

A malom, lasker és morabaraba játékok ultra erős és kiterjesztett megoldása

Danner Gábor, Programtervező Informatikus MSc II. év, SZTE
Gévay Gábor, Programtervező Informatikus MSc II. év, ELTE

Témavezető: Prof. Csirik János, egyetemi tanár

Absztrakt

Kombinatorikus játékok erős megoldása azt jelenti, hogy a játék összes lehetséges állásának kiszámítjuk a játékelméleti értékét, lehetővé téve egy tökéletesen játszó program hatékony megvalósítását. Már régóta foglalkoznak népszerű játékok megoldásával, mint pl. különböző sakkvéjátékok, dáma, amőba és malom. A malom játékról a 90-es évek elején megállapította Ralph Gasser, hogy a kezdőállás értéke döntetlen. A lasker nevű variánst 2004-ben oldotta meg Peter Stahlhacke, és erről is azt találta, hogy optimális játék esetén döntetlennel végződik.

Négy célt valósítottunk meg. Egyrészt megoldottuk a malom Afrikában népszerű, morabaraba nevű variánsát, és azt találtuk, hogy tökéletes játékosok esetén a kezdő 49 lépésben nyer. Másrészt megvizsgáltuk mindhárom variáns esetében, hogy a játékok kezdőállásában megváltoztatva a fölrakható korongok számát, hogyan változik a végkimenetel. Azt találtuk, hogy a morabarabánál az egyenlő induló korongszámú állások többnyire nyerések, a másik két variánssal ellentétben, ahol ezek többnyire döntetlenek. Az eredeti malom játéknál is lett egy meglepő eredmény: 11 vagy 12 korong esetében a kezdő játékos veszít.

Harmadrészt kidolgoztuk a retrográd analízisnek egy olyan változatát, amely a szokásos nyerés/vesztés/döntetlenhez képest többféle kimenetelt képes kezelni, és gyorsabb, mint a korábbi hasonló célú algoritmus. Negyedrészt az előbbire építve kifejlesztettünk egy algoritmust, amely a malom és variánsai esetében a döntetlen értékű állások között különbséget tud tenni az állás heurisztikus kedvezősége szempontjából. Ennek jelentősége abban áll, hogy ha ennek hiányában választ a tökéletesen játszó program a különböző optimális lépéslehetőségek közül, akkor általában látszólagosan gyengéket lép. Ezek az optimális lépések ráadásul – paradox módon – valóban gyengék, olyan értelemben, hogy az ellenfélnek nem nehéz találni optimális ellenlépéseket, és így az eredeti malomnál és a laskernél a játszmák már nem túl erős játékos esetén is nagyon gyakran végződnek döntetlennel. Az ezt kiküszöbölő korábbi próbálkozások lokális heurisztikákat használtak, vagy játék közben tanultak, de mi a retrográd analízisbe építettük be az algoritmusunkat.

Tartalomjegyzék

1. Bevezetés	2
1.1. Állások játékelméleti értéke	2
1.2. Játékok megoldásának fogalmai	2
1.3. Irodalmi áttekintés	3
1.4. A dolgozat felépítése	4
1.5. Játékszabályok	4
2. Retrográd analízis	5
2.1. Az állapottér szektorokra bontása	8
2.2. A retrográd analízis implementálása a szektorokra bontott állapottéren	9
2.3. A hash függvény	12
3. Ultra erős megoldás	14
3.1. Választás a „tökéletes” lépések közül	14
3.2. A Retrográd analízis háromnál többféle kimenetelt is kezelő változata	15
3.3. A döntetlenek megkülönböztetése a végső szektor heurisztikus értéke alapján	17
3.3.1. A stabil állásokhoz használt heurisztika	18
3.4. A retrográd analízis döntetleneket megkülönböztető változata	19
3.4.1. A távoliság definiálása döntetlenekre	20
4. Eredmények	21
4.1. Morabaraba	21
4.2. Kiterjesztett megoldások	22
4.3. Ultra erős megoldások	26
4.4. A számítások ellenőrzése	26
5. Továbbfejlesztési lehetőségek	27
5.1. Külön állásértékelések a tökéletes illetve a tökéletlen játékos számára	27
5.2. Általánosítás más játékokra	27

1. Bevezetés

1.1. Állások játékelméleti értéke

Egy állás játékelméleti értékén azt értjük, hogy mi lesz a játszma kimenetele az adott állásból indulva, amennyiben a játékosok optimálisan játszanak. Pl. ha egy állás értéke nyerés, akkor az a játékos, aki lépni következik, megfelelő stratégiát használva mindenképpen meg tudja nyerni a játszmát, bárhogyan is játszik az ellenfél.

Továbbá beszélhetünk játékok játékelméleti értékéről, amikor is a kezdőállás fentebbi értelemben vett értékére gondolunk. Ha ez nyerés, akkor ezt szokás úgy is mondani, hogy az első játékosnak van nyerő stratégiája, ha vesztes, akkor a második játékosnak, ha pedig döntetlen, akkor mindkét játékosnak van nem vesztes stratégiaja.

A malom és az általunk vizsgált variánsai teljes információs (a játékosok ismerik a játék állapotát), kétszemélyes, lépésváltásos (a játékosok felváltva lépnek), determinisztikus (nincs szerepe a véletlennek), véges (az állapotokról és a játszmák hossza véges), zéró-összegű játékok, és háromféle kimenetelük lehet: az első játékos nyer, a második játékos nyer, illetve döntetlen. Ilyen játékokban mindig vagy valamelyik játékosnak van nyerő stratégiája, vagy mindkettőnek van nem vesztes stratégiája. Ez egyrészt következik a Neumann-féle minimax tételből, viszont erre a speciális esetre a legtávolabbi végállapottól való távolság szerinti indukcióval is könnyen bizonyítható, ami tulajdonképpen hasonlít a későbbiekben leírt retrográd analízis algoritmusánál szereplő bizonyításhoz.

1.2. Játékok megoldásának fogalmai

Játékok megoldásáról többféle értelemben beszélhetünk [1].

- *Ultra gyenge* értelemben megoldottnak mondunk egy játékot, amennyiben ismert a kezdőállás játékelméleti értéke, azaz egy bizonyítás arra vonatkozólag, hogy optimális játékosok esetén mi a kimenetele egy játszmának. Azonban ez a bizonyítás lehet nem konstruktív is, vagyis elképzelhető, hogy nem mond semmit arról, hogy pontosan hogyan is kell jól játszani az adott játékot.
- *Gyenge* értelemben megoldottnak mondunk egy játékot, ha ismert egy (valódi játszmák időkeretei közt működő) algoritmus is, amely képes elérni legalább a kezdőállás játékelméleti értékét (bármelyik játékos szerepében játszva). Azonban ha valamelyik játékos hibázása esetén optimálisan folytatva a játékot elérhető lenne a kezdőállás játékelméleti értékénél jobb eredmény, akkor az algoritmus nem feltétlenül éri ezt el. Ilyen értelemben van megoldva pl. a dáma [2].
- Akkor mondunk egy játékot *erősen* megoldottnak, ha egy olyan algoritmusunk van rá, amely tetszőleges játékállásból optimálisan tud játszani (vagyis minden esetben ki tudja

használni az ellenfél nem optimális lépéseit, sőt, akár olyan állásokból is optimálisan játszik, ahol már mindkét oldal hibázott). Az algoritmus természetesen tartalmazhat egy nagy méretű adatbázist, de nem feltétlenül ez a helyzet, amire közismert példa a nim játék.

- *Ultra erős* megoldásnak [3] hívunk egy stratégiát, mely növeli a játékelméleti értéknél jobb eredmény elérésének esélyét (az optimális lépések közül véletlenszerűen választó algoritmushoz képest) tökéletlen ellenfél ellen. (Egy játékos tökéletes, ha mindig fent tudja tartani a játékelméleti értéket.) Ez egy eredendő heurisztikus fogalom.
- Érdekes megvizsgálni alternatív kezdőállások játékelméleti értékét is (pl. malom esetén különböző számú induló korongkészlet). *Kiterjesztett* megoldásnak fogunk nevezni egy kiterjesztett állapottérre (azaz kezdőállapotok egy halmazából elérhető állásokra) vonatkozó erős megoldást.

Ha van egy adatbázisunk az összes lehetséges játékállás játékelméleti értékéről, akkor játszhatunk úgy, hogy mindig egy lépéssel előre tekintve maximalizáljuk (a mi szempontunkból) a lépésünk utáni állás értékét. Azonban ha körök is vannak a játékgráfban¹, akkor ez az algoritmus nem feltétlenül biztosítja a nyerést, hiszen előfordulhat, hogy visszajutunk egy olyan állásba, amelyben már korábban jártunk, és ha itt determinisztikusan választunk, akkor esetleg sosem ér véget a játszma. A véletlenszerű választás sem igazán oldja meg a problémát, mert akkor túl hosszúvá nyúlhatnak a játszmák.

Ezt úgy oldhatjuk meg, hogy még azt is kiszámoljuk minden állásról, hogy optimális játék esetén hány lépésben fog véget érni a játszma, amennyiben a nyerő játékos ezt minimalizálni igyekszik, a veszteső játékos pedig maximalizálni. Ezeket az értékeket távoliságnak fogjuk nevezni. Ezt az információt aztán játék közben második kulcsként használhatjuk a lépéslehetőségek rangsorolásánál. (Döntetlen értékű állásoknál nincs távoliság, hiszen nem tudnánk eldönteni, hogy maximalizáljuk, vagy minimalizáljuk.) Így optimális játék esetén az aktuális állás távolisága minden lépésnél eggyel csökken.

1.3. Irodalmi áttekintés

A malom játékot Ralph Gasser oldotta meg gyenge értelemben a 90-es évek elején [4]. Retrográd analízissel kiszámította minden olyan állás értékét, amelyekben már az összes korong föl lett rakva a táblára, azonban a fölrakásos állások közül csak annyinak határozta meg az értékét, amennyire szüksége volt a kezdőállás értékének meghatározásához: a kezdőállásra 18 mélységű alfa-béta vágást futtatott, aminek a leveleiben a retrográd analízissel kiszámolt adatbázist használta föl. Így megállapította, hogy a kezdőállás játékelméleti értéke döntetlen.

¹Nevezük játékgráfnak azt a gráfot, amelynek a csúcsai a játék összes lehetséges játékállása, és akkor megy irányított él u -ból v -be, ha egyetlen lépéssel átléphetünk u -ból v -be.

Gasser munkájának jelentőségét az is emeli, hogy kitalált egy olyan hash függvényt [5], amely nagyrészt képes figyelembe venni a malom tábla szimmetriáit, és így közel tizenhatodára csökken a számítás erőforrásigénye. Ezt a hash függvényt mi is használjuk, és a 2.3 fejezetben részletezzük a működését.

A laskert Peter Stahlhacke oldotta meg 2003-ban, erős értelemben [6]. Cikkének konklúziójában megemlíti, hogy tervez foglalkozni azzal, hogy a program a játékelméleti értéknél jobb eredményt érhesse el nem tökéletesen játszó ellenfél ellen.

A morabaraba megoldásáról nem található semmi az irodalomban.

Az Awari játék megoldásához kidolgozták a retrográd analízisnek egy olyan változatát, amely háromnál többféle kimenetelt is képes kezelni, azonban az lényegesen különbözik a mi-enktől [7] [8]. Kevesebb memóriát használ, és nem törekszik a távoliságok meghatározására.

Játékok ultra erős megoldásával foglalkozott Jansen [9], illetve Lincke [10], lokális heurisztikák használatával. Egy másik megközelítés, hogy ha a játszmák során tanulja meg a program az egyes állások kedvezőségét [11]. Az előbbiekkal szemben mi a retrográd analízisbe építettük be az algoritmusunkat.

1.4. A dolgozat felépítése

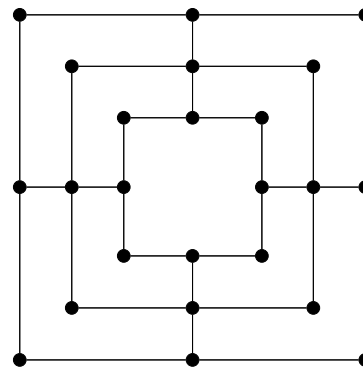
A következő alfejezetben leírjuk a malom és az általunk vizsgált variánsainak a szabályait. A 2. fejezetben leírjuk a szokásos retrográd analízist, az állapottér szektorokra bontását, majd pedig ehhez kapcsolódó implementációs részletek következnek egy pszeudokóddal.

