DM552 – Programming Languages

Project 1, Fall 2015

Mads Petersen
mpet006

Contents

1 Preface 2

2 Assignment 2

2.1 . . . 2

2.1.1 is_network/1 . . . 2

2.1.2 channels/2 . . . 2

2.1.3 run/3 . . . 2

2.1.4 is_SN/2 . . . 3

2.1.5 find_SN/3 . . . 3

2.2 . . . 5

2.2.1 is_standard/1 . . . 5

2.2.2 standardize/2 . . . 5

2.2.3 equivalent/3 . . . 5

2.3 . . . 6

2.3.1 Layered network representation . . . 6

2.3.2 layer/2 . . . 6

2.3.3 network_to_layered/2 . . . 6

2.3.4 layered_to_network/2 . . . 8

2.3.5 Combining . . . 8

3 Example executions 9

3.1 find_SN/3 . . . 9

3.2 is_standard/1 . . . 9

3.3 layer/2 . . . 10

4 Appendix 11

4.1 Code listing . . . 11

4.1.1 project1.pl . . . 11

1 Preface

Data and it's representation
Comparators will be represented by the predicate *c/2* where the 2 parameters are the two integers representing the input it operates on. So the standard comparator working on input 1 and 2 would look like : *c(1,2)*
A network is a list of these comparators, so the network consisting of the standard comparators operating on input 1 and 2, and 2,3, Would look like:*[c(1,2),c(2,3)]*
The binary lists used to test for sorting networks is just that, a list of 1's and 0's e.g.:*[1,1,0,1]* these are generated with the predicate *binaryList/2*. A layered network is a list of lists, each inner list containing a layer, so the layered network of the above example would look like:*[[c(1,2)],[c(2,3)]]*.
SWI-Prolog v. 7.2.3 x64 has been used for everthing.

2 Assignment

2.1

2.1.1 is_network/1

```
is_network(C) :- %Checks that C is a network in the chosen model.
  C = [H|T],
  isComparator(H),
  (is_network(T) ; T = []).
```

This predicate simple tests that every element of the parameter C is a comparator, this means that a network must contain at least one comparator or it isn't considered a network.

2.1.2 channels/2

```
channels(C,N) :- %Can check if a network C over N channels is valid,
          %or can be used to generate all possible networks over N channels.
          %Can also return the minumum number of channels the network is valid for.
  ( var(N)
  ->
    sort(C,Csorted),
    Csorted = [H|T],
    (T = [] ; channels(T,N)),
    comparatorInRange(H,N)
  ;
    C = [H|T],
    (T = [] ; channels(T,N)),
    comparatorInRange(H,N)
  ).
```

This predicate uses *comparatorInRange/2* to check that every comparator in *C* is one of the possible comparators for a network over *N* channels. This is done by generating all possible pairings of integers between 1 and *N*, and returning those that are comparators, which is simply those where $I \neq J$.

2.1.3 run/3

```
run(C,Input,Output) :- %Doesn't work for input not using all channels.
  length(Input,L),
  channels(C,L),
  same_length(Input,Output),
  runWork(C,Input,Output).

runWork(C,Input,Output) :-
  C = [c(I,J)|T],
  nth1(I,Input,Xi),
  nth1(J,Input,Xj),
  (Xi > Xj, swap(Input,I,J,InputNew), (T = [], Output = InputNew ; runWork(T,InputNew,Output))
      ;
  Xi =< Xj, (T = [], Output = Input ; runWork(T,Input,Output))).
```

This predicate only works for input that uses all channels of the network i.e. it doesn't work for input of length 3 or less run on a network with 4 or more channels. It fetches the inputs from posistions I, J and compares them, if $input[I] > input[J]$ then it swaps their posistions using the swap predicate. This is then repeated for every comparator in network C. The *swap*/4 predicate does simply that, swaps the posistions of elements passed as parameter 2 and 3, from the list passed as parameter 1. The output is then equivalent to the list in parameter 4.

2.1.4 is_SN/2

```
is_SN(S,N) :- %Tests if network S is a sorting network over N channels.
  is_network(S),
  findall(X, binaryList(X,N), [_|TestBatch]),
  comprehensiveTest(S,N,TestBatch).

comprehensiveTest(S,N,[H|T]) :- %Used in testing for sorting networks.
  run(S,H,Sorted),
  isSorted(Sorted),
  (T = [] ; comprehensiveTest(S,N,T)).

isSorted([_|[]]):- !. %Tests wheter an input is sorted correctly, in ascending order.
isSorted([F,S|T]) :-
  F =< S,
  isSorted([S|T]).
```

The *is_SN*/2 predicate tests wheter or not a comparator network C over N channels is a sorting network. This is done by testing if the network C can sort all binary lists of length N correctly. If it can then it is a sorting network, otherwise it is not. This is done by first creating all the possible binary lists with the predicate *binaryList*/2 that generates a binary list of length L. All possible binary lists is then found via the $findall/3$ predicate. This results in a list of binary lists, the first of these lists is ignored and the rest is used. This is done because the list that contains all 0's is equivalent to the list containing all 1's. The list of all 0's is always the first list generated. The these lists are sorted 1 by 1 and tested with *isSorted*/1 if they are sorted correctly. If all lists are sorted correctly then the network C is a sorting network.

