Verslag Logisch Programmeren

Ruben Vandamme

15 mei 2024

Contents

1 Inleiding

De testen zijn te runnen via run_all_test_files.pl. De structuur van de testen komt overeen met de structuur van de source files. In de extra folder zitten buiten het verslag nog twee bestanden. rewrited_test.js is een copy van de originele test.js, maar dan met een aangepast pad naar de swipl installatie. predicate_args_tooltip.py is een hulpmiddel om gemakkelijker argumenten van module-geëxporteerde predicaten te vinden.

2 Datastructuren

2.1 Bord

Een bord wordt voorgesteld als 8 rijen van telkens 8 speelstukken. De eerste rij staat voor de bovenste rij van het schaakbord. De Backus-Naur notatie voor een speelstuk is als volgt:

```
<stuk>        ::= "pawn" | "rook" | "knight" | ...
<kleur>       ::= "b" | "w"
<speelstuk>   ::= <kleur> "_" <stuk> | "empty"
```

Het start bord staat beschreven in board.pl (22-32). Verder staan er in deze file enkele predicaten die deze datastructuur manipuleren zoals *place*, *remove*, enz. Om een speelstuk op een bepaalde positie op te vragen is er het generatieve *piece_at_position* predicaat.

2.2 Spel

De representatie van een spel staat beschreven in game.pl (9-19). De status variabelen van een spel worden bijgehouden in een lijst. Deze variabelen hoeven niet langer geordend te zijn aangezien we nu *member* kunnen gebruiken om een variabele op te vragen.

2.3 Zet

De representatie van een zet staat beschreven in move.pl (15-26) en werd op dezelfde manier als hierboven geimplementeerd. De representatie is een directe interpretatie van een SAN zet, op de mogelijke waarde kingcheck voor de king-of-the-hill spelversie na. Dit maakt conversie erg gemakkelijk. Het nadeel is dat de beginpositie niet exact bewaard wordt, en we deze telkens opnieuw moeten opzoeken met het *begin_position* predicaat uit position.pl (53-113).

3 Algoritme

3.1 Mogelijke zetten

Een mogelijke volgende zet voor een bepaalde spel status wordt bepaald door het predicaat *next_move* in move.pl (34-53). Eerst wordt er gezocht naar een positie op het bord dat een speelstuk van eigen kleur bevat. Afhankelijk van welk speelstuk we hebben gebruiken we een andere methode om de mogelijke volgende posities van dit speelstuk te bepalen. Speelstukken als de toren, de loper, enz. maken gebruik van een abstract begrip ‘cell‘ (zie pieces.pl (93-136)). Dit is een directionele lijn die geblokkeerd kan worden door een ander speelstuk. Bv. voor een toren is dit horizontaal naar boven of diagonaal voor een loper. Om de mogelijke posities van een pion te bepalen, moeten we ook een geschiedenis van eerdere zetten meegeven. Zo kunnen we ook en-passant of de dubbele vooruit toevoegen. Nadat al deze zetten bekeken zijn, wordt er ook nog eens gefilterd op zetten die de koning niet in gevaar brengen.

3.2 Beste zet

Om te bepalen hoe goed een zet is, wordt er gekeken naar materiële score. Deze scores staan gedefinieerd in minimax.pl (177-188). Voor king-of-the-hill wordt

hier ook nog eens score bijgeteld voor hoe dicht de koning nabij het centrum is. Vervolgens gebruiken we een minimax algoritme met diepte-limiet twee. Via alpha-beta snoeien vermijden we takken in de zoekboom die toch nooit een beter resultaat zullen opleveren. We zouden dit algoritme kunnen verbeteren door een databank van goede begin/eind-zetten bij te houden of extra scores te geven aan bepaalde pionstructuren.

4 Prestatie

De bottleneck van het programma zit duidelijk in het *piece_at_position* predicaat in board.pl (111-116). Dit komt omdat elke keer dat er een speelstuk opgevraagd wordt, alle 64 posities op het bord bekeken kunnen worden. Ook het bepalen van de begin positie en het toepassen van de eerder besproken 'cell', maken uitbundig gebruik van dit predicaat. Het opslaan van de posities en deze selectief aan te passen, zou een grote verbetering moeten geven.

Een ander probleem is dat er bij elke zet opnieuw checks gecontroleerd worden die de geschiedenis doorlopen, zoals en-passant of rokade. We zouden dit kunnen verhelpen door voor een speelstuk een variabele bij te houden die ons vertelt of deze al bewogen heeft, en-passant kan uitvoeren, enz.

In figuur 1 staan de vijf predicaten die het meeste tijd in beslag nemen voor het bepalen van de beste volgende zet (beginbord en diepte-limiet 2). Bemerk de vele redo's voor de plus uit *piece_at_position*.

Figure 1: profile van minimax_next_game_state

```
================================================================================
Total time: 2.688 seconds
================================================================================
Predicate                      Box Entries =    Calls+Redos     Time
================================================================================
lists:nth0_det/3                 3,287,242 =3,287,242+0        15.9%
board:piece_at_position/3          861,989 =  431,800+430,189  13.6%
nth0/3                           3,611,535 =3,611,200+335      12.9%
plus/3                           5,861,231 =5,861,231+0        12.9%
$seek_list/4                     3,287,242 =3,287,242+0        10.6%
```

5 Conclusie

Het maken van een correcte schaakmachine nam veel tijd in beslag. Dit niet alleen omdat er veel speciale regeltjes zijn, maar ook omdat er weinig ondersteuning is voor het schrijven van prolog code.

Tijdens het maken van het project heb ik gemerkt dat ik nog altijd erg imperatief denk. Dit is vooral te merken in de benamingen voor de predicaten die vaak doe dit of doe dat noemen. Ook denk ik vaak aan predicaten als functies. Vooral in het minimax algoritme met zijn vele argumenten is dit

duidelijk. Echter, het was ook mooi om te zien dat de DCG-predicaten in twee richtingen gebruikt konden worden.

Ik ben vrij teleurgesteld over de slechte prestatie van het minimax algoritme. Het zou leuk geweest zijn als ik ook eens in het echt tegen de AI had kunnen spelen.