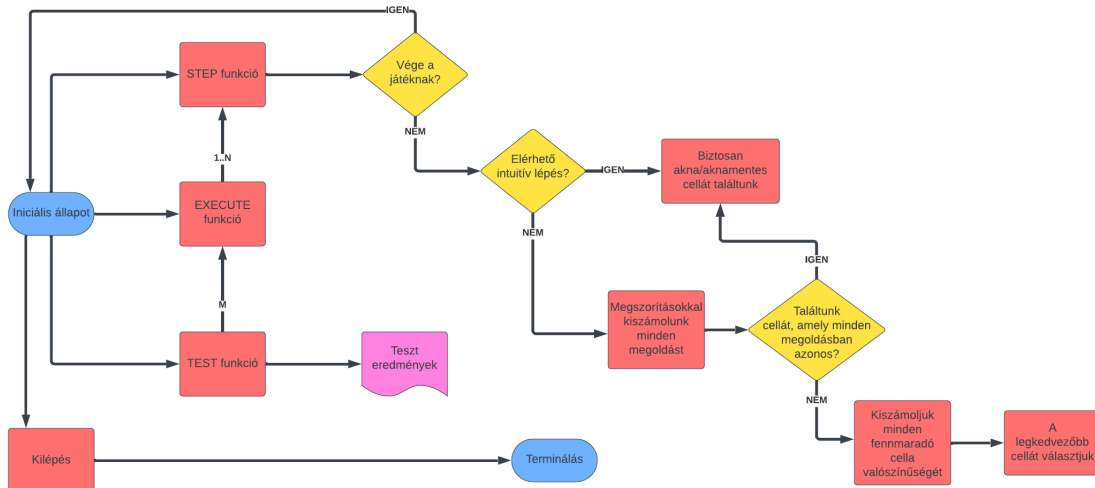
Egy megszorításhalmaz megoldása megegyezik 1db megszorítás megoldásával. A különbség annyi, hogy először ki kell gyűjtenünk a megszorításokban található egyedi cellákat. Ezután hasonlóképp felépíthetünk egy fagráfot. Amikor egy cellának értéket adunk, akkor ebben az esetben azt kell megvizsgálnunk, hogy megsérti-e valamely megszorítást az értékadás. Ha van olyan megszorítás, amit megsértett, akkor vághatunk.

Az általam készített algoritmus ötvözi Adamatzky sejtautomatájának ötletét és a CSCSP algoritmust. A játékot lépésenként oldja meg. Egy lépés három részből áll:

1. A triviális esetek kezelése. Ide tartoznak a korábban bemutatott AFN és AMN esetek. Ha találtunk triviális esetet, akkor a sejtautomata módszerét használjuk és a lépésnek vége. Ha nem találtunk, akkor a következő lépést is hatjsuk végre.
2. A nem triviális esetek kezelése. Ha az 1. lépés nem hozott eredményt, akkor számoljunk megszorításokkal. A 2. és 3. lépés megegyezik a CSCSP algoritmussal. A megszorításokkal ezen a ponton kaphatunk biztos eredményeket. Ha nem kaptunk eredményt, akkor a 3. lépésre haladunk.
3. Tippelés. Ha az előző lépésben nem kaptunk biztos eredményt, akkor a CSCSP algoritmust használva kell a legjobb tippet eszközölnünk.

A specifikációban egy lépésre a "STEP" funkcióval hivatkozunk. A "STEP" funkciót a játék végéig ismételve kapjuk az "EXECUTE" funkciót. A "TEST" funkciót az "EXECUTE" funkció 100 alkalommal való ismétlésével és a végeredmények eltárolásával kapjuk meg.

Az általam használt algoritmust a következő ábrával 3.5 szeretném szemléltetni:



3.5. ábra. Flowchart.

Azért így készítettem el a megoldásom, mert az esetek döntő többségében nincs szükségünk arra, hogy megszorításokat használjunk. A játék nagy részében elegendő

a triviális eseteket kezelünk. Ennél jóval kevesebb esetben kell megszorításokat használnunk és még ennél is kevesebb alkalommal tippelnünk. Bár a szerzők nem részletezték az algoritmusuk működését, a Yimin Tang, Tian Jiang és Yanpeng Hu által készített megoldás terve[8] nagymértékben hasonlít az enyémhez.

Az általam készített algoritmus annyiban tér el a CSCSP modelltől, hogy nem visszalépéses keresést használ egy megszorításhalmaz összes megoldásának kiszámolására, hanem egy rekurzív függvényt:

2. algoritmus Megszorításhalmaz összes megoldásának kiszámolása rekurzióval

```

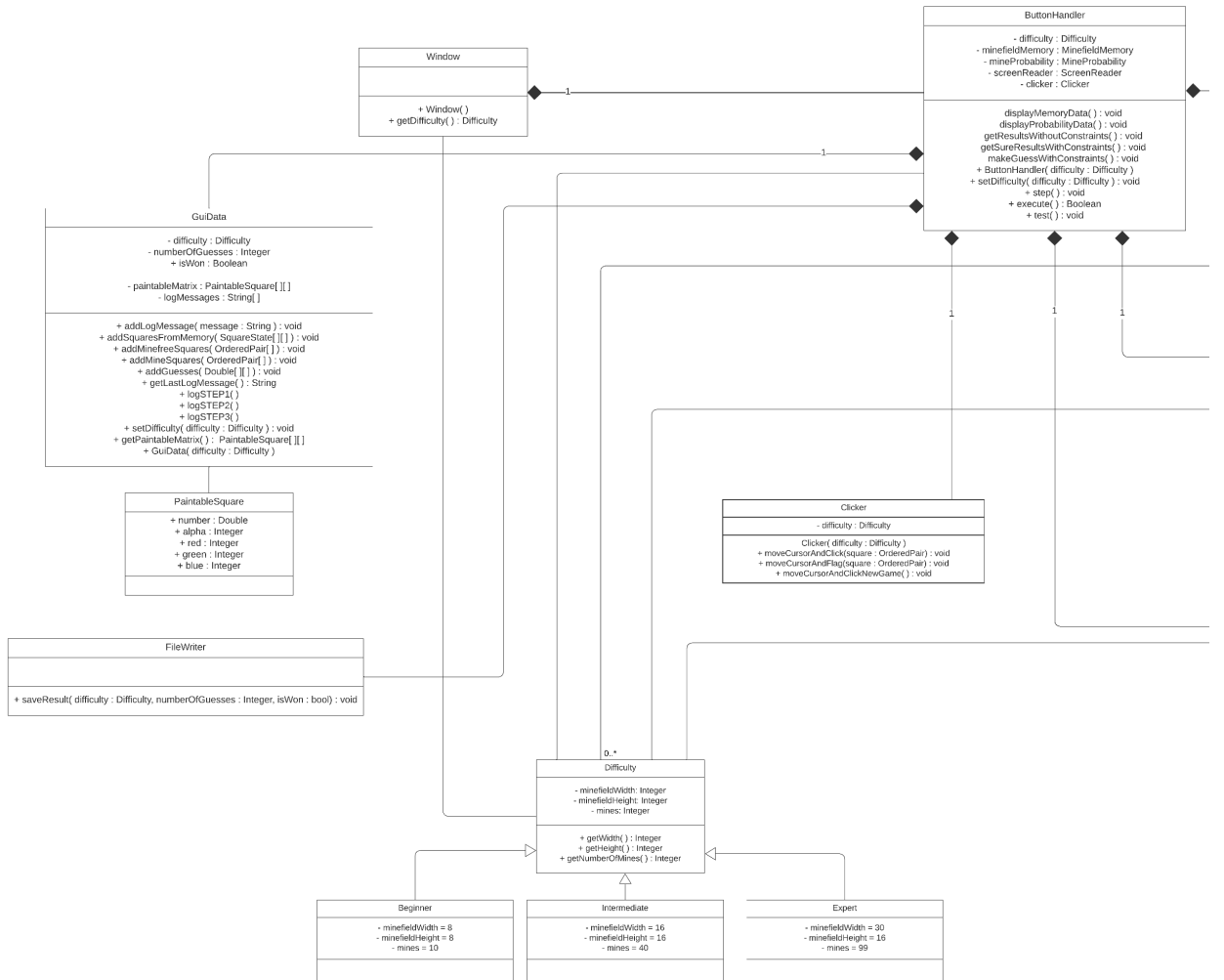
procedure MEGSZORÍTÁSHALMAZ.MEGOLD(jelenlegi_értékek := [ ])
    helyes_hozzarendeles  $\leftarrow$  Megszorításhalmaz.helyes_hozzárendelés-e(jelenlegi_ertekek)
    if helyes_hozzarendeles then
        teljes_hozzarendeles  $\leftarrow$  Megszorításhalmaz.helyes_hozzárendelés-e(jelenlegi_ertekek)
        if teljes_hozzarendeles then
            Megoldasok.hozzátesz(jelenlegi_értékek)
        else
            jelenlegi_ertekek_nullaval  $\leftarrow$  jelenlegi_ertekek
            jelenlegi_ertekek_nullaval += 0
            Megszorításhalmaz.megold(jelenlegi_ertekek_nullaval)
            jelenlegi_ertekek_eggyel  $\leftarrow$  jelenlegi_ertekek
            jelenlegi_ertekek_eggyel += 1
            Megszorításhalmaz.megold(jelenlegi_ertekek_eggyel)
        end if
    end if
end procedure

