Programming Language Homework 3

Author: F74054122 林家緯

Environment

Ubuntu 16.04 LTS

swipl

Problem 1 Goldbach's conjecture

題目說明

Goldbach's conjecture: 任何一個大於 2 的偶數皆可以拆分成兩個質數

找出一個數字的所有 Goldbach's conjecture, 輸出時兩個數字要按大小排列

執行

```
$ swipl -q -s problem1.pl
Input (0 to exit): 4
Output: 2 2
Input (0 to exit): 100
Output: 3 97
Output: 11 89
Output: 17 83
Output: 29 71
Output: 41 59
Output: 47 53
Input (0 to exit): 0
```

程式碼說明

```
is_prime(2).
is_prime(3).
is_prime(P) :- integer(P), P > 3, P mod 2 =\= 0, \+ has_factor(P,3).

has_factor(N,L) :- N mod L =:= 0.
has_factor(N,L) :- L * L < N, L2 is L + 2, has_factor(N,L2).

goldbach(4) :-
    write("Output: 2 2"), nl, main.
goldbach(N) :-
    N mod 2 =:= 0, N > 4 -> goldbach(N,3);
    write("Invalid input! Please enter an even number greater than 2!"), nl, main.
```

```
goldbach(N,P) :- Q is N-P, is_prime(Q), Q >= P, write("Output: "), write(P), write(" "),
write(Q), nl.
goldbach(N,P) :- P < N, next_prime(P,P1), goldbach(N,P1).

next_prime(P,P1) :- P1 is P + 2, is_prime(P1), !.
next_prime(P,P1) :- P2 is P + 2, next_prime(P2,P1).

main :-
    write("Input (0 to exit): "), readln([N|_]),
    N > 0 -> (goldbach(N), fail; main);
    halt.

:- initialization(main).
```

程式總共分爲 5 個部分:

1. 判斷質數
2. 判斷是否有因數
3. 計算 Goldbach
4. 尋找下一個質數
5. main函數，處理輸入

1. 判斷質數

```
is_prime(2).
is_prime(3).
is_prime(P) :- integer(P), P > 3, P mod 2 =\= 0, \+ has_factor(P,3).
```

定義 2, 3 爲質數

定義質數 P 爲 整數 & P > 3 & P 不爲偶數 & 在(3, sqrt(P))的範圍內找不到 P 的因數

2. 判斷是否有因數

```
has_factor(N,L) :- N mod L =:= 0.
has_factor(N,L) :- L * L < N, L2 is L + 2, has_factor(N,L2).
```

N 爲數字本身，L 爲目前待測因數

當 N mod L 爲 0 時，L 是 N 的一個因數，返回 true

當 L 小於 sqrt(N) 時，持續把 L + 2，並判斷新的 L 是否爲 N 的因數

3. 計算 Goldbach

```
goldbach(4) :-
    write("Output: 2 2"), nl, main.
goldbach(N) :-
    N mod 2 =:= 0, N > 4 -> goldbach(N,3);
    write("Invalid input! Please enter an even number greater than 2!"), nl, main.

goldbach(N,P) :- Q is N-P, is_prime(Q), Q >= P, write("Output: "), write(P), write(" "),
write(Q), nl.
goldbach(N,P) :- P < N, next_prime(P,P1), goldbach(N,P1).
```

當求 4 的 Goldbach 時，直接輸出，因爲是 Special Case

`goldbach(N)` 尋找 N 的 Goldbach，同時做 error check，只有 偶數 & N>4 才會繼續找 Goldbach

`goldbach(N, P)` N 爲需要找 Goldbach 的數字，P 爲 Goldbach 中較小的數字

1. 定義 Q 爲 N-P，當 Q 也爲質數時，則輸出 (因爲 P 的產生一定是質數)
2. 通過 `next_prime` 一直找出 P 的下一個質數，並驗證是否達到 Goldbach 的標準

4. 尋找下一個質數

```
next_prime(P,P1) :- P1 is P + 2, is_prime(P1), !.
next_prime(P,P1) :- P2 is P + 2, next_prime(P2,P1).
```

用於尋找 P 後的下一個質數，P1

方法爲把 P + 2 後驗證是否爲質數，若非質數，則繼續 +2 直到爲質數

5. main函數，處理輸入

```
main :-
    write("Input (0 to exit): "), readln([N|_]),
    N > 0 -> (goldbach(N), fail; main);
    halt.
```

