Transparanten bij het vak Inleiding Adaptieve Systemen: Optimal Control. M. Wiering

---

Reinforcement leren

Supervised leren: leren uit gegevens die allemaal voorzien zijn van de gewenste uitkomst

Reinforcement leren: leren door het uitproberen van acties, waarbij na sommige acties een **beloning** (reward) of **straf** (punishment) wordt gegeven

Voorbeelden:

- Bepaal route van een robot
  - Beloning: als gewenste positie bereikt is
  - Straf: als de robot ergens tegen opbotst
- Speel schaak, dammen, backgammon, ...
  - Beloning: als het spel gewonnen is
  - Straf: als het spel verloren is

Straf = beloning met negatieve waarde

Inhouds opgave

- Kortste pad algoritmen
- Optimal Control, dynamisch programmeren
- Reinforcement Leren: principes
- Temporal difference leren
- Q-leren
- Model gebaseerd leren

Leerdoelen:

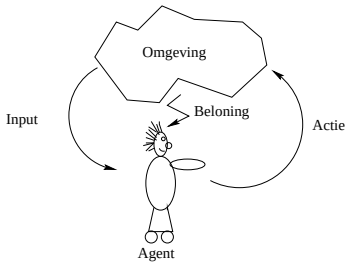
1. Markov decision problems begrijpen
2. Dynamisch programmeer algoritmen begrijpen en kunnen toepassen
3. De RL principes begrijpen en de RL algoritmen kunnen opschrijven/gebruiken.
4. Begrijpen waarom exploratie/generalisatie van belang is en manieren kunnen vertellen hoe we dat kunnen aanpakken.
5. Applicaties kunnen bedenken voor RL toepassingen.

Reinforcement leren :

Leer een agent te controlleren door acties uit te proberen en de verkregen feedback (beloningen) te gebruiken om gedrag te versterken (reinforce).

De agent interacteert met een omgeving door het gebruik van (virtuele) sensoren en actuatoren.

De belonings functie bepaalt welk gedrag van de agent het meest gewenst is.



Reinforcement leren (RL) en Evolutionaire Algoritmen (EA)

Stel je wilt leren schaken, dan wil je dus een evaluatie functie leren.

Je wilt de evaluatie van een stand weten; wat je kunt doen is de stand 1000 keer uitspelen (waarbij verschillende zetten aan bod komen) en bekijken hoe vaak er gewonnen wordt.

Door tegen jezelf te spelen, kun je op deze manier een steeds betere evaluatie functie leren (met RL).

Een andere mogelijkheid is om speler A tegen speler B te laten spelen. De winnaar gaat door en krijgt een tegenstander welke net iets afwijkt.

Door herhaaldelijk competities uit te voeren, kunnen dergelijke evolutionaire algoritmen leren schaken.

Enkele bekende applicaties

Samuel's checkers programma leerde dammen (op 64 velden) door tegen zichzelf te spelen en werd het eerste spel-programma dat de programmeur versloeg (1959).

Tesauro maakte TD-gammon (1992), een RL programma welke backgammon leerde spelen op wereldklasse nivo. TD-gammon werd veel beter dan Neuro-gammon, een programma welke supervised leren gebruikte.

Crites en Barto (1996) gebruikten RL om een controller te leren voor meerdere liften in een gesimuleerde omgeving.

Verder:

- Robot besturing
- Combinatorial optimization
- Network routing
- Verkeers controle

Kortste pad algoritmen

Reinforcement-leer algoritmen leren paden in een zoekruimte met de hoogste som van beloningen of de laagste padkosten.

Reinforcement leren kan dan ook goed gebruikt worden voor het vinden van het kortste pad.

We kennen wellicht al een aantal zoek-algoritmen welke het kortste pad kunnen berekenen (vb. Breadth-first zoeken).

Voor het kortste-pad probleem in een netwerk, bestaan er echter efficientere algoritmen.

De besproken zoek algoritmen zijn inefficient omdat ze de zoekboom voor elke zoekknoop steeds opnieuw expanderen.

Dijkstra's kortste pad algoritme (1959) is het meest efficient, als de omgeving deterministisch en bekend is.

