Artificiell Intelligens och Java

Kandidatavhandling

Matias Myréen, 32696

Institutionen för Informationsteknologi

Våren 2011

Referat

Det kommer att antas att grunderna i Java är bekanta för läsaren och således kommer dessa inte att gås igenom. Däremot kommer fördelarna med Java när det gäller att programmera artificiella intelligenser att tas upp. Avhandlingen är delad i två huvuddelar, en teoridel och en mera praktiskt inriktad del. Teoridelen tar upp huvudtankegångarna som krävs för att förstå exempelprogrammen i praktikdelen.

Definitionen av en problemrymd och hur de tillstånd som en artificiell intelligens kan befinna sig i presenteras, så även datastrukturerna för tillståndsgrafer och hur en tillståndsgraf byggs upp. Sökningar genom tillståndgrafer med hjälp av kända sökalgoritmer, däribland djup-först-sökning (depth-first-search) och bredd-först-sökning (breath-first-search) gås genom. Därefter följer en genomgång av enkla heuristiker för tillståndsgrafer och hur bäst-först-sökning (best-first-search) definieras. Bakgrunden till maskininlärning och hur en artificiell intelligens kan lära sig eller bygga upp en tillståndsrymd presenteras med hjälp av exempel och generaliseringar. Teorin kommer att framhävas med hjälp av exempel skrivna i Java.

Innehållsförteckning

Referat i

Innehållsförteckning ii

1. Inledning 1

2. Java 3

3. Sökning 4

3.1 Bredd-först-sökning och djup-först-sökning 5

3.2 Heuristik och bäst-först-sökning 7

4. Maskininlärning 11

4.1 Maskininlärningsalgoritmer 12

5. Javaexempel på sökning 14

6. Javaexempel på maskininlärning 16

7. Avslutning 21

8. Källor 22

9. Bilaga 1 25

10. Bilaga 2 29

1. Inledning

Intelligens är ett svårdefinierbart begrepp och är något som många har filosoferat på genom århundradena. Det finns troligtvis lika många definitioner av intelligens som det finns personer som forskar i det. Till de allmännaste definitionerna hör bland annat förmågan att lära sig, att förstå vad man lärt sig, att tänka abstrakt, förstå idéer och språk [1]. Med artificiell intelligens menas alltså en konstgjord intelligens, något som människan har skapat. För en dator är det enkelt att förstå komplexa matematiska formler och problem och snabbt hitta en lösning på dessa, medan en människa med livslång erfarenhet av matematik kan behöva en betydligt längre tid eller kanske inte alls ser en lösning på problemet. En dator har dock mycket svårare att förstå naturliga språk, till exempel meningen ”Hon stal hans hjärta”. Meningen kan tolkas på två olika sätt; antingen handlar det om kärlek eller en organtjuv. Vilkendera det gäller kan ses ur sammanhanget. En dator som bara går igenom texten ser inte sammanhanget och kan således inte förstå vad meningen egentligen betyder. Orsaken till detta är att matematik har klara fastslagna gränser och regler, medan naturliga språk baserar sig mera på hur någonting sägs, i vilket sammanhang och de kan ha dolda meningar [2].

Hollywoods filmer har i 40 år drömt upp en bild av artificiella intelligenser. I Stanley Kubricks film ”2001: Ett rymdäventyr” från 1968 finns den psykotiska datorn HAL 9000 som ger en sorts bild av vad man strävat till att uppnå (kanske inte en psykotisk dator, men en maskin som uppfyller de flesta kraven på en intelligens och förstår naturliga språk). Dagens teknologi har inte nått fram till detta mål ännu, men teknologin går alltid framåt och en dag det kanske existerar en artificiell intelligens som är jämlik människans hjärna på att lära sig och förstå. Men det måste dock pekas ut att meningen med en artificiell intelligens inte alltid är att simulera mänsklig intelligens. En artificiell intelligens kan lösa ett problem som en människa eller ett djur skulle lösa det. En artificiell intelligens är dock inte bunden till de metoder en människa eller ett djur skulle använda utan den kan använda metoder

som är mera matematikbaserade och således enklare för en maskin att arbeta med. [3]

Målet med denna kandidatavhandling är dock inte att uppfinna något revolutionerande utan att ge en bakgrund till vissa av de algoritmer som existerar idag, och med hjälp av programmeringsspråket Java ge exempel på lösningar till några problem. Det existerar även andra bra språk för detta ändamål, till exempel Prolog och Lisp. Alla tre språken har sina fördelar, vilket tas upp i kapitel 2. Dessutom kan man även programmera med samma stil i Java som man gör i Prolog och Lisp, men det är inte sagt att man uppnår samma optimeringsgrad eller så kan programmen bli så invecklade att detta inte lönar sig. Java är även det nyaste av de tre språken och ett av de språk som används mest i informationsteknologiutbildningen och programvaruindustrin idag [4], därför är det intressant att se hur Java kan användas även inom artificiell intelligens forskningen.

2. Java

Området för artificiell intelligens är väldigt brett, och inget programmeringsspråk kan anses som det enda tänkbara språket att programmera en artificiell intelligens med. Artificiell intelligens har djupa grunder inom matematiken och därmed har även programmeringsspråk som baserar sig på matematik utformats för att enklare kunna representera problem och lösningar [5]. Speciellt kan man nämna Prolog, med grunder inom första ordningens predikatlogik, och Lisp, som är ett funktionellt språk. Java, som är objektorienterat, skiljer sig märkvärt från Lisp och Prolog speciellt i programmeringsstil, men det visar sig att Javas objektorientering lämpar sig också för att implementera en artificiell intelligens [6]. Det är väldigt sällan som det är siffror och strängar som en artificiell intelligens interagerar med; oftast är det någon sorts symboler eller tillstånd och dessa kan enkelt representeras av objekt i Java. Vidare går det i Java att definiera gränssnitt för objekt och implementera funktioner och klasser som agerar mot gränssnitten [7]. Detta leder till en viss abstraktion som även det är en av Javas styrkor, och programkod kan enkelt återanvändas för olika ändamål [6].

Java är ändå inte det ideala språket för att programmera en artificiell intelligens med. När det till exempel gäller expertsystem, det vill säga ett program som härmar en mänsklig expert på ett specifikt område, så är det Prolog som briljerar. Det går med ett simpelt program skrivet i Prolog att beskriva ett enkelt expertsystem. Motsvarande program i Java skulle vara mycket mera omfattande och komplext [6]. Allt är ändå möjligt att implementera i Java, men varje programmeringsspråk har sina egna specifika problem som de kan briljera med att lösa på ett enkelt sätt. Eftersom Javas styrka ligger i abstraktion, kan sökning i tillståndsrymder och maskininlärning lösas i Java på ett lämpligt sätt [6][7].

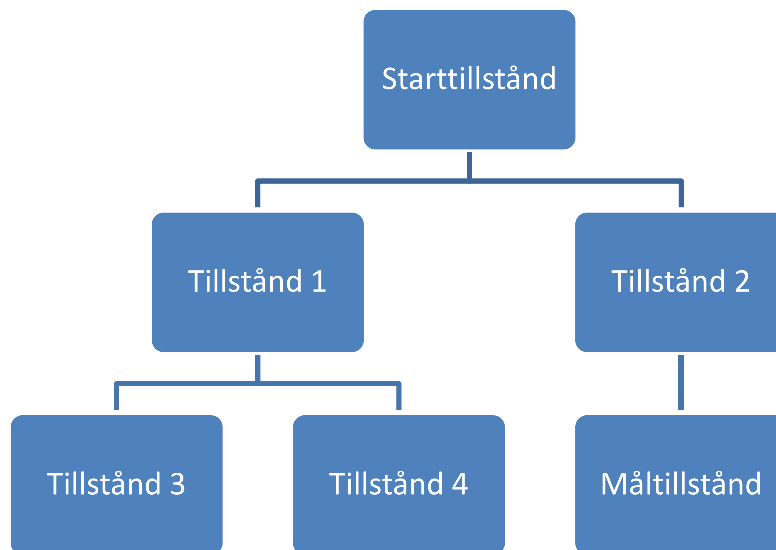
3. Sökning

Tidig forskning inom artificiell intelligens fokuserades mest på optimering av sökalgoritmer för tillståndsgrafer. Detta var ett logiskt första steg för en hel del uppgifter som en artificiell intelligens ska lösa. Detta görs genom att definiera en klar problemrymd och ur den få en tillståndsgraf. Den bästa lösningen på problemet hittas genom en effektiv sökning genom tillståndsgrafen. Sökning är således grunden för både utveckling av teoretiska modeller och lösningar på praktiska tillämpningar [5] [8]. Med hjälp av både effektiva algoritmer och heuristik, tilldela värden till tillstånden baserat på hur nära en lösning tillståndet är, kan sökningarna optimeras. Även genom att spara extra data i minnet kan sökningar effektiviseras och således ger de ett bättre resultat. Extra minneslagring gör söktiden snabbare och kan effektivisera framtida sökningar inom samma tillståndsgraf [6] [8].

