



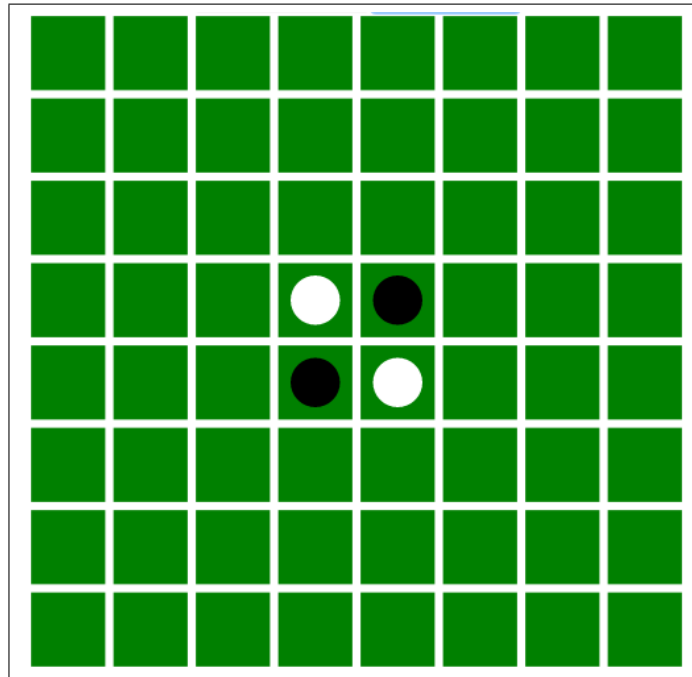
Universiteit Utrecht

Oplossingsalgoritmes voor Othello

BACHELORSRIPTIE

Rixt de Vries

Wiskunde



Begeleider:

Rob Bisseling

18 januari 2018

Inhoudsopgave

1	Inleiding	2
1.1	Geschiedenis	2
1.2	Wat is er al gedaan	2
2	Voorkennis	3
2.1	Spelregels	3
2.2	Minimax	3
2.3	Branch and bound en alfa-beta pruning	4
2.4	Negamax	5
3	Kleinere versies oplossen	6
3.1	Het 4×4 bord	6
3.2	Het 6×6 bord	8
4	Resultaten	9
5	Conclusie en Vervolgonderzoek	11
A	Code van 6×6 Algoritme met pruning	11

1 Inleiding

1.1 Geschiedenis

De geschiedenis van dit spel is niet helemaal zeker, want er zijn twee Engelsmannen die allebei beweren het spel in 1883 te hebben bedacht. John W. Mollett en Lewis Waterman noemen beiden de ander een fraudeur. Wat we wel zeker weten is dat het spel oorspronkelijk Reversi heette. Er zijn ook aanwijzingen dat het spel nog ouder is dan dat, want er zijn 18e eeuwse verwijzingen gevonden, dus misschien hebben beide mannen wel ongelijk.[1]

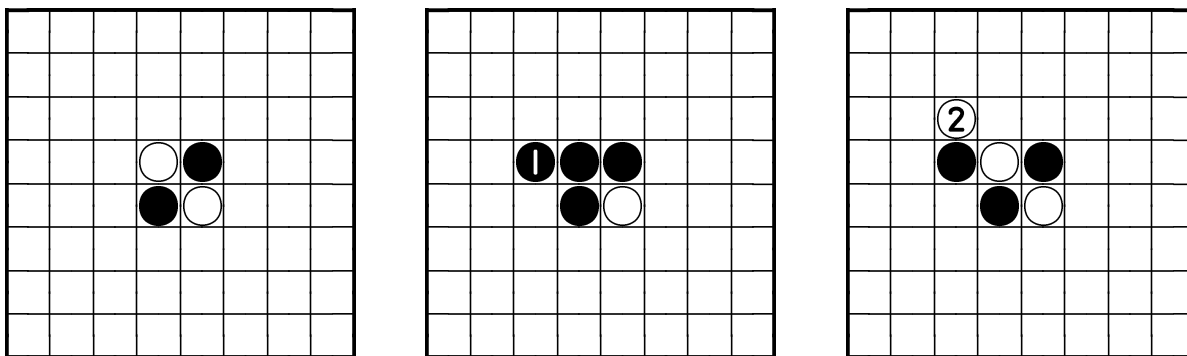
In 1893 begon het Duitse Ravensburger met het uitgeven van Reversi. Het was een van de eerste spellen die het uitgaf, slechts tien jaar na de oprichting van Ravensburger. In 1971 is het spel opnieuw uitgegeven door de Japanse Goro Hasegawa onder de naam Othello [2]. Er zijn twee kleine dingen veranderd. De opening van het spel is altijd hetzelfde met de eerste vier stenen in een diagonaal patroon in het midden, zoals te zien is in figuur 1. Ten tweede zijn er voor beide spelers twee keer zoveel stenen beschikbaar.

Sinds 1977 wordt elk jaar het WK reversi gehouden. In het eerste jaar deden er nog maar vijf mensen uit vijf landen mee, tegenwoordig zijn er elk jaar zo rond de twintig à dertig landenteams[3]. Ook zijn er tijdschriften en nieuwsbrieven speciaal over Othello verschenen, zoals de nieuwsbrief van de Nederlandse Othello Vereniging en van de British Othello Federation en het tijdschrift ‘The Othello Quarterly’ dat van 1979 tot 2005 in de Verenigde staten werd uitgegeven door ‘the United States Othello Association’.

1.2 Wat is er al gedaan

Het spel Othello is nog niet wiskundig opgelost. Dat betekent dat nog niemand weet wat de eindscore wordt als beide spelers perfect zouden spelen, en dat we al helemaal niet weten welke zetten ze zouden moeten doen om perfect te spelen. Dit komt omdat het heel veel rekenwerk kost om alle mogelijke zetten door te rekenen, zodat het zelfs met een supercomputer heel lang duurt.

Kleinere versies van het spel zijn wel al opgelost. Het bord wordt verkleind naar bijvoorbeeld vier bij vier vakjes. Dit is slechts vier keer zo klein, maar gelijk veel minder rekenwerk. Op meerdere plekken op internet vond ik dat mensen hiervoor algoritmes hadden geschreven waarmee hun computer onder een seconde kon berekenen dat wit zou winnen met acht stenen meer dan zwart. Ook het 6×6 bord is al doorberekend. Met als uitkomst dat wit wint met vier stenen meer dan zwart. Joel Feinstein, meervoudig Brits kampioen Othello, deed dit in 1993 met de 6×6 versie van zijn programma genaamd MODOT. Het had ongeveer twee weken nodig om ongeveer 40 miljard posities te bekijken en te berekenen welk pad optimaal was. Tegenwoordig zijn er meerdere programma's die de 6×6 versie van Othello in minder dan 100 uur kunnen oplossen. Dat is nog steeds drastisch meer dan de ene seconde die het 4×4 programma nodig had, terwijl er maar 20 vakjes bijgekomen zijn. Dus 20 vakjes extra betekent bijna 360.000 keer zoveel rekentijd. 8×8 Othello is nog niet opgelost, maar er wordt verwacht dat de uitkomst gelijkspel is. Van het 6×6 naar het 8×8 bord komen er nog eens 28 vakjes bij. Stel dat het nog eens 360.000 keer zo lang wordt, wat waarschijnlijk een te lage uitkomst geeft, dan zit je al op 36.000.000 uur, dat is ruim 4100 jaar. Het is door dit rekensommetje opeens heel duidelijk waarom Othello nog niet is opgelost.