Ezután leírjuk a retrográd analízisnek egy olyan általánosítását, amely képes háromnál többféle kimenetelt kezelni, és ezt alapul használjuk egy olyan algoritmushoz, amely osztályozza a döntetlen értékű állásokat az ultra erős megoldás érdekében. Végül elemezzük a számítások eredményeit, és fölvázoljuk az algoritmusnak egy lehetséges továbbfejlesztését.

1.5. Játékszabályok

A malom játékot az 1. ábrán látható gráfon játszik, melynek a csúcsaira kétféle színű korongok kerülhetnek. A tábla kezdetben üres, a játékosoknál pedig 9-9 fehér, illetve fekete korong van. A játékosok felváltva lépnek, a fehér kezd. A játék a felrakási fázissal indul: ekkor egy lépés egy korong felrakását jelenti egy üres mezőre. Amikor az összes korong felkerült a táblára, a mozgatási fázis következik, amikor is a lépések abból állnak, hogy egy játékos egy korongját egy szomszédos üres mezőre mozgatja. Ha a játszma során bármikor egy lépés megtételekor kialakul egy malom, azaz három egyforma színű korong egy sorban, akkor a játékos lépése kiengesztül azzal, hogy leveszi az ellenfélnek egy olyan korongját, amely nincs malomban. Amikor egy játékosnak már csak három korongja maradt, akkor nem csak szomszédos mezőre mozgat-hat, hanem tetszőlegesen.

Ha egy játékosnak már csak két korongja maradt, akkor vesztett, illetve akkor is, ha nincs legális lépése. Úgy fogjuk tekinteni, hogy ha megismétlődik egy játékállás, akkor a játszma véget ér, döntetlen eredménnyel. Utóbbi szabályra azért van szükség, hogy a játszmák ne lehessenek végtelenül hosszúak. (A gyakorlatban ehelyett szokás úgy játszani, hogy akkor végződik döntetlennel a játszma, ha nagyon sok (pl. 50) lépésig nem történik koronglevétel, azonban ez elegendően nagy lépésszám választása esetén a megoldás szempontjából nem különbözik az általunk használt szabálytól. Használható a sakkhoz hasonló szabály is, ahol egy állás háromszori előfordulása eredményez döntetlent, azonban ez se befolyásolja a megoldást.)



1. ábra. A malom játék táblája.

A fentebbiekből még nem egyértelmű, hogy mi történik, ha egy lépéssel egyszerre két malmot zárunk be. Az általunk implementált szabályokban ilyenkor csak egy korongot lehet venni. Ezen kívül felvetődik a kérdés, hogy mi történjen, ha le kéne venni egy korongot, de az ellenfél összes korongja malomban van. Úgy tekintjük, hogy ilyenkor le lehet venni malomban lévő korongot is. Az előbbi két kérdéses szabállyal kapcsolatos döntést főként az alapján hoztuk meg, hogy az a weboldal is ezeket alakalmazza, ahol teszteltük a programunkat ember játékosok ellen.

A malom játék lasker nevű variánsát Emanuel Lasker egykori sakkvilágbajnok találta ki azzal a céllal, hogy kevesebb legyen a döntetlen kimenetelű játszma. Abban különbözik az eredetitől, hogy nincs külön felrakási és mozgatási fázis, hanem bármikor lehet mozgatni és felrakni is (amíg van fel nem rakott korongunk), továbbá 9 helyett 10 koronggal kezdenek a játékosok.

A morabaraba nevű variáns főként Afrikában népszerű. Itt 12 korongja van a játékosoknak, és a tábla gráfja is különbözik (lásd a 2. ábra a 4. fejezetben). Továbbá van még egy speciális szabály: ha a tábla betelt, akkor a játszma döntetlennel ér véget (ahelyett, hogy az első játékos vesztené, legális lépés híján). Internetes keresés alapján az látható, hogy ezzel a szabállyal kapcsolatban egyáltalán nincs egyetértés a játékosok között, úgyhogy implementáltuk azt a változatot, amikor van ez a szabály, és azt is, amikor nincs.

2. Retrográd analízis

A retrográd elemzés során úgy számoljuk ki játékállások értékét, hogy ismert értékű állásokból (kezdetben végállapotokból) visszafelé haladva terjesztünk értékeket a minimax elv szerint. Az algoritmus alapötletét már 1910-ben leírta Ernst Zermelo [12], Richard Bellman pedig 1965-ös cikkében [13] a dinamikus programozás alkalmazásaként tárgyalta. Az első implementációt Thomas Ströhlein készítette, és az 1970-es doktori disszertációjában írt róla német nyelven [14].

Népszerűvé azonban Ken Thompson 1986-os cikkével vált [15], amelyben számos sakkvégjátékot elemzett a segítségével. A következőkben leírt változat hasonlít a Ralph Gasser által az eredeti malom játék megoldásához használthoz [5].

Egyelőre tekintsük úgy, hogy a játékgráfban tartozik csúcs az olyan játékállaspárok mindkét tagjához, amelyek csak abban különböznek, hogy az egyikben a fehér, a másikban a fekete következik. (Az implementációnkban valójában csak minden állásnak azt a változatát tároljuk, amelyikben a fehér következik, lásd a 2.1 alfejezet.)

Először tekintsük azt az esetet, amikor nincsenek irányított körök a játékgráfban. Első lépésként kitöltjük a végállapotok értékét (az első játékos szemszögéből). Ezután minden lépésben kitöltjük egy olyan s állásnak az értékét, aminek már minden rákövetkezőjének ki van töltve az értéke: ha az első játékos következik, akkor a rákövetkezők értékeinek a maximumát vesszük, ha pedig a második, akkor a minimumát. Így végül minden állás értéket kap. Azt kell bizonyítanunk, hogy nem akad el az algoritmus, azaz mindig találunk megfelelő s állást. Ezt indukcióval végezzük.

Legyen a $h(n)$ halmaz azon csúcsok halmaza, amelyektől a legtávolabbi végállapot (előrefelé lépegetve) legföljebb n lépésre van. Be fogjuk bizonyítani, hogy minden n -re a $h(n)$ halmazban lévő csúcsok mindegyike sorra fog kerülni. A $h(0)$ -ban lévő állásokra már az első lépésben ráírjuk az értékeiket, tehát ezekre az állítás nyilvánvaló. Tegyük fel, hogy $(n - 1)$ -ig bizonyítottuk az állítást, vagyis a $h(n - 1)$ halmazbeli csúcsokra már bizonyítottuk, hogy rá fog kerülni az elméleti értékük, és nem akadt el az algoritmus. Ekkor minden $h(n) \setminus h(n - 1)$ -beli állásnak az értéke is meghatározható, hiszen az összes ilyen állás összes rákövetkezője $h(n - 1)$ -ben van, tehát már tudjuk, hogy fog rájuk kerülni érték, vagyis bizonyítottuk az állítást $h(n)$ -re.

A fenti algoritmus kiegészíthető úgy, hogy kiszámítsa az állás értékén túl a távoliságokat is, azaz hogy optimális játék esetén hány lépésben érünk el egy végállapotot: a végállapotok a 0 távoliságú állások, a többi állásnál pedig az azonos értékű rákövetkezőknek megnézzük a távoliságát, majd ezekből kiválasztjuk a maximális vagy minimális távoliságot, aszerint, hogy vesztesénnél vagy nyerésnél járunk (a soron következő játékos szempontjából), és 1-gyel megnöveljük.

Azonban a malom játékban vannak körök a játékgráfban, úgyhogy most erre az esetre is adunk egy algoritmust. A fenti algoritmus akkor is működik, ha a kimenetel többféle lehet, mint nyerés, veszteség, vagy döntetlen (pl. reversiben a korongkülönbség, vagy az awariban a megszerzett kövek számának különbsége), azonban most előbb olyan játékokra ismertetjük a körökre is működő algoritmust, amelyeknek csak háromféle kimenetelük lehet.

A továbbiakat célszerűbb lesz úgy megfogalmazni, hogy az állásokra írt értékek a soron következő játékos szemszögéből értendők. (Ez abban különbözik a fentebb használt értékelésektől, hogy azoknak az állásoknak az értéke, ahol a második játékos következik, az ellentettjére változik.) Tehát a továbbiakban a nyerő- és veszteső állás fogalmakat úgy értjük, hogy optimális játék esetén az a játékos fog nyerni, illetve veszíteni, aki éppen következik. Ez a módosítás

egyébként megfelel a minimax és a negamax algoritmusok közti különbségnek.

Először is vegyük észre, hogy nem mindig van szükség egy állás értékének meghatározásához az összes rákövetkezőjének az értékének az ismeretére: ha egy s állásnak van egy olyan rákövetkezője, amiről tudjuk, hogy az értéke vesztes, akkor már rögtön ráírhatjuk a nyerést az s -re, hiszen a maximumkeresés során nem kaphatunk ennél jobb értéket. Viszont egy állás vesztesnek való elkönyveléséhez szükség van az összes rákövetkezőjének az értékének az ismeretére, hiszen pontosan akkor lesz egy állás vesztes, ha minden rákövetkezője nyerés. Ezek után az a kérdés, hogy mikor könyvelhetünk el egy állást döntetlennek. Az algoritmus úgy fog működni, hogy csak a lefutásának a legvégén hoz olyan döntést, hogy egy állás értéke biztosan döntetlen, azonban ez nem lehet probléma, ugyanis figyeljük meg, hogy olyan állások, amelyeknek végül döntetleneknek kell lenniük, nem játszanak szerepet olyan döntésekben, hogy egy-egy állás már biztosan nyerés vagy biztosan vesztes.

Tehát az új algoritmus abban fog különbözni a fentebbitől, hogy már akkor is kiválaszt egy s csúcsot, és ráírja az értékét, ha valamelyik rákövetkezője vesztes. Így szervezhetjük úgy az algoritmust, hogy a távoliság sorrendjében dolgozzuk fel a játékgráf csúcsait. A helyesség bizonyítása a fentebbi teljes indukcióhoz hasonló, azonban most a távoliság szerint végezhető az indukció. Azok a csúcsok, amelyekre nem került explicit címke, döntetlennek tekintendők.

Az implementációhoz használhatunk egy sort: amikor egy csúcshoz bejegyezzük az értéket, akkor berakjuk egy sorba. Az algoritmus azt fogja csinálni, hogy minden lépésben kivessz egy elemet a sorból, és „terjeszti”, azaz meghatározza a megelőző állásait, ezeknek frissíti az állapotát, és a most elkészülteket berakja a sorba. A sor tárolható akár a lemezen is, mivel csak szekvenciális fájlműveleteket igényel.

Az állások állapota kétféle lehet: count és value. Mindkét esetben van egy szám is bejegyezve az álláshoz. A value azt jelenti, hogy már biztosan tudjuk az értékét (ez lehet egyszerre tárolva a távolisággal: a távoliság paritása mutatja, hogy nyerésről vagy vesztesről van-e szó), és ez van bejegyezve, countnál pedig azt adja meg a beírt szám, hogy hány olyan rákövetkezője van, amelyből még nem terjesztettünk. A terjesztés során így két eset van: ha vesztesből terjesztünk countba, akkor kapunk egy nyerést, aminek a távoliséga eggyel nagyobb annál, amiből terjesztettünk; ha nyerésből terjesztünk countba, akkor csökkenteni kell a countot, ami után megtörténhet, hogy éppen eléri a nullát, és ekkor egy vesztesést kaptunk. (Egy harmadik eset lenne az, ha value-ba terjesztünk, ekkor azonban nem kell csinálni semmit.) Az algoritmus lefutásának a végére pontosan a döntetlen állások maradnak countok.

Az állapotok inicializálásánál egyrészt be kell jegyeznünk a végállapotok értékeit, másrészt a többi állás kezdeti countját ki kell számolnunk. Ez történhetne úgy, hogy közvetlenül kiszámoljuk a rákövetkezők számát, de később leírt okok miatt ehelyett úgy működik a programunk, hogy végigmegy az összes álláson, mindegyiknek legenerálja a megelőző állásait, és eggyel növeli ezek countját. (Az algoritmus pszeudokódja megtalálható a következő fejezet végén.)

