Project Logisch Programmeren - Schaakcomputer

Arno Vermote - 01806792
Logisch Programmeren [C003783A] - Universiteit Gent

10 juni 2021

Samenvatting

Het project van het vak Logisch Programmeren bestaat uit het maken van een schaakcomputer met de programmeertaal Prolog. Er wordt vereist dat deze schaakcomputer een minimax-boom opstelt om de beste zet te vinden. We zijn erin geslaagd om dit te implementeren met zoekdiepte 3 en een gemiddelde uitvoeringstijd van 13.69 seconden.

1 Introductie code

Eerst leggen we de inhoud van de Prologbestanden uit om de structuur van de code duidelijker te maken.

- main.pl: Dit bevat het main-predicaat dat het programma uitvoert.
- parser.pl: Met deze module wordt de input verwerkt door deze om te zetten naar de interne bordvoorstelling. (zie deelparagraaf 2.1)
- mover.pl: Deze module zoekt en controleert iedere mogelijke zet op het gegeven bord.
- score.pl: Hierin wordt voor alle mogelijke zetten de beste zet bepaald aan de hand van een spelboom met scorefunctie.
- writer.pl: Na het vinden van de beste zet moet het nieuwe bord weggeschreven worden. Deze module zorgt daarvoor.

2 Algoritmen en Datastructuren

In dit onderdeel overlopen we hoe het programma een nieuwe zet kan genereren en welke algoritmen en datastructuren daarbij gebruikt worden.

2.1 Interne bordvoorstelling

2.1.1 Bord

In de module `parser:171` wordt de inputstring die het spelbord voorstelt geparset. Dit gebeurt lijn per lijn, teken per teken. De input wordt dan omgezet naar een 2-dimensionale lijst van schaakstukken (`parser:78:parse_board/2`). De benodigde indices om een schaakstuk uit die lijst te halen komen overeen met de volgende voorstelling van een x- en y-as getekend over het schaakbord:

```
0          7
  ---------> x-as (kolommen)
  |
  |
  |
7 v
y-as (rijen)
```

Ieder schaakstuk wordt vervolgens voorgesteld door een lijst met 2 elementen: kleur en type. Zo is `[color(white),type(queen)]` een geldig schaakstuk. Lege plaatsen op het bord worden voorgesteld door het element `[color(none),type(empty)]` op de juiste plaats in het schaakbord te stoppen.

Deze voorstelling maakt het eenvoudig om schaakstukken te verzetten: er moet gewoon een element in de 2-dimensionale lijst verhuisd worden. Met behulp van extra predicaten zoals `mover:50:board_replace/4` is dit intuïtief.

2.1.2 Metadata

De metadata omvat de mogelijkheden tot rokade en en-passant-bewegingen. Deze worden opgeslagen in een lijst met 2 sublijsten (in de code vaak geschreven als `SpecialMoves`) die deze mogelijkheden voorstellen voor zwart en wit. De huidige speler wordt apart doorgegeven in het programma, meestal met de variabele `CurrentPlayer`.
Wanneer er in het programma `BoardConfig` gebruikt wordt, dan is dat een lijst met 2 elementen: `[SpecialMoves,Board]`.

2.2 Bepalen mogelijke zetten

Voor dit deel van het programma hebben we enkel de module `mover` nodig. Het predicaat `mover:456:find_moves/4` geeft ons een lijst van alle mogelijke bordconfiguraties na het uitvoeren van 1 zet. Deze configuraties worden bepaald door `mover:460:move/5`. Het programma probeert alle mogelijke combinaties van broncoördinaten en doelcoördinaten in het `move`-predicaat, en dat predicaat zal waar zijn als het een geldige zet is, gezien de

huidige bordconfiguratie. Hier wordt dus gekeken of dat type schaakstuk die zet mag maken, of het type schaakstuk de juiste kleur is, of de huidige speler niet schaak staat... Zoals in de opgave van het project gevraagd wordt, controleert `move/5` ook op rokade- en en-passant-mogelijkheden.

2.3 Keuze beste zet

De implementatie van dit deel staat in de module `score`. Nu we voor een gegeven bord alle geldige zetten kunnen verzamelen, kunnen we ook een score toekennen aan iedere nieuwe bordconfiguratie. De configuratie met de hoogste score zal dan de volgende zet zijn die we wegschrijven.

