Gebze Technical University
Computer Engineering

CSE 341
Programming Languages
2020 Fall

HOMEWORK 4 REPORT

Yusuf Can Kan
161044007

Course Assistant: Muhammet Ali Dede

1 Implemented and Unimplemented Parts

I implements all the part except part 6.

2 PART1

After adding all the possible flight paths to the database, we need a rule that indicates there exist a rule between given two cities with considering direct routes and connected cities. For this we need a list for storing visited cities.

Firstly route(X,Y) must predicate that there is a path between X and Y. After we obtained that we must look all the possible Y neighboors for obraining more possible reachable routes. While doiing that we need to check if we had calculated founded cities previous predicates. If we found that we don't need to make recursive calls over that cities.

```
route(X, Y) :- routefind(X, Y, []).
routefind(X,Y,visited) :- flight(X,Z), %finds every path from city X
                          not(member(Z, visited)), % looks finding city is visited previous recusive calls
                          (Y = Z ;routefind(Z,Y,[X|visited])). % Y=Z => finded node is Y
                                                               % Find new routes (connected cities) recursivly.
```

In order to hold a previous cities, route calls routefind predicate.

In routefind;

flight(X,Z) => finds all the possible routes from X to other cities, or finds if there is a route if city Z given,
not(member(Z,visited)) => checks founded city is visited previously, if it is visited it returns false,
Y=Z => shows there is a route between city X and founded city,
routefind(Z,Y,[X|visited])) => adds X to visited cities and makes same predication for city Z. In this way ot looks the connected cities recursively.

```
?- [part1].
true.

?- flight(istanbul,izmir).
true.

?- flight(rize,van).
true.

?- flight(van,istanbul).
true.

?- flight(ankara,rize).
false.

?- flight(burdur,konya).
false.

?- flight(isparta,izmir).
true.

?-
```

Direct Flight Test

```
?- route(istanbul,X).
X = izmir ;
X = isparta ;
X = burdur ;
X = antalya ;
X = konya ;
X = ankara ;
X = van ;
X = rize ;
X = gaziantep ;
X = gaziantep ;
X = antalya ;
X = konya ;
X = ankara ;
X = van ;
X = rize ;
X = ankara ;
X = konya ;
X = antalya ;
X = gaziantep ;
X = van ;
X = rize ;
X = van ;
X = ankara ;
X = konya ;
X = antalya ;
X = gaziantep ;
X = rize ;
X = rize ;
X = van ;
X = ankara ;
X = konya ;
X = antalya ;
X = gaziantep ;
false.
```

```
?- route(edirne,X).
X = edremit ;
X = erzincan ;
false.

?- route(edirne,izmir).
false.

?- route(istanbul,gaziantep).
true .
```

Route Tests

3 PART2

After adding the distance informations to the file, we need to find the distance informations between 2 cities with all the alternative paths. When we find this information, we can find the minimin of the list and we can take the minimum distance inside that list.

```
%finds path cost from X to Y.
routeDistance(X,Y,Visited,Dist) :-
    distance(X,Z,D), not(member(Z, Visited)),(Y = Z, Dist is D;
    routeDistance(Z,Y,[X|Visited],K),Dist is K+D).
```

First I implemented the route distance function.
distance(X,Z,D) => this predicated all the possible flights from X to Z with distances D.
not(member(Z,Visited)) => Checks if city visited or not. It comes from previous part.
Y=Z, Dist is D => Shows there is a route between Y and Z and Distance is D.
Routedistance(Z,Y,[X|Visited],Dist is K+D) => calcualtes unvisited routes with using connected route recursivelt. Z is connected city. X added to the visited city as same reason in part 1, at the end distance is setted with summary of all the distance between connected city distances (alternative routes).

Route distance finds all the alternative distances between X and Y cities.

```
%calculates all the alternative path costs
distanceList(X, Y, Distances) :- findall(D,routeDistance(X,Y,[],D), Distances).
```

With using findall property of prolog, we take all the D (distance) information of all the paths between X and Y, created a list with them and setted to the Distance. So now distance contains the information of all the path distances between X and Y.

Now all we need to do is find the minimum of the list.

```
sroute(X,Y,D) :- route(X,Y),distanceList(X,Y,List),min_list(List,D).
```

Finally, sroute predicates the result that we want. First route looks for if there is a road between city X and city Y (from previous part), distance list fins all the alternative paths between city X and Y and stores everything in a list. In here again we used another prolog property which is min_list, that finds the minimim element in a list. We used this for finding the minimum distance and setting the D.