(a) startpositie

(b) mogelijke eerste zet

(c) Wat wit kan doen

Figuur 1: Begin van het spel

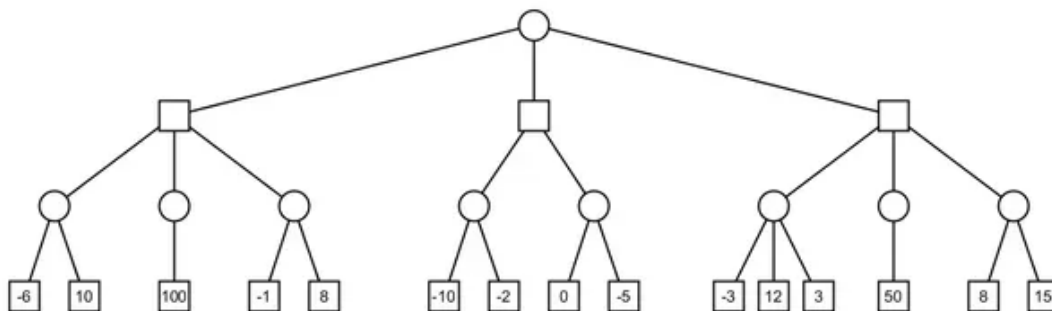
2 Voorkennis

2.1 Spelregels

Othello is een bordspel voor twee personen. Het wordt gespeeld op een bord van acht bij acht vakjes en met stenen die aan de ene kant zwart en de andere kant wit zijn. De ene speler speelt zwart en de ander wit. De beginopstelling van het bord heeft in de middelste vier vakjes twee zwarte stenen en twee witte stenen kruislings (zie figuur 1(a)). Zwart begint altijd met spelen. Je wint als aan het eind de meeste stenen jouw kleur hebben, waarmee wordt bedoeld dat de kleur waarmee je speelt naar boven gedraaid is. Als je aan de beurt bent, kun je op een leeg vakje grenzend aan (mag ook diagonaal) de stenen die er al liggen een steen plaatsen van jouw kleur zodat een rij stenen van de andere kleur ingesloten wordt door twee stenen van jouw kleur, waarvan er een al eerder op het bord geplaatst was. Vervolgens mag je alle stenen van de andere kleur ertussenin omdraaien zodat jouw kleur boven komt te liggen en dan is de ander aan de beurt. Dit insluiten kan zowel horizontaal als verticaal als diagonaal en zelfs op meerdere manieren tegelijk. Zie voor een voorbeeld van de opening van het spel figuur 1. De vakjes op het bord worden net als bij schaken aangeduid met een letter en een cijfer. De kolommen worden van links naar rechts A t/m H genoemd en de rijen van boven naar beneden 1 t/m 8. Bij kleinere versies nummer je op dezelfde manier, maar minder ver.

2.2 Minimax

Een spelboom is een manier om het verloop en alle opties van een spel weer te geven door middel van een graaf met knopen (speltoestanden) en kanten (zetten die spelers doen). In de wortel van de boom, op diepte 0, staat de startpositie van het bord. De kanten daaronder geven de mogelijke zetten van speler 1 aan en komen uit in een van de knopen met een nieuwe toestand van het bord. Deze knopen bevinden zich op diepte 1. De kanten hieronder geven de mogelijke zetten van speler 2 aan, enzovoorts. Onderaan de boom bevinden zich alle mogelijke eindstanden van het spel, waaruit bijbehorende eindscores af te lezen zijn. Zie voor een voorbeeld van een spelboom figuur 2, de cirkels geven de toestanden aan als speler 1 aan de beurt is, de vierkanten als speler 2 aan de beurt is.



Figuur 2: Spelboom waarin speler 1 twee beurten krijgt en 2 maar één.

Een spelboom wordt o.a. gebruikt voor spellen waarbij van tevoren al alle informatie bekend is en er geen kans bij betrokken is. Othello is zo'n spel en in theorie zou je dus ook elke toestand van het bord in een spelboom kunnen zetten waarbij twee toestanden met elkaar verbonden zijn door het leggen van een steen. In de praktijk zou dit heel veel tijd kosten, omdat er ongeveer 10^{28} mogelijke toestanden zijn. Het aantal toestanden van de 6×6 versie wordt geschat op $36 \cdot 10^{11}$. De zoekboom van het 4×4 bord heeft 237.781 knopen, wat een bovengrens aangeeft voor het aantal mogelijke toestanden, omdat toestanden vaker in de boom kunnen voorkomen. Gelukkig kunnen we gebruik maken van een computer, die alle 237.781 toestanden in iets meer dan een seconde kan doorwerken.

Er zijn verschillende methodes om deze spelboom door te werken en vervolgens op een eindantwoord uit te komen. Othello is een zero-sum game, dat is een spel waarbij de winst van de ene speler gelijk is aan het verlies van de andere speler. Voor zero-sum games is Minimax een handig algoritme voor het doorzoeken van een spelboom. Minimax is het minimaliseren van de maximale score van je tegenstander. De precieze definitie is $\bar{w}_i = \min_{a_{-i}} \max_{a_i} w_i(a_i, a_{-i})$, waarbij i het nummer van de speler is (bij Othello $i = 1$ of 2), $-i$ de tegenstander is, a_i de actie die speler i doet en w_i de waarde van speler i [4]. Speler 1 probeert zijn score te maximaliseren, we noemen speler 1 ook wel de maximizer. Speler 2, de minimizer probeert de score van speler 1 te minimaliseren. Door vanaf het einde van het spel, waar de eindscores bekend zijn, terug naar het begin te rekenen, kun je aan elke knoop in de boom een waarde toekennen. De waarde die een knoop krijgt, hangt af van de speler die aan de beurt is en de waardes van de knopen die een level dieper liggen en vanuit de huidige knoop te bereiken zijn. De speler die aan de beurt is kan namelijk kiezen tussen en van de waardes eronder en kiest de voor hem/haar meest gunstige uitkomst. De waarde in een knoop geeft aan wat de eindstand wordt voor speler 1 als beide spelers vanaf die toestand optimaal spelen. Na het toepassen van minimax staat in de wortel van de boom de uitkomst van het hele spel.[5]

2.3 Branch and bound en alfa-beta pruning

Branch and bound is een methode voor het doorzoeken van een spelboom. Het wordt in combinatie met Depth-first-search gebruikt. Depth-first-search slaat op de volgorde van het doorzoeken van een spelboom, je gaat eerst de diepte in, dus je doorzoekt één vertakking tegelijk. Bij het doorzoeken van een tak stuur je steeds de tot dan toe best gevonden uitkomst terug naar boven. Als je

in een tak komt waar de maximale score onder de tot dan toe beste score zit, hoeft je die tak niet meer te doorzoeken. Dat kan soms veel tijd schelen. Door dit in je code te stoppen wordt het dus sneller en heb je eerder een uitkomst.

