MARKOV-BESLUTSPROCESSER I SCALA

Karl Herler, 34067

Referat

Markov-beslutsprocesser är en typ av planeringsalgoritm inom artificiell intelligens som ofta används för att planera i situationer som innehåller en viss slump och osäkerhet (*stokastik*), t.ex. för att styra en robot i verkliga världen, där osäkerhet kan komma från störningar i sensorer eller handlingar som inte nödvändigtvis lyckas. Markov-beslutsprocesser har sin basis i dolda Markovmodeller och Bayesisk sannolikhet. Målet med algoritmen är att skapa en optimal "policy" för agenten* att agera efter för varje tillstånd den kan vara i, d.v.s. att oberoende av vilket tillstånd agenten är i så har den en (optimal) "regel" att agera efter för att förbättra sin situation.

Scala är ett rätt så nytt programmeringsspråk (första versionen lanserades 2003) och namnet Scala kommer från orden "scalable" (skalbar) och "language" (språk). Språket i sig innehåller inslag av både funktionella och imperativa språk och är implementerat för att köras i antingen Java Virtual Machine (JVM) eller Microsofts .NET-plattform. Språket är mycket flexibelt och stöder många principer som är relevanta för modern AI-programmering.

*) Agent = den artificiella intelligensen samt dess "sensorer" och sätt att interagera med omgivningen

Innehållsförteckning

1 Inledning

En Markov-beslutsprocess är en planeringsalgoritm som tillhör ämnet artificiell intelligens. För att förstå Markov-beslutsprocesser behöver man först förstå de problem som Markov-beslutsprocesserna strävar efter att lösa, och man behöver även förstå varför man använder just denna algoritm för det och var det inte lönar sig att använda algoritmen.

Artificiell intelligens är ett ämne som är känt för att vara mycket svårdefinierat. Detta kan delvis förklaras av att intelligens i sig inte är lätt att definiera. En tidig definition som försöker undvika problemet med att definiera intelligens är den som gavs av Alan Turing [33]. Turing föreslog att man, istället för att strikt försöka definiera termerna, skulle kunna definiera en maskin som intelligent om maskinens beteende inte kan urskiljas från beteendet hos något som vi definierar som intelligent (en människa). Även om denna definition undviker problemen med att definiera intelligens har den vissa problem. Specifikt begränsar Turings definition den artificiella intelligensens prestanda till det man jämför den med. T.ex. skulle en maskin som är snabbare än en människa på att räkna relativt lätt kunna urskiljas från en människa och därmed inte vara intelligent. John McCarthy [6], som för övrigt var den som myntade uttrycket artificiell intelligens, ger en enklare definition av det: *"The science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."*

För artificiell intelligens, precis som för annan problemlösning, är det oftast relevant att veta vissa saker om miljön där intelligensen agerar. Russell och Norvig [31, s. 40-44] definierar de fem faktorer som beskriver en problemmiljö: Fullt observerbar vs partiellt observerbar, deterministisk vs stokastisk, statisk vs dynamisk, diskret vs kontinuerlig och harmlös vs fientlig [min översättning]. Planeringsalgoritmer för kombinationen fullt observerbar, deterministisk, statisk,

diskret och harmlös är mycket välundersökta och problem med den kombinationen av faktorer löses ofta av grafsökningsalgoritmer. Grafsökning har även tillämpats mycket framgångsrikt i partiellt observerbara och fientliga miljöer. Grafsökning blir dock problematiskt i miljöer med stokastik och dynamik, både på grund av att graferna lätt blir för stora för att hantera och för att det är svårt att med säkerhet minska på dem.

Markov-beslutsprocesser används i områden fullt observerbar, stokastisk, dynamisk, diskret och harmlös eller fientlig. Det finns även en variant av Markov-beslutsprocesser som kallas partiellt observerbara Markov-beslutsprocesser som kan användas i partiellt observerbara miljöer. Kombinationen stokastisk och dynamisk är intressant eftersom mycket i den verkliga världen har sådana parametrar. T.ex. en robots rörelser där stokastik kan uppstå på grund av hjulens bristande grepp, sensordata där stokastiken kan komma från misstolkningar eller störningar i mätningarna, eller planering där stokastiken kan orsakas av att miljön förändras mellan planeringsfasen och genomförandefasen. Eftersom stokastiska och dynamiska miljöer är så vanliga är det nödvändigt att det finns algoritmer för att artificiella intelligenser skall kunna fungera i verkligheten.

