MARKOV-BESLUTSPROCESSER I SCALA

Karl Herler, 34067

Kandidatavhandling

Karl Herler, 34067

Institutionen för informationsteknologi

Våren 2012

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 "handlingspolicy" 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.

Sökord: Artificiell intelligens, programmering, Markov-beslutsprocesser, Scala, automatstyrning, robotteknik

*) 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 [28]. 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 [8], 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 [17, 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 på grund av att graferna lätt blir för stora för att hantera. Detta beror på att övergångar inte nödvändigtvis resulterar i det förväntade tillståndet och då måste man lägga till flera noder per handling. Det är dessutom svårt att med säkerhet minska på de resulterande graferna om planen görs innan handling, eftersom alla möjliga tillstånd behövs för att beskriva en komplett plan som når fram till målet. Det finns alltså ingen generell *pruningsalgoritm* för beslutsträdsplanering i miljöer med stokastik.

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 symbolisk representation, stöd för delvis autonoma agenter o.s.v. Vissa programmeringsspråk är även bättre lämpade än andra för programmering av artificiella intelligenser. Jag har valt att undersöka programmerings-språket 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 precis 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. [12]. Scalas fördelar för artificiell intelligens och speciellt Markov-beslutsprocesser kommer troligen att ses bland annat i stödet för olika programmeringsparadigm, Scalas möjligheter att beskriva beteenden, 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 [15]. Markov-beslutsprocesser används i dag med stor framgång inom bland annat robotik, ekonomi [16], tillverkningsindustri [1], biologi [13], geologi, spel och signalbehandling [3].

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 [15].

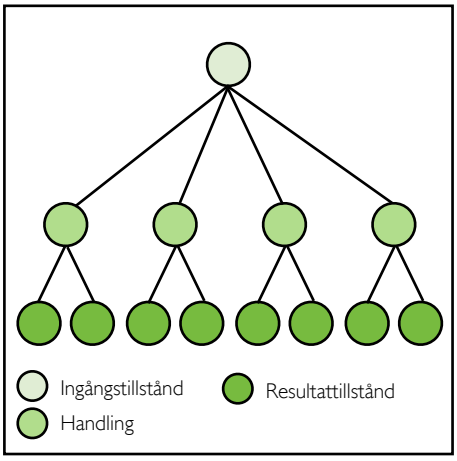
2.1 Varför Markov-beslutsprocesser?

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 [17 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* [17 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. (antalet rörelser * antalet möjliga resultat av de rörelserna) Den resulterande grafen har då n^8 tillstånd, där n är antalet rörelser till målet i värsta fall. Figur 1 visar ett exempel på förgreningsfaktorn i ett beslutsträd med stokastiska handlingar.



Figur 1

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* [5].

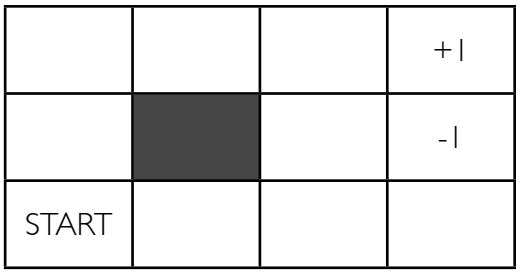
2.2 Förkunskaper

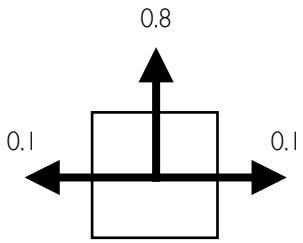
Eftersom Markov-beslutsprocesser är en algoritm med sin grund i sannolikhetslära behövs det en viss kunskap inom det ämnet. Jag kommer dock att anta att läsaren är bekant med grundläggande sannolikhetslära såsom oberoende och beroende handlingar, kausalt och diagnostiskt resonemang. Förståelse för Bayes 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 1.1 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 samt ett onåbart tillstånd (2,2) som är markerat med grå färg*.
Figur 1.2 är ett exempel på en övergångsmodell där agentens handlingar förflyttar den ett tillstånd (en ruta i figur 1.1) i önskad riktning med en sannolikhet på 0.8, ett tillstånd 90° åt annat håll än önskad handling med en sannolikhet på 0.2. I detta exempel resulterar kollisioner med väggar i ingen rörelse alls.
Källa: Russel & Norvig, [17, 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 ”handlingspolicy” 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 "handlingspolicy" för olika tillstånd. Nyttofunktionen dikterar hur

handlingspolicyn kommer att se ut. Handlingspolicyn används för att göra beslut om handling. En handlingspolicy 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 handlingspolicyn 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 handlingspolicyn som Markov-beslutsprocessen skapar:

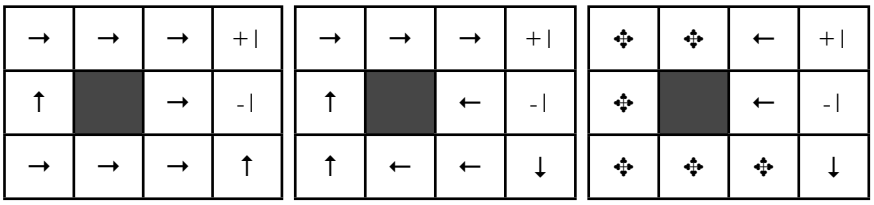
1. *Additiva belöningar* för en sekvens av tillstånd:

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

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 till exempel 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. Jag kommer att gå in på beräkningarna som skapar dessa handlingspolicyn senare.



Figur 2.1 visar en handlingspolicy med en hög negativ konstant (*R(s) < -1.6284*) på exemplet i figur 1.1. Figur 2.2 visar en policy med en låg negativ konstant (-0.0221 < R(s) < 0). Figur 2.3 visar en handlingspolicy med en positiv konstant (R(s) > 0). Figur 2.3 konvergerar inte och i de tillstånd som markerats med ✣ finns det ingen entydig handling så vilken handling som helst kan väljas av agenten. Källa: Russell & Norvig [17, 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*. [15]

I Bellman-ekvationen ovan är nyttan av ett tillstånd *s* definierat som belöningen för tillståndet *s* plus den förväntade nyttan för den handling som maximerar nyttan åt intelligensen. Intelligensen "testar" olika handlingar och ser vilken som är nyttigast och räknar sedan ut nyttan av den handlingen tillsammans med risken att handlingen inte lyckas som planerat. Det värde den får är då en multiplikation av sannolikheten att hamna i tillstånd *s'* (om man gör handling *a* i tillstånd *s, T(s, a, s')*) och nyttan för tillståndet (*U(s')*). Man multiplicerar resultatet av $\max_a \sum_{s'} T(s,a,s')U(s')$ (som man just beräknat) med ett värde gamma (γ) som kan ses som en bränsleförbrukning. Till det hela adderas en oftast negativ funktion belöning (*R(s)*), som kan ses som kostnaden att

slösa tid. I det normala fallet av ekvationen kan *R(s)*, γ , *T(s, a, s')*, *U(s')* ses som funktioner som avbildas på de reella talen, där $\sum_{s'} T(s,a,s')$ bildar en komplett sannolikhet, d.v.s. värdet 1. [17, s. 621]

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 tillstånd 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 [17]. Översatt av mig.

Figur 4 visar resultatet av value iteration (som hittas i figur 3) tillämpad tills den konvergerat på tillstånden i figur 1.1 och med övergångsfunktionen i figur 1.2. De kompletta uträkningarna för detta resultat finns i Bilaga 1. Figur 5 visar den resulterande handlingspolicyn för exemplet i figur 4.