2.1.5 find_SN/3

```
find_SN(N,K,S) :- %Finds a sorting network S of length K over N channels, Takes VERY long for
    networks of more than 4 channels.
  channelsOfLengthL(N,K,S),
  is_SN(S,N).

find_SN1(N,K,S) :- %Finds a sorting network S of length K over N channels, is faster than
    find_SN. But doesn't find all possible solutions.
  possibleSolutionsOfLengthL(N,K,S),
  is_SN(S,N).

find_SN2(N,K,S) :- %Finds a sorting network S of length K over N channels, is faster than
    find_SN1. But finds fewer possible solutions.
  partialSolutionsOfLengthL(N,K,S),
  is_SN(S,N).

possibleSolutionsOfLengthL(N,L,C):- %Generates a more limited solution space that should
    contain a solution to a network on any number of channels.
  length(C,L),
  findall(List,(between(1,N,X),possibleComparatorsForChannelX(X,N,List)),List),
  permutation(List,PList),
  Rem is N,
  possibleSolutionsOfLengthL(C,N,L,Rem,PList).
possibleSolutionsOfLengthL(C,N,L,0,_):-
  Lnew is L - N,
  channelsOfLengthL(N,Lnew,C).
possibleSolutionsOfLengthL([H|T],N,L,Rem,[PListH|PListT]) :-
  member(H,PListH),
  RemNew is Rem - 1,
  (T = [] ; possibleSolutionsOfLengthL(T,N,L,RemNew,PListT)).
```

find_SN/3 finds a sorting network S of length K over N channels. It does this by going through all possible networks, of length K if specified, otherwise of all lengths starting from 1. Until a network that is a sorting network on N channels is found. This takes massive amounts of time when going above 4 channels since, on 4 channels there are 10^K possible networks. And they each has to be run on $(2^N) - 1 = 15$ different inputs. The shortest network that is a sorting networks on 4 channels is of length 5, thats 100.000 networks times 15 runs. When considering 5 channels there are 15^K possible networks, that each has to be run with $(2^N) - 1 = 31$ different inputs. On 5 channels the shortest possible is length 9. That equals 38.443.359.375 or $3,84 \cdot 10^{10}$ times 31 runs. That is a serious exponential growth. Solving for 5 inputs is possible but takes around 8 hours when checking only standard networks and using *find_SN*/3, when using *find_SN*1/3 it takes around 5,5 hours, using *find_SN*2/3 it can be cut to 5,5 minutes. All of the above numbers is when using standard comparators i.e. $I < J$. For non standard comparators the base numbers are doubled, so 20^K for 4 inputs, 30^K for 5. *find_SN*1/3 and *find_SN*2/3 does much the same as *find_SN*/3 except generates smaller solution spaces. They are based on the assumption that for any possible network to be a sorting network, all input channels must be represented. The predicate *possibleComparatorsForChannel*/3 is then used to generate lists of all the possible comparators that includes channel X in a network over N channels. And a list with comparators representing each input, i.e. 1 through N, is created. For instance, in a network with 3 channels, the comparators possible for channel 1 are : $c(1,2), c(1,3)$, the comparators for channel 2 are: $c(1,2), c(2,3)$, and for channel 3: $c(1,3), c(2,3)$. Now in a possible solution, one from each of these lists must be present. The difference between *find_SN*1/3 and *find_SN*2/3 is that the list of comparators representing each channel is generated with the first element be a comparator with channel 1 represented, the second element will be a comparator with channel 2 represented and so forth to channel N. In *find_SN*1 permutations of this list is then used. While in *find_SN*2 only solutions, that have channel 1 represented in the first element and channel 2 in the second and so on, are found.

2.2

2.2.1 is_standard/1

```
sc(I,J) :-  %The definition of a comparator.
  integer(I), integer(J),
  I < J.

is_standard(C) :- %Checks that C is a network in the chosen model.
  C = [H|T],
  isStandardComparator(H),
  (is_standard(T) ; T = []).

isStandardComparator(Comparator) :- %Tests that the element Comparator is a comparator in the
    chosen model.
  Comparator = c(I,J), sc(I,J).
```

This is handled by adding a new predicate $sc/2$ that represents a standard comparator operating on the inputs I, J, then checking that every comparator in a network C is a standard comparator. If this is true then C is a standard network.