Alfa-beta pruning is een variant van branch and bound en een zoekalgoritme waarmee je de minimax methode kunt uitbreiden. Het programma houdt de waarden van α en β bij. De definitie van α is de tot nu toe beste score voor de maximizer langs het pad naar de wortel en die van β is de tot nu toe beste score voor de minimizer langs het pad naar de wortel. In het begin liggen de waarden zo ver mogelijk uit elkaar, oftewel $\alpha = -\infty$ en $\beta = +\infty$. Aan het eind kun je α zien als de score die de maximizer in elk geval kan halen als deze optimaal speelt en β als de score die de maximizer maximaal kan halen, β wordt door de minimizer dus zo klein mogelijk gemaakt.

Als de maximizer aan de beurt is, krijgt de huidige knoop de waarde van α , en als de minimizer aan de beurt is krijgt het de waarde van β . In het begin blijven deze waarden $\pm\infty$, totdat een eindknoop bereikt wordt. Als de waarde van een knoop geupdate wordt, doordat een van de kinderen een eindscore doorgeeft, kijkt het programma of het nog nodig is om naar de andere kinderen te kijken. Wanneer in een bepaalde knoop α groter wordt dan β , betekent het dat de eindscores in de nog niet doorzochte takken eronder nooit beter kunnen uitpakken voor de speler die op dat moment aan de beurt is. Dat betekent dat hij nooit zal kiezen om de zet te maken die door die kant gerepresenteerd wordt, dus kunnen we dat hele stuk van de boom overslaan, dat noemen we een beta-cutoff. Stel dat een boom een diepte heeft van d en gemiddeld per knoop v vertakkingen heeft, dan kost het doorzoeken van die boom met alleen minimax $O(v^d)$ tijd. Dit is gelijk aan het ergste geval als je alfa-beta pruning gebruikt. Stel nu dat je optimaal snoeit en altijd als eerste de beste optie doorzoekt, dan hoeft je voor elke optie voor de eerste zet van zwart alleen maar de eerste (dus beste) optie voor de eerste zet van wit te doorzoeken, en voor die optie weer alle mogelijke zetten die zwart daarna kan doen, enz. Want de rest van de boom onder de alternatieve zetten van zwart kun je wegstrepen, omdat je een waarde hebt gevonden die nooit beter wordt dan de gevonden waarden onderaan de tak van de eerste optie die zwart kan doen, vanwege de optimale ordening van zetten. Dit geldt ook voor elke beurt die zwart daarna heeft. Dus heeft zwart nog steeds gemiddeld v opties en wit telkens maar 1, dus kan het doorzoeken van de boom in slechts $O(v \cdot 1 \cdot \dots \cdot v \cdot 1) = O(\sqrt{v^d})$ tijd. [7]

2.4 Negamax

Negamax is een variant van minimax speciaal voor zero-sum games. Als je in je code gebruik wilt maken van alfa-beta pruning is dat makkelijker te implementeren als je je code in negamaxvorm geschreven hebt. Je hoeft bij negamax alleen maar te maximaliseren en geen onderscheid meer te maken tussen de twee spelers. Je geeft aan de negamaxmethode mee in welke knoop op welke diepte je zit en wat de waarde van α en β zijn en wie er aan de beurt is, de maximizer heeft waarde 1 en de minimizer waarde -1. De beginwaarden zijn `Negamax(wortelKnoop, diepte, -∞, +∞, 1)`, dat betekent dat we vanaf de begintoestand van het bord kijken waar de maximizer aan de beurt is en dat $\alpha = -\infty$ en $\beta = +\infty$. Als de diepte nul is of je bent aan het einde van het spel, dan geeft de methode de waarde van deze knoop terug keer de kleur die aan de beurt is, zodat de score negatief wordt als de minimizer aan de beurt is, zodat ook die naar een zo hoog mogelijke (maar negatieve) score streeft. Als het spel nog niet afgelopen is, gaan we kijken naar de kinderen van de knoop (`kindKnopen`), oftewel de zetten die we vanuit de huidige knoop kunnen maken, die we in een lijst zetten en ordenen zodat de beste vooraan staan. Ook houden we de beste waarde

bij, die begint op $-\infty$. Daarna gaan we per kindknoop weer kijken wat daarvan de kinderen zijn door de methode zelf aan te roepen, maar dan een diepte minder, en de waarde van $-\beta$ wordt nu aan α doorgegeven en andersom, want de andere kleur is nu aan de beurt ($-\text{kleur}$). Daarna kijken we of we een nieuwe beste waarde gevonden hebben, door de beste waarde met de nieuwe waarde ‘w’ (die pas aan het eind van het spel wordt gegeven) te vergelijken die in de recursie gevonden is. Deze w vergelijken we ook met α , want α krijgt de hoogste waarde van de twee. Als α groter of gelijk is aan β , wat kan gebeuren, want ze komen steeds dichterbij elkaar, dan stopt de methode met de huidige tak doorkijken, want een van de spelers zal deze tak nooit kiezen, omdat er nooit een gunstiger score uit gaat komen voor deze persoon dan een al doorgekeken tak.[5]

Algorithm 1 6x6

```

1: procedure NEGAMAX(knoop, diepte,  $\alpha$ ,  $\beta$ , kleur)
2:   if diepte = 0 of knoop is eindKnoop then
3:     return kleur * heuristische waarde van knoop
4:   kindKnopen := MaakZetten(knoop)
5:   kindKnopen := OrdenZetten(kindKnopen)
6:   besteWaarde :=  $-\infty$ 
7:   for elk kind in kindKnopen do
8:     w := -Negamax(kind, diepte-1,  $-\beta$ ,  $-\alpha$ , -kleur)
9:     besteWaarde := max(besteWaarde, w)
10:   $\alpha$  := max( $\alpha$ , w)
11:  if  $\alpha \geq \beta$  then
12:    break
13:  return besteWaarde

```

3 Kleinere versies oplossen

Spellen zoals Othello kunnen wiskundig opgelost worden. Dat betekent dat met behulp van een computer alle toestanden, dus de hele spelboom doorberekend wordt om antwoord te geven op de vraag ‘Wie wint het spel?’ gegeven dat beide spelers perfect spelen, oftewel altijd de zet doen die voor hun de beste eindscore geeft vanaf het begin. Als je dit niet alleen voor de beginpositie doorrekent, maar vanaf elke toestand, dus ook als er eventueel al fouten gemaakt zijn, heb je het spel sterk bewezen. Omdat het spel begint met vier stenen in het midden, heb ik ervoor gekozen om geen kleinere borden met een oneven aantal vakjes op te lossen, maar alleen naar de even versies van 4×4 hokjes en 6×6 hokjes te kijken.

