Verslag Schaak Engine

Lukas Barragan Torres

Inhoudsopgave

1. Inleiding

Dit verslag biedt een overzicht van het ontwerp en de implementatie van een schaak-engine. We behandelen de bordrepresentatie, bewegingslogica, het gebruikte algoritme, en de evaluatie-heuristiek.

2. Bord- en Spelvoorstelling

Mijn bord heb ik voorgesteld als een twee dimensionale lijst [1]. Een element van deze lijst is een `atom` die een stuk of een leeg vak voorstelt [2].

Een spelstaat wordt voorgesteld als een functor `state(Board, Info, ToMoveColor)` [3]. `Board` is hierbij de 2d-lijst die ik zojuist introduceerde, `Info` houdt bij welke speler nog lang en kort kan rokeren en eventueel op welk vak `en passant` kan genomen worden. Ten slotte geeft `ToMoveColor` de speler aan die op dat moment aan zet is.

3. Bewegingen

Een beweging is op verschillende niveaus gedefinieerd.

Op het allerlaagste niveau maakte ik een predicaat `on_same(+Axis, +From, +To)` [4], waarbij `Axis` de waarde `row`, `column`, `anti_diagonal` of `diagonal` kan aannemen. Dit predicaat is waar indien twee coördinaten op dezelfde `Axis` liggen (bijvoorbeeld `co(0, 0)` en `co(7, 7)` op `diagonal`).

Het predicaat `in_sight(+Axis, +Co1, +Co2, +Board)` [5] is waar indien de ene coördinaat de ander kan 'zien'. Meer specifiek, er zit geen stuk tussen de twee coördinaten op het bord.

Het predicaat `piece_rule(+Piece, +Move, +State)` [6] maakt op zijn beurt gebruik van `in_sight` om de bewegingen voor elk stuk vast te leggen. Zo kan bijvoorbeeld een toren bewegen naar alle plekken die `in_sight` zijn op zijn rij en kolom.

`basic_piece_move_unsafe(+Name, +Move, +State, -NewState)` [7] zal een stuk verzetten en een nieuwe staat teruggeven. Het predicaat kijkt of het stuk op de oorsprong wel van de huidige speler is en of er geen stukken van eigen kleur genomen worden.

Ten slotte zal `legal_move(?Move, +State, -NewState)` [8] bovenop `basic_piece_move_unsafe` kijken of de zet niet resulteert in het schaak zetten van de eigen koning.

4. Algoritme

Mijn algoritme [9] is een standaardimplementatie van mini-max met Alpha Bèta snoeien zoals in [10]. Het interessantere deel van de code is de heuristiek die een spelstaat evalueert [11].

Ik heb ervoor gekozen om mijn zoekdiepte te verlagen van 3 naar 2 in voordeel van een geavanceerdere heuristiek. De heuristiek bekijkt 4 dingen [12] om een evaluatie te maken:

1. De materiaalbalans tussen de twee spelers (gewicht 1).
2. Met hoeveel stukken een speler de centrale vakken controleert (gewicht 0.1).
3. Hoeveel vakken een speler controleert (gewicht 0.02).
4. Hoe ver de pionnen van een speler zijn op het bord (gewicht 0.01).

Deze configuratie kan de engine beperken in zijn zoekdiepte, maar kan beslissingen opleveren die op lange termijn in het spel grote voordelen opleveren. Zo zal nummer 2 en 3 zorgen voor een goede ontwikkeling in het middenspel en zal nummer 4 er voor zorgen dat de engine zijn pionnen zal promoveren in het eindspel.

5. Testen

5.1. Unit Testen

De unit tests evalueren de functionaliteit van individuele stukken voor het uitvoeren van zetten [13]. Daarnaast wordt de parser getest [14]. In het bestand `san.pl` wordt gecontroleerd of de parser alle mogelijke standaard algebraïsche notaties (SAN) correct kan parsen. Hoewel de parser technisch gezien in staat is meer notaties te parsen dan strikt noodzakelijk (zoals e5=Q), worden deze gevallen later afgehandeld.

5.2. Integratie Testen

Er is één grote integratie test, geschreven in python [15]. Deze test neemt een map met `pgn`-bestanden en kijkt voor elk spel of op elke positie alle mogelijke zetten overeen komen met de uitvoer van mijn engine (als het TEST-argument wordt meegegeven). Deze test wordt uitgevoerd op een 800-tal games van Bobby Fischer.

6. Conclusie

Ik geloof dat ik alle vereiste functionaliteit correct heb geïmplementeerd en een nauwkeurige engine heb ontwikkeld. Hoewel mijn engine bij het evalueren van posities iets minder diep zoekt dan de verwachte 3-4 zetten, gaat deze slechts tot een diepte van 2 vanwege de zwaardere heuristiek. Ik ben ervan overtuigd dat dit geen significante invloed heeft op de algehele sterkte van de engine. Dit kan echter mogelijk wel een verschil maken bij het vinden van mat-in-3 stellingen.

Ik ben tevreden met hoe ik de functionaliteit voor het verplaatsen van stukken heb gestructureerd in verschillende abstractielagen. Daarnaast heb ik geprobeerd om optimaal gebruik te maken van de mogelijkheden van Prolog. Wel moet ik toegeven dat het implementeren van de `en passant`-functionaliteit enkele uitdagingen met zich meebracht.

In totaal heb ik 44 unit tests en 1 grote integratietest geschreven, wat naar mijn mening een goede dekking van de code aantoont.

Bibliografie

[1] 'src/board.pl, lijn 35'.

[2] 'src/pieces.pl, lijn 52-55'.

[3] 'src/board.pl, lijn 22'.

[4] 'src/board.pl, lijn 96'.

[5] 'src/board.pl, lijn 83'.

[6] 'src/moves/piece_rules.pl, lijn 42'.

[7] 'src/moves/piece_rules.pl, lijn 10'.

[8] 'src/moves/moves_main.pl, lijn 15'.

[9] 'src/alpha_beta.pl'.

[10] Wikipedia contributors, 'Alpha–beta pruning — Wikipedia, The Free Encyclopedia'. [Online]. Beschikbaar op: https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1198585667

[11] 'src/score_heuristics.pl'.

[12] 'src/score_heuristics.pl, lijn 28-33'.

[13] 'tests/extra/unit/moves/*.pl'.

[14] 'tests/extra/unit/parser/*.pl'.

[15] 'tests/extra/integration/test_all_moves.py'.