2.2.2 standardize/2

The standardize algorithm presented in the assignment:

```
let (i, j) be the first comparator in S where j > i.
replace it with (i, j) and interchange i and j in all comparators after it.
Repeat until the network is standard.
```

run on a non-standard sorting network:$[c(1,4), c(3,2), c(1,3), c(2,4), c(2,3)] \rightarrow$
$(i, j) = (1, 4)$
replacing it with itself, same as not doing anything, so the network is unchanged:$[c(1,4), c(3,2), c(1,3), c(2,4), c(2,3)]$
swapping i and j in all comparators after it $\rightarrow [c(1,4), c(2,3), c(3,1), c(4,2), c(3,2)]$
repeating $\rightarrow (i, j) = (1, 4)$ i and j are unchanged since this is still the first comparator where $j > i$
replacing with itself, network unchanged $\rightarrow [c(1,4), c(2,3), c(3,1), c(4,2), c(3,2)]$
swapping i and j in all comparators after it $\rightarrow [c(1,4), c(3,2), c(1,3), c(2,4), c(2,3)]$
Initial network is obtained, any further iterations will only result in one of the two networks already obtained.
None of the them are standard, and the second one is not a sorting network anymore.
Because the stardization algorithm doesn't work, it is not implemented.

2.2.3 equivalent/3

Is not implemented for the same reasons as 7 is not.

2.3

2.3.1 Layered network representation

The representation of a layered network is a list of lists, where each inner list represents a layer of the network. The outer list is the network. A network with 2 layers, each layer with 1 comparator would thus look like: $[[c(1,2)],[c(2,3)]]$.

2.3.2 layer/2

Is done by using either 10 or 11 with both parameters instantiated.

2.3.3 network_to_layered/2

```
network_to_layered([],[]). %Converts a network C into a layered network representation L of C.
network_to_layered(C,[HLayer|TLayers]) :-
  possibleFirsts(C,Firsts), %Returns the indices of the comparators in C that possibly can
      execute on the input first.
  permutation(Firsts,HLayer), %Any permutation of the comparators that can execute first is a
      possible layer, since within a layer the order of execution is irrelevant.
  selectchkList(HLayer,C,CNew), %The comparators that are layered together are then removed
      from C.
  network_to_layered(CNew,TLayers). %The recursive call is made with the new network that
      consists of C minus the comparators that are now a layer.

possibleFirsts(C,L) :- %Finds all comparators that can execute first in a network.
  notFirsts(1,C,[],Result), %finds all comparators that can't execute first on the input.
  sort(Result,ProperResult), %The index list must be sorted for selectchkIndexList/3 to work
      correctly.
  selectchkIndexList(ProperResult,C,L). %these comparators are then removed from the network,
      the remaining is those that can execute first.

notFirsts(_,[_|[]],_,[]) :- !. %Finds out what comparators can't be the first ones by finding
    out what comparators that are NOT specifically after others
notFirsts(Index,C,Temp,Result) :-
  after(Index,C,R1), %Calls after/3 to find out what comparators must execute directly after
      the index passed as Index.
  union(R1,Temp,TempNew), %The comparators are then joined to the ones already found,
      initially that is an empty list.
  IndexNew is Index + 1, %The next index is prepared to be supplied to the next recursive call
      .
  length(C,LenC), %Length of the network C is found so it can be compared.
  (
    LenC > IndexNew, %indicates that there are more elements to go through.
    notFirsts(IndexNew,C,TempNew,Result) %makes the next recursive call with the next Index,
        and the accumulation of the indicies found previously.
  ;
    LenC = IndexNew, %indicates that all indices have been tested.
    union(R1,Temp,Result) %the result is the elements found in the latest call as well as all
        elements found in previous calls.
  ).
```

