Debreceni Egyetem

Informatikai Kar

Mesterséges intelligencia alkalmazások

Témavezető
Dr. habil Nagy Benedek
egyetemi docens

Készítette:
Czomba László
Programtervező informatikus(BSc.)

Debrecen
2010

Tartalomjegyzék

I. Bevezetés

Szakdolgozatomban mesterséges intelligenciával kereső algoritmusokkal foglalkozok. Különböző keresési stratégiákat fogok bemutatni, és kiemelni előnyeiket, hátrányaikat. Igyekszem részletesen foglalkozni a heurisztikus keresőrendszerekkel is. Bemutatok egy logikai játék állapottér reprezentációját heurisztikával együtt. Előbbinek a *Magyar Körök*et vagy angol nevén *Hungarian Rings*-t választottam. Elkészítem a reprezentációját, és a programkódját C# nyelven, valamint a grafikus megjelenítését WPF technológiával. A keresőrendszer algoritmusát a feladat tulajdonságai szerint alakítom, figyelembe véve a lehetőségeket az optimalizálásra, hogy a keresés minél hatékonyabb legyen.

II. Logikai játékok

Logikai játékoknak nevezzük az olyan feladatokat, játékokat, melyek megoldásához egy stratégiát kell alkalmaznunk, hogy a célt elérjük. Ilyen stratégia nélkül pusztán a véletlenre hagyatkozva, szinte lehetetlen a megoldásuk a lehetséges kimenetelek száma miatt. Észjátéknak is nevezhetjük ezeket, alkalmasak arra, hogy gondolkodásunkat javítsák, agyunkat megmozgassák. Felsorolás szinten olyanokról beszélek, mint a sakk, memória, nim, amőba, sodoku és megszámlálhatatlan táblajáték.

a. Sakk

Az általános sakkot két játékos egy nyolcszor nyolcas, négyzetrácsos táblán játssza, ahol kezdésben mindkét félnek ugyanannyi és ugyanolyan bábuja van. Nyolc gyalog, két bástya, két huszár, két futó, egy király és egy királynő. Egy játékos akkor nyer, ha az ellenfele királyát támadja, és ő nem tudja elhárítani. A sakkot más szabályok szerint is lehet játszani:

- Francia sakk: abban különbözik az eredetitől, hogy itt az a győztes, akinek hamarabb elfogynak a bábui. A többi eltérés abban mutatkozik, hogy ha valaki támadó helyzetbe kerül, akkor kötelező ütni, valamint a király is leüthető és nincs sakkadás.
- Fischer-féle sakk: több elnevezése van (reformsakk, 960-as sakk), arra vonatkozó változtatás, hogy a kezdő stratégiák szerepe csökkenjen, mégpedig úgy, hogy a tiszt sorban a tisztek sorrendje sorsolás alapján dől el, pár apró megszorítással.
- Tandemsakk: nagyon különleges, abból adódóan, hogy csapatban játsszák, ketten egy csapatban, Ha az egyik játékos leüt egy bábut, akkor azt a csapattársának adja, aki azt a sajátjaként használja tovább.

b. Memória

Ezt a játékot kártyákkal lehet könnyen játszani, melyek közt mindegyiknek van egy párja, amely ugyan olyan. A játék során kettesével kell felfedni a kártyákat, és ha megegyeznek, akkor azokkal nem kell tovább játszani, ha nem, újra el kell tüntetni. A játék akkor ér véget, mikor az összesnek meg van a párja. A játék kezdődhet úgy is, hogy bizonyos ideig lehet nézni a kártyákat, de lehet azonnal, vakon kezdeni.

c. Nim

Minden nim játék alapja, hogy bizonyos számú kupacból veszünk el valamennyi darabot, legkevesebb egyet, legfeljebb az összest, és az veszít, aki az utolsót elveszi. A játékot

lehet módosítani, hogy több kupac legyen, valamint az egyszerre elvehetőek számának a maximalizálásával.

d. Amőba

Négyzetrácsos táblán két játékos játszhatja. Kezdetben a tábla üres, majd a játékosok felváltva teszik le egyesével a jelüket a táblára. Az a nyertes, akinek hamarabb összegyűlik a jeleiből egymás mellett öt.

e. Sodoku

Egy kilencszer kilences négyzetrácsos táblán háromszor hármas mezőket különítünk el, összesen kilencet. Az egyes rácsokon az egyestől a kilencesig lehetnek számok. A feladat az, hogy minden mezőn szerepeljen egy szám, mégpedig úgy, hogy az egyes sorokban és oszlopokban, valamint abban a háromszor hármas mezőben, amelyikhez tartozik, pontosan egyszer forduljon elő.

f. Bűvös kocka

Az eredeti kocka alakú és minden oldalát kilenc-kilenc darab kiskocka alkotja. Egyszerre egy sort lehet mozgatni, és így kell elérni, hogy minden oldala pontosan egyszínű kockákból álljon. Ennek is több változata van, melyek alakjukban, valamint a kiskockák számában különbözhetnek.

III. Állapottér-reprezentáció

Mesterséges intelligenciát használó kereséshez szükséges a megoldandó problémát modellezni, azt formálisan leírni, ami végül kiterjeszthető és számítógéppel végrehajtható. A modellezés folyamán ki kell emelni azokat a tulajdonságait a feladatnak, amik lényegesek és a megoldáshoz szükségesek. A formális leíráshoz az állapottér- reprezentáció használható. Az előbb említett egyedi tulajdonságok, amivel a feladatot lehet jellemezni, megadhatók valamilyen diszkrét értékekkel, amik összessége állapotonként más-más értékeket vesznek fel. Egy állapotot *a*-val jelölök, halmazukat A-val. Lehetséges, hogy a felvehető értékek alkotta halmazok Descartes szorzata túlmutat a valódi állapotterén, ilyenkor kényszerfeltétel bevezetéssel lehet szűkíteni a halmazt. A megoldás megkeresése mindig egy konkrét állapotból indul, jelölése *k* ($k \in A$). A feladat megkonstruáláskor fontos eldönteni, hogy mi a célja a programnak. Céljai lehetnek egy bizonyos célállapot felderítése, esetleg az összes, de lehet a célállapothoz vezető lépések sorozata is, vagy valamilyen szempont figyelembe vételével a legjobb megoldás megtalálása. A célállapotok halmazra a C jelölést fogom használni. Célállapotot meg lehet adni felsorolással, ha ismerjük az összest, vagy egy feltétellel, ami kijelöli a $C \subset A$ valódi részhalmazát az állapottérnek ($C = \{a|$ célfeltétel$(a)\}$). Végül ahhoz, hogy a *k* állapotból elérjünk egy $c \in C$ állapotba, szükséges az állapotokon értelmezett műveletek végrehajtása. Egy operátornál szükséges megadni az alkalmazási feltételt, ami kijelöli az értelmezési tartományát, vagyis azokat az állapotokat, melyeken végre lehet hajtani az adott műveletet. Továbbá, hogy az *a* állapoton alkalmazott $o \in O$ operátor milyen *a'* állapotot eredményez. Ha esetleg a végrehajtási költség nem egységnyi, azt is itt lehet megadni. Az állapottér-reprezentáció előbb megadott négyesére a $p=<A, k, C, O>$ jelölést alkalmazom. Az ismereteket grafikus módon is lehet ábrázolni gráfreprezentációval. Az egyes állapotok és az operátorokat egy reprezentációs gráfon tüntetjük fel, csúcsok és élekként. Az élek lehetnek súlyozottak, az operátorok súlyai szerint. *Gráfreprezentációról akkor beszélünk, ha megadjuk a reprezentációs gráfot, valamint az abban keresendő irányított út kezdő és végpontját*[1]. Egy kereső rendszert értékelhetünk optimalitás, teljesség, futási idő és tárigény szerint. Optimális megoldásról akkor beszélünk, ha az operátorok valamilyen metrikája szerint a legkevésbé költséges megoldást találja meg a rendszer. Ha a rendszer minden olyan esetben, amikor az állapottér tartalmazza a megoldást, megtalálja azt, teljes a keresés. Tárigénye a keresőnek attól függ, hogy a megismert állapotok közül mekkora részét tartja az adatbázisban. Időigény alatt pedig a megoldás megtalálásához szükséges időt értjük, vagy mialatt belátja, hogy nincs megoldás.

IV. Kereső stratégiák

A keresési stratégia megválasztása az egyik leglényegesebb pontja a keresési algoritmus elkészítésének. A stratégiákat több szempont szerint lehet osztályozni. Egyrészről lehet módosítható, illetve nem módosítható. Az előbbinek két nagy csoportja van, úgymint visszalépéses és gráfkereső stratégia. Másik kérdés, hogy használ-e valamilyen többlet információt a keresés, eszerint beszélünk informált, vagy nem informált keresésről. Osztályozhatjuk aszerint, hogy elsődleges vagy másodlagos stratégia. Továbbá a hatékonyságot nagymértékben befolyásoló tényező, hogy keresés során többször előforduló állapotokat, kezeli-e az algoritmus, és ha igen, akkor hogyan. Az általános működésük megfelel egy egyszerű algoritmikus feladatmegoldásnak. Valamilyen módon kiválaszt egy állapotot, megvizsgálja, hogy célállapot-e. Ha igen, vége az algoritmusnak, ha nem, akkor valamilyen kiterjesztő műveletet végrehajt az állapoton és ez által változik az adatbázis ismerete.

a. Nem módosítható keresés

Legfontosabb tulajdonságuk, hogy a kereső algoritmus adatbázisa egyetlen csúcsot tartalmaz, mégpedig az aktuális csúcsot. Egy adott állapot szülőjét nem tartjuk nyilván, a kezdő csúcsból odáig vezető utat nem ismerhetjük. Miután egy operátor végrehajtásra kerül az aktuális csúcsot az algoritmus „elfelejti” és csak az újonnan előállót tartja meg. Fontos hogy a lehetőségek közül a meglévő információk alapján a legjobb operátort választja, mert nincs lehetőség a visszatérésre, hacsak az operátorok nem vezetnek vissza. Ha a keresés egy bizonyos szemantika alapján keres, és az a szemantika a bejárt út egyik elemére irányítja a keresést, vagyis körre fut, akkor ugyan azt az utat fogja újra és újra bejárni. Előnye, hogy tárigénye minimális, viszont kevésbé hatékony.

b. Módosítható keresés

A keresési algoritmusok nyilvántartanak egy adatbázist, melyek a bejárt keresési fának egy részét vagy az egészét megőrzik. Az adatbázis csúcsokból épül fel. Egy csúcs tartalmaz egy állapotot, egy hivatkozást a szülő csúcsára, egy operátort, amit alkalmazva a szülő csúcson előállt valamint a mélységét. Esetleg tartalmazhat egy költséget is amennyiben az operátorok költsége nem egységnyi, ekkor a csúcs költsége a szülő csúcs költségének és az operátor költségének az összege. A kereső rendszer ebből választ ki egy csúcsot kiterjesztésre, valamilyen szempont szerint. Az egyik legfontosabb különbség a rendszerek között az, hogy

mi alapján választja ki kiterjesztésre a következő csúcsot. Lehetőség van a „rosszul” megválasztott út módosítására, vagy a visszavonására, hogy aztán egy másik úton haladjon tovább, valamint nem kell újra és újra ugyan azokat a hibákat elkövetnie

1. Visszalépéses kereső

Az adatbázis nem az egész idáig bejárt fát tartalmazza, csak az kezdő csúcsból vezető aktuális útvonal csúcsait. Ezáltal lehetőség van a körök elkerülésére az adott útvonalon belül, viszont egy másik alternatív úton már előfordult csúcsot nem tud elkerülni. Több ok indokolhatja a visszalépést az útvonalon:

- *ha nem vezet belőle tovább út, azaz nincs belőle kivezető él; ekkor ugyanis egy zsákutca végére jutottunk*
- *ha egy csúcsról valamilyen (heurisztikus) ismeret alapján kiderül, hogy nem érdemes a belőle kiinduló utakat megvizsgálni, és így azokat „levághatjuk”;*
- *ha az aktuális csúcsból kiinduló minden útról visszaléptünk, azaz egyik sem vezet célba, tehát zsákutca torkolatban állunk;*
- *ha körre futunk, azaz a nyilvántartott út egy csúcsába jutunk el ismét;*
- *ha túl messzire távolodtunk el a kezdőcsúcstól, és az előre megadott mélységi korlátnál hosszabb utakat nem kívánunk foglalkozni*[1]

2. Gráfkereső

A keresés során az összes kiterjesztett és levél csomópont bekerül egy globális adatbázisba. A globális adatbázis lehetőséget nyújt, hogy a visszalépéses keresés nagy hátrányát kiküszöbölje, mégpedig a már felfedezett állapotokat tartalmazó csúcsokat nem vizsgálja és terjeszti ki újra és újra. A kiválasztás során, valamilyen szempont szerint legalkalmasabb csúcsot választja és terjeszti ki. Az új csúcsot esetleg csúcsokat megvizsgálja, hogy szerepelnek-e az adatbázisban. Ha nem akkor újként bekerülnek az adatbázisba. Amennyiben igen, több lehetőség nyílik a helyzet kezelésére. Két fontos kérdés adódik, hogy az adatbázisban lévő állapot az már kiterjesztett, vagy még levél, valamint, hogy az újonnan létrejövő állapot költsége mekkora a régihez képest. Ha levél, a megoldások egyszerűek, ha kisebb az új költsége, akkor a régi állapothoz tartozó költséget és szülő állapotot az újéra módosítjuk, ha nem akkor nem törődünk vele. Amennyiben az adatbázisban már kiterjesztett csomópontként van jelen és az előállítási költség több a meglévő állapotnál, el kell dönteni, hogy milyen helyzetkezelési stratégiát alkalmazunk. Egyik megoldás, hogy egyáltalán nem törődünk vele. Másik alternatíva, hogy kiterjesztett helyett levélként tekintjük, viszont a tőle

származókkal nem foglalkozunk. A harmadik variáció, ha bejárjuk a teljes adatbázist, és azoknál a csúcsoknál, melyeknél az előállítási útban szerepel az adott csúcs, a kellő adatokat aktualizáljuk. A döntés az optimalitás, a futási idő és a tárigény közötti választást eredményezi. Első esetben az optimalitás sérül, mivel nem lesznek helyesek a költség értékek. A másodiknál a futási idő és a tárigény is, mivel azonos állapotokkal többször kell ellenőrizni, kiterjeszteni és tárolni. Az utolsó esetben csak a futási idő nő, azáltal, hogy az ismétlődő csúcsoknál, valamilyen módon be kell járni a függő csúcsokat a keresési gráfban.

c. Nem informált

Mikor az algoritmust mindenfajta, a feladatra jellemző másodlagos információ nélkül készítjük el, azt eredményezi, hogy a keresés vakon, véletlenszerűen fog működni. A másodlagos információ, olyan ismeretek, összefüggések, melyek érvényesek a feladatra, segít a megoldásban, viszont a reprezentációban nem jelennek meg. Azokban az esetben hasznosak, mikor nem ismerünk olyan információt, ami közelebb visz a megoldáshoz, vagy túl költséges az előállítása.

d. Informált

Az előbb említettel ellentétben, a hatékonyságot jelentősen lehet növelni, ha a kereső motorba, minél több, hasznos információt építünk. Gyakori az algoritmusokban használt heurisztika függvény. A függvény egyetlen változója egy állapot, és értékkészlete nemnegatív értékek halmaza. A feladata az, hogy egy állapot jóságát adja meg, függetlenül minden más információtól. A megkötés az, hogy célállapotban nulla értéket kell felvennie, minden más esetben pedig egy nullánál nagyobb pozitív számot. Annál hatékonyabb, minél pontosabban tudja megbecsülni a célállapothoz szükséges átalakítások számát.

Keresőrendszer megvalósítások

A. Hegymászó keresés

Nem módosítható keresés, elnevezését onnan kapta, hogy úgy halad előre, mint egy hegymászó. Mindig a legjobbnak ígérkező állapotot választja kiterjesztésre, és sosem lép vissza. Tárigény szempontjából a leghatékonyabb, mert adatbázisa csak az épen érintett állapotot tárolja, viszont a többi értékelésben rosszak az eredményei. Az optimális megoldására nincs lehetőség, mivel nincsenek információi a keresési útvonal összköltségéről. Teljes csak abban az esetben lehet, ha a kiválasztó eljárás képes a cél felé irányítani a

keresést, úgy hogy nem ragad le egy körben.

B. Backtrack keresés

A visszalépéses stratégia egyik megvalósítása, hatékonyságát az befolyásolja, hogy a korábban említett visszalépési feltételek közül melyeket implementálja a keresőrendszer, valamint milyen szempont alapján választja ki az új állapotok közül azt, amellyel a keresést folytatja.

C. Ág és korlát keresés

A backtrack keresési stratégia egyik változata, mellyel az optimalitását lehet biztosítani. A keresést egy előre beállított korláttal indítjuk, és csak azokat a csúcsokat választjuk kifejtésre, melyek költsége nem haladja meg azt. Valójában, ha a keresési gráf fa, és véges, akkor elhagyható a kezdeti korlát. Amennyiben a problémának van megoldása a kezdeti korláttól kisebb költséggel, akkor képes a legoptimálisabbat megtalálni, mégpedig úgy, hogy ha talál megoldást, akkor folytatja a keresést, miután költségkorlátnak az aktuális megoldás költségét beállította.

D. Iteratívan mélyülő keresés

Az iteratívan mélyülő keresés, nem elsődleges stratégia. Úgy épül be a keresésbe, hogyha egy adott költség vagy mélységkorláttal megadott keresési térben nincs megoldás, a korlát megnövelése után újrakezdi a keresést. Alkalmazható visszalépéses és gráfkereső eljárásnál is. Gráfkereső rendszerben egyik implementációjának a szélességi keresés tekinthető.

E. Szélességi keresés

Gráfkereső stratégia egyik megvalósítása. Az operátorok költsége minden esetben egységnyi, és a stratégia vezérlése mindig a legkisebb költségű csúcsot választja kiterjesztésre. A keresés szintenként halad, és így válik iteratívan mélyülővé. Két ki nem terjesztett csúcs között legfeljebb egy egységköltségnyi különbség lehet. Az algoritmus optimális és teljes, viszont a futási idő és a tárigény nagy. Abban az esetben érdemes használni, ha a megoldásról tudjuk, hogy kevés lépésből megtalálható, vagy nincs semmilyen ismeretünk a megoldáshoz vezető útról.

F. Mélységi keresés

Szintén gráfkereső algoritmus, viszont a szélességi kereséssel ellentétben mindig a legnagyobb költségű csúcsot választja kiterjesztésre. Kezdetben szükséges lehet egy korlátot megadni, amit a kereső nem lép túl, ez a probléma természetétől függ. Kiterjesztésre csak olyan csúcsot választ, amelynek mélysége kevesebb, mint a megadott mélységkorlát. A keresés akkor ér véget, ha megvan a megoldás, vagy már nincs több kiterjesztetlen csúcs. A mélységkorlátot lehet iteratívan használni, így tovább folytatódhat a keresés. Továbbá, ha adott csúcsnál több operátor is alkalmazható, a döntéshez használható heurisztikus információ, a hatékonyság javításához.

G. Optimális keresés

Az operátor költség egyes feladatok reprezentációjában eltérő lehet, más-más operátorok költsége különbözhet. A vezérlő ebben a keresési stratégiában a szélességi kereséshez hasonlóan a legkisebb költségű állapotot választja kiterjesztésre. Amennyiben van megoldás az optimálisat találja meg.

H. Best first keresés

Legjobbat-először algoritmus, amely a csomópont kiválasztáshoz heurisztikus függvényt használ. Előnye, hogy amint észreveszi, hogy egy csúcs nem ígéretes, nem ragad le nála, hanem keres egy jobbat, amivel folytathatja az eljárást. A megoldás sikeressége a heurisztikus függvény helyességén múlik, minél jobb, annál hamarabb találja meg a megoldást. Viszont nem optimális, mivel nem veszi figyelembe az operátorok és a teljes út előállítási költségét. Amennyiben gráfkereső algoritmussal implementáljuk teljes, de idő és tárigénye attól függ, hogy milyen pontosan tudja megbecsülni a cél eléréséhez szükséges műveletek számát.

I. A algoritmus

Az optimális és a best first keresés előnyeit egyesíti, úgy hogy a csomópont kiválasztásához az eddigi útköltség és a heurisztikus függvény által „megjósolt” lépések számának összegével becsüli a teljes útköltséget.

J. Az A* algoritmus

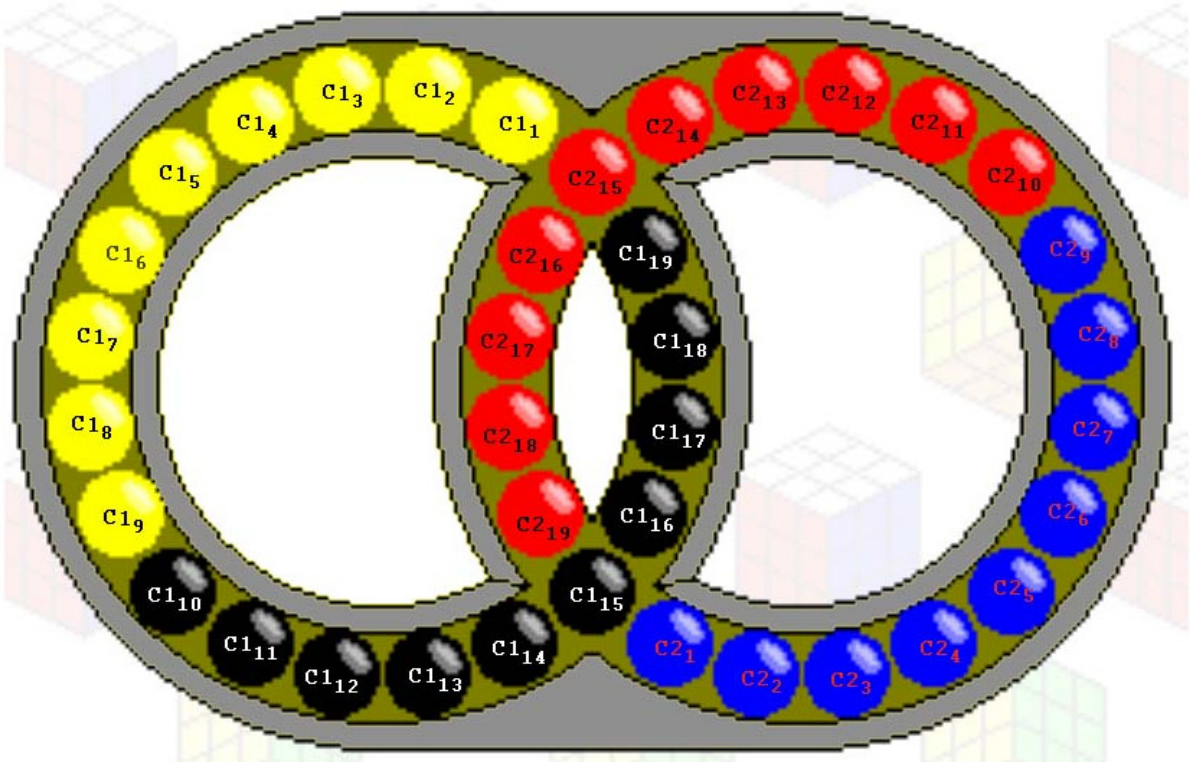
Az A algoritmus optimalitását a heurisztikus függvény minőségével tudjuk biztosítani. Amennyiben a függvény megfelel egy bizonyos megszorításnak, elfogadható heurisztikának

nevezzük. Ez annyit mond ki, hogy a heurisztika alsó becslése a csomóponton keresztül vezető út költségének. Ennek a megszorításnak eleget tevő A algoritmust A* algoritmusnak nevezzük. A keresés hatékonysága attól függ, hogy mennyire helyes az operátorok költsége, valamint minél pontosabb a heurisztika becslése a hátralévő lépések számát illetően. Optimális és teljes, az idő és tárigénye pedig a keresőgráf átmérőjétől és a heurisztika pontosságától függ.

