

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

EGY ÁLLÁSISMÉTLÉS-FELISMERŐ ALGORITMUS A SAKKBAN

Készítette: Németh Nándor
ELTE IK
MSc 1. évfolyam

Témavezető: Gregorics Tibor
egyetemi docens
ELTE IK
Programozáselmélet és
Szoftvertechnológiai Tanszék



Budapest, 2012. november

TARTALOM

1. Bevezetés	3
2. Az állásismétléses döntetlen játékszabálya	5
2.1 A játékszabály a FIDE alapján	5
2.2 A játékszabály értelmezése	5
2.3 A sakkprogramok viszonya a játékszabályhoz.....	5
3. Az állásismétlések felismerésének jelentősége	9
3.1 A futásidő-változás.....	9
3.2 A program logikája.....	9
4. Fogalmak.....	11
5. Helyes eredményt adó algoritmusok az állásismétlések felismerésére	12
5.1 Az alap algoritmus	12
5.2. Az Axon sakkprogram algoritmus.....	14
6. Statisztikusan helyes eredményt adó algoritmusok	17
6.1. Az állások sorozatának ábrázolása hash kulcsok listájával.....	17
6.2. Segéd hash tábla használata, kulcsok tárolásával.....	18
6.3 Segéd hash tábla használata, állások darabszámával	21
7. Az új algoritmus	22
7.1 Az algoritmus ötlete	22
7.2 Az algoritmus tömbindexes lépésgenerálás esetén	24
7.3 Az algoritmus bittérképes lépésgenerálás esetén	29
7.4 Egy hasonló algoritmus bittérképes lépésgenerálás esetén	30
8. A más helyen álló bábuk számának felhasználása	32
9. Eredmények elemzése	35
9.1 A tesztelő program	35
9.2 A tesztelés menete	36
9.3 Mért eredmények.....	37
10. Összefoglalás	42
11. Felhasznált irodalom.....	43

1. BEVEZETÉS

A sakk játékszabályai között találhatunk olyanokat is, amelyek gondoskodnak arról, hogy a játszma ne lehessen végtelen hosszú. A sakkprogramba nem nehéz beépíteni az ún. 50 lépés szabályát, amely szerint, ha úgy történik a játszma során 50 lépés, hogy közben nincs sem ütés, sem gyaloglépés – és bármelyik játékos igényli –, akkor a játszma eredménye döntetlen. Ehhez képest jelentősen nagyobb feladat az ismétlődő állások észrevételére jó megoldást találni. Döntetlenre vezet ugyanis az, ha a táblán háromszor ugyanaz az állás jelenik meg.

A problémát viszonylag bonyolult algoritmusok oldják meg a sakkprogramokban, sőt, néha az eljárás időigénye miatt a programkészítők lemondanak arról, hogy programjukat „megismertessék” ezzel a lehetőséggel. A TDK dolgozatomban egy olyan algoritmust írok le, amely bizonyíthatóan helyesen oldja meg a háromszori állásismétlődés felismerését, és a szakirodalomban található, hasonlóan helyes eredményt adó módszereknél gyorsabban működik.

Az állásismétlések felfedezése a sakkprogram játékerejét sok esetben fokozni tudja. Hátrányos helyzetben a program észreveheti az esetleges állásismétlésben rejlő döntetlen lehetőséget, amely számára kedvező. Erős állás esetén viszont tud vigyázni arra, hogy ne adjon lehetőséget az ellenfélnek a döntetlenre, ha meg lehet akadályozni az állások ismétlődését.

A jelenlegi sakkprogramok elég változatos megoldásokat alkalmaznak a probléma megoldására. A feladatot a legegyszerűbb úgy megfogni, hogy az aktuális állást (visszafelé haladva) összehasonlítjuk a korábbi állásokkal. Egy másik módszer, hogy az állások helyett az egymást követő lépések alapján ismerjük fel az állásismétlést. Gyakran nincs is más lehetőség az állásismétlés észrevételére, hiszen a programok általában nem tárolják a korábbi állásokat a fölösleges memóriamozgatások elkerülése érdekében. Az itt említett algoritmusok futási ideje arányos a legutóbbi ütés, vagy gyaloglépés óta megtett lépések számával.

Léteznek hash-kulcsok segítségével működő (ún. statisztikusan helyes) eljárások is, amelyek az állásból előállítanak egy nagy egész számot, mint hash kulcsot. Ezek közös jellemzője, hogy megengednek egy minimális tévedési valószínűséget. Itt azonban találhatunk konstans futásidejű megoldásokat is.

Az általam javasolt módszer az aktuális állás, illetve az előtte megtett lépések alapján fedezi fel az esetleges állásismétléseket. Az alkalmazott algoritmus, bár az Axon nevű sakkprogramban megvalósított változattal megegyező számú lépést vizsgál meg, azonban általában jóval kisebb konstans szorzóval dolgozik. Az algoritmusom bizonyíthatóan helyesen működik. Kis módosításokkal alkalmazható mind az ún. táblaindexes, illetve a bittérképes lépésgenerálás esetén.

Az algoritmusom egy távolságfüggvényt számol ki két állás között. Ez a távolságérték azonban nem csak a döntetlenek felismerését szolgálhatja. Jelzi, hogy a két állás között milyen sok bábú mozdult el. Ez az érték esetleg felhasználható az értékelőfüggvényben. Megnyitásoknál az állások közötti nagy távolság támogatásának eredményeként a bábuk gyorsabban fejlődnek ki. A végjátékban jelezhetik a sok lépés ellenére is kis távolságban lévő állások a gyanúsán döntetlen helyzeteket.

A hash-kulcsokat alkalmazó módszerek sok esetben összekapcsolhatók a helyes eredményt adó eljárásokkal. Ekkor a hash-kulcsot alkalmazó algoritmusnak olyannak kell lennie, ami állásismétlés esetén biztosan jelez állásismétlést. Ekkor a statisztikusan helyes eljárás könnyen ellenőrizhető egy másik, pontos eredményt adó algoritmussal. Így összességében egy még gyorsabb futásidejű helyes megoldást kaphatunk. Az általam kifejlesztett algoritmus is rendelkezik ilyen felhasználási lehetőséggel.

KÖSZÖNETNYILVÁNÍTÁS

Szeretném megköszönni Gregorics Tibor tanár úrnak, hogy témavezetőként segített a dolgozatom elkészülésében. Köszönöm Fekete István tanár úrnak, hogy tanácsaival, észrevételeivel támogatta a dolgozat elkészítését. Köszönettel tartozom szüleimnek, hogy támogatták a témaötletemet. Köszönöm Gévay Gábornak, hogy ötleteivel segítette a dolgozat létrejöttét.

2. AZ ÁLLÁSISMÉTLÉSES DÖNTETLEN JÁTÉKSZABÁLYA

Legelőször nézzük meg az állásismétléses döntetlen játékszabályát a FIDE (Nemzetközi Sakkszövetség) [1] alapján! Ezt követően vizsgáljuk meg, hogy egy sakkprogramnak hogyan érdemes viszonyulnia a játékszabályhoz, hogy az állásismétlés szabályának ismeretét eredményesen használni tudja.

2.1 A JÁTÉKSZABÁLY A FIDE ALAPJÁN

A játszma döntetlen a lépésen lévő fél kérésére, ha a sakktáblán lévő állás legalább harmadszorra jött létre, vagy, hogyha a játékos a most következő lépésével létre tud hozni olyan állást, ami a lépésével harmadszorra jönne létre.

Két állás megegyezik, ha a sakktábla egyes mezőin ugyanolyan típusú bábuk állnak, ha ugyanaz a fél jön lépésen, és ha az állásokból kiindulva megtehető lépések lehetőségei (beleértve a sáncolást és az en passant lépés lehetőségét) megegyeznek.

2.2 A JÁTÉKSZABÁLY ÉRTELMEZÉSE

A sakkban vannak visszafordíthatatlan lépések, amelyek után már nem jöhet létre a játszma során ismét lépés előtt létrejött állás. Ezek legegyszerűbb példái az ütések illetve a gyaloglépések. Szintén ide sorolhatunk minden olyan lépést, ami valamelyik fél sáncolási jogát csökkenti a királyoldalon, a vezérszárnynon, esetleg mindkét irányba. Ezen kívül, ha van egy állásunk, ahol meg lehet tenni en passant lépést is, ott egyik lépés sem visszafordítható. Ennek oka, hogy az adott en passant lépést csak egyszer lehet meglépni, közvetlenül a gyalog duplalépése után.

A játszma során állásismétléseket keresni elég nyilván két visszafordíthatatlan lépés között.

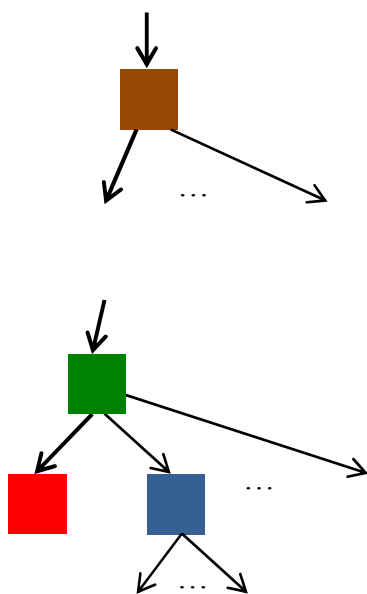
2.3 A SAKKPROGRAMOK VISZONYA A JÁTÉKSZABÁLYHOZ

A sakkprogramok a minimax algoritmus esetén megvizsgálják a sakktáblán látható állásból kiinduló lépéssorozatok során létrejövő állásokat egy bizonyos mélységig. A megvizsgált állások esetén azonban nehéz eldönteni, hogy belőlük létrejöhet-e már korábban kétszer létrejött állás. Emiatt a vizsgálatot a harmadszori állásismétlések felismerésére kell leszorítanunk.

A játékszabály nem mondja ki, hogy döntetlen a játszma egy állás harmadszori előfordulásánál. Ezt a játékosok kérhetik, ha kérni akarják. Tegyük fel, hogy a programunk a

játékfaban döntetlennek tekint egy állást mindig, amikor az állás már harmadszorra jött létre. Vizsgáljuk meg, hogy ezzel a stratégiával hátrányba kerülhet-e?

Elsőként foglalkozunk azzal az esettel, amikor az állásismétlés nem a játékfa gyökércsúcsát érinti, hanem egy belső csúcsot.



1. ábra

A játékfa egy belső csúcsán észlelt harmadszori állásismétlés

Az ábrán a barna csúcs jelzi a sakktáblán is látható állást. A pirossal jelzett állásnál vettük észre, hogy a vastag nyilakkal jelzett, az álláshoz vezető úton már harmadszorra jött létre az állás. A zöld csúcs jelzi a piros állás (közvetlen) szülőjét. A kék állás a harmadszorra létrejött állás egy esetleges testvére.

Három eset lehetséges. Ha a piros állás kiegyensúlyozott, akkor nem érdemes tovább folytatni a belőle kiinduló lépéslehetőségek vizsgálatát. Ha bármelyik fél akar és tud kezdeni valamit a helyzettől, akkor eddig a pontig nem juthat el a sakkjátszma, hiszen kiegyensúlyozott helyzetnél a harmadszori állásismétlés a felek döntetlenben való kölcsönös megegyezését jelenti.

Ha az állásban lépő fél hátrányban van az ellenfélhez képest, nyilván örömmel fogadja, ha döntetlent kérhet. Ha annak a félnek, aki létre tudta hozni az állást harmadszorra, van jobb ötlete is a döntetlen felajánlásánál, azaz például az ábrán látható kék állás számára kedvezőbb

variáció, akkor a minimax algoritmus biztosítani fogja, hogy az ábrán látható piros állás az abból az állásból lépő fél számára előnyös legyen.

Ha a harmadszor létrejövő állásban a lépő fél áll jobban, akkor folytathatnánk a létrejövő állás következményeinek vizsgálatát. Azonban, ha a lépések során eljutunk a zöld állásig, akkor az ott lépő félnek lehetősége lenne döntetlent kérni. Emiatt döntetlennek tekinthető az összes, játékfa belsejében harmadszorra létrejövő állás. Így a stratégiának nincsenek hátrányai.

Azt külön kell megvizsgálni, hogy a program a sakktáblán létrejött állás alapján mindig kérjen-e döntetlent, ha megteheti. Az egyetlen kérdéses eset, hogy a játékos előnyösen áll, és a sakktáblán harmadszorra létrejött állás van. Ebben az esetben, ha van ötlete a lépő félnek az újbóli állásismétlés megakadályozására, nyilván nem érdemes döntetlent kérnie. Azonban ez az eset elvileg nem jöhet létre, ha feltételezzük, hogy az ellenfél logikusan gondolkodik, hiszen ebben az esetben kérte volna a döntetlent az állást létrehozó lépés megtétele helyett.