A memóriaigény kiszámolásához tudnunk kell egy bejegyzés méretét. Szerencsére nincsen szükségünk egy állásnál egyszerre countra és value-ra. A value állapotú állásokhoz is elegendő

egy számot tárolni, ugyanis ha ez a szám az állás távolságát jelöli, akkor a paritása megadja, hogy vesztés vagy nyerés, hiszen a malom játékban mindenfajta nyerésnél az a játékos lép utoljára, aki nyert. A tipikus count és value értékek elférnek 1 byte-on. A Gasser által használt módszer az, hogy a 0..255 értékek közül a 255-höz közeliakat fenntartja countnak (a valódi count a bejegyzett számnak a 255-höz vett különbsége), a 0 közeliak pedig value-t jelentenek. Azonban így nem volt elég nagy az 1 byte-on ábrázolható értékkészlet, ezért a következő megoldást találtuk ki²: mivel csak viszonylag kevés olyan állás van, aminek a bejegyzése nem lenne ábrázolható, ezért ezeknek kijelöltünk egy speciális értéket, és fölvtünk a memóriába egy STL mapet (ami egy kiegyensúlyozott keresőfa), amibe eltároltuk csak ezeknek az állásoknak az értékét. Az ebbe való beleindexelés sokkal lassabb, mint a többi állás értékét tároló tömbbe, azonban mivel csak elenyészően kis számú állásnál van szükség a használatára, ezért gyakorlatilag nem befolyásolja a program futási sebességét.

2.1. Az állapottér szektorokra bontása

A fenti algoritmus az állások számával lineáris memóriát igényel, mivel az állásokhoz tartozó bejegyzéseket tároló tömbhöz véletlenszerű (nem szekvenciális) hozzáférésre van szükség, tehát nem tárolhatjuk a lemezen.

Az eredeti malom játékban az összes csúcsok száma³ 26.761.334.939, a laskerben 133.460.964.090, a morabarabában 111.999.566.300. Még ha minden csúcshoz elég csak 1 bájtot tárolni, ez akkor sem fér el az általunk használt számítógép memóriájában. Viszont particionálhatjuk a csúcsok halmazát az alapján, hogy melyik játékosnak hány korongja van fönt a táblán, illetve hányat rakhat még föl. Így négy szám azonosít egy szektort. (Pl. a 6,7,2,1 szektorban olyan állások vannak, ahol fehérnek 6, feketének 7 korongja van fönt, fehér még 2 korongot rakhat föl, fekete pedig 1-et.) Ennek az az előnye, hogy a korábban leírt algoritmus módosítható úgy, hogy egyszerre csak néhány szektornak kelljen a memóriában lennie (a legnagyobb szektor 603.332.730 állást tartalmaz).

A szektorokból építhetünk egy gráfot: legyenek a csúcsok a szektorok, és akkor megy irányított él az u szektorból a v szektorba, ha van az u szektorban olyan állás, amiből egy lépéssel átléphetünk egy v szektorban lévő állásba. (Bár a könnyebb implementáció érdekében a szektorgráf kiszámítása során a bejöhetnek fölösleges élek.)

Korábban említettük, hogy csak olyan állásokat tárolunk, amelyekben a fehér következik. Ennek az az oka, hogy, ha egy állást „negálunk”, azaz lecseréljük az összes fehér korongot feketére, és az összes fekete korongot fehérre, valamint megváltoztatjuk, hogy ki következik, akkor egy ekvivalens állapotba jutunk (a soron következő játékos szemszögéből megegyezik a játékelméleti érték). Ennek viszont van egy olyan kellemetlen mellékhatása, hogy a játékgráfban lévő körök nem korlátozódnak egy szektorra, vagyis a szektorgráf nem körmentes. Pl. a

²Gasser ezeket az állásokat speciálisan kezelte, mivel az ő számításai során csak kevés állásnál fordult ez elő.

³Ezek a szimmetriák nagy részének kiszűrésével kapott értékek (ld. a 2.3. fejezetben), viszont nem minden elérhetetlen állást szűrtünk ki.

4,6,0,0 szektorból minden olyan lépés, amely nem korongot vesz le, a 6,4,0,0 szektorbeli állásba vezet a negálás miatt. Így a legtöbb kör olyan, hogy két szektor között alternál (kivéve azoknál a szektoroknál, amelyek saját maguk negáltjai, amelyeknél viszont a körök egy szektoron belül maradnak). Emiatt viszont nem dolgozhatunk föl egy szektort önmagában, úgyhogy egy munkaegységnek fogunk nevezni két olyan szektort, melyek egymás negáltjai, és vezet köztük él mindkét irányba, a többi szektor pedig önmagában képez egy-egy munkaegységet. (Szektorok negálása úgy történik, hogy megcseréljük egymással az első két számot, és a második két számot.) A munkaegységek gráfja már körmentes, és játék közben csak előre fel haladunk az élein, vagy csúcson belül maradunk. (Általánosabb esetben a munkaegységeknek a szektorgráf erősen összefüggő komponenseinek kéne lenniük, azonban jól látható, hogy a malom játéknál ezek csak két szektorból állhatnak, melyek egymás negáltjai.)

2.2. A retrográd analízis implementálása a szektorokra bontott állapotter- ren

Egy munkaegység feldolgozásához a memóriában kell lennie egyrészt a munkaegység szektorainak (ezeket főszektoroknak fogjuk nevezni), másrészt azoknak a szektoroknak is meg kell jelenniük, amelyek a szektorgráfon egy lépéssel elérhetőek a munkaegység valamelyik szektorából (ezekre pedig mellékszektorokként fogunk hivatkozni). Ez a lasker esetében jelenti a legtöbb szektort, legfeljebb nyolcat (egyébként, ha a szabályokat úgy rögzítettük volna, hogy egyszerre két malom becsukása esetén két korongot vehetünk le, akkor több szektorral kéne foglalkozni egy munkaegység feldolgozása során). Azonban a mellékszektorokra nincs egyszerre szükségünk, amint azt az algoritmus részleteinek leírásánál látni fogjuk.

A feldolgozási sorrend most nem követheti tökéletesen a távoliságokat, hiszen az egyes szektorokban előforduló távoliságok értékkészletei átfedhetnek egymással, tehát egy munkaegység feldolgozásakor fel kell dolgoznunk olyan állásokat is, amelyeknél még fogunk találkozni kisebb távoliságúakkal egy későbbi munkaegységben. Azonban ez nem okoz problémát, ugyanis lényegében csak azért akartuk a távoliság szerinti sorrendben feldolgozni a csúcsokat, hogy ne fordulhasson elő az a helyzet, hogy egy később feldolgozott állás hatással kéne, hogy legyen egy már korábban feldolgozott, viszont látható, hogy ha az egy-egy munkaegység feldolgozása során szerepet játszó csúcsokra megtartjuk a távoliságssorrendet, akkor csak olyan esetben fordulhat elő, hogy egy u csúcsnál később dolgozunk fel egy kisebb távoliságú v csúcsot, ha a v csúcs később feldolgozott munkaegységben van, ami viszont azt jelenti, hogy a v csúcs biztosan nem lehetne hatással az u csúcs értékére, hiszen nem is juthatunk el u -ból v -be játék közben.

Tehát egy munkaegység feldolgozása során azt kell elérnünk, hogy a mellékszektorokban lévő sokféle különböző távoliságú kész állás⁴ (azaz amelyeknek az állapota value) a korábban

⁴Ez a probléma nem merül fel, ha a játszma végéig tekintett távoliság helyett a következő koronglevételig hátra lévő lépések számát használjuk.

emlegetett sorból a megfelelő pillanatban kerüljön ki, konkrétan akkor, amikor a főszektorokban lévő csúcsok feldolgozása eléri a megfelelő távolságot. Elméletben gondolkozva ezt megtehetnénk úgy, hogy a szokásos sor helyett egy prioritási sorba tesszük az elkészült csúcsokat a távolsággal, mint kulccsal (mind a főszektorokban, mind pedig a mellékszektorokban találhatóakat). Azonban ez a sor nagyon nagy is lehet, vagyis a lemezen célszerű tárolni.

Ebben a speciális helyzetben ez megoldható két sor folyamatos összefésülésével: az egyik sor mellékszektorokbeli nem döntetlen állásokat tárol (és végállapotokat a főszektorokból), és a munkaegység feldolgozásának az elején inicializáljuk, a másikba pedig a főszektorokból származó csúcsok fognak kerülni az algoritmus további részének futása során. Ezt a két sort folyamatosan összefésüljük egyetlen sorrá: mindig abból veszünk ki elemet, amelyiknek az elején éppen kisebb távolságú állás található. Az első sort a lemezen tároljuk, a másodikat pedig a memóriában, hogy kevesebb legyen a lemezművelet (ez a párhuzamosításhoz jön jól, ld. később).

A mellékszektorok állásainak értékeit tároló fájlokban nem a megfelelő sorrendben vannak az állások, de mivel nincsen néhány száznál többféle távolság, így rendezhetjük őket vödörös rendezéssel: nyitunk minden távolságnak egy vödörfájlt, végigolvassuk a szektorfájlokat, és minden állást hozzáfűzünk a megfelelő vödörfájl végére, majd a sor feldolgozása során pedig a megfelelő sorrendben végigolvassuk a fájlokat. (Azonban ahhoz, hogy ez a rendezési módszer valóban gyors legyen, nem szabad túl sokan lenniük a vödöröknek, ugyanis a hozzáfűzések csak akkor lesznek szekvenciális lemezműveletek, ha pufferelni tudjuk az írásokat.)

A szektorokra bontott számításnak egy másik előnye, hogy könnyen párhuzamosítható, hiszen azok a munkaegységek, amelyek között a munkaegységek függőségi grájában nincsen irányított út, egymástól teljesen függetlenül végezhetők el. Egy lehetséges megoldás a következő: egy külső vezérlőprogram számon tartja a munkaegységek állapotát, a köztük lévő függőségek gráját, valamint a számítógép erőforrásait, és időről időre elindít egy munkaegységet. Egy munkaegység akkor válik indíthatóvá, amikor minden megelőzője elkészült. Mivel a nagyobb munkaegységek kiszámítása több GB memóriát igényel, ezért a párhuzamosíthatóságot nagyban befolyásolja a rendelkezésre álló fizikai memória mennyisége. Továbbá a memória sebessége is annál nagyobb szerepet kap, minél több szálon fut a számítás, hiszen az állásokhoz tartozó bejegyzéseket tároló tömbhöz való hozzáférések megsűrűsödnek.

Az algoritmus időigénye $\mathcal{O}(E + VD + SB)$, ahol V az állások száma, E az ezek között húzódó élek, D a maximális befok a szektorgráfon, S a szektorok száma és B a maximális távolság. A B szorzó a vödörös rendezésből származik. A VD abból ered, hogy az állásokat többször dolgozzuk fel mellékszektorok tagjaként. Feltételeztük, hogy az állások megelőzőinek generálásának időigénye csak azoknak a megelőzőknek a számával arányos, amelyek benne vannak valamelyik (jelenlegi) főszektorban. (Enélkül a feltételezés nélkül az időigény $\mathcal{O}(ED + E + VD + SB)$ lenne.)

1. Inicializálás

Bejegyezzük a végállapotok értékeit ($\text{count}(0)$ a döntetlenekhez), és berakjuk a nem döntetlen végállapotokat a prioritási sorba (ezek value -k).

Kiszámítjuk a rákövetkezők számát minden más álláshoz, és bejegyezzük ezeket countokként.

