Programming Language Homework2 Report

張財實
資訊三乙
F74055047

Problem 1:

Execution： $ swipl -q -s plhw3p1.pl

Sample：

Input : 1000
Output : 3 997
17 983
23 977
29 971
47 953
53 947
59 941
71 929
89 911
113 887
137 863
173 827
179 821
191 809
227 773
239 761
257 743
281 719
317 683
347 653
353 647
359 641
383 617
401 599
431 569
443 557
479 521
491 509

this is an updated output, since we are required to list all pair

Implementation:

I assume all the input is a valid even number, greater than 2 start from 4, and so on. Then first, split the input number to 2 number, which means each number of left hand side and right hand side sum up will be the input number itself, but start with NUM/2.

For example 10 (split)→ 5, 5

Then, if the initial pair does not meet the criteria, decrease left and increase right until the criteria is met.

For example 10 (split)→ 5, 5 → 4, 6→ 3, 7 → 2, 8→ 1, 9 …

But we will not go so far as 5, 5 already met the criteria, just output it.

Main component:
- To test a prime number, a function predicate
- To test both of left and right are pimes, a function predicate

Code Explanation:

```
is_prime(Number, P):-
  N is truncate(sqrt(Number)),
  loop_divisor(N,Number, 100, Y),
  P = Y.
```

This is_prime predicate takes the NUMBER, then use the homwork 1 prime approach to test the number, get its square root first, then we will know the number's divisor can be only up to its square root, so we enter a loop call loop_divisor.

The result of the loop_divisor will give a return value to Y, then assign Y to P, make the conjecture predicate to determine the result.

```
loop_divisor(_, _, 0, not_prime).
loop_divisor(1, _, _, prime).



loop_divisor(N,Number, _, Y):-
  N > 0,
  X is mod(Number, N),
  M is N - 1,
  loop_divisor(M,Number, X, Y).
```

The divisor N must greater than 0, consider TRUE
We can check divisible by the mode predicate, and loop it again.

The first facts above says once the loop encounter a remainder(mod result) is 0, it means the number cannot be a prime, and return not_prime to previous layer.

The second facts says after the divisor loop through the end (N → …→ .. 2) but not 1, (meet 1), then the first fact does not occur, telling that the number is a prime, return prime.

```
conjecture(L, prime, R, prime):-
  L1 is L + 1,
  R1 is R - 1,
  format("Output: ~w ~w ~n", [L1,R1]).

conjecture(L, _, R, _):-
  is_prime(L, X),
  is_prime(R, Y),
  L1 is L - 1,
  R1 is R + 1,
```

```
  conjecture(L1, X, R1, Y).
```

The first predicate conjecture says that when both the 2nd and 4th argument given is prime, then the L and R value taken will the the answer for the NUMBER.
But the second predicate involved recursions, so the L and R is shifted at the last time, so we have to shift it back to the prime number.

The second predicate takes L and R as query numbers, inside this predicate, we will be checking whether both L and R is a prime number , then get the new L and R in advance. By each iteration, is_prime will tell L and R's identity (prime or not_prime), use these new L and R and result to recursively call conjecture again, until it meets the first predicate criteria.

```
program:- program_loop.

program_loop:-
  write("Input : "),
  readln(X),
  [Number|_] = X,
  Y is div(Number,2),
  conjecture(Y, not_prime, Y, not_prime),
  program_loop.
```

Program start just call program loop, and program loop never stop.

In program loop, we have to take the input, note that we must use readln (I try read several times, readln is the best way), and the read input is a type of list. Take of the list bracket and divide the number first (as mentioned in my implementation). Do the conjecture and boom, done!

```
main:-
  program,
  halt.

:- initialization(main).
```

Scripting format, must halt otherwise end of execution return command line.

Review :
Hard to understand prolog at first.

