Prolog

Name: Danny Rene Jensen
Username: Danje14
DM552

October 27, 2015

Contents

1 introduction

In this project we need to create a Prolog program, that can sort a list of N length. We will be using comparators to do this. A comparator(I,J) has two index numbers, I and J on the list that needs to be sorted. These indexes then needs to switch places, if they are not in the right order. Also in this project the list will be sorted in acceding order from the right.

2 Predefining

For defining a comparator the whole word is used followed by brackets with I and J variables for the indexes in the list. I and J needs to be integers.

```
comparator(I,J):-
        integer(I),
        integer(J).
```

3 2.1

3.1 1

is_network/1 just needs to check the whole comparator list through if all elements are comparators. This is easily done by just creating a base case where the list is empty and then create another predicate, where it checks all elements in the list, to see if every element is comparator(_,_) this is done recursively by checking the tail of the list.

3.1.1 Test

Lets test a network to see if the is_network/1 is working correctly:
$?-is\_network([comparator(1,2), comparator(2,3), comparator(1,3)]).$
$true.$
Okay, that was a comparator network. Now, let‘s see if it is not a comparator network: $?-is\_network([comparator(1,2), cow, comparator(1,3)]).$
$false.$
Okay, so if it has for example cow in the list instead of comparator it says false.

3.2 2

channels/2 has a base case, where the list of comparators is empty, then we don't care about how many channels there are.If the list isn't empty, the next channels predicate is used. This predicate checks if the list is a network by using is_network from above. After that, the channels predicate checks if every comparators I and J is lower or equal to number of channels N. This is done by using the built in predicate between/3, it would be impossible to try and sort the fifth place in a list, if the list only has four elements. This predicate could probably be optimised a bit by not letting it check if the list of comparators is a network in every recursive call. This predicate also has no end, meaning it is an infinite loop.

3.2.1 Test

If we look if the comparators have the right channels:
$?-channels([comparator(1,2), comparator(3,4)], 4).$
$true.$
So, these two comparators can run on 4 and above channels.
Now, what if the comparators use a channel out of bounds:
$?-channels([comparator(1,2), comparator(3,7)], 4).$
$false.$
It says false so the comparator(3,7) is too large for only 4 channels.
Now, what if we only give it the number of channels:
$?-channels(C, 2).$
$C = [];$
$C = [comparator(1,1)];$
$C = [comparator(1,2)];$
$C = [comparator(2,1)];$
$C = [comparator(2,2)];$
$C = [comparator(1,1), comparator(1,1)];$
$C = [comparator(1,1), comparator(1,2)];$
$C = [comparator(1,1), comparator(2,1)];$
$C = [comparator(1,1), comparator(2,2)];$
$C = [comparator(1,2), comparator(1,1)];$
$C = [comparator(1,2), comparator(1,2)];$
$C = [comparator(1,2), comparator(2,1)];$
$C = [comparator(1,2), comparator(2,2)];$
$C = [comparator(2,1), comparator(1,1)];$
$C = [comparator(2,1), comparator(1,2)];$
$C = [comparator(2,1), comparator(2,1)];$
$C = [comparator(2,1), comparator(2,2)];$
$C = [comparator(2,2), comparator(1,1)];$
$C = [comparator(2,2), comparator(1,2)];$
$C = [comparator(2,2), comparator(2,1)];$
$C = [comparator(2,2), comparator(2,2)];$
$C = [comparator(1,1), comparator(1,1), comparator(1,1)];$
$C = [comparator(1,1), comparator(1,1), comparator(1,2)];$
$C = [comparator(1,1), comparator(1,1), comparator(2,1)]$
This looks good, but this is an infinite loop, so it will never terminate. It will just add another comparator to the list.