2. A prioritási sor földolgozása

while a prioritási sor nem üres **do**

 Vegyük egy e állást a prioritási sorból

for all p megelőzőjére e -nek **do**

if p count **then**

if e nyerés **then**

 Csökkentjük p countját

if p countja 0 **then**

 Bejegyezzük, hogy p vesztes $1 + \text{DTW}(e)$ lépésben

 Betesszük p -t a prioritási sorba

else

 Bejegyezzük, hogy p nyerés $1 + \text{DTW}(e)$ lépésben

 Betesszük p -t a prioritási sorba

1. Algoritmus. A retrográd analízis pszeudokódja. $\text{DTW}(e)$ egy állás távolságát jelöli (Depth-To-Win). A malom esetén elég a DTW -t tárolni az állásokhoz, mivel a nyerés/vesztés megállapítható a paritásból. A prioritási sor kulcsa a DTW , és a legegyszerűbb esetben ez implementálható egy szokványos sorral, mivel az állások földolgozása monoton. Viszont amikor a végállapotoknak különféle értékeik lehetnek (mert figyelembe vesszük az állapottér fölosztását), akkor a két soros megoldás alkalmazható (ld. a 2.2 alfejezet). (A mellékszektorokban lévő állásokra végállapotokként tekintünk majd.) Vegyük észre, hogy e származhat fő- illetve mellékszektorból is.

1. Főszektorok előinicializálása

for all e főszektorbeli állásra **do**

if lehet becsukni malmot e -ben és az ellenfélnek csak 3 korongja van **then**

 Bejegyezzük, hogy e nyerés 1 lépésben

else

e -hez bejegyezzük 0-s countot

2. Countok beállítása a rákövetkezők számára és a value-k berakása a prioritási sorba

for all e fő- és mellékszektorbeli állásra **do**

for all p megelőzőjére e -nek **do**

if p egy főszektorban van és count **then**

p countját növeljük

if e value **then**

 Betesszük e -t a prioritási sorba

3. Beszorított állások kezelése

for all e főszektorbeli állásra **do**

if e bejegyzése $\text{count}(0)$ **then**

 Bejegyezzük, hogy e vesztes 0 lépésben

 Betesszük e -t a prioritási sorba

2. Algoritmus. A retrográd analízis inicializálása a malomra és az állapottér fölosztására szabva.

2.3. A hash függvény

Az egy szektorban lévő állásokhoz tartozó bejegyzéseket egyetlen nagy tömbben tároljuk, melybe az állásokból számolt hash függvénnyel indexelünk. Ezt túlzás lenne hash táblának nevezni, ugyanis ez a hash függvény injektív (amit tökéletes hash függvénynek szokás nevezni), aminek az az előnye, hogy nincs szükség ütközéskezelésre, és így az állásokat nem kell eltárolnunk a tömbelemekben (az állás eltárolása kb. 4 byte-on lenne lehetséges, azonban az állásokhoz tartozó bejegyzések a korábban bemutatott módon elférnek mindössze 1 byte-on). Egy jó tökéletes hash függvénynek még egy tulajdonsággal kell rendelkeznie. Ez a minimalitás, azaz hogy az értékkészlete a lehető legkisebb.

A számításigény csökkentéséhez kihasználhatjuk, hogy a malomtábla jelentős szimmetriát mutat: a szimmetriaművelettel egymásba átvihető állások csúcsait a játégráfban összevonhatjuk egyetlen csúccsá. A malomtábla egyrészt szimmetrikus a négyzet szimmetriaműveleteire, másrészt megcserélhető a külső és a belső négyzet. Így ezzel a módszerrel közel tizenhatodára csökkenthető a vizsgálandó állások száma. Ehhez a hash-függvényt úgy kell módosítani, hogy ugyanazt a hash-t rendelje olyan állásokhoz, amelyek szimmetriaműveletekkel átvihetők egymásba [5].

Egy, a lassúsága miatt a gyakorlatban használhatatlan ilyen hash függvény lenne a következő: felsoroljuk valamilyen sorrendben a szektor összes állását, és amikor meg kell határoznunk egy állás hash-ét, akkor indítunk egy számlálót nulláról, ellépegetünk egyesével a felsoroláson eddig az állásig, és közben minden olyan állásnál növeljük eggyel a számlálót, amelyik nem vihető át szimmetriaművelettel olyan állásba, amely már előfordult a lépegetés során. Gasser hash függvénye ehhez a függvényhez nagyon hasonló eredményt ad, azonban gyorsan számolható.

Először gondoljuk végig, hogy ha nem akarnánk kiszűrni a szimmetriákat, akkor hogyan tudnánk gyorsan kiszámolni a hash-t. Egy szektor minden állásában ugyanannyi fehér, és ugyanannyi fekete korong van, legyen ezek száma w és b . A fehér korongok összes lehetséges elhelyezkedése a táblán $\binom{24}{w}$, ami nem túl nagy szám ahhoz, hogy egy lookup table-ben eltárolhassunk egy sorszámot a fehér korongok minden lehetséges elhelyezkedéséhez. Ha kivesszük a táblából azokat a mezőket, amelyeken fehér korong van, akkor a fennmaradó mezőkön $\binom{24-w}{b}$ -féleképpen helyezhetjük el a fekete korongokat, amely elhelyezkedésekhez szintén eltárolhassunk lookup table-ben sorszámokat. Így a hash függvény által használt képlet:

$$\text{hash}(W, B) = f(W) \binom{24-w}{b} + g(B, W)$$

ahol W és B a fehér illetve a fekete korongok halmaza, az f függvény a fehér korongokhoz tartozó lookuptable-t használja, a g függvény pedig először eltünteti a táblának a fehér korongokat tartalmazó mezőit, majd pedig kiolvassa a fekete korongoknak a maradék mezőkön lévő elhelyezkedéséhez tartozó sorszámot.

Vegyük észre, hogy a szimmetriák nagy részének a kiszűrése megtehető úgy, hogy egysze-

rően az f lookup table-ét úgy építjük föl, hogy a fehér korongoknak a szimmetriaműveletekkel egymásba átvihető elhelyezkedéseikhez azonos számot rendeljen. Továbbá be kell még iktatni egy plusz lépést a képlet alkalmazása elé: ki kell jelölni minden szimmetriával egymásba átvihető fehér korongelhelyezkedés-halmazból egy kanonikus elhelyezkedést (pl. a lexikografikusan első), és hasheléskor a teljes állásra kell alkalmazni egy olyan transzformációt, hogy a fehér korongok ebbe a kanonikus elhelyezkedésbe kerüljenek. Erre azért van szükség, mert különben a fekete korongok lookup table-ébe rossz helyen indexelnénk bele. Ekkor csak olyankor fog előfordulni, hogy mégis különböző értéket rendel a hash függvény szimmetriával egymásba átvihető állásokhoz, amikor a fehér korongoknak van olyan szimmetriája, amely a feketéknek nincsen. Az emiatt a hash függvényben nem kiszűrt állások aránya a standard malomban 3,95%, a laskerben 4,2%, a morabarabában pedig 3,48%.

A fentebbi hash függvény használatakor föllép egy nem nyilvánvaló probléma. Nevezzük nagy ekvivalenciaosztályoknak a szimmetriaműveletek által kijelölt részhalmazokat, továbbá nevezzük kis ekvivalenciaosztályoknak a hash függvény által kijelölt részhalmazokat (azaz azon állások kerülnek azonos kis ekvivalenciaosztályba, amelyeknek azonos a hash-e). A kis ekvivalenciaosztályok a nagyokat osztják föl, azaz nincs olyan kis osztály, amely több különböző nagy osztályból tartalmazna állásokat. Nevezzünk egy állásra nézve egy szimmetriaműveletet kicsinek, ha nem viszi ki az állást a kis ekvivalenciaosztályából, továbbá nevezzük nagynak, ha nem kicsi. Ekkor a probléma a következő: Ha az A állás egy B megelőzőjének a nagy ekvivalenciaosztályában van egy olyan B' állás, amelybe egy olyan m szimmetriaművelettel juthatunk, amely B -re nézve nagy, de A -ra nézve kicsi, akkor B' -nek nem vesszük figyelembe azt a lépést, amellyel abba az állásba juthatunk belőle, amelyet A -ból kapunk m -et alkalmazva.

Két megoldási lehetőséget találtunk ki erre a problémára. Az egyik, hogy terjesztéskor legeneráljuk a megelőzőknek a nagy ekvivalenciaosztályának a kis ekvivalenciaosztályait, és az ezekhez tartozó hash-ek mindegyikénél elvégezzük a terjesztés hatását. A másik, hogy „tökéletesítjük a hash normalizálását” a következőképpen: minden nagy ekvivalenciaosztályból kijelölünk a hozzá tartozó hash-ek közül egy kanonikusát, az állások értékelését tároló tömbnek a nagy ekvivalenciaosztály többi hash-e által kijelölt pozícióira pedig bejegyezzük (speciális értékek segítségével), hogy mely szimmetriaművelettel kerülhetünk át a kanonikus hash-hez tartozó kis ekvivalenciaosztály egyik állásába. Így amikor a hash alapján bele kéne indexelnünk az értékeket tároló tömbbe, akkor mindig áttérünk a kanonikus hash által mutatott pozícióra (ha nem épp abban vagyunk). Ez utóbbi megoldást választottuk, mivel ez tűnt gyorsabbnak.

A countok inicializációja során a rákövetkezők számát úgy határoztuk meg, hogy minden állásnak generáltuk az összes megelőzőjét, és ezeknek megnöveltük 1-gyel a countját. Ha a rákövetkezők számát ehelyett úgy határoztuk volna meg, hogy minden állásnak generáljuk az összes rákövetkezőjét, és ezeket közvetlenül megszámloljuk, akkor föllépne még egy probléma, olyankor, amikor egy lépés két különböző szimmetriatulajdonságú állás között vezet. Pl. tegyük fel, hogy egy A állásból egy m kis szimmetriaművelet átvisz egy A' állásba, viszont egy B állást, amelybe át lehet lépni A -ból, m helyben hagy. Ekkor az $A \rightarrow B$ élt egyszer számoljuk,

amikor A -nak megszámloljuk a rákövetkezőit, viszont kétszer csökkentjük A -nak a countját, amikor terjesztünk B -ből, hiszen B megelőzőinek a generálásakor megjelenik A és A' is. Ezt a problémát a rákövetkezők számának fentebb leírt „fordított” inicializálásával oldottuk meg.

3. Ultra erős megoldás

3.1. Választás a „tökéletes” lépések közül

Miután az alapvariánshoz meghatároztuk az összes állás játékelméleti értékét és távoliságát, azt tapasztaltuk, hogy az így létrejött adatbázist használó, „mindig tökéleteset lépő” program – némileg meglepő módon – nem játszik igazán „erősen”, olyan értelemben, hogy könnyű ellene döntetlent elérni (legyőzni nem lehet, hiszen a kezdőállás értéke döntetlen). Ennek a fő okának az tűnik, hogy a malom játékban nagyon sok döntetlen értékű állás van, és így sokszor előfordul, hogy többféle lépés is optimális (az állások játékelméleti értékét tekintve), és így ezek közül nem nehéz egyet megtalálni az ellenfélnek. Emiatt már átlagos emberi játékosokkal való játék során is előfordul, hogy a program nem nyeri meg a játszmat, hanem csak döntetlent ér el.

A csupán a játékelméleti értéket figyelő programunkat teszteltük egyrészt saját magunk ellen, másrészt nálunk sokkal erősebb emberi játékosok ellen is (egy olyan weboldalon, ahol a játékosok pontozva vannak az Élő-pontrendszer alapján), továbbá egy heurisztikus program ellen, amely részleges játékfakiértékelést végez alfa-béta vágással, transzpozíciós táblával és az ehhez szokásos egyéb technikákkal.

A tesztelések során szembetűnő volt, hogy a „tökéletes” program sok esetben már a játszma korai szakaszában kifejezetten gyengének látszó állásba jutott. Persze annyira nem voltak rosszak ezek az állások, hogy a program veszítsen, viszont ezekben az állásokban az ellenfélnek sok olyan lépéslehetősége volt, amivel fenn tudta tartani a döntetlent. Néha ugyan előfordult, hogy akár egy erős játékos is hibázott, mivel megjelentek olyan állások, amelyekről nagyon nehéz volt ember számára látni, hogy mi a helyes lépés, azonban nem ezek voltak a jellemzőek. Többnyire könnyen találhatott egy ember is egy optimális lépést.

Mivel arra törekedtünk, hogy a programunk le is győzze az ellenfeleit a pusztá döntetlen helyett, így többféle megoldással is kísérleteztünk a fenti probléma kiküszöbölésére.