Fördelen med Java i det här skedet är att Javas stöd för abstraktion och gränssnitt gör det enklare att implementera olika sökalgoritmer som opererar på samma tillståndsgraf. Gemensamma faktorer så som representationen av tillståndsgrafen och funktionaliteten för att definiera tillåtna steg kan implementeras i en abstrakt klass med hjälp av Java. De specifika sökalgoritmerna kan sedan definieras skilt och således dela på funktionalitet mellan klasserna. På det här sättet kan sökalgoritmen fritt bytas ut eller optimeras utan att grundfunktionaliteten påverkas. [6] [8]

Det finns två huvudelement som måste definieras för att kunna bygga upp en tillståndsgraf. Först och främst måste det finnas en formell representation av tillstånden. Det kan tänkas att tillstånden är stegen mot en lösning. Första tillståndet är ursprungstillståndet, hur problemet ser ut just nu. Därefter förgrenar sig grafen för varje möjligt steg till nästa tillstånd, vilket kan vara ett eller flera beroende på hur stegen mellan tillstånden är definierade och om det tillåts att samma tillstånd besöks flera gånger [6] [8]. Den andra saken som krävs är operatorer för att generera följande tillåtna tillstånd från ett godtyckligt tillstånd. Beroende på hur problemrymden ser ut och detta är direkt beroende på problemet (till exempel

tillåtna steg i ett spel eller olika skeden i en kemisk process). Stegen kan även vara baserade på heuristik, logisk slutledning eller exempel tagna från tillgängliga data [8]. De olika möjliga tillstånden fungerar som noderna i grafen medan operatorerna utgör bågarna mellan de olika tillstånden (se exempel i figur 1). Det kan även krävas att tidigare data lagras över vilka tillstånd som besökts för att undvika att grafen upprepar sig själv.



Figur 1. Exempelgraf över möjliga tillstånd.

3.1 Bredd-först-sökning och djup-först-sökning

Grunddesignen i bredd-först-sökning och djup-först-sökning är i princip densamma. Båda algoritmerna går iterativt genom tillståndsgrafen. De börjar från starttillståndet och avslutar med ett måltillstånd, ifall ett eller flera sådana existerar. Ifall det nuvarande tillståndet algoritmerna befinner sig i inte är ett måltillstånd så genereras barntillstånd (child states) till det nuvarande tillståndet, och dessa placeras i en tillståndslista. [6]

I figur 2 finns pseudokoden en generell sökalgoritm som både bredd-först-sökning och djup-först-sökning följer. Den huvudsakliga skillnaden mellan de två algoritmerna är hur de opererar på tillståndslistan.

```
Sökning(Starttillstånd)
{
        placera starttillståndet i tillståndslistan;
        medan tillståndslistan inte är tom gör följande
        {
                Tillstånd = nästa tillstånd i tillståndslistan;
                Om tillståndet är ett måltillstånd
                        Algoritmen avslutas med framgång;
                Annars
                        Generera alla barntillstånd till det nuvarande tillståndet
                        och placera de tillstånd som inte har besökts tidigare i
                        tillståndslistan;
        }
}
```

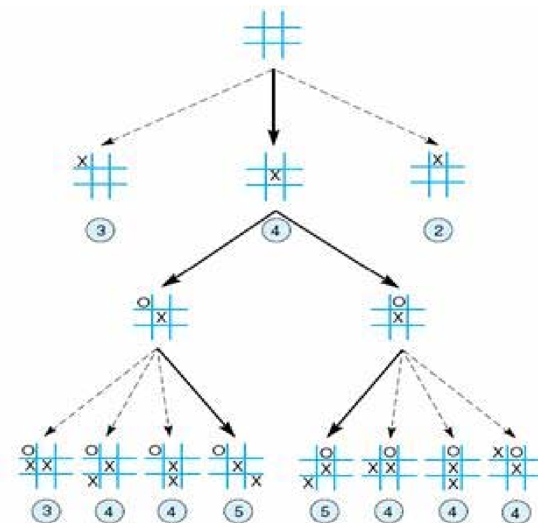
Figur 2. Pseudokod för generell sökalgoritm. [6]

Bredd-först-sökning använder sig av en tillståndslista som fungerar som en kö; det första tillståndet som sätts in i listan är också det första tillståndet som returneras av listan. Detta gör att bredd-först-sökning garanterat hittar den kortaste vägen till ett måltillstånd, på grund av att den går igenom alla tillstånd på ett godtyckligt djup d innan algoritmen börjar gå igenom tillstånd på djupet d+1 [6] [7]. Djup-först-sökning ser på tillståndslistan som en stack. En stack fungerar så att det sista tillståndet som lagts till listan är det första som kommer att returneras av listan. Detta leder till att djup-först-sökning gör en mera aggressiv sökning genom grafen [6] [7]. För att öka chanserna att hitta en rätt lösning begränsas ofta djup-först-sökning-algoritmen till ett maximalt tillåtet djup i grafen, annars finns det en risk att djup-först-sökning inte hittar en rätt lösning [6]. Det maximala tillåtna djupet ökas vartefter grafen gås igenom ifall ett måltillstånd inte hittades inom det specificerade djupet. Varken djup-först-sökning eller bredd-först-sökning tar i beaktande den information som det nuvarande och de tidigare tillstånden har att erbjuda; bredd-först-sökning tar endast i beaktande strukturen på tillståndsgrafen. Båda algoritmerna klassas därför som oinformerade sökalgoritmer. [6]

Ifall sökrymden är väldigt stor blir det väldigt opraktiskt att använda djup-först-sökning och bredd-först-sökning. Till exempel sökrymden för spelet schack har 2^{120} tillstånd, vilket är ett tal större än antalet mikrosekunder som gått sedan universums skapande. För att hitta lösningar i en stor sökrymd krävs det en algoritm som kan hitta, utgående från tidigare tillstånd och det nuvarande tillståndet, vilka barntillstånd som är de mest sannolika att fortsätta med [5] [6].

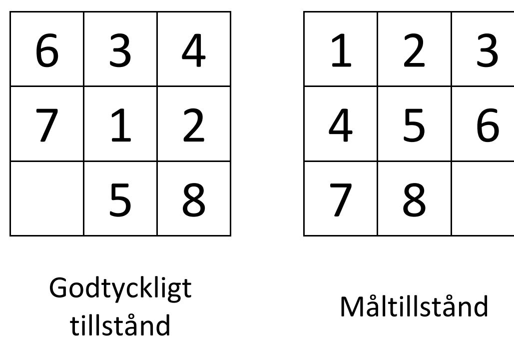
3.2 Heuristik och bäst-först-sökning

Det finns även sätt att förminska tillståndsgrafen för att göra sökningarna snabbare. I bondschack, med en spelplan som är 3x3 rutor stor, kan tillståndsgrafen snabbt definieras. Det finns max nio drag i ett spel, fem drag respektive fyra drag för de båda spelarna. I det första draget finns det 9 olika möjligheter, i det andra draget finns det 8 möjligheter och så vidare tills spelplanen är fylld. Antalet noder i tillståndsgrafen är då $9!$ stycken, med andra ord 362 880 stycken. Men tillståndsgrafen kan reduceras med hjälp av symmetrin som uppstår [5]. I det första draget finns det nu tre olika möjligheter: mitten, ett hörn eller en sida. Med hjälp av simpel heuristik går det även att gå igenom tillståndsgrafen för bondschack utan att använda sig av en sökalgoritm. För att helt förbise sökning måste det i varje tillstånd definieras ett attribut som står för antalet vinstmöjligheter som existerar (se exempel i figur 3). Nästa steg i tillståndsgrafen är alltid tillståndet med flest vinstmöjligheter [7]. Denna metod kallas för bergbestigningsmetoden (Hill-Climbing procedure) [5] och kan liknas med en blind bergbestigare som alltid väljer den brantaste sluttningen att klättra uppför tills det inte går att komma högre upp. Vissa brister existerar dock, metoden sparar ingen historia över var den varit. Algoritmen kan då alltså inte gå tillbaka i tillståndsgrafen och pröva en annan väg ifall den inte når ett måltillstånd. Den största bristen som bergbestigningsmetoden har är att den hittar enbart det lokala maximum. Det hittade lokala maximumet kan vara ett måltillstånd, men det är inte säkert [5].