3.1 Het 4×4 bord

De 4×4 versie van Othello is op een moderne computer in minder dan één seconde op te lossen door programma’s die gebruik maken van de minimax methode. Deze programma’s bekijken in deze korte tijd de bijna 10 miljoen mogelijke posities en geven allemaal als eindresultaat dat wit wint met 3 tegen 11, dus met acht stenen meer dan zwart.

Bij een bord van 4 bij 4 zijn er 16 vakjes, waarvan de meeste in de eindpositie een zwart of witte steen hebben. Er zijn dus ongeveer $2^{16} = 65.536$ mogelijke eindposities van het spel. Dit is

met behulp van een computer te berekenen.

Zwart begint en kan vier dingen doen, maar vanwege symmetrie zijn deze opties aan elkaar gelijk, dus heeft hij eigenlijk maar één keus. Nu heeft wit nog maar één steen op het bord. Een nieuwe witte steen kan op drie verschillende plekken neergelegd worden.

Ik heb een recursief algoritme geschreven dat door middel van depth first search berekent welke kleur wint als beide spelers optimaal spelen en wat dan de eindscore wordt. Het kijkt bij elke beurt of er legale zetten zijn en kiest daarvan de eerste uit die nog niet is geanalyseerd en gaat dan helemaal diep de boom in, tot het spel eindigt en geeft dan een score terug en vergelijkt met de scores uit andere takken.

Het bord sla ik op in een 32 bits integer. Ik heb het bord genummerd met de getallen 0 t/m 15 beginnend linksboven en dan met de leesrichting mee. De kleinste 16 bits geven aan waar de zwarte stenen liggen en de andere 16 zijn voor de witte stenen. Als de waarde van de integer 0 is, dan is het bord helemaal leeg. De startpositie heeft zwarte stenen op hokjes 6 en 9 en een witte op 5 en 10, maar de witte stenen komen 16 plekken verder op plek 21 resp. 26, dus de startwaarde van het bord is $2^6 + 2^9 + 2^{21} + 2^{26} = 69.206.592$.

Er wordt een nieuw spel aangemaakt en daarvoor wordt de `RecursieveMethode` aangeroepen. Die geeft uiteindelijk de `winnendeKleur`, `winnendPad` en `winnendVerschil` (in score) terug. Het eerst wat deze methode doet is het aantal keer dat het is aangeroepen ophogen met één. Daarna als het spel geëindigd is, geeft het de kleur die wint terug (`WieWint`). Vervolgens maakt het een lijst ('`zetten`') van alle legale zetten (`LegaleZettenGenerator`). Het maakt ook alvast een `winnendPad` (lijst), een `winnendeKleur` en een `winnendVerschil` (int) aan. Als de lijst `zetten` leeg is, roept het `Pas` aan en daarna zichzelf, om voor de andere speler een goede zet te onderzoeken. Als het vervolgens weer uit die recursie komt, maakt het eerst die `Pas`-zet weer ongedaan en heb je een `winnendeKleur`, `Pad` en `Verschil`. Het voegt ook nog de zet '-1' toe aan `winnendPad`, om aan te geven dat er is gepast. Als de lijst `zetten` daarentegen niet leeg is, neemt het een random zet uit die lijst en voert die uit (`ZetDoen`), daarna roept het zichzelf aan om voor de andere speler mogelijke zetten te onderzoeken. Als het daar weer uitkomt, heeft het een lijst '`resultaat`' aangemaakt, waarin een hele opsomming van beurten staat die een heel spel met twee spelers simuleert. Vervolgens maakt het deze zet ongedaan, zodat er een nieuwe tak van de boom onderzocht kan worden. Ook houdt het ondertussen de beste score bij zodat het programma de optimale zet kan bepalen en die in het `winnendPad` kan stoppen. Aan het eind van de methode geef je de `winnendeKleur` met `winnendPad` en `winnendVerschil` terug. Hier kwam bij mij elke keer uit dat wit wint met acht stenen meer dan zwart, precies wat anderen volgens internet ook met hun algoritme hebben gevonden.

Het spel eindigt als er twaalf stenen zijn gelegd, want dan is het bord vol, of als er twee keer achter elkaar is gepast, want dan kunnen beide spelers niks meer.

De processor van mijn computer is een Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz. Daarmee doet mijn computer er 1059 milliseconden (220.000 k/s) over om alle knopen van het 4×4 bord door te rekenen.

Algorithm 2 4×4

```
1: procedure ZETDOEN(positie)
2:   ply++
3:   kloon[ply]= bord[ply-1].Clone()
4:   ZetSteen(pos, Kleur)
5:   FlipStenen(pos, Kleur)
6:   aanZet = (bord[ply].aanZet == Kleur.Wit ? Kleur.Zwart : Kleur.Wit)
7:   aantalStenen++
8:   heeftGepast = false
```

3.2 Het 6×6 bord

We gaan nu naar de iets grotere 6×6 versie van het spel kijken. Omdat de spelboom exponentieel groter wordt, is het bij deze versie belangrijk om een zo efficiënt mogelijk programma te schrijven en goed na te denken over manieren voor het snoeien van de boom.

Voor het 6×6 bord heb ik mijn eerste algoritme aan moeten passen. Het bord heeft nu 36 vakjes en kan dus niet meer door 32 bits gerepresenteerd worden, zelfs 64 zou niet genoeg zijn, want ik heb op z'n minst 72 bits nodig. Ik heb gekozen om het bord op te slaan in een array van twee 64 bit integers en niet één integer van 128 bits, omdat ik een 64 bits processor heb, dus werkt het eerste op mijn computer sneller. Ik heb in de code ook overal `(0x1 << pos)` moeten veranderen in `(0x1L << pos)`, want er zijn vakjes boven de 31. De `0x1` is hexadecimaal voor 1, `pos` geeft de positie aan waarop die 1 moet komen en `(0x1 << pos)` betekent dat die 1 `pos` aantal vakjes naar links gebitshift wordt, maar bij het 6×6 bord kan het voorkomen dat `pos` groter is dan 32, dus meer dan het aantal bits in een integer, daarom heb ik er een long (`0x1L`) van gemaakt, want die heeft 64 bits en dat is zelfs genoeg voor het 8×8 bord. Ook moest ik de `FlipStenen` methode zo aanpassen dat bij een nieuwe steen zowel horizontaal, verticaal als diagonaal beide kanten opgekeken kan worden.