Egy egyszerű lehetőség, hogy a részleges játékfakiértékelő heurisztikus programban úgy használjuk föl az adatbázist, hogy a legfelső szinten csak optimális lépéseket tekintsen (ezt Gasser is említi a disszertációjában [5]). Ez be is vált, a két program ilyen módon kombinált változata erősebb mindkettőnél, hiszen nem veszti el sosem a játszmat, és többször nyeri meg, mint a pusztán az adatbázisra támaszkodó program.

Egy másik egyszerű próbálkozás az volt, hogy az optimális lépések közül válasszuk azt, amelyik minimalizálja a kialakuló állásban az optimális lépések számát (NOM - Number of Optimal Moves), hátha így nehezebben fog optimális válaszlépést találni az ellenfél. Ez valamennyit javított a helyzeten, azonban az a probléma, hogy a NOM nagyon nagy mértékű

ingadozást mutat a játszma egymás után következő néhány lépésénél. Vagyis az, hogy az egy lépés távolságra lévő állásban kicsi a NOM, nagyon kevésbé korrelál azzal, hogy a következő néhány lépésben is alacsonyban fogjuk tudni tartani. Ráadásul sokszor megfigyelhető, hogy az alacsony NOM nem jelenti azt, hogy igazán nehéz dolga lenne az adott állásban egy embernek. Pl. a felrakás során az olyan állásokban, amikor egy malom elkészülése fenyeget a következő lépésben, a NOM sok esetben 1. Viszont ilyenkor többnyire az a helyes lépés, hogy elzárjuk a készülő malmot, ami mindenkinek az első gondolata lesz.

Egy további probléma, hogy egyes állásokban, bár alacsony a NOM, de az össz lépéslehetőség is kevés, így az ellenfélnek mégse nehéz optimális lépést találnia. Ennek kiküszöbölésére használható a ROM, azaz Ratio of Optimal Moves, ami az optimális lépések száma, osztva az összes lépéslehetőségek számával. Azonban még mindig fenn áll a lehetőség, hogy az aránylag kevés optimális lépés triviálisan szembetűnő az ellenfélnek. Ennek elkerülésére próbálkoztunk a WROM-mal, azaz Weighted Ratio of Optimal Moves-zal, ami az optimális lépések súlya, osztva az összes lépéslehetőségek súlyával. Ideális esetben a lépés súlya arányos azzal a valószínűséggel, hogy az ellenfél azt választja; ebben az esetben a WROM egy adott állásra azt a valószínűséget adja meg, hogy az ellenfél optimális lépést lép. Ezt minimalizálva a program igyekszik nehezen átlátható, vagy éppen megtévesztő állásokat választani. A lépések súlyát az alábbi heurisztikával számítottuk ki: a lépést követő állásra meghívtuk kis mélységgel az alfa-bétás algoritmust, majd az állásra adott értékelését transzformáltuk. Erre egy lehetséges képlet a softmax függvényen alapuló $e^{\alpha h}$, ahol a h a heurisztikus helyzetértékelés (mely körülbelül a korongkülönbség skáláján mozog), az α pedig egyfajta mohósági paraméter. ($\alpha = 0$ esetén a ROM-ot kapjuk vissza, ha pedig α tart a pozitív végtelenbe, csakis a heurisztikusan legjobbnak tűnő lépések optimalitását vesszük figyelembe.)

A fentebbi heurisztikák mindegyikéből készíthető egy rekurzív változat, amely kevésbé móhó, azaz nem csak egy lépésre előre igyekszik az adott heurisztika értékét minimalizálni, hanem azt éri el, hogy a következő néhány lépés során tekintett átlagát vagy szorzatát az adott heurisztikának minimalizálja.

A fentebbieket, illetve hasonló módszereket Jansen részletesen elemzett [9] sakkvégjátékokra, illetve Lincke a malom játék esetében [10].

Mivel a lokális heurisztikákat használó próbálkozásaink nagyon csekély eredményre vezettek, ezért kidolgoztuk a következő fejezetekben leírt módszert arra, hogy már az adatbázis számítása során meghatározzuk az állásoknak egy olyan jellemzőjét, amit aztán felhasználhatunk a különböző döntetlen állások közti választásra.

3.2. A Retrográd analízis háromnál többféle kimenetelt is kezelő változata

Ugyan a retrográd analízist bemutató fejezet elején vázoltuk az algoritmusnak egy olyan változatát, amely nem csak a nyeréseket, veszteségeket, illetve a döntetleneket képes megkülönböztetni, hanem tetszőleges értékeket képes visszaterjeszteni a végállapotokból, azonban az algoritmus-

nak a játékgráfban lévő köröket is kezelő változata nem tudja ezt. Mivel a későbbiekben a döntetlenek megkülönböztetéséhez szükségünk lesz rá, ezért most vázoljuk az erre az esetre is működő, általunk kidolgozott általánosítását az algoritmusnak.

Az ötlet az, hogy az állások értékének meghatározásában haladjunk a szélsőségesebb értékű állások felől a 0 értékű állások felé. Pl. ha a lehetséges legnagyobb abszolútértékű nyerés illetve vesztes értéke 10 és -10, akkor kezdjük azzal, hogy meghatározzuk a 10 abszolútértékű állásokat. Ezután meghatározzuk a 9 abszolútértékű állásokat, majd a 8-asokat, stb. Ennek során az aktuális értéknél kisebb abszolútértékű állások hasonlóan viselkednek, mint a korábbi algoritmusnál a döntetlenek, továbbá az aktuális abszolútértékű állások a korábbi nyerések és veszteségek szerepét játsszák. (A 0 abszolútértékű állások végül ugyanúgy kaphatók meg, mint korábban, vagyis ezeket nem dolgozzuk föl)

A különböző értékű végállapotokat kezdetben a megfelelő sorrendben rakjuk be a sorba, és a szektorokra bontó algoritmusnál leírt, két sor összefésülését használó megoldással érjük el, hogy a megfelelő pillanatban kerüljenek ki a sorból (a rendezés első kulcsa az abszolútérték ellentettje, a második pedig a távolság).

A korábbi algoritmusnál kihasználtuk, hogy amikor egy s állás rákövetkezői közül akár már egyről is kiderült, hogy vesztes, akkor rögtön tudjuk, hogy az s állás mindenképpen nyerés lesz, hiszen a további maximumkeresés úgysem találhatna nyerésnél jobb értéket. Most is lesz egy hasonló jelenség, amely rámutat a kívülről befelé haladás jelentőségére.

Tegyük fel, hogy most a v abszolútértékű állásokat dolgozzuk föl. Ekkor, ha egy s állás rákövetkezői között meglátunk egy $-v$ abszolútértékű állást, akkor már tudjuk, hogy az s állás értéke biztosan v . Azért nem találhatna a maximumkeresés v -nél nagyobb értéket, mert akkor már a futásnak egy korábbi pillanatában meg kellett volna határoznunk s értékét, hiszen a $-v$ -nél kisebb értékű állások mindegyike kikerült már a sorból, tehát s -nek annak a rákövetkezőjéből is terjesztenünk kellett volna már, amelyből származhatna a v -nél nagyobb érték. Továbbá abban a pillanatban, amikor egy s állásnak már az összes rákövetkezőjéből terjesztettünk, és egyiknek se volt negatív az értéke, akkor s -hez bejegyezhetjük a $-v$ értéket, hiszen eddig csak legalább $+v$ értékűekből terjesztettünk bele.

Eddig azt láttuk be, hogy amikor kitöltjük egy állás értékét, akkor tényleg a valódi értékét írjuk bele. Most azt mutatjuk meg, hogy nem fordulhat elő az, hogy egy állás értékét már be kellett volna írunk, de még ezt nem tettük meg. Amikor már befejeződött a v abszolútértékű állások értékének kitöltése, akkor azoknak az állásoknak az értékének az abszolútértéke, amelyeket eddig nem töltöttünk ki, legfeljebb $v - 1$ lehet a valóságban is, hiszen nem lehetséges, hogy egy ilyen állásnak minden rákövetkezője legalább v értékű, vagy hogy van olyan rákövetkezője, amely legfeljebb $-v$ értékű.

A fenti algoritmus a távolságot továbbra is minimalizálja a „nyerő” (pozitív értékű) állásokban, és maximalizálja a „vesztő” (negatív értékű) állásokban. Azoknak az állásoknak pedig nincs távolságuk, amelyek countok maradnak.

A futásidő kis mértékben megnő, amiatt, hogy a vödrös rendezésnek nagyobb értékkészlettel

kell dolgoznia: $\mathcal{O}(E + VD + SRB)$, ahol R az első kulcs értékkészlete.

Ez az algoritmus minden olyan játéknál használható, amelynek megvannak a malomnak a bevezetésben leírt tulajdonságai, de háromnál többféle kimenetele is lehet, pl. Awari, Othello.

Végiggondolható, hogy a korábbi pszeudokódot kis változtatással átírhatjuk a most ismert algoritmusra:

- A sorműveleteknél használt összehasonlító operátornak első kulcsként az állás értékének az abszolútértékének az ellentettjét kell figyelembe vennie, és csak a második kulcs lesz a távoliság. Vagyis az azonos abszolútértékű állásokat (a régi algoritmusnak megfelelően) a távoliságok szerinti sorrendben dolgozzuk föl.
- Terjesztéskor az első kulcsot negálni kell.

3.3. A döntetlenek megkülönböztetése a végső szektor heurisztikus értéke alapján

Nevezzünk egy döntetlen értékű állást stabilnak, ha van olyan optimális játék belőle indulva, amelynek során nem lépünk már ki az állást tartalmazó munkaegységből. (Nem minden döntetlen állás stabil, hiszen pl. a felrakásos szektorokbeli döntetlen állások nyilván nem stabilak, továbbá sok olyan döntetlen értékű állás van egyéb szektorokban is, amelyeknél mindkét oldal csak úgy tudja fenntartani a döntetlent, hogy támad és levesz korongot, amikor is átkerül a játszma más munkaegységbe. Továbbá figyeljük meg azt is, hogy nem minden stabil állás része egy körnek azon a gráfon, amelyet úgy kapunk, hogy a játékgráfot leszűkítjük az optimális lépésekre.)

A döntetlenek osztályozásának alapötlete az, hogy ha már nem látunk nyerő lépést, akkor legalább igyekezzünk olyan döntetlenek felé menni, amelyekből végül olyan stabil állásokba juthatunk, amelyben az ellenfél nehéz helyzetben van (valamilyen heurisztika szerint). Ésszerűnek tűnik a feltételezés, hogy már az ilyen stabil állásokba való eljutás során is az ellenfél számára nehéz helyzetek fognak kialakulni. Ezt később az eredmények is alátámasztották: pl. amikor már a programunk olyan állásba tudott jutni, amelyből az optimális játszma a 6,3,0,0-ba vezet, akkor nagyon sokszor már ezen végső szektor elérése előtt vétett hibát az ellenfél.

Vagyis a stabil állásokhoz egy heurisztikus értéket rendelünk, és úgy módosítjuk a retrográd analízist, hogy az elérhető legjobb stabil állás értéke alapján különböztesse meg a döntetlen állásokat. Az állások játékelméleti értéke így kiegészül azzal, hogy mennyire jó stabil állásba juthatunk el, amennyiben a játékosok erre optimalizálnak, vagyis a korábban lehetséges nyereség, veszteség és döntetlen értékek közül a döntetlen szétválik több lehetséges értékre.

Első közelítésben elképzelhetünk egy olyan algoritmust, hogy a stabil állásokhoz rendelünk egy heurisztikus értéket, és aztán ezeket terjesztjük vissza retrográd analízissel, azonban ezzel van egy probléma: jó lenne, ha a régi értékelések szerinti stabil állások közül az új algoritmusban nem kéne mindnek végállapotnak lennie, ugyanis lesz olyan, a régi értékelések szerint

stabil állás, amely az új értékelések figyelembe vételével már nem lesz stabil, mert valamelyik játékos kikényszerítheti, hogy egy számára jobb, régi értékelés szerint stabil állásba kerüljön át a játszma. Ezért kidolgoztunk egy olyan algoritmust, amelynek használatakor nem kell előre lerögzíteni, hogy melyek a stabil állások.

