Implementation av Q-learning för fyra-i-rad

EDVIN EKBLAD
och OSKAR WERKELIN AHLIN

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Implementation av Q-learning för fyra-i-rad

E D V I N E K B L A D
o c h O S K A R W E R K E L I N A H L I N

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Johan Boye
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/ekblad_edvin_OCH_werkelin_ahlin_oskar_K10055.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Referat

Maskininlärning är ett forskningsområde inom datalogi som blivit av större vikt de senaste åren. Det används i många vardagliga sammanhang, såsom röstigenkänning i mobiltelefoner, ansiktsigenkänning i bildbehandlingssammanhang och mycket mer. Ett delområde inom området är belöningsbaserad inlärning, som är ett försök till att efterlikna intelligens i den verkliga världen, det bygger på den enkla idén att om man tidigare tjänat på att göra något bör man göra det igen. Q-learning är en algoritm som bygger på denna teknik, och i detta projekt implementerar vi den för det enkla brädspelet fyra-i-rad. Spelets tillståndsmängd visar sig vara för stor för att uppnå bra allmänna resultat, men hållbara resultat fås ändå genom att ett problem formuleras som minskar tillståndsmängden kraftigt.

Abstract

Implementation of Q-learning for Connect Four

Machine learning is a field of research in computer science which has gained importance lately. It is used in several everyday products, for features such as voice recognition in cell phones and face recognition in photography equipment. A subtopic of machine learning is reinforcement learning, which is an attempt att mimicking real-life intelligence. Reinforcement learning is built around the idea that if a positive reaction is received from a certain action, this action should be repeated. Q-learning is an algorithm which is built upon this technique, and it is implemented in this project for the board game Connect Four. The amount of individual states of the game proves to be too large to acquire good general results, however acceptable results are still produced by the problem being formulated in a way which greatly reduces the amount of states.

Innehåll

Kapitel 1

Förord

Arbetet har mestadels genomförts sida vid sida. Stundom har dock arbetet skett på separat plats, men då i samverkan via versionshanteringssystem som underlättat samarbetet. Att se resultatet av vår gemensamma möda har varit både givande och lärorikt. Charmen i att spela fyra-i-rad mot ett program som är självupplärt är lik glädjen i fostrandet av en ung individ.

Kapitel 2

Introduktion

Rapporten är skriven som kandidatexamensarbete vid skolan för datavetenskap- och kommunikation (CSC) vid KTH i Stockholm. Den undersöker inlärningsalgoritmen Q-learning i en implementation för brädspelet fyra-i-rad. Efter att en implementation av algoritmen gjorts i en så kallad agent, studeras här hur effektiv inlärningen är då agenten ställs emot olika typer av spelare. Resultaten av denna studie diskuteras i den senare delen av rapporten.

Rapporten förväntas förmedla kunskap om situationer där Q-learning implementeras för brädspel såsom fyra-i-rad, samt om vilka resultat som kan förväntas.

2.1 Bakgrund

2.1.1 Maskininlärning

Maskininlärning utvecklades som en ansats till att försöka få datorprogram att förbättra sig själva för att lösa ett visst problem eller nå ett visst mål på ett bättre eller mer effektivt vis. Det tillåter till exempel en dator att lära sig spela ett spel genom att öva och bli bättre, istället för att använda en algoritm eller metod som säger precis hur den ska agera givet förutsättningarna. I vissa situationer kan det exempelvis vara svårt att instruera en dator precis hur den ska agera då det kan bero på omständigheterna. Då lönar det sig att sätta upp mer generella mål, och låta datorn själv lära sig hur den ska agera för att nå dessa mål.

Ett område inom maskininlärning är belöningsbaserad inlärning, vilket beskrivs i nästa avsnitt.

2.1.2 Belöningsbaserad inlärning

Belöningsbaserad inlärning grundar sig till viss del i psykologin, och då speciellt dess teorier om trial-and-error. Tanken är att en agent (djur/program/organism) kan lära sig ett visst beteende genom att upplevelsen av handlingar förstärks ne-

gativt eller positivt, i minnet, tills agenten agerar på önskat vis i sin miljö. Denna psykologiska aspekt kombinerad med dynamisk programmering - möjligheten att lösa ett stort problem genom att dela upp det i flera små problem - är grunden till flera av dagens belöningsbaserade inlärningsalgoritmer[3].

Inom datalogin fungerar det på samma sätt. En agent tar lärdom av en situation genom att titta på om sitationen tidigare gynnat eller missgynnat agenten. Programmet som kör agenten ansvarar för att dela ut belöningar och bestraffningar i rätt tillfällen.