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

→	→	→	+1
↑		↑	-1
↑	←	↑	←

Figur 4 (vänster) beskriver 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 [17]. Figur 5 (höger) beskriver den resulterande handlingspolicyn efter den value iteration som beskrivs i figur 4. Källa: Russel & Norvig, sida 615 [17]

3 Scala

3.1 Bakgrund

Första versionen av programmeringsspråket Scala designades till stor del av Martin Odersky. Odersky är professor i "Programming methods" vid *École Polytechnique Fédérale de Lausanne (EPFL)* och han är även känd för att ha skrivit Java-kompilatorn *javac* samt *Generic Java* som gav Java version J2SE 5.0 konceptet Generics. [12, s. 19-20][11]. Både Scalas design och implementering är öppna för allmänheten att bidra till och Scala-projektet har nuförtiden fler utvecklare än vad som ryms att lista här. Designförslag och diskussion om Scalas framtid sker oftast genom *Scala Improvement Process* [19] som är en hemsida där vem som helst får ge och diskutera förslag gällande designfrågor. Hela källkoden finns att ladda ner på Scalas GitHub-konto [23] där implementationsförslag och fixar också hanteras.

Första versionen av Scala släpptes år 2003 efter att det hade varit i designfasen i ungefär två år. Den första versionen av Scala implementerades för att köra på Java-plattformens virtuella maskin (*Java Virtual Machine, JVM*) och kort efter det gjordes även en version för Microsofts .NET plattform (juni 2004). Implementationen av Scala som kör på JVM är den som har fått mest uppmärksamhet och det är också den versionen jag kommer att fokusera på. När detta skrevs var den senaste versionen av Scala version 2.9.1-1 och det är den som all kod som används här skrivs för och testas på. [12, s. 12][24]

3.2 Programmeringsspråket Scala

Programmeringsspråket Scala kan klassificeras som ett objekt-orienterat språk med statisk och stark typning. Scala stöder både en imperativ och en funktionell stil av programmering, ofta rekommenderas en funktionell stil om det är möjligt och funktionella lösningar på problem anses ofta mer idiomatiska av Scala-användarna.

Scalas statiska typning stöds av en kraftfull motor för typinferens och ett typsystem som i sig självt har bevisats vara Turing-komplett [2]. Typinferensen innebär att

programmeraren sällan explicit behöver skriva ut datatyper, men han/hon kan dra nytta av kompilatorns kontrollerande av datatyper.

Scala är även fullt objekt-orienterat, d.v.s. alla datatyper är klasser som kan förlängas eller skrivas över. Detta är en rätt så ovanlig egenskap bland objekt-orienterade programmeringsspråk. I exempelvis *Java*, *C++* eller *Objective-C*, som är de populäraste objekt-orienterade programmeringsspråken just nu [27][26], är inte datatyper som Integer (heltal) eller String (textsträngar) implementerade som klasser som användaren kan ändra på. Operationerna på dessa två typer är inte heller något som användaren av språket har kontroll över. I Scala är de dock äkta klasser. Detta medför att t.ex. additionsoperationen för datatypen `Scala.Int` (heltal) är definierad som Scala-kod som går att skrivas över av en subtyp. En konsekvens av detta designval är att dyadiska metoder, t.ex. addition eller subtraktion som beskrivs i t.ex. Java, men också i Scala som `x.something(y)` har en alternativ, ekvivalent formulering i Scala, nämligen: `x something y`. Detta innebär att operationen `x + y` översätts till `x.+(y)` eller ett metodanrop på metoden vid namnet `+` i objektet `x`:s klass.

3.3 Syntax och språkliga element

Scalas syntax liknar till stor del andra språk, speciellt programmeringsspråket Java, men man kan också se influenser från Ruby, Haskell och Erlang i Scalas syntax [12 s. 12-21]. Scalas syntax försöker vara mera koncis än Javas syntax, dels genom typinferens, men också genom att helt enkelt minska på extra kod och erbjuda "smarta" standardbeteenden, t.ex. att klasser automatiskt skapar konstruktörer och getters/setters för initiala värden eller att klasser är publika om inget annat nämns. Detta exemplifieras av figur 6 och 7 som båda gör exakt samma sak.

```
class MyClass {
   private int index;
   private String name;

   public MyClass(int index, String name) {
     this.index = index;
     this.name = name;
   }
}
```

Figur 6, en enkel Java klass. Källa: Odersky, Spoon & Venners [12, s. 14]

```
class MyClass(index: Int, name: String)
```

Figur 7, klassen i figur 6 översatt till Scala. Källa: Odersky, Spoon & Venners [12, s. 14]

Exemplet i figur 7 visar även en annan syntaxdetalj. I Scala beskrivs typer med notationen `variabelnamn: Datatyp, variabelnamn: Array[Datatyp]` för en högre nivå-datastruktur (i detta fall en räcka) och `variabelnamn: List[Int] -> Int` för högre nivå-funktioner, där funktionen i fråga tar en lista med heltal och producerar ett heltal.

Scala har även, som observerades i föregående stycke, möjlighet till funktioner på högre nivå och Scala kommer med ett stort bibliotek med dessa funktioner där t.ex. alla vanliga listoperationer finns (t.ex. `map`, `zip`, `foldLeft`). Scala har även byggt in ett praktiskt sätt att beskriva korta anonyma funktioner. Ett exempel på detta ses i figur 8.

```
MyList.foldLeft(0)(_ + _) //1
MyList.foldLeft(0)((a, b) => a + b) //2
MyList.foldLeft(0)((a: Int, b: Int) => a + b) //3
```

Figur 8. Här appliceras foldLeft-operationen med ett ackumulatorvärde på 0 som utför en addition av ackumulatorn och alla element i listan. Detta kan ses som en summering av listelement. De olika versionerna är expanderingar av samma kod.

I övrigt är Scalas syntax lik Java med vissa undantag. Ett stort undantag är att alla uttryck returnerar något, till och med `for`, `if` och `while`. Om en operation inte returnerar något klokt har returvärdet datatypen `Unit` och om det ibland returnerar något har det högnivå-datatypen `Option[Datatyp]`. Det finns även andra högnivå-datatyper för distribuerade program, t.ex. `Future[Datatyp]` eller `Promise[Datatyp]` för data som inte nödvändigtvis existerar ännu. Att alla uttryck returnerar ett värde är en förändring som betyder att t.ex. for-slingor kan förkortas delvis: `val x = for(x <- X) yield x*2`. En annan skillnad i Scala är variabler. Scala följer konceptet som funktionella program följer där variabler är oföränderliga (*immutable*), men Scala erbjuder möjligheten att använda föränderliga (*mutable*, muterbara) variabler om man så önskar. Oföränderliga variabler definieras med `val`, föränderliga med `var` och funktioner med `def`. Det finns även en hel del man kan lägga före `val/var`, som t.ex. `private`, `protected` eller `lazy`. Eftersom allt i Scala returnerar något kan man även definiera funktioner som variabler (med `val` eller `var`). Skillnaden här är bara när de

exekveras, en funktion i en `val` exekveras när den kommer inom räckvidd (eller då den behövs om den använder `lazy`) medan `def` exekveras efter behov.

I Scala finns det även ett koncept som kallas för *traits*, vilket kan översättas till ”egenskaper” på svenska. Dessa traits är en form av stapelbara förändringar till klasser eller objekt som kan användas för att förlänga eller förbättra klasser. Traits kan ses som en typ av multipelt arv utan de situationer med tvetydighet som multipelt arv skapar. Traits kan även användas för ändamål som multipelt arv inte kan och Traits kan användas för att modulera kod och bygga rikare gränssnitt till klasser. Traits används flitigt av Scala-programmerare och även i de implementationer av Markov-beslutsprocesser som jag ger. Ett exempel på användning av traits är att skapa caching för en klass med långa beräkningar med t.ex. två traits, ett som heter Caching och ett som heter ToMem. Då skulle klassdefinitionen se ut som: `Class MyHeavyComputation extends Caching with ToMem`.