Figur 3. Tillståndsrymd över bondschack med antalet vinstmöjligheter för varje tillstånd angivet. [9]

Algoritmen fastnar alltså i ett tillstånd ifall alla dess barntillstånd har ett sämre värde än det nuvarande tillståndet. Detta fall kommer inte fram i bondschackexemplet, men till exempel i lösning av femtonspelet kommer det fram att det behövs mera bergsbestigarmetoden för att åstadkomma en bäst-först-sökning. För att förenkla spelet förminskas spelplanen till 3x3 rutor. I spelet, som nu kallas för åttapusslet, finns åtta stycken numrerade brickor och en tom plats (se ett godtyckligt tillstånd i figur 4), i den ursprungliga versionen var detta femton stycken numrerade brickor och en tom. Det enda måltillståndet definieras även i figur 4. Transaktioner mellan tillstånd definieras utgående från hur den tomma rutan kan byta plats med någon av de angränsande brickorna.

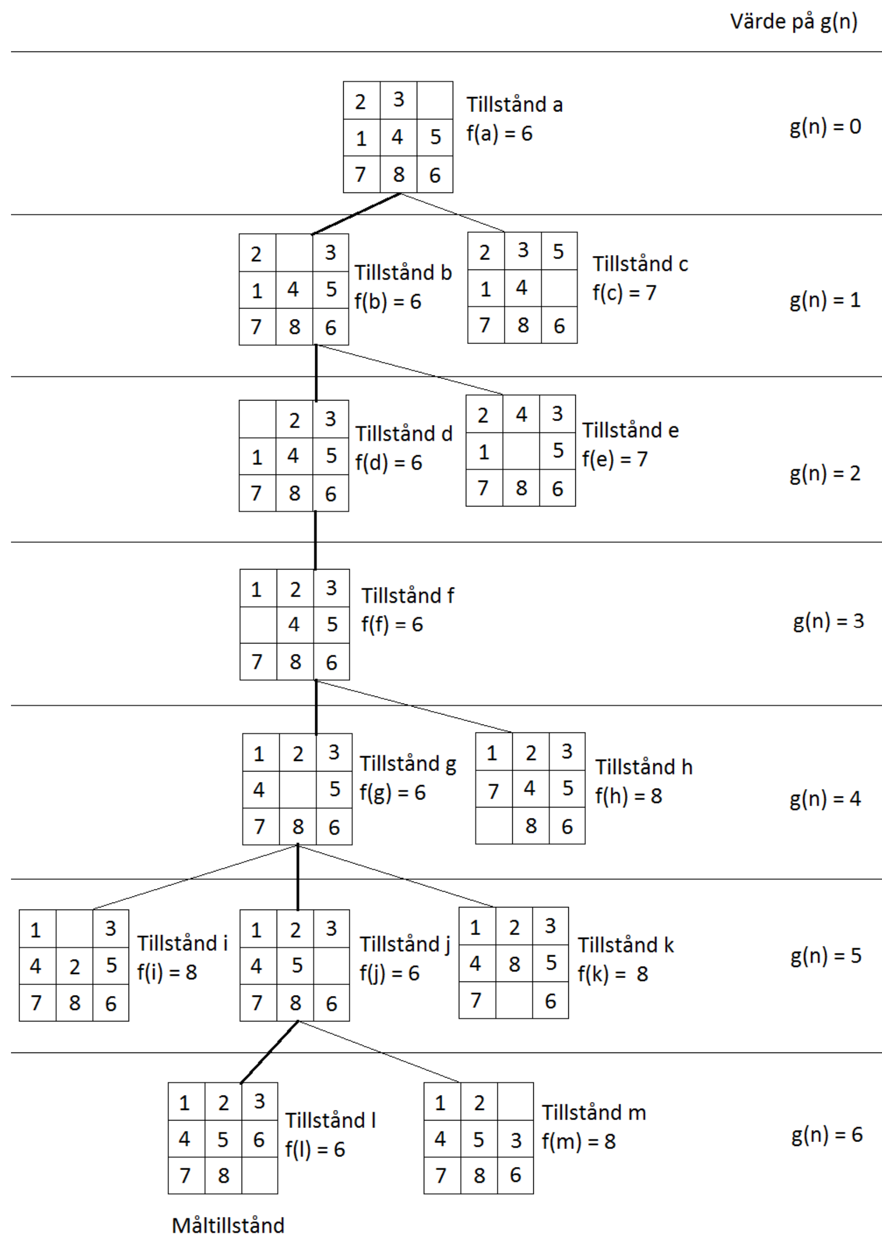


Figur 4. Exempeltillstånd för åttapusslet.

Både djup-först-sökning och bredd-först-sökning kan användas för att hitta fram till måltillståndet, men för att alltid kunna välja den bästa vägen igenom tillståndsgrafen, vilket djup-först-sökning och bredd-först-sökning inte gör, måste någon typ av heuristik användas [5] [7].

Ifall en bergbestigaralgoritm används på åttapusslet definieras ett värde för varje tillstånd. Värdet är antalet rutor som är på rätt plats. I måltillståndet är detta värde nio och i det godtyckliga tillståndet i figur 4 är det noll. Algoritmen kommer aldrig att flytta på en bricka som är på sin rätta plats. Naturen av åttapusslet gör att det ibland är nödvändigt att flytta på brickor som är på rätt plats för att nå måltillståndet, så det behövs mera information för att kunna lösa åttapusslet med hjälp av heuristik [5]. I en bäst-först-sökning för åttapusslet definieras värdet på varje tillstånd som en funktion $f(n)$ sådan att $f(n) = g(n) + h(n)$ [5] [10], där n är ett godtyckligt tillstånd, $g(n)$ är hur långt ifrån det första tillståndet n är, med andra ord djupet i grafen, och $h(n)$ är hur många brickor som är på fel plats [5]. Samma pseudokod som finns i figur 2 kan användas även för bäst-först-sökning. Men istället för att använda en stack, som i djup-först-sökning, eller en kö, som i bredd-först-sökning, så använder sig bäst-först-sökning-algoritmen av en sorterad lista i form av en prioritetskö [5] [10]. Det tillstånd som returneras av prioritetskön är tillståndet med lägst värde på $f(n)$. Algoritmen använder sig av $g(n)$ på grund av att värdet på $g(n)$ förhindrar algoritmen från att gå allt för djupt ner i grafen utan prioritera istället tillstånd som dyker upp tidigare i grafen.

Figur 5 visar hur en tillståndsrymd kan se ut för bäst-först-sökning. I det här fallet kommer algoritmen aldrig att gå tillbaka och pröva en annan väg, men det bör noteras att det är möjligt för den att göra det [5] [8] [10]. Orsaken till att $f(n)$ är sex för varje tillstånd som gås igenom är att algoritmen i det här fallet lyckas att i varje steg placera en bricka rätt, vilket inte alltid är fallet. Den här implementationen av bäst-först-sökning garanterar även att den optimala lösningsstigen kommer att hittas, i detta fall stigen a, b, d, f, g, j och l.



Figur 5. Tillståndsrymden genererad av bäst-först-sökning i åttapusslet. Inspirerad av George Lugers graf [5].

Bäst-först-sökningsalgoritmen är generell och funktionen $f(n)$ kan beräknas på en del olika sätt [10], alltifrån uppskattningar av hur bra ett tillstånd är till hur sannolikt det är för ett tillstånd att nå ett måltillstånd [5]. Det måste dock alltid finnas kunskap om hur tillståndsrymden ser ut, eller kommer att se ut, för att kunna implementera sökning som en artificiell intelligens.

4. Maskininlärning

En definition av inlärning lyder ”any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population” given av Herbert Simon [11]. Med andra ord innebär inlärning att göra något bättre för varje gång det görs. Erfarenheten från tidigare försök berättar vad som kan förväntas av följande försök och vad som ska göras för att uppnå ett bättre resultat.