2.2 Begrepp

Här introduceras begrepp som ej antas vara självklara för läsaren.

2.2.1 Agent

När begreppet 'agent' används i den här rapporten menas den datorstyrda spelare som implementerats med algoritmen Q-learning för att spela fyra-i-rad.

2.2.2 Q-tabell

Datastrukturen som Q-learning använder sig av.

2.2.3 Q-värde

Med begreppet 'Q-värde' menas ett värde i Q-tabellen.

2.3 Teori

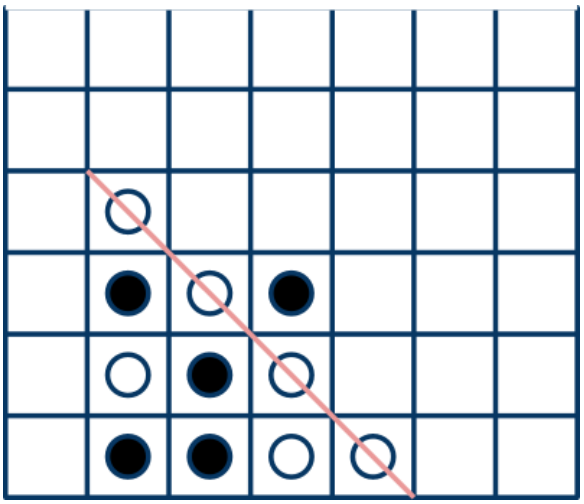
2.3.1 Fyra-i-rad

Fyra-i-rad är ett brädspel som spelas av två spelare. De två spelarna besitter brickor i en färg skild från motspelarens. Spelplanen har sex rader och sju kolumner vari spelarna turas om att placera brickor. Brickorna måste placeras så nära botten som möjligt (brickor får ej placeras i luften).

Spelets avslutning

Den spelare som först lyckas placera sina brickor så att en diagonal, vågrät eller lodrät linje bestående av fyra brickor i spelarens egen färg uppstår vinner. Då har motståndarspelaren förlorat. Ett exempel på detta visas i figur 2.1.

Ett annat scenario då spelet avslutas uppstår när hela spelbrädet är fyllt av brickor, men ingen spelare har vunnit. Då avslutas spelet som oavgjort.



Figur 2.1. Exempel på spel där vit spelare vunnit genom att skapa en diagonal av fyra brickor.

Brickor kan inte placeras i fyllda kolumner, det vill säga en kolumn med sex stycken placerade brickor. Sådana drag kallas ogiltiga drag.

Komplexitet

En spelplan består av 6 rader och 7 kolumner, och varje ruta kan antingen innehålla en bricka som tillhör den första spelaren, en bricka som tillhör den andra spelaren eller ingen bricka alls. Det finns alltså tre möjligheter per ruta, och 42 rutor. Enligt enkla sannolikhetsregler genererar detta en övre gräns med ett värde $3^{42} \approx 10^{20}$ olika tillstånd. Dock uppstår inte alla dessa, exempelvis kan inte planen besättas av endast ena spelarens brickor. Enligt [2] beräknas en mer precis siffra på antalet tillstånd vara $7.1 * 10^{13}$.

Antag att man lagrar ett flyttal av storleken 8 byte. För varje spelbräde måste då lagra samma storlek lagras. Det betyder att vi behöver hålla en datastruktur som är $8 * 7.1 * 10^{13}$B$\approx 5.7 * 10^{14}$B$\approx$ 568 TB stor i minnet. Tillstånden är alltså för många för att alla samtidigt ska kunna lagras i en nutida dator.

2.3.2 Q-learning

Q-learning är en belöningsbaserad algoritm som baserar sig på tillstånd och händelser [1]. För att förstå hur algoritmen implementeras för fyra-i-rad är det viktigt att förstå vad tillstånd respektive händelser representeras av i det konkreta fallet. Ett tillstånd är ett spelbräde med en viss uppställning av brickor. En händelse är skeendet då en bricka placeras någonstans på spelbrädet, ett drag görs helt enkelt.