Dijkstra's kortste pad algoritme (DKPA)

DKPA maakt gebruik van de structuur van het probleem (een graaf i.p.v. een boom)

Het houdt voor elke zoekknoop de volgende informatie bij:

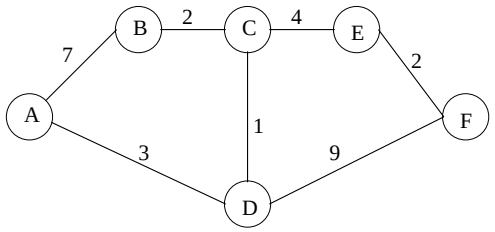
- De toestand
- De minimaal gevonden afstand tot de beginknoop
- De vaderknoop welke aangeeft wat de vorige toestand in het best gevonden pad is.

We verdelen de toestanden over 2 verzamelingen:

- een verzameling waarvan we weten dat we het kortste pad hebben bepaald
- een verzameling waarvan we dat nog niet weten.

De datastructuur: $(knoop, kosten, vader, geexp)$.

Voorbeeld zoekprobleem



(1) Initialisatie: Zet beginknoop op padkosten 0. Hiervan is optimale pad bekend (maar knoop is niet geexpandeerd). Zet andere knopen op maximale waarde
Bekend = $[(A, 0, [A], 0)]$
Onbekend = $[(B, 1000, []), (C, 1000, []), (D, 1000, []), (E, 1000, []), (F, 1000, [])]$

(2) Expandeer bekende, niet-geexpandeerde knoop (A):
Bekend = $[(A, 0, [A], 1)]$
Onbekend = $[(B, 7, [BA]), (C, 1000, []), (D, 3, [DA]), (E, 1000, []), (F, 1000, [])]$

(3) Onbekende knoop met kortste padkosten wordt bekend (D):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 0)]$
Onbekend = $[(B, 7, [BA]), (C, 1000, []), (E, 1000, []), (F, 1000, [])]$

(2) Expandeer bekende, niet-geexpandeerde knoop (D):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1)]$
Onbekend = $[(B, 7, [BA]), (C, 4, [CDA]), (E, 1000, []), (F, 12, [FDA])]$

(3) Onbekende knoop met kortste padkosten wordt bekend (C):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 0)]$
Onbekend = $[(B, 7, [BA]), (E, 1000, []), (F, 12, [FDA])]$

(2) Expandeer bekende, niet-geexpandeerde knoop (C):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1)]$
Onbekend = $[(B, 6, [BCDA]), (E, 8, [ECDA]),$
$(F, 12, [FDA])]$

(3) Onbekende knoop met kortste padkosten wordt bekend (B):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1),$
$(B, 6, [BCDA], 0)]$
Onbekend = $[(E, 8, [ECDA]), (F, 12, [FDA])]$

(2) Expandeer bekende, niet-geexpandeerde knoop (B):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1),$
$(B, 6, [BCDA], 1)]$
Onbekend = $[(E, 8, [ECDA]), (F, 12, [FDA])]$

(3) Onbekende knoop met kortste padkosten wordt bekend (E):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1),$
$(B, 6, [BCDA], 1), (E, 8, [ECDA], 0)]$
Onbekend = $[(F, 12, [FDA])]$

(2) Expandeer bekende, niet-geexpandeerde knoop (E):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1),$
$(B, 6, [BCDA], 1), (E, 8, [ECDA], 1)]$
Onbekend = $[(F, 10, [FECDA])]$

(3) Onbekende knoop met kortste padkosten wordt bekend (F):
Bekend = $[(A, 0, [A], 1), (D, 3, [DA], 1), (C, 4, [CDA], 1),$
$(B, 6, [BCDA], 1), (E, 8, [ECDA], 1), (F, 10, [FECDA])]$

Complexiteit Dijkstra's kortste pad

De complexiteit van Dijkstra's kortste pad is $O(n^2)$ voor een naieve implementatie (n is het aantal toestanden).

De complexiteit kan verbeterd worden door het gebruik van efficientere datastructuren om het minimum te vinden van de toestanden waarvan er 1 bekend (en minimaal) is.

Dijkstra's algoritme werkt alleen als alle operatoren een positieve kosten hebben.

Dijkstra's algoritme werkt alleen voor deterministische omgevingen.

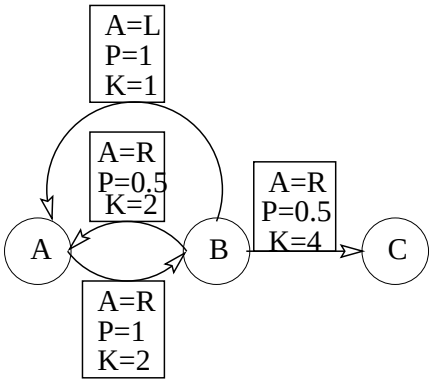
Dijkstra's algoritme kan alleen gebruikt worden als de kosten en het effect (de opvolgende toestand) van operatoren bekend zijn.

