CS 240A : Databases and Knowledge Base
Homework #2

Ronak Sumbaly
UID : 604591897

October 13, 2015

Question 1

Query : We have a **fedemp(Name Salary, Manager)** relation and we want to find the employees who have Hoover as a manager (at any level in the management chain) and make more than 100K.

fedemp.fac

```
%Facts
% fedemp(Name:string, Salary:integer, Manager:string)

fedemp('Elliot Ness', 150000, 'Hoover').
fedemp('Ryan Baker', 75000, 'Hoover').
fedemp('Bill Bathgate', 140000, 'Elliot Ness').
fedemp('Susan Crandall', 50000, 'Bill Bathgate').
fedemp('Sam Anders', 125000, 'Steve Smith').
```

fedemp.deal

```
% Schema
database({fedemp(Name:string, Salary:integer, Manager:string)}).

% Get all Employees who have Manager = 'Hoover'
managerHoover(Name,Salary) ← fedemp(Name,Salary,'Hoover').

% Get all Employees who are managed by an Employee whose Manager = 'Hoover'
managerHoover(Name2, Salary2) ← managerHoover(Name, Salary), fedemp(Name2, Salary2, Man-
ager), Manager = Name.

% Get all Employee whose Salary > 100K
employeeK(Name) ← managerHoover(Name, Salary), Salary > 100000.
export employeeK(Name).
```

Output

```
employeeK(Elliot Ness).
employeeK(Bill Bathgate).
```

Question 2

Query : We have the problem of finding flight from San Francisco to New York, in less than 3 segments.

flights.fac

```
%Facts
% flights(FlightNo:string, Origin:string, Destination:string, Cost: integer)

flights('HY120', 'DFW', 'JFK', 225).
flights('HY130', 'DFW', 'LAX', 200).
flights('HY140', 'DFW', 'ORD', 100).
flights('HY150', 'DFW', 'SFO', 300).
flights('HY210', 'JFK', 'DFW', 225).
flights('HY240', 'JFK', 'ORD', 250).
flights('HY310', 'LAX', 'DFW', 200).
flights('HY350', 'LAX', 'SFO', 50).
flights('HY410', 'ORD', 'DFW', 100).
flights('HY420', 'ORD', 'JFK', 250).
flights('HY450', 'ORD', 'SFO', 275).
flights('HY510', 'SFO', 'DFW', 300).
flights('HY530', 'SFO', 'LAX', 50).
flights('HY540', 'SFO', 'ORD', 275).
```

flights.deal - With Lists

```
% Schema
database({flights(FlightNo:string, Origin:string, Destination:string, Cost: integer)}).

% Get all flights starting from SFO
segments(Destination, ['SFO'], 1, Cost) ← flights( _, 'SFO', Destination, Cost).

% Get all flights originating from SFO and its next destination
segments(Destination2, [Destination1 | L], Nseg1, Cost2) ← segments(Destination1, L, Nseg2 ,Cost1),
flights(_, Destination1, Destination2, Cost3), member(Destination2, L), Cost2 = Cost1 + Cost3, Nseg1
= Nseg2 + 1.

% Take out only those flights that have final destination as JFK and less that 3 segments
outputSegments(Destination, L , Nseg, Cost) ← segments(Destination, L, Nseg, Cost), Destination =
'JFK', Nseg < 3.

% Get minimum cost flight
outputMin(min<Cost>) ← outputSegments(Destination, L , Nseg, Cost).

% Get flight path with minimum cost
route(Destination, L, NSeg, Cost) ← outputSegments(Destination, L , NSeg, Cost), outputMin(Cost2),
Cost = Cost2.

export outputSegments(Destination, List , Nseg, Cost).
```

Output

```
route(JFO, [DFW, SFO], 2, 525).
route(JFO, [ORD, SFO], 2, 525).
```

Question 3 - Extra Credit

Query : Write simple Deal rules to classify a given testing set of tuples, assuming that they are virtualized as well. For simplicity you can assume that the testing set is the training set without the "Play Tennis" column. Also, write simple rules that compute the accuracy of the classifier on this testing set.