network_to_layered/2 works by recursively calling *possibleFirsts*/2 on the network C. This returns the a list of all elements that must execute as the first ones i.e. the comparators that must act on the input before any other. There can be several potential comparators in this list, provided that none of them operate on the same channels. All of comparators returned as the possible firsts are then layered together, since they can operate simultaneously on the input, and this is the definition of a layer. These comparators are then removed from the original network C and the new network is passed to the next recursive call, this is done until there are no more comparators left in C.
possibleFirsts/2 works by fiinding what comparators are NOT able to execute first, and then removing these comparators from the network. The remaining comparators are then those that can execute first. This is done by finding what comparators must execute after other comparators. All comparators that are in a list that must execute after an other comparator are then excluded, since they obviously can't be first.

```
after(I,C,Exlist) :- %Finds out what comparators in a network that HAS to execute AFTER index
    I.
  I1 is I+1, %Finds the index of the element after the one passed as parameter.
  directlyAfter(I,I1,C,Exlist). %makes the call to directlyAfter with the element passed as
      parameter as well as the one after it, and the network C.

directlyAfter(_,_,[],[]) :- !. %Used to find which comparators MUST execute after others for a
     given network.
directlyAfter(_,_,[_|[]],[]) :- !. %Base cases.
directlyAfter(E1Index,E2Index,C,Exlist) :-
  nth1(E1Index,C,E1),%Gets the elements that are represented by the indices passed as
      parameters.
  nth1(E2Index,C,E2),
  length(C,L),
  (
    sharesChannel(E1,E2), !, %If the 2 elements share a channel, cut.
    (
      E2Index = L, %The end of the network C is reached.
      Exlist = [E2Index|[]], ! %The result is obtained so cut.
    ;
      E3Index is E2Index + 1, %If the end of network C is not reached, next element is found.
      Exlist = [E2Index|T], %Element is added to the exclusion list, since they share a
          channel and therefore Element 2 has to execute after Element 1.
      directlyAfter(E2Index,E3Index,C,T1), %Make a recursive call checking what elements must
          execute after the element just added.
      directlyAfter(E1Index,E3Index,C,T2), %Make a recursive call and check if Element 3 must
          execute after element 1.
      union(T1,T2,T) %The complete exclusion list is the combined lists found above.
    )
  ;
    (
      E2Index = L, ! %If the elemts do not share a channel and the end of the network C is
          reached we are done.
    ;
      E3Index is E2Index + 1, %Next element is found.
      directlyAfter(E1Index,E3Index,C,Exlist) %The next recursive call is made without adding
          any elements to the exclusion list.
    )
  ).

sharesChannel(c(E1I,E1J),c(E2I,E2J)) :- %Determines if to comparators share an input channel
    or not.
  c(E1I,E1J), c(E2I,E2J), %Both must be legal comparators within the chosen model.
  (
  E1I = E2I, !;  %Each statement proves they share a channel, so we don't have to keep going
      if just 1 is proven true.
  E1J = E2J, !;
  E1I = E2J, !;
  E1J = E2I, !
  ).
```

directlyAfter/4 checks if the element found at index E2Index, must execute after the element found at E1Index, as well as all other elements that must execute after E2Index. This is then used to find out which elements can be first. Because all elements that must execute after another element obviously can't be first.

2.3.4 layered_to_network/2

```
layered_to_network(L,C) :- %Gives all possible networks that the layered network L can be
    turned into.
  flatten(L,FlatL), %Flattens the layered network.
  length(FlatL,Len), %Length of the flatten network is found.
  length(C,Len), %The length of the network it translated to must be the same length as the
      flattened network.
  layered_to_networkWorker(FlatL,C). %Calls the worker predicate to generate the actual
      possible translations.

layered_to_networkWorker([],[]). %base case.
layered_to_networkWorker(L,C) :-
  possibleFirsts(L,FirstEs), %Find out which elements are candidates for the first element in
      the translated network.
  member(FirstE,FirstEs), %one of these candidates is then chosen, with another possible
      solution being another choice if there are more than 1.
  [FirstE|T] = C, %The first/next element of network C is the element chosen from among the
      possibilites.
  selectchk(FirstE,L,LNew), %the chosen element is removed from the available elements.
  layered_to_networkWorker(LNew,T). %the next recursive call is made until we hit the base
      case, which is that there are no more elements left to choose from.
```

layered_to_network/2 works the opposite of *network_to_layered*, it does so by flattening any layered network representation passed. And then determining, one element at a time using *possibleFirsts*/2, which element is next/first.

2.3.5 Combining

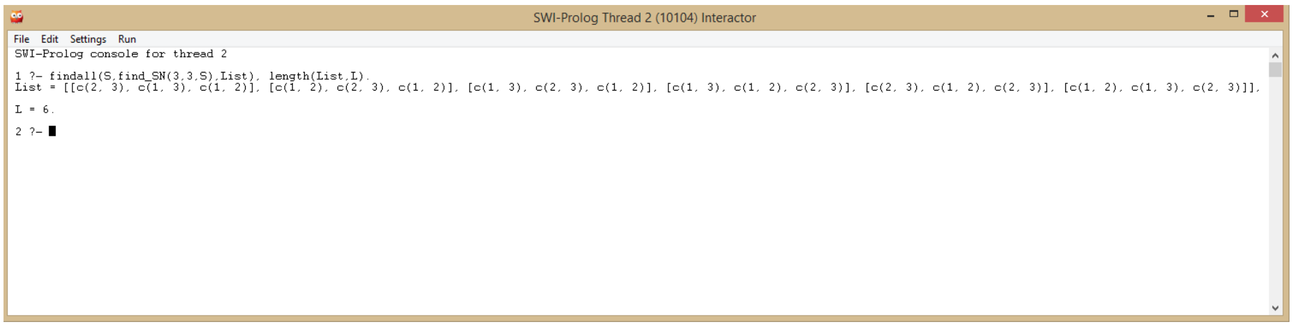
```
layer(C,L) :- %Combines layered_to_network, network_to_layered and layer, must be called with
    a network or variable as first parameter and variable of layered network as second.
  (var(C) %If C is a variable
  ->
    layered_to_network(L,C) %Then use layered_to_network/2
  ;
    network_to_layered(C,L) %Else use network_to_layered/2
  ).
```

This is simply done, with the *layer*/2 predicate, by testing if *C* is a variable, if it is, then a layered network must be translated, otherwise either a network must be layered or both parameters are instantiated. In the latter case, both *layered_to_network*/2 and *network_to_layered*/2 can be used, *network_to_layered* is chosen here. When translating a network to a layered representation, only optimal layerings are found i.e. even tho every layer of size 2 or greater can be split into smaller layers, and still be a correct layering, these are not found as solutions.