Algoritmen baserar sin inlärning på att lagra erfarenhet i en datastruktur med

förmågan att spara ett värde för varje kombination av spelbräde och gjort drag. Att ett värde måste lagras för varje tillstånd och händelse som upplevts är en nackdel med Q-learning, som ofta leder till att datastrukturen blir väldigt stor.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{\alpha'} Q(s',a') - Q(s,a)] \tag{2.1}$$

När ett drag ska väljas så går algoritmen igenom alla möjliga drag i nuvarande tillstånd, tittar på de lagrade Q-värdena för dessa, och väljer det drag som genererar det maximala Q-värdet och beräknas sedan om Q-värdet för det föregående tillståndet enligt pseudokoden ovan. Om det maximala Q-värdet är samma för flera tillstånd så slumpar agenten mellan dessa. Koefficienten α betecknar inlärningshastighet, r betecknar belöning, och γ är avdragsfaktorn.

Inlärningshastigheten styr, som ordet betyder, med vilken hastighet som Q-värden sprider sig bakåt. Avdragsfaktorn styr med vilken vikt framtida belöningar har.

Belöningsvariabeln r är bara skild från noll i tillstånd där agenten antas befinna sig i ett fördelaktigt tillstånd. Vilka tillstånd som ska belönas beror på vad som eftersträvas med agenten, vad som ska läras. Implementationen av algoritmen ansvarar för att ge belöning i rätt tillfälle. I utveckling av en spelande agent med målet att bli så skicklig som möjligt kan det vara lämpligt att t.ex. belöna vinnande tillstånd och tillstånd där motståndaren blockeras från att nå ett vinnande tillstånd.

Q-värden propageras alltså bakåt till det föregående tillståndet då ett drag har gjorts. Alltså, om en agent når ett belönande tillstånd ett spel efter N spelade drag så krävs det minst N spelade spel innan denna belöning når det första tillståndet. Dock är det antagligen fler än så.

Man kan även låta implementationen bestraffa agenten i dåliga tillstånd (t.ex. förlorande sådana) genom att dela ut en negativ belöning. På detta sätt får man agenten att undvika sådana tillstånd.

När en agent har besökt tillräckligt många tillstånd kommer den att hitta händelser som gynnar den, den lär sig att särskilja vissa drag som tar den till de tillstånd för vilka belöning delades ut. Man säger att Q-tabellen konvergerar mot en hållbar policy, och agenten har då lärt sig en optimal väg som leder till belöningarna. När Q-tabellen konvergerat så kommer storleken på Q_k för alla tillstånd k att vara optimal.

Explorativa drag

Det händer ofta att Q-tabellen konvergerar mot en policy som inte är den optimala. Då kan det vara nyttigt att genomföra så kallade explorativa drag. Det innebär att agenten i valet av nästa tillstånd bortser från sökningen av det maximala nästkom-

mande Q-värdet, och istället väljer nästa tillstånd på måfå. Oftast styrs beslutet att göra ett explorativt drag av en enkel sannolikhet.

De explorativa dragen har fördelen att de kan bortse från en konvergens mot en policy som inte är den optimala. Förhoppningen är att de av slumpen ska leda agenten till en starkare och bättre policy.

Kapitel 3

Experiment

Här börjar den praktiska delen av rapporten. Den innehåller användna metoder, resultat, diskussion samt slutsatser.

3.1 Metoder

I detta avsnitt beskrivs vilka metoder som använts för att samla in resultat.

3.1.1 Inlärningshastighet mot primitiv spelare

Här undersöks hur snabbt agenten kan förbättra sin prestationsförmåga mot en mycket primitiv och för människan förutsägbar motståndare. Prestation mäts som ett medelvärde av antal brickor lagda under ett spel. Agenten ställs in på att bli bestraffad vid förlust, bestraffad vid vinst och belönad vid oavgjort. Detta för att få den att lägga så många brickor på planen som möjligt. Källkod för agenten finns i Bilaga A.

Den primitiva spelaren spelar genom att hela tiden lägga sin bricka i vänstrast möjliga kolumn. När en kolumn är full, så lägger spelaren i den nästa till höger kommande kolumnen, och så vidare.

Syftet är att testa inlärning för en någorlunda begränsad tillståndsmängd, då datastrukturen förväntas bli stor i det allmänna fallet.

Efter varje tusen spelade spel beräknas ett medelvärde för antal överlevda drag under de tusen spelade matcherna. Dessa värden ritas sedan upp i en graf, och resultaten studeras.

3.1.2 Inlärningshastighet mot slumpmässig spelare

Målet med denna undersökning är att se hur snabbt agenten kan förbättra sin prestationsförmåga mot en slumpmässig spelare. Prestation mäts som ett medelvärde av antal vunna spel per tio tusen spelade spel. Agenten ställs in på belöning vid vinst, och bestraffning vid förlust och oavgjort spel.