Hoe meer zetten je doet, hoe groter de boom wordt. En niet zomaar lineair, maar exponentieel, want voor elke zet heb je meestal meerdere keuzes. Mijn algoritme doet er enkele seconden over als je wil dat hij minder dan tien zetten berekent, om de eerste tien zetten te berekenen heeft hij net onder de drie minuten nodig, maar vanaf twaalf stappen doet hij er opeens 2 uur en 32 minuten over. Daarom heb ik alfa-beta pruning in m'n algoritme verwerkt in combinatie met negamax. Het algoritme is duidelijk een stuk sneller geworden door slechts een paar extra regels code. Mijn algoritme kan nu in die enkele seconden tot 14 stappen diep uitrekenen. Dat is een groot verschil, maar ik ben er nog niet. Het kan nog sneller.

Ook heb ik de zetten van tevoren op verschillende manieren geordend voordat het algoritme ze ging doorspitten om te kijken welke de beste was. Als de zetten op de juiste manier geordend worden, komen de beste zetten vooraan te staan en worden er uiteindelijk grotere takken afgesneden, zodat het algoritme minder iteraties nodig heeft om alle relevante knopen door te zoeken, omdat er minder relevante knopen zijn. Dit heeft als nadeel dat het per iteratie meer tijd kost om de zettenlijst te ordenen, dus dit moet je ook weer niet overdrijven, want anders doet je code er alsnog langer over. Eerst heb ik een extra methode geschreven die aan alle vakjes een waarde geeft (heuristiek).

De vakjes in het midden zijn niet zulke grote overwinningen, maar als je een vakje aan de rand van het bord hebt, is de kans al een stuk kleiner dat die weer van je wordt afgepakt en als je een hoek hebt, dan raak je die nooit meer kwijt. De vakjes grenzend aan een hoek wil je daardoor juist liever niet pakken, want dan kan je tegenstander de beurt erna een hoek veroveren. Met deze strategie en wat voorbeelden van het internet heb ik de waardes gegeven zoals je in figuur 3 kunt zien.

Na deze manier van ordenen heb ik ook een tweede manier geprobeerd, namelijk op verschil van aantal stenen. Als een zet in de legale zettenlijst wordt geplaatst, wordt meegegeven hoeveel stenen de speler die aan de beurt is meer zou hebben na die zet dan de andere speler. Hoe groter dat getal, hoe meer naar voren die zet in de lijst geplaatst wordt. Daarna heb ik nog een derde manier van zetten ordenen geprobeerd. Hierbij wordt een zet die een beta-cutoff veroorzaakt opgeslagen, zodat als deze elders in de boom weer voorkomt, hij vooraan in de lijst wordt geplaatst, want we willen natuurlijk meer van de boom kunnen snoeien en bij deze zet zijn we er zeker van dat dat gebeurt.

```
static int[] StaticEval = {
    10, -2, 4, 4, -2, 10,
    -2, -2, 2, 2, -2, -2,
    4, 2, 0, 0, 2, 4,
    4, 2, 0, 0, 2, 4,
    -2, -2, 2, 2, -2, -2,
    10, -2, 4, 4, -2, 10,
};
```

Figuur 3: Meegegeven Heuristiek

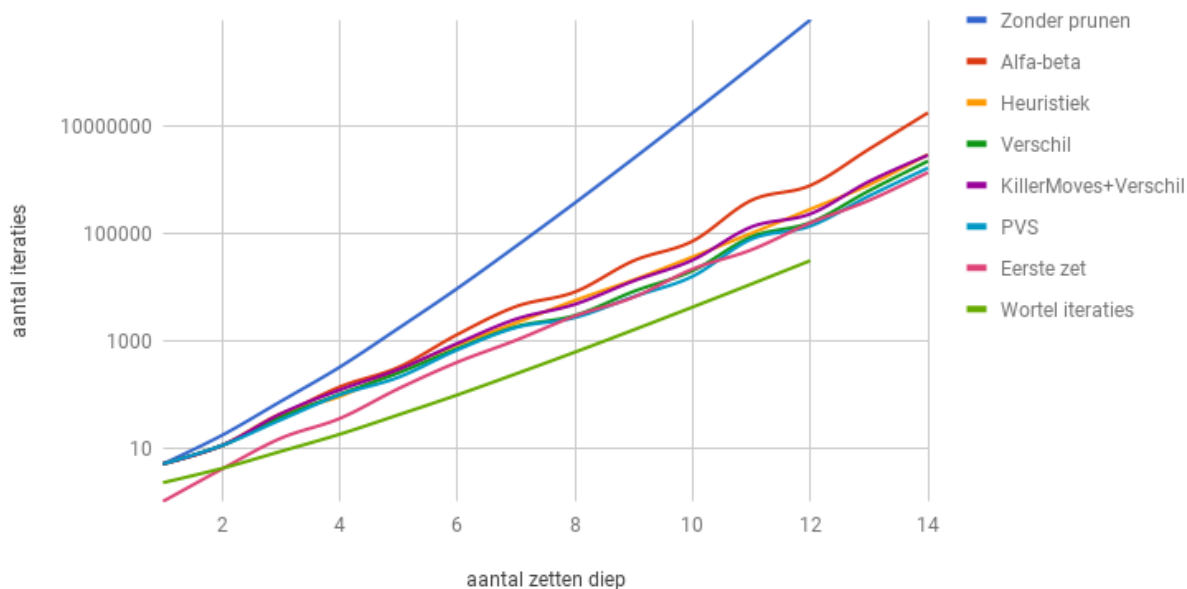
Nu we een goed geordende zettenlijst hebben, kunnen we Principal Variation Search (PVS) implementeren, een methode die minstens zoveel wegsnoeit uit de boom als alfa-beta. Het gaat er vanuit dat de beste zet altijd vooraan staat, dus daarom doen we dit pas nadat we een goede ordenmethode hebben gevonden.

Vanwege de symmetrie van het bord kunnen we de eerste zet van zwart al doen op vakje 8. Met deze ene extra regel in de code is het programma vier keer zo snel geworden, doordat de eerste zet nu maar één optie heeft, in plaats van vier.