3 Example executions

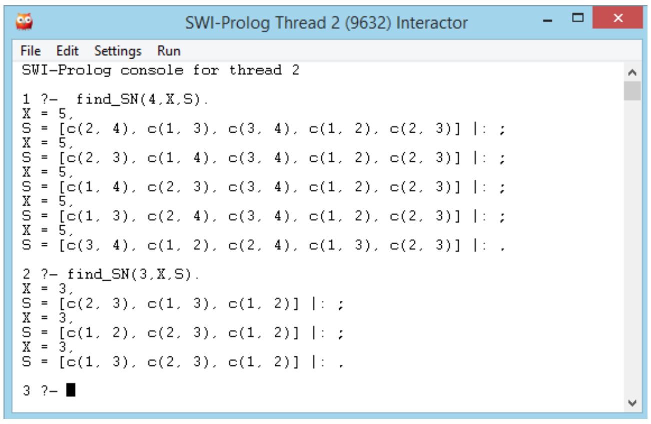
3.1 find_SN/3

Since $find_{S}N/3$ uses $is_{n}etwork/1$, $channels/2$, $run/3$ and $is\_SN/2$, an example of it should prove all the others. The following example finds all sorting networks of length 3 over 3 channels and counts how many of them there are.



```
SWI-Prolog console for thread 2

1 ?- findall(S,find_SN(3,3,S),List), length(List,L).
List = [[c(2, 3), c(1, 3), c(1, 2)], [c(1, 2), c(2, 3), c(1, 2)], [c(1, 3), c(2, 3), c(1, 2)], [c(1, 3), c(1, 2), c(2, 3)], [c(2, 3), c(1, 2), c(2, 3)], [c(1, 2), c(1, 3), c(2, 3)]],
L = 6.

2 ?-
```

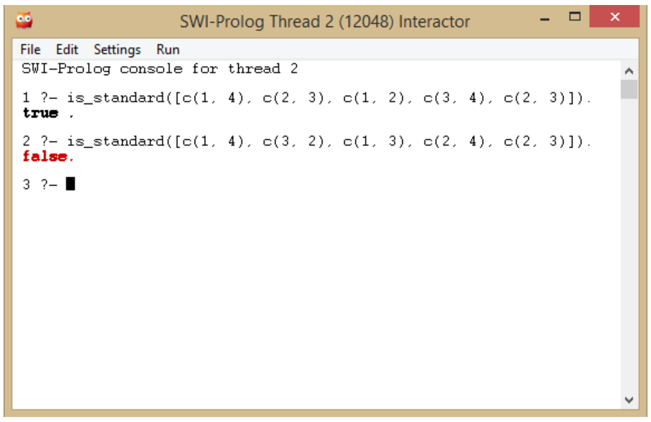
Functionality with variable as length:



```
SWI-Prolog console for thread 2

1 ?-  find_SN(4,X,S).
X = 5,
S = [c(2, 4), c(1, 3), c(3, 4), c(1, 2), c(2, 3)] |: ;
X = 5,
S = [c(2, 3), c(1, 4), c(3, 4), c(1, 2), c(2, 3)] |: ;
X = 5,
S = [c(1, 4), c(2, 3), c(3, 4), c(1, 2), c(2, 3)] |: ;
X = 5,
S = [c(1, 3), c(2, 4), c(3, 4), c(1, 2), c(2, 3)] |: ;
X = 5,
S = [c(3, 4), c(1, 2), c(2, 4), c(1, 3), c(2, 3)] |: .

2 ?- find_SN(3,X,S).
X = 3,
S = [c(2, 3), c(1, 3), c(1, 2)] |: ;
X = 3,
S = [c(1, 2), c(2, 3), c(1, 2)] |: ;
X = 3,
S = [c(1, 3), c(2, 3), c(1, 2)] |: .

3 ?-
```

3.2 is_standard/1

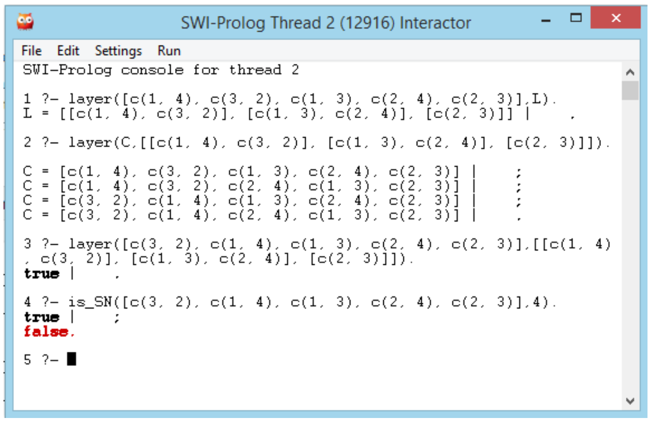
Since $is\_standard/1$ is the only predicate from section 2.2 that is implemented, it is the only one shown, a standard network and a non-standard is tested.



```
SWI-Prolog console for thread 2

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

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

3 ?-
```