V. Hungarian Rings

a. Állapottér-reprezentáció

A Hungarian Rings egy egyszerű játék, ahol két körben kell különböző színű golyókat mozgatni. A két kör két pontban érintkezik, ahol át lehet csúsztatni a golyókat. Harmincnyolc golyó van összesen és négy szín. Mindkét körben lehet mozgatni a golyókat jobbra és balra is. A játék célja, hogy mindkét körben legyen azonos színből kilenc és másik azonos színből tíz golyó megszakítás nélkül.



A játék és ebből kifolyólag az állapottér is paraméterezhető a körök, golyók és színek száma szerint, valamint, hogy hány golyó mélységben metszik egymást a körök.

Az eredeti játéknak az A állapottérrel, a *k* kezdőállapottal, a C célállapotok halmazával, az operátorok alkalmazási előfeltételével valamint hatásával megadjuk a problémánk egy lehetséges
p=< A , *k*, C, O> állapottér reprezentációját.

Állapottér:

A probléma lényeges jellemzője, hogy milyen színű golyó hol helyezkedik el.
Fontos, hogy míg egy körön belül ugyan húsz golyó van, de a játék értelmében tizenkilenc tartozik össze, mégpedig úgy, hogy a baloldali kör első golyója jobb fent, a jobboldalié bal lent kezdődik, és a tizenötödik golyóik szakítják meg a másik kör folytonosságát.

Alaphalmaz:

H = {1, 2, 3, 4}:
Minden egyes szám egy az összes többitől eltérő színt jelöl.

Egy állapot:

$h = \{C1_1, \ldots, C1_{19}, C2_1, \ldots, C2_{19}\} \subseteq H^{38.}$

Mivel a H^{38} halmaz nem minden eleme állapot, ezért szükséges az alábbi kényszerfeltétel megfogalmazása:

$$\text{egyezo}(x, y) = \begin{cases} 1, \text{ha } x = y \\ 0 \text{ egyébként} \end{cases}$$

kényszerfeltétel(h) =

$$\sum_{i=1}^{19} \text{egyezo}(C1_i, 1) + \sum_{i=1}^{19} \text{egyezo}(C2_i, 1) = 9 \wedge \sum_{i=1}^{19} \text{egyezo}(C1_i, 2) + \sum_{i=1}^{19} \text{egyezo}(C2_i, 2) = 10 \wedge$$
$$\sum_{i=1}^{19} \text{egyezo}(C1_i, 3) + \sum_{i=1}^{19} \text{egyezo}(C2_i, 3) = 9 \wedge \sum_{i=1}^{19} \text{egyezo}(C1_i, 4) + \sum_{i=1}^{19} \text{egyezo}(C2_i, 4) = 10$$

$A = \{h = \{C1_1, \ldots, C1_{19}, C2_1, \ldots, C2_{19}\} | \text{ kényszerfeltétel}(h)\} \subseteq H^{38}$.

Kezdőállapot:

Tetszőleges kezdőállapottal elindított szélességi keresés során a kereső képes az összes különböző állapotot előállítani, és mivel az operátor önmaga inverze csak a forgatások számában különbözik, így kezdőállapotnak bármely olyan h állapotot választhatunk, amely eleget tesz a kényszerfeltételnek.

$k = \{h \mid \text{kényszerfeltétel } (h)\}$.

Célállapotok halmaza:

$C = \{h \in \{ C1_i = s_1 \quad (i=1...9)$

$C1_i = s_2 \quad (i=10...19)$

$C2_i = s_3 \quad (i=1...9)$

$C2_i = s_4 \quad (i=10\ldots19)$,

$C1_i = s_1 \quad (i=1...10)$

$C1_i = s_2 \quad (i=11...19)$

$C2_i = s_3 \quad (i=1...10)$

$C2_i = s_4 \quad (i=11\ldots19)$,

$C1_i = s_1 \quad (i=1...10)$

$C1_i = s_2 \quad (i=11...19)$

$C2_i = s_3 \quad (i=1...9)$

$C2_i = s_4 \quad (i=10\ldots19)$,

$C1_i = s_1 \quad (i=1...9)$

$C1_i = s_2 \quad (i=10...19)$

$C2_i = s_3 \quad (i=1...10)$

$C2_i = s_4 \quad (i=11\ldots19)\}$

$\{s_1, s_2, s_3, s_4\} = \{1, 2, 3, 4\}$, s_1, s_2, s_3, s_4 legyen az 1, 2, 3, 4 egy permutációja

Operátorok halmaza

O = {C1_m, C1_e, C2_ m, C2_e}

Előfeltételre nincs szükség, hisz minden operátor minden valódi állapotból, állapotba visz át.

Operátor hatása:

A C1_m operátor a h ={$C1_1$, …, $C1_{19}$, $C2_1$, …, $C2_{19}$} ∈ A állapotra alkalmazva a következőképpen definiált h' ={$C1'_1$, …, $C1'_{19}$, $C2'_1$, …, $C2'_{19}$} ∈ A állapotot adja:

$$C1'_j = \begin{cases} C1_{j+1} & \text{ha } j \neq 19 \\ C2_{15} & \text{ha } j = 19 \end{cases}$$

$$C2'_j = \begin{cases} C1_1 & \text{ha } j = 15 \\ C2_j & \text{egyébként} \end{cases}$$

A C1_e a h ={$C1_1$, …, $C1_{19}$, $C2_1$, …, $C2_{19}$} ∈ A állapotra alkalmazva a következőképpen definiált h' ={$C1'_1$, …, $C1'_{19}$, $C2'_1$, …, $C2'_{19}$} ∈ A állapotot adja:

$$C1'_j = \begin{cases} C1_{j-1} & \text{ha } j \neq 1 \\ C2_{15} & \text{ha } j = 1 \end{cases}$$

$$C2'_j = \begin{cases} C1_{19} & \text{ha } j = 15 \\ C2_j & \text{egyébként} \end{cases}$$

A C2_m operátor a h ={$C1_1$, …, $C1_{19}$, $C2_1$, …, $C2_{19}$} ∈ A állapotra alkalmazva a következőképpen definiált h' ={$C1'_1$, …, $C1'_{19}$, $C2'_1$, …, $C2'_{19}$} ∈ A állapotot adja:

$$C1'_j = \begin{cases} C2_1 & \text{ha } j = 15 \\ C1_j & \text{egyébként} \end{cases}$$

$$C2'_j = \begin{cases} C2_{j+1} & \text{ha } j \neq 19 \\ C1_{15} & \text{ha } j = 19 \end{cases}$$

A C2_e operátor a h ={$C1_1$, …, $C1_{19}$, $C2_1$, …, $C2_{19}$} ∈ A állapotra alkalmazva a következőképpen definiált h' ={$C1'_1$, …, $C1'_{19}$, $C2'_1$, …, $C2'_{19}$} ∈ A állapotot adja:

$$C1'_j = \begin{cases} C2_{19} & \text{ha } j = 15 \\ C1_j & \text{egyébként} \end{cases}$$

$$C2'_j = \begin{cases} C2_{j-1} & \text{ha } j \neq 1 \\ C1_{15} & \text{ha } j = 1 \end{cases}$$

b. Paraméterezett állapottér

A játék a legtöbb logikai játékhoz hasonlóan, tovább bővíthető a méretei által, amivel a lehetőségek száma és a megoldásához szükséges idő is nő. Az állapottérben az eredeti játék is modellezhető, de nagyobb és kisebb problémák megkonstruálására is van lehetőség. Ugyan a következőkben megadott értékek is tovább növelhetők elviekben, viszont az elkészített program ezekre van felkészítve, ezekre működik biztonságosan. A következő sorokban megadom a paraméterek által felvehető értékeket és jelölésüket, bizonyos feltételeket betartva.

A körök száma: Ksz $\in \{2, 3, 5\}$.

A golyók száma körönként: Gsz $\in \{8, 10, 12, 14, 16, 18, 20\}$ ha a körök száma 2,

Gsz$\in \{10, 12, 14, 16, 18, 20\}$ ha a körök száma 3, vagy a golyók száma 18 ha a körök száma 5.

A színek száma körönként: Szsz $\in \{1, 2\}$.

Belső golyónak értelmezem azokat a golyókat, melyek átlógnak a szomszédos körbe, de nem szerepelnek benne. Számuk szerint:

Bg = 2, ha körök száma 2 és a golyók száma 8, vagy a körök száma 3 és a golyók száma 10, vagy a körök száma 3 és a golyók száma 12 vagy a körök száma 5 és a golyók száma 18.

Bg $\in \{2, 3\}$ ha a körök száma 2 és a golyók száma 10, vagy a körök száma 3 és a golyók száma 14.

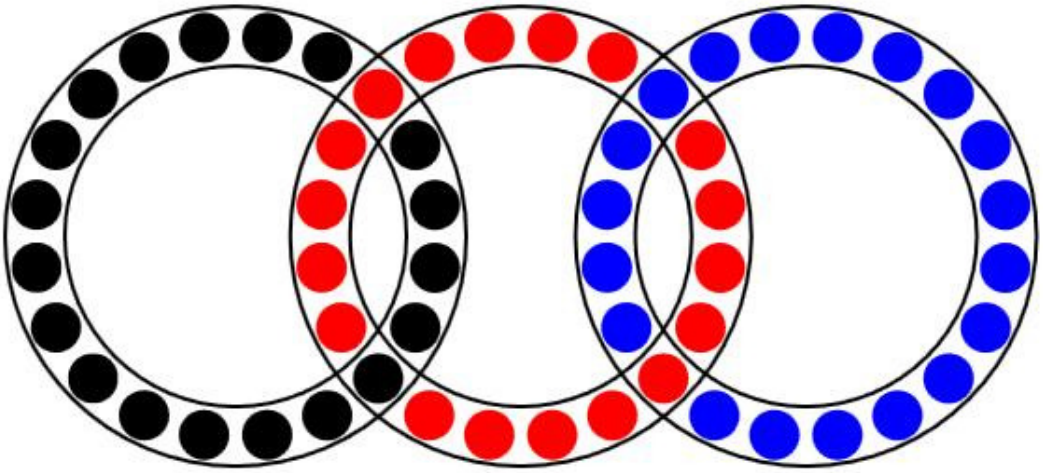
Bg $\in \{2, 3, 4\}$ ha a körök száma 2 és a golyók száma 12, vagy a körök száma 3 és golyók száma a 16, vagy a körök száma 3 és a golyók száma 18.