4 Resultaten

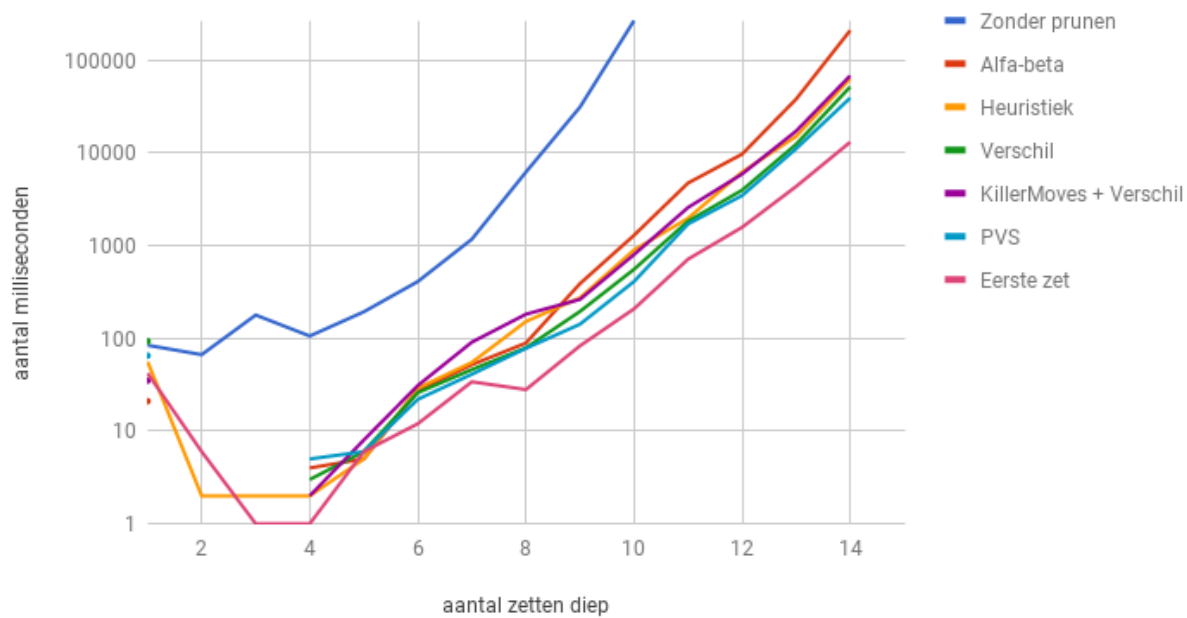
In paragraaf 3.2 heb ik beschreven met welke methodes ik allemaal heb geprobeerd mijn algoritme sneller te maken. Hier zien we wat er uit het experiment is gekomen. Ik heb het 6×6 algoritme niet het hele spel laten uitrekenen, maar een aantal stappen diep, van één stap diep tot en met 14. Daarbij heb ik het aantal iteraties en de berekentijd bijgehouden. Dit heb ik eerst gedaan zonder snoeitechnieken en daarna met steeds een snoeitechniek erbij. Eerst met alleen alfa-beta pruning, daarna ook met heuristiek van het bord, dat bleek maar een beetje te helpen, dus heb ik daarna zonder heuristiek en met verschil getest, dat bleek beter te werken. De combinatie van die twee werkte minder goed dan een van de twee. Daarna heb ik killermoves samen met verschil getest, dat ging ook niet veel beter, dus besloot ik van deze drie ordenmethodes alleen verschil aan te laten staan. Daarna heb ik PVS erbij aangezet en als laatste nog de eerste zet. Zie de grafieken in figuur 4 voor de winst qua iteraties en tijd. Na alleen al alfa-beta pruning aangezet te hebben, is mijn algoritme al gelijk een stuk sneller. Verder worden er nog kleine winsten behaald met het ordenen van zetten en PVS en het doen van de eerste zet scheelde qua tijd ook nog significant.

Aantal iteraties van de algoritmes



(a) Aantal iteraties van de algoritmes

Berekeningstijd van de algoritmes



(b) Berekeningstijd van de algoritmes

Figuur 4: Resultaten van de algoritmes

Ook zien we bij de Iteratiegrafiek dat de lijnen van de snoeitechnieken niet recht zijn maar binnen hun baan op en neer schommelen. Dat heeft te maken met oneven-even pariteit. Bij oneven beurten heeft zwart een zet meer mogen doen dan wit en worden er dus $O(v \cdot 1 \cdot \text{dots} \cdot v)$ iteraties doorgerekend, wat net iets meer is dan $O(\sqrt{v^d})$. We zien dus ook bij een oneven aantal zetten telkens een bobbeltje in de grafieken waarin gesnoeid wordt.

5 Conclusie en Vervolgonderzoek

Snoeien in de takken van een spelboom kan de rekentijd van een algoritme significant omlaag brengen als je de goede methodes toepast. Er zijn verschillende speltactieken waarvan je gebruik kunt maken om de uitkomst van Othello sneller te berekenen. Sommigen werken beter dan anderen en sommigen kun je combineren en anderen juist niet. Ook zien we dat wit bij kleinere versies van het spel in het voordeel is, want als beide spelers optimaal spelen, zal wit altijd winnen.

Wat ook nog heel nuttig kan zijn, is gebruik maken van de hoeveelheid dubbele borden die wel moeten voorkomen in de gehele spelboom. Op de Engelse wikipediapagina over alfa-beta pruning staat een link naar Transpositietabellen. Daarin kun je posities die een goede score opleveren met behulp van alfa-beta pruning opslaan zodat de spelboom vanaf die positie maar één keer doorgerekend hoeft te worden, want wat daarna gebeurt is bij elke van deze identieke toestanden hetzelfde.

Referenties

- [1] Wikipedia, “Reversi.” website. <https://en.wikipedia.org/wiki/Reversi>.
- [2] “Modern living. japanese othello,” *TIME*, no. 2, 11-1976.
- [3] W. O. Federation, “World othello championship.” website. <https://www.worldothello.org/about/world-othello-championship/woc/1977>.
- [4] E. Z. S. Maschler, Michael; Solan, *Game Theory*. Cambridge University Press, 2013.
- [5] H. Ahmadi, “An introduction to game tree algorithms.” website. <http://www.hamedahmadi.com/gametree/#negamax>.
- [6] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

A Code van 6×6 Algoritme met pruning

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Diagnostics;
```

```

namespace Prune6bij6
{
    using Kleur = Byte;
    class Hoofd
    {
        static void Main(string[] args)
        {
            for (int i = 5; i < 19; ++i)
            {
                Console.WriteLine("Diepte " + (i - 4));
                Bord spel = new Bord();
                spel.tot = i;
                spel.Print();

                Stopwatch sw = new Stopwatch();
                sw.Start();
                Tuple<List<int>, int> eindResultaat = spel.NegaMax(-1000,1000, i);
                sw.Stop();
                Console.WriteLine("Aantal iteraties: " + spel.recursieveAanroepen);
                Console.WriteLine("Tijd: " + sw.ElapsedMilliseconds + " ms");
                Console.WriteLine("knopen/s: " + (1000.0 / sw.ElapsedMilliseconds) *
                    spel.recursieveAanroepen);
                Console.WriteLine("BetaCuts: " + spel.betaCutsFirst/
                    (double)(spel.betaCuts == 0 ? 1 : spel.betaCuts));
            }
            Console.ReadKey();
        }
    }

    public class Bord
    {
        const Kleur Zwart = 0;
        const Kleur Wit = 1;
        const Kleur Geen = 2;

        static int[] StaticEval = {
            10, -2, 4, 4, -2, 10,
            -2, -2, 2, 2, -2, -2,
            4, 2, 0, 0, 2, 4,
            4, 2, 0, 0, 2, 4,
            -2, -2, 2, 2, -2, -2,
            10, -2, 4, 4, -2, 10,
        };
        static int[,] KillerMoves = new int[36, 2];
    }
}

```