3.3 layer/2

Since *layer*/2 uses all of the predicates in section 2.3, this will be tested, both by translating a network C to a layered network, and by translating it back again. And lastly to test if the two instantiated versions are layered/unlayered versions of each other.



```
SWI-Prolog console for thread 2

1 ?- layer([c(1, 4), c(3, 2), c(1, 3), c(2, 4), c(2, 3)],L).
L = [[c(1, 4), c(3, 2)], [c(1, 3), c(2, 4)], [c(2, 3)]] |    .

2 ?- layer(C,[[c(1, 4), c(3, 2)], [c(1, 3), c(2, 4)], [c(2, 3)]]).

C = [c(1, 4), c(3, 2), c(1, 3), c(2, 4), c(2, 3)] |    ;
C = [c(1, 4), c(3, 2), c(2, 4), c(1, 3), c(2, 3)] |    ;
C = [c(3, 2), c(1, 4), c(1, 3), c(2, 4), c(2, 3)] |    ;
C = [c(3, 2), c(1, 4), c(2, 4), c(1, 3), c(2, 3)] |    .

3 ?- layer([c(3, 2), c(1, 4), c(1, 3), c(2, 4), c(2, 3)],[[c(1, 4)
, c(3, 2)], [c(1, 3), c(2, 4)], [c(2, 3)]]).
true |    .

4 ?- is_SN([c(3, 2), c(1, 4), c(1, 3), c(2, 4), c(2, 3)],4).
true |    ;
false.

5 ?- █
```

First a network is translated to a layered representation, then it is translated back to unlayered, and several possiblities are show. One of these possibilities are chosen and compared against the layered network. Then the unlayered version that was chosen is proven to still be a sorting network.