En annan skillnad i klasstrukturen är att Scala introducerar en så kallad situationsklass, *Case Class*. Dessa klasser finns för att förenkla arbetet med typbaserad mönsteranpassning och skapandet av små klasser. Situationsklasser är i grund och botten vanliga klasser med lite stödfunktioner som möjliggör bättre kontroll för likhet och ordning, men de möjliggör också mönsteranpassning på fält i ett objekt. Förutom detta kommer de också med ett så kallat fabriksobjekt vars mening i huvudsak är att avlägsna behovet för nyckelordet `new` när man refererar till icke-definierat objekt. Figur 9 visar ett exempel på en situationsklass som används i Markov-beslutsprocesser för att beskriva möjliga tillstånd i en tvådimensionell värld.

```
case class PossibleState(probability: Double, coordinates: (Int, Int))
```

Figur 9

4 Markov-beslutsprocesser i Scala

Jag valde att implementera Markov-beslutsprocesser i flera versioner eftersom det ger en inblick i skillnader mellan de två paradigmen i fråga. Det visar också mera av Scalas kvaliteter och designval.

4.1 Arkitektur

Hur Markov-beslutsprocesser ser ut som implementerade har till stor del att göra med tillståndsrymdens dimensionalitet. För detta exempel har jag valt att använda en tvådimensionell rymd. För att skapa bättre moduläritet har jag placerat kod som är beroende av dimensionaliteten i *traits* som heter något med dimensionaliteten (t.ex. `TwoDimensionalWorld`, `TwoDimensionalTransition`). Dessa traits placeras sedan in till en klass för Markov-beslutsprocesser och en för *Value Iteration*.

Jag valde att separera själva MDP:n från *value iteration*-delen eftersom jag programmerade flera olika versioner av *Value Iteration*. Jag har även deklarerat klasser för tillstånd, möjliga tillstånd och handlingar, vilket inte är direkt nödvändigt men det ger lite extra tydlighet och möjlighet att beskriva mera avancerade modeller för dem.

Jag har valt att inte använda några abstrakta klasser även om det finns möjlighet för dem, eftersom de för denna avhandling skapar onödig komplexitet och inte är direkt nödvändiga för tillfället.

Alla arkitekturella klasser finns i sin helhet i bilaga 2.

4.2 Imperativ version

Jag valde att börja med en imperativ version av Markov-beslutsprocesser för att de är närmast originalversionen av algoritmen och för att en imperativ stil är mera bekant för de flesta programmerare. Jag kommer inte att presentera algoritmen i sin helhet här på grund av utrymmesbrist men jag visar valda delar. Hela implementationen finns beskriven i bilaga 2.

```
while((delta > epsilon*(1-gamma)/gamma)) {                    //1
   delta=0
   var uprim = for(uu <- util) yield { for(u <- uu) yield u }  //2

   for (i <- (0 until util.length)) {                         //3
      for (j <- (0 until util(0).length)) {
         if (util(i)(j).enabled && !util(i)(j).goal) {        //4
            uprim(i)(j) = State(bellman(util(i)(j), util),    //5
                                util(i)(j).coordinates,
                                util(i)(j).enabled,
                                util(i)(j).goal)
         }

         if (abs(uprim(i)(j).utility - util(i)(j).utility) > delta) { //6
            delta = abs(uprim(i)(j).utility - util(i)(j).utility)
         }
      }
   }
   util = uprim
}
```

Figur 10. Value Iteration

Jag har valt att annotera områden i koden som jag vill diskutera mera med siffror i kommentarerna, t.ex. `//1`.

Implementationen i figur 10 följer algoritmen i figur 3:s design så noggrant som möjligt i Scala.

1. Här finns första förändringen till algoritmen. Det enda denna förändring gör är att vända på *do {} until*-konstruktionen till en *while {}* eftersom Scala inte stöder *until*-konstruktioner. Skillnaden är att predikatet måste vändas om från $\delta < \varepsilon(1-\gamma)/\gamma$ till $\delta \geq \varepsilon(1-\gamma)/\gamma$. Eftersom γ kan vara 1, vilket innebär att $\varepsilon(1-\gamma)/\gamma = 0$ går det tyvärr inte att använda $\delta \geq \varepsilon(1-\gamma)/\gamma$ utan det måste modifieras till $\delta > \varepsilon(1-\gamma)/\gamma$.

2. Scala-räckor är, som de Java-räckor de är baserade på, by-reference-baserade, så för att kopiera en flerdimensionell räcka itererar jag genom den och kopierar varje värde. Det finns även en konstruktion som heter *reflection* i Scala som kan åstadkomma detta utan iterering, men den är onödigt avancerad för detta ändamål [25]. De for each-slingor med *yield* som används här är mera idiomatisk Scala än de slingor som används i annotation 3. For each-slingor med yield kallas även för *for-comprehensions*.

3. Dessa två slingor motsvarar "för varje tillstånd s i S" i Markov-beslutsprocessen. Det behövs två slingor av denna stil eftersom världen i detta exempel är tvådimensionell. Slingor av denna typ rekommenderas oftast inte i Scala eftersom risken för fel är stor och det finns bättre sätt att göra detta. Ett alternativt sätt att åstadkomma detta resultat är med en *for-comprehension* av typen: `for (rows <- util; state <- util)` eller med en `map(map())`.

4. Anmärkningsvärt med annotation fyra är att Scala adresserar element i en räcka med parenteser (x) istället för hakparenteser [x]. Detta är en konsekvens av Scalas val att vara renodlat objekt-orienterat, eftersom datastrukturen Array också är en klass och metodanrop utan namn omvandlas till anrop på metoden `apply` d.v.s. `Array(2)` omvandlas till `Array.apply(2)`. Detta är sant för alla klasser i Scala så länge en apply-metod är definierad i klassen.

5. Detta är själva Bellman-uppdateringen. Räckan U innehåller i detta fall objekt av datatypen *State* (tillstånd), vars struktur beskrivs i bilaga 2. För att kalkylera nyttan av ett tillstånd anropas funktionen `bellman`. Denna funktion beskrivs i figur 11.

6. Algoritmens konvergens kontrolleras i annotation 6, d.v.s. den sista delen av inre-slingan i figur 3.

```
def bellman(s: State, u: Array[Array[State]]): Double = {
   var res = new Array[(Action, Double)](actions.length)
   for (i <- (0 until actions.length)) {
      val transitions = t(s, actions(i))                                    //1
      val nonNormalizedUtilities = new Array[Double](transitions.length)
      for (j <- (0 until transitions.length)) {
         nonNormalizedUtilities(j) = (
            transitions(j).probability *
            u(transitions(j).coordinates._1)(transitions(j).coordinates._2).utility
         )
      }
      res(i) = (actions(i), nonNormalizedUtilities.sum)                     //2
   }

   val (direction, maxUtility) = res.maxBy(_._2)                            //3
   r+(gamma*maxUtility)                                                     //4

}
```

Figur 11.