3.3 3

The run/3 predicate is where the sorting happens. First of, we need a base case that states that if there are no comparators, the input list and output list are the same, which makes sense if you don't sort a list, it stays the same. The next base case is for an empty input list, if the input list is empty the output list must also be empty no matter the comparator list, but this list must actually also be empty, because if we have no input list, there can't be any comparators. If we have a input list and a comparator list that is not empty, then it needs to sort the list on the indexes I and J from each comparator. run/3 needs to extract the two variables in the input list on index I and J, and sort them. This happens in the sorting/4 predicate. When this is done run/3 needs to replace the two variables on the same indexes in the input list, this is done in the replace predicate.

```
replace(1,X,[_|T],[X|T]):-!.
replace(I,X,[H|T],[H|R]):-
        I > 0,
        NI is I-1,
        replace(NI,X,T,R).
```

The replace/4 predicate takes the index of the variable that needs to be placed back into the list, the element to put back into the list, the input list and it outputs the list with the replaced variable. replace/4 runs through the input list and reduces I with 1, until I is 1, then it replaces the element on that index and it is done. After the run sorted the first variables on I and J, it takes the next comparator.

3.3.1 Test

Can it sort a list with some comparators:
$? - run([comparator(1,2), comparator(3,4)], [2,1,4,3], L).$
$L = [1,2,3,4].$
This looks good, it sorted the elements. But what if it tries to sort two elements, that are already on their right spots:
$? - run([comparator(1,2)], [1,2,4,3], L).$
$L = [1,2,4,3].$
It did nothing, because the two elements were already on their right spot. It also did not sort element 3 and 4, because there where no comparator for these.

3.4 4

The goal with is_SN/2 is to check if the network S really is a sorting network. First of all is_SN/2 needs to check if S is a network. This is done with the is_network predicate from above. Then we need to check if the network's comparators are legal on the N channels. This is done with the channels/2 predicate from above. If this is true it can continue, but it needs something to try and sort the comparator list with, to check if it is a sorting network. For it to be a sorting network, it needs to be able to sort a whole list of N channels regardless of how the elements in the channels list are. So to check this we use the zero-one principle. This zero-one principle means that we create all the possible list of 0 and 1, with N length, e.g if N=2 then we get the four lists: [0,0],[0,1],[1,0],[1,1]. To generate this list the net/2 predicate is used:

```
net(0,[]):-!.
net(N,[X|T]):-
        N > 0,
        N1 is N-1,
        between(0,1,X),
        net(N1,T).
```

It recursively adds 0 and 1 to the lists using the between/3 predicate. But this only gives us one list at a time and we want all lists at once. To do this net/2 is put into a findall/3 predicate, which outputs all possibilities, which net/2 can create in a list. This means, we now have a list of lists with all possible arrangements of 0 and 1. This list is then given to sn_sort/2 with the comparator list that we need to check if it is a sorting network.

```
sn_sort(_,[]):-!.
sn_sort(S,[X|T]):-
        run(S,X,L),
        msort(X,C),
        is_equal(L,C),
        sn_sort(S,T).
```

snsort/2 throws a list of unsorted 0 and 1 into run/3 with the comparator list S, to sort the list. Then it also sorts the list with build in msort/2 predicate. msort/2 sorts a given list and outputs it, just like sort/2 but msort/2 doesn't throw duplicates away like sort/2 does, and since the list to sort on only consists of 0 and 1, there are bound to be duplicates in the list and we want the whole list with duplicates.When sn_sort/2 have sorted the list both ways, it checks if they are equal. This is done with the is_equal/2 predicate.

```
is_equal([],[]):-!.
is_equal([X|T],[X|S]):-
        is_equal(T,S).
```

is_equal/2 looks at every element in both lists and compares them if they are equal. After all this is done, sn_sort/2 takes the next list of 0 and 1.

3.4.1 Test

Lets check, if it can detect a sorting network:
$?-is\_SN([comparator(1,2),comparator(2,3),comparator(1,2)],3).$
$true.$
So the comparator network was a sorting network.
Now, if we remove a comparator so it won't be able to sort a list any more:
$?-is\_SN([comparator(1,2),comparator(2,3)],3).$
$false.$
So, this is no sorting network, because it can not sort all possible lists.