Dynamisch Programmeren

We kunnen dynamisch programmeren gebruiken als:

- De omgeving stochastisch is (er zijn meerdere mogelijke opvolgende toestanden als een operator gebruikt wordt in een bepaalde toestand)
- Operatoren een negatieve kosten kunnen hebben
- De effecten van operatoren bekend zijn.

Als een omgeving stochastisch is, kan het gebeuren dat de padkosten van een toestand afhankelijk is van vadertoestanden. Dit zorgt voor cyclische afhankelijkheden tussen toestanden.



Rekenen met Dynamisch programmeren

Als een omgeving stochastisch is, moeten we de gemiddelde padkosten berekenen. Tijdens het genereren van een pad, laten we een random nummer generator bepalen wat de uitkomst van een operator is.

Zo zijn voor bovenstaand probleem de volgende paden mogelijk (als steeds actie = R gekozen wordt):

$[A, B, C]; [A, B, A, B, C]; [A, B, A, B, A, B, C]$ etc.

Elk van deze paden heeft een kans:

$P([A,B,C]) = 0.5$; $P([A,B,A,B,C]) = 0.25$; etc.

Elk van deze paden heeft ook een totale kosten:

$K([A,B,C]) = 6$; $K([A,B,A,B,C]) = 10$ etc.

De verwachte padkosten V van A naar C als steeds actie R gekozen wordt is dan het gemiddelde:

$$V([A,C]) = P([A,B,C])K([A,B,C]) + P([A,B,A,B,C])K([A,B,A,B,C]) + \ldots$$

NB. Als in toestand B de actie L gekozen wordt, dan zijn de padkosten $V[A,C]$ oneindig groot!

We schrijven daarom ook vaak $V^{\Pi}[A,C]$, waarin de policy Π aangeeft welke actie er in elke toestand gekozen wordt.

Gebruik maken van afhankelijkheden

Als we naar het probleem kijken, zien we dat de kosten van A naar C gelijk zijn aan de kosten om van A naar B te gaan plus de kosten om van B naar C te gaan.

Dit kunnen we opschrijven als:

$$V[A,C] = K[A,B] + V[B,C] = 2 + V[B,C] \quad (1)$$

Hetzelfde kunnen we doen voor toestand B (we gaan er weer van uit dat actie R gekozen wordt:

$$\begin{aligned} V[B,C] &= 0.5(K[B,A] + V[A,C]) + 0.5(K[B,C] + V[C,C]) \\ &= 0.5 * 2 + 0.5V[A,C] + 0.5 * 4 + 0.5 * 0 \\ &= 3 + 0.5V[A,C] \end{aligned}$$

Door gebruik te maken van de afhankelijkheden:

(1) $V[A,C] = 2 + V[B,C]$, en
(2) $V[B,C] = 3 + 0.5V[A,C]$

Kunnen we $V[A,C]$ en $V[B,C]$ berekenen.

Methoden om de V-functie te berekenen

Dit kunnen we doen door:

- Te itereren: begin met V[A,C] = 0 en V[B,C] = 0. Bereken de waarden dan steeds opnieuw door gebruik te maken van de vergelijkingen.
- Te elimineren: We kunnen afhankelijkheid (1) invullen in (2) en verkrijgen:

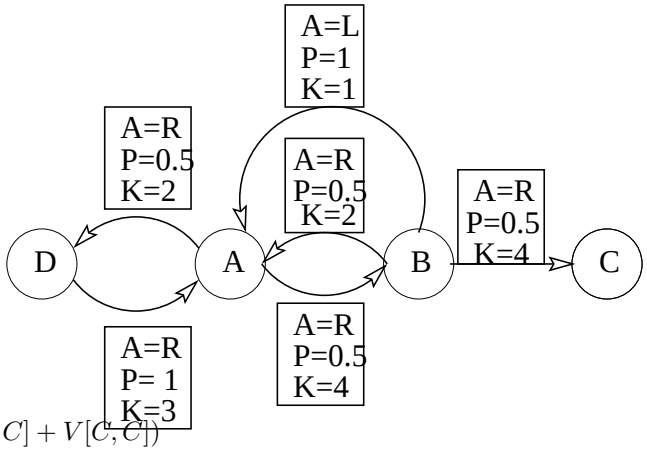
$$\begin{aligned} V[B,C] &= 3 + 0.5(2 + V[B,C]) \\ 0.5V[B,C] &= 4 \\ V[B,C] = 8 \end{aligned}$$

Hieruit volgt: $V[A,C] = 10$.