```

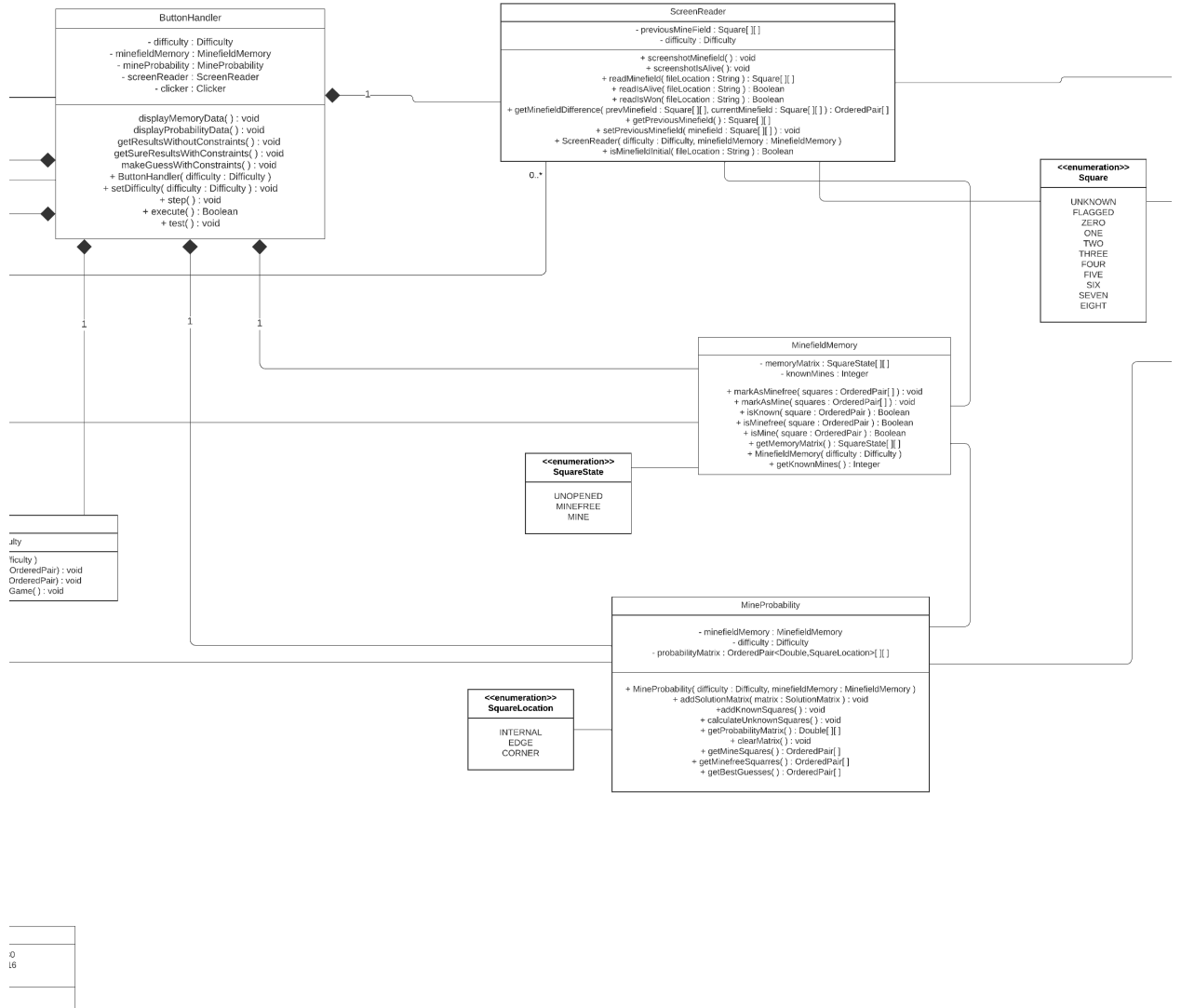
Az algoritmus magyarázata: a megoldás egy tömb, amelyben a megszorításhalmaz változóihoz (megfelelő sorrendben) értékeket adunk. Kezdetben nem adunk értéket egyik változónak sem (üres tömb). Alapértelmezetten a változók értéke 0. A helyes megoldás itt azt jelenti, hogy a változókhoz rendelt értékek összege nem nagyobbak a halmaz egyik megszorításának összegénél sem.

- Ha helyes a megoldás, de nem teljes: a rekurziót megismétljük úgy, hogy a jelenlegi értékekhez hozzáveszünk 0-t. Ezután megismétljük úgy, hogy hozzáveszünk 1-et.
- A helyes és teljes megoldásokat elmentjük.
- A nem helyes megoldásokkal nem folytatjuk a rekurziót.