4 Appendix

4.1 Code listing

4.1.1 project1.pl

```
:- use_module(library(lists)).

c(I,J) :-  %The definition of a comparator.
  integer(I), integer(J),
  I \= J.
  %I < J. Uncomment this line, and comment the one above to only consider standard networks as
      solutions, this is considerably faster.

sc(I,J) :-  %The definition of a comparator.
  integer(I), integer(J),
  I < J.

is_standard(C) :- %Checks that C is a network in the chosen model.
  C = [H|T],
  isStandardComparator(H),
  (is_standard(T) ; T = []).

isStandardComparator(Comparator) :- %Tests that the element Comparator is a comparator in the
    chosen model.
  Comparator = c(I,J), sc(I,J).

comparatorInRange(c(I,J),N) :- %Can be used to generate possible comparators for a N channel
    network, or to check if a comparator is possible in an N channel network.
  (
  var(N)
  ->
    N = J, c(I,J)
  ;
    between(1,N,I), between(1,N,J), c(I,J)
  ).

isComparator(Comparator) :- %Tests that the element Comparator is a comparator in the chosen
    model.
  Comparator = c(I,J), c(I,J).

possibleFirsts(C,L) :- %Finds all comparators that can execute first in a network.
  notFirsts(1,C,[],Result), %finds all comparators that can't execute first on the input.
  sort(Result,ProperResult), %The index list must be sorted for selectchkIndexList/3 to work
      correctly.
  selectchkIndexList(ProperResult,C,L). %these comparators are then removed from the network,
      the remaining is those that can execute first.

notFirsts(_,[_|[]],_,[]) :- !. %Finds out what comparators can't be the first ones by finding
    out what comparators that are NOT specifically after others
notFirsts(Index,C,Temp,Result) :-
  after(Index,C,R1), %Calls after/3 to find out what comparators must execute directly after
      the index passed as Index.
  union(R1,Temp,TempNew), %The comparators are then joined to the ones already found,
      initially that is an empty list.
  IndexNew is Index + 1, %The next index is prepared to be supplied to the next recursive call
      .
  length(C,LenC), %Length of the network C is found so it can be compared.
  (
    LenC > IndexNew, %indicates that there are more elements to go through.
    notFirsts(IndexNew,C,TempNew,Result) %makes the next recursive call with the next Index,
        and the accumulation of the indicies found previously.
  ;
    LenC = IndexNew, %indicates that all indices have been tested.
    union(R1,Temp,Result) %the result is the elements found in the latest call as well as all
        elements found in previous calls.
  ).

after(I,C,Exlist) :- %Finds out what comparators in a network that HAS to execute AFTER index
    I.
  I1 is I+1, %Finds the index of the element after the one passed as parameter.
  directlyAfter(I,I1,C,Exlist). %makes the call to directlyAfter with the element passed as
      parameter as well as the one after it, and the network C.
```

```
directlyAfter(_,_,[],[]) :- !. %Used to find which comparators MUST execute after others for a
     given network.
directlyAfter(_,_,[_|[]],[]) :- !. %Base cases.
directlyAfter(E1Index,E2Index,C,Exlist) :-
  nth1(E1Index,C,E1),%Gets the elements that are represented by the indices passed as
      parameters.
  nth1(E2Index,C,E2),
  length(C,L),
  (
    sharesChannel(E1,E2), !, %If the 2 elements share a channel, cut.
    (
      E2Index = L, %The end of the network C is reached.
      Exlist = [E2Index|[]], ! %The result is obtained so cut.
    ;
      E3Index is E2Index + 1, %If the end of network C is not reached, next element is found.
      Exlist = [E2Index|T], %Element is added to the exclusion list, since they share a
          channel and therefore Element 2 has to execute after Element 1.
      directlyAfter(E2Index,E3Index,C,T1), %Make a recursive call checking what elements must
          execute after the element just added.
      directlyAfter(E1Index,E3Index,C,T2), %Make a recursive call and check if Element 3 must
          execute after element 1.
      union(T1,T2,T) %The complete exclusion list is the combined lists found above.
    )
  ;
    (
      E2Index = L, ! %If the elemts do not share a channel and the end of the network C is
          reached we are done.
    ;
      E3Index is E2Index + 1, %Next element is found.
      directlyAfter(E1Index,E3Index,C,Exlist) %The next recursive call is made without adding
          any elements to the exclusion list.
    )
  ).

sharesChannel(c(E1I,E1J),c(E2I,E2J)) :- %Determines if to comparators share an input channel
    or not.
  c(E1I,E1J), c(E2I,E2J), %Both must be legal comparators within the chosen model.
  (
  E1I = E2I, !;  %Each statement proves they share a channel, so we don't have to keep going
     if just 1 is proven true.
  E1J = E2J, !;
  E1I = E2J, !;
  E1J = E2I, !
  ).

selectchkList([],List2,List2):-!.
selectchkList([H|T],List2,Result) :- %Removes every element in List1 from List2 with selectchk
    /3
  (
    member(H,List2)
  ->
    selectchk(H,List2,Result1)
  ;
    Result1 = List2
  ),
  selectchkList(T,Result1,Result).

selectchkIndexList([],List2,List2):-!. %Removes the indices given in the first list from the
    second list and the result is the third list.
selectchkIndexList(_,[],[]):-!.
selectchkIndexList([H|T],List2,Result) :- %Index list has to be sorted.
  (
  length(List2,L2),
  H =< L2, H > 0
  ->
    nth1(H,List2,_,Result1)
  ;
    Result1 = List2
  ),
  (
    T \= [],
    subtractFromList(1,T,T1),
    selectchkIndexList(T1,Result1,Result)
  ;
    T = [],
    Result = Result1,!
  ).

subtractFromList(_,[],[]):-!. %Subtracts an integer from every element in a list.
subtractFromList(Int,[H|T],[ResH|ResT]) :-
  ResH is H - Int,
  subtractFromList(Int,T,ResT).

is_network(C) :- %Checks that C is a network in the chosen model.
