# Verslag labo 4: Chess Agent

Deze laatste opdracht omvat het maken van een Chess Agent. In den beginne, werd deze uitgewerkt met “search algorithms”. Zoekalgoritmes evalueren hun “game state” enkele stappen in de toekomst om zo een beslissing te maken in het heden. Het uitwerken hiervan heeft zijn vruchten afgeworpen, maar met extra tijd besloot het team aan andere methodes te werken. Machine learning met neural networks was één hiervan. Meer uitleg hierover later.

1. **Minimax Algorithm**

Minimax is verreweg het meest gekende zoekalgoritme in schaak. Voor een meegegeven diepte, zal het algoritme elke “game state”, of node, evalueren met een getal. De “static evaluation function” speelt hierbij een belangrijke rol. Zijn opbouw in moeilijkheidsgraad is zeer breed. Bij de start van dit project is hij simpel. Naarmate de tijd verstrijkt, kunnen we zijn implementatie uiterst slim, maar ook complex, maken. De evaluation function moet zodanig uitgevoerd zijn zodat:

* Negatieve evaluatie: zwart in voordeel.
* Positieve evaluatie: wit in voordeel.

Wit neemt altijd het maxima van zijn onderste nodes. Zwart het minima. Het resultaat van dit algoritme is een positief of negatief getal. Dit duidt op het beste resultaat van het bord na het evalueren van alle states.

De code is voorzien van commentaar. Deze beschrijft, lijn per lijn, de exacte werking van ons programma. Dit verslag is geen replica hiervan, maar kaart eerder de problemen aan bij het komen tot een resultaat.

Het grootste probleem bij Minimax was het returnen van twee waarden: de evaluatie en de beste move van elke node. Bij implementaties op het internet wordt het vinden van de beste move uitgevoerd in een aparte functie. Het bleek interessanter dit mee te verwerken in Minimax. Daardoor is er maar één lijn code nodig in “train\_stockfish.py”. De witte speler hoeft maar één regel uit toe voeren bij het achterhalen van zijn move.

Na een aantal dingen te proberen, kwamen we redelijk snel tot een gewenst resultaat. In een tuple, (“boardValue”, “foundMove”), houden we deze waarden bij. In de hoogste node, zoeken we de grootste evaluatie en horend hierbij, de best move.

1. **Evaluation Function**

De eerste verwezenlijking van deze functie is het meest gekend. Elk schaakstuk krijgt een waarde. Deze zijn groter naarmate de stukken belangrijker worden. Zo heeft de koningin een veel grotere waarde dan een toren. Bij de witte stukken zijn deze getallen positief, bij de zwarte negatief. De functie evalueert het bord door de waarden van alle stukken op te tellen. Hoe groter de evaluatie, hoe beter de witte speler er voor staat.

Deze implementatie heeft echter een paar problemen. Vaak krijgen nodes dezelfde evaluatie omdat er evenveel stukken op bord blijven. De functie hangt sterk af van captures. De speler zijn grootste prioriteit is het nemen van een pion, wat niet altijd voordelig blijkt te zijn.

In het begin van het spel krijgt elke node de waarde 0 toegewezen. De reden hiervoor is simpel. Er zijn evenveel zwarte als witte stukken. Dit betekent dat de AI telkens dezelfde move zal kiezen (de laatste).

Wat is het besluit? De intelligentie van Minimax is zeer afhankelijk van de evaluation function. Hoe meer situaties we kunnen differentiëren, hoe beter Minimax is in het zoeken naar de beste move.

Leg uit hoe we dit implementeren zonder for-lus en zo de uitvoeringstijd kunnen verlagen.

Om dit te realiseren, hebben we een extra check toegevoegd aan het programma. De evaluatie hangt nu niet af van de schaakstukken hun waarde, maar ook van hun positie. Een toren staat best op de hoeken van het schaakbord want daar heb je het beste uitzicht. De evaluatie wordt groter of kleiner afhankelijk van zijn plaats op het bord.

Een evaluation function manueel ontwerpen brengt veel moeilijkheden met zich mee. Het is zeer lastig om de juiste waarden te vinden; dit voor zowel de “pieceValues” als de “positionValues”. Er is een lange zoektocht nodig om getallen te vinden die het best mogelijke resultaat teruggeven. Een vaste manier bestaat niet; alle values worden precies gekozen met de natte vinger. De beste “chess agents” doen dit dynamisch door middel van machine learning. Op het einde van dit verslag examineren we dit probleem met “neural networks”.

1. **Alpha-beta Pruning en Quiescence Searching**

“Alpha-beta pruning” is een manier om de uitvoeringstijd van het Minimax algoritme aanzienlijk te verlagen. We slagen nodes over die geen effect hebben op het resultaat. Er is al een betere waarde beschikbaar voor de witte of zwarte speler. Het verwerken van verdere nodes zou zinloos zijn. Pruning geeft geen absolute zekerheid op tijdswinst. Het zal aankomen op de volgorde van moves of sommige nodes geskipt kunnen worden. Het implementeren hiervan vroeg slechts het toevoegen van een paar lijnen code.

“Quiescence Searching” was een poging om de intelligentie slimmer te maken. Ook hier was veel over te vinden op het internet. Het zou namelijk het “horizon effect” moeten oplossen. Strategieën vallen vaak onder de radar bij het ontwikkelen van een Chess Agent. We geven hiervan een voorbeeld. Minimax neemt met zijn koningin het zwarte paard. Hij weet echter niet dat zijn koningin in de volgende move “gecaptured” kan worden door een zwarte pion. “Quiescence Searching” is een extra zoekalgoritme dat start bij de “leaf nodes”. Hij zal deze enkel uitvoeren bij de zonet uitlegde strategieën, die de evaluation function onmogelijk kan detecteren.

We hebben er uiteindelijk voor gekozen om dit algoritme te laten vallen. Het kiezen van een move werd met enkele seconden vertraagd; soms vele meer. Zijn complexiteit en inefficiëntie zijn de grootste redenen bij het weglaten van dit stuk code.

1. **Dynamic Evaluation Function**

Leg uit: machine learning + neural networks.