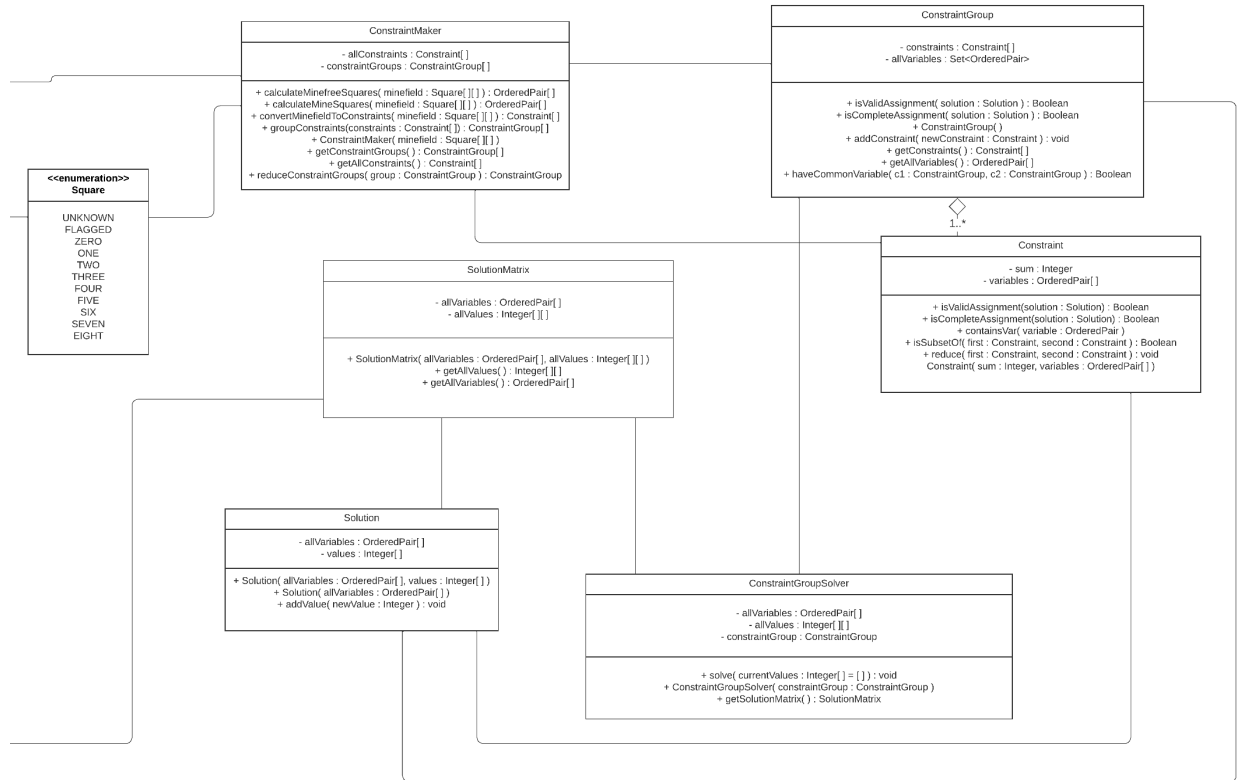
A következő három képen 3.6, 3.7, és 3.8 az alkalmazás UML osztálydiagramja látható három részben (balról jobbra):



3.6. ábra. UML osztálydiagram 1.rész



3.7. ábra. UML osztálydiagram 2.rész



3.8. ábra. UML osztálydiagram 3.rész

Az osztályok fő feladatainak leírása:

- Window - az alkalmazás ablaka. Ez az osztály kommunikál a játék ablakával és kezeli az eseményeket.
- ButtonHandler - a gombok funkcionalitását valósítja meg.
- GuiData - az ablak információit tárolja - minden adatot, ami a színes mátrix kiajzolásához szükséges, a CSV fileba írható eredményeket, valamint a log üzeneteket.
- PaintableSquare - Egy színes cella értékét (0-1 közötti szám), valamint a színét tárolja RGBA kóddal. A GuiData osztály használja fel.
- FileWriter - Egy rekord fileba írását valósítja meg.

- Clicker - Az egérrel történő jobb- és bal kattintást valósítja meg.
- Difficulty - Ez az osztály a játék nehézségi szintjeihez köthető információkat tartalmazza. Leszármazott osztályai: Beginner, Intermediate, Expert.
- ScreenReader - A játéktérrel készít képernyőképet, valamint a képernyőkép kiértékelését valósítja meg. A képernyőkép kiértékelését követően megkapjuk mátrix formában a játéktér jelenlegi állását.
- Square - Felsoroló osztály, amely az egyes cellák lehetséges értékeit adja meg. A Screenreader osztály használja.
- ConstraintMaker - A ScreenReader által beolvasott cellákat (Square) átalakítja megszorításokká (Constraint). Ezeket a megszorításokat halmazokba rendezi (ConstraintGroup), majd redukálja azokat.
- Constraint - egy megszorítás reprezentációja.
- ConstraintGroup - Megszorítások (Constraint) halmaza. Két megszorítás akkor tartozik egy halmazba, ha legalább egy változójuk közös.
- ConstraintGroupSolver - Egy megszorításhalmaz (ConstraintGroup) összes lehetséges megoldását számolja ki (minden cella értéke 0 vagy 1 lehet. megadja az összes lehetséges konfigurációt, amiben a cellák értékei nem sértenek egy megszorítást sem a halmazból).
- Solution - A megszorításhalmazhoz (ConstraintGroup) tartozó 1 db lehetséges megoldás. Részei: a megszorításhalmazban megtalálható cellák helyei + az ezekhez tartozó 1 db lehetséges konfiguráció.
- SolutionMatrix - A ConstraintGroupSolver egy ilyen mátrixban gyűjti az összes megoldást (Solution). A mátrix fejlécében a cellák helyei találhatók, a mátrix sorai pedig egy-egy megoldást (0-1 sorozatok), amiket ezek a cellák felvehetnek.
- MinefieldMemory - Az alkalmazás memóriája, itt tárolja a már feltárt cellák értékeit.
- SquareState - felsoroló osztály. Amit ábrázol: egy cella a memória szempontjából három állapotban lehet: akna, aknamentes, nem ismert. A MinefieldMemory használja.
- MineProbability - A kiszámolt és már memóriából ismert információkból készít egy mátrixot. A mátrix cellái egy 0-1 közötti számot tartalmaznak. Jelentésük: mekkora eséllyel van akna az adott cellán. Ebből a mátrixból tudunk biztosan helyes lépéseket leszűrni, valamint ha tippelnünk kell, akkor ebből a mátrixból

választjuk azt a cellát, ami a legkisebb eséllyel akna.

- SquareLocation - Felsoroló osztály. Egy cella helye a játéktéren három fajta lehet: sarok, él, belső cella. Ezt az osztályt a MineProbability osztály használja egy heurisztikára: ha tippelnünk kell és több cellának is megegyezik a valószínűségi értéke, akkor válasszunk egy olyat, amely sarok $>$ él $>$ belső cella. Ez azért lényeges, mert minél kevesebb szomszédja van egy cellának, azokról annál több információt hordoz.