Bg $\in \{2, 3, 4, 5\}$ ha a körök száma 2 és a golyók száma 14, vagy a körök száma 3 és a golyók száma 20.

Bg $\in \{2, 3, 4, 5, 6\}$ ha a körök száma 2 és a golyók száma 16.

Bg $\in \{2, 3, 4, 5, 6, 7\}$ ha a körök száma 2 és a golyók száma 18.

Bg $\in \{2, 3, 4, 5, 6, 7, 8\}$ ha a körök száma 2 és a golyók száma 20.



1. beállítás: 3 kör és 20 golyó, valamint a belső golyók száma 4 és 1 szín körönként.

További mutatószámai a játéknak az első, a második és a harmadik metszéspont, valamint a belső köröknél értelmezett külső golyók száma. Ezek a „mágikus” számok azokat a golyóindexeket jelölik, amik több körben szerepelnek, vagy amik éppen kilógnak. Számítási módjuk aszerint is változik, hogy hány kör van, és szélső-e vagy belső a kör, mivel másképpen metszik egymás. Jelölésük és értékeik:

Külső golyók száma (belső köröknél): k = Gsz / 2 – Bg – 2.

Első metszéspont: e = k + 1.

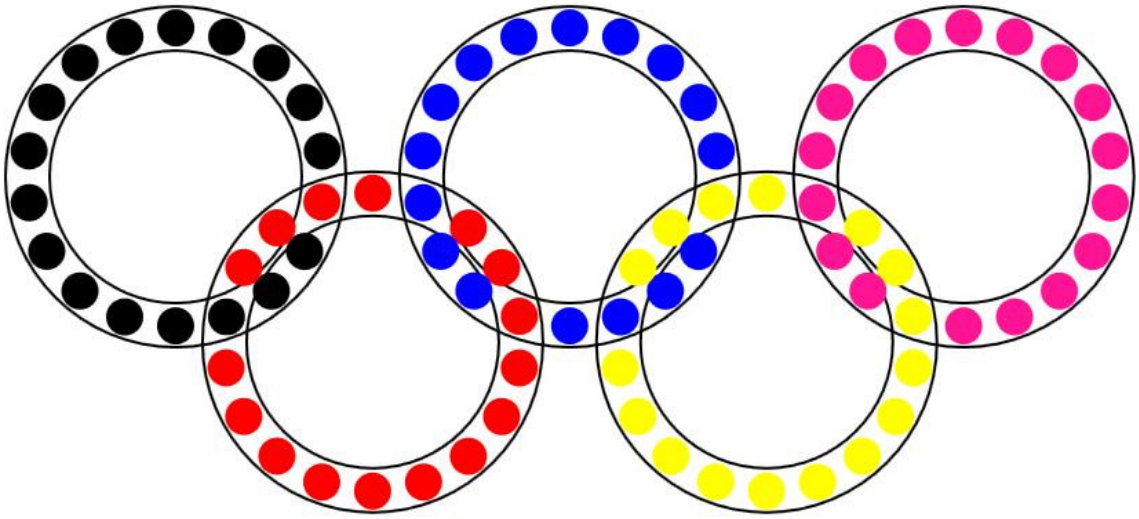
Második metszéspont: m = e + Bg + 1. Megfigyelhető, hogy m egyenlő Gsz / 2-vel.

Harmadik metszéspont: h = Gsz – Bg – 2.

Az első és utolsó körnél az első és egyetlen valódi metszését a h + 1 jelöli.

Továbbá belső köröknél nem csak egy „szakadás” van, hanem kettő, nevezetesen az „első és utolsó” golyók közt és az m-1 és az m indexűek közt.

Az első beállításnál a külső golyók száma négy, az első metszéspontnál az ötödik golyó szerepel, míg a második metszéspont a tízedik golyó helye. A harmadik metszéspontnál a tizennegyedik golyó van. A szélső körökben metszéspont csak a tizenötödik golyónál van.



2. beállítás: 5 kör valamint 18 golyó és 1 szín körönként.

Ez, a játék egy speciális változata, miszerint a körök az olimpiai ötkarikát modellezik. Ezzel a golyószámmal a többi beállítás másképp nem módosítható, és a mutatószámok is előre rögzítettek. A két szakadás megegyezik a három körösben leírtakkal. Ebben az esetben két szám jelöli a külső golyók számát, mégpedig k1 egy és k2 kilenc, továbbá e kettő, m öt és h tizennégy. Itt is a szélső körök egyetlen metszéspontja a h + 1.

A három és öt körös beállításnál is a belső körökben, a játék értelmében kettővel kevesebb golyó van, hisz minden kapcsolódó kör miatt eggyel kevesebb tartozik hozzá. Formálisan úgy lehetne leírni a golyók valódi számát, hogy ahány van egy körben mínusz az adott körhöz kapcsolódó körök száma.

Alaphalmaz:

$H = \{1,\ldots, Ksz*Szsz\}$:

Egy állapot:

$h = \{C1_1, \ldots, C1_{Gsz-1}, CKsz_1, \ldots, CKsz_{Gsz-1}\} \in \{H \times H \times \ldots \times H\}$

Mivel a $H^{Ksz*(Gsz-1)}$ halmaz nem minden eleme állapot, ezért szükséges az alábbi kényszerfeltétel megfogalmazása:

$$\text{egyezo}(x,y)=\begin{cases}1, \text{ha } x=y\\ 0 \text{ egyébként}\end{cases}$$

<u>kényszerfeltétel(h)</u> =

$$(Szsz=1\wedge((\forall j\forall k\forall l(j\in\{1\ldots Ksz\}/\{1,Ksz\}\wedge k\in\{1\ldots Ksz\}/\{1,Ksz\}\wedge l\in\{1,Ksz\}\wedge$$
$$\sum_{k}\sum_{i=1}^{Gsz-2}\text{egyezo}(Ck_i,j)+\sum_{l}\sum_{i=1}^{Gsz-1}\text{egyezo}(Cl_i,j)=Gsz-2))\vee$$
$$(\forall j\forall k\forall l(j\in\{1,Ksz\}\wedge k\in\{1\ldots Ksz\}/\{1,Ksz\}\wedge l\in\{1,Ksz\}\wedge$$
$$\sum_{k}\sum_{i=1}^{Gsz-2}\text{egyezo}(Ck_i,j)+\sum_{l}\sum_{i=1}^{Gsz-1}\text{egyezo}(Cl_i,j)=Gsz-1)))\vee$$
$$(Szsz=2\wedge((\forall j\forall k\forall(j\in\{1\ldots Ksz\times 2\}/\{1,Ksz*2\}\wedge k\in\{1\ldots Ksz\}/\{1,Ksz\}\wedge l\in\{1,Ksz\}\wedge$$
$$\sum_{k}\sum_{i=1}^{Gsz-2}\text{egyezo}(Ck_i,j)+\sum_{l}\sum_{i=1}^{Gsz-1}\text{egyezo}(Cl_i,j)=Gsz/2-1))\vee$$
$$(\forall j(j\in\{1,Ksz*Szsz\}\wedge k\in\{1\ldots Ksz\}/\{1,Ksz\}\wedge l\in\{1,Ksz\}\wedge$$
$$\sum_{k}\sum_{i=1}^{Gsz-2}\text{egyezo}(Ck_i,j)+\sum_{l}\sum_{i=1}^{Gsz-1}\text{egyezo}(Cl_i,j)=Gsz/2)))$$

$A=\{h=\{C1_1,\ldots, C1_{Gsz-1}, CKsz_1,..., CKsz_{Gsz-1}\}|$ kényszerfeltétel(h)$\}\subseteq H^{Ksz*(Gsz-1)}$.

Kezdőállapot:

A kétkörös változatban leírtak itt is érvényesek, miszerint kezdőállapotnak bármely olyan h állapotot választhatunk, amely eleget tesz a kényszerfeltételnek.

$k = \{h \mid$ kényszerfeltétel (h)$\}$.

Célállapotok halmaza:

$$S(x,y)=\begin{cases}0, & \text{ha } x=y\\ 1, & \text{ha } x!=y\end{cases}$$

$C = \{ h=\{ C1_1, \ldots, C1_{Gsz}, CKsz_1 ,..., CKsz_{Gsz} \}|$

$$(\forall i(i\in\{1,\ldots,Ksz\}/\{1,Ksz\}\wedge \sum_{j=1}^{Gsz-2}S(Ci_j,Ci_{j+1}) = Szsz\text{ - }1)\wedge$$
$$(\forall k(k\in\{1,Ksz\}\wedge \sum_{l=1}^{Gsz-1}S(Ck_1,Ck_{l+1}) = Szsz\text{ - }1))\}\subset A$$

Operátorok halmaza

O = {forgat(*k, f*)}

Egy operátorcsalád van, melyben az operátorok csak paramétereik értékeiben különböznek. Az első paraméter *k* (*k*∈ {1,..., Ksz}) a forgatott kör indexe, míg a második *f* (*f*∈ {1,…,Gsz}), a forgatások számát adja meg. A paraméterektől függően akármelyik kört képesek megforgatni az óramutató járásával megegyező irányban, legkevesebb eggyel, legfeljebb a golyók számával.

Előfeltételre nincs szükség, hisz az operátor minden állapotból, valódi állapotba visz át.

Operátor hatása:

A forgat operátor a h ={ $C1_1$, …, $C1_{Gsz-1}$,…, $CKsz_1$, …, $CKsz_{Gsz-1}$ }∈ A állapotra alkalmazva a következőképpen definiált h' = { $C1_1$', …, $C1_{Gsz-1}$', …,$CKsz_1$', …, $CKsz_{Gsz-1}$' }∈ A állapotot adja:

$$\forall i \forall j((i \in \{1 \ldots Ksz\}/\{1, Ksz\} \wedge j \in \{1 \ldots Gsz-2\}) \vee (i \in \{1, Ksz\} \wedge j \in \{1 \ldots Gsz-1\}))$$

$$h'_{i,j} = \begin{cases} h_{i+1,m-f-1}, & \text{ha } i = k-1 \wedge f < m \wedge ((i = 1 \wedge j = h+1 \wedge \\ & \quad (ksz = 5 \vee ksz = 3)) \vee (ksz = 5 \wedge i = 3 \wedge j = e)) \\ h_{i+2,h}, & \text{ha } i = k-1 \wedge ksz = 5 \wedge i = 1 \wedge j = h+1 \wedge f = m \\ h_{i+1,gsz+m-f-1}, & \text{ha } i = k-1 \wedge f > m \wedge ((ksz = 5 \wedge i = 3 \wedge j = e) \vee \\ & \quad (ksz \in \{3,5\} \wedge i = 1 \wedge j = h+1)) \\ h_{i+1,gsz-f-1}, & \text{ha } i = k-1 \wedge ksz = 5 \wedge i = 2 \wedge j = h \wedge f < k2+b+2 \\ h_{i+2,e}, & \text{ha } i = k-1 \wedge ksz = 5 \wedge i = 2 \wedge j = h \wedge f = k2+b+2 \\ h_{i+1,gsz-f}, & \text{ha } i = k-1 \wedge j = h \wedge ((ksz = 5 \wedge i = 4) \vee \\ & \quad (ksz = 2 \wedge i = 1) \vee (ksz = 3 \wedge i = 2) \vee \\ & \quad (i = 5 \wedge i = 2 \wedge f > k2+b+2)) \\ h_{i+2,h+1}, & \text{ha } i = k-1 \wedge f = m \wedge ((ksz = 5 \wedge i = 3 \wedge j = e) \vee \\ & \quad (ksz = 3 \wedge i = 1 \wedge j = h+1)) \\ h_{i-1,gsz-f}, & \text{ha } i = k+1 \wedge ((f > k2+b+2 \wedge (((ksz = 3 \vee ksz = 5) \wedge \\ & \quad i = ksz \wedge j = h+1) \vee (ksz = 5 \wedge i = 3 \wedge j = h))) \vee \\ & \quad (ksz \in \{2,3,5\} \wedge i = 2 \wedge j = e)) \end{cases}$$