Problem 2:

Execution： $ swipl -q -s plhw2p2.pl
Sample :
[changchaishi@iknowright plhw3]$ swipl -q -s plhw3p2.pl
|: 6
|: 1 2
|: 2 3
|: 1 4
|: 4 5
|: 4 6
|: 3
|: 3 4

|: 5 6
|: 1 2
1
4
1

Implementation:

We have three main part here,

1) Get the family facts

I already assume the child will only have one parent, then every line we will be adding a fact parent(a,b) telling a is b's parent. We have to do it dynamically since we declare the fact by the variable from stdin.

2) Get the query tree

After building the facts, the facts are store in background of the process, then we will be taken N query of 2 node, checking their lowest common ancestor. But, we need to store the N entry first, we do all the query at once.

3) Examine the input

Now we got parent facts, query input, we can start analyze the lowest common ancestor of the query at once, then print the output. This is the stage to process the algorithm.

Code Explanation:

```
:- dynamic parent/2.
parent(root,_).
```

To dynamically form parent facts, allow variable to form facts.

If a variable does not have parent, make the parent of this variable root.

```
append([],X,X).
append([X|Y],Z,[X|W]) :- append(Y,Z,W).
```

Append two list and the result return to the third argument, I mean I will use it this way, prolog and have a lot of return combinations thought.

```
member(X,[X|_], success).
member(_,[], fail).
member(X,[_|T],A) :- member(X,T,A).
```

Telling an variable is in the list or not. Actually I can use the 'member' from prolog itself but, I am too dump to use it, so I desired a return from 3rd argument so I can use if else afterwards.

```
program:-
  readln(In),
  [I|_] = In,
  Input is I - 1,
  loop_tree(Input),
  readln(Qin),
  [Qinput|_] = Qin,
  loop_query(Qinput,[],Pairs),
  output(Pairs).
```

This time I will explain the code from beginning, not from the end to bigin.

Program:
1) read first N, get the N, ask loop_tree to loop N times
2) read M, get the M, ask loop_query to loop M times
3) output is to make the query happen, and produce outcome

```
loop_tree(0).
loop_tree(X):-
  readln(I),
  parse_pair(I, A, B),
  asserta(parent(A, B)),
  M is X - 1,
  loop_tree(M).
```

It is a form of basic loop.

I is a list containing two number [a, b]
parse_pair is to get the number parse the list. [a,b] to A, B
Now we have number A and B, we have to make facts, they are parent child relation, add the facts dynamically we have to use asserta(FACT)

Well, assume after this stage we got all relation facts

```
loop_query(0, Temp, Pairs):-
  Pairs = Temp.
loop_query(X, Temp, Pairs):-
  readln(I),
  append(Temp, [I], NewTemp),
  M is X - 1,
  loop_query(M, NewTemp, Pairs).
```

Same as above, but now, we just store these list to a list, so it is list of list, we wait for output to call the list and do its job. (append use here)

```
parse_pair(P, A, B):-
  [A|T] = P,
  [B|_] = T.
```

Implementation the parse list.

```
output([]).
  output([H|T]):-
  parse_pair(H, A, B),
  get_lists(A1, A2 , A, B),
  common_ancester(A1, A2),
  output(T).
```

Output is a loop, to loop through all the query list and print the output.
Inside it, we first parse the list, get of of the node.
Then get_list : get_list is to get all the ancestor of a node in list
Now we have two list, common ancestor predicate is to compare their ancestor and print the result.

```
get_lists(A1, A2 ,C1, C2):-
```

```
  parents([C1], A1, C1),
  parents([C2], A2, C2).
```

Ask parents predicate to give a list of the node

```
parents(Temp, Result, root):-
  Result = Temp.

parents(I, F,X):-
  parent(Y1, X),
  append(I, [Y1], Fnew),
  NewF = Fnew,
  parents(NewF, F, Y1).
```

Every time the node get its parent, the parent is going to be added to the return list, then next loop(recursion) the node will become ‘parent’, doing it recursively until the parent reaches root, then stop. Return the ancestor list.

```
common_ancestor([root|_], _).
common_ancestor([H|T], L2):-
  member(H, L2, X),
  X = success
  ->
  writeln(H)
  ;
  common_ancestor(T, L2).
```

Now it is time to call common ancestor when we got both A, B’s ancestor list.