3.3. Implementáció

Ebben a fejezetben két kulcsfontosságú részét szeretném kiemelni a programnak: hogyan kommunikálnak egymással a korábban megtervezett osztályok és hogy hogyan reagál a program a felhasználói inputokra.

3.3.1. Osztályok kommunikációja

Kezdetben a felhasználó valamely gombot megnyomja a felhasználói felületen, ha meg akar hívni egy funkciót. Ezt az eseményt a callback függvény kapja el (lásd Eseménykezelés), majd továbbítja az adott Window objektumnak. Az itt meghívott `Window.HandleButtonClick(int ButtonID)` függvény minden ilyen esemény bekövetkezésekor leellenőrzi, hogy az aknakereső ablaka megfelelő helyen van-e. Ha igen, akkor beállítja a `ButtonHandler` osztály `Difficulty` változóját a jelenlegi nehézségi szintre [`ButtonHandler.setDifficulty()`]. Ez nagyon fontos része a programnak:

```
1
2 void ButtonHandler::setDifficulty(Difficulty difficulty) {
3
4     this->difficulty = difficulty;
5     delete this->screenReader;
6     this->screenReader = new ScreenReader(difficulty,
7         minesweeperHandle);
8
9     screenReader->screenshotMinefield();
10    bool isInitial = screenReader->isMinefieldInitial("
11        minesweeper_screenshot.png");
12    if (!isInitial) { return; }
13
14    //only do the following part if the user started a new game
15    delete this->minefieldMemory;
16    delete this->mineProbability;
17    delete this->clicker;
18    delete this->guiData;
19
20    this->minefieldMemory = new MinefieldMemory(difficulty);
21    this->mineProbability = new MineProbability(difficulty, this->
22        minefieldMemory);
23    this->clicker = new Clicker(difficulty, minesweeperHandle);
```

```
21  this->guiData = new GuiData(difficulty);
22
23  substepNumber = 1;
24 };
```

3.1. forráskód. ButtonHandler::setDifficulty

Itt a Difficulty osztály leellenőrzi, hogy a játék iniciális állapotban van-e, tehát hogy a felhasználó új játékot kezdett-e. Így tudunk különbséget tenni a következő esetek között anélkül, hogy a felhasználó expliciten megmondaná, hogy melyiket szeretné:

- A felhasználó a jelenlegi, nem befejezett játék következő megoldási lépését szeretné megkapni. Ekkor az isInitial változó értéke HAMIS lesz.
- A felhasználó új játékot kezdett a jelenlegi-, vagy eltérő nehézségi szinten. Ekkor az isInitial változó értéke IGAZ lesz. Ennek következtében a ButtonHandler osztály dinamukusan allokal minden másik osztályból származtatva új adattagokat magának. Például ilyenkor új MinefieldMemory és új MineProbability objektumokat rendelünk hozzá. Erre azért van szükség, mert a legtöbb osztály rendelkezik valamilyen mátrix adattaggal, amelyeknek ilyenkor nem lesz megfelelő a mérete/ki kell törölni a tartalmát.

3.3.2. Eseménykezelés

Az eseménykezelés a win32 api-nak megfelelően egy callback függvényben van definiálva, amely a window.cpp osztályban található. A program az összes eseményt ezen a message loop-on keresztül kapja meg:

```
1
2 #define TIMER_MOVE_DELAY 1
3
4 LRESULT CALLBACK Window::WindowProc(HWND hwnd, UINT uMsg, WPARAM
   wParam, LPARAM lParam)
5 {
6
7     Window* window = reinterpret_cast<Window*>(GetWindowLongPtr(
        hwnd, GWLP_USERDATA));
8     PAINTSTRUCT ps;
9     HDC hdc;
10    Gdiplus::Graphics* gf;
```

```
11
12     static bool timerRunning = false;
13
14     switch (uMsg) {
15         case WM_DESTROY:
16             PostQuitMessage(0);
17             return 0;
18
19         case WM_COMMAND:
20             if (window == nullptr)
21                 break;
22
23             window->handleButtonClick(LOWORD(wParam));
24             return 0;
25
26         case WM_PAINT:
27
28             hdc = BeginPaint(hwnd, &ps);
29             gf = new Gdiplus::Graphics(hdc);
30             window->paintBackground(*gf, ps);
31             EndPaint(hwnd, &ps);
32
33
34             hdc = BeginPaint(window->getDrawArea(), &ps);
35             gf = new Gdiplus::Graphics(hdc);
36             window->paintMatrix(*gf);
37             delete gf;
38             gf = nullptr;
39             EndPaint(window->getDrawArea(), &ps);
40             return 0;
41
42         case WM_ERASEBKGND:
43             // Return TRUE to indicate that background was erased
44             // to prevent flickering
45             return TRUE;
46
47         case WM_MOVE:
48             // Reset the timer every time the window is moved
49             SetTimer(hwnd, TIMER_MOVE_DELAY, 250, nullptr); // Set
50                 a 1-second timer
51             timerRunning = true;
```

```
50         break;
51
52     case WM_TIMER:
53         // Handle the timer event (timer has expired)
54         if (wParam == TIMER_MOVE_DELAY) {
55             // Call the moveMinesweeperWindow function here
56             window->moveMinesweeperWindow();
57             timerRunning = false; // Reset the flag
58         }
59         break;
60
61     default:
62         return DefWindowProc(hwnd, uMsg, wParam, lParam);
63     }
64 }
```

3.2. forráskód. Message loop

Magyarázat:

- WM_DESTROY: Ez az esemény akkor következik be, ha a felhasználó bezárja az alkalmazást - ekkor kilépünk.
- WM_COMMAND: Akkor következik be, ha a felhasználó valamelyik gombra rákattint a három közül.
- WM_PAINT: Akkor következik be, ha az alkalmazás ablakát újra kell rajzolni. Ekkor expliciten csak a valószínűségi mátrix újrarajzolásával foglalkozunk, a többi ablakelem kirajzolása automatikusan megtörténik.
- WM_ERASEBKGND: Akkor következik be, ha az ablak hátterét törölni kell (ilyen eset például, amikor az ablakot átméretezzük).
- WM_MOVE: Ha az ablakot áthelyezzük, elindítunk egy időzítőt. Amíg az időzítő fut, az aknakereső ablaka nem 'húzza vissza' az alkalmazás ablakát a saját helyére (a két ablaknak együtt kell mozognia).
- WM_TIMER: Ha lejár az időzítő, tehát a felhasználó áthelyezte az ablakot, mozgassuk el az aknakereső ablakát is a megfelelő helyre.

3.4. Tesztelési terv

A teszteseteket két csoportba lehet osztályozni: unit tesztek és end-to-end tesztek. A unit tesztek a googletest framework segítségével implementáltam. Minden

osztálynak külön teszt-osztálya van, amelyeket így neveztem el a projektben: <osztály neve>Test. Az end-to-end tesztet a "TEST" gomb szolgáltatja, így a program egyben a saját teszt környezete is. Azért esett erre a döntésre a választásom, mert a Clicker és ButtonHandler osztály funkcionalitását nehéz unit tesztelni. A unit tesztek elkészítéséhez a fehér doboz tesztelési módszert választottam.