$$
h'_{i,j} = \begin{cases}
h_{i-1,gsz-f-1}, & ha\, i = k+1 \wedge f < k2+b+2 \wedge ((((ksz=3 \vee ksz=5) \wedge \\
& \quad i = ksz-1) \wedge j=h+1) \vee (ksz=5 \wedge i=3 \wedge j=h))) \\
h_{i-2,e}, & ha\, i = k+1 \wedge ksz=5 \wedge i=5 \wedge j=h+1 \wedge f=k2+b+2 \\
h_{i-1,m-f-1}, & ha\, i = k+1 \wedge ksz=5 \wedge i=4 \wedge j=e \wedge f<m \\
h_{i-2,h}, & ha\, i = k+1 \wedge ksz=5 \wedge i=4 \wedge j=e \wedge f=m \\
h_{i-1,gsz+m-f-1}, & ha\, i = k+1 \wedge ksz=5 \wedge i=4 \wedge j=e \wedge f>m \\
h_{i-2,h+1}, & ha\, i = k+1 \wedge i=3 \wedge f=k2+b+2 \wedge ((ksz=3 \wedge \\
& \quad j=h+1) \vee (ksz=5 \wedge j=h)) \\
h_{i-1,h}, & ha\, i = k \wedge ((i=ksz \wedge j=f) \vee (ksz=5 \wedge i=3 \wedge \\
& \quad ((f<m \wedge j=f) \vee (f>m \wedge j=f-1)))) \\
h_{i-1,e}, & ha\, i = k \wedge ksz=5 \wedge i=4 \wedge \\
& \quad ((m+f-1<gsz \wedge j=m+f-1) \vee \\
& \quad (m+f-1>gsz \wedge j=f-k2-m+1)) \\
h_{i+1,h+1}, & ha\, i = k \wedge (((ksz=5 \wedge i=4) \vee (ksz=3 \wedge i=2 \wedge)) \wedge \\
& \quad ((f<m \wedge j=f) \vee (f>m \wedge j=f-1))) \\
h_{i+1,e}, & ha\, i = k \wedge ((i=1 \wedge j=f) \vee (ksz=5 \wedge \\
& \quad i=3 \wedge ((m+f-1<gsz \wedge j=m+f-1) \vee \\
& \quad (m+f-1>gsz \wedge j=f-k2-m+1))) \\
h_{i,gsz-f+j+1}, & ha\, i = k \wedge (((ksz=5 \wedge i \in \{2,3,4\} \wedge ((j+1<f \wedge j>m-1) \vee \\
& \quad (f>h \wedge j<f-k2-m+1))) \vee (ksz=3 \wedge i=2 \wedge \\
& \quad ((f>j+1 \wedge j>m-1) \vee (f>m+1 \wedge j<f-m)))) \vee \\
& \quad ((i=ksz \vee i=1) \wedge ksz \in \{2,3,5\} \wedge j<f)) \\
h_{i,gsz-f+j}, & ha\, i = k \wedge ((ksz=5 \wedge i \in \{2,3,4\} \wedge j>f-h+1 \wedge \\
& \quad f>j \wedge j<=m-1) \vee (ksz=3 \wedge i=2 \wedge \\
& \quad f>1 \wedge j<m \wedge j>f-m \wedge j<f)) \\
h_{i,j-f+1}, & ha\, i = k \wedge (f>1 \wedge j>m-1 \wedge ((ksz=5 \wedge \\
& \quad i \in \{2,3,4\} \wedge j<m+f-1) \vee (ksz=3 \wedge \\
& \quad i=2 \wedge j<gsz-m+f)))
\end{cases}
$$

$$h'_{i,j} = \begin{cases} h_{i,j-f}, & \text{ha } i = k \wedge ((((ksz = 5 \wedge i \in \{2,3,4\}) \\ & \quad \vee (ksz = 3 \wedge i = 2)) \wedge ((f < k2 + b + 1 \wedge \\ & \quad j > f + m - 1) \vee (j > f \wedge f < m - 1 \wedge j < m))) \vee \\ & \quad ((i = ksz \vee i = 1) \wedge ksz \in \{2,3,5\} \wedge j > f)) \\ h_{i-1,h+1}, & \text{ha } i = k \wedge i = 2 \wedge (((ksz = 5 \vee ksz = 3) \wedge m + f - 1 < gsz \wedge \\ & \quad j = m + f - 1) \vee (m + f - 1 > gsz \wedge ((ksz = 5 \wedge \\ & \quad j = f - k2 - m + 1) \vee (ksz = 3 \wedge j = m + f - 1))))) \\ h_{i+1,h}, & \text{ha } i = k \wedge ksz = 5 \wedge i = 2 \wedge ((f < m \wedge j = f) \vee (f > m \wedge j = f - 1)) \\ h_{i,j}, & \text{egyébként} \end{cases}$$

c. Megvalósítás

A programot az objektumorientált C# nyelven készítettem el. Külön osztállyal valósítom meg az állapotot, az operátort, a csúcsot és a keresőt. Az operátor osztálya a `ForgatoOperator` nevet viseli és az Operator.cs nevű fájl tartalmazza. Összesen két mezője van, az egyik azt az információt tárolja, hogy melyik kört forgatja, míg a másik azt, hogy mennyivel. Az Allapot.cs fájlban implementálom a `HungarianRingsAllapot` osztályt. Összesen tizenegy statikus mezőt tartalmaz, ezek közül egyik az operátor különböző értékekkel példányosított listája, egy a heurisztika meghatározására szolgáló egész érték, melynek kiszámolási módját később fejtem ki, és további kilenc, amelyek a játékra vonatkozó paraméterek és mutatószámok. Tartalmaz még egy, egész számok tárolására alkalmas kétdimenziós tömböt példányváltozóként. Az első dimenzió a körre vonatkozik a második pedig, a körön belüli golyó indexére. Minden színt egy pozitív egész szám jelöl. Az első színt az egy, az utolsót a körök és színek számának a szorzatának az eredménye. A program során az indexelést nullától kezdve készítettem, így a program sokszor az adott értékektől eggyel kisebb számmal dolgozik. Az állapotot három módon lehet példányosítani. Egyik lehetőség rá, hogy a konstruktorban csak az állapottér paramétereit adjuk meg; másik mód, hogy a paramétereken kívül egy kezdő tömbbel megadjuk a kezdő állapotot. Ez utóbbi megoldás magában rejti azt a veszélyt, hogy nem valódi állapotnak megfelelő tömböt adunk meg, viszont mivel ez a konstruktor úgy kerül meghívásra, hogy a felhasználói felületen valódi állapotból operátor sorozattal kialakított állapotnak a tömbjét adjuk meg, így nem vezet ki a valódi állapotok halmazából. A harmadik konstruktor paraméter listája üres, mivel annak annyi a faladata, hogy egy alkalmas méretű mátrixot példányosítson. Az első két konstruktor

használ egy közös metódust, `init` néven, ami a statikus mezők beállítást végzik. Az egyetlen mező, amit kiemelnék az a `maxH`, ami az adott paraméterekkel rendelkező állapottéren felvehető maximális heurisztika értéket tartalmazza. A többi mezőre az állapottér reprezentációnál megadott számolási módszert használja. Amennyiben a kezdő állapotot nem adjuk meg kezdéskor, úgy meghívódik a `kezdo` metódus. Feladata, a mátrix feltöltése, ami megfelel egy valódi állapotnak. Ezt úgy érem el, hogy kezdetben a felvehető színek számosságú tömböt feltöltök, mégpedig úgy, hogy az adott indexű elem, az adott színből tartalmazható golyók számának az értéke legyen. Körönként egy szín esetén a szélső köröknek megfelelő indexű tömbelemben a golyók számától eggyel kevesebb, belső golyóknál kettővel, míg két szín esetén, a belső körök indexeinél és a külső körök kisebbik indexű tömbeleménél a golyók számának felétől eggyel kevesebb, valamint a szélső körök nagyobb számú indexeinél a golyók számának felével megegyező értéket tartalmazzon. Miután ezzel végez, minden egyes elemét a mátrixnak véletlenszerűen feltölti valamelyik színnel, azok közül, melynek értéke nagyobb nullától az előbb feltöltött tömb adott indexű eleménél, majd ezt az értéket csökkenti eggyel. A célállapotot vizsgáló metódus az állapottér reprezentációban megadott célfeltétel pontos megvalósítása. Körönként ellenőrzi, hányszor fordul elő, hogy egymás mellett nem azonos szín szerepel. Mivel ha egy szín van, akkor ennek az értéknek nullának kell lenni célállapotban, míg két szín esetén egynek. Ha valamelyik körben nem így van, akkor az eredmény hamis, különben igaz. Az `Alkalmaz` metódus szintén az állapottér reprezentációban leírt operátor hatásának az implementációja. Itt négy lényegesen különböző feladat adódik. Az egyik, az az eset, mikor azt a kört módosítjuk, amelyik a forgatott kör előtt van, második eset, mikor mögötte, harmadikban, a módosuló kör éppen a forgatott, negyedikben pedig egyik sem. A nehézség az, hogy a játékot negyvenháromféle képpen lehet implementálni, ha a színek számát nem vesszük figyelembe, például két kör, nyolc golyó és belső golyók száma kettő, vagy három kör tizenhat golyó és a belső golyók száma három, és így tovább. Két nehéz lehetőség közül tehát azt választottam, miszerint egy operátor van, és abban logikai feltételekkel adom meg, hogy tetszőleges paraméterű állapottérben tetszőleges forgatással mi történjen, nem pedig minden egyes elfogadható variációjú állapottérben, az összes lehetséges operátort implementáljam. Az eredményeknek a két stílusban kiszámolva, helyesen elkészített algoritmusok mellett meg kell egyeznie, viszont ebben több programozói kihívást láttam. Első lépésben létrehozok egy üres állapotot, aztán a mátrix elemein egyesével a feltételek szerint feltöltöm értékekkel. Két állapot azonosságának ellenőrzését a `bool` értéket visszaadó `Equals` metódus végzi. Két