Den slumpmässiga spelaren väljer sina drag genom att slumpa fram en siffra mellan (inklusivt) ett och sju, och lägger sin bricka i respektive kolumn, om det är ett giltigt drag. Råkar draget vara ogiltigt så slumpas en ny siffra fram.

Syftet är att testa inlärning för en mycket snabbt växande tillståndsmängd.

3.2 Implementation

Beskrivningen av implementationen är uppdelad i två delar. Den första delen beskriver implementationen av datastrukturen. Den andra delen beskriver implementationen av programmet.

3.2.1 Datastruktur

Här beskrivs implementationen av datastrukturen. Datastrukturen ska för varje unik spelplan och drag kunna lagra ett värde. Eftersom en spelplan och ett drag leder till en ny spelplan, så väljs datastrukturen istället till kunna lagra värden för spelplaner. Det som behövs är alltså en funktion som tar en spelplan och mappar denna till ett värde, som både ska kunna returneras och modifieras snabbt.

Javas HashMap väljs för denna implementation, med en Integer som nyckel. Hashfunktionen f av en spelplansmatris S bestäms till följande formel, med n som antal rutor på en spelplan:

$$f(S) = \sum_{k=0}^{n-1} 3^k S_k \tag{3.1}$$

Denna genererar garanterat ett unikt värde för varje spelplan. Värdet ’0’ i matrisen betyder ingen bricka placerad, värdet ’1’ betyder första spelarens bricka, och motsvarande gäller för värdet ’2’. Problemet är att värdet av f kommer överstiga maxstorleken för ett heltal i Java, och för att åtgärda det delar vi in spelplanen i tre olika delar, och låter varje del av spelplanen ha sin egen hash, men då istället med $n = 14$.

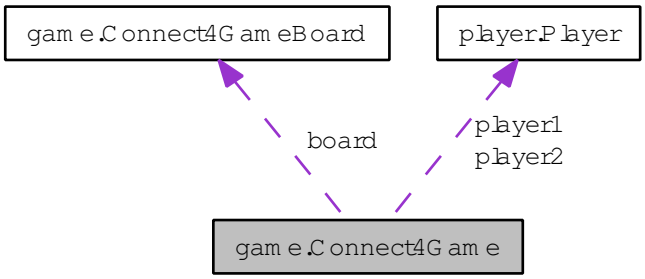
Datastrukturen väljs alltså till en tredimensionell HashMap. Typdeklarationen i java ser ut som följande.

```
private static HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> map;
```

Hela datastrukturen kan även med hjälp av enkla metodanrop från Javas standardbibliotek sparas ned och laddas från disk.

3.2.2 Program

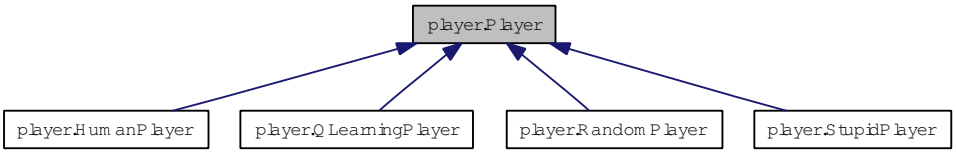
Programmet är som tidigare nämnt skrivet i Java. Klasserna är indelade i tre olika paket, 'game', 'player' och 'qlearning'.



Figur 3.1. Ett klassdiagram över de mest fundamentala delarna programmet.

Ett spel körs genom att huvudklassen Connect4Game skapar ett spelbräde och två spelare. Spelarna kan som synes vara antingen HumanPlayer, QLearningPlayer, RandomPlayer eller StupidPlayer.

Connect4Game låter sedan de två spelarna göra drag turvis tills spelet är över. Spelarna gör sina drag genom att Connect4Game kallar på en metod i den abstrakta klassen Player. Den tvingar sedan aktuell subklass (beroende på vilken spelare som ärvt från klassen) att välja drag. Detta gör att Connect4Game anropar till alla Players på samma sätt, men samtidigt kan subklasserna själva bestämma hur nästa drag ska väljas, vilket är komfortabelt.



Figur 3.2. Ett klassdiagram över logiken som hanterar spelarna.