Det finns fyra olika huvudtankesätt för maskin inlärning, varav två har grunder i matematiken, nämligen symbolbaserad inlärning (symbol-based learning) och sannolikhetsinlärning (probabilistic learning). De två andra tankesätten har grunder i naturen, konnektionistisk inlärning (connectionistic learning) och genetisk och evolutionsbaserad inlärning (genetic and emergent learning). [5]

Symbolbaserad inlärning börjar med en mängd entiteter och relationer i en problemdomän. Inom symbolisk inlärning är målet att uppnå användbara generaliseringar som kan uttryckas med hjälp av dessa symboler. Symbolbaserad inlärning har sina grunder inom predikatlogiken. [5]

Sannolikhetsinlärning har sina grunder i Bayesisk sannolikhetslära. Det finns två huvudorsaker varför sannolikhetslära behövs för att förstå och förutspå händelser i problemrymden. För det första kan valen i en problemrymd vara helt oberoende av varandra, helt beroende av varandra eller delvis beroende av varandra. För det andra kan relationerna i den konstant förändrande problemrymden bli så komplexa att dessa samband bäst kan formuleras med stokastiska modeller. [5] [7]

Konnektionistisk inlärning modellerar kunskap som sammankopplade nätverk av individuella enheter. Inspirationen till konnektionism är tagen från arkitekturen i hjärnor, hur noder där är sammankopplade och skapar fungerande mönster. Neurala nätverk är ett bra exempel på en modell som använder sig av konnektionistisk inlärning. [5] [7]

Genetisk och evolutionsbaserad inlärning är, som namnet säger, inspirerad av naturlig evolution. Den börjar med en mängd av möjliga lösningar och därifrån utväljs de snabbaste av dem som leder till rätta lösningar. Dessa kombineras för att få fram en ny generation av möjliga lösningar som är lite bättre än de första. [5] [7]

4.1 Maskininlärningsalgoritmer

Ett enkelt sätt att utföra maskininlärning är med hjälp av exempel. Detta kallas för inlärning genom induktion och den mest grundläggande formen är att lära sig något utantill. Utantillärning (rote learning) innebär att alla exempel memoreras och appliceras sedan när en exakt likadan situation dyker upp igen. Eftersom utantillärning inte generaliserar exemplen uppstår det problem när situationer som inte ingår i exemplen dyker upp eller exemplen innehåller motsägande information [5]. För att vara effektiv måste en induktiv inlärningsalgoritm använda sig av flera inlärda exempel. Det eftersträvas att man med hjälp av heuristik och tillförlitliga generaliseringar kan hantera oförutsebara situationer och att algoritmen kan klara av dem med säkerhet på att den bästa lösningen används.

Maskininlärningsalgoritmerna kan delas upp i en mängd delklasser, men alla kommer inte att tas upp här:

Övervakad inlärning (supervised learning), där ramverket ges av användaren och det definierar den mängd resultat som kan åstadkommas. Exemplen modifierar ramverket och förändrar det enligt indata. Algoritmen får endast indata av sådan typ som är färdigt definierad. [5] [12]

Oövervakad inlärning (unsupervised learning), är raka motsatsen till övervakade inlärningsalgoritmer. Oövervakade inlärningsalgoritmer hittar gemensamma nämnare bland de testfall och indata som liknar varandra, och utgående från dessa härleds sedan generaliseringar för testfall och indata. [5] [12]

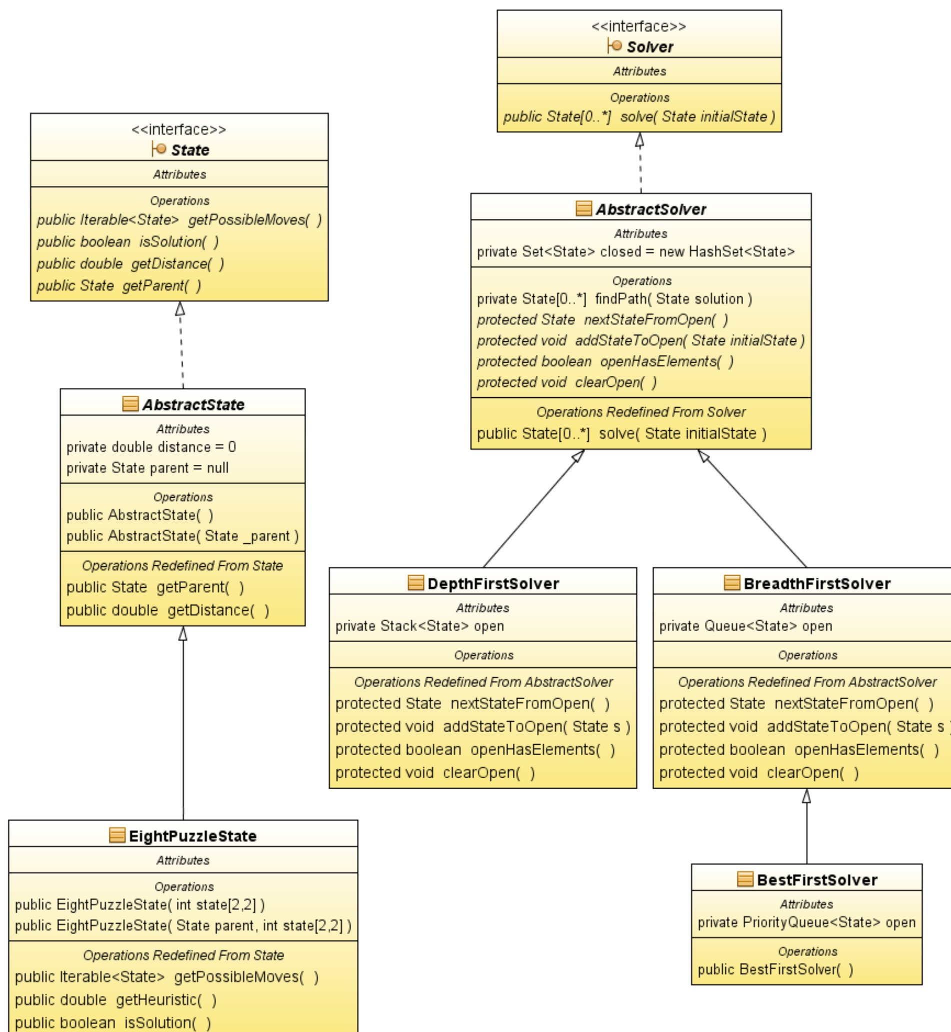
Delvis övervakad inlärning (semi-supervised learning) är en kombination av de två ovanstående. Algoritmen får en liten mängd färdigdefinierade indata och en större

mängd odefinierade indata. På ungefär samma sätt som en övervakad algoritm gör det generaliseras den interna modellen med hjälp av de givna indata. Modellen justeras med hjälp av den större mängden odefinierade indata för att ge en bättre lösning på problemet. [13]

I förstärkningsinlärning (reinforcement learning) är skillnaden från övervakad inlärning det att varken korrekta indata eller utdata är definierade på förhand. I förstärkningsinlärning är det enbart omgivningen som är definierad. I omgivningen finns tillstånd och transaktioner mellan tillstånd. Algoritmen använder sig av en sorts belöningssystem, för varje transaktion i omgivningen så definieras det hur bra just det steget var. Algoritmen optimerar processen genom att hitta den väg genom omgivningen som har den bästa totala belöningen. [5][14]

5. Javaexempel på sökning

Tidigare i kapitel 3 beskrevs det hur olika sökningsalgoritmer fungerar i teorin. Hur tillstånd och transaktioner definierades i både åttapusslet och bondschack togs upp. Det här kapitlet visar hur lösning av åttapusslet kan göras med hjälp av ett enkelt program skrivet i Java. Programkoden, förutom EightPuzzleState , är tagen i sin helhet från William Stubblefields och George Lugers bok AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java [6]. Exemplet använder sig av kända algoritmer och designmönster. En överblick över alla gränssnitt och klasser som exemplet använder sig av finns i figur 6 och all programkod hittas i Bilaga 1.



Figur 6. UML klassdiagram över de klasser som ingår i sökningsexemplet.

Likadant som i kapitel 3 börjar allting med hur ett tillstånd definieras. Ett gränssnitt State för tillstånd berättar åt de klasser som använder sig av tillstånd vad som förväntas av de olika funktionerna, namn och vad som returneras av funktionen. Klasser som använder sig av State känner enbart till de funktioner som är namngivna här. Alla klasser som implementerar State måste definiera alla dessa funktioner för att uppfylla kriterierna för State, gäller inte för abstrakta klasser. Funktionaliteten för att skapa barntillstånden för ett godtyckligt tillstånd hittas i getPossibleMoves, med andra ord ska transaktionerna mellan tillstånd placeras här.

AbstractState använder sig av State gränssnittet och implementerar de funktioner som kommer att vara gemensamma för alla konkreta tillstånd. De två olika konstruktorerna finns för att täcka de två typer av tillstånd som finns i ett tillstånd, starttillstånd utan föräldratillstånd (parent state) och barntillstånd som har ett föräldratillstånd. Den privata variabeln distance, som visar djupet för tillståndet, är definierad till noll från början för starttillståndet, när konstruktorn för barntillstånd används så uppdateras distance beroende på förälderns distance. Det bör noteras att ingenting problemspecifikt är definierat ännu i det här skedet. State och AbstractState innehåller endast den information som själva problemlösaren behöver använda. Själva problemet och hur barntillstånd definieras utgående från ett godtyckligt tillstånd definieras i de konkreta klasserna som är problemspecifika, i det här fallet EightPuzzleState. I EightPuzzleState representeras spelplanen med hjälp av en tvådimensionell matris med dimensionerna 3x3.