1. En övergångsfunktion definierar möjliga övergångar för ett givet tillstånd och en given handling. Denna funktion är beroende av tillståndsrymden och den beskrivs i bilaga 2. Funktionen tar som ingångsvärden ett tillstånd och en handling och returnerar en lista med möjliga resulterande tillstånd och deras sannolikhet av datatypen `PossibleState`. Dessa möjliga tillstånd itereras sedan genom och multipliceras med nyttan för respektive tillstånd för att sedan returnera en lista med reella tal (nyttan av olika handlingar). Detta beskrivs i Bellman-ekvationen av $T(s,a,s')U(s')$.

2. Ur listan av potentiella nyttovärden för handling *a* skapas en tupel med handlingen och summan av dess potentiella nyttovärden. Detta beskrivs med $\sum_{s'} T(s,a,s')U(s')$.

3. Ur den resulterande listan av handlingar och deras nyttovärden väljs det värde som maximerar nyttan med `maxBy` metoden som tar en funktion för att extrahera det som skall maximeras (i detta fall är det maximering enligt det andra värdet i tupeln) och sparar dessa i variablerna `direction` och `maxUtility`. Detta motsvarar $\max_a \sum_{s'} T(s,a,s')U(s')$.

4. Slutligen multipliceras värdet med γ och till det adderas belöningsfunktionen *R[s]* och returneras.

4.3 Funktionell version

För att bygga en funktionell implementation av Markov-beslutsprocessen behövdes det en rekursiv version av den imperativa while {}-slingan. Denna implementation visas i figur 12.

```
def recursiveValueIteration(util: Array[Array[State]]): Array[Array[State]] = {
   val newUtil = util.map(_.map { u =>                                          //1
      if (u.enabled && !u.goal)
         State(bellman(u, util), u.coordinates, u.enabled, u.goal)              //2
      else
         u
   })

   val flatUtil = util.flatten
   val delta = newUtil.flatten.zip(flatUtil).foldLeft(0.0)((a, b) =>
                  max(a, abs(b._1.utility - b._2.utility))                      //3
               )

   if ((delta > epsilon*(1-gamma)/gamma)) recursiveValueIteration(newUtil)      //4
   else newUtil
}
```

Figur 12

1. Denna version itererar, som den imperativa, genom alla tillstånd. I detta fall behöver inte någon räcka initieras eftersom map-operationen returnerar en räcka av samma längd som den räcka som operationen görs på (men inte nödvändigtvis en räcka av samma datatyp).

2. Samma sak som i imperativa versionens punkt 5.

3. Man ser snabbt att uppdateringen av deltavärdet (δ) inne i den imperativa varianten egentligen söker efter maximala skillnaden mellan den förra iterationen och den nuvarande. Denna kod åstadkommer exakt samma resultat genom att jämföra skillnaden mellan nyttan av varje uppdaterat tillstånd med dess tidigare värde och därifrån plocka ut den som maximerar m.h.a. en fold-operation. En observant läsare märker att denna denna del av lösningen är O(3n), till skillnad från den imperativa versionen där denna operation utförs i samband med andra operationer och är O(1).

4. Detta är rekursionssteget i algoritmen, här används samma kriterium för konvergens som i while-slingan i den imperativa varianten av algoritmen.

```
def bellman(s: State, u: Array[Array[State]]): Double = {
   val (direction, maxUtility) = actions.map(a =>
      (a.direction, t(s, a).map(p =>                                    //1
         (p.probability*u(p.coordinates._1)(p.coordinates._2).utility)
      ).sum)                                                            //2
   ).maxBy(_._2)                                                        //3

   r+(gamma*maxUtility)                                                 //4
}
```

Figur 13

Funktionen för Bellman-ekvationen, som beskrivs i figur 13, fungerar på exakt samma sätt som sin imperativa variant och annotationerna är placerade på samma ställen som i den imperativa versionen. Därför går det att direkt jämföra och byta ut kodelement.

1. Här används map-operationen över de olika handlingarna och sedan över de olika möjliga resultaten av dessa handlingar. Om bara de två map-operationerna utfördes skulle en tvådimensionell lista med handlingar och möjliga tillstånd i formen $\forall a \in A \bullet \forall s \in (S \mid T(s,a,s')) \bullet T(s,a,s')U(s')$ returneras. Det görs dock några handlingar till inne kodblocken mellan dessa element som ändrar på resultatet.

2. Här plockas summan av potentiella nyttovärden för handling a ut och sedan skapas en tupel med handlingen och summan av dess potentiella nyttovärden. Detta beskrivs med $\sum_{s'} T(s,a,s')U(s')$.

3. Ur den resulterande listan av handlingar och deras nyttovärden väljs det värde som maximerar nyttan med `maxBy` metoden som tar en funktion för att extrahera det som skall maximeras (i detta fall är det maximering enligt det andra värdet i tupeln) och sparar dessa i variablerna `direction` och `maxUtility`. Detta motsvarar $\max_a \sum_{s'} T(s,a,s')U(s')$.

4. Slutligen multipliceras värdet med γ och till det adderas belöningsfunktionen $R[s]$ och returneras.

5 Slutsats

Mitt mål med denna undersökning var att ta reda på huruvida programmeringsspråket Scala lämpar sig för modern artificiell intelligens-programmering. Jag valde att använda Markov-beslutsprocesser för att undersöka detta.

Att evaluera ett programmeringsspråks kvalitet för artificiell intelligens överlag är en svår uppgift, eftersom artificiell intelligens är ett mycket brett ämne som tillämpas på allt från att klassificera galaxer till att styra tvättmaskiner. I ett så brett ämne är det naturligt att behoven är väldigt olika, så min implementation av Markov-beslutsprocesser är inte ens nära på representativ för hela ämnet. En stor del av algoritmerna inom modern artificiell intelligens är dock, som Markov-beslutsprocesser, probabilistiska i sin natur, t.ex. olika typer av partikelfilter, Bayes nätverk, linjär regression eller artificiella neurala nätverk, men även här är algoritmernas användningsområden så pass breda att behoven för användarna av algoritmerna är rätt så olika.

Jag skulle, med viss reservation, säga att Scala är ett bra programmeringsspråk för den del av artificiell intelligens som jag har undersökt närmare. Scala har många fördelar, men likaså en del nackdelar.

Bland de fördelar som Scala har vill jag främst nämna möjligheterna att uttrycka komplext beteende som konstruktion av många mindre komponenter i olika former med hjälp av *traits*, situationsklasser och Scalas mycket sofistikerade objektorientering. Jag föredrar även Scalas syntax över många andra programmeringsspråk eftersom den är rätt så förståelig av människor från olika programmeringsbakgrunder och efter en kort introduktion till Scalas objekt-orientering förstår de flesta även Scala-unika delar av syntaxen. Scala har även fördelar som inte rymts med i denna avhandling, som t.ex. mycket kraftfulla verktyg för att skapa domän-specifika språk och bibliotek för skalbara distribuerade program med hjälp av *actors* och *mjukvaru-transaktionellt minne* (*Software Transactional Memory, STM*).

Som nackdelar kan man notera att Scala har fått en del kritik av sina användare under de senaste månaderna [14][4]. En stor del av denna kritik har att göra med problem, t.ex. versionsfragilitet, som enligt min åsikt orsakas av Scalas ungdom. Versionsfragilitet betyder att kod som är skriven för en tidigare version av Scala inte nödvändigtvis kompilerar på en nyare version utan förändringar. Versionsfragilitet innebär också att kod som är kompilerad i en tidigare version inte nödvändigtvis fungerar i en nyare version utan omkompilering. Detta är ett problem eftersom många bibliotek används som kompilerade binära filer och då måste alla bibliotek vara kompilerade till samma version. En annan kritiker anser att Scalas kompilator är för långsam i jämförelse med andra språk. Jag håller med om dessa båda poänger och det är även något som Odersky själv nyligen talade om [10].