3.3.1. A stabil állásokhoz használt heurisztika

Nevezzük *tranzienseknek* az olyan szektorokat, amelyeknek a munkaegységéből mindenképpen azonnal kilépünk. Ezek standardnál és morabarabánál a felrakásos szektorok, laskernél pedig csak azok a szektorok, amelyeknél valamilyen színű korongból nincs fönt a táblán. Nevezzünk egy szektort *EKS*-nek, ha egyenlő a két játékos korongjainak a száma a táblán és a fölraakhatóak közt is, azaz önmaga negáltja.

Először azt ismertetjük, hogy mi lesz az a heurisztika, amely irányítja, hogy mely állások felé törekszünk, azaz megmondja a stabil állások értékét. Az általunk kidolgozott algoritmus olyan esetben működik, ha a szektorokhoz rendelünk értékeket, és a stabil állások értéke egyszerűen az őt tartalmazó szektor értéke. Természetes gondolat lehet a korongkülönbség használata (azaz, hogy mennyivel van több korongom, mint az ellenfélnek). A végső eredmények alapján úgy tűnik, hogy talán ez sem lett volna sokkal rosszabb heurisztika, mint amit végül használtunk, azonban ezt nem láttuk előre.

Ehelyett a következőt csináltuk: a már meglévő adatbázisból nyert statisztikák alapján hozzárendeltünk minden szektorhoz egy heurisztikus értéket. Ezután ez alapján sorba rendeztük az összes szektort, és az ezen rendezésben kapott sorszáma alapján hozzárendeltünk minden szektorhoz egy *rangot*. A rangok középpontja a 0, és kell tenni egy korrekciót: a nem tranzien্স munkaegységekben lévő szektorok rangjának egymás ellentettjének kell lennie. Emiatt a nem tranzien্স EKS szektorok értékét sajnos 0-ra kellett korrigálnunk, valamint lyukak is keletkeztek a rangokban. Ezek a rangok lesznek a végleges szektorértékek. A rangokra áttérésre azért volt szükség, mert a szektorértékeknek a tárolás szempontjából célszerű kicsi egész számoknak lenniük.

A következő képletet alkalmazzuk a nem tranzien্স szektorok értékének meghatározására. Azt ragadja meg, hogy számunkra azok a jó szektorok, amelyekben egyrészt sok nyerő és kevés vesztes állás van, másrészt amelyeknek a negált szektorában (ahol az ellenfél jön) ez pont fordítva van (egyébként később látni fogjuk, hogy az algoritmus csak akkor működik helyesen, ha az egy nem tranzien্স munkaegységen belüli két szektor értéke egymásnak ellentettje):

$$val_s = (W_s + L_{-s} + D_s/2 + D_{-s}/2)/(T_s + T_{-s})$$

ahol a W_s , L_s , D_s változók a szektorban lévő nyerések, veszteségek, illetve döntetlenek, T_s pedig az összes állások száma.

Tranziens szektoroknál a képlet hasonló, azonban ez csak egy szektort vesz figyelembe:

$$val_s = (W_s + D_s/2)/T_s$$

Vegyük észre, hogy a tranziens szektorértékek nem lehetnek hatással az állások értékére, mert nem lehet bennük stabil állás. Viszont szerepük lesz a távolságokban, ld. a 3.4.1 alfejezetben.

A tesztelések alapján úgy tűnt, hogy a fentebbi heurisztika túl nagy értéket rendelt a 8,9,0,0 szektorhoz. Pl. a program jobban szeretett volna ebbe a szektorba jutni, mint a 6,3,0,0-ba. Végülis manuálisan lecsökkentettük a 8,9,0,0 rangját, újraszámoltuk az adatbázist, és a program valóban erősebb lett.

A nyeréseknek és vesztesékeknek az értéke a rangok értékkészletén kívül lesz. Innentől az állások játékelméleti értéke az ezekre is kiterjesztett értékkészletből fog kikerülni.

A következő fejezetben ismertetjük a retrográd analízisnek azt az általunk kidolgozott változatát, amely kiszámítja az állások játékelméleti értékén túl azt is, hogy a döntetlen állásokból optimális játék esetén milyen értékű stabil állásba juthatunk el, anélkül, hogy előre rögzítenénk, hogy melyek a stabil állások.

3.4. A retrográd analízis döntetleneket megkülönböztető változata

Ez az algoritmus az előző alfejezetben leírt heurisztikát fogja használni, és a 3.2 alfejezetben leírtakon fog alapulni. Idézzük föl, hogy egy álláshoz két dolgot tárolunk: a játékelméleti értéket, és a távolságot. Ezekre első, illetve második kulcsként fogunk hivatkozni. A nehézséget az jelenti, hogy a korábbi algoritmusnál a stabil állások bejegyzései mindig csak countok lehetnek (amelyek tekinthetők impliciten 0 értékűnek), azonban most arra lenne szükségünk, hogy ezekhez különböző értékeket rendelhessünk. Ezt úgy fogjuk elérni, hogy relatív értékekkel számolunk: az állásokhoz az lesz bejegyezve, hogy az aktuális szektor értékéhez képest mennyivel jobb vagy rosszabb értékű szektorba juthatunk el, és így a stabil állások abszolút nézetből a megfelelő value-nak fognak látszani (kivéve persze a 0 értékű szektorok esetén, hiszen az értékük 0 lenne). Ehhez arra van szükség, hogy amikor egy állás értékét terjesztjük két szektor között, akkor korrigáljuk az értékét: először áttérünk az abszolút nézetre, aztán megteesszük a szokásos negálást, majd pedig az új szektor értékére korrigálunk. Korrigáláskor a countokat 0 értékűnek tekintjük, és ha egy value 0-ra korrigálódik, akkor countnak tekintjük (a beleírt érték ilyenkor mindegy, ugyanis csak mellékszektorbeli állásnál fordulhat ez elő, vagyis nem terjeszthetünk bele).

Ez implementálható pl. úgy, hogy már akkor megteesszük a korrekciókat, amikor a mellékszektorokat betöltjük a memóriába. Ehhez a tartalmazó mellékszektor és a megfelelő főszektor értékének összegét kell hozzáadni az állások értékéhez. Ez akkor tehető meg kényelmesen, ha minden mellékszektorbeli állás csak egy főszektorba lesz terjesztve. A malom variánsainál ez teljesül, más játékoknál pedig az 5.1. alfejezetben leírtakkal érhetjük ezt el. A két sornál használt rendezésnek a fentebbiek szerint módosított értékeket kell figyelnie. Vegyük észre, hogy

főszektorok közti terjesztésnél nincs szükség korrigálásra, mivel a szektorok értékeit úgy határoztuk meg, hogy a nem tranziens munkaegységek két szektorának az értékeinek az összege 0 legyen. (Továbbá meggondolható, hogy ez a minimax elv miatt is szükséges az algoritmus működéséhez.)

3.4.1. A távoliság definiálása döntetlenekre

A távoliság definíciójának kiterjesztése nem nyilvánvaló a döntetlen állások esetére, ugyanis nem világos, hogy mikor kinek célszerű maximalizálásra, illetve minimalizálásra törekednie. Azt kell végiggondolni, hogy tulajdonképpen mi is a célja a két játékosnak a távoliságra optimalizálással. A hagyományos retrográd analízisnél azért kell a nyerő játékosnak minél közelebb hoznia a játszma végét, hogy ne történhessen meg az, hogy annak ellenére, hogy nyerő állásban vagyunk, nem tudjuk megnyerni a játszmát, mert időről időre mindig eltávolodunk tőle. Tökéletes játékosok esetén a veszteső játékosnak valójában nem válik előnyére, hogy távolítani igyekszik a játszma végét, azonban azért van erre mégis szükségünk az algoritmusban, mert enélkül nem tudnánk megoldani, hogy a nyerő játékos a közelítésre optimalizáljon, hiszen a nyerő játékos szemszögéből a „legrosszabbra kell fölkeszűlnünk”.

Most az a kérdés, hogy amikor az állás értéke döntetlen, és így nincs nyerő vagy veszteső játékos, akkor közelítsünk vagy távolítsunk? Amint azt az előző alfejezet bevezetőjében említettük, arra is számítunk, hogy egy jobb értékű szektorba való törekvés azért is előnyös, mert már az oda vezető út során is nehezebb helyzetben lesz az ellenfél. Vagyis annak is van jelentősége, hogy az oda vezető úton mennyi időt töltünk milyen értékű szektorokban: sokat lépegetve egy számunkra jó szektorban nagyobb eséllyel hibázik az ellenfél. Vagyis azokban a helyzetekben, amikor az állás értékénél jobb értékű szektorban vagyunk, maximalizálni célszerű a távoliságot. Viszont ahhoz, hogy ezt meg tudjuk tenni, minimalizálnunk kell a távoliságot olyankor, amikor rosszabb értékű szektorban vagyunk (meg egyébként az is megtörténhetne különben, hogy ott ragadunk benne, ld. a hagyományos távoliságminimalizálás okát).

Ez a gondolatmenet azt sugallja, hogy terjesztéskor a távolisághoz – a szokásos módon – mindig hozzáadunk 1-et, és amikor arra kerül a sor, hogy két (azonos értékű) value-t a távoliság szerint kell összehasonlítani, akkor megnézzük, hogy most jobb vagy rosszabb szektorban vagyunk-e, mint a végcél (vagyis az állás relatív értéke pozitív vagy negatív), és eszerint döntjük el, hogy a nagyobb vagy a kisebb távoliság a jobb. (Ez megfelelne annak, ahogy a 3.2. alfejezetben leírt algoritmus kezeli a távoliságokat.) Azonban ezzel az a probléma, hogy így nem vennénk figyelembe azt, hogy attól, hogy most rosszabb (vagy jobb) szektorban vagyunk, mint a végcél, nem biztos, hogy végig ugyanilyen tulajdonságú szektorokon át fog haladni az optimális játszma: előfordulhat, hogy az optimális játszma során a végcélhoz képest hol jobb, hol rosszabb szektorban vagyunk. Ennek figyelembe vételéhez azt csináljuk, hogy amikor korrigáláskor az állás értéke előjelet vált, akkor negáljuk a távoliságot. Így az optimális útnak azok a lépései, amelyeket ellentétes optimalizálású szektorban teszünk, ellentétes előjellel fognak számítani. Pl. tegyük fel, hogy két út, a és b közül kell választani, és az a út során egy, a végcélnál

rosszabb szektorban 1-gyel több lépést teszünk. Ekkor a b út a jobb. Ha viszont továbbá az a úton egy, a végcélnél jobb szektorban is több lépést teszünk, mégpedig 2-vel, akkor már az a út a jobb.

Föllép egy apró kellemetlenség, amikor az optimális út során már a végső s_1 szektor elérése előtt elérünk egy, a végső szektorral azonos értékű s_2 szektort. Az ilyen utaknak csak az s_2 előtti részében tett lépéseit veszi figyelembe a távolság. Ennek az az oka, hogy az s_2 -ben countoknak fogjuk látni az abszolút nézetben s_1 értékű állásokat, és ezért elvesztjük a mellékszektorokból jövő távolságinformációkat. Mindazonáltal, ez nem tűnik nagy problémának.

Tehát a g állás fentebbiek szerint számolt távolságát a következő képlet adja meg:

$$d(g) = \text{sgn}(k1(g)) \sum_{i=0}^{n-1} \text{sgn}(k1_g(g_i)),$$

ahol g_i a g -ből induló optimális út állásai, n az optimális út első countjának vagy végállapotának sorszáma, $k1$ egy állás relatív első kulcsát adja meg, $k1_g$ pedig ennek még az ellentettjét veszi, amennyiben nem az a játékos következik, amelyik g -ben.

Ha nem tesszük föl, hogy ismert az optimális út, akkor a távolság a következő képlettel adható meg:

$$d(g) = \text{sgn}(k1(g)) \min_{p \in P_g} \sum_{v \in p} \text{sgn}(k1_g(v)),$$

ahol P_g a g -ből induló, első count vagy végállapot előtti állásig tartó, abszolút első kulcs szerinti optimális utak halmazát jelöli, v pedig egy állás egy úton.

Lehetne még egy kicsit javítani a heurisztikán, ha a távolságot nem mindig 1-gyel növelnénk, hanem egy annak megfelelő számmal, hogy mennyivel vagyunk jobb (vagy rosszabb) szektorban, mint a végcél. Azonban a szektorértékek skálája – főként gyakorlati okokból – ordinális, ami nem illeszkedne jól ehhez, úgyhogy ezt nem implementáltuk.