A tesztek közül szeretném kiemelni a *ConstraintMaker* – *reduceConstraintGroup* – 1 tesztet, amely teljesen redukál egy megszorítás halmazt. A program béta verziójában gondot okozott, hogy az algoritmus egy megszorítást csak egyszer redukált. Így több olyan információt nem kapott meg a program, amelyek egyértelmű megoldáshoz vezettek volna. A helyes megoldás, hogy a megszorításhalmazt addig redukáljuk, amíg páronként nem tudunk több redukálást elvégezni a megszorításokra.

A fejlesztés során ScreenReader osztály tesztelése okozta a legtöbb gondot, azon belül is a *readIsAlive* függvény. Ennek a függvénynek a tesztelésére képeket készítettem a játéktábláról játék közben. Ezeket a képeket többször is el kellett készítenem a fejlesztés során. A Clicker osztály metódusainál eleinte gondot okozott, hogy a képernyőbeállításom az alapértelmezett 125%-on volt, ezért a kattintások el voltak csúszva.

3.4.1. Unit tesztek

ConstraintGroupSolver osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>solve - 1</i>	Egy megszorítás halmaz, amely eltérő megszorításokat tartalmaz.	Az eredmény egy mátrix. Fejlécében minden változó egyszer jelenik meg. Tartalma az összes lehetséges megoldás, amit a változók a megadott sorrendben felvehetnek.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>solve - 2</i>	Egy megszorítás halmaz, amely azonos megszorításokat tartalmaz.	Az eredmény egy mátrix. Fejlécében csak a megszorításból kinyert változók jelennek meg. Tartalma az összes lehetséges megoldás, amit a változók a megadott sorrendben felvehetnek.

3.2. táblázat. ConstraintGroupSolver osztály tesztesetek

ConstraintGroup osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>equality - 1</i>	Két megszorítás halmaz, amelyek ugyanazokat a megszorításokat tartalmazzák. a megszorításokat eltérő sorrendben adtuk a halmazokhoz.	IGAZ.
<i>equality - 2</i>	Két megszorítás halmaz, amelyek eltérő megszorításokat tartalmaznak.	HAMIS.
<i>addConstraint - 1</i>	Üres megszorítás halmazhoz adunk egy új megszorítást.	LEHETSÉGES.
<i>addConstraint - 2</i>	Nem üres megszorítás halmazhoz adunk egy új megszorítást, amelynek legalább egy változója megtalálható a megszorítás halmazban.	LEHETSÉGES.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>addConstraint</i> - 3	Nem üres megszorítás halmazhoz adunk egy új megszorítást, amelynek egy változója sem található meg a megszorítás halmazban.	NEM LEHETSÉGES.
<i>addConstraint</i> - 4	Ellenőrizzük, hogy a megszorítás halmaz összes változója egyenlő-e a megszorításokban megtalálható egyedi változókkal.	IGAZ.
<i>isValidAssignment</i> - 1	Helyes és teljes megoldást rendelünk egy megszorítás halmazhoz.	HELYES MEGOLDÁS.
<i>isValidAssignment</i> - 2	A megoldás olyan változókat is tartalmaz, amelyek nincsenek benne a megszorítás halmazban. Nem minden változóhoz rendel értéket. A hozzárendelt értékek nem sérítik meg a megszorításokat.	HELYES MEGOLDÁS.
<i>isValidAssignment</i> - 3	A megoldás megsérti legalább az egyik megszorítást.	NEM HELYES MEGOLDÁS.
<i>isCompleteAssignment</i> - 1	Helyes és teljes megoldást rendelünk egy megszorítás halmazhoz.	TELJES MEGOLDÁS.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isComplete- Assignment - 2</i>	A megoldás olyan változókat is tartalmaz, amelyek nincsenek benne a megszorítás halmazban. Minden változóhoz rendel értéket.	TELJES MEGOLDÁS.
<i>isComplete- Assignment - 3</i>	A megoldás legalább egy változóhoz nem rendel értéket.	NEM TELJES MEGOLDÁS.
<i>haveCommon- Variable - 1</i>	A két megszorítás halmaznak van közös változója.	IGAZ.
<i>haveCommon- Variable - 2</i>	A két megszorítás halmaznak nincs közös változója.	HAMIS.

3.3. táblázat. ConstraintGroup osztály tesztesetek

ConstraintMaker osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>convertMinefield- ToConstraints - 1</i>	Egy 2x2-es játéktábla konvertálása megszorításokká. A játéktábla tartalma: [TWO, UNKNOWN]; [UNKNOWN, FLAGGED];	$1 = [0,1] + [1,0]$

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>convertMinefield- ToConstraints - 2</i>	Egy 3x3-as játéktábla konvertálása megszorításokká. A játéktábla tartalma: [UNKNOWN, UNKNOWN, UNKNOWN]; [UNKNOWN, THREE, FLAGGED]; [UNKNOWN, ONE, UNKNOWN]	$2 = [0,0] + [0,1] + [0,2] + [1,0] + [2,0] + [2,2]$
<i>convertMinefield- ToConstraints - 3</i>	Egy 4x4-es játéktábla konvertálása megszorításokká. A játéktábla tartalma: [ZERO, ZERO, ZERO, ZERO]; [UNKNOWN, UNKNOWN, TWO, ZERO]; [UNKNOWN, UNKNOWN, THREE, ONE]; [UNKNOWN, FLAGGED, UNKNOWN, UNKNOWN]	$2 = [1,1] + [2,1]; 2 = [1,1] + [2,1] + [3,2] + [3,3]; 1 = [3,2] + [3,3];$
<i>GroupConstraints - 1</i>	Egy valós, kezdő játékból vett (8x8-as) játékalás konvertálása megszorításokká és azoknak halmazokba rendezése. A játékalásból három megszorítás halmaz készíthető.	Megkapjuk mind a három megszorítás halmazt és azok tartalmazzák az összes megszorítást.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>GroupConstraints</i> - 2	Ebben a tesztesetben nem egy játékállásból indulunk ki, hanem már elkészített megszorításokból, amelyek egy tömbben találhatók. A teszt lényege, hogy nem számít annak a sorrendje, hogy a tömbben a megszorítások milyen sorrendet vesznek fel. Ha két megszorítás egy halmazba kerül, akkor a tömbben bárhol elhelyezkedhetnek.	Az eredmény három megszorítás halmaz.
<i>reduce-ConstraintGroup</i> - 1	Több redukálást is el kell végezni a megszorítás halmazban.	Megfelelően redukált megszorítás halmaz.
<i>reduce-ConstraintGroup</i> - 2	Nem tudunk egy redukálást sem elvégezni a megszorítás halmazban.	Az eredmény megegyezik a bemeneti megszorítás halmazzal.
<i>reduce-ConstraintGroup</i> - 3	Csak egy redukálást tudunk elvégezni a megszorítás halmazban.	A megszorítások száma nem változik a redukálás után. A megszorítás halmaz összes változója nem változik a redukálás után.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>calculate-MinefreeSquares - 1</i>	Egy 3x3-as játéktábla a bemenet: [ZERO, UNKNOWN, FLAGGED]; [UNKNOWN, ONE, UNKNOWN]; [UNKNOWN, ZERO, UNKNOWN];	Megkapjuk az összes cellát, amely megszorítások nélkül kiszámolható, hogy biztosan aknamentes: [0,1], [1,0], [1,2], [2,0], [2,2] (ebben az esetben az összes ismeretlen cella).
<i>calculate-MineSquares - 1</i>	Egy 3x3-as játéktábla a bemenet: [UNKNOWN, UNKNOWN, FLAGGED]; [ZERO, THREE, ZERO]; [ZERO, ZERO, ZERO];	Megkapjuk az összes cellát, amely megszorítások nélkül kiszámolható, hogy biztosan akna: [0,0], [0,1] (ebben az esetben az összes ismeretlen cella).