För att implementera Markov-beslutsprocesser effektivt krävs det en del specifika hjälpmedel såsom stöd för stokastik, stöd för delvis autonoma agenter o.s.v. Vissa programmeringsspråk är även bättre lämpade än andra för detta ändamål. Jag har valt att undersöka programmeringsspråket Scalas lämplighet genom att undersöka hur implementationer av Markov-beslutsprocesser i Scala ser ut. Jag valde Scala eftersom det vid första anblicken verkade vara ett mycket lämpligt språk för artificiell intelligensprogrammering. Markov-beslutsprocesser är ett intressant exempel på modern artificiell intelligens och därmed ett passande exempel för att illustrera de krav som sätts av modern artificiell intelligensprogrammering. Dessutom har Scala, så vitt jag vet, inte använts till detta ändamål förr, i alla fall inte för allmän kännedom.

Programmeringsspråket Scala är ett nytt programmeringsspråk vars första version lanserades år 2003. Namnet Scala kommer från orden “scalable” (skalbar) och “language” (språk). Med skalbar menar Odersky att språket kan användas till både små och stora projekt. Språket i sig innehåller inslag av både det funktionella och det imperativa paradigmet och är implementerat för att köras i främst Java Virtual Machine (JVM), men det finns även en implementation av Scala för Microsofts .NET-plattform.[7]. Scalas fördelar för artificiell intelligens och speciellt Markov-beslutsprocesser kommer troligen att ses bland annat i stödet för olika programmeringsparadigm, aktörmodell-samtidighet (Actor model concurrency) samt Scalas interoperabilitet med Java och dess rika programbibliotek och kompatibilitet med existerande system.

2 Markov-beslutsprocess

En *Markov-beslutsprocess*, ofta förkortad *MDP (Markov Decision Process)*, är en algoritm för planering i situationer med slumpfaktorer, så kallad stokastik. Planeringsalgoritmen Markov-beslutsprocess har många namn beroende på från vilken bakgrund man kommer in på ämnet: *kontrollerade Markovprocesser*, *kontrollerade Markovkedjor* eller *Markov-beslutskedjor* [3, s. 411]. Algoritmen presenterades för första gången av Richard E. Bellman år 1957 [9]. Markov-beslutsprocesser används i dag med stor framgång inom bland annat robotik, ekonomi [30], tillverkningsindustri [1], biologi [8], geologi, spel och signalbehandling [2].

Markov-beslutsprocesser är, som alla planeringsalgoritmer, en typ av optimerings-algoritm. Optimeringsmålet för MDP:n är att hitta den bästa möjliga strategin att agera efter i alla diskreta situationer och tillstånd. Detta optimeringsmål skiljer sig från t.ex. grafsökning, vars mål är att hitta en enskild optimal lösning för problemet (eller ett delproblem) och sedan agera efter den lösningsstrategin. Det är möjligt att bevisa att den optimeringstrategi som Markov-beslutsprocesser använder sig av resulterar i en optimal lösning för hela problemet, detta bevis är dock aningen komplicerat och ryms inte med i denna avhandling men finns väldokumenterat i Bellmans artikel [9].

2.1 Varför Markov-beslutsprocesser?

Eftersom det finns en hel del olika planeringsalgoritmer är frågan "När skall man använda just Markov-beslutsprocesser?" mycket relevant och ett bra sätt att få en överblick över omgivningen för planeringsalgoritmer inom artificiell intelligens.

De äldsta men ännu idag vanligaste planeringsalgoritmerna är grafsökning i olika varianter. Grafsökning används för en hel del problem inom artificiell intelligens, som t.ex. ruttplanering, handelsresandeproblemet (travelling salesman problem), schackspelande, design av kretskort, proteindesign och internetsökning [31 s. 64-68]. Man kan till och med se att delar av Markov-beslutsprocess-algoritmer är en form av grafsökning [3 s. 411]. Grafsökning fungerar speciellt bra på problem med klart definierad struktur, med ett eller flera distinkta mål, en statisk miljö och deterministiska handlingar. T.ex. ruttplanering där miljön är känd och vi vet målet kan direkt implementeras som grafsökning med t.ex. *A* algoritmen* [31 s. 94-104]. Problem uppstår dock med grafsökning då man introducerar stokastik i problemet. Problemen uppstår främst i tre former:

1. **Förgreningsfaktorn blir för stor.** I en miljö där man kan röra sig i fyra riktningar och där det finns en risk att rörelsen misslyckas och resulterar i ingen rörselse alls, har grafen en förgreningsfaktor på 4^2 per möjligt rörelsebeslut. Den resulterande grafen har då n^{16} stadier, där n är antalet rörelser till målet i värsta fall.