3.5 5

find_SN as it's name says it needs to be able to find all the sorting networks on N channels. This is done by using channels/2 to find all comparator networks on N with the length of K, this is done with the length/2. Then check these with is_SN/2. But since channels/2 is an infinite loop it finds all the network, but sadly never terminates, because channels/2 tries to find more comparator networks with K length.

3.5.1 Test

Can find_SN/3 find all the sorting networks on 3 channels:
$?-find\_SN(3,3,S)$.
$S = [comparator(1,2), comparator(1,3), comparator(2,3)]$;
$S = [comparator(1,2), comparator(2,3), comparator(1,2)]$;
$S = [comparator(1,3), comparator(1,2), comparator(2,3)]$;
$S = [comparator(1,3), comparator(2,3), comparator(1,2)]$;
$S = [comparator(2,1), comparator(1,3), comparator(1,2)]$;
$S = [comparator(2,3), comparator(1,2), comparator(2,3)]$;
$S = [comparator(2,3), comparator(1,3), comparator(1,2)]$;
$S = [comparator(3,2), comparator(1,3), comparator(2,3)]$;
This looks really good, it found all the sorting networks, but the predicate sadly does not terminate after this, because of channels.

4 2.2

4.1 6

In a standard comparator I is smaller then J, is_standard/1 needs to check the first comparator in the comparator list, whether this is the case, if it is, it checks the next comparator until it is done with the list or it finds a comparator that is not standard.

4.1.1 Test

Let‘s test if a standard comparator list works:
$?-is\_standard([comparator(1,2), comparator(2,3), comparator(1,4)])$.
$true$.
It returned true so this works, now can it detect a non standard comparator:
$?-is\_standard([comparator(1,2), comparator(2,3), comparator(4,1)])$.
$false$.
It certainly could, because 4 is larger than 1 and in the last comparator it returned false.

4.2 7

standardize/2 just takes a list of comparators and makes sure every comparator is standard. This is done by taking I and J, put them into a list and sort them with msort/2. Then take the list msort/2 outputs and take the two elements out and place them into the comparator in the output list of standardize/2, and then take next comparator in the input list. Maybe the run time could be better if standardize/2 instead checked if it needs to sort them first, but then it would need more predicates, and it would be longer. This is not nice to look at and the run time is really small to begin with.

4.2.1 Test

Can this predicate standardize a comparator network:
$?-standardize([comparator(2,1), comparator(3,2), comparator(1,2)], C2)$.

$C2 = [comparator(1,2), comparator(2,3), comparator(1,2)]$.
It could, and it did not switch the last comparator, because it already was standard. It even works to check if a standard comparator list is equivalent to a non standard:
$? - standardize([comparator(2,1), comparator(3,2), comparator(1,2)],$
$[comparator(1,2), comparator(2,3), comparator(1,2)])$.
$true$.

4.3 8

The equivalent/3 predicate has to check if two lists are identical, even though the comparators are in different order and are not standard. For equivalent/3 to work, it needs to check if the lists are networks. This is done using the is_network/1 predicate from above. If they are networks, then it needs to check if the two lists have the same length. The two lists can't be equivalent if they are not sharing all the same comparators, but these can be non standard comparators too, so we need to standardize the two lists. This is done with standardize/2 predicate from above. These two standard comparator lists are then thrown into the check1/3 predicate.

```
check1(_,[],_):-!.
check1(_,[],[]):-!.
check1(N,[comparator(I,J)|T],C2):-
       check2(comparator(I,J),C2),
       check1(N,T,C2).
```

