Inlämningsuppgift #2: Regelbaserade system och minimax

Deadline: 240920 klockan 23:59

Introduktion

Inlämningsuppgiften har tre separata delar

1) Konstruktion av ett förenklat regelbaserat system (expertsystem)
2) Implementering av minimax-algoritmen
3) Formalisering i predikatlogik

Del 1 och Del 2 har en uppgift vardera för betyget 3 och för betyget 5. Uppgiften i Del 3 är bara för betyget 5.

Del 1: Implementationen av regelbaserat system

Uppgiften går ut på att utveckla ett förenklat expertsystem. För betyget 3 skall forward chaining implementeras, och för betyget 5 skall även backward chaining läggas till.

I den utdelade koden finns systemets arkitektur beskrivet med följande typdeklarationer:

```
type Fact = String
type Premise = [Fact]
type Conclusion = [Fact]
type Rule = (Premise,Conclusion)
type RuleSet = [Rule]
type WM = [Fact]
```

Systemets fakta utgörs alltså av strängar. Såväl reglernas premisser som slutsatser är därmed listor av fakta. Själva regelmängden blir en lista av regler. Systemets arbetsminne (WM) är en lista med kända fakta. I startkoden finns exempel på en regelmängd och ett arbetsminne:

```
startWM :: WM
startWM = ["a","e"]

rules :: RuleSet
rules = [(["a","b"],["c"]), (["c"],["d"]), (["g"],["h"]),
(["a","d"],["f","g"]),(["e"],["d"])]
```

Innebörden är förstås att i det här exemplet är ”a” och ”e” kända, och att vi har totalt fem regler i systemet. Notera att systemet bara tillåter konjunktioner i premisserna, men att även

slutsatserna kan bestå av (konjunktioner av) mer än ett faktum. Den första regeln kan alltså utläsas som ”IF a and b THEN c”. På samma sätt blir regel 4 ”IF a and d THEN f and g”.

Framåtlänkat system (betyg 3)

Den övergripande principen vid forward chaining är att i varje iteration prövas samtliga regler. För var och en av reglerna kontrolleras dess premiss, och om den är sann läggs alla fakta i slutsatsen till i arbetsminnet. Detta upprepas tills en hel genomgång av regelmängden inte lett till någon förändring i arbetsminnet.

I den utdelade koden finns ett embryo till forward chaining:

```
runForward :: WM
runForward = forwardChaining (onePass startWM rules) startWM rules

forwardChaining:: WM -> WM -> RuleSet -> WM
forwardChaining  newWM oldWM rules = if newWM == oldWM then oldWM
    else forwardChaining (onePass newWM rules) newWM rules
```

Körningen startas alltså med runForward, varifrån funktionen forwardChaining anropas med ett anrop på funktionen onePass som tar arbetsminnet och regelmängden som argument, samt ytterligare en kopia på arbetsminnet och regelmängden. onePass går igenom samtliga regler en gång, och uppdaterar arbetsminnet. Första raden av funktionen forwardChaining kontrollerar sedan om newWM (alltså resultatet av körning av funktionen onePass) är identisk med gamla arbetsminnet, vilket innebär att inga nya fakta kunde härledas, varvid körningen avslutas. Om inte görs ett nytt anrop på onePass.

Det enda som återstår att implementera är därmed funktionen onePass, vilken alltså skall gå igenom samtliga regler i regelmängden, och lägga till de reglers slutsatser vars premisser är uppfyllda. En funktion som kan vara användbar för detta är subSet, vilken finns med i den utdelade koden. TIPS: funktionen onePass kan utan problem göras utan användning av varken högre ordningens funktioner eller skapande av hjälpfunktioner. Om ni upptäcker att ert program börjar växa är ni därmed fel ute!

När ni väl kontrollerat att programmet fungerar för den bifogade regelmängden skall ni skapa en egen regelmängd där ni använder beskrivande termer i stället för enstaka bokstäver. Den valda tillämpningen måste absolut inte vara seriös, utan får gärna vara lite rolig. Minimum antal regler är 10.

Bakåtlänkat system (betyg 5)

För betyget 5 skall systemet utökas med bakåtlänkning, dvs. man skall kunna fråga systemet om det kan härleda ett visst faktum. (Självklart måste detta göras utan användning av den framåtlänkade delen, annars blir ju uppgiften trivial, eller hur?)