2. **Grafen blir för djup (med oändliga cykler)**. Eftersom en handling med stokastik kan misslyckas obestämt antal gånger, måste grafen representera en oändlig sekvens med misslyckade handlingar.

3. **Många stadier besöks om och om igen.** Detta medförs av föregående problem.

Eftersom vi har dessa problem är det nödvändigt att hitta en alternativ algoritm för att hantera stokastik i planeringsproblem och till denna typ av planeringsproblem har vi Markov-beslutsprocesser. Markov-beslutsprocesser kan hantera stokastiken med hjälp av *Markovkedjor* och *Bayesisk sannolikhetslära*.

Även om Markov-beslutsprocesser är användbara i många situationer där traditionell grafsökning har svårigheter, ställer Markov-beslutsprocesser vissa

krav på miljön som den tillämpas i. Ett av kraven är att miljön måste vara fullt observerbar. Det vill säga att under hela algoritmens exekvering måste den artificiella intelligensen kunna observera miljön i sin helhet och veta var denna är i miljön. Det finns varianter av Markov-beslutsprocess-algoritmer där detta krav inte existerar, bland annat i varianten *partiellt observerbar Markov-beslutsprocess*. Markov-beslutsprocess-algoritmen är även en diskret algoritm, det vill säga att händelser sker i diskreta steg. Om miljön som algoritmen används i är kontinuerlig och det inte går att approximera med diskreta händelser, finns det även en kontinuerlig variant av algoritmen som kallas *Markov-beslutsprocess i kontinuerlig tid* [3].

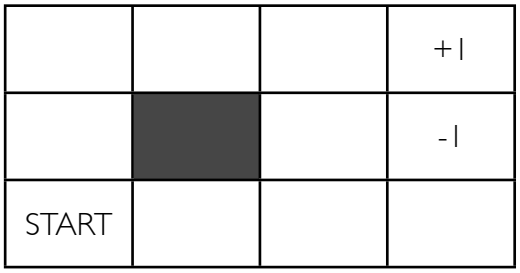
2.2 Förkunskaper

Eftersom Markov-beslutsprocesser är en algoritm med sin grund i sannolikhetslära behövs det en viss kunskap inom det ämnet. Jag kommer dock att anta att läsaren är bekant med grundläggande sannolikhetslära såsom oberoende och beroende handlingar, kausalt och diagnostiskt resonemang. Förståelse för Bayes sats och dess tillämpningar antas.

Ett grundantagande i Markov-beslutsprocessen är att alla händelser i processen är *Markovianska*, det vill säga att de har den så kallade *Markovegenskapen*. Markovegenskapen betyder att sannolikheten för händelsen är oberoende av tidigare händelser. Man kan även säga att Markovegenskapen betyder att miljön antas sakna minne och att historiska händelser inte påverkar nuvarande händelser.

Förståelse för *Markovmodeller* eller *Markovkedjor* förutsätts inte för att förstå denna avhandling men är nyttiga för en större förståelse för Markov-beslutsprocesser och rekommenderas åt läsare med mera intresse för ämnet.

2.3 Problemformulering



Figur 1 är en grafisk representation av ett exempelproblem för en Markov-beslutsprocess där vi har en enkel miljö med 3x4 tillstånd, ett positivt sluttillstånd med R((3,4))=*+1* och ett negativt *R((2,4))=-1*. Källa: Russel & Norvig, [31, s. 614]

För att använda en Markov-beslutsprocess behöver man: information om hela miljön som algoritmen agerar i, ett starttillstånd, ett eller flera mål, ett, flera eller inget tillstånd man vill undvika, samt en modell för handlingar, även kallad övergångsmodell.

Resultatet av Markov-beslutsprocessen är en ”beslutspolicy” som är definierad för alla tillstånd i miljön som algoritmen körs i. Policyn skrivs oftast som en funktion $\pi(s)$, där s är ett tillstånd. En optimal policy definieras som π^*. Intelligensen agerar sedan efter policyn genom att välja den övergång som maximerar intelligensens “nytta”, d.v.s. den övergång som är mest fördelaktig. Hur detta går till gås igenom i större detalj i nästa sektion.

2.4 Markov-beslutsprocess-algoritmen

Själva Markov-beslutsprocessen består av två faser: skapande av en “nyttofunktion” och skapandet av en "policy" för olika tillstånd. Nyttofunktionen dikterar hur policyn kommer att se ut, medan policyn används för att göra beslut om handling. En policy som skapas för en samling mål går inte att använda för en

annan samling mål eller för en annan belöningsfunktion. Om miljön eller övergångsmodellen förändras måste även hela planen räknas om.