This predicate only serves to run through the first comparator list recursively and throw every comparator into the check2/2 predicate.

```
check2(comparator(I,J),[comparator(X,Y)|_]):-
       X is I,
       Y is J,!.

check2(comparator(I,J),[comparator(X,Y)|T]):-
       dif(comparator(I,J),comparator(X,Y)),
       check2(comparator(I,J),T).
```

The first check2/2 is where the two comparators are identical and that means the comparator from list one is in list two, so it needs to brake and then check/1 would take the next comparator from list one. The second check2/2 just runs recursively through list two, if the comparator from list one is not equal to the comparator from list two. To check if they are different the build-in dif/2 is used, this predicate takes two elements and compares them. If they are different, it returns true. check2/2 is a bit special because it has no base case, it has none because if the comparator from list one is not in list two, they are not equivalent and it should return false.

4.3.1 Test

Let's see if two lists are equivalent:
$? - equivalent(3, [comparator(2,1), comparator(3,2), comparator(1,2)],$

$[comparator(1,2), comparator(1,2), comparator(3,2)]).$
$true.$
So even when the comparators are switched around and standardized or not it can still find out if two lists are equivalent.

5 2.3

5.1 9

A layered network is just a list of comparator lists, so is_network/1 from above is used on every comparator list inside the layered network list.

5.1.1 Test

If layered_network takes a layered network list:
$?-layered\_network([[comparator(1,2), comparator(3,4)], [comparator(1,2)]]).$
$true.$
So, this is a layered network, but let‘s try with one that is not:
$? - layered\_network([[comparator(1,2), cow], [comparator(1,2)]]).$
$false.$
Here, once again, it will not take the list with cow in it, because this is no comparator network and therefore no layered network.

5.2 10

To see if a comparator list and a layered network list is identical, we need to make sure all comparators are in the layered network list. The layered/2 predicate is used to go through the whole comparator list, and throw each comparator into the checklist/2 predicate.

```
checklist(comparator(I,J),[X|_]):-
        checkcomp(comparator(I,J),X),!.

checklist(comparator(I,J),[_|L]):-
        checklist(comparator(I,J),L).
```

This predicate is used to go into the right list inside the layered list. The checklist/2 predicate then use the checkcomp/2 predicate to run through the whole layered list, until the comparator is found.

```
checkcomp(comparator(I,J),[comparator(X,Y)|T]):-
        dif(I,X),
        dif(J,Y),
        checkcomp(comparator(I,J),T).

checkcomp(comparator(I,J),[comparator(X,Y)|_]):-
        I is X,
        J is Y,!.
```

checkcomp/2 takes each individual list inside the layered list, runs through all comparators in there, until it either fails or finds the comparator. If checkcomp/2 fails checklist/2 uses it‘s second predicate to take the next list, this continues until either the element is found and then layered takes the next comparator, or if the comparator is not in the whole layered list it fails.

5.2.1 Test

Let’s see, if a list and its layered list is working:
$?-layered([comparator(1,2), comparator(3,4), comparator(1,2)],$
$[[comparator(1,2), comparator(3,4)], [comparator(1,2)]]).true.$
It is, so it can find out if a comparator list and a layered list is equivalent. But what if they are not:
$?-layered([comparator(1,2), comparator(3,4), comparator(2,4)],$
$[[comparator(1,2), comparator(3,4)], [comparator(2,3)]]).false.$So, it can also tell if a layered list isn’t equivalent to the comparator list.

5.3 11

To create a layered network from a comparator network we must make sure that no comparator uses the same channel in the same layer.
The network_to_layered/2 predicates job is to create the lists inside the layered network. This new list is then thrown into the layer/4 predicate with the whole comparator list and an empty list. This empty list is used to as a checker, this means that every comparator, that can be put on the layer in the layered network is put into the list and this list is then used to check if a comparator can be put into this list. The A list in layer/4 is an output list with all the comparators that could not be put on that list. Then network_to_layered/2 just runs through again with the comparator list A.

```
layer([],[],_,[]):-!.
layer([comparator(I,J)|T],L2,Checklist,[comparator(I,J)|Lt]) :-
       checklayer(comparator(I,J),Checklist),
       layer(T,L2,[comparator(I,J)|Checklist],Lt).

layer([comparator(I,J)|T],[comparator(I,J)|Rt],Checklist,L):-
       not(checklayer(comparator(I,J),Checklist)),
       layer(T,Rt,Checklist,L).
```