使用 `readln`，只讀取 Input 的第一個 token (i.e. 第一個數字)

使用 `fail` 來強制 backtracking，達到輸出所有結果的效果

增加 N = 0 時退出程序的功能

Problem 2 Lowest Common Ancestor

題目說明

給定 N 個 node 的 N-1 個 Parent-child 關系，計算某兩個 node 的 Lowest Common Ancestor

執行

```
$ swipl -q -s problem2.pl
```

```
Input Relations...
|: 6        -> # of nodes
|: 1 2      -> node 1 is the parent node of node 2
|: 2 3      -> node 2 is the parent node of node 3
|: 1 4      -> node 1 is the parent node of node 4
|: 4 5      -> node 4 is the parent node of node 5
|: 4 6      -> node 4 is the parent node of node 6
Start Query...
|: 3        -> # of queries
|: 3 4      -> Which node is the LCA of node 3 and node 4?
LCA is: 1
|: 5 6      -> Which node is the LCA of node 5 and node 6?
LCA is: 4
|: 1 2      -> Which node is the LCA of node 1 and node 2?
LCA is: 1
```

(注: `->` 後爲輸入說明，執行時無需輸入 `->` 的內容)

程式碼說明

```
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(X,B),ancestor(A,X).

lca(A,B) :-
  A==B -> write("LCA is: "), write(A), nl;
  ancestor(A,B) -> write("LCA is: "), write(A), nl;
  parent(X,A),lca(X,B).

add_relation(N) :-
  N > 0
  -> readln([A|R]), nth0(0, R, B), assert(parent(A, B)), add_relation(N-1);
  write("Start Query..."), nl, readln([M|_]), query_lca(M).

query_lca(M) :-
  M > 0
  -> readln([A|R]), nth0(0, R, B), lca(A, B), query_lca(M-1);
  halt.

main :-
  write("Input Relations..."), nl,
  readln([N|_]), add_relation(N-1).

:- initialization(main).
```

程式共分爲 5 個部分:

1. 定義 ancestor 關系
2. 計算 A, B 的 LCA
3. 輸入 parent-child relation
4. 查詢 LCA
5. main函數，處理輸入

1. 定義 **ancestor** 關系

```
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(X,B),ancestor(A,X).
```

1. A 是 B 的 parent，則 A 也是 B 的 ancestor
2. 對於任意X，X 是 B 的 parent，A 是 X 的 ancestor，則 A 是 B 的 ancestor

2. 計算 **A, B** 的 **LCA**

```
lca(A,B) :-
  A==B -> write("LCA is: "), write(A), nl;
  ancestor(A,B) -> write("LCA is: "), write(A), nl;
  parent(X,A),lca(X,B).
```

根據 LCA 的定義，可以列出以下 pseudo code

```
LCA(x, y) =
{ x                  , if x = y or x = parent(y) }
{ LCA(parent(x), y)  , otherwise }
```

1. 當 x 爲 y 的 parent 時，x 就爲 LCA
2. 當 x 和 y 相同時，x 爲 LCA
3. 當不符合 1,2 時，尋找 x 的 parent 與 y 的 LCA

3. 輸入 **parent-child relation**

```
add_relation(N) :-
  N > 0
  -> readln([A|R]), nth0(0, R, B), assert(parent(A, B)), add_relation(N-1);
  write("Start Query..."), nl, readln([M|_]), query_lca(M).
```

使用 `readln` 讀入使用者的輸入，A 爲第一個數字，R 爲那一行剩下的 Token，再從 R 中取出第一個 Token，完成讀入兩個 Input 的效果

使用 `assert(parent(A, B))` 來加入 fact

每讀完一條 Input 就將N -1，當 N = 0 時則開始讀入 Query 的個數， M 爲輸入的 Query 次數

4. 查詢 **LCA**

```
query_lca(M) :-
  M > 0
  -> readln([A|R]), nth0(0, R, B), lca(A, B), query_lca(M-1);
  halt.
```

和 3 相同的處理方式，讀入兩個數字 A, B，查詢 `lca(A, B)` 然後將可查詢次數 - 1，可查詢次數用盡則結束程式

5. **main**函數，處理輸入

```
main :-
  write("Input Relations..."), nl,
  readln([N|_]), add_relation(N-1).
```