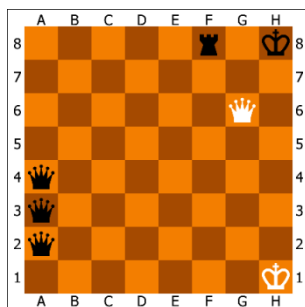
A másik fontos kérdés, hogy tényleg szükséges-e a másodszori állásismétlésig várni, vagy a sima állásismétlés esetén is tekinthetjük az állást döntetlennek? Bármelyik fél, ha ki tud kényszeríteni egyszer egy állásismétlést, akkor semmi sem tudja megakadályozni abban, hogy újra létrejöjjön az állás.

A gondolatmenet abban az esetben lehet téves, ha a programunk keresési mélysége kisebb az ismétlődő állások közötti lépések számával. Ebben az esetben ugyanis véletlen szerencse is közbejátszhatott abban, hogy az állás újból létrejött. Ekkor tehát az alkalmazott keresési mélység nem tekinthető elegendőnek az állásismétlés megismételhetőségének vizsgálata szempontjából. Ebből következik, hogy ha az ilyen állásokról nem feltételezzük, hogy döntetlenek, ugyanúgy hibát követhetünk el. Azonban, ismerve, hogy sakkban az előnyben lévő félt megnyitásban, középjátékban, illetve 5-6-nál több bábút tartalmazó végjáték-állásokban legfeljebb állásismétlésekkel lehet megakadályozni, hogy az előnyt tovább növelje, feltételezhetjük, hogy az állás másodszori létrejöttét döntetlennek feltételezve kisebb hibát követünk el, mint folytatva az állásból kiinduló lépések elemzését.

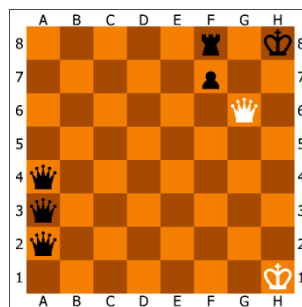
További előnye a módszernek, hogy egy kikényszeríthető, viszonylag rövid állásismétlődés felismerése jóval kisebb mélységű kereséssel megoldható, mintha a harmadszori állásismétlést vennénk alapul, aminek felismerése legalább 4 féllépéssel nagyobb mélység esetén (több ezerszer annyi állás megvizsgálása során) sikerülhet.

A következő példák során tekintsük az állásokat olyan állásoknak, amelyek a játékfa belsejében jöttek létre. A belőlük kiinduló keresési mélység határozza meg, hogy az adott

állásokat hátrányosnak, kiegyensúlyozottnak (vagy döntetlennek), esetleg előnyösnek fogja észlelni a minimax algoritmus. Ezek a különbségek jelentősen megváltoztathatják a játékfa gyökércsúcsában a lehetséges lépések jóságát. Gyakran előfordul, hogy idő híján nem teheti meg a sakkprogram, hogy olyan nagy mélységekben vizsgálódjon, ami észrevénné a harmadszori állásismétlést is.



4/8 féllépés szükséges



9/17 féllépés szükséges

2. ábra

A döntetlen felismeréséhez szükséges keresési mélység másodszor, illetve harmadszor létrejövő állás esetén.

Ezen kívül, az algoritmus jelentősen nagyobb részfákat vág le a játékfaból, hogyha az első állásismétlés után vágnánk, szemben a harmadszori állásismétlés utáni vágással, legalább 4 mélységgel nagyobb részfákat.

Látható tehát, hogy elég a játékfa vizsgálata során minden létrehozott állásról meg kell vizsgálni, hogy létrejött-e már korábban, vagy sem. Azaz, a játszma korábbi állásai, vagy a játékfa gyökércsúcsától az adott állásig vezető út mentén vizsgált állások között megtalálható-e az állás. A játszma korábban létrejövő állásai közül elég azokat az állásokkal foglalkozni, amelyek után nem történt a játékfa gyökeréig visszafordíthatatlan lépés.

A jelenlegi színvonalasabb sakkprogramok mind a most bemutatott módszerrel állnak neki az állásismétléses döntetlen játékszabályához.

3. AZ ÁLLÁSISMÉTLÉSEK FELISMERÉSÉNEK JELENTŐSÉGE

Az állásismétléses döntetlenek sokkal gyakoribb jelenségek, mint az 50 lépés szabálya miatt bekövetkezők. Mivel jelentősen bonyolultabb, hosszabb futásidejű algoritmusok szükségesek a probléma megoldására, elég gyakori, hogy egyszerűbb sakkprogramokból kihagyják ennek a szabálynak a figyelését. A most következő sorokban próbálok érvelni amellet, hogy miért fontos megismertetni a programunkat ezzel a lehetőséggel is.

3.1 A FUTÁSIDŐ-VÁLTOZÁS

Ha egy állásban kiderül, hogy döntetlen állás, akkor tovább már nem kell vizsgálni az adott állásból kiindulva, hiszen az állás értéke a döntetlen pontszám. És ekkor megvizsgálható: az állásismétléses döntetlen viszonylag bonyolult algoritmusra tényleg vesz-e annyi időt, mint amennyit nyertünk az általa létrehozott vágások miatt?

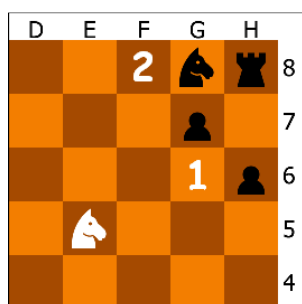
A tapasztalat azt mutatta, hogy a minimax algoritmusban mindössze 3000-7000 állásvizsgálatonként fordul elő egy állásismétlés miatti vágás. Minden 1000-2000. állásra nem kell meghívnunk az értékelőfüggvényt a vágás miatt. Mivel minden állásra le kell, hogy fusson az állásismétlés felismerő eljárás, ezért, ahhoz, hogy a vágásokkal gyorsabban futó minimax algoritmust kapjunk, az értékelőfüggvény átlagos futásidejének 1000-2000-szer nagyobbnak kellene lennie, mint az átlagos állásismétlés észlelő algoritmusnak. Az értékelőfüggvény jó részét nem kell minden állásra külön kiszámítani, hiszen megtehetjük inkrementálisan is, lépésenként módosítva az állások pontszámát. Emiatt elég bonyolult értékelőfüggvény kéne ahhoz, hogy a gyorsulást megtapasztaljuk.

A tesztprogramomban a futásidő az állásismétlés felismerésével az állásismétlés észlelés nélküli algoritmushoz képest az idő 0,5-1%-ával növekszik meg (lásd eredmények értékelése).

3.2 A PROGRAM LOGIKÁJA

Ha a program a játékfában nem ismeri fel a lépésismétléses döntetlent, akkor nem tesz semmit az oda-vissza lépegetések megakadályozására. Ha a program helyzete előnyösebb, akkor is kialakulhat döntetlen játszma. Hiszen elképzelhető, hogy egy állásban a program épp a korábbi lépésének megfordítottját találja a legmagasabb pontszámú lépésnek. Ezek után megvan a partiban az oda-vissza lépegetés lehetősége, amit az ellenfél könnyedén kihasználhat egy döntetlen szerzésére.

A gyengébb fél néha ki is kényszeríthet egy oda-vissza lépegetést, mint ahogy a következő állásban láthatjuk. Tegyük fel, hogy a fekete áll jobban, azonban a bástyáját nem szívesen vesztené el. A mintaállásban a feketének előre kellett volna számolnia, hogy a világos ló így folyamatosan támadhatja a bástyát, és így tennie kellett volna valamit az állás létrejötte ellen. Az ilyen helyzetek létrejöttét 6 féllépés mélységű keresés tudja megakadályozni, ha megteheti.

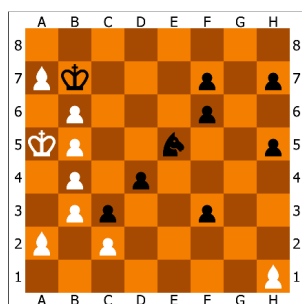


3. ábra
világos lép

A számok a világos huszár lépéseit jelzik.

Gyakoribb eset, amikor a királyt tudja valamelyik fél folyamatosan sakokban tartani. Ennek felismerése szintén fontos feladat. Ilyenkor gyakran előforduló jelenség, hogy a király több, mint két mezőn állhat, és emiatt jelentősen növekedhet a szükséges keresési mélység, ami felismeri az állásismétlést.

Az állásismétlődések figyelése sok esetben nagyon nehéznek tűnő problémák esetén is sokat könnyít az optimális játszmafolytatás megtalálásában. Például Nenad Petrovic 271 lépéses mattfeladványában [3] a mattot adó fél lépéseinek kb. 90%-a megkapható 6 féllépés mélységű kereséssel, ha figyelünk arra, hogy már létrejött állásokat ne hozzunk újra létre. Bár a horizonton belül nem látunk semmi izgalmasat, a döntetlen elkerülése érdekében mégis megtalálható a folytatás.



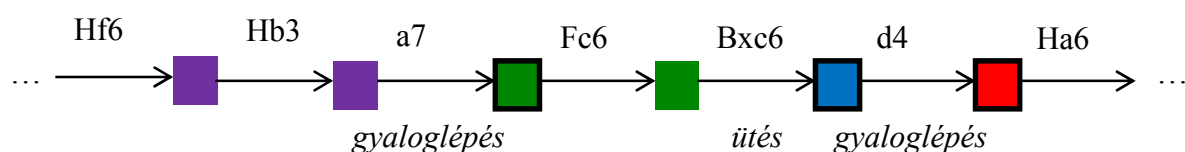
4. ábra
világos lép

271 lépésből matt

4. FOGALMAK

A következőkben bemutatandó algoritmusok könnyebb megértéséhez érdemes bevezetni néhány fogalmat.

Egy **ütközési osztály**ba tartoznak azok az állások, amelyeket ugyanaz a visszafordíthatatlan lépés előz meg. A játszmák elején előfordul, hogy vannak olyan állások, amelyek előtt nincs visszafordíthatatlan lépés. Nyilván ezek is egy ütközési osztályba tartoznak. Példaként nézzük a következő képzeletbeli lépéssorozatot. Az azonos színű állások egy ütközési osztályba tartoznak:



5. ábra
Ütközési osztályok

A legtöbb esetben csak egy ütközési osztályban kell az állásismétlésekkel foglalkozni.

Ütköző állásnak nevezem az ütközési osztályba tartozó állások közül a legkorábban létrejöttet, azaz a visszafordíthatatlan lépés után létrejövő állást. Ez lehet a sakkjátszma során már megtett állás, de lehet, hogy már a játékfaban jött létre. Nyilván az alapállás is ütköző állás. Az ábrán, a bekeretezett négyzetek jelzik az ütköző állásokat. Nyilván minden nem ütköző állás előtt található az ütközési osztályának legelején egy ütköző állás. Az ábrán a lilával jelzett ütközési osztály első eleme nem látható.

Az **aktuális állás** a következőkben nem a sakkjátszmában épp most létrejött állást, hanem a játékfaban az adott pillanatban elemzett állást fogja jelenteni. Erről vizsgáljuk meg, hogy létrejött-e vagy sem a korábbiakban.

A problémát a sakkprogramozók kétféleképpen közelítették meg. A **helyes eredményt adó algoritmusok** mindig az általunk várt eredményt adják, helyességük bizonyítható. Ezzel szemben a **statisztikailag helyes eredményt adó algoritmusok** nem feltétlenül adnak jó eredményt, kis valószínűséggel tévedhetnek. Ezek közé tartoznak a hash-kulcsok segítségével működő megoldások [4][5], amelyek, bár kis valószínűséggel hibázhatnak, cserébe gyorsabb futásidőt biztosítanak.

A következő sorokban bemutatok néhány, állásismétlést felismerő algoritmust. Ezután ismertetem az általam javasolt módszert. Ezeket egy tesztelő sakkprogram segítségével hasonlítottam össze futásidő, illetve hibázások szempontjából.

5. HELYES EREDMÉNYT ADÓ ALGORITMUSOK AZ ÁLLÁSISMÉTLÉSEK FELISMERÉSÉRE

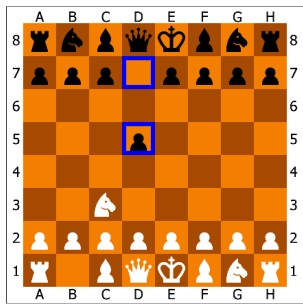
Helyes eredményt adó algoritmusoknak nevezem a dolgozatomban azokat az eljárásokat, amelyek pontosan akkor jeleznek állásismétlést, amikor az tényleg létrejött. A 6. fejezetben találhatunk olyan algoritmusokat is, amelynél megengedünk kis valószínűségű tévedést.