3.4. táblázat. ConstraintMaker osztály tesztesetek

Constraint osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isCompleteAssinment - 1</i>	A megszorítás összes változója megtalálható a megoldásban és azok mindegyikéhez értéket rendel a megoldás.	IGAZ.
<i>isCompleteAssinment - 2</i>	A megoldás nem tartalmaz egy változót, amely megtalálható a megszorításban.	HAMIS.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isComplete- Assinment - 3</i>	A megszorítás összes változója megtalálható a megoldásban, de az egyikhez nem rendelt értéket.	HAMIS.
<i>isValid- Assinment - 1</i>	A megszorítás összes változójához rendelt értéket a megoldás. A hozzárendelt értékek összege megegyezik a megszorítás összegével.	IGAZ.
<i>isValid- Assinment - 2</i>	A megszorítás összes változójához rendelt értéket a megoldás. A hozzárendelt értékek összege több, mint a megszorítás összege.	HAMIS.
<i>isValid- Assinment - 3</i>	A megszorítás összes változójához rendelt értéket a megoldás. A hozzárendelt értékek összege kevesebb, mint a megszorítás összege.	HAMIS.
<i>isValid- Assinment - 4</i>	A megszorítás nem minden változójához rendelt értéket a megoldás. A hozzárendelt értékek összege nem haladja meg megszorítás összegét.	IGAZ.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isValidAssinment</i> - 5	A megszorítás nem minden változójához rendelt értéket a megoldás. A hozzárendelt értékek összege meghaladja a megszorítás összegét.	HAMIS.
<i>ContainsVar</i> - 1	Az $1 = [0,2] + [1,2] + [6,6]$ megszorítás tartalmazza az $[1,2]$ változót.	IGAZ.
<i>ContainsVar</i> - 2	Az $1 = [0,2] + [1,2] + [6,6]$ megszorítás tartalmazza a $[8,9]$ változót.	HAMIS.
<i>equality</i> - 1	Két egyenlő megszorítás a bemenet.	IGAZ.
<i>equality</i> - 2	Két egyenlő megszorítás a bemenet, amelyeknek a változói különböző sorrendben vannak.	IGAZ.
<i>equality</i> - 3	Két eltérő megszorítás a bemenet.	HAMIS.
<i>isSubsetOf</i> - 1	Az $1 = [0,0] + [0,1]$; megszorítás valódi része az $1 = [0,0] + [0,1] + [1,0]$; megszorításnak	IGAZ.
<i>isSubsetOf</i> - 2	Az $1 = [0,0] + [0,1] + [1,0]$; megszorítás valódi része az $1 = [0,0] + [0,1]$; megszorításnak	HAMIS.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isSubsetOf - 3</i>	Az $1 = [0,0] + [0,1]$; megszorítás valódi része az $1 = [4,4] + [5,5] + [6,6] + [7,7]$; megszorításnak	HAMIS.
<i>isSubsetOf - 4</i>	Az $1 = [4,4] + [5,5] + [6,6] + [7,7]$; megszorítás valódi része önmagának	HAMIS.
<i>reduce - 1</i>	Redukáljuk az $1 = [0,0] + [0,1] + [1,0]$; megszorítást az $1 = [0,0] + [0,1]$; megszorítással (a második megszorítás valódi része az elsőnek)	Az első megszorításból lett: $0 = [1,0]$; a második megszorítás nem változott.
<i>reduce - 2</i>	Redukáljuk az $1 = [0,0] + [0,1]$; megszorítást az $1 = [0,0] + [0,1] + [1,0]$; megszorítással (az első megszorítás valódi része a másodiknak)	A második megszorításból lett: $0 = [1,0]$; az első megszorítás nem változott.
<i>reduce - 3</i>	Redukáljuk a $2 = [6,6] + [7,7] + [8,8]$; megszorítást a $3 = [9,9] + [10,10] + [11,11] + [12,12]$; megszorítással (a két megszorítás nem valódi része egymásnak)	A megszorítások nem változnak.

3.5. táblázat. Constraint osztály tesztesetek

Difficulty osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>initialization - 1</i>	Inicializálunk Kezdő, Haladó, Profi nehézségi szinteket.	Az inicializált nehézségi szintek megfelelően tartalmazzák a következő információkat: játéktábla szélessége, játéktábla magassága, aknák száma.

3.6. táblázat. Difficulty osztály tesztesetek

MinefieldMemory osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>MarkAsMineFree - 1</i>	Kezdő játék memóriában aknamentesnek jelöljük a [3,2]-es cellát.	A jelölés előtt a cella 'nem feltárt'. Jelölés után a cella 'feltárt' és 'aknamentes'.
<i>MarkAsMine - 1</i>	Kezdő játék memóriában aknának jelöljük a [3,2]-es és [4,2]-es cellákat.	A jelölés előtt a cellák állapota 'nem feltárt'. Jelölés után a két cella állapota 'feltárt' és 'akna'.