decisionTree.fac

```
%Facts
% tree(Parent:string, Col:integer, Val:string, Child:integer)
tree(0,1,'Sunny',1).
tree(0,1,'Overcast',2).
tree(0,1,'Rain',3).
tree(1,3,'High',4).
tree(1,3,'Normal',5).
tree(3,4,'Strong',6).
tree(3,4,'Weak',7).

% childOutput(Child:integer, PlayTennis:string)
childOutput(2, 'Yes').
childOutput(4, 'No').
childOutput(5, 'Yes').
childOutput(6, 'No').
childOutput(7, 'Yes').

% test_data(Id:integer, Col:integer, Val:string)
test_data(1,1,'Sunny').
test_data(1,2,'Hot').
test_data(1,3,'High').
test_data(1,4,'Strong').

test_data(2,1,'Overcast').
test_data(2,2,'Hot').
test_data(2,3,'Normal').
test_data(2,4,'Strong').

% patterns(Id:integer, Col:integer, Val:string)
patterns(1,1,'Sunny').
patterns(1,2,'Hot').
patterns(1,3,'High').
patterns(1,4,'Weak').

patterns(2,1,'Sunny').
patterns(2,2,'Hot').
patterns(2,3,'High').
patterns(2,4,'Strong').

patterns(3,1,'Overcast').
patterns(3,2,'Hot').
patterns(3,3,'High').
patterns(3,4,'Weak').
patterns(4,1,'Rain').
patterns(4,2,'Mild').
patterns(4,3,'High').
patterns(4,4,'Weak').

patterns(5,1,'Rain').
patterns(5,2,'Cool').
patterns(5,3,'Normal').
patterns(5,4,'Weak').

patterns(6,1,'Rain').
patterns(6,2,'Cool').
patterns(6,3,'Normal').
patterns(6,4,'Strong').

patterns(7,1,'Overcast').
patterns(7,2,'Cool').
patterns(7,3,'Normal').
patterns(7,4,'Strong').

patterns(8,1,'Sunny').
patterns(8,2,'Mild').
patterns(8,3,'High').
patterns(8,4,'Weak').

patterns(9,1,'Sunny').
patterns(9,2,'Cool').
patterns(9,3,'Normal').
patterns(9,4,'Weak').

patterns(10,1,'Rain').
patterns(10,2,'Mild').
patterns(10,3,'Normal').
patterns(10,4,'Weak').

patterns(11,1,'Sunny').
patterns(11,2,'Mild').
patterns(11,3,'Normal').
patterns(11,4,'Strong').

patterns(12,1,'Overcast').
patterns(12,2,'Mild').
patterns(12,3,'High').
patterns(12,4,'Strong').

patterns(13,1,'Overcast').
patterns(13,2,'Hot').
patterns(13,3,'Normal').
patterns(13,4,'Weak').

patterns(14,1,'Rain').
patterns(14,2,'Mild').
patterns(14,3,'High').
patterns(14,4,'Strong').

labels(1,'No').
labels(2,'No').
labels(3,'Yes').
labels(4,'Yes').
labels(5,'Yes').
labels(6,'No').
labels(7,'Yes').
labels(8,'No').
labels(9,'Yes').
labels(10,'Yes').
labels(11,'Yes').
labels(12,'Yes').
labels(13,'Yes').
labels(14,'No').
```

decisionTree.deal

```
% Schema
database({tree(Parent:integer, Col:integer, Val:string, Child:integer),
childOutput(Child:integer, PlayTennis:string), patterns(Id:integer, Col:integer, Val:string),
labels(Id:integer, Val:string), test_data(Id:integer, Col:integer, Val:string)}).

% Find all the child nodes of the parent present in test data
findChild(Parent,Child,ID) ← tree(Parent,Col,Input,Child), patterns(ID,Col,Input).

% Create duplicate of child nodes of testing data
childValue(Parent, Child, ID) ← findChild(Parent, Child, ID).

% Recursively get the end child of each instance in the testing data
childValue(Parent, ChildA, ID) ← childValue(ParentA, ChildA, ID), findChild(Parent, Child, ID),
ParentA = Child.

% Output the final class of the child
output(ID,Class) ← childValue(0, Child, ID), childOutput(Child, Class).

% Compare the predicted output class with actual class value
outputCorrect(ID,Val) ← output(ID,Class) , labels(ID, Class2), if (Class= Class2 then Val = 1 else Val
= 0).

% Calculate the number of instances in testing data
totalID(count<ID>) ← output(ID,Class).

% Get number of correctly classified instances.
accuracy(count<Val>) ← outputCorrect(ID,Val).

% Calculate the accuracy percentage
outputAccuracy(Val1) ← accuracy(Val),totalID(ID) ,Val1 = (Val / ID) * 100.
export output(ID,Class).
export outputAccuracy(Val).
```

Output

Query: output(ID,Class)

```
output(3,Yes).
output(7,Yes).
output(12,Yes).
output(13,Yes).
output(1,No).
output(2,No).
output(8,No).
output(9,Yes).
output(11,Yes).
output(4,Yes).
output(5,Yes).
output(10,Yes).
output(6,No).
output(14,No).
```

Query: outputAccuracy(Val)

```
outputAccuracy(100.0).
```