Megjegyezzük, hogy ha a távolságot nem a játék végéig mérnénk, hanem az első koronglevételig, akkor a fentebbi bonyodalmak elkerülhetők lennének.

4. Eredmények

A kiszámított adatbázisok elérhetők a <http://compalg.inf.elte.hu/~ggevay/> címről.

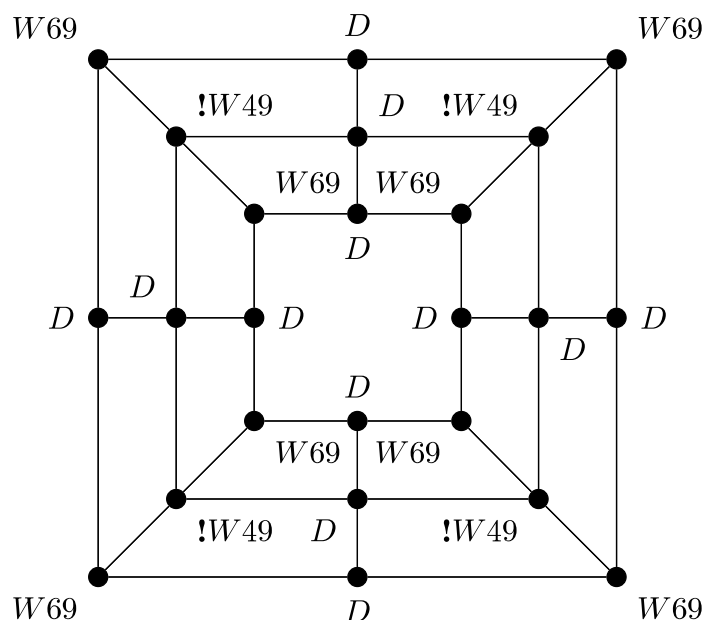
Az 1. táblázat statisztikákat mutat a variánsokról. Megjegyezzük, hogy az értékek azért térnek el a Stahlhacke által írtaktól [6], mert egy kicsit más szabályokat használunk.

4.1. Morabaraba

A morabaraba játékelméleti értéke nyerés, 49 lépésben. (A másik két variáns döntetlen.) A 2. ábra a morabaraba kezdőállásában mutatja az összes lehetséges lépés értékét. A tábla szimmet-

Variáns	Állások száma (kiterjesztett)	Nyerés/Döntetlen/Vesztés %	Max. távolság
Standard	27 Mrd (284 Mrd)	53,6 / 22,3 / 24,1	206
Lasker	133 Mrd (398 Mrd)	52,5 / 14,3 / 33,2	214
Morabaraba	112 Mrd (284 Mrd)	60,7 / 4,6 / 34,7	124

1. táblázat. Statisztikák a három variánsról



2. ábra. Az összes lépéslehetőség játékelméleti értéke távolságokkal a morabaraba kezdőállásban (W nyerést jelent, L vesztesét, D pedig döntetlent). Az optimális lépéseket ! jelzi.

riái miatt ezek közül csak négy tekintendő különböző lépésnek. Ezek közül kettő döntetlen, a másik kettő pedig 49 illetve 69 lépésben nyeres.

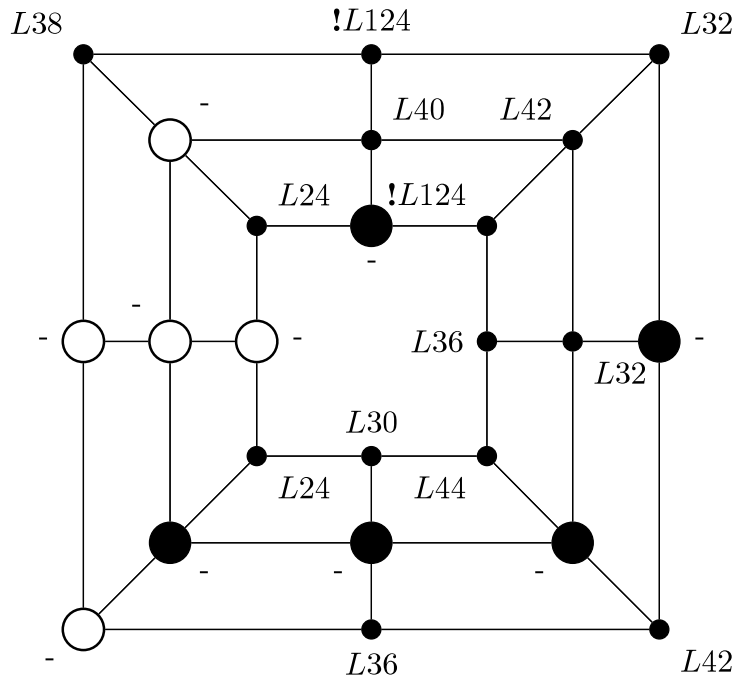
A 3. ábra a legnagyobb távolságú állást, a 4. ábra a távolságok eloszlását, míg az 5. ábra egy optimális játszmát mutat.

4.2. Kiterjesztett megoldások

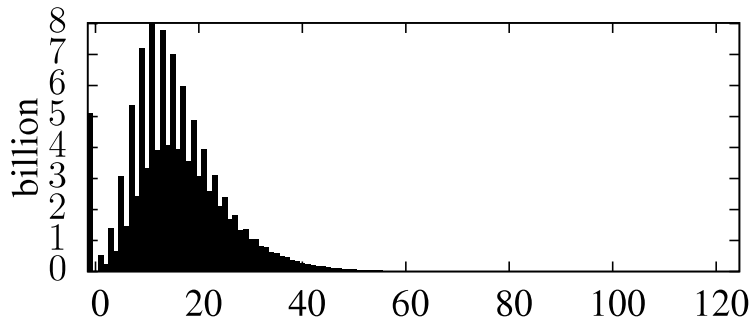
A 2., 3. és 4. táblázat a kiterjesztett megoldás során vizsgált kezdőállások játékelméleti értékét mutatja. Uiterwijk és van den Herik azt vizsgálta, hogy mennyi előnye van a soron következő játékosnak mnk -játékokban⁵ és a domineering játékban [16]. A malom variánsok kiterjesztett megoldás-táblázatai is vizsgálhatóak ilyen szempontból.

Először koncentráljunk a főátlókon kívüli mezőkre. A táblázatok főátlói alatt, illetve fölött olyan állások értékei helyezkednek el, ahol fehérnek, illetve feketének több felrakható korongja van. Jól látható, hogy akár csak eggyel több korong is nagy előnyt jelent, hiszen pl. teljesen dominálja azt a szempontot, hogy ki kezd: nincs olyan kezdőállás, ahol az a szín nyer, amelyik kevesebb korongot rakhat föl.

⁵Az mnk -játékok az amőba általánosításai $m \times n$ -es táblaméretre és 5 helyett k hosszúságú sorozatokra.



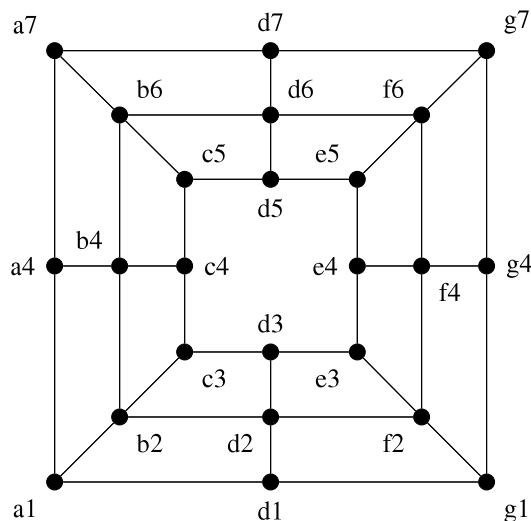
3. ábra. A morabaraba legnagyobb távolságú állása (124 lépés). A játékosok még fölrahatnak 7-7 korongot.



4. ábra. A távolságok eloszlása a morabarabánál. A legelső oszlop a döntetlenek számát mutatja. Minden második oszlop páros távolságot, azaz veszteszt reprezentál. Látható, hogy ezek alacsonyabbak a szomszédos oszlopoknál, mivel előnye van annak a játékosnak, aki éppen következik.

f2, f6, b6, b2, a1, c5, g1, e3, d1xf6, f6, g7, g4, a7, e4, e5, f4xe5, a4xe3, e3, e5, c3, d3, c4xd3, d3, d2, a7-d7, c4-b4, d7-a7xc5, f6-d6, g7-f6, g4-g7, g1-g4, c3-c4, a7-d7, c4-c3, g4-g1xg7, f4-g4, f6-f4, d6-d5, b6-a7xg4, d5-d6, e5-f6xd6, b4-c4, f6-g7xe3, c4-b4, g7-f6xe4, b4-b6, f6-g7xb2, c3-e5, f4-g4xd2

5. ábra. Egy optimális morabaraba játszma. A jelöléseket ld. a 6. ábrán.



6. ábra. A mezők jelölése.

	3	4	5	6	7	8	9	10	11	12
3	W23	D	D	L12	L12	L12	L12	L12	L12	L12
4	D	D	D	D	L20	L16	L16	L16	L16	L16
5	W9	D	D	D	L24	L22	L20	L20	L20	L20
6	W9	D	D	D	L34	L26	L24	L24	L22	L22
7	W9	W17	W25	W33	D	L36	L28	L26	L26	L24
8	W9	W15	W21	W25	W33	D	L36	L30	L28	L26
9	W9	W15	W19	W23	W27	W35	D	L36	L32	L28
10	W9	W15	W19	W23	W25	W29	W35	D	L36	L28
11	W9	W15	W19	W21	W25	W27	W29	W31	L44	L28
12	W9	W15	W19	W21	W23	W25	W27	W27	W27	*L26

2. táblázat. A malom különböző kezdőállásainak játékelméleti értéke és távolisága. A sorok és oszlopok első száma rendre a fehér, illetve fekete által fölrakható korongok számát mutatja. A *-gal jelölt mező döntetlenné változik, ha használjuk azt a szabályt, hogy a teli tábla döntetlent jelent. (Ez az egyetlen eset a három variánsban, amikor ez a szabály változtat egy kezdőállás értékén.)

	3	4	5	6	7	8	9	10	11	12
3	W23	D	D	L12	L12	L12	L12	L12	L12	L12
4	D	D	D	L24	L16	L16	L16	L16	L16	L16
5	W9	D	D	D	L24	L20	L20	L20	L20	L20
6	W9	W19	D	D	L40	L26	L24	L22	L22	L22
7	W9	W15	W21	W39	D	L40	L30	L28	L26	L26
8	W9	W15	W19	W25	W33	D	L44	L34	L32	L30
9	W9	W15	W17	W23	W29	W35	D	L46	L36	L34
10	W9	W15	W17	W21	W25	W31	W37	D	L44	L38
11	W9	W15	W17	W21	W25	W29	W33	W39	D	L46
12	W9	W15	W17	W21	W25	W29	W31	W35	W41	D

3. táblázat. A lasker különböző kezdőállásainak játékelméleti értéke és távolisága.

	3	4	5	6	7	8	9	10	11	12
3	W9	D	L12	L12	L12	L12	L12	L12	L12	L12
4	W9	D	D	L20	L18	L16	L16	L16	L16	L16
5	W9	D	D	L22	L20	L20	L18	L18	L18	L18
6	W9	W17	W23	D	L26	L26	L24	L24	L22	L22
7	W9	W13	W19	W25	W45	L32	L30	L26	L26	L28
8	W9	W13	W19	W21	W27	W53	L50	L34	L32	L30
9	W9	W13	W17	W21	W25	W31	W43	L56	L40	L36
10	W9	W13	W17	W21	W23	W29	W37	W51	L54	L40
11	W9	W13	W17	W21	W25	W27	W33	W39	W45	L52
12	W9	W13	W17	W21	W25	W27	W31	W33	W37	W49

4. táblázat. A morabaraba különböző kezdőállásainak játékelméleti értéke és távolisága.