Solver fungerar som gränssnittet för problemlösarna, observera att endast en funktion är definierad utåt. Idén bakom funktionen solve är att den får in ett konkret tillstånd som starttillstånd, där all information om problemet finns, och returnerar en lista över de tillstånd som leder till ett måltillstånd från det givna starttillståndet.

AbstractSolver använder sig av Solver grännssnittet och implementerar solve enligt pseudokoden i figur 2 i kapitel 3. Det visade sig i kapitel 3 att det enda som skiljer de

olika sökalgoritmerna åt är hur de implementerar listan över öppna tillstånd. Därför lämnas implementationen av den öppna listan till underklasser av AbstractSolver och metoder för att accessera element i listan lämnas som abstrakta metoder som underklasserna måste implementera. Det som är gemensamt för alla underklasser är listan över besökta tillstånd och implementeras således i AbstractSolver, den sköts som ett HashSet som inte tillåter ekvivalenta element.

Det enda som definieras i de konkreta sökalgoritmklasserna (DepthFirstSolver, BreadthFirstSolver och BestFirstSolver) är hur den öppna listan definieras och hur den används. Det visade sig även att i det här fallet kunde BestFirstSolver använda sig av implementationen för bredd-först-sökning, eftersom båda använder sig av en kö så är gränssnittet lika för båda. Det som BestFirstSolver definierar ytterligare är hur två tillstånd jämförs med varandra inuti prioritetskön.

Java ger ett enkelt sätt att dela upp programmet i lämpliga bitar. Funktionaliteten sprids ut så att maximal mängd kod återanvänds och funktioner blir definierade på ett lämpligt abstraktionsdjup. Samma kod kan även återanvändas för andra problem, eftersom sökalgoritmerna baserar sig helt på hur det konkreta tillståndet har implementerats, allt som krävs är att de konkreta tillstånden ska implementera AbstractState och uppfylla de krav som definieras där.

6. Javaexempel på maskininlärning

Kapitel 4 tog upp olika sorter och typer av maskininlärningsalgoritmer. I dethär kapitlet kommer ett exempel på övervakad induktiv inlärning att byggas upp. Även detta exempel är taget i sin helhet från William Stubblefields och George Lugers bok AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java [6], exemplet använder sig av kända algoritmer och designmönster.

En bank behöver ett automatiskt system för att kategorisera nya låntagare enligt kreditrisk. Som bas används attribut som kredithistoria, skuld, pant och årsinkomst. Utgående från en mängd exempelfall kommer ett beslutsträd att byggas upp. För att

bestämma risken för en ny låntagare gås trädet igenom, och när en lövnod, en ända, nås är det den riskkategorin som låntagaren placerar sig i. Beslutsträdet kommer att byggas upp med hjälp av ID3-algoritmen, påkommen av Ross Quinlan [15].

Möjliga värden:

Pant är antingen tillräcklig eller ingenting.

Inkomst är 0 - 15,000 €/år, 15,000 - 35,000 €/år eller över 35,000 €/år.

Skuld är endera hög eller låg.

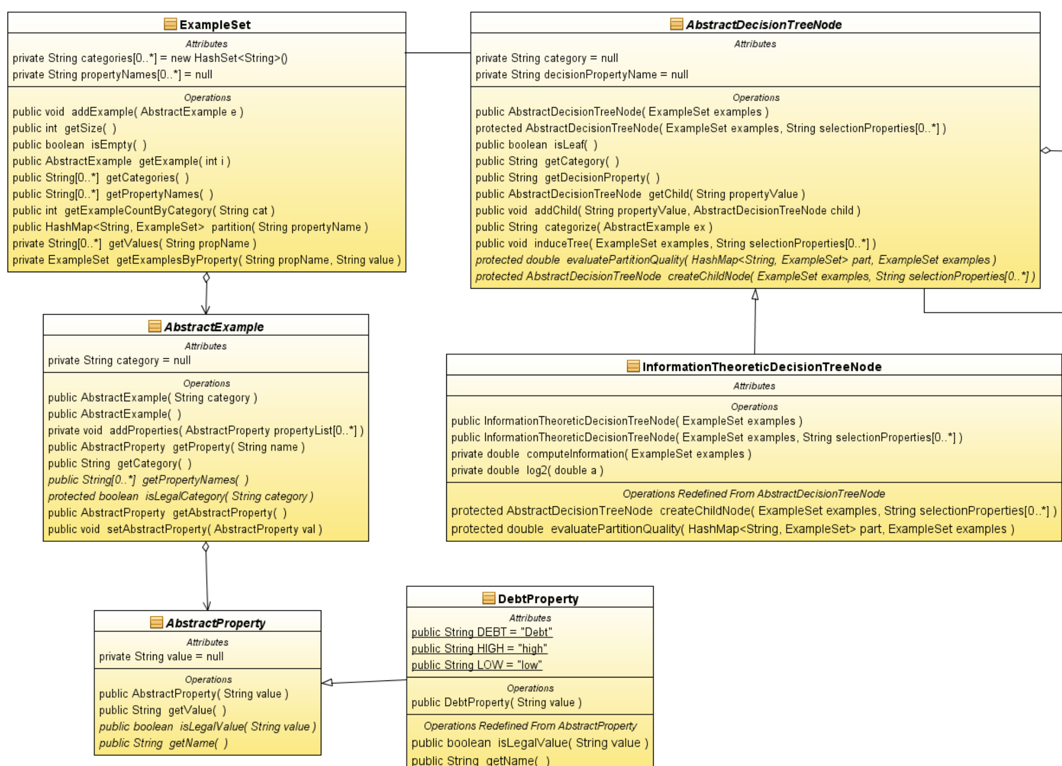
Kredithistoria är bra, dålig eller okänd.

Risk är låg, medel eller hög. Det är även risken som ska bestämmas.

För att kunna fortsätta bes banken att ge en mängd träningsexempel. Dessa fås från verkliga fall från bankens kundregister. Det är dock viktigt att peka ut att alla möjliga kombinationer inte behöver ges. Utgående från de möjliga värdena går det att räkna ut att det finns 36 olika kombinationer, men algoritmen klarar av att bygga upp ett beslutsträd med hjälp av ett färre antal kombinationer. Utmaningen är att beslutsträdet måste täcka alla träningsexempel korrekt och ha en hög sannolikhet att vara korrekt i alla fall utanför träningsexemplen. I detta exempel kommer 14 träningsexempel att användas, dessa hittas i figur 7. UML diagrammet i figur 8 kommer att refereras till kontinuerligt och programkoden för exemplet finns i Bilaga 2.

Risk	Pant	Inkomst	Skuld	Kredithistoria
Hög	Ingen	0 - 15,000 €/år	Hög	Dålig
Hög	Ingen	15,000 – 35,000 €/år	Hög	Okänd
Medel	Ingen	15,000 – 35,000 €/år	Låg	Okänd
Hög	Ingen	0 - 15,000€/år	Låg	Okänd
Låg	Ingen	Över 35,000 €/år	Låg	Okänd
Låg	Tillräcklig	Över 35,000 €/år	Låg	Okänd
Hög	Ingen	0 - 15,000€/år	Låg	Dålig
Medel	Tillräcklig	Över 35,000 €/år	Låg	Dålig
Låg	Ingen	Över 35,000 €/år	Låg	Bra
Låg	Tillräcklig	Över 35,000 €/år	Hög	Bra
Hög	Ingen	0 - 15,000€/år	Hög	Bra
Medel	Ingen	15,000 – 35,000 €/år	Hög	Bra
Låg	Ingen	Över 35,000 €/år	Hög	Bra
Hög	Ingen	15,000 – 35,000 €/år	Hög	Dålig

Figur 7. Tabell över träningsexempel.



Figur 8. UML klassdiagram över de klasser som ingår i inlärningsexemplet [6].

För att kunna representera tabellen i Java behövs det först och främst ett sätt att representera ett attribut, vilket motsvaras av en ruta i figur 7. För detta ändamål skapas en abstrakt klass AbstractProperty, som grund för de konkreta klasserna. AbstractProperty innehåller enkel funktionalitet som getValue, men den kräver att underklasser ska definiera funktionen isLegalValue för att kontrollera att värdet användaren försöker tilldela attributet är giltigt. För att definiera alla rutor som finns i figur 7 implementeras konkreta klasser i stil med DebtProperty, vilken representerar en ruta för skuld. Notera att DebtProperty inte innehåller en funktion för att byta värde på value, värdet på value kan definieras enbart via konstruktorn. Detta är en följd av att just den här inlärningsalgoritmen får problem ifall det tillåts att exempelfall ändras efterhand. Den måste köras pånytt med de modifierade exemplen för att fungera.

