Halma Bot: Monte Carlo versus Alpha-Beta

Marijn Biekart-11032278, Artemis Çapari-11336390,
Jesper van Duuren-10780793, Jochem Hölscher-11007729
en Reitze Jansen-11045442
Zoeken, Sturen en Bewegen
30 juni 2017

Inleiding

Voor het vak Zoeken, Sturen, Bewegen kregen wij de opdracht een vernieuwend 'ding' te maken. Zo was er bijvoorbeeld de keuze om met één van de robots in het Robolab op Science Park aan de slag te gaan, of een algoritme te implementeren dat een spel op kan lossen. Wij hebben voor het laatste gekozen en besloten een 'Halma bot' te maken. Dit is een simulator op de computer waar men het bordspel Halma, ook wel bekend als Chinees dammen, mee kan spelen.
Het doel van het spel is de eigen pionnen zo snel mogelijk aan de diagonale overkant van het bord te krijgen. Om de beurt doen de spelers een zet, waarbij er bewogen mag worden in elke richting. Een speler heeft de keuze om te springen over een eigen of vijandelijke pion. Als een pion gesprongen heeft, mag deze in dezelfde beurt nog meer sprongen maken in alle mogelijke richtingen. Elke sprong mag echter over slechts één pion gaan.
Wij maakten voor ons onderzoek gebruik van de fast-paced versie van Halma, wat betekent dat een pion niet direct voor een andere pion hoeft te staan om eroverheen te kunnen springen. Het is echter van belang dat de afstand tussen de beginpositie van de pion en de pion waar overheen wordt gesprongen gelijk is aan de afstand tussen de laatstgenoemde en de eindpositie van de pion. Het springen heeft verder geen effect voor de pion waar overheen gesprongen wordt. De speler die als eerste al zijn pionnen aan de overkant heeft weten te brengen, is de winnaar van het spel.
De Halma bot is maatschappelijk relevant te noemen, omdat hij er voor zorgt dat mensen altijd Halma kunnen spelen, ook als er geen fysieke tegenstander in de buurt is. Dit zou bijvoorbeeld eenzame ouderen kunnen helpen

zich minder eenzaam te voelen.
Er zijn verschillende algoritmes die voor het implementeren van een spel gebruikt kunnen worden. Wij hebben er voor ons onderzoek twee gebruikt en ze met elkaar vergeleken.
Ten eerste hebben wij Alpha-Beta pruning gebruikt. Dit is een algortime sneller dan minimax, terwijl er geen informatie verloren gaat [2]. Dit wordt gerealiseerd door het aantal staten die in de zoekboom geëvalueerd worden te minimaliseren.
Ten tweede gebruikten wij het Monte Carlo algoritme. Dit algoritme slaat alle succesvolle staten op in een database en kan zo op basis van winstpercentage steeds betere keuzes maken voor een volgende zet.
In ons onderzoek hebben wij deze twee algoritmes tegenover elkaar gezet, om te kijken welk algoritme meer geschikt was voor de Halma bot. Onze onderzoeksvraag luidde dan ook: "Welke van de twee algoritmes (Alpha-Beta en Monte Carlo) kan beter Halma spelen?"
De hypothese was dat het Monte Carlo algoritme beter zou presteren. Dit was onze verwachting, omdat Alpha-Beta pruning minder de diepte in gaat en een meer oppervlakkig algoritme blijft. Op het eerste gezicht zal het lijken alsof Alpha-Beta pruning beter werkt, maar omdat het Monte Carlo algoritme beter gaat presteren naarmate hij vaker doorlopen wordt, zal blijken dat dit algoritme onder dezelfde omstandigheden toch betere resultaten op zal leveren dan Alpha-Beta pruning [1] [3].

Methode

Voor het maken van de Halma bot moesten er aan verschillende aspecten gedacht worden. Allereerst is er een programma geschreven dat de spelregels van Halma genereert. Dit programma genereert een *board-class* die het spelbord bijhoudt en verschillende *player-classes*, voor de verschillende algoritmes en de menselijke speler. In de board-class worden door middel van $self.get\_move\_players(player)$ legale zetten verzameld voor een bepaalde speler en kan hiermee getoetst worden of de zetten van de speler legaal zijn. Dit alles is geschreven in Python.
Ten tweede is er voor gezorgd dat de simulator de juiste fysieke kenmerken heeft. Deze kenmerken zijn een extensie van de visuals waar wij mee gewerkt hebben in de schaak-opdracht. Dit programma genereert een 3D spelbord met pionnen en is geschreven in VPython.
De twee belangrijke algoritmes die zijn gebruikt, zijn Alpha-Beta en Monte Carlo. Het Alpha-Beta algoritme heeft als doel het aantal te evalueren staten

in een zoekboom te minimaliseren. Het volgende voorbeeld geeft pseudo-code weer om het Alpha-Beta algoritme te illustreren. De pseudo-code is in het Engels, omdat onze code ook in het Engels geschreven is.

```
function alphabeta(node, depth, α, β, maximizing Player)
     if depth = 0 or node is a terminal node
          return the heuristic value of node
      if maximizing Player
          v := -∞
          for each child of node
              v := max(v, alphabeta(child, depth - 1, α, β,
FALSE))
              α := max(α, v)
              if β ≤ α
                  break (* β cut-off *)
          return v
       else
          v := +∞
          for each child of node
              v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
              β := min(β, v)
              if β ≤ α
                 break (* α cut-off *)
     return v
```

[4]. Ons Alpha-Beta algoritme werkt met een diepte van twee.