állapotot akkor tekintek egyezőnek, ha az állapotok mátrixainak minden indexpár mellett megegyezik az értékük. A szakirodalom szerint a heurisztika függvény nem az állapottér része, hanem a keresőé, viszont mivel a metódus ebben az osztályban van elkészítve, itt adom meg a leírását. Egy állapotot akkor tekintek „jónak”, a megoldás szerint értékesnek, ha abból a színből, ami az adott körhöz tartozik, minél több van egymás mellett. Ezt úgy érem el, hogy az egymás mellett szereplő azonos színű golyók darabszámának négyzetének az összegét tekintem. Mivel ez pont ellentétben van az általános heurisztikus felfogásnak, miszerint a jobb állapotokon kisebb értéket vesz fel a függvény, azt egy egész számból vonom ki. Azt hogy mennyiből, azt az elején határozom meg az `init` nevű metódussal, a `maxH`-ban. Az érték az előbb leírt négyzetösszegek maximuma lehet. Aszerint, hogy hány kör, golyó és szín van egy egyszerű képlettel számolom ki az értékét. Így mikor az összes golyó jó helyen van, akkor lesz a heurisztikájának nulla az értéke. Több nehézség van viszont a kivitelezésben. Implementációs szinten a golyók elhelyezkedését mátrixban tároljuk, és a kezeléséhez magamnak kell a körkörös viselkedést modellezni. Nem csak akkor tekinthető egymás mellettinek két golyó, ha indexeik közti különbség egy, hanem ha a szomszédos kör metszéspontjában megfelelő színű a golyó. Így egy végén kezdődő és másik körön átívelő, aztán esetleg újra a vizsgált körben végződő színsor is egymás mellettinek tekintendő, hisz egy forgatással később az akár szabályosan is a körbe rendezhető. Tehát első lépésként meg kell keresni az első olyan indexet, ahol az egymás mellett szereplők különböző színűek, és onnantól kezdve végigszámlálni a négyzetösszegeket. Más a helyzet a közbenső köröknél, mivel ott, a golyók számának a felétől eggyel kevesebb van egymás mellett, aztán egy kihagyás, végül újra mégannyi. Itt a körbeszámlálás egy összetett algoritmus, ami a leghosszabb sorozatnál kezdődik. Mindkettő módszernél megegyezik viszont az, hogy csak azokat a golyókat figyeli, ami az adott körhöz érték szerint hozzátartozik. A metszésekben szereplő golyók pontos helyével nem foglalkozik az algoritmus, mivel az elején készül egy segéd mátrix, amiben összeállítja a mátrixot, ahogyan az a játék szempontjából tűnik, és azon végzi el a vizsgálatokat.

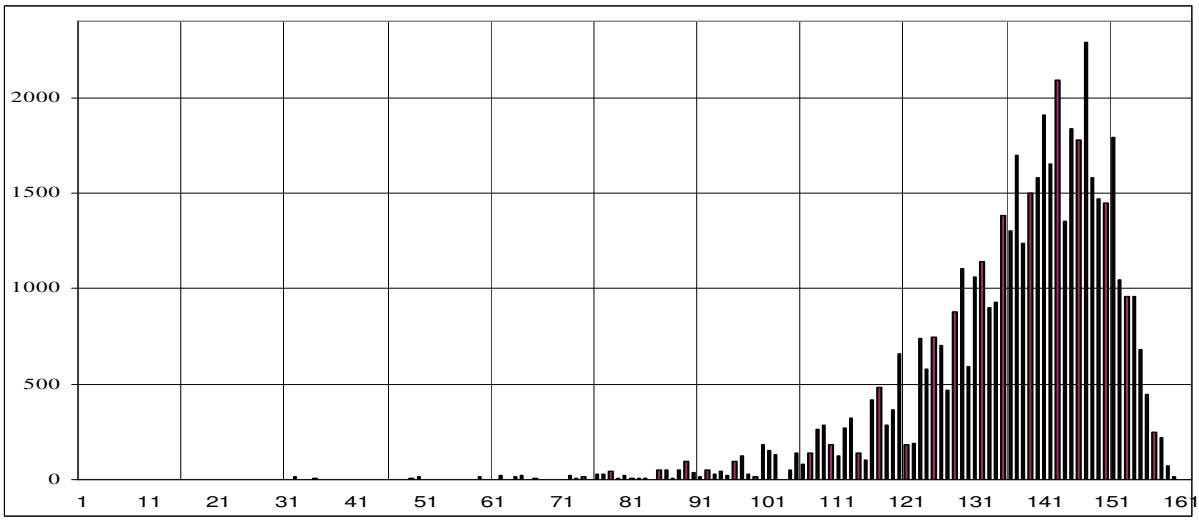
d. Alkalmazott keresési stratégia

A kereséshez módosított Best first algoritmust használtam, amivel az adott problémához optimalizáltam a keresést. A kereséshez szükséges osztályok a Kereso.cs és Csucs.cs-ben vannak. A `Csucs` osztály öt mezője tartalmazza az egy csúcshoz tartozó információkat, úgymint a `HungarianRingsAllapot` állapotot, az állapot szülő csúcsának a referenciáját, egy `ForgatoOperator` típusú operátort, ami a szülő csúcsra alkalmazva előállítja a csúcshoz tartozó állapotot, a csúcs mélységét és a heurisztikáját. Két konstruktora van, az egyik egyetlen `HungarianRingsAllapot` típusú objektumot vár, ami a kezdő csúcsot állítja elő, a másik, ez előbb említett típusún kívül vár még egy `ForgatoOperator` típusú operátort is. Az utóbbi konstruktor első paramétere a szülő állapot, az operátor a rajta alkalmazott operátor, az új állapot a két paraméter által előálló állapot és a mélysége, a szülő csúcsétól eggyel több. Az `Equals` metódus akkor tekint két csúcsot egyezőnek, ha a két csúcsban szereplő állapot megegyezik. A keresési gráf fává alakítása ebben az osztályban megvalósítható lenne. Szükség lenne arra a változtatásra, hogy a csúcs tartalmazza a belőle előálló állapotok referenciáját. Amennyiben a keresés során az `Equals` metódus egyenlőnek találna két csúcsot, megvizsgálná, hogy az új csúcs mélysége kevesebb-e, mint a régié, és ha igen az utód referenciákon végigjárva, beállítaná a pontos értékeket. A valódi keresés a `BestFirstKereso` osztályban van implementálva. Két egész értékű mezője a kiterjesztett és a levél csúcsainak a számát tárolja. Az előbbieket zárt, míg az utóbbiakat nyílt csúcsoknak nevezi a szakirodalom. Egy mezője a célállapot csúcsot hivatott tartalmazni, ha a keresés sikeresen befejeződik, egyébként a referenciája `null` marad, így később a grafikus felület algoritmusa innen azonosítja, hogy a keresést a felhasználó megszakította. Az általános Best first kereső, miután kiválaszt egy csúcsot, és megbizonyosodik, hogy nem célállapot, kiterjeszti, végigvizsgálja a teljes nyílt csúcsok adatbázisát, ha nincs egyforma, folytatja a teljes zártak adatbázisával, amennyiben itt sincs, beszúrja az új állapotokat a levél csomópontokhoz és végül az egészet átrendezi heurisztika szerint növekvő sorba. Eztán újra a legkisebb heurisztikájú állapotot tartalmazó csúcsot kiválasztja a nyílt csúcsok listájának az első helyéről, ami a legígéretesebbnek tűnik, hogy azzal előröl kezdje az algoritmust. Az eredeti algoritmuson abban változtattam, hogy figyelembe veszem és kihasználom a heurisztikák előre kiszámolható, diszkrét értékeiből adódó lehetőségeket, ötvözve a nyelv adottságaival. A keresés során nem egy-egy listát tárolok a nyílt és zárt csúcsok adatbázisára,

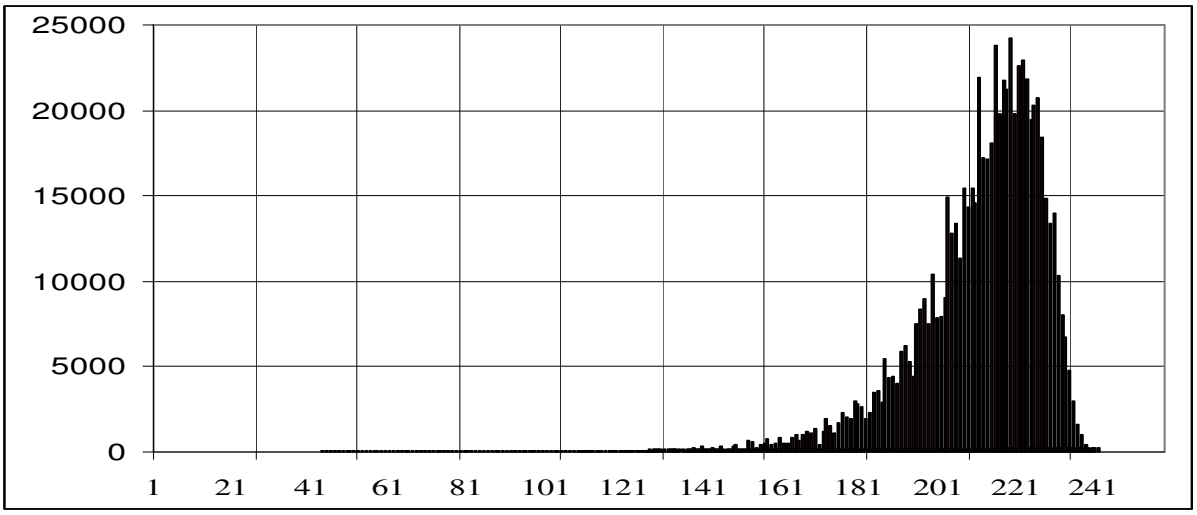
hanem mindkettőből pont annyit, amennyi értéket a heurisztikafüggvény fel tud venni az állapottér állapotain. Mikor egy új állapotot állít elő a keresés, azt abba a listába helyezi, amelyiknek az indexe megegyezik az állapot heurisztikájával, azt eredményezve, hogy nem kell az összes csúcsra megvizsgálni az egyezősséget, hanem elegendő azok között elvégezni a keresést, ahol lehetőség van az előfordulásra. Másik előnye a módosításnak, hogy miután kiderült egy állapotról, hogy az még eddig nem tartalma egyik adatbázisnak sem, rendezés nélkül lehet elhelyezni a megfelelő listába, hiszen ott is egyforma heurisztikájú csúcsok szerepelnek. Továbbá szükséges egy `SortedDictionary` objektum tárolása, melyre azért van szükség, hogy az növekvő rendezettséggel tárolja, hogy milyen heurisztikájú csúcsból, mennyi fordul elő a nyíltak adatbázisában, így nem kell újra és újra végigjárni a listák tömbjét, nyílt csúcsot keresve, hanem elég, a példány első értékének kulcs attribútumának megfelelő indexűek közül az elsőt kiválasztani. A nyíltak, zártak számát és a kulcs-érték párok tömbjét a keresés során végig nyilvántartom, és mikor a `SortedDictionary` példánynak nincs több eleme a keresés véget ér. A keresés teljes, mivel az összes csúcs nyílván van tartva, és ahogyan azt az állapottér reprezentációban a kezdőállapotnál leírtam, az operátorral az összes különböző állapot előállítására lehetőség van, így véges lépésben a keresés a cél előállításával terminál, miután a terminális mezőt megfelelően beállítja, hacsak meg nem szakítják a keresést. Továbbá az operátor előfeltétel vizsgálatára sincs szükség, mivel az mindig végrehajtható, és még le is redukálható a számuk. Az utóbbi állítás annak eredménye, hogy az ugyanazon kör többszöri forgatása, nem eredményez újabb és újabb állapotot, legfeljebb a golyók számával egyenlőt, utána az állapotok csak feleslegesen ismétlődnének, ezzel a vizsgálatok számát növelve mindössze. Tehát egymás után kétszer nem forgatjuk ugyan azt a kört. Ezt elérni úgy lehet, hogy megvizsgáljuk, hogy az az operátor, melyik kör forgatását eredményezte, amely által az adott állapot létrejött, és azon operátorokat melyek első paramétere megegyezik az előbb meghatározottal, azt végre sem hajtjuk. Ezáltal meg lehet spórolni golyó számnyi csúcs előállítási idejét, valamint ezeknek a vizsgálatáét az adatbázisban, míg azt meg nem találná valamelyikben, hiszen azzal egy időben, mikor az előállt, az összes többi különböző is elkészült. Összefoglalva, az előbb említett változtatással és az adatbázisok szétdarabolásával igyekeztem a keresési idejét és költségét csökkenteni. A legtöbb esetben a csúcs ez által maga az állapot `Equals` metódus hívásainak a megszüntetésével, valamint a csúcsok heurisztika alapján történő sorba rendezésével. Azért van ezekre szükség, mert egy kisebb paraméterértékekkel rendelkező állapottér esetén is, csak az egyedi állapotok száma is óriás. A következő táblázat, a különböző állapotok számát tartalmazza, az alábbi beállítások mellett:

körök száma	golyók száma	színek száma	
		1	2
2	8	3432	4204200
	12	705432	1,5057E+11
	16	155117520	6,4233E+15
	20	35345263800	3,01626E+20
3	10	75957810500	8,44134E+16
	14	2,81594E+16	7,66178E+25
	18	1,149E+22	8,74E+34
	20	7,50479E+24	3,11E+39
5	18	4,10E+53	5,17E+74