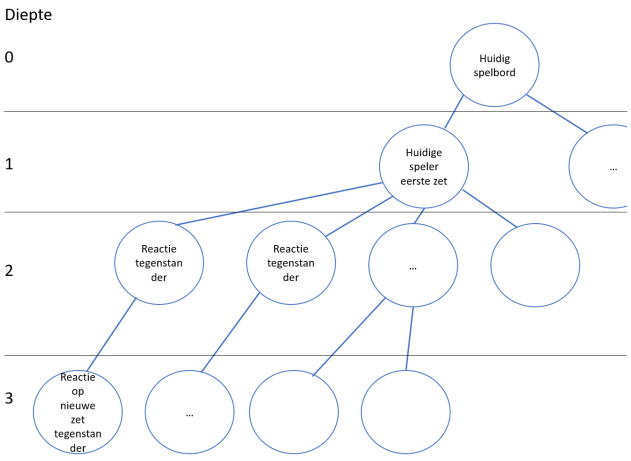
2.3.1 Score

De score van het bord wordt bepaald in `score:35:score/2`. Een positieve score stelt een bord voor dat voordelig is voor de witte speler. Een negatieve score is beter voor de zwarte speler. We geven ieder schaakstuk een bepaalde waarde [1][2], en die waarde wordt dan vermenigvuldigd met (1) of (-1) respectievelijk of het schaakstuk wit of zwart is. De som van al deze waarden is dan de score van dat bord.

2.3.2 Minimax-boom

De minimax-boom wordt opgesteld met diepte 3 zoals visueel voorgesteld op Figuur 1. Meer uitleg over de keuze van deze diepte staat in paragraaf 6.

`score:80:minimax/3` zal het predicaat `score:89:minimax_step/6` aanroepen. Dat laatste predicaat werkt dan recursief tot de op voorhand bepaalde diepte is bereikt. Het genereert dan van iedere top op die diepte een score. Vervolgens wordt de beste score[1] van alle kinderen doorgegeven naar de ouder. Herhaal dan dit proces voor alle toppen op de diepte van de ouders, tot er 1 optimaal bord wordt gevonden in de wortel van de boom.



Figuur 1: Voorbeeld spelboom

3 Uitwerking voorbeeld

Alle bestanden in deze paragraaf staan in de directory `extra/example/`. We gebruiken onderstaande voorbeeldinput[2] (`inputboard_example.txt`):

Na het parsen wordt het bord voorgesteld zoals in `parsed_board_example.txt`.

`mover:456:all_moves/4` bepaalt nu alle mogelijke zetten door alle mogelijke unificaties op te slaan in een lijst. Een mogelijke unificatie moet true geven voor `mover:460:move/5`. We zoeken in essentie een bron- en doelcoördinaat die samen een zet voorstellen. We bestuderen even aan welke voorwaarden we moeten voldoen:

- Het bron- en doelcoördinaat moeten geldige coördinaten op het spelbord zijn.
- Het schaakstuk op het broncoördinaat moet de kleur hebben van de huidige speler.
- Het doelcoördinaat moet een geldig coördinaat zijn (bvb: niet vooruit bewegen met een loper, niet over een schaakstuk springen met een toren, niet eindigen op een schaakstuk van eigen kleur) voor dat specifieke schaakstuk.
- De koning van de huidige speler mag na deze zet niet schaak staan.

We kunnen met de huidige bordconfiguratie dus bijvoorbeeld unificeren met broncoördinaat[3] `(3,4)` en doelcoördinaat `(7,4)`. Alle mogelijke zetten staan beschreven in `all_moves_example.txt`.

Herhaal nu voor iedere mogelijke zet het bovenstaande proces door diepte-eerst te itereren: we zoeken nu voor het nieuwe bord opnieuw alle mogelijke stappen. Deze keer gaat het dus om de stappen van de tegenstander. Blijf herhalen tot we diepte 3 bereiken in de boom. In de code gebeurt dit in een combinatie van `score:89:minimax_step/6` en `score:106:best_move/7`. `minimax_step` stelt alle kinderen van een top op (diepte-eerst) en `best_move` zoekt daarna naar het beste kind.