Ha viszont megvizsgáljuk ugyanezeknek az állásoknak a távoliságát is, akkor egy finomabb képet kaphatunk: majdnem minden fehér nyerés a főátló alatt gyorsabb nyerés, mint a megfelelő fekete nyerés (azaz amelyikben fordítottak a korongszámok), vagyis valamennyi előnye mégiscsak van a soron következő játékosnak. (Ez az előny jelentősebb a nem standard variánsoknál.)

A főátlókban azoknak az állásoknak a játékelméleti értéke látható, amelyekben a játékosok azonos számú korongot rakhatnak föl. Laskerben ezek a 3-3 kivételével mind döntetlenek. A morabarabában viszont a 6-6 korongszám fölöttiek értéke nyerés (ahogy az eredeti kezdőállás is). Azonban a standard variánsban 11 vagy 12 korongnál a fehér veszít!

Egy lehetséges oka ennek a meglepő jelenségnek a következő. A fölrakási fázisban bárhová tehetünk korongot, míg később már sokkal kevesebb helyre juttathatunk korongot egyetlen lépéssel. Ez azt jelenti, hogy miután a fekete fölteszi az utolsó korongját, a fehér (általánosságban) gyengébb lépésekkel reagálhat a fekete által kialakított szituációra. Vegyük észre azt is, hogy már nagyon kevés hely marad a táblán, miután 22 vagy 24 korong fölkerült (különösen, ha valamelyik játékos védekezően játszott, azaz megakadályozta ellenfele malomkészítéseit). Emiatt ilyenkor nagy esély van arra, hogy beszorítással véget ér a játszma. Ez a két megfigyelés kiegészíti egymást: éppen akkor van jó esély a játszma befejeződésére, amikor a fekete van előnyben.

Megvizsgálhatjuk annak a néha használt szabálynak a jelentőségét is, hogy teli táblakor a játszma döntetlennel ér véget (ahelyett, hogy a fekete nyerne). Az egyetlen lehetséges változás, amelyet ennek a szabálynak a behozása okozhat a kezdőpozíciók játékelméleti értékében, az az, hogy a táblázat jobb alsó cellája vesztésről döntetlenre változik. Azt találtuk, hogy ez egyedül a standard malomnál következik be.

Figyeljük meg, hogy van néhány hely a táblázatokban, ahol eggyel több felrakható fehér korong rosszabb értéket eredményez. Ezek a 3,3 és a 11,12 a standard variánsnál, a 3,3 a laskernél, és a 10,7 a morabarabánál. Hasonló a helyzet morabarabában 7,11-nél, ahol eggyel több fekete korong távolabbi veszteséget okoz fehér számára.

Szektor	Rang	%	Szektor	Rang	%	Szektor	Rang	%
N/A	0	64,21%	N/A	0	55,66%	3,4,0,0	-11	37,86%
6,3,0,0	112	15,6%	6,3,0,0	231	13,78%	4,3,0,0	11	27,97%
3,6,0,0	-112	12,34%	3,6,0,0	-231	12,67%	N/A	0	11,51%
3,5,0,0	-60	1,3%	4,3,0,1	-88	2,96%	5,3,0,0	54	8,38%
5,3,0,0	60	1,19%	3,4,1,0	88	2,59%	3,5,0,0	-54	7,69%
5,4,0,0	29	0,78%	5,3,0,0	184	1,23%	5,4,0,0	28	1,01%
9,8,0,0	59	0,72%	3,5,0,0	-184	1,16%	4,5,0,0	-28	0,96%
8,7,0,0	113	0,65%	9,8,0,0	354	0,96%	11,10,0,0	86	0,71%
7,8,0,0	-113	0,54%	8,7,0,0	361	0,71%	10,11,0,0	-86	0,65%
8,9,0,0	-59	0,53%	8,9,0,0	-354	0,70%	10,9,0,0	99	0,40%
Más		2,13%	Más		7,58%	Más		2,87%
(a)			(b)			(c)		

5. táblázat. Az első kulcsok eloszlása a különböző döntetlenek között a standard malomnál, a laskernél, illetve a morabarabánál.

4.3. Ultra erős megoldások

Korábban említettük, hogy az ultra erős megoldások adatbázisaiban a döntetlen értékű állásoknál azt is tároljuk, hogy optimális játék esetén melyik szektorban fog véget érni a játszma. Az 5. táblázat ezeknek az értékeknek a gyakoriságait mutatja. Sajnos a standard malom döntetlen állásainak 64,2%-a 0 értékű. Pl. ezek között van a kezdőállása is. Ezt azt jelenti, hogy még az ultra erős megoldásunkkal is sok játszma ér véget döntetlennel (egy 0 értékű szektorban). Azonban ezeknek a száma lényegesen kevesebb, mint a döntetlenek megkülönböztetése nélkül. Az ultra erős megoldás hatásosságának méréséhez egy nem tökéletesen játszó, alfa-béta vágást használó program ellen játszottunk 40 játszmát. Amikor a tökéletes program csak az erős megoldás adatbázisát használta, csak 2 játszmát tudott megnyerni (a többi természetesen döntetlen lett), viszont az ultra erős adatbázist használva 13-at.

Kísérleteztünk a 3.1. fejezetben leírt lokális heurisztikákkal is, de ezek nem okoztak jelentős változást az eredményekben.

A standard variánst játszó programunkat teszteltük egy olyan weboldalon is, ahol általában emberi játékosok játszanak egymás ellen. Az ultra erős program 19 játszmát játszott a weboldal legerősebb játékosával, és ebből 6-ot megnyert, továbbá játszott 14 játszmát egy másik elég erős (a weboldal pontozási rendszerében „mester” szintű) játékos ellen, amelyekből 11-et nyert meg.

Egy érdekesség, hogy a lasker kezdőállásának az értéke viszont nem 0, hanem a 9,8,1,1 szektor érhető el, amelynek a rangja 399.

4.4. A számítások ellenőrzése

A kiszámított adatbázisok helyességének ellenőrzéséhez Gasser módszerét alkalmaztuk [5]: egy ellenőrző program végigment az adatbázisok összes állásán, és leellenőrizte, hogy az egyes

állások értéke konzisztens-e a rákövetkezők értékével (a távolságot is figyelembe véve).

5. Továbbfejlesztési lehetőségek

5.1. Külön állásértékelések a tökéletes illetve a tökéletlen játékos számára

Amikor az erős megoldás a célunk, akkor nem kell megkülönböztetni a játékosokat, hiszen mindkettőről azt tesszük fel, hogy tökéletesen játszik. Azonban az ultra erős megoldásunk feltételezése szerint az egyik játékos tökéletes, míg a másik tökéletlen. Vegyük észre, hogy eddig nem jelöltük meg az állásoknál a soron következő játékos típusát. Ezt megjelölve – az állapottér megduplázása árán – még kifinomultabb heurisztikát használhatunk. A szektoroknak megjelenik egy ötödik paramétere, amely megadja, hogy milyen típusú játékos következik lépni. Így minden nem tranzien্স munkaegységnek pontosan két szektora lesz: egy *tökéletes következik* és egy *tökéletlen következik*. Mivel ezen szektorpárok értékeinek mindig egymás ellentettjeinek kell lenniük, ezért innentől kényelmesebb arról beszélni, hogy munkaegységekhez rendelünk értékeket. Ezek az értékek a *tökéletes következik* szektor értékével fognak megegyezni.

Így már használhatunk olyan heurisztikát, amelyik nagy értéket rendel olyan munkaegységekhez, amelyekben a legtöbb *tökéletes következik* pozíció nyerés (és így a tökéletlen játékos nagyobb eséllyel lép bele egy nyerésbe, döntetlen helyett), viszont figyelmen kívül hagyja a *tökéletlen következik* állásokat (hiszen a program úgysem vét hibát, amikor kiválasztja, hogy melyik *tökéletlen következik* állásba lépjen át). Korábban semleges (azaz 0) közeli értéket rendeltünk azokhoz a munkaegységekhez, amelyekben sok a döntetlen, viszont most már rendelhetünk (a többi tökéletes szektorhoz képest) alacsony ill. (a többi tökéletlen szektorhoz képest) magas értéket az ilyen munkaegységek tökéletes ill. tökéletlen szektoraihoz. Döntetlen kezdőállású játékoknál ez azért előnyös, mert egy sok döntetlent tartalmazó munkaegység rossz a tökéletes játékosnak, mivel nehéz az ellenfelet hibázásra kényszeríteni, viszont jó a tökéletlen játékosnak, mivel könnyebben érhet el döntetlent (nyerni meg úgysem nyerhet).

Vegyük észre, hogy ez megoldja azt a problémát is, hogy a nem tranzien্স EKS szektorokhoz a 0 értéket kellett rendelni, és így sok állás értéke lett 0.

5.2. Általánosítás más játékokra

A leírt döntetlenmegkülönböztető algoritmus használható minden olyan játéknál, amelynek megvannak azok a tulajdonságai, amelyeket a bevezetésben írtunk a malomról. Viszont ahhoz, hogy eredményes legyen, léteznie kell egy kellően finom partícionálásának az állapottérnek, amelynél a partíciókhoz értelmes heurisztikus értékek rendelhetők, és a körök nem lépnek át partíciók között. Általános esetben az előző alfejezetben leírt játékos-típus-megkülönböztetésre is szükség van⁶. A partíciók megfelelnek a munkaegységeknek, és azok a részhalmazok, ame-

⁶Így az is garantált, hogy minden mellékszektorbeli állást csak egy főszektorba kell korrigálni.

lyekben egy adott játékos következík lépni, megfelelnek a szektoroknak.

A témában elfogadás alatt lévő publikációink

G. E. Gévay, G. Danner, „Calculating Ultra-Strong and Extended Solutions for Nine Men’s Morris, Morabaraba, and Lasker”, *Journal of Artificial Intelligence Research*

Hivatkozások

- [1] L. V. Allis, M. van der Meulen, and H. J. Van Den Herik, „Proof-number search,” *Artificial Intelligence*, vol. 66, no. 1, pp. 91–124, 1994.
- [2] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, „Checkers is solved,” *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.
- [3] J. Schaeffer and R. Lake, „Solving the game of Checkers,” *Games of No Chance*, vol. 29, pp. 119–133, 1996.
- [4] R. Gasser, „Solving Nine Men’s Morris,” *Computational Intelligence*, vol. 12, no. 1, pp. 24–41, 1996. [Online]. Available: <http://dx.doi.org/10.1111/j.1467-8640.1996.tb00251.x>
- [5] R. U. Gasser, „Harnessing computational resources for efficient exhaustive search,” Ph.D. dissertation, ETH Zürich, Switzerland, 1995.
- [6] P. Stahlhacke, „The game of Lasker Morris,” FSU Jena, Tech. Rep., 2003. [Online]. Available: <http://www.althofer.de/stahlhacke-lasker-morris-2003.pdf>
- [7] T. R. Lincke, „Exploring the computational limits of large exhaustive search problems,” Ph.D. dissertation, ETH Zürich, Switzerland, 2002.
- [8] J. Romein and H. Bal, „Solving the game of Awari using parallel retrograde analysis,” *IEEE computer*, vol. 36, no. 10, pp. 26–33, 2003.
- [9] P. J. Jansen, „Using knowledge about the opponent in game-tree search.” DTIC Document, Tech. Rep., 1992.
- [10] T. Lincke, „Perfect play using Nine Men’s Morris as an example,” diploma thesis, ETH Zürich, 1994.
- [11] W. Slany, „Graph ramsey games,” *arXiv preprint cs/9911004*, 1999.
- [12] E. Zermelo, „Über eine anwendung der mengenlehre auf die theorie des schachspiels,” in *Proceedings of the fifth international congress of mathematicians*, vol. 2. II, Cambridge UP, Cambridge, 1913, pp. 501–504.

- [13] R. Bellman, „On the application of dynamic programming to the determination of optimal play in chess and checkers,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 53, no. 2, p. 244, 1965.
- [14] T. Ströhlein, „Untersuchungen über kombinatorische spiele,” Ph.D. dissertation, Technische Hochschule München, 1970.
- [15] K. Thompson, „Retrograde analysis of certain endgames,” *ICCA Journal*, vol. 9, no. 3, pp. 131–139, 1986.
- [16] J. W. H. M. Uiterwijk and H. J. van den Herik, „The advantage of the initiative,” *Information Sciences*, vol. 122, no. 1, pp. 43–58, 2000.