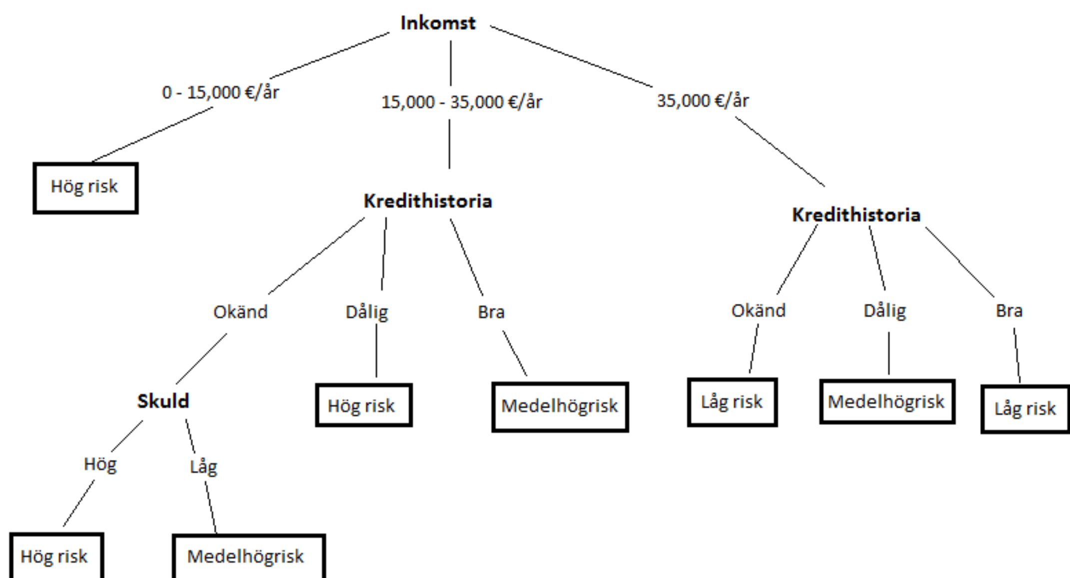
För att representera en rad i figur 7 skapas en AbstractExample klass, den byggs upp av en lista av AbstractProperty klasser. Kodexemplet som finns i Bilaga 2 kontrollerar även att alla AbstractProperty klasser innehåller rätt attribut för just den typen, till exempel att ett DebtProperty attribut innehåller det värde eller de värden som är definierade i DebtProperty klassen. AbstractExample har även olika konstruktorer för kategoriserade exempel (risken är känd) och för okategoriserade exempel (risken är okänd). Detta är för att hålla AbstractExample så generell som möjligt, senare i exemplet kommer det att krävas att varje exempel har en kategorisering.

Koden för hur hela tabellen i figur 7 implementeras finns i klassen ExampleSet. I funktionen addExample finns funktionaliteten som kontrollerar att varje exempel har en kategorisering. Utöver det finns det en funktion partition för att dela upp exempelfallen i partitioner baserade på ett givet attribut.

Klassen som skapar ett beslutsträd utgående från exempelfallen är AbstractDecisionTreeNode. Den intressantaste funktionen är induceTree, för här sker själva induktionen i beslutsträdet. Funktionen hanterar fyra olika möjligheter som kan ske. Det första är ifall alla element i delträdet är av samma kategori, en

lövnod skapas, det vill säga en nod som saknar barn. Den andra och tredje möjligheten sker ifall det inte finns tillräckligt med information för att kategorisera elementen i delträdet, då skapas en lövnod som saknar kategori. Den fjärde och sista möjligheten utför de rekursiva stegen som bygger upp trädet. Hur trädet kommer att delas är baserat på hur underklasser definierar metoden evaluatePartitionQuality.

Huvudidén bakom ID3-algoritmen dyker upp i InformationTheoreticDecisionTreeNode, närmare bestämt i den nu implementerade funktionen evaluatePartitionQuality, metoden använder sig av informationsteori för att räkna ut vilket attribut som innehåller mest information, det vill säga det attribut som delar det aktuella trädet eller delträdet på bästa möjliga sätt. Teorin om hur matematiken bakom algoritmen fungerar tas upp i George Lugers bok [5]. Hur det fungerar i praktiken visas i evaluatePartitionQuality, som finns i Bilaga 2. Hur det slutliga beslutsträdet, vilket även är det optimala beslutsträdet, kommer att representeras visas i figur 9.



Figur 9. Beslutsträd genererat med ID3 algoritmen.[5]

7. Avslutning

Genom åren har det funnits två huvudområden för forskningen inom artificiell intelligens, maskininlärning, som togs upp tidigare i avhandlingen, och förståelsen av naturliga språk. De här områdena har fungerat som mål, utmaningar och milstenar inom forskningen. Orsaken till varför det är svårt att återskapa inlärning och förståelse av naturliga språk är att de omfattar många andra mänskliga förmågor. Men teknologin går alltid framåt och i början av 2011 lyckades IBM producera en superdator, döpt till Watson efter IBM:s grundare, som lyckades slå två av de bästa Jeopardyspelarna i en två dagar lång match. Watson var inte kopplad till internet utan använde sig av sofistikerade algoritmer och totalt fyra terabyte färdigt inlagrade data, däribland Wikipedia i sin helhet [16]. Mjukvaran bakom Watson är skriven i både Java och C++ [17], båda språken är objektorienterade samt använder sig av liknande designmönster och programmeringsstil. Watsons framgång visar att Java, och andra objektorienterade språk, lämpar sig för att programmera artificiella intelligenser. Watson använder även Apaches Hadoop ramverk för det distribuerade datornätverket som bygger upp Watsons hårdvara.

En dator som spelar Jeopardy är dock inte huvudtanken bakom Watson, utan spelet användes mera för att visa att det är möjligt för en maskin att förstå naturliga språk. IBM säger själva att målet med Watson är en maskin som förstår frågor som en människa kan ställa och återger svar på ett sådant sätt som människor kan förstå, med andra ord i ett naturligt språk [18]. Programmet som gör Watson ska även säljas som ett expertsystem för läkare för att hjälpa dem med att diagnostisera och behandla svåra medicinska fall.

Datorer blir bättre på att utföra beräkningar och nya milstenar för artificiell intelligens kommer alltid att sättas upp. Men de artificiella intelligenser som science fiction-författarna har drömt om det senaste århundradet är ändå flera decennier borta, men en sak är dock säker: att forskningen inom området kommer att fortsätta att hela tiden gå framåt.

8. Källor

[1] Nationalencyklopedin, Intelligens. http://www.ne.se/lang/intelligens, Hämtad 30.3.2011

[2] C. Bitter, D.A. Elizondo, Y. Yang, ”Natural language processing: a prolog perspective”, Artificial Intelligence Review, Volym 33, Nummer 1-2, s:151-173, http://www.springerlink.com.ezproxy.vasa.abo.fi/content/j63ul26g04802564/fulltext.pdf Hämtad 30.3.2011

[3] John McCarthy, ”What is Artificial Intelligence? – Basic Questions” 12.11.2007 http://www-formal.stanford.edu/jmc/whatisai/node1.html Hämtad 30.3.2011

[4] Tiobe Software: Tiobe Index, Mars 2011
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
Hämtad 30.3.2011

[5] George Luger, ”Artificial Intelligence Structures and strategies for complex problem solving” (Sjätte upplagan), Pearson Education Inc, 2009.

[6] George Luger, William Stubblefield, “AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java”, Pearson Education Inc, 2009.