3.7. táblázat. MinefieldMemory osztály tesztesetek

MineProbability osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>addSolutionMatrix - 1</i>	Kezdő játékhoz tartozó valószínűségi mátrixhoz egy darab megoldásmátrixot adunk hozzá.	A valószínűségi mátrixban [0-1] közötti értékekkel megjelennek az ismert cellák valószínűségei. Az ismeretlen cellák -2 értéket kapnak.
<i>addSolutionMatrix - 2</i>	Kezdő játékhoz tartozó valószínűségi mátrixhoz három darab megoldásmátrixot adunk hozzá.	A valószínűségi mátrixban [0-1] közötti értékekkel megjelennek az ismert cellák valószínűségei. Az ismeretlen cellák -2 értéket kapnak.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>clearMatrix - 1</i>	Kezdő játékhoz tartozó valószínűségi mátrixhoz három darab megoldásmátrixot adunk hozzá, majd töröljük a valószínűségi mátrix tartalmát.	A valószínűségi mátrixban minden cella -2 értéket kap.
<i>addKnownSquares - 1</i>	Kezdő játékhoz tartozó memóriában megjelölünk 'akna' és 'aknamentes' cellákat. Ezeket hozzáadjuk a valószínűségi mátrixhoz.	A valószínűségi mátrixban a memóriából kapott cellák -1, az ismeretlen cellák -2 értéket kapnak.
<i>calculateUnknownSquares - 1</i>	Kezdő játékhoz tartozó memóriában megjelölünk 'akna' és 'aknamentes' cellákat. Ezeket hozzáadjuk a valószínűségi mátrixhoz. Ezen kívül a valószínűségi mátrixhoz három darab megoldásmátrixot adunk.	A valószínűségi mátrixban a memóriából kapott cellák -1, a többi cella [0,1] közötti értéket kap. A [0,1] közötti értékek a megoldásmátrixokból kaphatók meg, vagy ismeretlen cellák esetén egy általános valószínűséget kapunk.
<i>calculateUnknownSquares - 2</i>	Haladó játékhoz nem adunk sem a memóriából információt, sem megoldásmátrixok útján.	A valószínűségi mátrixban minden cella esetén egy általános valószínűséget kapunk ([0-1] közötti értékek).
<i>getBestGuess - 1</i>	Kezdő játékhoz nem adunk sem a memóriából információt, sem megoldásmátrixok útján.	A legjobb lépések ez esetben a sarokcellák.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>equality - 1</i>	Ebben a tesztesetben az osztály belső cella reprezentációját vizsgáljuk, azon belül is az egyenlőségét. Egy ismeretlen belső cella ([-2,INTERNAL]) nem egyezik meg egy ismeretlen élcellával ([-2,EDGE]), de két ismeretlen élcella meg egyezik (ugyanolyan eséllyel választjuk őket).	IGAZ.
<i>smallerThan - 1</i>	Ebben a tesztesetben az osztály belső cella reprezentációját vizsgáljuk, azon belül is az egyenlőtlenségét. A következő cellaértékeket vizsgáljuk: a := [-2, CORNER]; b := [0.562, EDGE]; c := [1, INTERNAL]; d := [0.567, INTERNAL];. a < b és b < c és b < d.	IGAZ

3.8. táblázat. MineProbability osztály tesztesetek

ScreenReader osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>readIsAlive - 1</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvényrel, miközben a játék iniciális állapotban volt.	IGAZ.
<i>readIsAlive - 2</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvényrel, miközben a játék elvesztett állapotban volt.	HAMIS.
<i>readIsAlive - 3</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvényrel, miközben a játék megnyert állapotban volt.	HAMIS.
<i>readIsWon - 1</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvényrel, miközben a játék iniciális állapotban volt.	HAMIS.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>readIsWon - 2</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvénnyel, miközben a játék elvesztett állapotban volt.	HAMIS.
<i>readIsWon - 3</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotIsAlive függvénnyel, miközben a játék megnyert állapotban volt.	IGAZ.
<i>readMinefield - 1</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: kezdő, a játék állása: iniciális állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>readMinefield - 2</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: kezdő, a játék állása: elkezdett állapotban volt.	A függvény megfelelően olvassa a cellákat.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>readMinefield</i> - 3	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: kezdő, a játék állása: megnyert állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>readMinefield</i> - 4	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: haladó, a játék állása: iniciális állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>readMinefield</i> - 5	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: haladó, a játék állása: elkezdett állapotban volt.	A függvény megfelelően olvassa a cellákat.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>readMinefield</i> - 6	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: haladó, a játék állása: megnyert állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>readMinefield</i> - 7	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: profi, a játék állása: iniciális állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>readMinefield</i> - 8	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: profi, a játék állása: elkezdett állapotban volt.	A függvény megfelelően olvassa a cellákat.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>readMinefield - 9</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: profi, a játék állása: megnyert állapotban volt.	A függvény megfelelően olvassa a cellákat.
<i>getMinefield-Difference - 1</i>	Két azonos nehézségi szintű játéktábla a bemenet, amelyek iniciális állapotban vannak.	Nincs különbség, üres tömböt kapunk.
<i>getMinefield-Difference - 2</i>	Két azonos nehézségi szintű játéktábla a bemenet, amelyek különböző állapotban vannak.	Egy tömböt kapunk, amely tartalmazza az összes cellát, ahol a két játéktábla értékei nem egyeznek meg.
<i>isInitial - 1</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvénnyel, miközben a játék nehézségi szintje: kezdő, a játék állása: iniciális.	IGAZ.

<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>isInitial - 1</i>	Egy kép a bemenet, amelyet a program futása közben készítettem a screenshotMinefield függvényvel, miközben a játék nehézségi szintje: kezdő, a játék állása: elkezdett.	HAMIS.

3.9. táblázat. ScreenReader osztály tesztesetek

SolutionMatrix osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>equality - 1</i>	Két megoldásmátrix a bemenet, amelyek megegyeznek, de a mátrix oszlopainak sorrendje fel van cserélve.	IGAZ
<i>equality - 2</i>	Két megoldásmátrix a bemenet, amelyek különböznek.	HAMIS.
<i>equality - 2</i>	Két megoldásmátrix a bemenet, amelyek megegyeznek, de a mátrix oszlopainak- és sorainak sorrendje is fel van cserélve.	IGAZ.

3.10. táblázat. SolutionMatrix osztály tesztesetek

Solution osztály		
<i>metódus - teszt ID</i>	<i>bemenet</i>	<i>elvárt kimenet</i>
<i>addValue - 1</i>	A megoldáshoz hozzáadtunk egy 1-es értéket.	A hozzáadás sikeres volt.

3.11. táblázat. Solution osztály tesztesetek

3.4.2. End-to-end tesztek

Az end-to-end tesztek eredményét úgy kaptam meg, hogy a "TEST" gomb funkcionalitását használva 100-esetes tesztek végzettem a játék minden nehézségi szintjén. Ilyen tesztek közül 5-5 darabot készítettem (ez minden nehézségen összesen 500-500 darab végigjátszást jelent). Az így kapott eredmények a következő táblázatban találhatók:

	Kezdő	Haladó	Profi
Nyerési átlag (%)	73.6	69.8	25.8
Nyerési átlag szórása	2.24	3.06	3.66
Tippek száma átlagosan	2.25	2.4	4.27
Tippek számának szórása	1.29	1.39	2.52

3.12. táblázat. Tesztelési eredmények

4. fejezet

Összegzés

4.1. Összefoglalás

A program megoldja az akna kereső játékot, annak is a MinesweeperX változatát. Ezt úgy teszi, mintha egy ember lenne, aki játszik a játékkal: az egeret használja a játék navigálásához és képernyőképek alapján olvassa be a játék jelenlegi állását. Erre azért volt szükség, mert így nem használunk semmilyen csalást a játék megnyeréséhez. Pusztán azokkal az információkkal dolgozunk, amelyek egy emberi játékos számára is elérhetők. A játék tábláját nem cellánként értelmezzük, hanem megszorításokként, amelyeket a cellákból nyerünk ki. Ezzel a gondolatmenettel olyan összefüggésekre jöhetünk rá, amelyek cellánkénti értelmezésnél nem láthatók. A program segíthet a játékosoknak fejleszteni logikai és stratégiai készségeiket, ami a játéktudásukat javítja.