```

public int tot = 1;
public int ply;
public Staat[] staten = new Staat[64];
public long recursieveAanroepen = 0;
public long betaCuts = 0;
public long betaCutsFirst = 0;
//We nemen een array van 2 getallen van 64 bits, het bord is 36 vakjes die
//drie staten kunnen hebben, leeg, zwart en wit.

public Bord()
{
    //Zetten-teller
    ply = 0;
    staten[0] = new Staat();
    //Startopstelling
    staten[ply].bord[0] = (0x1L << 15) | (0x1L << 20);
    staten[ply].bord[1] = (0x1L << 14) | (0x1L << 21);

    //Zwart begint.
    staten[ply].aanZet = Zwart;
    staten[ply].heeftGepast = false;
    staten[ply].heeftDubbelGepast = false;
    staten[ply].aantalStenen = 4;

    //Dit is de eerste zet van zwart die vanwege symmetrie al gezet kan worden.
    ZetDoen(8);
}

int Positie(int rij, int kolom)
{
    return rij * 6 + kolom;
}

//HeeftSteen wordt gebruikt om het bord weer te geven.
public bool HeeftSteen(int pos, Kleur kleur)
{
    //Kleur = 0 is zwart, 1 = wit
    return (staten[ply].bord[kleur] & (0x1L << pos)) != 0;
}

Kleur WieWint()
{
    if (Verschil(Zwart) > 0)
        return Zwart;
    else if (Verschil(Wit) > 0)

```

```

    return Wit;
else
    return Geen;
}

int Verschil(Kleur kleur)
{
    int witte = 0, zwarte = 0;
    for (int i = 0; i < 36; i++)
    {
        if (HeeftSteen(i, Wit))
        {
            witte++;
        }
        else if (HeeftSteen(i, Zwart))
        {
            zwarte++;
        }
    }
    if (kleur == Wit)
    {
        return witte - zwarte;
    }
    else
        return zwarte - witte;
}

public int Heuristiek(int pos)
{
    //Haal onderstaande regels uit de comments voor KillerMoves
    /*if (KillerMoves[ply, 0] == pos)
    {
        return -1000;
    }
    else if (KillerMoves[ply, 1] == pos)
    {
        return -900;
    }*/
    long[] kopie = (long[])staten[ply].bord.Clone();
    FlipStenen(pos, staten[ply].aanZet);
    int vers = Verschil(staten[ply].aanZet);
    staten[ply].bord = (long[])kopie.Clone();
    return /*-Bord.StaticEval[pos];*/ - vers;
    //Heuristiek van het bord wordt niet meegenomen, verschil wel.
}

```

```

public Tuple<List<int>, int> NegaMax(int alpha, int beta, int depth)
{
    //Deze methode geeft aan het einde van het spel een score terug.
    //Als deze positief is, wint zwart. Als negatief, dan wint wit.
    //Verder geeft de methode het winnende pad van zetten terug.
    ++recursieveAanroepen;

    //Bitwise & is sneller dan modulo
    if ((recursieveAanroepen & 8191) == 0)
    {
        Console.WriteLine(recursieveAanroepen);
    }

    //Als EindSpel=true, stop recursie
    if (EindSpel())
    {
        //Print();
        //Console.WriteLine(Vershil(WieWint()));
        return new Tuple<List<int>, int>(new List<int>(), Vershil(staten[ply].aanZet));
    }
    List<int> zetten = LegaleZettenGenerator();
    //for(int i=0; i<zetten.Count; i++)
    //    Console.WriteLine(zetten[i]);
    //Print();
    List<int> winnendPad = new List<int>();
    int winnendVershil = 0;
    //Orden zetten zoals in de methode Heuristiek staat beschreven:
    var zets = zetten.OrderBy(item => Heuristiek(item));

    //Als je geen legale zet kunt vinden, Pas.
    if (zetten.Count() == 0)
    {
        zetten = new List<int>() { -1 };
    }

    int besteScore = -1000;
    //Je kunt ook de zetten randomizeren, dat stond in het begin aan:
    /*var rnd = new Random();
    var zets = zetten.OrderBy(item => rnd.Next());*/

    int i = 0;
    foreach (int zet in zets)//Verander zetten weer in zets voor random of ordening
    {
        ZetDoen(zet);
    }
}

```



```

//De kleur die wint en het bijbehorende pad
Tuple<List<int>, int> resultaat;
//PVS
if (i == 0) //Maak van >= weer == en PVS staat aan.
{
    resultaat = NegaMax(-beta, -alpha, depth - 1);
}
else
{
    resultaat = NegaMax(-alpha-1, -alpha, depth - 1);
    if(alpha < -resultaat.Item2 && -resultaat.Item2 < beta)
    {
        resultaat = NegaMax(-beta, resultaat.Item2, depth - 1);
    }
}
//Zetten ongedaan maken
ZetOngedaanMaken();

int score = -resultaat.Item2;

if (score > besteScore)
{
    besteScore = score;
    winnendPad = resultaat.Item1;
    winnendPad.Add(zet);

    winnendVerschil = -resultaat.Item2;

    if(score > alpha)
    {
        if(score >= beta)
        {
            if(i == 0)
            {
                betaCutsFirst++;
            }
            betaCuts++;
            //Haal hieronder comments weg voor KillerMoves
            /*KillerMoves[ply, 1] = KillerMoves[ply, 0];
            KillerMoves[ply, 0] = zet;*/
            return new Tuple<List<int>, int>(winnendPad, beta);
        }
        alpha = score;
    }
}
}

```

```

        ++i;
    }
    if(bestescore == alpha)
    {
        winnendVerschil = besteScore;
    }
    else
    {
        winnendVerschil = alpha;
    }