We zitten nu in een mogelijke bordconfiguratie waarbij we 3 stappen verder zitten dan het originele inputbord. De laatste zet is dus gezet door de speler die we willen laten winnen (zwart). Doordat we zwart willen laten winnen, nemen we het bord dat het beste

[1] 'Beste' score hangt hier af van de context: indien de top in de boom een zet van de huidige speler voorstelt, zullen we de score proberen maximaliseren, anders minimaliseren.

[2] Zie de tests in de tests-directory voor meer voorbeelden.

[3] Volgens de interne voorstelling zoals beschreven in deelparagraaf 2.1

is voor zwart, dus het bord met de meest negatieve score. We geven de score door aan de ouder.

Deze ouder stelt nu een zet van de tegenstander voor. Dit is nu onze huidige top. We vergelijken nu alle toppen van diens ouder en we verkiezen de hoogste score. We gaan er dus van uit dat onze tegenstander de optimale zet voor zichzelf maakt.

De huidige diepte is nu 1. We werken analoog aan diepte 3. De beste score wordt dan uiteindelijk de zet die zal genomen worden.

We werken de score-functie van de bovenstaande unificatie met `(3,4)→(7,4)` op diepte 1 uit:

- Rij 0: geen schaakstukken: subscore 0
- Rij 1: koning zwart → waarde=0: subscore 0
- Rij 2: pion zwart → waarde=$1*(-1)$: subscore -1
- Rij 3: geen schaakstukken: subscore 0
- Rij 4: pion wit → waarde=$1*1$ + koningin zwart → waarde=$9*(-1)$: subscore -8
- Rij 5: geen schaakstukken: subscore 0
- Rij 6: 2∗pion wit → waarde=$2*1*1$: subscore 2
- Rij 7: koning wit → waarde=0: subscore 0
- $\sum_{subscores} = 0+0+(-1)+0+(-8)+0+2+2 = -5$

De implementatie van bovenstaande score-functie staat in `score:35:score/2`.

4 Teststrategie codecorrectheid

In dit onderdeel leggen we kort onze teststrategie uit. Deze testen staan in de directory `tests`. Het uitvoeren van de testen gebeurt door het script `run_tests.sh` uit te voeren.

4.1 PLUnit

We testen hoofdzakelijk de volgende zaken:

- Parser: Gegeven een inputstring, kan de parser dit correct omzetten naar de interne voorstelling?
- Mover: Gegeven een bordconfiguratie, zijn de gegeven zetten mogelijk? Kunnen alle mogelijke zetten gevonden worden?
- Score: Gegeven een bordconfiguratie, kan de AI de juiste keuze maken tussen het nemen van een vijandige koningin en pion? Kan de AI een schaakmat-in-één-zet vinden?

Alle unit-testen slagen.

4.2 Simulatietest

In `run_tests.sh` wordt er nog een andere test uitgevoerd. Dit is een simulatietest tegen de random speler. We controleren dat het programma geen ongeldige zetten teruggeeft en steeds binnen de verwachte tijd een antwoord biedt. Voor de huidige implementatie van de schaakcomputer slaagt ook deze test.

5 Tijd en Geheugen

5.1 Systeemconfiguratie

Alle testen worden uitgevoerd op het volgende systeem:

- OS: Ubuntu Server 20.04 LTS
- CPU: Intel Core i5-7400

We willen extra opmerken dat de code onstabiel runt in een Windows Subsystem for Linux-omgeving (WSL2). We krijgen dan soms `ERROR:Out of global stack`, vooral bij grotere zoekdieptes (> 3). Na uitgebreid debuggen kon hier geen verklaring voor gevonden worden.

5.2 Testmethode

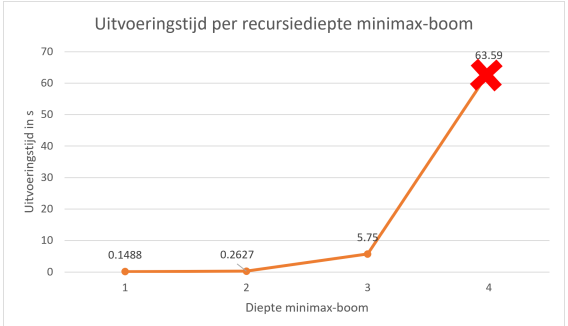
Als we praten over uitvoeringstijden, dan gaat dit steeds over de gemiddelde uitvoeringstijd van 1 beurt over 10 runs. Dit is omdat we testen tegen een willekeurige speler, en de uitvoeringstijden tussen 2 runs sterk kunnen verschillen. Het geheugengebruik is het gemiddelde gebruik na 10 runs.