Hiervoor gingen we ervan uit dat de policy Π al bekend was. Dan kan V^{Π} berekend worden.

Voorbeeld

Beschouw de volgende cyclische graaf. Stel dat altijd actie = R wordt geselecteerd door de policy. Bereken nu de waarde functie.



Berekenen van actie waarden

Normaal gesproken willen we de optimale policy Π^* berekenen.

Hiervoor gebruiken we Quality (Q)-waarden voor acties.

Voorbeeld: $Q([A,C],R) = 2 + V[B,C]$
Voorbeeld: $Q([B,C],L) = 1 + V[A,C]$
Voorbeeld:
$Q([B,C],R) = 0.5(2+V[A,C])+0.5(4+V[C,C])$

Gegeven de waarden $Q([B,C],L)$ en $Q([B,C],R)$ kunnen we beste actie selecteren (degene met laagste Q-waarde).

Op deze manier kunnen we V uitdrukken in Q:

$$V([B,C]) = \min_a Q([B,C],a)$$

Gegeven de Q-waarden kunnen we nu ook de beste actie selecteren:

$$\Pi([B,C]) = \arg\min_a Q([B,C],a)$$

Het algemene geval

We willen de doeltoestand niet altijd expliciet vermelden, er kunnen immers meerdere doeltoestanden zijn.

Daarom schrijven we simpelweg $V(S)$ en $Q(S,A)$ voor de waarden (verwachte padkosten) vanuit toestand S.

Verder gebruiken we $P(S,A,T)$ voor de kans op een overgang van toestand S naar toestand T als actie A geexecuteerd wordt.

$K(S,A,T)$ beschrijft de kosten om met actie A van toestand S naar toestand T te gaan.

Nu kunnen we een dynamisch programmeer algoritme gebruiken voor het berekenen van alle V- en Q-waarden en de optimale policy.

Value iteration:

- **(1) Initialisatie** V(S) = 0 ; Q(S,A) = 0 voor alle toestanden en acties.
- Herhaal stappen (2-4) voor alle (S,A) paren totdat de waardefunctie V niet of nauwelijks meer verandert.
- **(2) Iteratie : bereken Q-waarden** $Q(S,A) = \sum_T P(S,A,T)(K(S,A,T)+V(T))$
- **(3) Iteratie : bereken V-waarden** $V(S) = \min_A Q(S,A)$
- **(4) Iteratie : bereken nieuwe policy acties**: $\Pi(S) = \arg\min_A Q(S,A)$

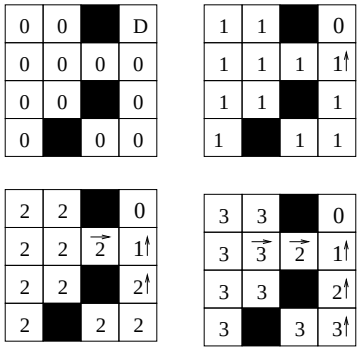
Dit levert de optimale policy op. Helaas kan het itereren totdat de waarde-functie V niet meer verandert oneindig lang duren.

Als we eerder stoppen, verkrijgen we een suboptimale oplossing, welke beter wordt naarmate er langer geitereerd wordt.

Voorbeeld: dynamisch programmeren

Beschouw een deterministische doolhof. De kosten van alle acties zijn 1. G is doeltoestand ($V(G) = 0$).

Als we value iteration toepassen krijgen we achtereenvolgens:



De complexiteit van dynamisch programmeren voor een deterministische doolhof = $O(NAL)$, waarbij N het aantal toestanden, A het aantal acties, en L het langste optimale pad is.

Dynamisch programmeren kunnen we niet gebruiken als de effecten en kosten van operatoren onbekend zijn.

Conclusie

- Dijkstra's kortste pad algoritme kan gebruikt worden als de omgeving bekend is en alle acties positieve kosten hebben en alle acties deterministisch zijn.
- Dynamisch programmeren kan gebruikt worden als de omgeving bekend is. Acties kunnen negatieve kosten hebben en niet-deterministisch zijn.
- Dynamisch programmeren berusten op een V-functie welke de waarde om in een toestand te zijn schat en op een Q-functie welke de waarde van een actie in een toestand schat.

- DP kan niet gebruikt worden als de omgeving onbekend is.