AI-programmering

Inlämningsuppgift #1: GPS och A*

Deadline: 240913 klockan 23:59

Introduktion

Inlämningsuppgiften behandlar sökning – såväl oinformerad som informerad – samt funktionell programmering. Uppgiften utgår från en given (och något förenklad) implementation i Haskell av programmet General Problem Solver (GPS), utvecklat av Simon, Shaw och Newell 1957(!).

Vi kommer nedan att gå igenom den utdelade Haskell-koden, vilken består av en generell del (själva implementeringen av GPS) och en problemspecifik del. I det bifogade exemplet löses det så kallade hink-problemet (water jugs).

Inlämningsuppgiften har två delar:

1. För betyget 3 skall den problemspecifika delen anpassas så programmet löser problemet Solitär, se mer nedan.
2. För betyget 5 skall programmet även utökas till att implementera A*-algoritmen och då lösa 8-pusslet, se mer nedan.

Innan vi övergår till den utdelade koden är det viktigt att påpeka att vissa delar av lösningen är mindre effektiva eller generella än vad som är möjligt i Haskell. Anledningen är helt enkelt en önskan att hålla koden någorlunda enkel och lättläst. Specifikt hade t. ex. en konsekvent användning av typen Maybe givit en elegantare (och mer Haskellesque) lösning.

Implementationen av GPS

Programmet består alltså av en generisk del och en problemspecifik del. Principen är förstås att om programmet skall användas för att lösa andra problem än exemplet water jugs, så skall alla förändringar ske i den problemspecifika delen.

På högnivå arbetar GPS enligt följande:

1. Put the start state S in an empty list called OPEN. OPEN contains nodes for expansion.
2. Create an empty list CLOSED which contains already expanded nodes.
3. If OPEN is empty, finish with failure.
4. Select the first node N in OPEN. Remove N from OPEN and put it in CLOSED.
5. If N represents a goal state, finish with success. (We’ve found a way from S to N)

6. Expand N, i.e., use all applicable rules to find a set, NEW, of nodes that we can reach from N.
7. Add the nodes in NEW that are not already in OPEN or CLOSED, to OPEN.
8. GoTo 3

Algoritmens sökstrategi bestäms av hur sammanslagningen av OPEN och NEW sker. I den utdelade koden läggs de nya tillstånden sist i OPEN, vilket innebär att sökningen sker bredden först, dvs. när en lösning väl hittas kommer den alltid att vara den kortast möjliga.

Den utdelade koden följer beskrivningen ovan, men måste även hantera en detalj som är bortabstraherad i högnivå-beskrivningen ovan, nämligen hur man återskapar vägen (lösningen i form av de olika operationerna) från CLOSED-listan. Enkelt uttryckt sker detta genom att varje tillstånd håller reda på föregående tillstånd, sin ”förälder” (parent), och så avslutas algoritmen med att följa dessa länkar bakåt från mål till start. Vi får därmed som lösningen en sekvens av tillstånd från start till mål.

Givet detta är det kanske enklast att först titta på några problemspecifika delar. Vi har diskuterat water jugs på föreläsningen, men här är en repetition:

Du har två omärkta hinkar där den stora rymmer 4 liter och den lilla 3 liter. Du har även tillgång till en pump med vatten. Uppgiften är att få exakt två liter vatten i den stora hinken.

Vi inför en typ (egentligen ett *typalias*) samt tre enkla funktioner:

```
type State = (Int,Int)

startState :: State
startState = (0,0)

startParent :: State
startParent = (-1,-1)

isGoal :: State -> Bool
isGoal (x,_) = x==2
```

Så, ett tillstånd består här av två heltal där det första är volymen vatten i stora hinken, och det andra volymen vatten i lilla hinken. Starttillståndet är `(0,0)` och vi måste ge `startParent` ett unikt värde, vi väljer `(-1,-1)`. Vi skapar också ett predikat `isGoal` som returnerar SANT för tillstånd där den stora hinken innehåller exakt två liter.

Om vi sedan övergår till den generella delen hittar vi följande:

```
type StatePair = (State,State)

run :: [State]
run = gps [(startState,startParent)] []
```

Vi har alltså en typ kallad `StatePair` vilken består av två tillstånd. Betydelsen är att det andra tillståndet är förälder till det första. Till själva `gps` skickas sedan OPEN (bestående av tillståndsparet som representerar starttillståndet) samt en tom lista som motsvarar CLOSED. `run` returnerar sedan en lista bestående av de tillstånd som utgör lösningen, från start till mål.