The layer/4 predicate is split into two. The first predicate is used, if the comparator can be put into this layer in the layered network. If checklayer/2 returns true, that means no comparator shares the same channels as the one we are currently looking at. Then it can throw the comparator into the list. The recursive call is then made with the empty list, we want to throw the comparator into, but the comparator is put into the Checklist instead. This is the empty list A from network_to_layered/2, This list is the one, that we are comparing every comparator to. This check list is needed, because if we threw the comparator into the Lt list, which is the one we want the comparator in at the end, the first comparator would be put into this list, but every time after that it would only go into the second layer/2 predicate, and in here it would then fail. The second layer/2 predicate is used almost the same way as the first, this time if the checklayer/2 predicate returns false, layer/2 will then put the comparator into the list A so network_to_layered/2 can use this in it's next run-through and not into Checklist.

```
checklayer(_,[]):-!.
checklayer(comparator(I,J),[comparator(X,Y)|T]):-
        dif(I,X),
        dif(J,Y),
        dif(I,Y),
        dif(J,X),
        checklayer(comparator(I,J),T).
```

checklayer/2 just runs through the whole Checklist from layer/2 and compares all the channels on every comparator in the list with the one we are trying to put into the layered network. This has no base case as we want it to fail so the not/1 predicate in the second layer/2 predicate is true.
Sadly, the network_to_layered/2 predicate runs through 2 times one with the right output and a second time where it fails.

5.3.1 Test

Can the predicate create a single layered network:
$? - network_to_layered([comparator(1,2), comparator(3,4)], L).$
$L = [[comparator(1,2), comparator(3,4)]];$
$false.$
Yes, it could, but sadly this predicate is giving us two results, the list and false.
Can it also create more than one list in the layered network list:
$?-network_to_layered([comparator(1,2), comparator(3,4), comparator(1,4)], L).L = [[comparator(1,2), comparator(3,4)], [comparator(1,4)]];$
$false.$
It could, but sadly it also gets two answers.

5.4 12

To convert a layered network to a comparator network, we just put every list inside the layered list into one single list. This is done with append/3.

5.4.1 Test

Let's test, if it can find the comparator list from a layered list:
$?-layered_to_network([[comparator(1,2), comparator(3,4)], [comparator(1,4)]], C).$
$C = [comparator(1,2), comparator(3,4), comparator(1,4)]$. That went well.
What if the layered list only has one layer:
$? - layered_to_network([[comparator(1,2), comparator(3,4)]], C).$
$C = [comparator(1,2), comparator(3,4)]$. This went perfect too.

6 Conclusion

All in all, each part of the program is running as it should, with some small hiccups.