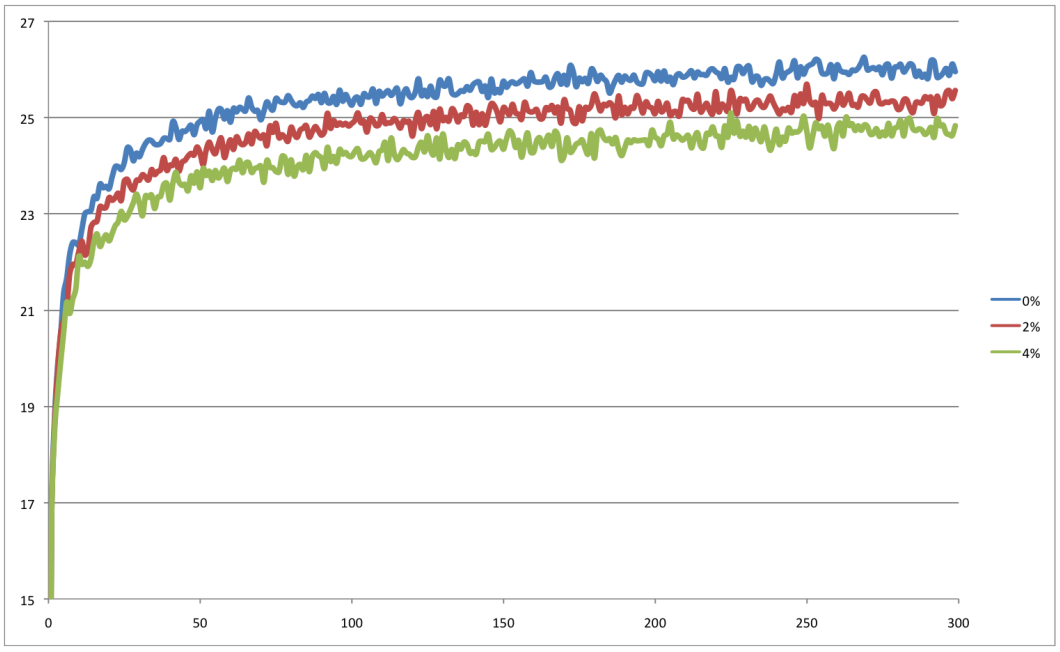
HumanPlayer väljer drag genom att läsa indata från terminalen, för att ge en mänsklig spelare möjlighet att välja drag. Denna klass är lämplig att använda då man vill provspela mot en upplärd agent. StupidPlayer är en datorstyrd spelare som alltid lägger sina brickor i vänstrast möjliga kolumn. RandomPlayer är spelaren som väljer drag genom att använda genererade slumptal. QLearningPlayer är den med hjälp av Q-learning implementerade algoritmen.

3.3 Resultat

3.3.1 Metod 1

Agenten har här spelat 300,000 matcher mot en primitiv spelare. För varje tusen spelade spel har medelvärdet av agentens antal överlevda drag räknats ut och ritats upp i diagrammet nedan. Medelvärdet beror inte på tidigare beräknade medelvärden, för varje tusen nya spel beräknas alltså ett nytt oberoende medelvärde.

Agenten minskar sin prestation med ungefär ett fjärdedels drag per procent explorativ grad som lades till. Färgerna i diagrammets linjer (figur 3.3) betecknar vilken grad av explorativitet som använts hos agenten vid försöket. Med grad av explorativitet menas här sannolikheten att agenten avviker från det uträknade maximala Q-värdet och istället välja ett slumpmässigt drag.