主要用來讀取一開始代表 node 個數的數字 N， 因爲會有 N-1 個 relation，所以將 N-1 傳入 `add_relation()`

Problem 3 Reachable

題目說明

給定每個 node 之前的 edge 連接關系，計算某兩個 node 是否 Reachable

執行

```
$ swipl -q -s problem3.pl
Input Edges...
|: 6 6      -> # of nodes and # of edges
|: 1 2      -> node 1 and node 2 are connected
|: 2 3      -> node 2 and node 3 are connected
|: 3 1      -> node 3 and node 1 are connected
|: 4 5      -> node 4 and node 5 are connected
|: 5 6      -> node 5 and node 6 are connected
|: 6 4      -> node 6 and node 4 are connected
Start Query...
|: 2        -> # of queries
|: 1 3      -> Are node 1 and node 3 connected?
Yes
|: 1 5      -> Are node 1 and node 5 connected?
No
```

(注: `->` 後爲輸入說明，執行時無需輸入 `->` 的內容)

程式碼說明

```
input_edges(N) :-
    N > 0
    -> readln([A|R]), nth0(0, R, B), assert(edge(A, B)), assert(edge(B, A)),input_edges(N-
1);
    write("Start Query..."), nl, readln([M|_]), query(M).

query(M) :-
    M > 0
    -> readln([A|R]), nth0(0, R, B),
        (start_dest(A, B) -> write("Yes"), nl; write("No"), nl), query(M-1);halt.

start_dest(S,D) :-
    start_dest_(S,D,[]).

start_dest_(D,D,_Visited).
```

```
start_dest_(S,D,Visited) :-
   maplist(dif(S),Visited),
   edge(S,X),
   start_dest_(X,D,[S|Visited]).

main :-
    write("Input Edges..."), nl,
    readln([_|R]), nth0(0, R, N), input_edges(N).

:- initialization(main).
```

程式共分爲 4 個部分:

1. 輸入各 node 之間的 edge
2. 查詢 Reachable
3. Reachable 的演算法
4. main函數，處理輸入

1. 輸入各 **node** 之間的 **edge**

```
input_edges(N) :-
    N > 0
    -> readln([A|R]), nth0(0, R, B), assert(edge(A, B)), assert(edge(B, A)),input_edges(N-
1);
    write("Start Query..."), nl, readln([M|_]), query(M).
```

使用和 Problem 3(3) 類似的做法，讀入兩個數字 A, B，輸入 node 之間的連接關系，因爲爲無向圖，所以加入 `edge(A,B)` 之外另外加入 `edge(B,A)`

輸入完成後呼叫查詢的函數

2. 查詢 **Reachable**

```
query(M) :-
    M > 0
    -> readln([A|R]), nth0(0, R, B),
        (start_dest(A, B) -> write("Yes"), nl; write("No"), nl), query(M-1);halt.
```

使用和 Problem 3(3) 類似的做法，讀入兩個數字 A, B，呼叫 `start_dest(A, B)` 來查詢是否 Reachable，如果是 Reachable 則輸出 Yes，否則輸出 No

3. **Reachable** 的演算法

```
start_dest(S,D) :-
    start_dest_(S,D,[]).

start_dest_(D,D,_Visited).
start_dest_(S,D,Visited) :-
   maplist(dif(S),Visited),
   edge(S,X),
   start_dest_(X,D,[S|Visited]).
```

`start_dest(S, D)` 提供呼叫界面，實際上交由 `start_dest_(S, D, [])`處理

計算 Reachable 的方式是檢查兩個 node 之間是否存在一個路徑可以到達

爲了避免圖中有 Cycle, 加入一個 Visited 的 list，確認每個 node 最多訪問一次

當 `start_dest_(D, D, _Visited)` 時，代表存在一條路徑，返回 true

`start_dest_(S, D, Visited)` 先確認 S 不在 Visited 中，接着遍歷每個與 S 相連的 `edge(S, X)`，接着由 X 當做起點呼叫，並將 S 加入 `Visited`

4. main函數，處理輸入

```
main :-
    write("Input Edges..."), nl,
    readln([_|R]), nth0(0, R, N), input_edges(N).
```

使用類似 Problem 3(3) 的方法，用來讀入 edge 的個數（好像 node 的個數並不重要）