Vi är nu beredda att studera implementeringen av `gps`:

```
gps :: [StatePair]->[StatePair]->[State]

gps [] _ = error "No solution"

gps (s:restOfOpen) closed

 | isGoal (fst s) = reverse(retrieveSolution (fst s) (s:closed))

 | otherwise = gps newOpen (s:closed)

     where newOpen = restOfOpen ++ zip newStates (repeat (fst s))

           newStates = filter (`notElem` map fst (restOfOpen ++ closed)) (makeMoves (fst s))
```

Signaturen `gps :: [StatePair]->[StatePair]->[State]` säger att funktionen tar in två listor av tillståndspar (OPEN och CLOSED) och returnerar en lista av tillstånd (lösningen).

`gps [] _ = error "No solution"` innebär att om OPEN är tom så finns det ingen lösning och programmet avslutas. Annars finns det två alternativ: om första delen av första tillståndsparet `s` i OPEN (`s:restOfOpen`) alltså `fst s` är ett måltillstånd, vilket prövas med `isGoal (fst s)`, så har vi hittat en lösning och vi anropar `retrieveSolution` för att med hjälp av CLOSED återskapa sekvensen av tillstånd. Om `s` inte är ett måltillstånd skall tillståndet `fst s` expanderas, och alla de tillstånd som vi kan nå från `fst s`, men som inte redan finns i OPEN eller CLOSED, skall läggas till sist i OPEN. Detta ger oss `newOpen`, varefter `gps newOpen (s:closed)` anropas rekursivt.

Här är det värt att notera en viktig sak: `newStates`, alltså listan av de tillstånd som kan nås från `fst s`, är en lista av **tillstånd** `[State]` medan OPEN (och CLOSED) är listor av **tillståndspar** `[StatePair]`. Satsen `zip newStates (repeat (fst s))` skapar därför en lista av tillståndspar där första delen av varje par är ett tillstånd som kan nås från `fst s` och andra delen av varje par är just `fst s`. Det här är exakt vad vi vill när vi gör: `newOpen = restOfOpen ++ zip newStates (repeat (fst s))`. Vi har alltså helt enkelt lagt till alla upptäckta nya tillstånd, med `fst s` som parent sist i OPEN.

Det som återstår att gå igenom av `gps` är skapandet av `newStates`. Raden `newStates = filter (`notElem` map fst (restOfOpen ++ closed)) (makeMoves (fst s))` kan kanske vid första anblicken verka relativt komplicerad, men låt oss betrakta de olika delarna. Innan

vi analyserar koden påminner vi oss om vad vi vill göra: vi skall skapa en lista av de tillstånd som man kan nå från nuvarande tillståndet (i koden `fst s`) och sedan ta bort alla tillstånd som redan finns i OPEN och CLOSED. `(makeMoves (fst s))` skapar här först listan av tillstånd som vi kan nå från `fst s`, medan resten av raden tar bort tillstånd som finns i OPEN eller CLOSED. Borttagandet utnyttjar den inbyggda funktionen `filter`, vilken som bekant tar en Boolesk funktion och en lista som argument. Den Booleska funktionen `(`notElem` map fst (restOfOpen ++ closed))` blir något komplex då OPEN och CLOSED enligt tidigare innehåller tillståndspar, medan vi här arbetar med tillstånd. Rent praktiskt (och kanske ganska fult) skapar vi därför en ny lista av tillståndspar som kombinerar `restOfOpen` och `closed`, innan vi med `map fst (restOfOpen ++ closed)` därifrån skapar ytterligare en ny lista bestående av bara tillstånden, dvs. utan föräldrarna. `filter` använder sedan `` `notElem` `` på den listan för att avgöra vilka nya tillstånd i `(makeMoves (fst s))` som inte finns i OPEN eller CLOSED. Notera här användningen av infix-notation.

Vi flyttar nu fokus till nyckelfunktionen `makeMoves :: State -> [State]`, vilken alltså tar in ett visst tillstånd s och skall returnera en lista bestående av de tillstånd som kan nås från tillståndet s. Man kan här säga att `makeMoves` utgör ett gränssnitt mellan den generella delen och den problemspecifika; `gps` **kräver** att det finns en dylik funktion med den exakta signaturen. `gps` ställer däremot inga krav på **hur** `makeMoves` är implementerad, eller ens hur tillstånd representeras för det specifika problemet.