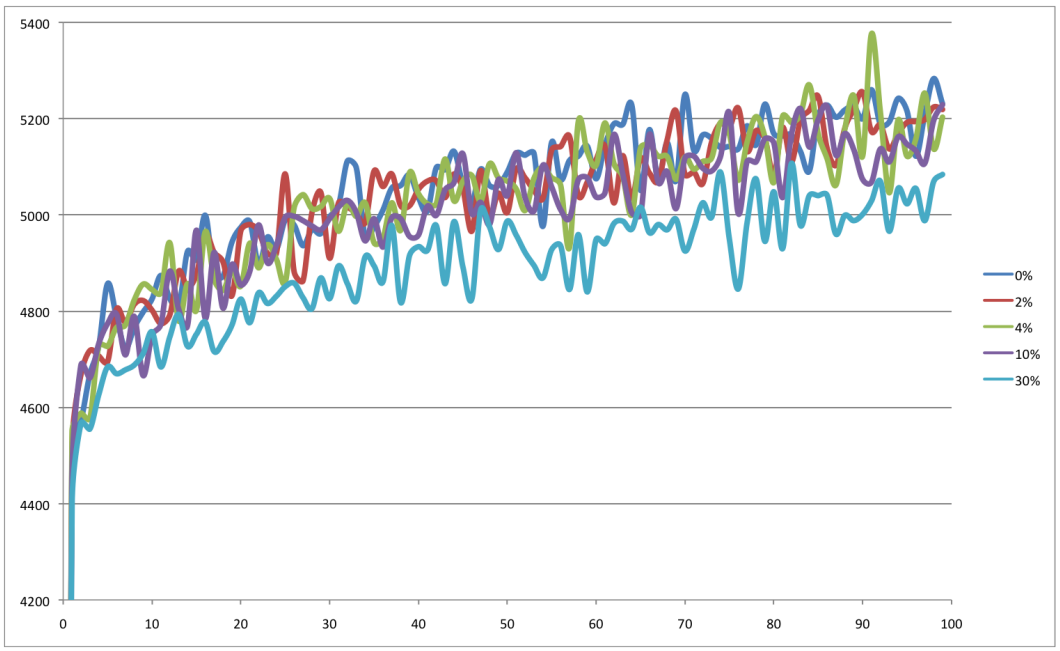


Figur 3.3. Resultat av 300,000 spelade spel mot primitiv spelare. På x-axeln tusental spelade spel, och på y-axeln medelvärde för antal drag de tusen senaste spelen. De olika färgerna betecknar olika grad av explorativitet.

3.3.2 Metod 2

Här har agenten spelat mot en slumpmässig spelare. Färgerna i diagrammet (figur 3.4) representerar agentens explorativitet.

De lägre graderna av explorativitet (0-4%) visar sig ge bäst resultat. En högre grad verkar försämra resultaten.



Figur 3.4. Resultat av 1,000,000 spelade spel mot slumpmässig spelare. På x-axeln tiotusental spelade spel, och på y-axeln medelvärde för antal drag de tio tusen senaste spelen. De olika färgerna betecknar olika grad av explorativitet.

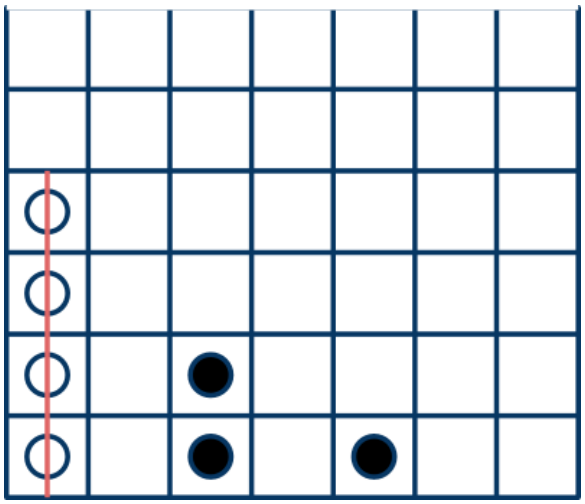
3.4 Diskussion

3.4.1 Metod 1

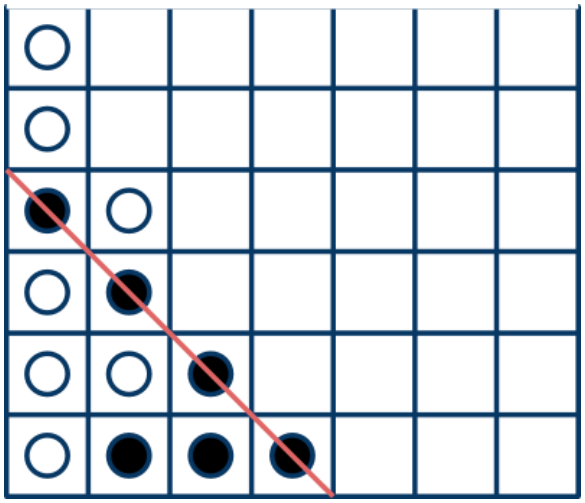
De första tiotalen spelade spelen presterade agenten dåligt, som väntat. Innan agenten lärt sig att den skulle lägga en av sina tre första brickor i den första kolumnen, för att blockera den första möjliga vinsten, så förlorade den efter ungefär sju drag. Se figur 3.5.

När alla de fall då agenten lägger sina tre första brickor i annat än den första kolumnen har uppstått, så har Q-värdena för alla dessa tillstånd tilldelats ett negativt Q-värde, tack vare bestraffningen av förlorande tillstånd. Då har agenten lärt sig att blockera den första lodräta möjligheten till vinst som uppstår. På detta sätt fortsätter det när agenten lär sig att skydda sig mot förlust. Efter ungefär ett tusen spelade spel kunde en spelplan se ut som i figur 3.6.