Het Monte Carlo algoritme is een leer-algoritme en kan op basis van heuristiek steeds betere keuzes maken. Hier volgt pseudo-code voor het algoritme.

```
database = load_data()
path = set()

def decide_move(board):
	best_move = ''
	best_score = -1
	best_board = None

	for move in legal_moves(board):
		score = 0
		new_board = board.make_move(move)

		if new_board in database:
			score = database[new_board]
		else:
			score = get_score(new_board)
			database.add(new_board)

		if score > best_score:
			best_move = move
			best_score = score
			best_board = new_board

	path.add(best_board)


def game_ended(win):
	if win:
		for board in path:
			database[board].better_score()
	else:
		for board in path:
			database[board].worse_score()

store_data(database)
```

Beide algoritmes zijn geschreven in Python.
Om de werking van de verschillende algoritmes met elkaar te vergelijken, is gekeken naar het winstpercentage van de algoritmes en de tijd die een algoritme nodig heeft om een volgende zet te bepalen. In de discussie zal verder in worden gegaan op andere aspecten waar eventueel rekening mee gehouden kan worden.

Resultaten

Na het testen van beide algoritmes is gebleken dat de allebei de algoritmes niet zo goed werkten als we verwacht hadden.
Monte Carlo deed er gemiddeld 0.01 seconde over om een volgende zet te bedenken, terwijl Alpha-Beta daar gemiddeld 3.1 seconden voor nodig had. Na 58 keer Alpha-Beta tegen Monte Carlo te laten spelen, is gebleken dat Alpha-Beta 26 keer heeft gewonnen. Het is hierbij opgevallen dat Alpha-Beta in het begin meer won, terwijl Monte Carlo aan het einde beter presteerde.

Discussie

Het is dus gebleken dat het Monte Carlo algoritme sneller een beslissing kan nemen dan het Alpha-Beta algoritme. Verder heeft het Monte Carlo algoritme een hoger winstpercentage. De verwachting was dat het winstpercentage van Monte Carlo echter nog hoger zou zijn.
Een mogelijke verklaring voor de tijdmetingen is dat het Alpha-Beta algoritme maar twee zetten vooruit kijkt. Monte Carlo heeft daarentegen een eigen "geleerde" database met zetten tot zijn beschikking. Toch is het ook een redelijk opvallend resultaat, omdat Alpha-Beta als doel heeft het zoekproces te versnellen [2]. Waarschijnlijk komt dit in ons geval omdat onze implementatie van Alpha-Beta niet optimaal is en maar twee diep kan gaan.
Het redelijk lage winstpercentage van Monte Carlo kan worden verklaard door het feit dat wij het algoritme niet erg lang hebben kunnen laten trainen. Als wij hiervoor meer tijd hadden gehad, had het algoritme wellicht beter gepresteerd. De reden dat Monte Carlo aan het einde van het testen beter presteerde dan aan het begin, is waarschijnlijk dat het heeft geleerd van spelen tegen Alpha-Beta.
Bij het vergelijken van de resultaten is met een aantal aspecten geen rekening gehouden. Zo werkt het Monte Carlo algoritme uiteindelijk sneller, maar duurt het langer om het algoritme te trainen. Zonder die training is het verschil tussen Monte Carlo en Alpha-Beta een stuk minder groot. Verder is Monte Carlo een "duurder" algoritme als er gekeken wordt naar de opslagruimte die nodig is voor de database. Omdat het algoritme alle zetten opslaat die leiden tot winst, groeit de database erg snel. Als men hier rekening mee houdt, valt er dus ook voor Alpha-Beta iets te zeggen.
Kortom, als er puur gekeken wordt naar de benodigde tijd en het winstper-

centage, is Monte Carlo een beter algoritme dan Alpha-Beta voor het spelen van Halma. Het is echter de vraag hoe de trainingsduur en benodigde opslagruimte van Monte Carlo opwegen tegen de prestatie van het Alpha-Beta algoritme.

Referenties

[1] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *AIIDE*. 2008.

[2] Donald E Knuth and Ronald W Moore. "An analysis of alpha-beta pruning". In: *Artificial intelligence* 6.4 (1975), pp. 293–326.

[3] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[4] Stuart J Russel and Peter Norvig. "A Modern Approach (2nd ed.)" In: *Artificial Intelligence* (2003).