  C = [H|T],
  isComparator(H),
  (T = [] ; is_network(T)).

channels(C,N) :- %Assignment defined predicate.
  ( var(N)
  ->
    sort(C,Csorted),
    Csorted = [H|T],
    (T = [] ; channels(T,N)),
    comparatorInRange(H,N)
  ;
    C = [H|T],
    (T = [] ; channels(T,N)),
    comparatorInRange(H,N)
  ).

channelsOfLengthL(N,L,[H|T]) :- %Generates all possible networks of length L, over N channels.
  length([H|T],L),
  (T = [] ; channels(T,N)),
  comparatorInRange(H,N).

partialSolutionsOfLengthL(N,L,C):- %Even more limited solution space.
  length(C,L),
  findall(List,(between(1,N,X),possibleComparatorsForChannelX(X,N,List)),PList),
  Rem is N,
  possibleSolutionsOfLengthL(C,N,L,Rem,PList).

possibleSolutionsOfLengthL(N,L,C):- %Generates a more limited solution space that should
    contain a solution to a network on any number of channels.
  length(C,L),
  findall(List,(between(1,N,X),possibleComparatorsForChannelX(X,N,List)),List),
  permutation(List,PList),
  Rem is N,
  possibleSolutionsOfLengthL(C,N,L,Rem,PList).
possibleSolutionsOfLengthL(C,N,L,0,_):-
  Lnew is L - N,
  channelsOfLengthL(N,Lnew,C).
possibleSolutionsOfLengthL([H|T],N,L,Rem,[PListH|PListT]) :-
  member(H,PListH),
  RemNew is Rem - 1,
  (T = [] ; possibleSolutionsOfLengthL(T,N,L,RemNew,PListT)).

possibleComparatorsForChannelX(X,N,PList) :- %Generates a list PList that contains all
    comparators that contain channel X possible over N channels.
  findall(C,(comparatorInRange(C,N),(C = c(_,X); C = c(X,_))),PList).

run(C,Input,Output) :- %Doesn't work for input not using all channels.
  length(Input,L),
  channels(C,L),
  same_length(Input,Output),
  runWork(C,Input,Output).

runWork(C,Input,Output) :-
  C = [c(I,J)|T],
  nth1(I,Input,Xi),
  nth1(J,Input,Xj),
  (Xi > Xj, swap(Input,I,J,InputNew), (T = [], Output = InputNew ; runWork(T,InputNew,Output))
      ;
  Xi =< Xj, (T = [], Output = Input ; runWork(T,Input,Output))).

swap(As,I,J,Cs) :- %Swaps the the elements I and J, in the lists As and Cs.
   same_length(As,Cs),
   append(BeforeI,[AtI|PastI],As),
   append(BeforeI,[AtJ|PastI],Bs),
   append(BeforeJ,[AtJ|PastJ],Bs),
   append(BeforeJ,[AtI|PastJ],Cs),
   Inew is I - 1, % This is done to use the same indices as nth1/3.
   Jnew is J - 1,
   length(BeforeI,Inew),
   length(BeforeJ,Jnew).

binaryList(List,Len) :- %Generates binary number lists for use in testing if a comparator
    network is a sorting network.
  List = [H|T],
  between(0,1,H),
  (Len > 1, LenNew is Len - 1, binaryList(T,LenNew) ; Len = 1, T = []).

isSorted([_|[]]):- !. %Tests wheter an input is sorted correctly, in ascending order.
isSorted([F,S|T]) :-
  F =< S,
  isSorted([S|T]).

is_SN(S,N) :- %Tests if network S is a sorting network over N channels.
  is_network(S),
  findall(X, binaryList(X,N), [_|TestBatch]),
  comprehensiveTest(S,N,TestBatch).

comprehensiveTest(S,N,[H|T]) :- %Used in testing for sorting networks.
  run(S,H,Sorted),
  isSorted(Sorted),
  (T = [] ; comprehensiveTest(S,N,T)).

find_SN(N,K,S) :- %Finds a sorting network S of length K over N channels, Takes VERY long for
    networks of more than 4 channels.
  channelsOfLengthL(N,K,S),
  is_SN(S,N).

find_SN1(N,K,S) :- %Finds a sorting network S of length K over N channels, is faster than
    find_SN. But doesn't find all possible solutions.
  possibleSolutionsOfLengthL(N,K,S),
  is_SN(S,N).

find_SN2(N,K,S) :- %Finds a sorting network S of length K over N channels, is faster than
    find_SN1. But finds fewer possible solutions.
  partialSolutionsOfLengthL(N,K,S),
  is_SN(S,N).

network_to_layered([],[]). %Converts a network C into a layered network representation L of C.
network_to_layered(C,[HLayer|TLayers]) :-
  possibleFirsts(C,Firsts), %Returns the indices of the comparators in C that possibly can
      execute on the input first.
  permutation(Firsts,HLayer), %Any permutation of the comparators that can execute first is a
      possible layer, since within a layer the order of execution is irrelevant.
  selectchkList(HLayer,C,CNew), %The comparators that are layered together are then removed
      from C.
  network_to_layered(CNew,TLayers). %The recursive call is made with the new network that
      consists of C minus the comparators that are now a layer.

layered_to_network(L,C) :- %Gives all possible networks that the layered network L can be
    turned into.
  flatten(L,FlatL), %Flattens the layered network.
  length(FlatL,Len), %Length of the flatten network is found.
  length(C,Len), %The length of the network it translated to must be the same length as the
      flattened network.
  layered_to_networkWorker(FlatL,C). %Calls the worker predicate to generate the actual
      possible translations.

layered_to_networkWorker([],[]). %base case.
layered_to_networkWorker(L,C) :-
  possibleFirsts(L,FirstEs), %Find out which elements are candidates for the first element in
      the translated network.
  member(FirstE,FirstEs), %one of these candidates is then chosen, with another possible
      solution being another choice if there are more than 1.
  [FirstE|T] = C, %The first/next element of network C is the element chosen from among the
      possibilites.
  selectchk(FirstE,L,LNew), %the chosen element is removed from the available elements.
  layered_to_networkWorker(LNew,T). %the next recursive call is made until we hit the base
      case, which is that there are no more elements left to choose from.

layer(C,L) :- %Combines layered_to_network, network_to_layered and layer, must be called with
    a network or variable as first parameter and variable of layered network as second.
  (var(C) %If C is a variable
  ->
```

```
    layered_to_network(L,C) %Then use layered_to_network/2
  ;
    network_to_layered(C,L) %Else use network_to_layered/2
  ).
```