Som bekant fungerar bakåtlänkning enligt följande: om vi vill visa ett visst faktum, som inte återfinns i arbetsminnet, hittas först samtliga regler som har faktumet som en del av

slutsatsen. Slutsatsen är sedan sann om någon av dessa reglers premisser är sanna, eller kan härledas. Signaturen blir därmed så klart:

```
backwardChaining :: RuleSet -> Fact -> WM -> Bool
```

Funktionen backwardChaining kan skrivas relativt kompakt, och utan användning av ytterligare egentillverkade funktioner. Det är dock absolut tillåtet att bryta ner funktionen något, och då specifikt implementera ett fåtal extra funktioner, om det ökar programmets läsbarhet. Återigen måste ni dock undvika att programmet blir alltför stort. TIPS: De inbyggda funktionerna *any* och *all* kan vara nyttiga.

En mycket viktig sak är att ni måste undvika att ert program fastnar i cykler. Om vi t. ex. försöker bevisa ”b” från följande regelmängd och arbetsminne:

```
startWM = ["a","e"]
rules = [([“c”],[“b”]), ([“b”],[“c”])]
```

skall detta misslyckas, dvs. programmet får inte fastna i en oändlig rekursion. På samma sätt måste programmet lyckas att härleda ”b” från följande regelmängd och arbetsminne:

```
startWM = ["a","e"]
rules = [([“c”],[“b”]), ([“b”],[“c”]), ([“e”],[“b”])]
```

Del 2 – minimax

I den här delen får valfritt programspråk användas. Rekommendationen är dock att använda Haskell, och beskrivningen nedan utgår från utdelad Haskellkod. Det går dock absolut lika bra att välja ett annat programspråk, men då måste ni så klart även skapa träden själva i det språket.

Vi kommer i den här inlämningsuppgiften inte att använda minimax i något verkligt spel, utan bara implementera själva sökningen. Vad det gäller spelträden gör vi tre avgränsningar:

1) Spelarna har i varje drag **exakt två alternativ**, dvs. trädet är binärt
2) Alla trädens löv befinner sig på **samma avstånd från roten**
3) Sista nivån i trädet är **full**

I klartext innebär detta att om vi t. ex. skall titta tre drag framåt kommer vårt träd att ha 8 löv. Eller, mer generellt, om vi tittar n drag framåt har trädet 2^n löv.

Om vi betraktar utdelad Haskell-kod hittar vi först följande deklaration av en datatyp träd:

```
data Tree a = Leaf Int | Node (Tree a) (Tree a) deriving (Show)
```

Det här skapar en rekursiv datatyp där ett träd antingen är ett löv (Leaf) och då innehåller ett heltal, eller en inre nod (Node) som består av två träd. I den här formen av träd finns det alltså bara information i löven, vilket passar utmärkt för vår tillämpning. Funktioner som arbetar på träd är naturligt rekursiva, och består typiskt av ett basfall (för löven) och ett grenat (sic!) rekursivt anrop. Innan vi går vidare med minimax ger vi några exempel så vi blir bekväma med träd i Haskell. (Dessa exempel har alltså inget med själva inlämningsuppgiften att göra!) Låt oss börja med att se hur dylika träd representeras internt:

```
Node (Leaf 1) (Leaf 2)
Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))
```

Första raden motsvarar ett träd med endast en nod (roten) och sedan två löv med värdena 1 och 2. Andra raden är ett träd med ytterligare en nivå, och löven har värdena 1, 2, 3, 4.

Vi kan så klart definiera en stor mängd funktioner som arbetar på träd. Här är två exempel:

```
treeSum :: Tree Int -> Int
treeSum (Leaf x) = x
treeSum (Node left right) = treeSum left + treeSum right

maxValue :: Tree Int -> Int
maxValue (Leaf x) = x
maxValue (Node left right) = max (maxValue left) (maxValue right)
```

Den första funktionen summerar värdena i samtliga löv, medan den andra hittar det största värdet i trädet.