Since the parents predicates added the list in reverse order, that means the list will be like:

[node, parent, grandparent, granddpp, …. , root]

So we can take the A list as the one to iterate and B list to compare, once the first encounter ancestor from the A is in member of B ancestor, that means that ancestor will be the lowest, so we print the answer.

```
main:-
  program,
  halt.

:- initialization(main).
```

Here we have it !

Review : The script is not working as in command line mode.

Problem 3:

Execution： $ swipl -q -s plhw3p3.pl
Sample :
[changchaishi@iknowright plhw3]$ swipl -q -s plhw3p3.pl
|: 6 6
|: 1 2
|: 2 3
|: 3 1

|: 4 5
|: 5 6
|: 6 4
|: 2
|: 1 3
|: 1 5
Yes
No

Implementation:
The framework is mostly same with problem2, only some predicates are different.

We have three main part here,
1) Get the family facts
I already assume the child will only have one parent, then every line we will be adding a fact parent(a,b) telling a is b's parent. We have to do it dynamically since we declare the fact by the variable from stdin.
2) Get the query tree
After building the facts, the facts are store in background of the process, then we will be taken N query of 2 node, checking their lowest common ancestor. But, we need to store the N entry first, we do all the query at once.

3) Examine the input
Now we got parent facts, query input, we can start analyze the lowest common ancestor of the query at once, then print the output. This is the stage to process the algorithm.

Code Explanation:

```
:- dynamic relative/2.
  rev_relative(A,B):-
  asserta(relative(B,A)).
```

Instead of parent, we can call two node connecting relatives.

And we provide reverse relative, such as a relative to b so, b is relative to a too.

```
union([],[],[]).
union(List1,[],List1).
union(List1, [Head2|Tail2], [Head2|Output]):-
  \+(member(Head2,List1)), union(List1,Tail2,Output).
union(List1, [Head2|Tail2], Output):-
  member(Head2,List1), union(List1,Tail2,Output).
```

Union is essential, just like disjoint set.

The LOOP TREE and LOOP QUERY are SAME with PROBLEM2, so just focus on OUTPUT implementation.

```
output([]).
output([H|T]):-
  [A|T2] = H,
  [B|_] = T2,
  do_union_self([B],[B],[B], ListB),
  check_connected(A,ListB),
```

```
  output(T).
```

do_union_self, a predicate to spread and find its relativem return a list.

For example if B do its union, if it gets [a, b, c, d]
Then we just check whether A is inside the list, if it is, they are connected.

So, problem 3 is just to spread the union and check whether each other are in the same group(list)

```
do_union_self(Ini, [], Temp, Final):-
  length(Ini, X),
  length(Temp, Y),
  X \= Y
  ->
  do_union_self(Ini, Ini, Ini, Final)
  ;
  Final = Ini.

do_union_self(Ini, [Ele|T], Temp, Final):-
  findall(X,relative(Ele,X), AllRelative),
  union(Ini, AllRelative, UnionList),
  do_union_self(UnionList, T, Temp, Final).
```

What do_self_union is actually doing is, for example:

1-2
2-3
3-4
1-5

do_self_union(1)
1 → 1-2-5 (union 1-2 then 1-5) length = 3
2 → traverse 1-2-5
3 → 1-2-3-5 (union from 2-3) length =4
4 → traverse 1-2-3-5
5 → 1-2-3-4-5 (union from 3-4) length =5
6 → traverse 1-2-3-4-5 (union nothing) length = 5

Now when traverse thru the list does not make the union list grow, means that the union is finale, then return it.

```
check_connected(A,ListB):-
member(A,ListB),
writeln("Yes").

check_connected(A,ListB):-
\+member(A,ListB),
writeln("No").
```

Just to print output if we know the union everything will be easy.

```
main:-
  program,
```

```
  halt.

:- initialization(main).
```

Here we have it !