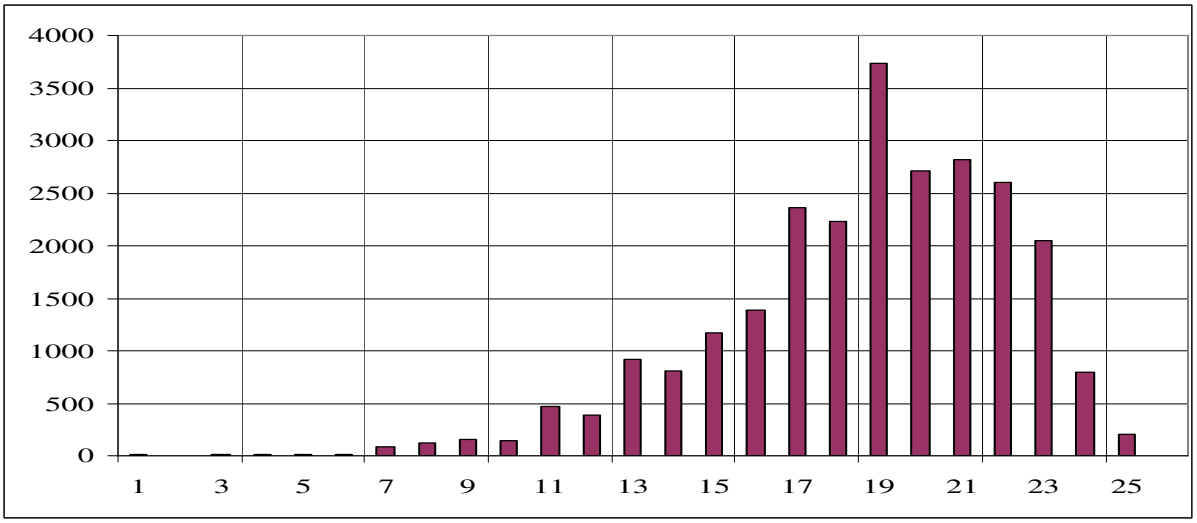
Kitűnik, hogy két szín, húsz golyó és egy színnel a számjegyek száma meghaladja a tízet. Két szín esetén tizenkét golyónál, míg három körnél már egy szín esetén is meghaladja tizenkét golyó mellett. A következőkben négy diagrammal szemléltetem, miként oszlanak el a csúcsok száma heurisztika értékük szerint osztályozva. Először két kör, tíz golyó és egy színre beállítva a paramétereket, az összes állapot száma 6 402 373 705 728 000, míg a különbözőek száma 48620, és a heurisztika 0 és 161 között fordulhat elő. Másodjára két kör, tizenkét golyó és egy színnél, a 1 124 000 727 777 607 680 000-ből 705 432 különböző állapot van 0 és 161 közti heurisztikákkal. A harmadik beállítás csak illusztráció, és a játék által nem beállítható állapottér: két kör hat golyó és két szín. Azért futtattam le a tesztet, hogy a kétszínes állapottéren lehetőség legyen az összehasonlításra. Ekkor 25 200 különböző állapot van 0 és 25 közti lehetséges heurisztikákkal. Az utolsó diagram szintén kétkörös állapottérről készül nyolc golyóval és két színnel, mikor a 4 204 200 különböző állapot a 0 és 49 közti értékeken oszlik meg. További vizsgálatokra az előforduló állapotok száma miatt nem került sor, mivel futási idejük több nap, illetve több hét lenne.



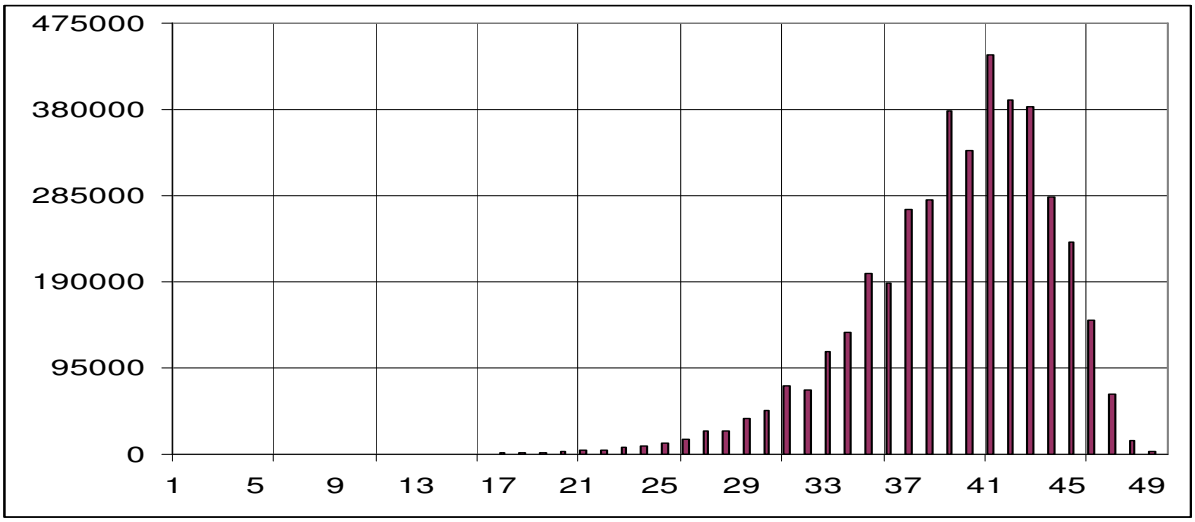
1. ábra: 2 kör, 8 golyó és 1 szín



2. ábra: 2 kör, 12 golyó és 1 szín



3. ábra: 2 kör, 6 golyó és 2 szín

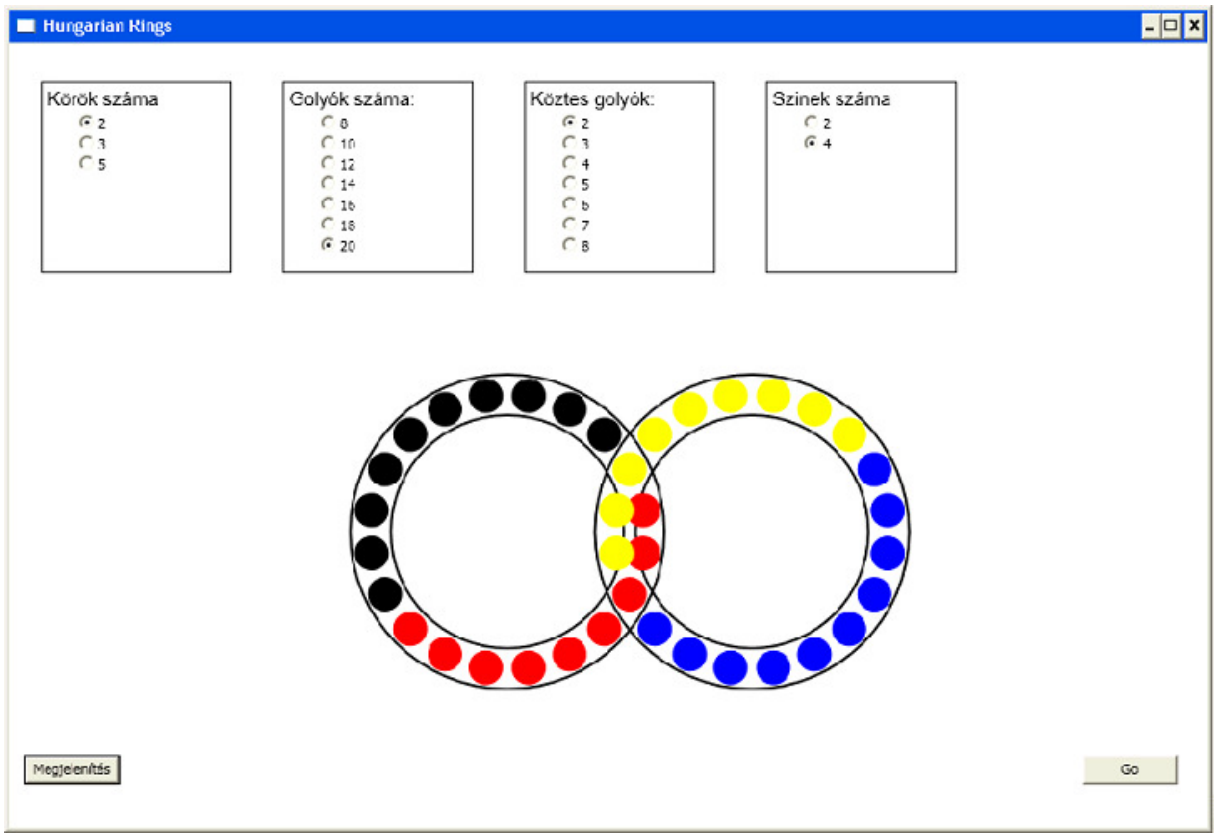


4. ábra: 2 kör, 8 golyó és 2 szín

A diagramokat szélességi kereséssel megfelelő állapottereken, és a célfeltételt álladó hamisra állítva készítettem, így a programok a nyílt csúcsok elfogyásával terminálták, az összes csúcs kifejtése után. Látható az állapotok eloszlása mind a négy esetben, ezáltal alkalmas a kereséses csökkentésére, hisz a részhalmazok számossága csak töredéke az egésznek. Viszont a szélességi kifejtéseknél továbbá az is nyilvánvaló lett, hogy a golyók és színek számának szorzata, jó közelítést ad az állapottér átmerőjére. Ez alatt azt a mélységet értem, amin belül a keresési gráf összes különböző csúcsa bejárható. A heurisztika függvény ebből kifolyólag, nem alkalmas A* algoritmus implementálására, mivel jóval meghaladja értékkészlete a lépések számát. Az algoritmus készítése során sok időt fordítottam alkalmasabb heurisztikus függvény írására, de nem sikerült jobbat megalkotni.

e. Felhasználói felület

A programot grafikus felületen keresztül lehet használni. Az alkalmazásnak összesen két ablaka van. Az első ablak a paraméterek megadására szolgál. Az megjelenítés gombbal tudjuk ellenőrizni, hogy helyesen vannak e beállítva az értékek. A Windows Presentation Foundation nem ad olyan eszközt a programozónak, amivel így lehet megjeleníteni a köröket, ezért nekem kellett egy megfelelő algoritmust előállítanom, ami elég rugalmas, hogy minden esetben megfelelően jelenítse meg a felhasználó által elvárt játékot. Az alapgondolat az volt, hogy a geometriai ismeretekre támaszkodva, minden egyes kis kör helyét „kézzel adom meg” vagyis kiszámoltatom.