Det syns att inlärningen ger tydliga resultat genom att antal överlevna drag ökar strängt allt med att agenten får erfarenhet från nya spelade spel. Det noteras att även att denna ökning sker långsammare och långsammare. Det beror på att ju fler brickor det finns på ett spelbräde, desto mindre sannolikhet är det att samma spelbräde uppstår igen. Därmed går det långsammare att få erfarenhet av dessa.

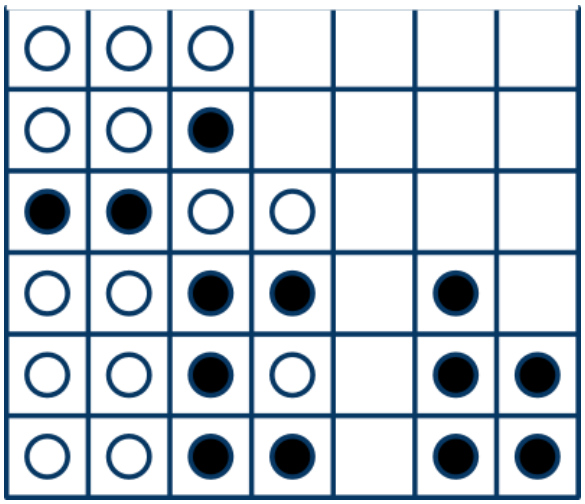


Figur 3.5. Agenten (svart) med dålig erfarenhet förlorar mot den primitiva spelaren.



Figur 3.6. Agenten (svart) med ett tusental spels erfarenhet har lyckas överleva och blockera två möjliga vinstchanser. Agenten bestraffas här för sin vinst.

I figur 3.7 ser man en förklaring till varför agenten ej överlever mer än ca 25 drag. Det är nu agentens tur att lägga en bricka, och agenten spelar mot StupidPlayer. Om agenten lägger sin bricka i kolumn fyra kommer StupidPlayer att vinna i nästa drag, vilket resulterar i en bestraffning som agenten försöker undvika. Om agenten istället lägger sin bricka i någon av de andra kolumnerna kommer agenten antingen att vinna, eller låta StupidPlayer vinna genom att StupidPlayer lägger sin bricka i kolumn fyra. Båda dessa tillstånd bestraffas, och spelet är avslutat.



Figur 3.7. Ett typiskt spelbräde efter 27 drag. Oberoende av vad svart väljer för drag kommer den primitiva spelaren att vinna.

Explorativa drag

Att använda fler explorativa drag visar sig tydligt i resultaten att leda till färre överlevna drag. Det beror på att strategin som krävs för att motstå den primitiva spelaren är så pass strikt, så att lägga slumpmässiga drag kommer endast bortse från en helt inövad strategi.

Minnesanvändning

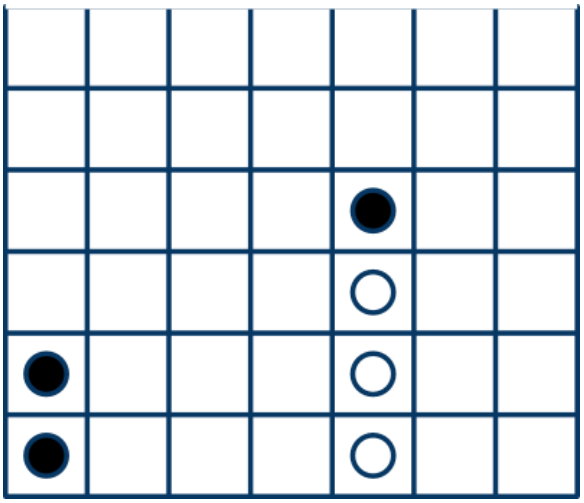
Efter cirka tre miljoner spelade spel mot den primitiva agenten så var datastrukturen som innehåller Q-tabellen inte större än 300MB. En förklaring till detta är att eftersom tillstånden som uppstår upprepas hela tiden, så finns det inga nya tillstånd att allokera minne för, och därmed står minnesanvändningen efter en tids inlärning still.

3.4.2 Metod 2

Till skillnad från Metod 1 så utgick inte det här experimentet ifrån en spelaragent som spelar med en konstruktiv strategi. Agenten spelade istället mot en helt slumpmässig agent som på måfå väljer sina drag. Andelen vunna spel var därför inte särskilt låg till en början. Syftet var att få ett test som skiljer sig från Metod 1.