5.1 AZ ALAP ALGORITMUS

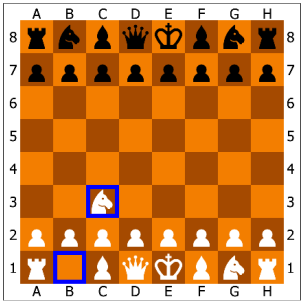
Tetszőleges sakktábla ábrázolás esetén alkalmazható módszer, hogy az utolsó ütközési osztályba tartozó állásokat összehasonlítjuk a játékfaban vizsgált aktuális állással.

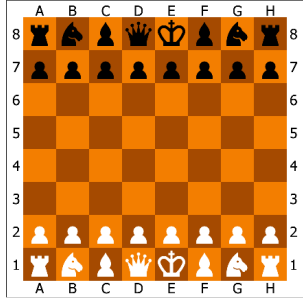
...

d7-d5



Hb1-c3





3. Ha végeztünk a d7-d5 gyaloglépés következményeinek elemzésével, akkor törölnünk kell a verem felüldefiniált alját. Ha a következő lépésünk pl. huszárlépés lesz, akkor a döntetlen ellenőrzésbe be kell vonni a korábbi állásokat is.
← **a verem felüldefiniált alja**

2. Visszafordíthatatlan lépések esetén jelezni kell, hogy az ezalatti állásokkal már nem kell összehasonlítani a játékfaban létrejövő új állásokat.

1. Mivel huszárlépés jön, nem szabad felüldefiniálni a verem alját, hiszen ez nem visszafordíthatatlan lépés.
← **a verem eredeti alja**

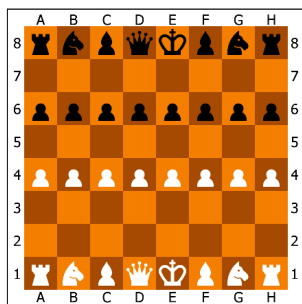
6. ábra

A verem aljának felüldefiniálása a kezdőállásból kiinduló keresésnél

Ennek megvalósítása egy verem segítségével történhet, melynek tetszőleges eleme olvasható, azonban írni és törölni mindig a legfelső elemét fogjuk. A veremben a minimax algoritmus indulásának elején a sakktáblán is látható állás ütközési osztályának elemei tartoznak. Új állás a megvizsgálásakor bekerül a verembe, míg egy állásban léphető lépések következményeinek elemzése után kikerül belőle. Így a verem egy ütköző állástól kezdve tárolja a minimax algoritmusban éppen vizsgált álláshoz vezető úton lévő összes létrejött állást. Mivel a játékfában is megvizsgálunk visszafordíthatatlan lépéseket, érdemes néha a verem alját felüldefiniálni. A feladat az aktuális állást a veremben megkeresni, ismerve azt, hogy az állás a verem teteje és a verem aljának legutóbbi felüldefiniálása között fordulhat csak elő.

Nyilván, nem kell minden egyes állást összehasonlítani a vizsgálttal, hiszen csak minden második állásban lép ugyanaz a fél. Tudjuk még, hogy ha egy állás után legalább négy féllépés kell ahhoz, hogy újra létrejöjjön. Ez akkor következik be, ha mind a két fél oda-vissza mozgatta a bábuit. Így az aktuális állástól visszafele a $\{k: k \geq 4 \text{ és } 2|k\}$ sorszámú állásokat kell vizsgálni.

Az új állás verembe helyezésének algoritmus a konstans időben történhet, viszont az állások összehasonlítása jelentős futásidőt igényel. A legrosszabb esetben 48 összehasonlítás szükséges annak eldöntésére, hogy az állás létrejött-e, hiszen ezt követően az 50 lépés szabályát érdemes használni a döntetlenek felismerésére. Az algoritmus elég lassan fut, ha közeledünk az 50. visszafordítható lépéshez.



7. ábra

Fix 2 mélységű programok esetén egyik fél sem mer támadni, hiszen kapásból észreveszi az ellenfél visszaütését. Ha a döntetlenért járó pontszámot 0 alá visszük, egyik fél sem akar kiegyezni egy állásismétléses döntetlennel. Az 50. lépés fele haladva jelentősen lassul a program futása.

A módszert javíthatjuk az összehasonlítás időszükségletének csökkentésével. Például, táblaindexes sakktábla ábrázolás esetén lehet egy jelentősen gyorsító módszer, hogy minden állás összehasonlítása előtt megnézzük, hogy a legutolsó lépés kiinduló-, és célmezeje megegyezik-e az épp vizsgált állás adott mezejeivel. Így általában egy-két összehasonlítás alatt kiderül, ha a két állás nem egyezik meg.

Látható, hogy az algoritmus futásideje első sorban attól függ, hogy a vizsgált állás előtt hány lépéssel történt a legutolsó visszafordíthatatlan lépés. Így a módszer a vizsgálandó állások számával lineáris futásidővel működik, de jelentős a konstans szorzó.

További probléma, hogy általában a korábban létrejött állásokat nem tárolják a programok a memóriában. Ehelyett csak egy sakktáblával dolgoznak, azon teszik meg a lépést, és azon vonják vissza. Így, az állások vermét csak a döntetlenfelismerő algoritmus miatt kell használni, ami a sakktábla ábrázolás helyigénye miatt jelentős adatmozgatást eredményez.

Megtehető, hogy az állásokat valamilyen módon egy ekvivalens, kevesebb bitet tartalmazó kóddá tömörítjük. Ekkor elég ezek között a kódok között nézni az egyenlőséget. Azonban a tömörítés gyorsaságát figyelembe véve nem mehetünk el túl kevés bitet tartalmazó kóddá. És a tömörítés is időigényes lehet.

A megvalósítás:

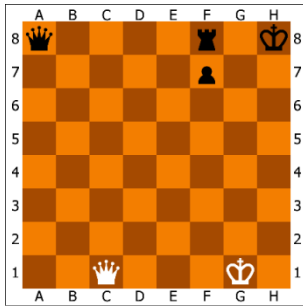
A tesztprogramban megvalósított sakktábla összehasonlításos módszer elsőként a lépő bábu célmezejével nézi az esetleges egyezést, majd, ha ez megegyezik, akkor vizsgálja meg a sakktábla összes mezején az egyezést.

5.2. AZ AXON SAKKPROGRAM ALGORITMUSA

Vučković és Vidanović az Axon sakkprogramjában alkalmazott egy elég érdekes állásismétlést felismerő algoritmust. Ennek részletes leírása megtalálható [8], illetve [9]-ben. A következő sorokban pár szóban bemutatom a problémára adott megoldásukat.

A most bemutatandó megoldás az egyes állások esetében a vizsgálandó állások számával továbbra is megközelítőleg lineáris futásidővel működik, a konstans szorzón igyekeztek faragni. A jelentős különbség az előbb bemutatott algoritmus, és az Axon-ban alkalmazott között, hogy az Axon nem az állásokat próbálja összehasonlítani, hanem a megtett lépéseken végighaladva fedezi fel az esetleges állásismétlést. A játékfaban az aktuális állásig megtett lépések kódjait szinte minden program tárolja valamilyen, jól olvasható módon. Éppen ezért, ez a módszer nem igényel annyi más szempontból fölösleges adatmozgatást.

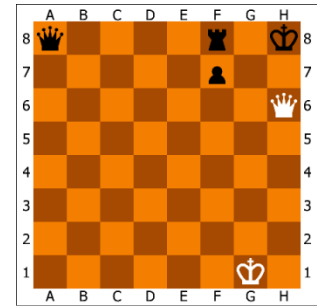
A következő példán [8] keresztül érthetjük meg az algoritmus működését.



ütköző állás

- (...)
1. Vh6+ Kg8
 2. Vg5+ Kh7
 3. Vh5+ Kg7
 4. Vg5+ Kh8
 5. Vh6+

közben megtett lépések



aktuális állás

8. ábra

Az Axonban alkalmazott algoritmust bemutató minta lépéssorozat

Az Axon-ban a lépéseket (*lépő bábu, kiindulási mező, célmező*) hármaskokban tárolják. Ennek megfelelően a példában látható lépések felírhatók a következő sorozattal (A példában a V a világos vezért, míg a K a sötét királyt jelenti):

V C1-H6 , K H8-G8 , V H6-G5 , K G8-H7 , V G5-H5 , K H7-G7 , V H5-G5 , K G7-H8 , V G5-H6

Képzeljük el, hogy a lépéssorozat valójában nem a példában látott módon történt meg, hanem az aktuális állásból kiindulva, a lépések fordítottjai mentén haladtunk végig, egészen az ütköző állásig. A feladat tehát azt meghatározni, hogy az aktuális állásból visszalépkedve az ütköző állásig, előfordult-e már az aktuális állás.

Az algoritmus során minden egyes báburól, ami megmozdult a visszalépegetés során, tároljuk, hogy honnan indult, és a visszalépegetés aktuális állapotában épp melyik mezőn áll. Amíg egy bábu nem ér vissza a kiindulási mezejére, addig a lépéssorozata **nyitott kört** (unclosed chain) alkot. Ha visszaért a kiindulási mezejére, akkor megtett egy **teljes kört**.

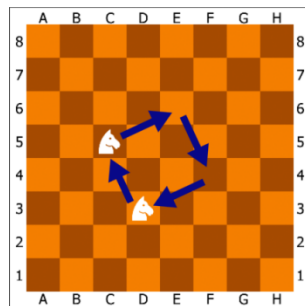
Kezdetben még egy bábu sem lépett, így a **nyitott körök száma** 0. Az algoritmus minden egyes lépésében vesszük a következő lépést, és megvizsgáljuk, hogy új nyitott kört kezdtünk-e meg a lépéssel, folytattunk, vagy esetleg befejeztünk-e egy nyitott kört? Ha új nyitott kört kezdtünk meg, akkor a nyitott körök számát növelnünk kell, míg ha teljes körre bukkantunk, akkor a nyitott körök számát csökkenthetjük. Arra vagyunk kíváncsiak, hogy van-e olyan lépés a vizsgált lépéssorozatban, amely után a nyitott körök száma újra 0 lesz-e.

Az algoritmus során a nyitott körök esetében csak a bábu kiindulási, illetve aktuális mezejét kell tárolni, hiszen nem használjuk fel az algoritmusban a két mezőt összekötő útvonalat. A nyitott köröket egy halmazban tárolhatjuk. Azokat a nyitott köröket, amelyek teljes körre váltak, kivehetjük a halmazból, többet nem lesz rájuk szükség. Így a minta lépéssorozat esetén a következő események történnek a nyitott körök halmazában:

Megtett visszalépés	A nyitott körök	számuk
V H6-G5	V H6-G5	1
K H8-G7	V H6-G5 , K H8-G7	2
V G5-H5	V H6-H5 , K H8-G7	2
K G7-H7	V H6-H5 , K H8-H7	2
V H5-G5	V H6-G5 , K H8-H7	2
K H7-G8	V H6-G5 , K H8-G8	2
V G5-H6	V H6-H6 , K H8-G8	2
-	K H8-G8	1
K G8-H8	K H8-H8	1
-	-	0

A következő lépéseket már nem szükséges visszavonni, hiszen elérkeztünk egy, az aktuálissal megegyező álláshoz.

Az eljárás során azért szükséges tárolni a bábuk neveit, mert elképzelhető, hogy két bábu helyet cserél egymással, és így jön létre az állásismétlés. Ekkor az algoritmus feladata felismerni, hogy két azonos típusú bábunál a nyitott köreik végpontjai megegyeznek.



9. ábra

Két huszár helycseréje például a következő lépéssorozattal:

1. C5-E6
2. E6-F4
3. D3-C5
4. F4-D3

Ekkor az Axon-ban alkalmazott algoritmus működése (eltekintek a menetközben megtett sötét bábuk lépéseinek vizsgálatától):

Megtett visszalépés	A nyitott körök	számuk
H D3-F4	H D3-F4	1
H C5-D3	H D3-F4 , H C5-D3	2
H F4-E6	H D3-E6 , H C5-D3	2
H E6-C5	H D3-C5 , H C5-D3	2
-	-	0

A nyitott körök halmazát a programozók egyszerű tömbbel valósították meg. Így a halmazban való keresés lineáris időben történhet. Ennek következményeképp a futásidő a lépések számával akár négyzetes is lehet. Gyakran előfordul azonban, hogy csak kevés bábu lépetet sokat, ekkor a halmaz mérete nem nő jelentősen a megtett lépések számával, így az algoritmus közel lineáris futásidővel dolgozik.

6. STATISZTIKUSAN HELYES EREDMÉNYT ADÓ ALGORITMUSOK

A statisztikusan helyes eredményt adó állásismétlés felismerő algoritmusok olyan eljárások, amelyek kis hibavalószínűséggel működnek. Kétféle lehetséges hibát véhetnek: vagy nem vesznek észre egy állásismétlést, vagy egy állásban állásismétlést jeleznek, miközben az valójában nem történt meg.