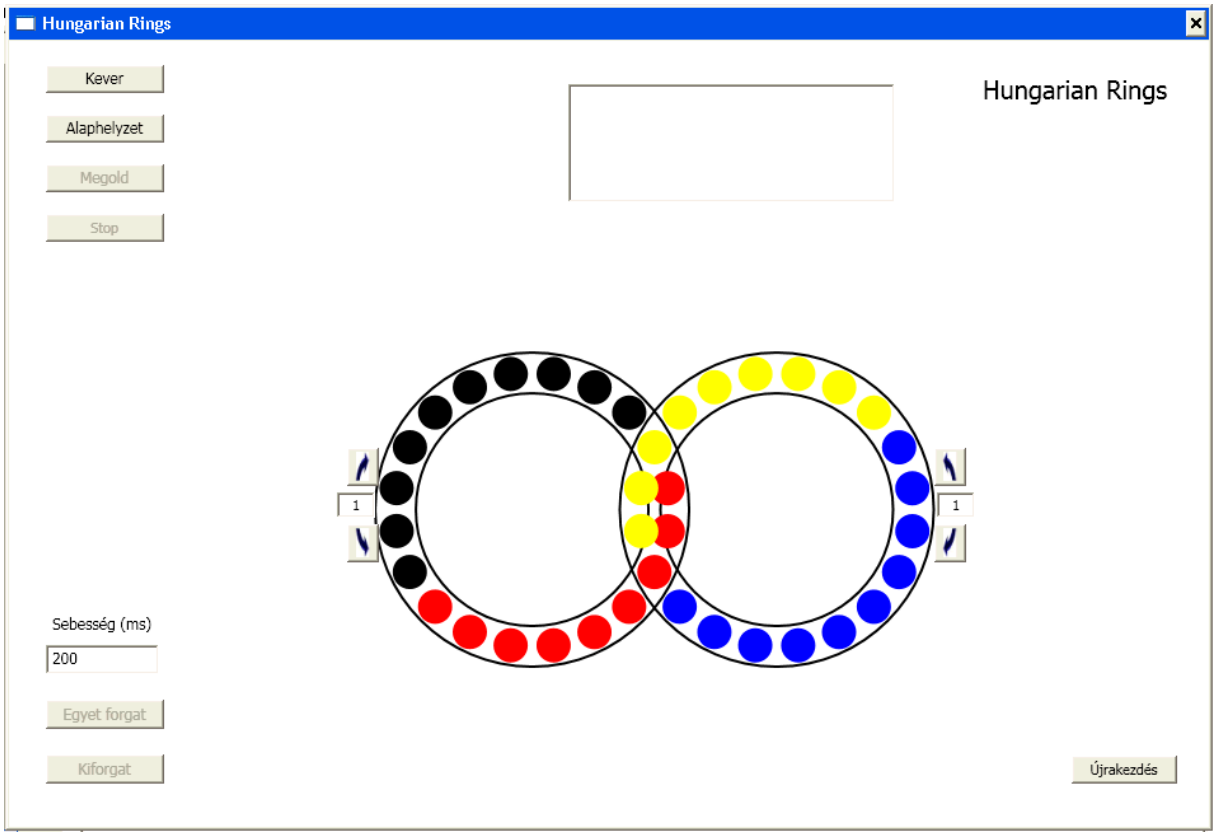


5. ábra: Felhasználói felület első ablaka

Több információ is rendelkezésre áll bonyolultabb számolások nélkül is. Tudjuk, hogy minden kis kör egyenlő távolságra helyezkedik el a nagy kör középpontjától és egymástól, így egyszerűen kiszámolható, bármely két egymás mellett elhelyezkedő „golyó” és a nagy kör középpontja által alkotott háromszög minden szöge, valamint ezek által, a körök egymáshoz mért távolsága is. A következő geometriai tétel, amit alkalmazok, a koordinátageometriában használt két pont távolságára szolgáló azonosság. Amit meg kell határozni, az a következő kirajzolásra váró kör középpontja. Ezt egy privát metódussal végzem el, ami hat lebegőpontos és egy egész értéket vár. Az első két lebegőpontos, a középpont koordinátáját jelöli, a következő kettő az előbb kirajzolt kör középpontjának a koordináta értékei, aztán rendre az előbbiektől mért távolságok, és végül a következő elhelyezkedése. A kirajzolást a legjobboldalival kezdem, amiből páratlan belsőgolyó érték esetén egy, párosban két darab van. Az előbbinél, az első y koordinátája megegyezik a középpontéval, az x pedig egyenlő a középponttól jobbra a nagykör sugarával. Másik esetben a pontmeghatározó eljárással úgy számolom ki, mintha az előbbi ponttól fél távolságra lenne. Az eljárás lényegében egy két ismeretlenes másodfokú egyenletrendszert old meg, aztán az eredményekből az utolsó paraméter segítségével meghatározza, hogy melyik az algoritmus szempontjából helyes koordináta. A golyók első közel egynegyede az előbbitől balra és fentebb, a második balra és

lentebb, a harmadik jobbra és lentebb, míg a negyedik negyede jobbra és fentebb van. Vannak még a beállítástól függően olyanok, melyeknek egyik koordinátája megegyezik az előzővel, de minden lehetőség le van kezelve. Továbbá függnek a koordináták attól is, hogy hány körös a játék és hányadik körhöz határozzuk meg a golyókat, mivel akkor eltolást kell hozzájuk adni. Az utolsó ehhez kapcsolódó feladat az, hogy azokat a pozíciókat ne őrizze meg duplán, amelyek a metszéseknél vannak, hanem csak egyszer, azoknál a körnél, amelyeknél tényleg szerepelnek. Ilyen trükköket használva meg lehet határozni, egy körhöz tartozó kis körök középpontjait, melyek alkalmasan eltolva a többi pontot is megadják. Illetve van két metódus, a körök színnel való kitöltéséhez és a nagykörök elhelyezéséért. A másik gombbal lehet ablakot váltani, ahol lehetőség van a valódi játékra, melyet végezhetünk a keresővel és kézzel is. Az ablakban állandóan jelen van a kirajzolt játék, amit gombokkal lehet módosítani. A gombokat négy csoportba lehet osztani. Az egyik csoport a keresőhöz tartozik. A Kever gombbal egy tetszőleges és szabályos kezdőállapot adható meg, a Keres gombbal a keresést lehet elindítani, míg a Stop gombbal megszakítható. A másik csoportba azok tartoznak, amelyekkel a golyók elhelyezkedését kézzel lehet állítani. Minden kör mellet van két gomb, melyekkel a rajtuk lévő irányba lehet forgatni. A gombokat használhatjuk kezdőállapot beállítására is, viszont arra is lehet, hogy egy összekevert játékot kirakjunk. A forgatások számát a gombok között lévő textbox-ban lehet megadni. A szövegdobozban csak egy és a golyók száma közötti érték szerepelhet, különben az alapértelmezett egy érték lép érvénybe. Valamint az Alaphelyzet gombbal, a golyók sorrendjét a kezdéskor megadottra lehet állítani. A forgató gombok megnyomása, magával vonja a keresés megszakítását is, amennyiben az algoritmus fut, mivel azaz állapot már nem egyezik meg azzal, amivel elindítottuk. Harmadik csoport gombjai felelősek a keresés megoldásának a megjelenítésére. Ez alatt azt értem, hogy mindig az éppen megjelenített állapoton végzik el a következő egy, vagy éppen összes műveletet, ami a megoldáshoz tartozik, és az eredménynek megfelelően rajzolják ki a golyókat. Egy operátor hatásának a megjelenítését úgy készítettem el, hogy amennyivel az operátor forgatja, annyiszor egy forgatásnak tűnik, hogy jobban követhető legyen. Ha többet forgat a golyók számának a felétől, akkor az operátor forgatási irányával ellenkezőkép forognak a golyók, amennyivel éppen szükséges. Egy további szövegdoboz lehetőséget ad arra, hogy megadjuk az egy-egy forgatások közt eltelő ezredmásodpercek számát. Negyedik csoportba csak az Újrakezdés gomb tartozik, amivel az egész alkalmazás újra elindítható, ha más paraméterekkel szeretnénk a játékot használni. Végezetül, még megemlíteném, hogy a keresési algoritmus futása alatt, végig egy textbox panel jeleníti meg a nyílt és zárt csúcsok számát, és azt, hogy mennyi ideje fut, hogy a felhasználó nyomon követhesse az

eredményeket. Továbbá a keresési algoritmus más szálon fut, hogy a keresés ideje alatt, ne veszítsük el a kapcsolatot a grafikus felülettel és a megfelelő gombokat szabadon használhassuk.



VI. Összegzés

A szakdolgozatomhoz a témát azért választottam a mesterséges intelligencia területéről, mert érdekel ez a megközelítés a problémák megoldásához. Számomra ami kiderült az az, hogy egy probléma állapottér reprezentációt könnyű megadni, viszont olyat készíteni sok munkát igényel, amivel a leghatékonyabb keresést lehet megvalósítani. Sok zsákutcába futottam, és sokszor kellett megváltoztatni a reprezentációt, hogy kevesebb számítással érje el ugyan azt az eredményt. Ilyen eset volta például az is, mikor az operátor működését megváltoztattam. Kezdetben egyszerre csak eggyel lehetett a golyókat elforgatni, de így sokkal több ellenőrzést és számítást kellett végrehajtania, valamint abban az esetben, a később alkalmazott optimalizálás, miszerint egy kör egymás után kétszeri forgatását nem végezzük el, nem megvalósítható. Amit nem sikerült elérnem, az a pontos heurisztika elkészítése, mivel jóval felülbecsüli a hátralévő lépések számát. Azért választottam a Best First keresőt az A algoritmus helyett, mivel az operátorköltség egységnyi, viszont sekély mélységben hosszabb keresés után találja meg a megoldást. Remélem, hogy sikerült középutat találnom a pontosság és az érthetőség között, hogy azok az olvasók is hasznosnak találják a dolgozatom, akik nem foglalkoztak még a mesterséges intelligenciával és az általános kereső algoritmusokkal, ezáltal jó kiindulópont a mélyebb ismeretek megszerzéséhez.

Köszönetnyilvánítás

Köszönettel tartozom Dr. Nagy Benedek Tanár Úrnak a szakdolgozat elkészítésének hosszú ideje alatt nyújtott iránymutatásaiért és türelméhez. Továbbá a tanszék többi oktatójának, mivel az általuk tartott mesterséges intelligenciával foglalkozó kurzusok oka az, hogy témámnak ezt választottam.

Irodalomjegyzék

[1] Mesterséges Intelligencia, Aula kiadó, 1999 Szerk.: Futó Iván. Szerzők: Dombi József, Fadgyas Tibor, Fekete István, Futó Iván, Gregovics Tibor, Gulyás László, Gyimóthy Tibor, Hegedüs Csaba, Jelasity Márk, Kis Tamás, Kutor László, Molnár Bálint, Nagy Sára, Prászéky Gábor, Ruttkay Zsófia, Sántáné Tóth Edit, Szepesvári Csaba, Szeredi Péter, Tatai Gábor, Tóth László, Turán György, Vámossy Zoltán, Váncza József

[2] Stuart Russel, Peter Norvig: Mesterséges intelligencia modern megközelítésben, Panem könyvkiadó, 2005