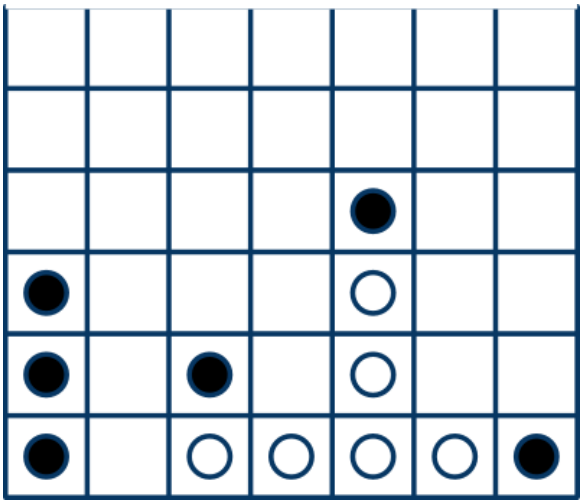
Inlärningshastigheten förväntades ändå vara avtagande, vilket resultaten bekräftade att den var. Man kan i figur 3.4 se en avtagande ökning för alla provade agenter. Det beror på

Spelskickligheten hos en agent i ett tillstånd är likt tidigare proportionell mot sannolikheten att tillståndet uppstår (antal brickor lagda). Spelbrädet kunde se ut som figur 3.8 efter endast ett fåtal brickor lagda. Agenten visar tecken på väletablerade spelstrategier.



Figur 3.8. Agenten blockerar en möjlig vinstchans, tack vare stor erfarenhet på grund av få lagda brickor.

Men efter ungefär dubbelt så många så minskar spelkvaliteten, då det uppstår tillstånd som agenten inte lärt sig hur den ska behandla än. Då förlorar agenten, som visas i figur 3.9.



Figur 3.9. Många brickor har lagts, agentens erfarenhet av tillstånden som uppstår har minskat avsevärt. Agenten förlorar snabbt.

Explorativa drag

Till skillnad från Metod 1 så verkar inte en låg grad av explorativitet (c.a. 2-4%) vara missgynnande för inlärningen, det verkar snarare som om inlärningen sker ungefär lika snabbt. Dock är det klart att en högre grad av explorativitet (10-30%) ledde till försämrade resultat. Men att explorativiteten ändå ger bättre resultat här än i Metod 1 beror på att agenten inte spelar mot en lika enformig spelare. Fler tillstånd uppstår här, och explorativiteten förstör därför inte lika mycket.

Minnesanvändning

Minnesanvändningen för det här testet var mycket stor, som väntat. Efter en miljon spelade spel så krävde Q-tabellen en storlek av 2.5 GB RAM. Testet kunde köras tack vare tillgång till ett kraftfullt datorsystem inom skolans nät.

3.5 Slutsats

Att implementera en allmänt skicklig fyra-i-rad-spelare med hjälp av Q-inlärning, på det sätt som gjorts här, skulle kräva orimligt lång inlärningstid och orealistiskt mycket minne. För att uppnå bra resultat med inlärningen så krävs det att man formulerar ett problem som begränsar tillståndsmängden kraftigt, såsom i den här rapporten gjordes i den första undersökningen. Detta gjorde att resultat kunde fås snabbare.

Den andra undersökningen visade som väntat att tillståndsmängden är för stor för att agenten ska kunna etablera en konstruktiv strategi genom ett helt spel.

Man hade kunnat använda sig av sökfunktioner som med en enkel lokal sökning framåt ett eller flera steg framåt i tillståndsmängden kunde ge råd till Q-inlärningen, vilket drag som var att föredra. En viktning av detta råd och Q-tabellen kunde sedan skapa ett gemensamt råd om vilka drag som bör göras. Till en början planerades detta som en del av projektet, men p.g.a. brist på tid så var detta tyvärr inte möjligt.

Bilaga A

Källkod

Källkoden för implementationen kan hämtas här:
`http://www.nada.kth.se/~edvine/dkand10/source.zip`

Litteraturförteckning

[1] Richard S. Sutton & Andrew G. Barto, *Reinforcement Learning: an Introduction.*
`http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node65.html`
MIT Press, Cambridge, MA, 1994.

[2] Victor Allis, *A Knowledge-based Approach of Connect-Four.*
`http://www.connectfour.net/Files/connect4.pdf`
Department of Mathematics and Computer Science, Amsterdam, The Netherlands, 1988.

[3] Andrew G. Barto, Sridhar Mahadevan, *Recent Advances in Hierarchical Reinforcement Learning.*
`http://rlai.cs.ualberta.ca/papers/barto03recent.pdf`
Department of Computer Science University of Massachusetts, Amherst, MA, 2003.