4 PART3

4.1 Schedule

The schedule predicates that given students to place and time of class.

```
schedule(S,P,T):- enroll(S,X), where(X,P), when(X,T).
```

Enroll finds the class, where finds the place and when finds the time.

```
?- schedule(a,P,T).
P = z23,
T = 10 ;
P = z11,
T = 12 ;
false.

?- schedule(b,P,T).
P = z23,
T = 10 ;
false.

?- schedule(c,P,T).
P = z11,
T = 12 ;
false.

?- schedule(d,P,T).
P = z06,
T = 14 ;
false.

?- schedule(e,P,T).
P = 207,
T = 16 ;
false.
```

4.2 Usage

Usage gives the usage times of classroom.

```
usage(P, T) :- where(C,P), when(C,T).
```

Where finds the classroom and then finds the time.

```
?- usage(z23,T).
T = 10.

?- usage(207,T).
T = 16 ;
T = 17.

?- usage(z11,T).
T = 12.

?- usage(z06,T).
T = 14.
```

4.3 Conflict

If there is a conflict between classes or time, it returns true.

```
conflict(X,Y):- X \= Y, ((when(X,T1), when(Y,T2), T1==T2); (where(X,P1), where(Y,P2), P1==P2)).
```

X\=Y => x is not equal y
(when(X,T1), when(Y,T2), T1==T2) => If there is a time conflicts
(where(X,P1), where(Y,P2), P1==P2)=> If there is a place conflict.

```
?- conflict(455,452).
true.
```

4.4 Meet

Meet gives true if 2 student are present in the same classroom at the same time.

```
meet(X,Y):- enroll(X,C1), enroll(Y,C2), C1==C2,!.
```

Enroll finds the clases that student X and Y takes and C1==C2 controls if they are them same.

```
?- meet(a,b).
true.

?- meet(a,c).
true.

?- meet(a,e).
false.
```

5 Part4

5.1 Element

In this part we need to check if S1 is an element of S2 set.

```
%checks, does elelemt E iosis member of S
element(E,S) :- member(E,S),!.
```

In here i used the prolog member property, which it looks in list S for if that set contains element S in it.

```
?- element(4,[1,2,4]).
true.

?- element(2,[10,15,20,30,2,12,66]).
true.

?- element(2,[10,15,20,30,12,66]).
false.
```

5.2 Union

In this part we need to implement union oredicate that predicates S3 is union of S2 and S3.

```
%if last elements are equal, the statement is true.
union([],S2,S2).

%check first element of S1 is a member of S2.
%If it is call union recursively.
union([E|S1],S2,S3):-
    element(E,S2),
    union(S1,S2,S3).

%if first element of S1 is not element of S2 or S3 pass
% first elements of s1 and s3 and call union recursively
union([E|S1],S2,[E|S3]) :-
    \+ element(E,S2),
    union(S1,S2,S3).
```

Union([E|S1],S2,S3) => This call takes the elemenmt of s1 and checks if S2 contains it or not. The element predicate controls this case, looks for E to S2. If it is true union call itself recursively without element of S1.

Union([E|S1],S2,[E|S3]) => If first element of S1 is not exist in S2, it is equal to fisrt elem, S3 call recursivelt itselt withour E of S1 and S3. This is other way for upper predicate.

Union([],S2,S2). If remained sets ar3e equal predicate is returns.

```
?- union([1,2,7,8],[3,4],X).
X = [1, 2, 7, 8, 3, 4].

?- union([1,2],[3,4],[1,2,3,4]).
true.

?- union([1,2],[3,4],[1,3,4]).
false.

?- union([3,4,9,10],[10,4,7,19,12],[10,3,9,12,19]).
false.
```

5.3 Intersect

In this part intersect predicate will look for S3, in intersection of S1 and S2.

```
intersect(S1,S2,S3) :- calculateIntesect(S1,S2,E), equivalent(E,S3).
%If there is not S3 left, predicate must be true.
calculateIntesect([],_, []).
calculateIntesect([E|S1], S2, [E|S3]):- element(E, S2), !, intersect(S1, S2, S3).
calculateIntesect([_|S1], S2, S3):- intersect(S1, S2, S3).
```

In here the calculate intersect parts calculates the intersection of S1 and S2 and sets the E, after that equivalent predicate looks if E is equivalnt to S3. (equevalnt is descricribed in next part).

In CalculateIntersect part;
calculateIntesect([E|S1], S2, [E|S3]):- tries to predicate if elements(E) of is in S2 and S2 contains this E. ! means that there will be no backtracking with unnesesairly because we don't need it.
calculateIntesect([],_, []). => If there is no element left is S1 and S3, we does not care abour S2 and return the predicate.
calculateIntesect([_|S1], S2, S3): => in this predicate it passes E of S1 and calls intersect recursively.
With this we obtain intersection elements one by one in S3 so we can use it in intersect call.

```
?- intersection([1,2,3,4],[3,4],X).
X = [3, 4].

?- intersection([1,2,3,4],[3,4],[1]).
false.

?- intersection([1,2,3,4,5,6],[3,4,5,6],[3,4,5,6]).
true.

?- intersection([2,5,7,8],[7,5],[5,7]).
true.

?- intersection([2,5,7,8],[7,5,3],X).
X = [5, 7].

?- intersection([2,5,7,8,1],[7,5,3,1],X).
X = [5, 7, 1].
```