Az algoritmus nem oldja meg nagyobb pontossággal a feladványt, mint egy tapasztalt játékos, de sokkal gyorsabban igen. A kezdő játékot körülbelül 1-2, a haldó nehézségi fokozatot 5-10, a profi játékot 20-25 másodperc alatt megoldja. Ha kezdő játékosok vagyunk és gyakorolni szeretnénk a játékot, a felhasználói felület segítségével megállhatunk egy adott döntésnél és elemezhetjük azt. Az emberi agy mindenben mintázatokot keres, hogy könnyebben értelmezni tudja a minket körülvevő világot. A megszorítások használata a megoldás során hasonló célt szolgál. A feladvány rengeteg féle variációban előfordulhat, de vannak mintázatok, amelyeknek a megoldása mindig azonos. Ezeket begyakorolva válhatunk profi játékosná.

4.2. Továbbfejlesztési lehetőségek

Párhuzamosítást lehetne használni, amikor a megszorításhalmazokból kiszámoljuk a megoldásmátrixokat. Jelenleg a program ezeket egymás után oldja meg. Mivel a megszorításhalmazok egymástól különálló feladatok, ezért nincs akadálya, hogy többszálúsággal felgyorsítsuk ezt a folyamatot.

A program jelenleg csak 100%-os képernyő méretezésen működik megfelelően. A továbbiakban ezt orvosolni lehetne azzal, hogy a program elején elmentjük egy változóba a képernyő méretezését, melynek értéke a következők lehetnek egy modern Windows rendszeren: 1, 1.25, 1.5, 1.75; Ezzel a számmal meg kell szorozni azokat a koordinátákat, amelyek segítségével a Clicker osztály kattint a képernyőn, valamint a ScreenReader osztály olvas a képernyőről.

A valószínűségi mátrix és a log felület csak a "STEP" gomb megnyomásakor frissülnek, az "EXECUTE" gomb megnyomásakor nem. Úgy lehetne módosítani a programot, hogy ezek az ablakelemek frissüljenek megoldás közben is folyamatosan. Ehhez egy komplexebb renderelési módszert kell alkalmazni a színes négyzetek megjelenítésére. A számítógépes grafika kínál megoldást a problémára: tripla pufferezés. Ezt a módszert alkalmazva három puffert használunk rendereléshez: elülső puffer, középső puffer és hátsó puffer. Az elülső puffer továbbra is a képernyőn jelenleg látható frame-et tartalmazza, a hátsó puffer pedig a képernyőn kívüli állítja elő az új frame-et. A renderelés során a grafikus adatok a hátsó pufferbe kerülnek. Amikor egy frame renderelése befejeződött, a hátsó puffer felcserélődik a középső pufferrel. Ez azt jelenti, hogy a középső puffer tartalmazza a legutóbb elkészült frame-et, míg az elülső puffer továbbra is az előző frame-et jeleníti meg. Az elülső puffer végül frissül a középső puffer tartalmával, amikor az megfelelő (például függőleges törléskor), ami segít megelőzni a képernyő szakadását.

A felhasználói felületet modernebb stílussal lehetne ellátni. A program jelenleg a Windows alatt alapértelmezett kinézetű ablakelemeket használ.

Az alkalmazás jelenleg egy dummy weboldalról tölthető le. Ezt felhasználói szempontból nézve érdemes lenne orvosolni.

Köszönetnyilvánítás

Köszönöm Fülöp Endrének, szakdoglozati konzulensemnek a segítségét.
Köszönöm Édesanyámnak és barátaimnak, hogy mindig mellettem voltak egyetem
éveim alatt.

Irodalomjegyzék

- [1] Wikipedia. *Minesweeper (video game)*. [https://en.wikipedia.org/wiki/Minesweeper_\(video_game\)](https://en.wikipedia.org/wiki/Minesweeper_(video_game)). [Accessed on 2023-12-03].
- [2] Andrew Adamatzky. „How cellular automaton plays Minesweeper”. *Applied Mathematics and Computation* 15.2 (1997), 45–60. old. DOI: 10.1016/S0096-3003(96)00117-8.
- [3] Chris Studholme. „Minesweeper as a constraint satisfaction problem”. *Unpublished project report* (2000).
- [4] Richard Carini. *Circuits, Minesweeper, and NP Completeness*. https://web.math.ucsb.edu/~padraic/ucsb_2014_15/ccs_problem_solving_w2015/NP3.pdf. [Accessed on 2023-12-03].
- [5] David Becerra. *Algorithmic Approaches to Playing Minesweeper*. 2015. URL: <http://nrs.harvard.edu/urn-3:HUL.InstRepos:14398552>.
- [6] Kasper Pedersen. „The complexity of Minesweeper and strategies for game playing”. *Project report, univ. Warwick* (2004).
- [7] Andrew Fowler és Andrew Young. „Minesweeper: A statistical and computational analysis”. (2004). [Accessed on 2023-12-03].
- [8] Yimin Tang, Tian Jiang és Yanpeng Hu. „A minesweeper solver using logic inference, CSP and sampling”. *arXiv preprint arXiv:1810.03151* (2018).

Ábrák jegyzéke

1.1. A cellák lehetséges állapotai [1]	3
2.1. Az játék ikonja (bal) és a megoldó program ikonja (jobb)	7
2.2. Felhasználói felület iniciális állapotban	8
2.3. Felhasználói felület a program használata közben	9
2.4. Felhasználói felület az "EXECUTE" parancs végrehajtását követően	9
2.5. A tesztek eredményei táblázatos formában	10
2.6. Hibaüzenet a program megnyitásánál	10
2.7. Hibaüzenet a program futása közben I.	11
2.8. Hibaüzenet a program futása közben II.	12
3.1. Mockup	19
3.2. Példák konzisztens és inkonzisztens játéktáblákra	21
3.3. Példa megszorításokra. [5]	25
3.4. Példa egy megszorítás megoldásaira.	28
3.5. Flowchart.	29
3.6. UML osztálydiagram 1.rész	31
3.7. UML osztálydiagram 2.rész	32
3.8. UML osztálydiagram 3.rész	33

Táblázatok jegyzéke

3.1. felhasználói történetek	17
3.2. ConstraintGroupSolver osztály tesztesetek	41
3.3. ConstraintGroup osztály tesztesetek	43
3.4. ConstraintMaker osztály tesztesetek	46
3.5. Constraint osztály tesztesetek	49
3.6. Difficulty osztály tesztesetek	50
3.7. MinefieldMemory osztály tesztesetek	50
3.8. MineProbability osztály tesztesetek	52
3.9. ScreenReader osztály tesztesetek	58
3.10. SolutionMatrix osztály tesztesetek	58
3.11. Solution osztály tesztesetek	59
3.12. Tesztelési eredmények	59

Algoritmusjegyzék

1.	Algorithm 1 Single Point Strategy	25
2.	Megszorításhalmaz összes megoldásának kiszámolása rekurzióval . . .	30

Forráskódjegyzék

3.1. ButtonHandler::setDifficulty	36
3.2. Message loop	37