A sakkprogramokban gyakran alkalmaznak hash kulcsokat az állások azonosítására, nemcsak állásismétlések kedvéért. Ezek a hash kulcsok lényegesen kevesebb bitet tartalmaznak annál, amennyi szükséges az összes legális sakkállás megkülönböztetésére. Ennek következményeként nem biztosítható, hogy minden egyes hash-kulcshoz csak egyetlen állás tartozzon. Ez pedig a hash-kulcsokat alkalmazó algoritmusok megbízhatóságát csökkenti. Megadható olyan hash-kulcs generáló módszer is, amely több biten, pontosan tárolja az állásokat. Viszont ebben az esetben jelentősen megnő a hash kulcsok generálásának ideje.

A következőkben három megoldást mutatok be, amely hash-kulcsokat használ az állásismétlések felfedezésére. Ez a három algoritmus nem igényel túl nagy memóriát. A játékfa különböző ágain létrejövő azonos állásokat felismerő transzpozíciós táblák segítségével hasonlóan megoldható a probléma.

Ha az algoritmusok esetén a hibát le tudjuk korlátozni arra az esetre, hogy állásismétlést vettünk észre, közben nem is volt állásismétlés, akkor az állásismétlés felismerése után ellenőrizni tudjuk az eredményt egy mindig jó eredményt adó eljárás lefuttatásával. Ezáltal egy nagyon gyors módszerrel lehetőségünk van kiszűrni a különböző állások nagy részét, majd egy lassabbal biztosítani tudjuk, hogy a két algoritmus végül a tökéletes eredményt adja.

6.1. AZ ÁLLÁSOK SOROZATÁNAK ÁBRÁZOLÁSA HASH KULCSOK LISTÁJÁVAL

A következő sorokban a hash-kulcsokkal megvalósítható legegyszerűbb állásismétlés felismerésről beszélek. A módszer [6]-ban van leírva.

Az algoritmus ötlete megegyezik az 5.1 pontban található, a problémára adható legegyszerűbb módszer működési elvével. A különbség csak annyi, hogy a teljes sakkállások helyett azoknak csak a hash kódját tároljuk. Ezáltal, bár külön ki kell számolni az egyes állások hash-kódját, cserébe nem igényel az megoldás nagy memóriamozgatásokat, illetve az összehasonlítások elvégzése is két szám összehasonlítására egyszerűsödik. Ezért az algoritmus futásideje szintén lineárisan változik a megvizsgálandó állások számával. Viszont,

a konstans szorzó nagyon kicsi. Ennek következtében jelentősen gyorsabban lefut, mint az előbb bemutatott eljárás.

Az algoritmus esetén használhatunk például a sakkprogramokban megszokott 64 bites hash kulcsokat.

Az esetlegesen előforduló hiba csak akkor történik, ha az állásunk, és egy korábbi állásunknak ugyanaz a hash kulcsa. Ebben az esetben előfordulhat az, hogy azt hisszük, állásismétlés jött létre, közben pedig nem történt. Fordított hiba, azaz nem felismert állásismétlés ezt az algoritmust használva nem jöhet létre. Így az eljárás egy ellenőrzés segítségével mindig jó eredményt adóvá alakítható.

6.2. SEGÉD HASH TÁBLA HASZNÁLATA, KULCSOK TÁROLÁSÁVAL

Megtehetjük, hogy külön hash táblát [4] használunk a döntetlen állások felismerésére. Ez megvalósítható egy 2^e elemű tömbbel. A tömbben m bites számokat tárolhatunk. Ekkor a hash kulcsok mérete $e+m$. Egy állás behelyezésekor a hash kulcs első e db. bitje (elsődleges kulcs) határozza meg, hogy a második, m db. bitet (másodlagos kulcs) a tömb melyik elemébe írhatjuk. Az olvasás hasonló módon történhet. Minden új állás esetén megvizsgáljuk, hogy az állás már benn van-e a hash táblában. Ha nincs, akkor behelyezzük a másodlagos kulcsot az elsődleges kulcs által meghatározott helyre, ha megtaláltuk, akkor feltételezhetjük, hogy állásismétlés történt. A játékfa aktuális állásából induló részfájának megvizsgálása után a hash táblának a megadott eleméből törölhetjük az állást.

Az algoritmus nem igényel nagy méretű hash táblát, hiszen átlagosan 2-6 állás tárolódik benne, és a legritkább esetben fordul csak elő, hogy $99 + \text{keresési mélység}$ -re növekszik a tárolt állások száma, ami az 50 lépés szabálya miatti felső korlát. A korlát azért több 99-nél, mert hash táblából csak egyet tárolunk a memóriában, nem érdemes minden ütközési osztályhoz külön-külön újat nyitni, főleg az átlagos memóriapazarlás.

Az algoritmus ebben a formájában konstans időben lefut. Viszont elég sok hibalehetőség jöhet létre.

Vizsgáljuk meg azt az esetet, hogy ha egy állás másodlagos hash kulcsa megegyezhet az üres tömbelemet jelző számmal! Ekkor az állások egy nagyon kis halmazát általában döntetlen állásnak fogjuk tartani. A halmaz számossága fordítottan arányos a másodlagos hash kulcsok lehetséges értékeinek számával. Ha meg akarjuk különböztetni az üreset jelző tömbelemet, akkor azt Zobrist-hash-elés esetén a legegyszerűbben úgy tehetjük meg, ha a másodlagos kulcs egy bitjét rögzítjük. Ekkor azonban a másodlagos kulcs ütközésének esélye kétszeresére növekedik.

Előfordulhat, hogy két állásnak az elsődleges kulcsa (az első e db. bit) megegyezik. Erre az esetre nyílt címzést [4] alkalmazhatunk. Az ilyen ütközések ritkasága miatt ez a módszer nem jelent jelentős futásidő növekedést. Feltéve, hogy az egyes egymást követő állások elsődleges hash kulcsai véletlenszerűek, az ütközések valószínűségét számoló képlet a következő:

$$\frac{P(n, k)}{n^{k-1}}, \quad \text{ahol } P(n, k) = n^{k-1}, \quad n \in [1, k-1]$$

A képletben n jelenti a tömbméretet, míg k a tömbben szereplő nemüres elemek számát. A $P(n, k)$ n -re nézve $k-2$ fokú konstans együtthatós polinom, így $k-1$ db. helyen felvett értéke egyértelműen meghatározza. Kiszámítása Newton-, vagy Lagrange-interpolációval [10] történhet.

A képletet megvizsgálva kapjuk, hogy például $n = 2^{14} = 16384$, és a szélsőségeshez közeli $k = 99$ esetén az ütközés valószínűsége 0,257, illetve $k = 1, 2, \dots, 6$ esetén 0,001 alatti érték. Szélsőséges k értéket nem tudok mondani, mivel sakkprogramok különböző mélységekben képesek keresni. Viszont a $k > 99$ esetek a gyakorlatban nagyon ritkán fordulnak elő, és abban az esetben is csak egy minimális lassulás várható az elsődleges kulcsok ütközésvalószínűségének növekedése miatt.

Ha el szeretnénk várni az algoritmustól, hogy állásismétlések esetén feltétlenül jelezzen, akkor a nyílt címzést feltétlenül alkalmaznunk kell, illetve biztosítanunk kell, hogy a másodlagos hash kulcs különbözzön az üres tömbelemtől. Ha nem alkalmazunk nyílt címzést, az azt eredményezi, hogy ha két azonos állás közötti lépéssorozat során egy harmadiknak megegyezik az elsődleges kulcsa a másik kettőével, viszont a másodlagos kulcs különböző, akkor nem felismerhető az állásismétlés. Ha az üres tömbelemet nem különböztetjük meg a másodlagos hash kulcsoktól, akkor hasonló probléma jöhet elő az üres elemmel azonos hash kulcsú állásoknál. Az ellenőrzés szempontjából a fő problémát ugyanis nem az jelenti, hogy az üres mezőt állás hash kulcsának nézzük, és így több esetben érezzük állásismétlésnek, ami valójában nem az.

Azzal, hogy a másodlagos kulcs különbözik az üres tömbelemtől, kaptunk egy eljárást, amit utólag ellenőrizhetünk viszonylag kevés ráfordítással, azonban több hibát ejt, mint az üres tömbelemeket nem megkülönböztető, szintén nyílt címzést használó algoritmus. Annak a hibái azonban nem javíthatók.

A megvalósítás, nem ellenőrizhető esetre:

A programomban két változatot írtam meg. Az első algoritmus 14 bites elsődleges, illetve 16 bites másodlagos kulccsal dolgozik. Nem különbözteti meg az üres, illetve a nemüres hash tábla mezőket. Az elsődleges kulcs generálása a következőképpen történik:

$E_{0,zobrist} = C$, ahol C egy adott 14 bites konstans. Az elsődleges kulcsban található 0 index jelenti, hogy ez az érték a minimax algoritmus futásának elindulásakor az utolsó ütközési osztály első állásának hash kódja. A 'zobrist' fogja jelezni, hogy a kódot a zobrist-hasheléssel valósítottam meg. Ekkor az i -edik állás hash kódja az i -edik állás zobrist hash kódjából:

$$E_i = E_{i,zobrist} \text{ xor } \left(\left\lfloor \frac{i}{2} \right\rfloor \cdot D \right)$$

A D itt szintén egy adott 14 bites konstans. Ezzel biztosítottam, hogy két azonos állás esetén, amikor különböző felek vannak lépésen, akkor az algoritmus ne okozzon elsődleges kulcsütközést.

A másodlagos kulcs 16 bites, ahol $M_0 = K$ adott 16 bites konstans. A többi állás esetén zobrist-hash-eléssel kaphatjuk meg a másodlagos kulcsot. Itt már annyira nem szükséges foglalkozni azzal, hogy két azonos állás esetén, melyek csak a lépő félben különböznek, a másodlagos hash kulcsuk megegyezik, hiszen az elsődleges kulcsuk különböző, így jó messzire kerülnek egymástól a hash táblában.

Új állás hash-táblába helyezés a nyílt címezés alapján történhet. A törlés esetén a legegyszerűbb módszer pont onnan törölni a másodlagos kulcsot, ahova behelyeztük.

A megvalósítás, ellenőrizhető esetre:

Az ellenőrizhető eset abban különbözik a nem ellenőrizhetőtől, hogy a másodlagos kulcs csak 15 bit méretű. Ekkor a felszabadult egy bit jelezheti, hogy van állás az adott tömbben.

Ha állásismétlést vett észre a hash táblás algoritmus, de az ellenőrzés szerint mégsem történt állásismétlés, akkor nem kerül be új állás a hash táblába, így az állás elemzése végén nem kell törölni belőle.

6.3 SEGÉD HASH TÁBLA HASZNÁLATA, ÁLLÁSOK DARABSZÁMÁVAL

A következőkben bemutatott algoritmus leírása [6]-ból származik.

Megtehetjük, hogy a hash táblában a másodlagos kulcs helyett egy számlálót tárolunk. Ekkor nem használhatunk nyílt címzést, és kizárólag elsődleges kulcsütközésről beszélhetünk. Kezdetben legyen 0 a hash tábla minden eleme! Ha egy állás létrejött, akkor a hash kulcsához tartozó tömbelemet kell megvizsgálni. Ha a tömbelem 0, akkor az adott kulcshoz még nem tartozik állás, biztosan nem jött létre állásismétlés. Ekkor növeljük a tömbelem értékét 1-gyel! Ha a tömbelem nem 0, akkor elképzelhető, hogy létrejött állásismétlés. Ezt további ellenőrzések támaszthatják alá, vagy cáfolhatják. Ha mégse történt, akkor növelnünk kell a kulcshoz tartozó tömbelem értékét 1-gyel. A játékfa állásból kiinduló részfájának elemzését követően az állás kulcsához tartozó elem értékét csökkenthetjük 1-gyel. Ekkor a számlálók értéke gyakorlatban nemigazán közelíti a 255-öt, így a tömbelemek egy bájtól tárolhatók, ehhez ugyanis nagyon nagy mélységű keresés lenne szükséges, ami a jelenlegi számítógépekkel kivárhatatlan lenne.

A megvalósítás:

Tesztelő sakkprogramban a hash tábla 2^{14} elemet tartalmaz. Az ellenőrzést a 7. fejezetben leírt algoritmusom biztosítja.

7. AZ ÚJ ALGORITMUS

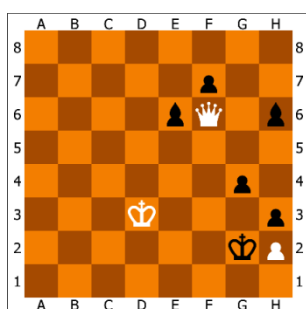
A dolgozatomban keretében egy olyan algoritmust mutatok be, amely bizonyíthatóan helyes megoldás a problémára, viszont, gyorsabb futásidőt biztosít, mint a többi, helyes eljárás.

7.1 AZ ALGORITMUS ÖTLETE

Adottak az utolsó ütközési osztály állásai közötti lépések (lépő bábu, kiinduló mező, célmező) hármassokkal, és az aktuális állás. Induljunk el visszafelé az aktuális állástól a lépések mentén, mint az Axon-ban alkalmazott módszer esetén!