Före jag går igenom detaljerna av algoritmen behövs några definitioner som används av algoritmen.

2.4.1 Tillstånd

Mängden tillstånd definieras ofta som S. Ett enskilt tillstånd som $s \in S$. Starttillståndet definieras ofta som: S_0. I t.ex. två-dimensionella miljöer kan tillstånd också definieras som koordinattupler, exempelvis: Tillståndet (3,4).

2.4.2 Belöningsfunktion

Markov-beslutsprocessen har även en så kallad belöningsfunktion (reward function) definierad för alla tillstånd. Belöningsfunktionen kan definieras på många sätt men den vanligaste varianten är en negativ konstant i alla tillstånd som inte är mål (terminerande tillstånd). För målen används en större konstant som kan vara positiv om tillståndet är ett önskvärt resultat och negativ om tillståndet är icke-önskvärt. Belöningsfunktionen definieras ofta som: *R(s)*, där *s* är ett tillstånd.

Belöningsfunktionens påverkan på resulterande policyn är i form av en slags ackumulering av belöngsfunktionsvärden och kan ses som en beskattning på långsamt beteende. Det finns i huvudsak två sätt som belöningsfunktionen påverkar den slutliga policyn som Markov-beslutsprocessen skapar:

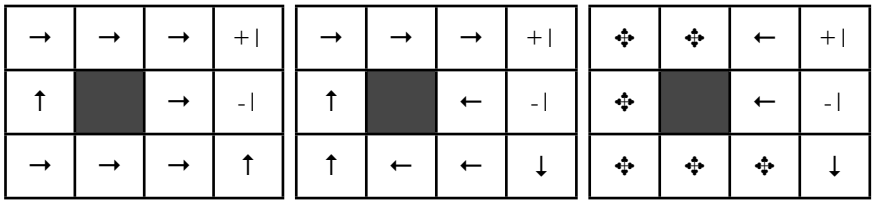
1. *Additiva belöningar* för en sekvens av tillstånd:

$$U_h([s_0, s_1, s_2, ...]) = R(s_0) + R(s_1) + R(s_2) + ...$$

2. *Diskonterade belöningar* för en sekvens av tillstånd: $U_h([s_0, s_1, s_2, ...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$, där γ är en diskonteringsfaktor i $0 < \gamma < 1$.

Av dessa alternativ är diskonteringen vanligare eftersom den ger mera flexibilitet och den additiva metoden kan ses som en delmängd av diskonteringsmetoden där $\gamma = 1$.

Belöningsfunktionen spelar en central roll i intelligensens beteende under beslutsprocessens exekvering. Om man väljer en hög negativ konstant får intelligensen ett beteende som kan karakteriseras som risktagande. Om konstanten är tillräckligt hög kan beteendet till och med karakteriseras som självskadande. En låg konstant i sin tur skapar ett säkert beteende, och en positiv konstant skapar ett målundvikande beteende.



Figur 2.1 Visar en policy med en hög negativ konstant (*R(s) < -1.6284*). 2.2. En låg negativ konstant (-0.0221 < R(s) < 0). 2.3. En positiv konstant (R(s) > 0). Källa: Russell & Norvig [31, s. 616]

2.4.3 Övergångsmodell

Övergångsmodellen definieras ofta som: *T(s, a, s')*, där *s* är ett tillstånd, *a* en handling (action) och *s'* ett potentiellt resulterande tillstånd av handling *a*. Övergångsmodellen är en konsekvens av miljön och intelligensen och kan variera i komplexitet. Övergångsmodellen är den huvudsakliga orsaken till osäkerhet.

2.4.2 Value iteration

För skapandet av planen finns det flera olika lösningsmetoder, varav de vanligaste är value iteration och policy iteration. Jag kommer bara att titta på value iteration , dels på grund av utrymmesbrist och dels eftersom policy iteration är beroende av value iteration och därmed kan ses som en abstraktion av value iteration.

Value iteration-underalgoritmen presenterades för första gången av Richard E. Bellman [ref fattas]. Algoritmen använder sig av dynamisk programmering och räknar iterativt ut ett "nyttovärde" för varje tillstånd.

Value iteration introducerar en ny funktion, nämligen nyttan av ett tillstånd *U(s)*. Nyttofunktionen kan definieras på flera olika sätt men är alltid beroende av belöningsfunktionen *R(s)* (se avsnitt 2.4.2) .