Jag skulle rekommendera Scala för tillämpningar på problem med en stor datamängd med stora beräkningar eftersom JVM är ett ypperligt val här. Likaså har Scala stöd för både avancerade numeriska beräkningar [20], kompilering för beräkningar på grafikkort [18] och ca 10x högre prestanda på parallella miljöer i jämförelse med Java-trådar. [9]. Det finns även färdiga programbibliotek för både maskininlärning [22] och språkteknologi (*Natural Language Processing*) [21] samt ett avancerat domänspecifikt språk för probabilistisk modellering.

Jag har dock svårt att rekommendera Scala för tillämpningar där Java Virtual Machine inte är lämplig för ändamålet. I många miljöer är JVM en ypperlig motor men det finns fortfarande situationer där JVM helt enkelt inte passar. En av de situationer där jag skulle ha svårt att rekommendera Scala är i *inbyggda system*, speciellt miljöer där hårdvaran är mycket begränsad, just eftersom JVM är rätt så tung. Scala är inte heller designat för inbyggda system och standardbiblioteket är rätt så begränsat på det området.

Som vidare forskning inom ämnet skulle det vara intressant med konkreta tillämpningar av Markov-beslutsprocesser och Scala på verkliga problem, speciellt inom robotik, eftersom JVM sällan används i sådana miljöer. Personligen skulle jag vara mest intresserad av Scalas tillämpningar på större reaktiva system för databehandling inom maskininlärning.

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] Dürig, M. Scala type level encoding of the SKI calculus. http://michid.wordpress.com/2010/01/29/scala-type-level-encoding-of-the-ski-calculus/. Hämtad 25.3.2012

[3] 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

[4] Kreps, J. Scala Macros: "Oh God Why?". http://blog.empathybox.com/post/19126121307/scala-macros-oh-god-why. Hämtad 25.3.2012

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

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

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

[8] McCarthy, J. What is artificial intelligence. http://www-formal.stanford.edu/jmc/whatisai/node1.html. Hämtad 4.3.2012

[9] Nordwall, P. Scalability of the Fork Join Pool. http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool. Hämtad 25.3.2012

[10] Odersky, M. Scala - a Roadmap. *Scala-language mailinglist*. https://groups.google.com/group/scala-language/browse_thread/thread/3d5e2ae8ed6a221f. Hämtad 25.3.2012

[11] Odersky, M. Research - Martin Odersky Home page. http://lampwww.epfl.ch/~odersky/. Hämtad 25.3.2012

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

[13] 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

[14] Pollak, D. Scala's version fragility make the Enterprise argument near impossible. http://blog.goodstuff.im/scala-s-version-fragility-make-the-enterprise-argument-near-impossible/. Hämtad 25.3.2012

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

[16] 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

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

[18] ScalaCL. http://code.google.com/p/scalacl/. Hämtad 25.3.2012

[19] Scala Improvement Process. http://docs.scala-lang.org/sips/index.html. Hämtad 25.3.2012

[20] Scalala Repository. https://github.com/scalala/Scalala. Hämtad 25.3.2012

[21] ScalaNLP-biblioteket. http://www.scalanlp.org/. Hämtad 25.3.2012

[22] Scala-recog-biblioteket. http://code.google.com/p/scala-recog/. hämtad 25.3.2012

[23] Scala Repository, The. https://github.com/scala/scala. Hämtad 25.3.2012

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

[25] Stackoverflow - What is the easiest way to deeply clone (copy) a mutable Scala object?. http://stackoverflow.com/a/1270941/1257630. Hämtad 25.3.2012

[26] TIOBE Programming Community Index for March 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Hämtad 25.3.2012

[27] Transparent Language Popularity Index, The. http://lang-index.sourceforge.net/. Hämtad 25.3.2012

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

Bilagor

Bilaga 1. Value iteration för Markov-beslutsprocessexemplet

(2,0)	(2,1)	(2,2)	+1
(1,0)		(1,2)	-1
(0,0) START	(0,1)	(0,2)	(0,3)

För att lättare beskriva vilket tillstånd ett uttryck hör ihop med adresseras tillstånden med tiplar. Handlingar kommer att hänvisas till med orden: “Upp”, “Höger”, “Ner” och “Vänster”. Jag kommer att visa lösningarna för de fem första iterationerna på mycket hög detaljnivå (alla unika uträkningar). Från femte iterationen tills algoritmen konvergerar visas bara iterationens resultat. Eftersom γ valdes till värdet 1.0 tar algoritmen längre att konvergera än i fall där faktorn väljs till ett värde lägre än 1, men det förenklar kalkylerna så det passar bra för ett exempelproblem.

Initial nyttofunktion

0	0	0	+1
0		0	-1
0 START	0	0	0

Första iterationen:

-0.04	-0.04	0.76	+1
-0.04		-0.04	-1
-0.04 START	-0.04	-0.04	-0.04

Tillstånd: (2,2)

Handlingar:

Upp: *0.8*0.0+0.1*1.0+0.1*0.0 = 0.1*

Höger: *0.8*1.0+0.1*0.0+0.1*0.0 = 0.8*

Ner: *0.8*0.0+0.1*1.0+0.1*0.0 = 0.1*

Vänster: *0.8*0.0+0.1*0.0+0.1*0.0 = 0.0*

Maximerande Handling: Höger

Nytta: *-0.04+1*0.8 = 0.76*

Tillstånd: (2,1)

Handlingar:

Upp: *0.8*0.0+0.1*0.0+0.1*0.0 = 0.0*

Höger: *0.8*0.0+0.1*0.0+0.1*0.0 = 0.0*

Ner: *0.8*0.0+0.1*0.0+0.1*0.0 = 0.0*

Vänster: *0.8*0.0+0.1*0.0+0.1*0.0 = 0.0*

Maximerande handling: Arbiträrt

Nytta: *-0.04+1*0.0 = -0.04*

Alla resterande kalkyler är identiska till (2,1) under första iterationen.

Andra iterationen:

-0.08	0.56	0.812	+1
-0.08		0.464	-1
-0.08 START	-0.08	-0.08	-0.08

Tillstånd: (2,2)

Handlingar:

Upp: *0.8*0.76+0.1*1.0+0.1*-0.04 = 0.704*

Höger: *0.8*1.0+0.1*0.76+0.1*-0.04 = 0.872*

Ner: *0.8*-0.04+0.1*1.0+0.1*-0.04 = 0.064*

Vänster: *0.8*-0.04+0.1*0.76+0.1*-0.04 = 0.04*

Maximerande Handling: Höger

Nytta: *-0.04+1* 0.872 = 0.812*

Tillstånd: (2,1)

Handlingar:

Upp: *0.8*-0.04+0.1*0.76+0.1*-0.04 = 0.04*

Höger: *0.8*0.76+0.1*-0.04+0.1*-0.04 = 0.6*

Ner: *0.8*-0.04+0.1*0.76+0.1*-0.04 = 0.4*

Vänster: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Maximerande handling: Höger

Nytta: *-0.04+1*0.6 = 0.56*

Tillstånd: (1,2)

Handlingar:

Upp: *0.8*0.76+0.1*-1.0+0.1*-0.04 = 0.504*

Höger: *0.8*-1.0+0.1*0.76+0.1*-0.04 = -0.728*

Ner: *0.8*-0.04+0.1*-1.0+0.1*-0.04 = -0.138*

Vänster: *0.8*-0.04+0.1*0.76+0.1*-0.04 = 0.04*

Maximerande handling: Upp

Nytta: *-0.04+1*0.504 = 0.464*

-0.08	0.56	0.812	+1
-0.08		0.464	-1
-0.08 START	-0.08	-0.08	-0.08

Tillstånd: (0,3)

Handlingar:

Upp: *0.8*-1.0+0.1*-0.04+0.1*-0.04 = -0.808*

Höger: *0.8*-0.04+0.1*-1.0+0.1*-0.04 = -0.136*

Ner: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Vänster: *0.8*-0.04+0.1*-1.0+0.1*-0.04 = -0.136*

Maximerande handling: Ner

Nytta: *-0.04+1* -0.04 = -0.08*

Tillstånd: (2,0)

Handlingar:

Upp: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Höger: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Ner: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Vänster: *0.8*-0.04+0.1*-0.04+0.1*-0.04 = -0.04*

Maximerande handling: Arbiträrt

Nytta: *-0.04+1* -0.04 = -0.08*

Kalkylen för tillstånd (2,0) är identisk för de resterande resterande tillstånden under denna iteration.