Het geheugengebruik wordt gemeten aan de hand van het `usr/bin/time -v node TestEngine.js` commando, met een vaste seed. De tijdsmetingen worden dan verzameld uit de output van `TestEngine.js`.

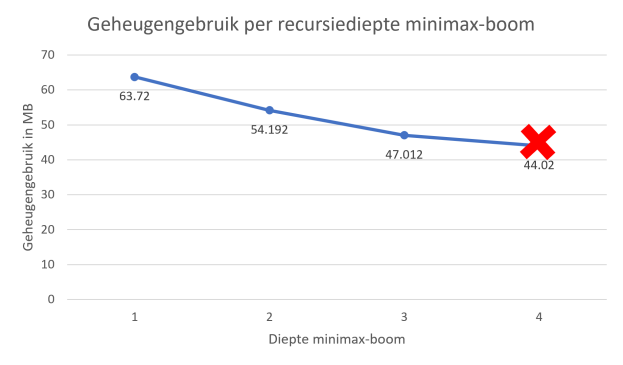
6 Resultaat

De meetresultaten staan in Figuur 2 en Figuur 3. We kunnen niet dieper dan diepte 3 testen: bij diepte 4 hebben we een time-out. Net voor de time-out was het gemiddelde 63.59 seconden.

De schaakcomputer is met diepte 3 in staat om consistent te winnen tegen de random speler.



Figuur 2: Uitvoeringstijd



Figuur 3: Geheugengebruik

Het valt op dat de uitvoeringstijd niet lineair maar exponentieel schaalt met de diepte. Dit is ook te verwachten, daar het aantal geldige bordconfiguraties ook exponentieel vergroot bij lineaire vergroting van het aantal zetten. Om een schaakcomputer te implementeren met een hoge recursiediepte zal er dus eerst nog sterk moeten geoptimaliseerd worden.

Het geheugengebruik lijkt hier geen belangrijke factor te zijn.

7 Toekomstig werk

Het grootste probleem is de uitvoeringstijd met diepte > 3. We vermoeden dat een implementatie met alpha-beta snoeien hier een verbetering zal zijn: we beperken hiermee het aantal zetten dat moet onderzocht worden door takken van de spelboom vroegtijdig af te breken als we zeker zijn dat er geen verbetering van de score mogelijk is in die tak. Dit heeft dus rechtstreekse invloed op de uitvoeringstijd.

Een geavanceerdere scorefunctie zou ervoor kunnen zorgen dat de zoekdiepte minder belangrijk wordt door een betere schatting te maken van de kwaliteit van het bord. Een mogelijke implementatie zou kunnen rekening houden met de locatie van de schaakstukken: zo zijn bijvoorbeeld pionnen meer waard als ze meer naar voor staan.

We zouden ook nog slimmer kunnen omgaan met de tijdslimiet. We merken op dat in de eindfase van het spel er niet veel schaakstukken overblijven, en de AI dus minder lang moet denken over de beurt. We zouden hier een dynamische diepte van de spelboom kunnen implementeren, die rekening houdt met het aantal schaakstukken op het bord.

8 Conclusie

We waren in staat om een schaakcomputer op te stellen die binnen de gevraagde tijd een winnende zet kan teruggeven, of een zet die de computer in een betere positie tot winnen brengt. Dit gebeurt aan de hand van een zoekalgoritme met een minimax-boom met diepte 3. Er is wel nog ruimte voor verbetering, hoofdzakelijk bij uitvoering met grotere diepte (> 3). We stellen voor om alpha-beta-snoeien te implementeren in een volgende versie.

Referenties

[1] *Chess piece relative value.* URL: `https://en.wikipedia.org/wiki/Chess_piece_relative_value`.

[2] Capablanca J. & de Firmian N. *Chess Fundamentals (Completely Revised and Updated for the 21st century).* 2006.