Den nyttofunktion som används av value iteration är rekursivt definierad och ser ut som följande:

$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U(s')$ och kallas för *Bellman-ekvationen*. [9]

Value iteration-algoritmen är en samling av n stycken Bellman-ekvationer, där n är antalet tillstånd. I varje iteration av value iteration uppdateras nyttofunktionens värden för varje tillstånd tills algoritmen når ett ekvilibrium. Ekvilibriet definieras av att skillnaden mellan tidigare uppdatering och nuvarande värde är inom ett toleransvärde ε (som algoritmen är garanterad att nå så länge *R(s)<0*), denna typ av konvergering kallas ofta *kontraktionsavbildning*.

Iterationssteget i algoritmen kallas för Bellman-uppdatering och definieras som:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U_i(s')$$

Figur 3 visar algoritmen i sin helhet:

funktion VALUE-ITERATION(mdp, ε) **returnera** en nyttofunktion
input: mdp, en MDP med tillstånd S, övergångsmodell T, belöningsfunktion R, diskonteringskoefficienten γ ,

ε, maximala felmarginalen som tillåts i nyttan för tillstånden

lokala variabler: U, U', vektorer med nyttan för tillstånd i S, initieras med nollor
δ , maximala förändringen i nyttan per iteration

repetera
$U \leftarrow U'$
$\delta \leftarrow 0$
för varje stadie s **i** S**:**
$U'[s] \leftarrow R[s] + \gamma \max_x \sum_{s'} T(s,a,s')U[s']$
om $|U'[s] - U[s]| > \delta$ **då**
$\delta \leftarrow |U'[s] - U[s]|$

tills $\delta < \varepsilon(1-\gamma)/\gamma$
returnera U

Figur 3 Källa: Russell & Norvig, sida 621 [31]. Översatt av mig.

Figur 4 visar resultatet av value iteration på världen i figur[x]

0.812	0.868	0.918	+1
0.762		0.660	-1
START	0.655	0.611	0.388

Figur 4 Resultatet av value iteration med $\gamma = 1$, $R(s) = -0.04$ och en övergångsfunktion ger det önskade resultatet med sannolikheten 0.8, och med en sannolik på 0.1 resulterar handlingen i en förflyttning åt höger respektive vänster. Källa: Russel & Norvig, sida 619 [31]

3 Scala

[TODO]

4 Markov-beslutsprocesser i Scala

4.1 Imperativ version

4.2 Funktionell version

4.3 Actor model parallell version

Litteraturlista

[1] Benjaar, S, Elhafsi, M., A Production-Inventory System With Both Patient and Impatient Demand Classes. *IEEE Transactions on Automation Science and Engineering*, Vol. 9, Nr. 1, 2012. s. 148-159

[2] Filippi, S. Cappé, O. Garivier, A. Optimally Sensing a Single Channel Without Prior Information: The Tiling Algorithm and Regret Bounds. *IEEE Journal of Selected Topics in Signal Processing*, Vol. 5, Nr. 1. s. 68-76

[3] LaValle, M. Steven, *Planning Algorithms*. Cambridge University Press 2009

[4] Luger, F. George, *Artificial Intelligence: Structures and strategies for complex problem solving*, Sixth Edition. Pearsons 2009

[5] Luger, F. George, Stubblefield, A. William, *AI Algorithms, Data structures, and Idioms in Prolog, List, and Java*. Pearsons 2009

[6] John McCarthy. *What is artificial intelligence*. http://www-formal.stanford.edu/jmc/whatisai/node1.html. Hämtad 4.3.2012

[7] Odersky, Martin, Spoon, Lex, Venners, Bill, *Programming in Scala*, Second Edition. Artima 2010

[8] Ozdemir, E., Sokmensuer, C., Gunduz-Demir, C. A Resampling-Based Markovian Model for Automated Colon Cancer Diagnosis. *IEEE Transactions on Biomedical Engineering*, Vol. 59, Nr. 1, 2012. s. 281-289

[9] Richard E. Bellman, A Markovian Decision Process. *Journal of Mathematics and Mechanics*. Vol. 6, No. 5, 1957, sid 679-684

[30] Ryzhov, I.O., Valdez-Vivas, M.R., Powell, W.B., Optimal learning of transition probabilities in the two-agent newsvendor problem. *Proceedings of the 2010 Winter Simulation Conference*. 2010. s. 1088-1098

[31] Russell, Stuart, Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Second Edition. Pearsons 2003

[32] Scala Standard Library 2.9.1.final. http://www.scala-lang.org/api/current/index.html. Hämtad 25.1.2012

[33] Turing A.M., Computing Machinery and Intelligence. *Mind*, nr. 59, 1950. s. 433-460