I den utdelade koden finns två funktioner som skapar träd på den ovan beskrivna formen från en heltalslista. Listan skall innehålla lövens värde i ordning från vänster till höger i trädet. Notera att listans längd alltså måste vara någon av 2, 4, 8, 16, 32 etc. Som ett exempel ger anropet:

```
generateTree ([3,1,6,8,2,5,4,7])

Node (Node (Node (Leaf 3) (Leaf 1)) (Node (Leaf 6) (Leaf 8)))
(Node (Node (Leaf 2) (Leaf 5)) (Node (Leaf 4) (Leaf 7)))
```

Efter alla dessa preludier är vi nu redo att ta oss an en implementering av minimax!

minimax (betyg 3)

Uppgiften innebär helt enkelt att koda funktionen minimax, vilken utgår från att vartannat drag görs av en spelare som vill **maximera** värdet, och vartannat drag görs av en spelare som vill **minimera** värdet. Observera att **det första draget alltid görs av spelaren som vill maximera**. minimax tar som argument förstås ett spelträd, och skall returnera spelets värde.

Även denna lösning bör bli extremt kompakt. Självklart bör ni kontrollera er lösning genom att skicka in ett antal träd där ni vet vad rätt svar är.

minimax med alfa-beta klippning (betyg 5)

Här skall ni utöka er implementering av minimax så den utnyttjar alfa-beta klippning. Specifikt skall er kod returnera inte bara spelets värde, utan även antalet löv som klippts bort! I en Haskell-lösning blir returtypen därmed (Int, Int) där första heltalet motsvarar spelets värde, och andra heltalet antalet klippta löv. TIPS: Håll koll på vilken nivå ni är i trädet så blir det enkelt att räkna ut hur många löv som försvinner vid ett visst klipp.

Lösningar i andra språk än Haskell

Era program skall så klart ha motsvarande funktionalitet men ni måste alltså då även skapa era spelträd (på samma format) själva. Ni behöver även ha någon form av I/O- hantering, exempelvis bör förstås resultaten från kröningarna skrivas ut på lämpligt sätt.

Del 3 – Formalisering i predikatlogik

Betrakta följande kunskapsbas, uttryckt i naturligt språk.

> Alla genier är galna. Det är inte så att alla som är galna är genier. Einstein var ett geni, liksom Turing. Alla som är före sin tid eller galna är missförstådda. Både Einstein och Turing var före sin tid. Turing var inte nobelpristagare, men det var Einstein. Det finns ett pris som är uppkallat efter Turing. Einstein har ett pris uppkallat efter sig. De som är galna och missförstådda är lyckliga om de både är nobelpristagare och har ett pris uppkallat efter sig.

Uppgift 3a
Översätt kunskapsbasen ovan till predikatlogik. **Använd exakt en sats i predikatlogik per svenska mening, samt följande predikat, och inga andra**:

Unära (1-ställiga): geni, galen, föreSinTid, missförstådd, nobelpristagare, pris, lycklig
Binära (2-ställiga): uppkallatEfter

Uppgift 3b
Konvertera samtliga predikatlogiska satser till konjunktiv normalform (CNF), enligt de arbetssteg som presenterats på föreläsning samt i avsnitt 9.5.1 i Russel & Norvig.

Uppgift 3c
Bevisa, med hjälp av motsägelsebevis och resolution, att Einstein var lycklig.

Uppgift 3d (frivillig extrauppgift, behöver inte lösas för Betyg 5)
Kan man bevisa att Turing är lycklig eller olycklig? Varför (inte)?

Organisation

Laborationen skall lösas i grupper om 3-4 studenter. Lämna in laborationen från gruppen i Canvas. Utöver detta skall lösningen demonstreras för examinerande lärare. Tid för detta bokas på Canvas.

Notera att för betyget 5 måste **även** uppgifterna för Betyg 3 lösas. Inlämningsuppgiften är godkänd om uppgifterna för Betyg 3 i Del 1 och Del 2 lösts på ett tillfredsställande sätt.

Inlämningen skall bestå av:
1. Väl strukturerad källkod
2. Väl valda körningsexempel där ni visar resultatet av att köra era program
3. En lösning på Del 3 kan absolut vara handskriven och inskannad/fotograferad, men den måste vara tydlig och lättläst.

På Canvas finns ett diskussionsforum kopplat till inlämningsuppgiften där ni kan ställa generella frågor så att alla får samma information. Notera att ingen handledning kommer ges, ej heller via e-mail eller liknande. Slutligen, kom ihåg att samarbeten mellan grupper är förbjudet då uppgiften utgör del av examinationen.

Lycka till!