[7] Mark Watson, ”Practical Artificial Intelligence Programming with Java” (Tredje upplagan), 11.11.2008. http://www.markwatson.com/opencontent/JavaAI3rd.pdf
Hämtad 30.3.2011

[8] Vasileios Vasilikos, Michail G. Lagoudakis, ”Optimization of heuristic search using recursive algorithm selection and reinforcement learning”, Springer Netherlands, Annals of Mathematics and Artificial Intelligence, Publicerat på internet 9.12.2010 http://www.springerlink.com/content/j8p1x127590wt607/fulltext.pdf
Hämtad 1.4.2010

[9] California State University Dominguez Hills, Computer Science Department, CSC 411: Artificial Intelligence (Fall 2006)
http://www.csc.csudh.edu/jhan/Fall2006/csc411/Assignments/A2Fall06Solution.htm Hämtad 24.2.2011

[10] Ariel Felner, ”Finding optimal solutions to the graph partitioning problem with heuristic search”, Springer Netherlands, Annals of Mathematics and Artificial Intelligence, Volym 45, Nummer 3-4, s:293-322,
http://www.springerlink.com/content/p7342945518p1u68/fulltext.pdf
Hämtad 1.4.2011

[11] Herbert Simon, ”Why should machines learn?” 19.8.1980
http://www-edlab.cs.umass.edu/cs689/2010-lectures/simon-why-should-machines-learn.pdf Hämtad 5.3.2011

[12] Isabelle Guyon, ”Introduction to Machine Learning” Videoföreläsning publicerad på internet 2.7.2007
http://videolectures.net/bootcamp07_guyon_itml/ Hämtad 5.3.2011

[13] Xiaojin Zhu, ”Semi-Supervised Learning Literature Survey” 19.7.2008
http://www.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf Hämtad 5.3.2011

[14] Rich Sutton, ”Reinforced Learning Frequently Asked Questions” 4.2.2004
http://webdocs.cs.ualberta.ca/~sutton/RL-FAQ.html Hämtad 5.3.2011

[15] J.R. Quinlan, ”Induction of Decision Trees” 1986
http://impact.asu.edu/cse494sp09/Induction%20of%20Decision%20Trees.pdf
Hämtad 17.3.2011

[16] IBM Watson, http://www-943.ibm.com/innovation/us/watson/
Hämtad 1.4.2011

[17] Dean Takahashi, "IBM researcher explains what Watson gets right and wrong (video)" 17.2.2011 http://venturebeat.com/2011/02/17/ibm-researcher-explains-what-watson-gets-right-and-wrong/ Hämtad 4.4.2011

[18] Jon Brodkin, "IBM's Jeopardy-playing machine can now beat human contestants" 10.2.2010 http://www.networkworld.com/news/2010/021010-ibm-jeopardy-game.html?page=2 Hämtad 29.3.2011

9. Bilaga 1

Alla implementerade klasser för exemplen för sökning. Klassdiagrammet finns i kapitel 5.

```
public interface State {

	public Iterable<State> getPossibleMoves();
	public boolean isSolution();
	public double getHeuristics();
	public double getDistance();
	public State getParent();
}
```

```
public abstract class AbstractState implements State {

	private State parent = null;
	private double distance = 0;

	public AbstractState() {}
	public AbstractState(State _parent){
		this.parent = _parent;
		this.distance = _parent.getDistance() +1;
	}
	public State getParent(){
		return parent;
	}
	public double getDistance() {
		return distance;
	}
}
```

```
public interface Solver {
	public List<State> solve(State initialState);
}
```

```
public abstract class AbstractSolver implements Solver {

	private Set<State> closed = new HashSet<State>();

	public List<State> solve(State initialState) {
		closed.clear();
		clearOpen();
		addStateToOpen(initialState);
		while(openHasElements()){
			State s = nextStateFromOpen();
			if(s.isSolution())
				return findPath(s);
```

```
                closed.add(s);
                Iterable<State> moves = s.getPossibleMoves();
                for(State move : moves)
                    if (!closed.contains(move))
                        addStateToOpen(move);
            }
            return null;
        }
        private List<State> findPath(State solution) {

            LinkedList<State> path = new LinkedList<State>();
            while (solution != null){
                path.addFirst(solution);
                solution = solution.getParent();
            }
            return path;
        }
        protected abstract State nextStateFromOpen();
        protected abstract void addStateToOpen(State initialState);
        protected abstract boolean openHasElements();
        protected abstract void clearOpen();
}
```

```
public class DepthFirstSolver extends AbstractSolver {

        private Stack<State> stack;

        public DepthFirstSolver(){
            stack = new Stack<State>();
        }

        protected State nextStateFromOpen() {
            return stack.pop();
        }
        protected void addStateToOpen(State s) {
            if(!stack.contains(s))stack.push(s);
        }
        protected boolean openHasElements() {
            return !stack.empty();
        }
        protected void clearOpen() {
            stack.clear();
        }
}
public class BreadthFirstSolver extends AbstractSolver {

        private Queue<State> queue;

        public BreadthFirstSolver(){
            queue = new LinkedList<State>();
        }

        protected State nextStateFromOpen() {
            return queue.remove();
```

```java
	}
	protected void addStateToOpen(State s) {
		if(!queue.contains(s)) queue.offer(s);
	}
	protected boolean openHasElements() {
		return !queue.isEmpty();
	}
	protected void clearOpen() {
		queue.clear();
	}
}

public class BestFirstSolver extends BreadthFirstSolver {

	private PriorityQueue<State> queue;

	public BestFirstSolver(){
		queue = new PriorityQueue<State>(1,
		new Comparator<State>(){
			public int compare(State s1, State s2){
				return Double.compare(
					s1.getDistance() + s1.getHeuristic(),
					s2.getDistance() + s2.getHeuristic());
			}
		});
	}
}



public class EightPuzzleState extends AbstractState {
	int rowIndex;
	int columnIndex;
	int[][] state = null;
	public EightPuzzleState(State parent, int[][] state){
		super(parent);
		int j = Arrays.asList(state).indexOf(0);
		rowIndex = j/3;
		columnIndex = j%3;
		this.state = state;
	}

	public Iterable<State> getPossibleMoves() {
		LinkedList<State> retur = new LinkedList<State>();
		int[][] newState = state.clone();
		switch(rowIndex){
		case 0:
			newState[rowIndex][columnIndex] =
					newState[rowIndex+1][columnIndex];
			newState[rowIndex+1][columnIndex] = 0;
			retur.add(new EightPuzzleState(this, newState));
			break;
		case 1:
			newState[rowIndex][columnIndex] =
					newState[rowIndex+1][columnIndex];
			newState[rowIndex+1][columnIndex] = 0;
```

```java
                retur.add(new EightPuzzleState(this, newState));
            case 2:
                newState = state.clone();
                newState[rowIndex][columnIndex] =
                            newState[rowIndex-1][columnIndex];
                newState[rowIndex-1][columnIndex] = 0;
                retur.add(new EightPuzzleState(this, newState));
            }
            newState = state.clone();
            switch(columnIndex){
            case 0:
                newState[rowIndex][columnIndex] =
                            newState[rowIndex][columnIndex+1];
                newState[rowIndex][columnIndex+1] = 0;
                retur.add(new EightPuzzleState(this, newState));
                break;
            case 1:
                newState[rowIndex][columnIndex] =
                            newState[rowIndex][columnIndex+1];
                newState[rowIndex][columnIndex+1] = 0;
                retur.add(new EightPuzzleState(this, newState));
            case 2:
                newState = state.clone();
                newState[rowIndex][columnIndex] =
                            newState[rowIndex][columnIndex-1];
                newState[rowIndex][columnIndex-1] = 0;
                retur.add(new EightPuzzleState(this, newState));
            }
            return retur;
        }

        public double getHeuristic() {
            double heuristic = 0;
            if(state[0][0] != 1) heuristic = (heuristic + 1.0);
            if(state[0][1] != 2) heuristic = (heuristic + 1.0);
            if(state[0][2] != 3) heuristic = (heuristic + 1.0);
            if(state[1][0] != 4) heuristic = (heuristic + 1.0);
            if(state[1][1] != 5) heuristic = (heuristic + 1.0);
            if(state[1][2] != 6) heuristic = (heuristic + 1.0);
            if(state[2][0] != 7) heuristic = (heuristic + 1.0);
            if(state[2][1] != 8) heuristic = (heuristic + 1.0);
            return heuristic;
        }

        public boolean isSolution() {
            if (this.getHeuristic() == 0) return true;
            else return false;
        }
}
```