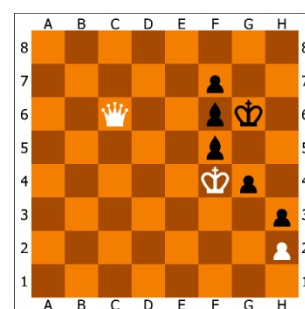
Az algoritmusom az Axon-nal ellentétben felhasználja a lépések mellett az aktuális állást is. Ez sem jelent fölösleges memóriamozgatás szükségletet, hiszen az eljárás futása során a memóriában tárolt sakktábla épp az aktuális állást tárolja. Ennek segítségével megoldható, hogy az Axon-ban alkalmazott bonyolult adatszerkezettel ne kelljen foglalkozni, elég legyen minden egyes visszalépésnél egy számot csökkenteni, vagy növelni.

Nevezzük **más helyen álló bábuk** számának azt az értéket, ahány bábu olyan helyre kerül a visszalépegetések során, amelyik mezőn nem ugyanolyan típusú bábu áll az aktuális állásban. Ez a definíció majdnem megegyezik a nyílt körök számának definíciójával. A különbség csak akkor jelentkezik, ha az egyik bábu egy vele azonos típusú bábu aktuális állásban elfoglalt helyére lép a visszalépegetések során. Ekkor ugyanis a más helyen álló bábuk száma 1-gyel csökken. Az Axon-ban alkalmazott, nyílt körök száma azonban nem csökken, csak akkor, ha két azonos típusú bábu befejezte a helycseréjét.



aktuális állás

...91 visszalépés...
(45 és fél teljes lépés)



visszalépések után kapott állás

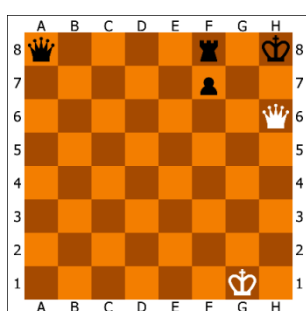
10. ábra

A két állás között a más helyen álló bábuk száma 5, a futók, a királyok és a vezér
(Forrás: Rebel Century – Loek van Wely, Maastricht, 2002. február 19.)

A más helyen álló bábuk száma kezdetben 0, hiszen az aktuális állás és önmaga közötti más helyen álló bábuk nincsenek. Minden egyes visszalépés során könnyen módosítható az érték, hogy a visszalépés utáni állás és az aktuális állás közötti más helyen álló bábuk számát kapjuk meg. Ez két feltételes növeléssel, illetve csökkentéssel megoldható.

Ha a más helyen álló bábuk száma 0 lesz, akkor létrejött újra az aktuális állás, hiszen ez azt jelenti, hogy minden bábu ugyanott áll az aktuális, és a visszalépegetés utáni állásban. Ezután már csak azt kell ellenőriznünk, hogy ugyanaz a fél jött-e lépésen, azaz, a visszalépések száma páros-e. Ha ez is teljesül, akkor állásismétlést találtunk.

Nézzük meg, hogyan is működik az algoritmus, a korábbi minta lépéssorozat esetén!



11. ábra

Az aktuális állás

A visszalépések sorozata az ütköző állásig:

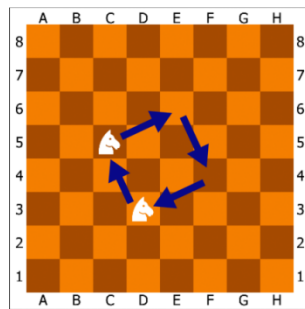
V H6-G5 , K H8-G7 , V G5-H5 , K G7-H7 , V H5-G5 , K H7-G8 , V G5-H6 , K G8-H8 , V H6-C1

Ekkor az algoritmus futása:

Megtett visszalépés	Más helyen álló bábuk száma a visszalépés után
(aktuális állás)	0
V H6-G5	1
K H8-G7	2
V G5-H5	2
K G7-H7	2
V H5-G5	2
K H7-G8	2
V G5-H6	1
K G8-H8	0

Tovább már nem szükséges vizsgálódnunk, hiszen már felismertük az állásismétlést.

A következő példában láthatjuk, hogy az algoritmus felismeri két bábú helycseréjét is:



12. ábra

Két huszár helycseréje például a következő lépéssorozattal:

1. **C5-E6** (+1)
2. **E6-F4** (nincs változás)
3. **D3-C5** (+1,-1)
4. **F4-D3** (-1)

7.2 AZ ALGORITMUS TÖMBINDEXES LÉPÉSGENERÁLÁS ESETÉN

Tömbindexes lépésgenerálás esetén a sakktábla-reprezentáció egy **tabla[]** tömbként fogható fel, melynek i -edik eleme megadja, hogy az adott állásban az állás valamilyen sorrendben vett i -edik mezején milyen bábú áll. Az algoritmus futásakor ebben a tömbben az aktuális állás található. Legyen a **lepesek[]** tömb a lépéseket tároló tömb, melynek tetejét a **teto** változó jelzi. C stílusú programozás kedvéért ezt értsük úgy, hogy a tömböt 0-tól indexeltük, és az utolsó nemüres elemének indexe $teto-1$. elem. A tömb nagyobb indexű elemeiben tárolhatjuk a később megtett lépéseket, így a tömb módosítása a vermekhez hasonlóan történik. Legyen az **utkozo** annak a lépésnek amely a tömbben lévő legutolsó visszafordíthatatlan lépés. Azaz, a visszalépések folyamán legfeljebb eddig a pontig kell visszamennünk az állásismétlés észlelésével, az **utkozo** által jelzett visszalépést már nem kell megtennünk. Az egyes lépéseket az állásismétlés szempontjából egy **(bábú,start,cel)** hármasban tárolhatjuk, azaz a bábú típusát, illetve a lépés két végmezejét. A lépések start mezeje a visszalépések célmezeje, illetve a lépések célmezeje a visszalépések start mezeje. Hogy az algoritmusban ezzel a problémával ne kelljen foglalkozni, ezért a start és cél mezők jelentsék a visszalépésre vonatkozó kiindulási-, és célmezőket. Az eljárásban a **szam** változó tárolja a más helyen álló bábuk számát.

A fenti jelölések segítségével az algoritmus legegyszerűbb változata a következőképpen néz ki:

```
bool allasismetles()
{
    int szam=0;

    for(int poz=teto-1;poz>utkozo;poz++){

        if(tabla[lepesek[poz].start]==lepesek[poz].babu) szam++;
        if(tabla[lepesek[poz].cel] ==lepesek[poz].babu) szam--;

        if(szam==0 && (poz+teto)%2==0) return true;

    }

    return false;
}
```

Kezdetben a más helyen álló bábuk száma természetesen 0. Ezt követően, módosítjuk a más helyen álló bábuk számát. Ha a visszalépés során a más helyen álló bábuk száma újra 0 lesz, akkor az aktuális állásban látható bábuk helyzete megismétlődött. Az állásismétlés feltétele azonban az is, hogy ugyanaz a fél jöjjön lépésen. Ezt biztosítja a ciklusmagban található harmadik elágazás feltétele.

Vizsgáljuk meg, hogy az algoritmus tényleg azt teszi-e, amit elvárunk tőle, azaz, pont akkor lesz 0 a más helyen álló bábuk száma, amikor a visszalépésben elért korábbi állás a bábuk helyzetét tekintve megegyezik az aktuális állással.

Kezdetben az aktuális állás önmagától nem különbözik, tehát a más helyen álló bábuk száma 0. Ezt követően vizsgáljuk meg a ciklusinvariánst, hogy megtartja-e az elvárt tulajdonságot? Négy eset lehetséges:

1. A visszalépésnek sem a kiinduló mezején, sem a célmezején nem olyan típusú bábu áll az aktuális állásban, mint a lépő bábu. Emiatt a visszalépés során a más helyen álló bábuk száma nem változhatott. A ciklusban az első két feltétel közül egyikbe sem tudunk ilyen esetben belépni, így valóban nem változik *szam* értéke.
2. A visszalépésnek a kiinduló mezején ugyanolyan típusú bábu áll, mint a lépő bábu, azonban a célmezőn más típusú bábu áll, esetleg üres a mező. Ekkor a visszalépés során a bábu elmozdul az aktuális állásban elfoglalt helyéről. Így, növekszik eggyel a más helyen álló bábuk száma. A ciklusban pont az első elágazás feltétele teljesül, így *szam* értéke valóban nő eggyel.

3. A visszalépésnek a célmezején ugyanolyan típusú bábu áll, mint a lépő bábu, azonban a kiinduló mezőn nem. Ebben az esetben a visszalépés során a bábu olyan mezőre lép, ahol amúgy is áll az aktuális állásban olyan típusú bábu. Ezáltal a más helyen álló bábuk száma csökken eggyel. Valóban, a ciklusban a második feltétel teljesül, így *szam* értéke csökken.
4. A visszalépés kiinduló-, és célmezején is ugyanolyan típusú bábu áll, mint a lépő bábu. Ebben az esetben a 2., és 3. eset egyszerre zajlik le. Azaz a más helyen álló bábuk száma nem változik, hiszen a lépés előtt és után is ugyanúgy a típusának megfelelő bábu áll az általa elfoglalt mezőn az aktuális állásban. A ciklusban mind az első, mind a második elágazás feltétele teljesül, így *szam* értéke végeredményben valóban nem változik.

A fentiek során beláthattuk, hogy a ciklus futása során a más helyen álló bábuk száma helyesen változik. Ezt követően az algoritmus helyességét nem nehéz megmutatni.

A megvalósítás a fenti formájában nem a leghatékonyabb. Némi optimalizálás után a következő változatot kaphatjuk:

```
bool allasismetles()
{
    if(utkozo>=teto-4) return false; // (1)

    int szam=0;
    int poz=teto-1;

    while(poz-utkozo>=nagyobb(szam,2)) { // (2)

        if(tabla[lepesek[poz].start]==lepesek[poz].babu) szam++; // (3)
        if(tabla[lepesek[poz].cel] ==lepesek[poz].babu) szam--;

        // (4)

        if(tabla[lepesek[poz-1].start]==lepesek[poz-1].babu) szam++;
        if(tabla[lepesek[poz-1].cel] ==lepesek[poz-1].babu) szam--;

        if(szam==0) return true; // (5)

        poz=poz-2;
    }

    return false; // (6)
}
```

Az (1)-ben az ellenőrzés azt vizsgálja, hogy érdemes-e elkezdni futtatni az algoritmust. Állásismétléshez legalább négy féllépésre lévő állások között képzelhető el. Ehhez mind a két

félnek egy-egy oda-vissza lépésére van szükség. Ezt az ellenőrzést elég sok sakkprogram megteszi.

A ciklusmagban (3) nem egy, hanem két visszalépést tehetünk meg egyszerre. Hiszen, ha állásismétlés jön létre, akkor a két azonos állás között biztosan páros számú féllépés található, ez biztosítja, hogy mindkét állásban ugyanaz a fél jön lépésen.

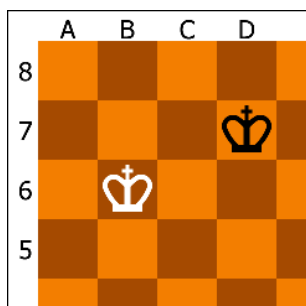
A (5) pontban kiugorhatunk a ciklusból, ha állásismétlést tapasztaltunk. (6)-be akkor kerülünk, ha befejeződött a ciklus, de nem vettünk észre közben állásismétlést.

A (2) pontban elég érdekes a ciklusfeltétel. Nézzük meg külön-külön, hogy a különbségnek mért kell mind 2-nél, mind *szam*-nál nagyobbnak lennie?

Nézzük a *szam* esetét! Ez a feltétel azt köti ki, hogy a további visszalépegetések számának legalább annyinak kell lennie, mint a más helyen álló bábuk száma. Ezt a kikötést simán megtehetjük, hiszen ha több lenne a más helyen álló bábuk száma, mint a megmaradt visszalépések száma, akkor a visszafordíthatatlan lépésig nincs elég visszalépés a más helyen álló bábuk számának 0-ra csökkentésére, hiszen ez az érték egy visszalépés esetén legfeljebb 1-gyel csökken. Ezt a gyorsítást az Axon-ban is alkalmazzák.

2-nél azért kell nagyobbnak lennie, hogy a ciklus minden esetben hibátlanul lefusson. Vizsgáljuk meg, lényeges-e ez a feltétel?