7 Appendix (source code)

```
comparator(I,J):-
        integer(I),
        integer(J).

/*opg.1*/
is_network([]).
is_network([comparator(_,_)|T]):-
        is_network(T).

/*opg.2*/
channels([],_).
channels([comparator(I,J)|T],N):-
        is_network([comparator(I,J)|T]),
        between(1,N,I),
        between(1,N,J),
        channels(T,N).

/*opg.3*/
run([],X,X):-!.
run(_,[],[]):-!.
run([comparator(I,J)|T],L,Ol):-
        nth1(I,L,A),
        nth1(J,L,B),
        sorting(A,B,C,D),
        replace(I,C,L,L2),
        replace(J,D,L2,K),
        run(T,K,Ol).

sorting(A,B,B,A):-
        A > B,!.
sorting(A,B,A,B):-
        A =< B,!.

replace(1,X,[_|T],[X|T]):-!.
replace(I,X,[H|T],[H|R]):-
        I > 0,
        NI is I-1,
        replace(NI,X,T,R).

/*opg.4*/
is_SN(S,N):-
        is_network(S),
        channels(S,N),
        findall(X,net(N,X),L),
        sn_sort(S,L).

net(0,[]):-!.
net(N,[X|T]):-
        N > 0,
        N1 is N-1,
        between(0,1,X),
```

```
        net(N1,T).

sn_sort(_,[]):-!.
sn_sort(S,[X|T]):-
        run(S,X,L),
        msort(X,C),
        is_equal(L,C),
        sn_sort(S,T).

is_equal([],[]):-!.
is_equal([X|T],[X|S]):-
        is_equal(T,S).

/*opg.5*/
find_SN(_,0,[]):-!.
find_SN(N,K,S):-
        0 < K,
        channels(S,N),
        length(S,K),
        is_SN(S,N).

/*opg.6*/
is_standard([]):-!.
is_standard([comparator(I,J)|T]):- I < J,
        is_standard(T).

/*opg.7*/
standardize([],[]):-!.
standardize([comparator(I,J)|T],[comparator(C,D)|S]):-
        msort([I,J],L),
        nth1(1,L,C),
        nth1(2,L,D),
        standardize(T,S).

/*opg.8*/
equivalent(_,[],[]):-!.
equivalent(N,C1,C2):-
        is_network(C1),
        length(C1,K),
        length(C2,K),
        standardize(C1,C3),
        standardize(C2,C4),
        check1(N,C3,C4).

check1(_,[],_):-!.
check1(_,[],[]):-!.
check1(N,[comparator(I,J)|T],C2):-
        check2(comparator(I,J),C2),
        check1(N,T,C2).

check2(comparator(I,J),[comparator(X,Y)|_]):-
        X is I,
        Y is J,!.

```

```
check2(comparator(I,J),[comparator(X,Y)|T]):-
       dif(comparator(I,J),comparator(X,Y)),
       check2(comparator(I,J),T).

/*opg.9*/
layered_network([]):-!.
layered_network([X|T]):-
       is_network(X),
       layered_network(T).

/*opg.10*/
layered([],_):-!.
layered([comparator(I,J)|T],L):-
       checklist(comparator(I,J),L),
       layered(T,L).

checklist(comparator(I,J),[X|_]):-
       checkcomp(comparator(I,J),X),!.

checklist(comparator(I,J),[_|L]):-
       checklist(comparator(I,J),L).

checkcomp(comparator(I,J),[comparator(X,Y)|T]):-
       dif(I,X),
       dif(J,Y),
       checkcomp(comparator(I,J),T).

checkcomp(comparator(I,J),[comparator(X,Y)|_]):-
       I is X,
       J is Y,!.

/*opg.11*/
network_to_layered([],[]):-!.
network_to_layered(C,[H|T]):-
       layer(C,A,[],H),
       network_to_layered(A,T).

layer([],[],_,[]):-!.
layer([comparator(I,J)|T],A,Checklist,[comparator(I,J)|Lt]) :-
       checklayer(comparator(I,J),Checklist),
       layer(T,A,[comparator(I,J)|Checklist],Lt).

layer([comparator(I,J)|T],[comparator(I,J)|Rt],Checklist,L):-
       not(checklayer(comparator(I,J),Checklist)),
       layer(T,Rt,Checklist,L).

checklayer(_,[]):-!.
checklayer(comparator(I,J),[comparator(X,Y)|T]):-
       dif(I,X),
       dif(J,Y),
       dif(I,Y),
       dif(J,X),
       checklayer(comparator(I,J),T).

```

```
/*opg.12
           done*/
layered_to_network([],[]):-!.
layered_to_network([X|T],C1):-
       append(X,C,C1),
       layered_to_network(T,C).
```