Tredje iterationen:

0.392	0.7376	0.8896	+1
-0.12		0.572	-1
-0.12 START	-0.12	0.3152	-0.12

Tillstånd: (2,2)

Handlingar:

Upp: *0.8*0.812+0.1*1.0+0.1*0.56 = 0.8216*

Höger: *0.8*1.0+0.1*0.812+0.1*0.464 = 0.9296*

Ner: *0.8*0.464+0.1*1.0+0.1*0.56 = 0.5272*

Vänster: *0.8*0.56+0.1*0.812+0.1*0.464=0.5776*

Maximerande Handling: Höger

Nytta: *-0.04+1* 0.9296 = 0.8896*

Tillstånd: (2,1)

Handlingar:

Upp: *0.8*0.56+0.1*0.812+0.1*-0.08 = 0.5212*

Höger: *0.8*0.812+0.1*0.56+0.1*0.56 = 0.7776*

Ner: *0.8*0.56+0.1*0.812+0.1*-0.08 = 0.5212*

Vänster: *0.8*-0.08+0.1*0.56+0.1*0.56 = 0.048*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.7776 = 0.7376*

Tillstånd: (2,0)

Handlingar:

Upp: *0.8*-0.08+0.1*0.56+0.1*-0.08 = -0.016*

Höger: 0.8*0.56+0.1*-0.08+0.1*-0.08 = *0.412*

Ner: *0.8*-0.08+0.1*0.56+0.1*-0.08 = 0.016*

Vänster: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.412 = 0.392*

0.392	0.7376	0.8896	+1
-0.12		0.572	-1
-0.12 START	-0.12	0.3152	-0.12

Tillstånd: (1,2)

Handlingar:

Upp: *0.8*0.812+0.1*-1.0+0.1*0.464 = 0.612*

Höger: *0.8*-1.0+0.1*0.812+0.1*-0.08 = -0.7248*

Ner: *0.8*-0.08+0.1*-1.0+0.1*0.464 = -0.1176*

Vänster: *0.8*0.464+0.1*0.812+0.1*-0.08=0.4464*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.612 = 0.572*

Tillstånd: (0,2)

Handlingar:

Upp: *0.8*0.464+0.1*-0.08+0.1*-0.08 = 0.3552*

Höger: *0.8*-0.08+0.1*0.464+0.1*-0.08 =-0.0256*

Ner: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Vänster: *0.8*-0.08+0.1*0.464+0.1*-0.08=-0.0256*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.3552 = 0.3152*

Tillstånd: (1,0)

Handlingar:

Upp: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Höger: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Ner: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Vänster: *0.8*-0.08+0.1*-0.08+0.1*-0.08 = -0.08*

Maximerande handling: Arbiträrt

Nytta: *-0.04+1* -0.08 = -0.12*

Kalkylen för tillstånd (1,0) är identisk för de resterande resterande tillstånden under denna iteration.

Fjärde iterationen:

0.5778	0.8192	0.9062	+1
0.2496		0.6289	-1
-0.16 START	-0.16	0.3936	0.1002

Tillstånd: (2,2)

Handlingar:

Upp:*0.8*0.8896+0.1*1.0+0.1*0.7376 = 0.88544*

Höger: *0.8*1.0+0.1*0.8896+0.1*0.572 = 0.94616*

Ner: *0.8*0.572+0.1*1.0+0.1*0.7376 = 0.63136*

Vänster:*0.8*0.7376+0.1*0.8896+0.1*0.572=0.7362*

Maximerande Handling: Höger

Nytta: *-0.04+1* 0.94616 = 0.9062*

Tillstånd: (2,1)

Handlingar:

Upp: *0.8*0.7376+0.1*0.8896+0.1*0.392 = 0.71824*

Höger:*0.8*0.8896+0.1*0.7376+0.1*0.7376=0.8592*

Ner: *0.8*0.7376+0.1*0.8896+0.1*0.392 = 0.71824*

Vänster:*0.8*0.392+0.1*0.7376+0.1*0.7376=0.4611*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.8592 = 0.8192*

Tillstånd: (2,0)

Handlingar:

Upp: *0.8*0.392+0.1*0.7376+0.1*0.392 = 0.42656*

Höger: 0.8*0.7376+0.1*0.392+0.1*-0.12 = *0.61728*

Ner: *0.8*-0.12+0.1*0.7376+0.1*0.392 = 0.01696*

Vänster: *0.8*0.392+0.1*0.392+0.1*-0.12 = 0.3408*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.61728 = 0.5778*

0.5778	0.8192	0.9062	+1
0.2496		0.6289	-1
-0.16 START	-0.16	0.3936	0.1002

Tillstånd: (1,2)

Handlingar:

Upp: *0.8*0.8896+0.1*-1.0+0.1*0.572 = 0.66888*

Höger: *0.8*-1.0+0.1*0.8896+0.1*0.3152=-0.67952*

Ner: *0.8*0.3152+0.1*-1.0+0.1*0.572+=0.20936*

Vänster: *0.8*0.572+0.1*0.8896+0.1*0.3152=0.578*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.66888 = 0.6289*

Tillstånd: (1,0)

Handlingar:

Upp: *0.8*0.392+0.1*-0.12+0.1*-0.12 = 0.2896*

Höger: *0.8*-0.12+0.1*0.392+0.1*-0.12 = -0.0688*

Ner: *0.8*-0.12+0.1*-0.12+0.1*-0.12 = -0.12*

Vänster: *0.8*-0.12+0.1*0.392+0.1*-0.12 = -0.0688*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.2896 = 0.2496*

Tillstånd: (0,3)

Handlingar:

Upp: *0.8*-1.0+0.1*-0.12+0.1*0.3152 = -0.78048*

Höger: *0.8*-0.12+0.1*-1.0+0.1*-0.12 = -0.208*

Ner: *0.8*-0.12+0.1*-0.12+0.1*0.3152 = -0.07648*

Vänster: *0.8*0.3152+0.1*-1.0+0.1*-0.12 = 0.14016*

Maximerande handling: Vänster

Nytta: *-0.04+1* 0.14016 = 0.1002*

0.5773	0.8192	0.9062	+1
0.2496		0.6289	-1
-0.16 START	-0.16	0.3936	0.1002

Tillstånd: (0,2)

Handlingar:

Upp: *0.8*0.572+0.1*-0.12+0.1*-0.12 = 0.4336*

Höger: *0.8*-0.12+0.1*0.572+0.1*0.3152 =-0.00728*

Ner: *0.8*0.3152+0.1*-0.12+0.1*-0.12 = 0.22816*

Vänster: *0.8*-0.12+0.1*0.572+0.1*0.3152=-0.0073*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.4336 = 0.3936*

Tillstånd: (0,1)

Handlingar:

Upp: *0.8*-0.12+0.1*-0.12+0.1*-0.12 = -0.12*

Höger: *0.8*-0.12+0.1*-0.12+0.1*-0.12 = -0.12*

Ner: *0.8*-0.12+0.1*-0.12+0.1*-0.12 = -0.12*

Vänster: *0.8*-0.12+0.1*-0.12+0.1*-0.12 = -0.12*

Maximerande handling: Arbiträrt

Nytta: *-0.04+1* -0.12 = -0.16*

Kalkylerna för tillstånd (0,1) och tillstånd (0,0) är identiska i denna iteration.