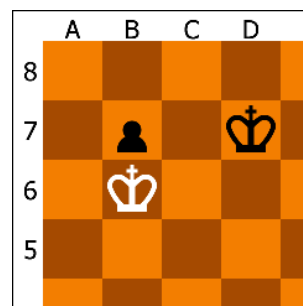
A ciklusba való belépéskor a különbség az első elágazás miatt legalább 3, így a 2-vel való összehasonlítás fölösleges. Mivel *szam* értéke mindig pozitív, és ha a ciklusmag végén 0 volt, akkor már újra nem vizsgáljuk a feltételt, ezért a különbség 1-nél sosem kisebb. Azonban, ha elhagynánk, a 2-vel való összehasonlítást, akkor a ciklusmagban a második visszalépés megtételekor (4 után) előfordulhatna, hogy a visszafordíthatatlan lépést vonjuk vissza. Azonban, ezzel lehet, hogy alulindexeltük a tömböt, mert például visszafordíthatatlan lépés nincsen, csak -1-gyel jeleztük, hogy most már itt van a tömb alja, például az alapállás esetén. A visszafordíthatatlan lépés lehet speciális lépés is, amit egész máshogy tárolunk, mint az egyszerű lépéseket, ami szintén hibát generál. Illetve, az is előfordulhat, hogy a visszafordíthatatlan lépés ütés volt. Ebben az esetben a visszafordíthatatlan lépés visszavonása könnyen eredményezhetne állásismétlést.



aktuális állás

(Sötét király, D7, D6)
 (Világos király, B6, B7)
 (Sötét király, D6, D7)
 (Világos király, B7, B6)

Az algoritmus által látott
visszalépések



négy féllépéssel ezelőtt

13. ábra

A visszafordíthatatlan lépések belekerülésének hatása az algoritmusba

Ahogy látjuk, itt a legelső visszalépéskor (a visszafordíthatatlan lépésnél) a visszalépésnek ez a 3-asban megadott módja nem elegendő arra, hogy felismerjük, hogy a lépés visszavonása esetén egy gyalogot is fel kell helyezni a sakktáblára. Emiatt az algoritmus a példaállásban állásismétlést észlelne, ha nem figyeljük a ciklusfeltételben, hogy a különbség 2-nél nagyobb legyen. Hasonló esettel találkozhatunk, ha a visszafordíthatatlan lépés en passant lépéslehetőség miatti visszafordíthatatlan lépés, vagy sáncolás lehetőség elvesztését okozó lépés.

A 2-vel való összehasonlítást el lehet hagyni a fenti problémák megoldásával. Ha nincs visszafordíthatatlan lépés az ütközési osztály elején, akkor is rakjunk egy kvázi visszafordíthatatlan lépést a *lepések* tömb alá. Azoknak a visszafordíthatatlan lépéseknek a helyére, amelyek speciális lépések, találjunk ki egy olyan lépést jelző hámaszt, ami biztosítja, hogy véletlenül se történjen a lépés visszavonásakor alul-, vagy felülindexelés, azaz, a kiindulási, és a célmező indexe is legyen értelmes index a sakktábla-reprezentációban. Ismerve, hogy *szam* nemnegatív a ciklus (4) pontjában, elképzelhető, hogy az értéke itt 0. Ezért, jó lenne az összes visszafordíthatatlan lépést egy olyan hármassal jelezni, amelyik biztosítja, hogy nő a *szam* változó értéke. Ez biztosítaná azt, hogy visszafordíthatatlan lépés vizsgálata után ár biztosan ne észleljünk állásismétlést. Megoldaná a 13. ábrán látható problémát is.

A sakkprogramokban gyakori, hogy a sakktáblát nem 64 elemű tömbbel, hanem egy nagyobb tömbben tárolnak ([2], bővebben [11]), ahol a sakktábla szélét adott **FAL** konstansokkal jelzik. Ekkor indexeljen a **falindex** a sakktábla-reprezentációban egy fal elemet, míg a **tablaindex** legyen a sakktábla egy tetszőleges mezejét azonosító érték. Ha a programunk sakktábla-reprezentációja nem tartalmaz ilyen szegély-mezőket, akkor növeljük a sakktáblát reprezentáló töm méretét 1-gyel több mezőre. Ekkor az így kapott 65. mező lesz az

algoritmushoz szükséges szegélymező. Feltételezve, hogy a sakkállást tároló tömböt nem fogjuk másolgatni, ez a módosítás semmilyen lassulást nem fog eredményezni. Ekkor a **(FAL,falindex,tablaindex)** hármass biztosítja, hogy ennek a visszalépésnek a megtételével nőjön a más helyen álló bábuk száma, hiszen itt nem történik más az algoritmus szerint, minthogy egy „FAL bábu” ellép az aktuális állásban elfoglalt helyéről egy olyan mezőre, ahol biztos nem áll az aktuális állásban FAL bábu.

A fenti módszerek alkalmazásával a ciklusfeltételből elhagyható a különbség 2-vel való összehasonlítása. A felsorolt problémákat azonban nem minden lépésgeneráló algoritmus esetén lehet, vagy érdemes kijavítani, elképzelhető, hogy jobb futásidőt kapunk egyes esetekben, ha bennhagyjuk a 2-vel való összehasonlítást a ciklusfeltételben.

7.3 AZ ALGORITMUS BITTÉRKÉPES LÉPÉSGENERÁLÁS ESETÉN

A következő sorokban bemutatom, hogy bittérképes sakktábla-reprezentáció esetén hogyan módosíthatjuk az algoritmust. Aki nem jártas a bittérképes lépésgenerálásban, az [7]-ben találhat egy jó leírást egy ilyen módszert alkalmazó sakkprogramról.

Az eljárás tetszőleges (pl. bittérképes) sakktábla-reprezentáció esetén hasonlóan képzelhető el, mint tömbindexes esetben. Vizsgáljuk a ciklusmagban egy visszalépés vizsgálatát! Ez két feltételes értékadást tartalmaz. Az egész algoritmusban egyedül ez a két feltétel használja ki a tömbindexes sakktábla-ábrázolást.

Általános sakktábla-reprezentáció esetén egy megfelelően paraméterezett **feltetel()** függvényt írhatunk az elágazás feltételébe, ahol:

$$\text{feltetel}(\mathbf{babu}, \mathbf{mezo}) = \begin{cases} \text{igaz, ha az aktuális állásban a } \mathbf{mezo} \text{ mezőn } \mathbf{babu} \text{ típusú bábu áll} \\ \text{hamis, különben} \end{cases}$$

Ekkor az i-edik visszalépés algoritmus a következőképpen képzelhető el, általánosan:

```
if (feltetel (lepesek[i].babu, lepesek[i].start)  szam++;  
if (feltetel (lepesek[i].babu, lepesek[i].cel)    szam--;
```

Így, tetszőleges sakktábla-reprezentáció esetén elég megadnunk a megfelelő feltetel() függvényt az állásismétlés észrevételéhez.

Bittérképes lépésgenerálás esetén a sakktábla tárolható egy **tabla[12]** tömbben, amelynek minden eleme egy bittérkép. Az i-edik bábutípus által elfoglalt mezőket a **tabla[i]** bittérkép magas értékű bitjei jelzik. Legyen **egybit()** egy olyan függvény, amely tetszőleges mezőindexre megad egy olyan bittérképet, amelynek egyedül az indexhez tartozó bitje 1. Ez a függvény egy 64-elemű bittérkép tömbbel, vagy esetleg egy shl (<<, shift left) bittologató

utasítással is megadható. Az algoritmusban az $\&$ (és) művelet a bitenkénti és műveletet jelenti.

```
bool feltetel(int babu,int mezo)
{
    return (tabla[babu]&egybit(mezo))!=0;
}
```

A ciklusfeltételből elhagyható a 2-vel való összehasonlítás, ha a *tabla* tömböt kiegészítjük egy újabb, **UTKOZO** indexű bittérképpel. Ekkor az visszafordíthatatlan lépések az (**UTKOZO**,*utkozo_igaz*,*utkozo_hamis*) hármassal jellemezhetők. A *tabla*[**UTKOZO**] bittérképet hozzuk úgy létre, hogy az *utkozo_igaz* mezőhöz tartozó bitje legyen 1, míg az *utkozo_hamis* mezőhöz tartozó bitje legyen 0. A többi bit értéke tetszőleges lehet. Ez a feltétel biztosítja az visszafordíthatatlan visszalépések esetén a más helyen álló bábuk számának növekedését, hisz a feltétel függvény az első esetben igaz, míg a második esetben hamis értékkel tér vissza.

7.4 EGY HASONLÓ ALGORITMUS BITTÉRKÉPES LÉPÉSGENERÁLÁS ESETÉN

Bittérképek esetén egy másik, hasonló algoritmus is adható a probléma megoldására. Az algoritmus felépítése megegyezik a korábban bemutatott algoritmussal, azonban a visszalépést kezelő algoritmus módosul.

Az algoritmusban a **szam** változó elképzelhető egy 12 elemű bitvektorként. Ha két állásban az *i*-edik típusú bábuk nem pont ugyanott állnak, akkor *szam* *i*-edik bitje legyen 1, különben legyen 0. Látható, hogy ha *szam* értéke 0-vá válik, akkor mindegyik bábutípus összes bábuja pont azokat a mezőket foglalja el mindkét állásban. A bitvektorban a magas értékű bitek száma pedig megadja azokat a bábutípusokat, amelyek nem ugyanazokat a mezőket foglalják el a két állásban.

A lépések tömbjében ne a korábban definiált hármassokat, hanem (**babu**,**bitterkep**) párokat tároljunk. Az adott lépésben *babu* lépett, és a lépése előtt (a visszalépés után) a bábutípusának megfelelő bittérkép a *bitterkep* volt. Ha egy visszalépés után a lépő bábu bittérképe más, mint az aktuális állásban, akkor legyen 1 a bitvektorban a bábuhoz tartozó bit, különben pedig legyen 0. A $|$ (vagy), illetve a $\wedge$ (kizáró vagy) utasításokat szintén bitenként kell értelmezni.

Tehát, az i -edik visszalépés algoritmus:

```
if (tabla[lepesek[i].babu] != lepesek[i].bitterkep) {  
    szam = (szam | (1 << babu));  
} else {  
    szam = (szam ^ (1 << babu));  
}
```

Az algoritmusban, ha a feltétel igaz, akkor *szam* *babu.* bitje 1 lesz. Ha a feltétel hamis, akkor a *szam* *babu.* bitje megváltozik. Ez nem ugyanazt csinálja, mint amit mi szeretnénk, hiszen mi az else-ágban azt szeretnénk, hogy a *babu.* bit 0-vá váljon. Tudjuk azonban, hogy a *babu* típusú bábuk közül egy tette meg a visszalépést, és így megegyezik a visszalépések során a bábutípushoz tartozó bittérkép az aktuális állásban tároltával. Mivel tetszőleges lépés esetén megváltozik az adott típusú bábukhoz tartozó bittérkép, így a visszalépés előtt még különböznie kellett az aktuális állásban tároltával. Ezért feltételezhetjük, hogy az else-ágba csak akkor kerülhettünk, ha korábban a *szam* *babu.* bitjét már 1-re állítottuk.

A korábbi megvalósításokhoz képest itt elég volt egy összehasonlítás az i -edik visszalépés vizsgálatakor.

Az állásfelismerő algoritmusban a ciklusfeltételt is módosítani kell, hiszen *szam*-ot nem számként, hanem bitvektorként értelmeztük. Ebben az esetben nehezebb feladat összehasonlítani a különbséget a *szam*-ban található magas bitek számával. Egy lehetséges megoldás, hogy a különbséget csak 2-vel hasonlítjuk össze. Ekkor azonban el kell tekintenünk a vágó heurisztikától, és minden, nem ismétlődő állás esetén a visszalépetést a visszafordíthatatlan lépésig teljesen meg kell tenni. Egy másik lehetséges megoldás, hogy egy **bitszam** változóban folyamatosan tároljuk, hogy a *szam* bitvektor hány eleme 1. Ez az érték a visszalépés algoritmusában a feltétel teljesülésekor változatlan maradhat, vagy nőhet, míg az else-ágban biztosan csökken. Így a feltétel teljesülésekor egy újabb feltételt kell megvizsgálni ahhoz, hogy eldöntsük, hogy változik *bitszam* értéke. Ekkor a ciklusfeltételben a *szam* helyett *bitszam*-ot kell összehasonlítani a különbséggel.

8. A MÁS HELYEN ÁLLÓ BÁBUK SZÁMÁNAK FELHASZNÁLÁSA

Az előző fejezetben leírt algoritmus által kiszámolt, más helyen álló bábuk száma nem csak állásisméltés felismerésre használható fel. A fejezetben bemutatom, hogy hogyan lehet a programunk értékelőfüggvényét pontosítani az állásisméltés felismerő eljárás által kiszámolt, más helyen álló bábuk számának felhasználásával.