10. Bilaga 2

```
public abstract class AbstractProperty {
	private String value = null;
	public AbstractProperty(String _value)
		throws IllegalArgumentException{
			if(isLegalValue(_value) == false)
				throw new IllegalArgumentException(_value +
					" is an illegal value for Property " +
					getName());
			this.value = _value;
	}
	public final String getValue(){
		return value;
	}
	public abstract boolean isLegalValue(String value);
	public abstract String getName();
}

public class DebtProperty extends AbstractProperty {
	public static final String DEBT = "Debt";
	public static final String HIGH = "high";
	public static final String LOW = "low";
	public DebtProperty(String value){
		super(value);
	}
	public boolean isLegalValue(String value) {
		return(value.equals(HIGH)|| value.equals(LOW));
	}
	public String getName() {
		return DEBT;
	}
}



public abstract class AbstractExample {
	private String category = null;
	private Map<String,AbstractProperty> properties = new
					HashMap<String,AbstractProperty>();

	public AbstractExample(String category, AbstractProperty...
					propertyList)
		throws IllegalArgumentException{
			if(isLegalCategory(category)==false)
				throw new IllegalArgumentException(
					category + " is an illegal category
					for example.");
			this.category = category;
			addProperties(propertyList);
	}
	private void addProperties(AbstractProperty[] propertyList)
		throws IllegalArgumentException{
```

```java
			Set<String> requiredProps = getPropertyNames();
			for(int i=0;i<propertyList.length;i++){
				AbstractProperty prop = propertyList[i];
				if(requiredProps.contains(prop.getName())
									== false)
					throw new IllegalArgumentException(
						prop.getName() +
						" illegal Property for example.");
				properties.put(prop.getName(), prop);
				requiredProps.remove(prop.getName());
			}
			if(requiredProps.isEmpty() ==false){
				Object[] p = requiredProps.toArray();
				String props = "";
				for(int i=0; i<p.length;i++)
					props += (String)p[i]+ " ";
				throw
					new IllegalArgumentException("Missing
						Properties in example: " + props);
			}
		}
		public AbstractProperty getProperty(String name){
			return properties.get(name);
		}
		public String getCategory(){
			return category;
		}
		public abstract Set<String> getPropertyNames();
		protected abstract boolean isLegalCategory(String category);
}



public class ExampleSet {
	private Vector<AbstractExample> examples = new
		Vector<AbstractExample>();
	private HashSet<String> categories = new HashSet<String>();
	private Set<String> propertyNames = null;
	public void addExample(AbstractExample e)
		throws IllegalArgumentException{
		if(e.getCategory() == null)
			throw new IllegalArgumentException("Example
				missing categorization.");
		if ((examples.isEmpty()) || e.getClass() ==
				examples.firstElement().getClass()){
			examples.add(e);
			categories.add(e.getCategory());
			if (propertyNames == null)
				propertyNames = new
					HashSet<String>(e.getPropertyNames());
		}
		else
			throw new IllegalArgumentException("All examples
				must be same type");
	}
	public int getSize(){
```

```
        return examples.size();
    }
    public boolean isEmpty(){
        return examples.isEmpty();
    }
    public AbstractExample getExample(int i){
        return examples.get(i);
    }
    public Set<String> getCategories(){
        return new HashSet<String>(categories);
    }
    public Set<String> getPropertyNames(){
        return new HashSet<String>(propertyNames);
    }
    public int getExampleCountByCategory(String cat)
        throws IllegalArgumentException{
        Iterator<AbstractExample> iter = examples.iterator();
        AbstractExample example;
        int count = 0;
        while(iter.hasNext()){
            example = iter.next();
            if(example.getCategory().equals(cat)) count++;
        }
        return count;
    }
    public HashMap<String, ExampleSet> partition(
                                    String propertyName)
        throws IllegalArgumentException{
        HashMap<String, ExampleSet> partition = new
            HashMap<String, ExampleSet>();
        Set<String> values = getValues(propertyName);
        Iterator<String> iter = values.iterator();
        while(iter.hasNext()){
            String val = iter.next();
            ExampleSet examples =
                getExamplesByProperty(propertyName, val);
            partition.put(val, examples);
        }
        return partition;
    }
    private Set<String> getValues(String propName){
        HashSet<String> values = new HashSet<String>();
        Iterator<AbstractExample> iter = examples.iterator();
        while(iter.hasNext()){
            AbstractExample ex = iter.next();
            values.add(ex.getProperty(propName).getValue());
        }
        return values;
    }
    private ExampleSet getExamplesByProperty(
            String propName, String value)
        throws IllegalArgumentException{
        ExampleSet result = new ExampleSet();
        Iterator<AbstractExample> iter = examples.iterator();
        AbstractExample example;
        while(iter.hasNext()){
            example = iter.next();
```

```
	if(example.getProperty(propName).getValue().equals(value))
				result.addExample(example);
		}
		return result;
	}
}



public abstract class AbstractDecisionTreeNode{
	private String category = null;
	private String decisionPropertyName = null;
	private HashMap<String,AbstractDecisionTreeNode>
		children = new HashMap<String,
				AbstractDecisionTreeNode>();
	public AbstractDecisionTreeNode( ExampleSet examples)
		throws IllegalArgumentException{
		induceTree(examples, examples.getPropertyNames());
	}
	protected AbstractDecisionTreeNode(
			ExampleSet examples,
			Set<String> selectionProperties)
			throws IllegalArgumentException{
		induceTree(examples, selectionProperties);
	}
	public boolean isLeaf(){
		return children.isEmpty();
	}
	public String getCategory(){
		return category;
	}
	public String getDecisionProperty(){
		return decisionPropertyName;
	}
	public AbstractDecisionTreeNode getChild(
		String propertyValue){
		return children.get(propertyValue);
	}
	public void addChild(String propertyValue,
			AbstractDecisionTreeNode child){
		children.put(propertyValue, child);
	}
	public String categorize(AbstractExample ex){
		if(children.isEmpty()) return category;
		if(decisionPropertyName == null) return category;
		AbstractProperty prop =
			ex.getProperty(decisionPropertyName);
		AbstractDecisionTreeNode child =
			children.get(prop.getValue());
		if(child == null) return null;
		else return child.categorize(ex);
	}
	public void induceTree(ExampleSet examples,
		Set<String> selectionProperties)
		throws IllegalArgumentException{
```

```
            //Alla instanser är av samma kategori, noden är ett löv
            if(examples.getCategories().size() == 1){
                  category =
                        examples.getCategories().iterator().next();
                  return;
            }
            //Mängden av exempel är tom. Skapa ett löv utan kategori
            if(examples.isEmpty()) return;
            //Mänden av attribut är tom, går inte att klassificera
            if(selectionProperties.isEmpty()) return;

            Iterator<String> iter = selectionProperties.iterator();
            String bestPropertyName = iter.next();
            HashMap<String, ExampleSet> bestPartition =
                  examples.partition(bestPropertyName);
            double bestPartitionEvaluation =
                  evaluatePartitionQuality(bestPartition, examples);
            while(iter.hasNext()){
                  String nextProp = iter.next();
                  HashMap<String, ExampleSet> nextPart =
                        examples.partition(nextProp);
                  double nextPartitionEvaluation =
                       evaluatePartitionQuality(nextPart, examples);
                  if(nextPartitionEvaluation >
                        bestPartitionEvaluation){
                        bestPartitionEvaluation =
                              nextPartitionEvaluation;
                        bestPartition = nextPart;
                        bestPropertyName = nextProp;
                  }
            }
            this.decisionPropertyName = bestPropertyName;
            Set<String> newSelectionPropSet = new
                  HashSet<String>(selectionProperties);
            newSelectionPropSet.remove(decisionPropertyName);
            iter = bestPartition.keySet().iterator();
            while(iter.hasNext()){
                  String value = iter.next();
                  ExampleSet child = bestPartition.get(value);
                  children.put(value, createChildNode(child,
                        newSelectionPropSet));
            }
      }
      protected abstract double evaluatePartitionQuality(
            HashMap<String, ExampleSet> part, ExampleSet examples)
            throws IllegalArgumentException;
      protected abstract AbstractDecisionTreeNode createChildNode(
            ExampleSet examples, Set<String> selectionProperties)
            throws IllegalArgumentException;
}



public class InformationTheoreticDecisionTreeNode extends
            AbstractDecisionTreeNode {
```

```
	public InformationTheoreticDecisionTreeNode(
			ExampleSet examples)
			throws IllegalArgumentException {
		super(examples);
	}
	public InformationTheoreticDecisionTreeNode(
		ExampleSet examples, Set<String> selectionProperties)
		throws IllegalArgumentException {
		super(examples, selectionProperties);
	}
	protected AbstractDecisionTreeNode createChildNode(
		ExampleSet examples, Set<String> selectionProperties)
		throws IllegalArgumentException {
		return new InformationTheoreticDecisionTreeNode(
			examples, selectionProperties);
	}

	protected double evaluatePartitionQuality(
		HashMap<String, ExampleSet> part, ExampleSet examples)
			throws IllegalArgumentException {
		double examplesInfo = computeInformation(examples);
		int totalSize = examples.getSize();
		double expectedInfo = 0.0;
		Iterator<String> iter = part.keySet().iterator();
		while(iter.hasNext()){
			ExampleSet ex = part.get(iter.next());
			int partSize = ex.getSize();
			expectedInfo +=
				computeInformation(ex)*partSize/totalSize;
		}
		return examplesInfo - expectedInfo;
	}
	private double computeInformation( ExampleSet examples)
		throws IllegalArgumentException{
		Set<String> categories = examples.getCategories();
		double info = 0.0;
		double totalCount = examples.getSize();
		Iterator<String> iter = categories.iterator();
		while(iter.hasNext()){
			String cat = iter.next();
			double catCount =
				examples.getExampleCountByCategory(cat);
			info += -
			(catCount/totalCount)*log2(catCount/totalCount);
		}
		return info;
	}
	private double log2(double a){
		return Math.log10(a)/Math.log10(2);
	}
}
```