Femte iterationen

0.6981	0.8488	0.9135	+1
0.4717		0.6478	-1
0.1625 START	0.3125	0.4919	0.1849

Tillstånd: (2,2)

Handlingar:

Upp: *0.8*0.90616+0.1*1.0+0.1*0.8192 = 0.906848*

Höger: *0.8*1.0+0.1*0.9061+0.1*0.62888 = 0.953504*

Ner: *0.8*0.62888+0.1*1.0+0.1*0.819 = 0.685024*

Vänster:*0.8*0.8192+0.1*0.9062+0.1*0.62888=0.8089*

Maximerande Handling: Höger

Nytta: *-0.04+1* 0.953504 = 0.9135*

Tillstånd: (2,1)

Handlingar:

Upp: *0.8*0.8192+0.1*0.9062+0.1*0.5773 = 0.803704*

Höger: *0.8*0.9062+0.1*0.8192+0.1*0.8192=0.888768*

Ner: *0.8*0.8192+0.1*0.90616+0.1*0.5773 = 0.803704*

Vänster:*0.8*0.5773+0.1*0.8192+0.1*0.8192=0.62566*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.888768 = 0.8488*

Tillstånd: (2,0)

Handlingar:

Upp: *0.8*0.5773+0.1*0.8192+0.1*0.5773 = 0.601472*

Höger: 0.8*0.8192+0.1*0.5773+0.1*0.2496 =0.738048

Ner: *0.8*0.2496+0.1*0.8192+0.1*0.57728=0.339128*

Vänster: *0.8*0.5773+0.1*0.5773+0.1*0.2496=0.54451*

Maximerande handling: Höger

Nytta: *-0.04+1** 0.738048 *= 0.6981*

0.6981	0.8488	0.9135	+1
0.4717		0.6478	-1
0.1625 START	0.3125	0.4919	0.1849

Tillstånd: (1,2)

Handlingar:

Upp: *0.8*0.90616+0.1*-1.0+0.1*0.62888 = 0.687816*

Höger: *0.8*-1.0+0.1*0.90616+0.1*0.3936=-0.670024*

Ner: *0.8*0.3936+0.1*-1.0+0.1*0.62888 = 0.277768*

Vänster: *0.8*0.6289+0.1*0.9062+0.1*0.3936=0.63308*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.687816 = 0.6478*

Tillstånd: (1,0)

Handlingar:

Upp: *0.8*0.57728+0.1*0.2496+0.1*0.2496=0.511744*

Höger: *0.8*0.2496+0.1*0.57728+0.1*-0.16 =0.241408*

Ner: *0.8*-0.16+0.1*0.2496+0.1*0.2496 = -0.07808*

Vänster: *0.8*0.2496+0.1*0.57728+0.1*-0.16=0.2414*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.511744 = 0.4717*

Tillstånd: (0,3)

Handlingar:

Upp: *0.8*-1.0+0.1*0.10016+0.1*0.3936 = -0.750624*

Höger: *0.8*0.10016+0.1*-1.0+0.1*0.10016=-0.00986*

Ner: *0.8*0.10016+0.1*0.10016+0.1*0.3936=0.129504*

Vänster: *0.8*0.3936+0.1*-1.0+0.1*0.10016=0.224896*

Maximerande handling: Vänster

Nytta: *-0.04+1* 0.224896 = 0.1849*

0.6981	0.8488	0.9135	+1
0.4717		0.6478	-1
0.1625 START	0.3125	0.4919	0.1849

Tillstånd: (0,2)

Handlingar:

Upp:*0.8*0.62888+0.1*0.10016+0.1*0.18816=0.53194*

Höger: *0.8*0.10016+0.1*0.6289+0.1*0.3936=0.18238*

Ner: *0.8*0.3936+0.1*0.10016+0.1*0.18816=0.343712*

Vänster:*0.8*0.1882+0.1*0.6289+0.1*0.3936=0.25278*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.53194 = 0.4919*

Tillstånd: (0,1)

Handlingar:

Upp: *0.8*0.18816+0.1*0.3936+0.1*-0.16 = 0.173888*

Höger: *0.8*0.3936+0.1*0.18816+0.1*0.18816=0.3525*

Ner: *0.8*0.18816+0.1*0.3936+0.1*-0.16 = 0.173888*

Vänster:*0.8*-0.16+0.1*0.1881+0.1*0.1881=-0.090368*

Maximerande handling: Höger

Nytta: *-0.04+1* 0.3525 = 0.3125*

Tillstånd: (0,0)

Handlingar:

Upp: *0.8*0.2496+0.1*0.18816+0.1*-0.16 = 0.202496*

Höger: *0.8*0.18816+0.1*0.2496+0.1*-0.16 =0.159488*

Ner: *0.8*-0.16+0.1*0.18816+0.1*-0.16 = -0.125184*

Vänster: *0.8*-0.16+0.1*0.2496+0.1*-0.16 = -0.11904*

Maximerande handling: Upp

Nytta: *-0.04+1* 0.202496 = 0.1625*

0.756	0.8605	0.9161	+1
0.6128		0.6556	-1
0.3849 START	0.416	0.528	0.272

0.7853	0.8650	0.9171	+1
0.6874		0.6585	-1
0.5303 START	0.4656	0.5533	0.3096

Sjätte iterationen

Sjunde iterationen

0.7993	0.8667	0.9176	+1
0.7257		0.6596	-1
0.6095 START	0.4957	0.5643	0.3336

0.8059	0.8674	0.9177	+1
0.7257		0.6596	-1
0.6511 START	0.5467	0.5706	0.3448

Åttonde iterationen

Nionde iterationen

0.809	0.8677	0.9178	+1
0.7536		0.6602	-1
0.6754 START	0.5902	0.5772	0.351

0.8104	0.8678	0.9178	+1
0.7579		0.6602	-1
0.6895 START	0.6184	0.5823	0.3568

Tionde iterationen

Elfte iterationen

0.811	0.8678	0.9178	+1
0.7599		0.6603	-1
0.6971 START	0.6353	0.5857	0.3615

0.811	0.8678	0.9178	+1
0.7608		0.6603	-1
0.7011 START	0.6447	0.5928	0.3647

12:e iterationen

13:e iterationen

0.812	0.8678	0.9178	+1
0.7612		0.6603	-1
0.7012 START	0.6499	0.6011	0.3707

0.812	0.8678	0.9178	+1
0.7614		0.6603	-1
0.7043 START	0.6526	0.6060	0.3779

14:e iterationen

15:e iterationen

0.812	0.8678	0.9178	+1
0.7614		0.6603	-1
0.7048 START	0.6539	0.6087	0.3826

0.812	0.8678	0.9178	+1
0.7615		0.6603	-1
0.7051 START	0.6546	0.61	0.3852

16:e iterationen

17:e iterationen

0.812	0.8678	0.9178	+1
0.7615		0.6603	-1
0.7052 START	0.655	0.611	0.3866

0.812	0.8678	0.9178	+1
0.7616		0.6603	-1
0.7053 START	0.6552	0.611	0.3872

18:e iterationen

19:e iterationen

0.812	0.8678	0.9178	+1
0.7616		0.6603	-1
0.7053 START	0.6552	0.611	0.3876

0.812	0.8678	0.9178	+1
0.7616		0.6603	-1
0.7053 START	0.6553	0.611	0.3877

20:e iterationen

21:e iterationen

0.812	0.8678	0.9178	+1
0.7616		0.6603	-1
0.7053 START	0.6553	0.611	0.3878

0.812	0.8678	0.9178	+1
0.7616		0.6603	-1
0.7053 START	0.6553	0.611	0.3878

22:e iterationen

23:e iterationen

Bilaga 2. Arkitekturella stödklasser för Markov-beslutsprocesser i Scala

I denna bilaga finns all kod som används i denna avhandling. Bilaga 3 innehåller instruktioner för hur läsaren lättast kan komma åt och själv testa programmet. Jag har även lagt med en kort beskrivning av koden ovanför varje block.