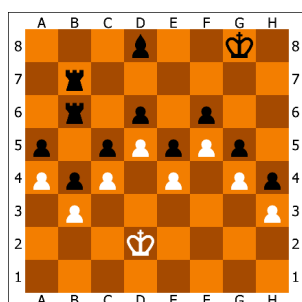
A más helyen álló bábuk száma egy távolságfüggvényt definiál tetszőleges két állás között, amelyek egy ütközési osztályba tartoznak. Az állások távolságát pedig fel tudjuk használni a megnyitásban, illetve a végjátékban is az értékelőfüggvény korrigálására. Azonban figyelniünk kell arra, hogy a távolságfüggvény 0-val tér vissza két A és B állás esetén is, amikor a két állásban ugyanott állnak a bábuk, csak a másik fél jön lépésen. Ez a két állás valójában nagyon messzire, legalább 5 féllépésnyire van egymástól, és az egyes felek helyzetét is jelentősen befolyásolja, hogy ki léphet az állásban. Ezért, sajnos nem teljesül rá a távolságfüggvények minden elvárt tulajdonsága. Azonban, általában mégis mond valamit arról, hogy két állás milyen messzire van egymástól.

Megnyitásoknál a fő szempont, hogy a könnyű tiszték mielőbb kikerüljenek a gyalogok mögül, illetve megtörténjen a király biztonságba helyezése. Ha a gyalogok lépéseit nem nézzük, akkor is látható, hogy általában egy bábuval ritkán kell többet is lépni egymás után, viszont elég sok bábut meg kell mozdtítani a jó megnyitásban. Ezért, az értékelőfüggvény kihasználhatja a megnyitásban, hogy az aktuális állás és az alapállás között mennyi a más helyen álló bábuk száma. A gyaloglépések miatt érdemes az állásisméltés felismerő algoritmus mintájára egy másik eljárást létrehozni, ami a gyalogok lépéseit is figyeli.

Végjátékokban gyakori, hogy egy állás, bár döntetlen, de ezt nehéz megállapítani viszonylag kis mélységű keresés során. Itt nyilvánvalóan az érdekel minket, hogy egy játékfában felbukkanó állás esetén megmondhassuk, érdemes-e arra vennünk az utat, vagy jobb lesz elkerülni. Akkor, ha hosszú lépéssorozatok során sem történt túl nagy változás, az állásban, akkor feltételezhetjük, hogy döntetlen fele igyekszik a lépéssorozat.

Mérhetjük az ütköző állás, és az aktuális állás közötti távolságot, illetve a két állás között megtett féllépések számát. Ha a távolság/lépésszám hányados kicsi, akkor az értékelőfüggvényben közelíthetjük a pontszámot a döntetlen pontszám felé, hiszen gyanús, hogy az egyik fél sem tud mit kezdeni a másikkal. A közelítés mértéke a hányadostól függ, minél kisebb a hányados, annál jobban közelíthetjük a pontszámot. Látható, hogy a hányados 0 is lehet, ha a két állásban a bábuk helyzete megegyezik. Ekkor, ha ugyanaz a fél jön lépésen,

akkor az állás pontszáma elérheti a döntetlen értéket, nem kell számolni az értékelőfüggvénnyel, illetve a további fakezelésekkel.



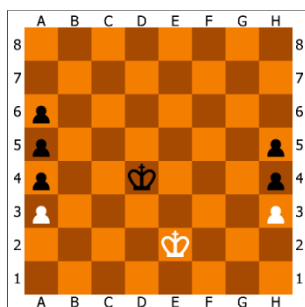
14. ábra

A világos, ha csak a királyával lép, akkor, bár a sötét fél meg tudja akadályozni az állásismétlődést, az állásból kiindulva 5 fölé nem engedi a távolságot, így a hányados $1/10$ -hez közelít.

Azonban, előfordulhat, hogy a két állás közötti távolságérték jelentősen más értéket jelez, mint ami valójában a két állás közötti távolság. Ezért, talán érdemes kisebb távolságokra egy javító táblázatot alkalmazni, amely korrigálja a más helyen álló bábuk számának ezt a hibáját. A táblázatot az alapján töltöttem ki, hogy legalább hány félépés kell az adott távolságok létrehozására.

távolság	korrigált távolság két állás között	
	ugyanaz a fél jön lépésen	különböző fél jön lépésen
0	0	5
1	4	1
2	2	3
3	4	3

Nagyobb távolságok esetén nincs jelentős különbség a távolságot létrehozó minimális félépésszám és a távolság között.



15. ábra

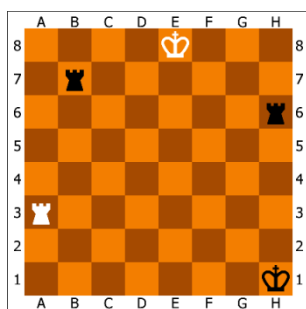
világos lép

A világos tartani tudja a legfeljebb 2 távolságot. Azonban, ha nem számolunk a korrigált távolsággal, akkor az ábrán látható állás létrejöhet úgy is, hogy a sötét lép, ami a világos veszét jelenti.

Akkor kapjuk meg a pontos távolságértéket, ha teljesen kiszámoljuk az állások közötti más helyen álló bábuk számát. Ezért, a ciklusnak az utolsó visszavonhatatlan lépésig le kell futnia az állásismétlés felismerő algoritmusban, végjátékok esetén, ahol a hosszú, ütésmentes lépéssorozatok miatt a legtöbb időt spórolhatnánk azon, hogy nem feltétlenül kell az összes visszalépést megtenni. Ezért, vagy meg kell elégednünk a néha pontatlan értékkel, ami nem feltétlenül okoz túl nagy hibát, vagy pontosan ki kell számolnunk a távolságértéket. Az ütköző

állás és az aktuális állás közötti távolság azonban konstans időben módosítható. Ha a sakktábla-reprezentáció nem túl nagy, akkor megtehetjük, hogy minden visszafordíthatatlan lépés után a memóriába másoljuk a létrejött állást, és ezután, pont fordítva, mint az állásismétlés felismerő algoritmusban, a megtett lépéseken előrehaladva számolhatjuk az ütköző állás és az aktuális állás közötti távolságot. Ezt az értéket minden egyes lépés megtételekor egy konstans művelettel tudjuk módosítani. Ekkor azonban csak egy adott korábbi állás, és az aktuális állás távolságát kapjuk meg, ami nem jelezne minden állásismétlést.

Néha előfordulhat, hogy néhány bábu mozgása során, bár a távolságérték alacsony marad sok lépésen keresztül, mégsem kósza oda-vissza lépegetésekről van szó. Ezeknek az eseteknek a felismerésére speciális heurisztikákra van szükség. Egy lehetséges módszer, hogy sakkadások esetén nem alkalmazzuk az algoritmust, illetve, hogy mérjük a király kiindulási mezejének változását a sakkok közben. Ezeknek a területeknek részletes tárgyalásával azonban nem foglalkozom a TDK dolgozatom keretein belül.



16. ábra
világos lép

Bár az elkövetkező sakkadások során a keresési mélység szempontjából nagyon sok lépésig nem nő a távolság 2 fölé, az állás ettől még nem döntetlen állás.

9. EREDMÉNYEK ELEMZÉSE

Miután az algoritmusom helyességét beláttuk, a következőkben vizsgáljuk meg a futásidejét!

9.1 A TESZTELŐ PROGRAM

Az megoldásom hatékonyságának mérésére egy tesztelő sakkprogramot írtam. A programomban tömbindexes lépésgenerálást alkalmaztam. Ennek megfelelően, azt a változatot tudtam tesztelni a programban, melynek kódja a *7.2 Az algoritmus tömbindexes lépésgenerálás esetén* című alfejezetben olvasható, az optimalizált változatot.

A programban beállítható paraméterek:

1. Keresési mélység:

Lehetőségünk van adott fix mélységű keresésre. Ezen kívül megadhatjuk a változó keresési mélységű keresés maximális mélységét. A programban a változó keresési mélységet úgy valósítottam meg, hogy ha valamelyik fél sakkot kap, akkor két féllépéssel növeljük a mélységet, míg a maximális keresési mélységet el nem érjük. Az állásismétlések szempontjából ez egy érdekes keresési mélység növelő algoritmus lehet, mert várhatóan kevés ütés van a fix mélységű keresésnél nagyobb mélységben létrejövő lépések között.

A keresési mélység megadásakor az (k,l) pár azt jelenti, hogy k mélységben biztosan minden ág végig lesz elemezve, de legfeljebb l mélységig vizsgálódik a minimax algoritmus.

2. Alfa-béta levágás:

A programban beállítható, hogy legyen-e alfa-béta levágás (ab), vagy sem (--). Ha van alfa-béta levágás, akkor az állásokban megtehető lépéseket a legfelső szint kivételével rendezve nézzük végig. A rendezés a lépés után létrejövő állások pontszáma alapján történik.

3. Állásismétlés felismerő algoritmus

Az állásismétlés felismerő algoritmusok közül a következőket valósítottam meg:

1. Nincs állásismétlés felismerés
2. Az általam javasolt algoritmus
3. Táblák összehasonlításával működő algoritmus
4. Az Axonban alkalmazott algoritmus
5. Hash kulcsok listája
6. Hash kulcsok listája az algoritmusom ellenőrzésével
7. Hash táblával működő, nem ellenőrizhető algoritmus
8. Hash táblával működő ellenőrizhető algoritmus
9. Hash táblával működő algoritmus az algoritmusom ellenőrzésével
10. Számlálókát tároló hash táblás eljárás, az algoritmusom ellenőrzésével

A szükséges hash kulcsokat a C++-ban elérhető `rand()` függvénnyel generáltam.

9.2 A TESZTELÉS MENETE

A tesztelés folyamán nagymesterek, illetve sakkprogramok játszmáiban létrejött állásokra futtattam a minimax algoritmust, különböző állásismétlés felismerő eljárásokkal. Eközben mértem az egyes megvalósítások esetén a minimax algoritmus futásidejét, illetve az esetleges hibázásait, és azok hatását a minimax algoritmus eredményére. Mértem külön az állásismétlés felismerő eljárások futásidejét is. Ehhez egy tömbbe kiírtam, hogy mikor vágott az algoritmus, majd újravéglettem a minimax algoritmust, ahol a tömbből olvastam ki a vágások helyét. A két minimax algoritmus futásidejeinek különbségeként megkapható az állásismétlés felismerés összfutásideje.

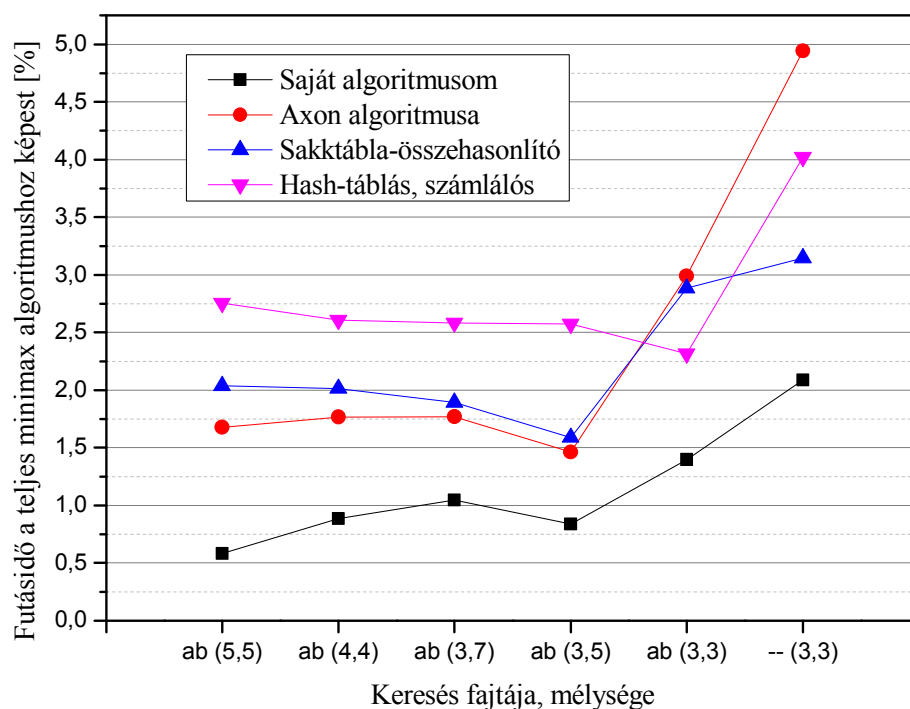
A 2010-ben Linaresben megrendezett nemzetközi sakkverseny (Linares International Chess Tournament) játszmáival teszteltem az eljárásokat. A verseny jellemzője, hogy átlagosan 40 lépésből álló játszmákat tartalmaz, azonban van benne 30 lépés alatt befejezett, illetve, 100 lépésnél hosszabb játszma is.

Ezeket a játszmákat kívül megvizsgáltam a 2002. február 19-én, Maastricht-ben megrendezett Rebel Century – Loek van Wely sakkmeccset. Ennek a játszmának érdekessége, hogy viszonylag hosszú, 105 lépésből álló játszma, amelynek a második felében kialakult egy 46 lépéses ütés-, és gyaloglépés-mentes szakasz.