I det aktuella fallet har vi valt en semi-generell lösning för `makeMoves`; vi representerar problemets operatorer med ett antal regler, implementerade som funktioner, vilka vi även lagt i en lista. Varje regel har typen `State->State`, alltså de tar in ett tillstånd och returnerar ett annat tillstånd. Här använder vi oss av ett hack som innebär att om regeln inte är möjlig så returneras samma tillstånd. Nedan ser vi två operationer uttryckta som regler:

```
fillLarge :: State->State
fillLarge (x,y) = if x < 4 then (4,y) else (x,y)

smallInLargeTillFull :: State->State
smallInLargeTillFull (x,y) = if x+y >= 4 && x < 4 && y > 0 then (4,y-(4-x)) else (x,y)
```

Den första regeln (”fyll den stora hinken”) returnerar helt enkelt tillståndet `(4,y)`, alltså efter operationen innehåller stora hinken 4 liter, medan lilla hinken har samma volym

vatten som innan. Den här operationen kan göras så länge stora hinken inte redan är full, alltså om `x < 4`. Annars returneras gamla tillståndet `(x,y)`.

Den andra regeln, vilket innebär att vi häller från den lilla hinken till den stora, tills den stora är full, kräver lite mer för att operationen skall vara möjlig. Mer konkret måste det finnas minst 4 liter i hinkarna totalt (`x+y >= 4`), och det måste även finnas något vatten i lilla (`y > 0`) och stora skall inte redan vara full (`x < 4`). Om så är fallet returneras det nya tillståndet `(4,y-(4-x))` annars alltså det gamla tillståndet `(x,y)`.

I koden finns de åtta möjliga operationerna implementerade som funktioner. Vi har även skapat en lista: `operators :: [State->State]` vilken innehåller samtliga dessa funktioner. För någon ovan vid funktionell programmering kan det verka märkligt att ha en lista med funktioner, men det är alltså fullt möjligt i Haskell. Givet detta förarbete kan vi nu skriva:

```
makeMoves :: State -> [State]
makeMoves state = filter (/= state) (map ($ state) operators)
```

`makeMoves` ”kör” helt enkelt, med hjälp av `map`, alla funktioner i listan `operators` på det aktuella tillståndet `state`, och vi får tillbaka en lista med de resulterande tillstånden. Vi filtrerar med `filter (/= state)` förstås bort de tillstånd som är samma som ursprungstillståndet, alltså de som var ett resultat av en regel som inte var giltig.

Nu återstår bara `retrieveSolution :: State -> [StatePair] -> [State]`, alltså funktionen som återskapar lösningen genom att rekursivt följa föräldrarna från måltillståndet till starttillståndet. Funktionen blir kompakt genom att utnyttja både `let` och `where` satser, se nedan.

```
retrieveSolution :: State -> [StatePair] -> [State]
retrieveSolution s closed =
    let parent = findParent s closed
    in if parent == startParent then [s] else s: retrieveSolution parent closed
        where findParent y xs = snd . head $ filter ((== y) . fst) xs
```

Vi använder därmed `findParent` till att hitta föräldern till det aktuella tillståndet s, när vi rekursivt, från mål till start, skapar en lista bestående av tillståndens föräldrar. Vi avslutar då `parent == startParent`. Listan vänds slutligen på i `gps`. Själva `findParent` fungerar

genom att filtrera bort alla tillstånd i listan som inte har tillståndet `y` som första del av paret (det här måste ge en lista med exakt ett element eftersom ett tillstånd bara kan ha en förälder då vi inte tar med tillstånd i `newStates` som redan finns i OPEN eller CLOSED) vilket plockas ut med `head` innan vi tar ut andra delen av paret (alltså föräldern) med `snd`. Även den här lösningen är, ärligt talat, lite klumpig.

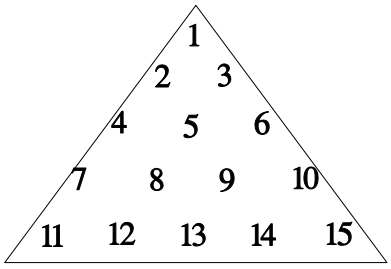
Vi har nu till slut gått igenom hela programmet. Notera att ni så klart kan testa både att köra hela programmet (med `run`) samt olika funktioner. Som ett exempel provar vi:

```
Main> makeMoves (3,0)
[(3,3),(4,0),(0,0),(0,3)]
```

Det här säger alltså att från tillståndet (3,0) kan vi med en operation nå tillstånden [(3,3),(4,0),(0,0),(0,3)]. De olika möjliga operationerna i det tillståndet är så klart ”fyll lilla”, ”fyll stora”, ”töm stora” och ”töm stora i lilla tills lilla är full”.