Nedan visas klassen som kör själva Markov-beslutsprocessen. I MDP-klassen mixas det in tre stycken *traits* som är utbytbara så länge de erbjuder de variabler som behövs. Här deklareras också diverse mål och konfigurationsvärden. Denna klass används i båda varianterna av Markov-beslutsproccessen utan några större förändringar, `ImperativeValueIteration` byts ut mot `FunctionalValueIteration`.

```
class MDP extends TwoDimensionalWorld
              with ImperativeValueIteration
              with TwoDimensionalTransition {

   val startState = (0, 0)
   val goalStates = List((1, 3), (2, 3))
   val r = -0.04
   val gamma = 0.8

   val valIter = valueIteration(u, t, r, gamma)

   private def prettify(u: Array[Array[State]]) = {
      (u.map(_.map(_.utility.toString) mkString ", ") mkString "\n")
   }

   override def toString = prettify(valIter)
}
object MDP {
   def main(args: Array[String]): Unit = {
     val mdp = new MDP
     println(mdp)
   }
}
```

Denna klass beskriver världen som Markov-beslutsprocessen agerar i. Beskrivningen av världen är beroende av dimensionaliteten, men alla beskrivningar av världen i min implementation kommer att vara räckor med tillstånd.

```
class TwoDimensionalWorld {
   val u = Array(
      Array(State(0.0, (0,0)), State(0.0, (0,1)),
            State(0.0, (0,2)), State(0.0, (0,3))),
      Array(State(0.0, (1,0)), State(0.0, (1,1), false),
            State(0.0, (1,2)), State(-1.0, (1,3), true, true)),
      Array(State(0.0, (2,0)), State(0.0, (2,1)),
            State(0.0, (2,2)), State(1.0, (2,3), true, true))
      )
}
```

Övergångsfunktionen som används för denna implementation är definierad i ett *trait* som kallas `TwoDimensionalTransition`. Traitet går att byta ut mot vilket annat trait som helst så länge det definierar en funktion `t` som tar som input ett tillstånd och en handling och avbildas på en lista med möjliga tillstånd (`PossibleState`). Denna implementation använder sig av Scalas mönsteranpassning för att identifiera handlingar och en stödfunktion som heter `move`.

```
trait TwoDimensionalTransition extends TwoDimensionalWorld {
   /**
    * A slightly modified version of the T(s,a,s') function, instead of just
    * returning a probability a given action this function returns the
    * probabilty of all actions for a given state. Based on the implementations given
    * for Artificial Intelligence, A modern approach.
    *
    * @input s, a State where the action is taken
    * @input a, an Action applied in State s
    *
    * @returns List[PossibleState],
    *    a list of PossibleStates, which are a decorated
    *    form of State that has a associated probability.
    **/
   def t(s: State, a: Action): List[PossibleState] = {
      a.direction match {
         case "up" => {
            List(
                  PossibleState(0.8,
                        move((s.coordinates),(s.coordinates._1+1, s.coordinates._2))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2+1))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2-1)))
               )
         }
         case "right" => {
            List(
                  PossibleState(0.8,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2+1))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1+1, s.coordinates._2))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1-1, s.coordinates._2)))
               )
         }
         case "down" => {
            List(
                  PossibleState(0.8,
                        move((s.coordinates), (s.coordinates._1-1, s.coordinates._2))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2+1))),
                  PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2-1)))
               )
         }
         case "left" => {
            List(
               PossibleState(0.8,
                        move((s.coordinates), (s.coordinates._1, s.coordinates._2-1))),
               PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1+1, s.coordinates._2))),
               PossibleState(0.1,
                        move((s.coordinates), (s.coordinates._1-1, s.coordinates._2)))
               )
         }
         case _ => throw new Exception(
                  "I'm afraid I can't let you do that Dave. (unknown direction)"
               )
      }
   }
```

Detta är move-funktionen för den tvådimensionella övergångsfunktionen i tidigare listning. Move-funktionen ser till att en rörelsen är möjlig och hanterar potentiella omöjliga rörelser. I detta fall utgör en rörelse som resulterar i att man skulle hamna i ett tillstånd som är "utanför" världen i att ingen rörelse sker. Denna funktion fungerar bara med den specifika övergångsfunktion som den stöder, t.ex. skulle denna funktion inte vara tillräcklig för en övergångsfunktion som tillåter diagonala övergångar.

```
def move(start: (Int, Int), dest: (Int, Int)): (Int, Int) = {
   val y = if ((dest._1 < 0) || (dest._1 >= u.length))
              start._1
           else
              dest._1
   val x = if ((dest._2 < 0) || (dest._2 >= u(y).length))
              start._2
           else
              dest._2
   if (u(y)(x).enabled) (y, x) else start
}
```

Följande fyra klasser är stödklasser som används för att skapa de olika datatyper som används i implementationerna av Markov-beslutsprocessen.

```
case class State(utility: Double, coordinates: (Int, Int),
                 enabled: Boolean = true, goal: Boolean = false)
case class PossibleState(probability: Double, coordinates: (Int, Int))
case class Action(direction: String)
```

Bilaga 3. Testningsinstruktioner för kodexemplen

Jag har laddat upp alla kodexempel på GitHub för att underlätta experimentering för användaren. För att vidare underlätta har jag använt ett verktyg som heter Simple Build Tool (SBT). Meningen med SBT är att erbjuda inkrementell kompilering och identifiering av körbar kod, SBT ser också till att Scala-miljön är rätt konfigurerad (att rätt version av Scala körs). SBT är ett öppet gratisverktyg skapat av Mark Harrah vars källkod finns att hitta på GitHub [https://github.com/harrah/xsbt] och installationsinstruktioner för olika plattformar här: https://github.com/harrah/xsbt/wiki/Getting-Started-Setup.

Arbetssättet för att lägga igång den miljö jag har använt och köra mina exempel på en unix-miljö är:

1. Installera SBT (kräver Java), enligt instruktionerna på https://github.com/harrah/xsbt/wiki/Getting-Started-Setup

2. Ladda ner en version av Markov-beslutsprocessen genom att antingen använda kommandot `git clone git://github.com/KarlHerler/Markov-decision-process.git` eller genom att ladda ner koden som en zipfil från: https://github.com/KarlHerler/Markov-decision-process/zipball/master och packa upp den.

3. Kör applikationen genom att skriva `sbt run` i terminalfönstret i den mapp som källkoden finns i. Första gången kan detta ta en stund efter som rätt version av Scala och SBT installeras.