9.3 MÉRT EREDMÉNYEK

Futásidő általános esetben:

A mindig pontos eredményt adó algoritmusok futásidejeinek a következő eredmények születtek:



17. ábra

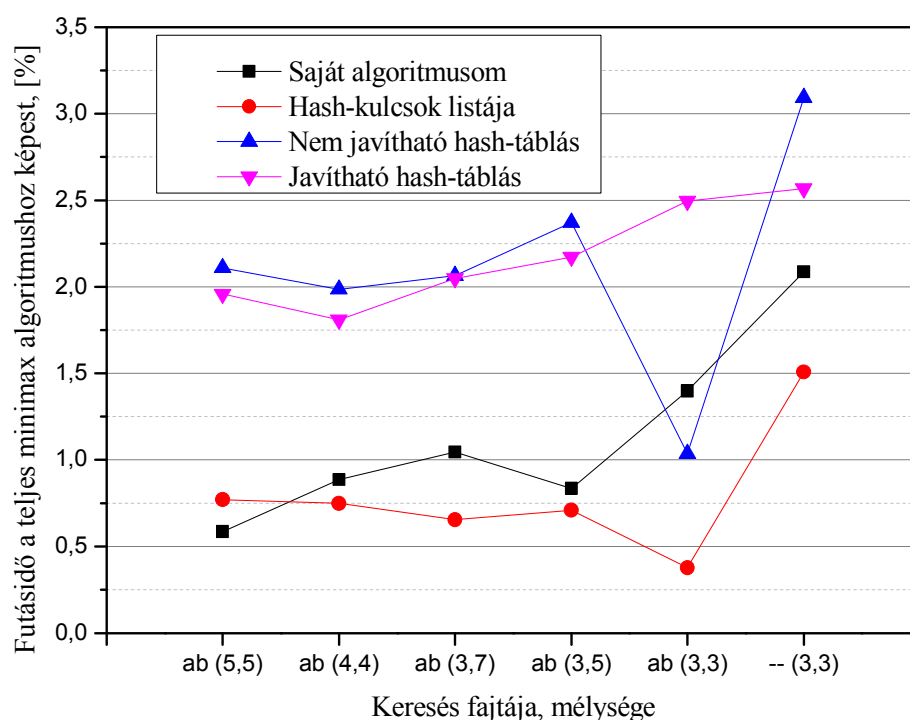
A mindig jó eredményt adó algoritmusok futásideje a minimax algoritmushoz képest
átlagos esetben, [%]

A grafikon alapján elmondható, hogy átlagosan az általam adott eljárás jelentősen jobb futásidőt biztosított, mint a többi, hasonlóan pontos eredményt adó algoritmus.

Az állásismétlés felismerés miatt létrejött futásidő-növekedés az állásismétlés felismerő eljárást nem használó minimax algoritmushoz képest:

ab (5,5)	0,39%
ab (4,4)	0,65%
ab (3,7)	0,48%
ab (3,5)	0,95%
ab (3,3)	1,25%
-- (3,3)	1,75%

Az algoritmusom futásideje, összevetve a nem mindig pontos eredményt adó megoldásokkal:



18. ábra

Az algoritmusom összevetése a nem mindig pontos eredményt adó, hash-kulcsokkal működő eljárásokkal

Az ábrán látható, hogy az esetek többségében úgy nem érdemes hash-kulcsokkal működő algoritmust beépíteni a programunkba, ha amúgy nem használnánk a hash-kulcsokat. Érdemes megjegyezni, hogy a hash-kulcsok listájában minden egyes ütköző állás hash kulcsa lehet 0, hiszen az 50 lépés szabálya miatt úgyis ismernünk kell, hogy hány féllépésig mehetünk vissza a hash-kulcsok összehasonlításában. Ezzel szemben egy kicsit bonyolultabb a visszafordíthatatlan lépések során a zobrist-hash-kulcsok generálása a hash-táblás

algoritmusokban. Talán ez lehet az oka, hogy azok az eljárások, bár átlagosan konstans futásidejűek, lényegesen lassabban futnak le, mint a lineáris futásidejű hash-kulcsok listája elvén működő algoritmus.

Ha a sakkprogramunknak amúgy is szüksége van a hash-kulcs generálásra, akkor magának az állásismétlés felismerő eljárásnak már nem kell újra kiszámolnia a hash-kulcsot, így jelentősen kisebb futásidővel tud működni. A mérések alapján elmondható, hogy a hash-táblát használó algoritmusok idejének kb. 2/3-a a hash-kulcs előállítására megy el. Ez alapján, jobb futásidővel tudnak működni, mint az általam javasolt megvalósítás. Az algoritmusom főciklusa az esetek túlnyomó részében nem fut le, mert a legutóbbi visszafordíthatatlan lépésig még nem telt el 4 féllépés. Azonban, a hash-kulcsot alkalmazó módszereknek ezekben az esetekben is működnie kell, a hash-kulcsot elő kell állítania, az állást be kell raknia, illetve ki kell szednie a hash-táblából. Ez okozhatja azt, hogy ha nincs már legenerálva a hash-kulcs, az eljárás jelentősen hosszabb futásidejű átlagosan, mint a saját algoritmusom.

A hash-kulcsokkal működő módszerek állásismétlés-felismeréseinek ellenőrzését nem sok idő elvégezni, mivel kevés esetben tévednek, és az állásismétlések száma is kevés. A következő táblázatban láthatjuk az egyes keresések típusai, illetve mélysége esetén az átlagos vágásszámot, a játékfaban megvizsgált állások százalékában:

ab (5,5)	0,062%
ab (4,4)	0,0094%
ab (3,7)	0,015%
ab (3,5)	0,012%
ab (3,3)	0,018%
-- (3,3)	0,028%

Mivel a hash-kulcsok alapján működő eljárások tévesztései nem érik el a valódi, észrevett állásismétléseket, így elmondható, hogy a futásidő az ellenőrzés esetén is csak ritkán nőhet 0,004%-kal (feltételezve, hogy az ellenőrző algoritmus, ha minden állásra kiszámítanánk, 2%-kal növelné a futásidőt, míg, ha az állásoknak csak 0,2%-ára vizsgáljuk meg, akkor $2\% \cdot 0,002 = 0,004\%$ -kal nő a futásidő).

Futásidő, hosszú ütés-, és gyaloglépésmentes lépéssorozatok esetén:

A második teszt alapján elmondható, hogy a hosszú, egy ütközési osztályba tartozó lépéssorozatok jelentősen növelik a lineáris futásidejű algoritmusok futását, ezzel szemben a konstans idejű, hash-táblás megvalósítások sokkal jobban szerepeltek. Megfigyelhető, hogy az Axon állásisméltés-felismerő eljárásának futásideje a saját algoritmusomhoz képest jelentősen nő. Ennek oka az lehet, hogy az Axon-ban alkalmazott állásisméltés-felismerő algoritmus futásideje nem tekinthető minden esetben lineárisnak, hiszen jelentősen több műveletet kell elvégezni, ha több nyitott kört tárolunk a futáshoz szükséges segédhalmazban.

A következő táblázat mutatja, hogy általános esetben, illetve hosszú lépéssorozatok esetén a saját algoritmusomnak szükséges idő hányszorosát igényli az Axonban alkalmazott módszer, illetve, hogy mennyivel nőtt a saját eljárásom futásidő-szüksége az átlagos esethez képest. Látható, hogy a lineáris futásidő is jelenősen meg tud nőni az átlagos helyzethez képest.

	Általános esetben	Hosszú lépéssorozatok esetén	A saját algoritmusom futásidő-növekedése
ab (5,5)	2,87	3,33	691%
ab (4,4)	2,00	3,09	540%
ab (3,7)	1,69	3,73	878%
ab (3,5)	1,75	3,50	1484%
ab (3,3)	2,14	2,50	829%
-- (3,3)	2,37	3,62	336%

Érdekes eredmény, hogy az megoldásomnál jobb eredményt ad a sakktábla-összehasonlító elven működő algoritmus. Ennek oka az lehet, hogy általában nincs ott se szükség a sakktáblák összehasonlítására, mivel a legutolsó féllépés célmezején ritkán áll ugyanaz a bábu, mint az aktuális állásban. Így, az átlagos ciklusmagban töltött idő jelentősen kevesebb, mint az eljárásom esetén, és ezzel hosszú csendes lépéssorozatok esetén az algoritmus le tudja dolgozni azt a hátrányát, hogy minden sakkállást másolnia kell.

Vágások száma, hibás vágások, hibás lépésre adott pontszámok:

Az állásismétlések száma az előző oldalon van összefoglalva a táblázatban. A hash-kulcsot alkalmazó módszerek hibázásának aránya jelentősen növekszik a keresési mélységgel. Így elmondható, hogy míg fix 3-as keresési mélység esetén a hash-táblát alkalmazó algoritmusok hibázási aránya a valódi állásismétléseknek csak 1/6-oda, míg ez az érték fix 5-ös mélység mellett 1/2 körüli. A tesztelés során a 64-bites hash-kulcsok listája alapján működő eljárás nem követett el hibát, illetve a nem javítható hash-táblás algoritmus (14+16 bites hash kulccsal) nem hagyott figyelmen kívül egy állásismétlést sem. A javítható, illetve a nem javítható hash-táblát használó változatok hibázásai között nem volt jelentős különbség, az feltűnőbben jelentkezett mélyebb keresés esetén. Az esetlegesen várt hibák a hash-kulcsok bitszámának csökkentésével megjelentek.

Általában egy állásismétlés fel nem fedezése kisebb hibát jelent a legjobb lépés megtalálása során, mint egy állásismétlésnek hitt állás esetén való vágás. Ezek miatt, a keresési mélység növekedésével egyre többet hibáznak az ellenőrizetlen hash-táblás algoritmusok, fix 5-ös keresési mélység esetén minden 50. állás anyagi értékelése hibás eredményt adott a hibás játékfa vágások miatt. Ha nem alkalmazunk állásismétlés felismerést, akkor kb. harmadannyi állás esetében téved legalább egy gyalog pontszámnyit a minimax algoritmus.

10. ÖSSZEFOGLALÁS

Dolgozatom keretében sikerült egy új, helyes eredményt adó algoritmust találnom az állásismétlések felismerésére. Ez a módszer jóval egyszerűbb, mint a szakirodalomban megtalálható szintén helyes eredményt adó, az Axon nevű sakkprogramban alkalmazott algoritmus. Ezen kívül az eljárás jóval egyszerűbb módon ismeri fel az állásokat. Az implementáláshoz harmad-negyedannyi sorra van szükség, mint az Axon-ban megvalósított módszerhez.

Az algoritmust a statisztikusan helyes megoldásokkal összevetve sem marad el lényegesen a futásidő szempontjából. Azok a statisztikusan helyes eljárások, amelyek minden állásismétlést észrevesznek, de néha akkor is jeleznek állásismétlést, ha valójában nem történt, az algoritmusom segítségével ellenőrizhetők. Azaz, minden egyes állásismétlés felismerésük után az eljárásomat lefuttatva ellenőrizhetem a felismerés helyességét. Az ellenőrzés esetén jelentősen kevesebb állást kell vizsgálni, így ez a módszer nem eredményez lényeges lassulást a minimax algoritmusban (a futásidő a 0,004%-ával nő). Így a gyors, hash-kulcsokkal működő megoldásokhoz képest gyakorlatilag plusz futásidő szükséglet nélkül kaphatunk helyes algoritmust a két módszer ötvöztetésével.

11. FELHASZNÁLT IRODALOM

- [1] www.fide.com/FIDE/handbook/LawsOfChess.pdf
(2012. november 19.)
- [2] Kovács P. Attila: *Sakkprogramozásról mindenkinek*, Novotrade, 1987, [240], ISBN-963-02-4691-0
- [3] <http://www.chessbase.com/puzzle/games/puz7.htm>
(2012. november 19.)
- [4] http://people.inf.elte.hu/fekete/docs_2/hash-eles.pdf
(2012. november 19.)
- [5] Alert L. Zobrist: *A New Hashing Method with Application for Game Playing*, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1969, [10]
- [6] <http://chessprogramming.wikispaces.com/Repetitions>
(2012. november 19.)
- [7] <http://www.sluijten.com/winglet/index.htm>
(2012. november 19.)
- [8] Vladan Vuckovic, Djordje Vidanovic: *A new approach to draw detection by move repetition in computer chess programming*, CoRR, 2004, [15]
- [9] Vladan Vuckovic, Djordje Vidanovic: *An algorithm for the detection of move repetition without the use of hash-keys*, Yujor, Vol. 15, No. 2, pp. 257-274, 2007, [18]
- [10] Gergő Lajos: *Numerikus módszerek*, ELTE Eötvös Kiadó, 2010, [332], ISBN-978-963-312-034-7
- [11] <http://chessprogramming.wikispaces.com/Board+Representation>
(2012. november 19.)