Uppgift 1 – Solitär. (För betyget 3)

Uppgiften innebär att anpassa de problemspecifika delarna av den utdelade koden för att lösa problemet Solitär. I solitär används ett bräde med femton positioner, se nedan



Spelet är ofta i trä, dvs. positionerna är hål, och spelpjäserna antingen peggar eller kulor. Från början finns det peggar i alla positioner utom nummer 1. Spelet är löst då det återstår exakt en pegg, och den återfinns i position nummer 1. Ett tillåtet drag består av att en pegg flyttas från en position över en annan intilliggande position, **med en pegg i,** till en **tom position**. När draget gjorts tas den överhoppade peggen bort. En lösning av spelet har därmed förstås alltid exakt 13 drag.

Drag kan ske horisontellt och diagonalt, men inte vertikalt. Exempel på tillåtna drag är därmed 4-5-6 (betyder från position 4, över position 5 till position 6, varvid peggen i 5 tas

bort) och 3-5-8 men inte 1-5-13. I startpositionen finns alltså exakt två tillåtna drag; 4-2-1 respektive 6-3-1.

Observera att ni i er lösning absolut inte får göra några förändringar i de generella delarna av det utdelade programmet. Hela poängen med programmets struktur är att hålla isär det generella och det problemspecifika. Det ni därmed måste ändra är representationen av tillstånd, samt förstås - utifrån vald representation - funktionerna `startState`, `startParent`, `isGoal` och `makeMoves`. Det är givetvis möjligt att skapa en lösning som ligger väldigt nära exemplet water jugs, alltså genom att deklarera varje möjligt drag som en egen funktion, och sedan använda en funktionslista `operators`. Dock är det betydligt fler möjliga operationer här, och det finns även en hel del symmetrier som kan utnyttjas.

Uppgift 2 – A* och 8-pusslet. (För betyget 5)

Uppgiften innebär att anpassa de generella delarna av den utdelade koden så att sökningen utnyttjar A*-algoritmen, i stället för bredden först. Det nya programmet skall testas på *8-pusslet*, vilket har 8 brickor, numrerade 1-8 (se nedan). Ett drag sker genom att flytta en bricka till den tomma positionen. Spelet är normalt i plast, och brickorna kan inte lyftas, varför det bara är möjligt att flytta en intilliggande bricka ett steg horisontellt eller vertikalt. I exempelpositionen nedan finns därmed fyra möjliga drag: någon av brickorna 1, 4, 8 eller 5 kan flyttas till den tomma mittenpositionen. (Ett alternativt synsätt är att den tomma positionen byter plats med någon intilliggande bricka.)

2	1	6
4		8
7	5	3

Det finns ingen konsensus om vad som utgör måltillståndet i 8-pusslet. Ett exempel är:

1	2	3
4	5	6
7	8	

Enkelt uttryckt innebär A*-algoritmen att OPEN, efter sammanslagningen med `newStates`, sorteras utifrån stigande f-värde innan det rekursiva anropet. f-värdet är som bekant summan av antalet drag hittills, plus värdet av den heuristiska funktionen h.

Observera att er lösning måste utgå från den utdelade koden, dvs. programmets övergripande struktur skall vara den samma. Ni måste även hålla isär generell och problemspecifik kod, se det som att det ni konstruerar är en generell A*-implementering, vilken skall kunna användas på olika problem.

Tips: En framkomlig väg är att utöka representationen till tillståndstripplar, där ett tillstånd utöver aktuellt tillstånd och parent också innehåller värderingen av det aktuella tillståndet. Värderingen behöver innehålla både f-värdet och antal drag gjorda hittills. Använd gärna Manhattan-avståndet som heuristisk funktion, den är mer informerad än andra enklare alternativ. Fundera även noga på om ni här på samma sätt kan exkludera alla tillstånd som redan finns i OPEN och CLOSED. Ni behöver här inte skapa en mer generell version än att den skall fungera för 8-usslet och liknande problem. Notera slutligen att det inte är möjligt att nå alla tillstånd i 8-pusslet från ett godtyckligt tillstånd, så pröva olika starttillstånd.

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 **båda** uppgifterna lösas, dvs. inte bara Uppgift 2. Inlämningsuppgiften är godkänd om Uppgift 1 lösts på ett tillfredsställande sätt.

Inlämningen skall bestå av två filer per uppgift:

1. Väl strukturerad källkod där det tydligt framgår vad som är generellt och problemspecifikt.
2. Väl valda körningsexempel där ni visar resultatet av att köra ert program.

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!