    //Teruggeven wie er wint en met welke zetten
    Tuple<List<int>, int> tuple =
        new Tuple<List<int>, int>(winnendPad, winnendVerschil);

    return tuple;
}

public bool EindSpel() //Bepaalt wanneer het spel eindigt
{
    return staten[ply].aantalStenen >= tot || staten[ply].heeftDubbelGepast;
}

void Pas() //Methode voor passen
{
    ply++;
    staten[ply] = (Staat)staten[ply - 1].Clone();
    staten[ply].bord = (long[])staten[ply - 1].bord.Clone();

    staten[ply].aanZet = (staten[ply].aanZet == Wit ? Zwart : Wit);

    if (!staten[ply].heeftGepast)
        staten[ply].heeftGepast = true;
    else if (!staten[ply].heeftDubbelGepast)
        staten[ply].heeftDubbelGepast = true;
}

public List<int> LegaleZettenGenerator()
{
    List<int> zetten = new List<int>();
    for (int i = 0; i < 36; i++)
    {
        //Check of vakje i leeg is
        if (HeeftSteen(i, Zwart) || HeeftSteen(i, Wit))
            continue;
    }
}

```

```

        //Zoja maak kopie van bord en roep FlipStenen aan
        long[] kopie = (long[])staten[ply].bord.Clone();
        FlipStenen(i, staten[ply].aanZet);
        if (!kopie.SequenceEqual(staten[ply].bord))
        {
            zetten.Add(i);
        }
        staten[ply].bord = (long[])kopie.Clone();
    }
    return zetten;
    /*Maak kopie van bord en roep FlipStenen aan, als het bord
    veranderd is tov eerst dan is het een legale zet*/
}

public void ZetDoen(int pos)
{
    if (pos == -1)
    {
        Pas();
    }
    else
    {
        ply++;
        staten[ply] = (Staat)staten[ply - 1].Clone();
        staten[ply].bord = (long[])staten[ply - 1].bord.Clone();
        //Zet een steen neer en draait stenen om, zetcount +1
        ZetSteen(pos, staten[ply].aanZet);
        FlipStenen(pos, staten[ply].aanZet);
        //Daarna is de ander aan de beurt
        staten[ply].aanZet = (staten[ply].aanZet == Wit ? Zwart : Wit);

        staten[ply].aantalStenen++;

        staten[ply].heeftGepast = false;
    }
}

public void ZetOngedaanMaken()
{
    ply--;
}

public void FlipStenen(int pos, Kleur kleur)
{
    int kolom = pos % 6;

```

```

int rij = pos / 6;
Kleur andereKleur = (kleur == Wit ? Zwart : Wit);
//Zoeken tot je een lege plek of eigen steen tegenkomt, als lege plek: stop,
//Als eigen steen: ga stapjes terug voor andere stenen flippen
//Rijen checken
if (kolom < 4)
{
    int tot = 0;
    for (int i = 1; i < (6 - kolom); i++)
    {
        if (HeeftSteen(pos + i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos + i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos + t, kleur);
    }
}
if (kolom > 1)
{
    int tot = 0;
    for (int i = 1; i < (kolom + 1); i++)
    {
        if (HeeftSteen(pos - i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos - i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos - t, kleur);
    }
}
}

```

```

//Kolommen checken
if (rij < 4)
{
    int tot = 0;
    for (int i = 1; i < (6 - rij); i++)
    {
        if (HeeftSteen(pos + 6 * i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos + 6 * i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos + 6 * t, kleur);
    }
}
if (rij > 1)
{
    int tot = 0;
    for (int i = 1; i < (rij + 1); i++)
    {
        if (HeeftSteen(pos - 6 * i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos - 6 * i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos - 6 * t, kleur);
    }
}
//Diagonaal naar beneden (backslash)
//Kijk naar de hokjes in het 4x4 blok linksboven
if (rij < 4 && kolom < 4)
{

```

```

int tot = 0;
for (int i = 1; i < (6 - Math.Max(rij, kolom)); i++)
{
    if (HeeftSteen(pos + 7 * i, kleur))
    {
        tot = i;
        break;
    }
    else if (!HeeftSteen(pos + 7 * i, andereKleur))
    {
        break;
    }
}
for (int t = 1; t < tot; t++)
{
    FlipSteen(pos + 7 * t, kleur);
}
}
//Kijk naar de hokjes in het 4x4 blok rechtsonder
else if (rij > 1 && kolom > 1)
{
    int tot = 0;
    for (int i = 1; i < (Math.Min(rij, kolom) + 1); i++)
    {
        if (HeeftSteen(pos - 7 * i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos - 7 * i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos - 7 * t, kleur);
    }
}
}
//Diagonaal omhoog (slash)
//Kijk naar de hokjes in het 4x4 blok rechtsboven
if (rij < 4 && kolom > 1)
{
    int tot = 0;
    for (int i = 1; i < (7 - Math.Max(rij + 1, 6 - kolom)); i++)

```

```

    {
        if (HeeftSteen(pos + 5 * i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos + 5 * i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos + 5 * t, kleur);
    }
}
//Kijk naar de hokjes in het 4x4 blok linksonder
else if (rij > 1 && kolom < 4)
{
    int tot = 0;
    for (int i = 1; i < (7 - Math.Max(6 - rij, kolom + 1)); i++)
    {
        if (HeeftSteen(pos - 5 * i, kleur))
        {
            tot = i;
            break;
        }
        else if (!HeeftSteen(pos - 5 * i, andereKleur))
        {
            break;
        }
    }
    for (int t = 1; t < tot; t++)
    {
        FlipSteen(pos - 5 * t, kleur);
    }
}
}

//FlipSteen flipt de andere kleur naar de aangegeven kleur
void FlipSteen(int pos, Kleur kleur)
{
    if (kleur == Wit) //Eerst de zwarte steen op pos weghalen
        staten[ply].bord[Zwart] ^= (0x1L << pos); //XOR met elkaar
    else if (kleur == Zwart) //Eerst de witte steen op pos weghalen

```

```

        staten[ply].bord[Wit] ^= (0x1L << pos);
    ZetSteen(pos, kleur);//Dan de juiste kleur toevoegen op pos
}

//ZetSteen zet de steen in Kleur kleur op positie pos
public void ZetSteen(int pos, Kleur kleur)
{
    staten[ply].bord[kleur] |= (0x1L << pos);
}

//Methode om bord te printen en te kijken of een positie zwart of wit is
public void Print(/*int stand*/)
{
    //2 loopjes zwarte steen print 0, witte steen print W.
    for (int i = 0; i < 36; i++)
    {
        //Zwarte stenen zijn false en 0
        if (this.HeeftSteen(i, Zwart))
            Console.Write('0');
        //Witte stenen zijn true en W
        else if (this.HeeftSteen(i, Wit))
            Console.Write('W');
        //Lege vakjes zijn puntjes
        else
            Console.Write('.');
        //Na zes vakjes begint een nieuwe regel
        if (((i + 1) % 6) == 0)
        {
            Console.Write('\n');
        }
    }
    //Console.WriteLine(staten[ply].aanZet + " is aan de beurt. \n");
}
}

public class Staat : ICloneable
{
    public long[] bord = new long[2]; van int naar int64-array
    public Kleur aanZet;
    public bool heeftGepast;
    public bool heeftDubbelGepast;
    public int aantalStenen;
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}

```


}
}
}