5.4 Equivalent

In this part we need to find if given 2 sets are equivalent to each other or not.

```
%checks all the permutation fo S1 and S2 if they are equvalent.
equivalent(S1, S2) :-
    permutation(S1, S2).
```

In thisp art I used a prolog permutatio property. It directly helps us to check if given two list is equivalent or not.

```
?- equivalent([1,2,3,4],[2,1,3,4]).
true .

?- equivalent([1,2,7,8,3,4],[7,2,1,3,4,8]).
true .

?- equivalent([1,2,7,8,3,4],[7,2,1,3,9,4,8]).
false.
```

6 PART5

In this part, first we need to divide data randomly and set the arithmetic operations equally for every case. This means that, first we need to set our arithmetics.

```
operator(X,Y,Z):- Z=X+Y. %Set Z, X+Y
operator(X,Y,Z):- Z=X-Y. %Set Z, X-Y
operator(X,Y,Z):- Z=X*Y. %Set Z, X*Y
operator(X,Y,Z):- Y=\=0 ,Z=X/Y.   % avoid division by zero and set Z = X/Y
```

In this part, in order to assign operator I implement operator predicate which it takes 2 value and combine them differently and set it to the Z. X and Y are the input values and Z is the output value. As you can see in order to prevent the division by zero problem we check if Y(divider part) is equal to zero or not. This predications will be using in assignment of +,-,/,* operators between given list elements.

```
calculate(List,X,Y):-
    divide(List,Left,Right), %divide number list into 2 part for
                             %left and right side of the equation
    addOperator(Right,X), %this part adds a arithmetic operatin betweend
                          %the elements of list nd constructs the
                          %right side of the equation.
    addOperator(Left,Y),  %left side
    % calculare created left and right side of expression and assign it.
    LeftResult is Y, RightResult is X, LeftResult =:= RightResult.
                                                  %if two side is equal,
                                                  %it means we find our expresion.
```

All of our main operations are done by calculate predicate. In this predicate it uses 3 sub predictions which are;

divide(List,Left,Right) => sets the left and right part of the equation. It takes the whole list and divides into two withour looking their length or equality.

addOperator(Right,X) => Adds the operator between given list elements. Sets it in X. This part assignes the operators(+,-,/,*) the right part of the equation.

addOperator(Left,Y) => Adds the operator between given list elements. Sets it in Y. This part assignes the operators(+,-,/,*) the right part of the equation.

In last line, it has the lists with all the operators assigned between numbers and it has the left and right part. It calculates and assignes the left part of the equation to LeftResult and assignes and calculates the right part of the equation in RightResult and in final it compares if they are equal or not.

When we look to the addOperator predicate more closely we can see that it takes 2 list parameters. In here again we need to operate dividing operation because in order to place any opetor between two value first we need to obtain value. Otherwise we can’t feed the operator predicate.

```
addOperator(List,Result):- divide(List,Left,Right), %divide number list into 2 part for
                           %left and right side of the equation
    %Create right side of equations
    addOperator(Right,RightSide),
    %Create left and right side of equations
    addOperator(Left,LeftSide),
    operator(LeftSide,RightSide,Result). %add arithmetic expression between left and right
                                         % and set result to it.
```

In here divide divides the list into different sub parts and addOperator does this recursively. When there is only 1 element left in both of the side it assigns the operator and feeds the result with it.

Program Usage Format

For testing this part please fill the input.txt file and just load the pl file as shown in below.

```
?- ['part5.pl'].
true.
```

It will write the result in output.txt file automatically.

Test Results

Test 1 “input.txt” file;

```
5 3 5 7 49
```

Test 1 “output.txt” file;

```
49 = (5-(3-5))*7
49 = (5-3+5)*7
```

Test 2 “input.txt” file;

```
3 4 6 2 10 23
```

Test 2 “output.txt” file;

```
4+(6*2+(10-23)) = 3
4+(6*2+10-23) = 3
4*(6-2)+(10-23) = 3
4+6*2+(10-23) = 3
4+(6*2+10)-23 = 3
4*(6-2)+10-23 = 3
4+6*2+10-23 = 3
6*2+(10-23) = 3-4
6/(2/10)-23 = 3+4
6*2+10-23 = 3-4
6/2*10-23 = 3+4
10-23 = 3-4*(6-2)
10-23 = 3-(4+6*2)
10-23 = 3-4-6*2
23 = (3/(4+6)+2)*10
```

Test 3 “input.txt” file;

```
2 3 5 7 11
```

Test 3 “output.txt” file;

```
3-(5+(7-11)) = 2
3-(5+7-11) = 2
3-5-(7-11) = 2
3-(5+7)+11 = 2
3-5-7+11 = 2
(3*5+7)/11 = 2
7-11 = 2*(3-5)
11 = 2-(3-(5+7))
11 = 2-(3-5-7)
11 = 2-3+(5+7)
11 = 2-(3-5)+7
11 = 2-3+5+7
```

Test 4 “input.txt” file;

```
1 1 59 1
```

Test 4 “output.txt” file; (There is no output because there is no result.)