U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2024/2025 – 1st Semester

Prolog

Collecting Solutions / Graphs and Trees

2024/12/04

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Collecting Solutions
- Graphs and Search
  - Puzzles and Games
- Binary Trees

Collecting Solutions

- So far, we obtained multiple solutions to a query interactively, in the terminal, one by one
- Or by accumulating the results of a query in a list

```
get_all_children(Parent, Children):-
      get_children(Parent, Children, []).

get_children(Parent, Children, Temp):-
      parent(Parent, Child),
      not (member(Child, Temp)), !,
      get_children(Parent, Children, [Child|Temp]).
get_children(_Parent, Children, Temp):-
      reverse(Temp, Children).
```

Is the cut necessary?

Why is this approach inefficient?

Collecting Solutions

- Prolog provides three predicates to obtain multiple solutions to a query: *findall*, *bagof* and *setof*
  - They allow systematic collection of answers to any goal
  - The template is similar to all three predicates

```
findall(Term, Goal, List).
```

See section 4.13 of the SICStus Manual for more information on collecting solutions

findall

- ***findall*** finds all solutions, including repetitions if present
  - If there are no solutions, an empty list is returned

```
| ?- findall(Child, parent(homer, Child), Children).
Children = [lisa, bart, maggie] ? ;
no

| ?- findall(Parent, parent(Parent, _Child), List).
List = [homer,homer,homer,marge,marge,marge] ? ;
no

| ?- findall(Child, parent(bart, Child), List).
List = [] ? ;
no
```

findall

- We can use a conjunctive goal (parentheses are required)

```
| ?- findall(C, ( parent(homer, C), female(C) ), Daughters).
Daughters = [lisa, maggie] ? ;
no
```

- We can obtain more than one variable using a compound term

```
| ?- findall(Parent-Child, parent(Parent, Child), L).
L = [homer-lisa, homer-bart, homer-maggie, marge-lisa, ...] ? ;
no
```

- If all we want is a count, we can use anything

```
| ?- findall(_, parent(homer, _C), _L), length(_L, N).
N = 3 ? ;
no
```

bagof

- ***bagof*** has similar behavior, but results are grouped by variables appearing in Goal but not in the search Term

```
| ?- findall(Child, parent(Parent, Child), Children).
Children = [lisa, bart, maggie, lisa, bart, maggie] ? ;
no

| ?- bagof(Child, parent(Parent, Child), Children).
Parent = homer, Children = [lisa, bart, maggie] ? ;
Parent = marge, Children = [lisa, bart, maggie] ? ;
no
```

bagof

- While *findall* returns an empty list if there are no results, *bagof* fails

```
| ?- findall(Child, parent(bart, Child), L).
L = [] ? ;
no

| ?- bagof(Child, parent(bart, Child), L).
no
```

Existential Quantifier

- We can direct *bagof* to ignore additional variables in *Goal* by using existential quantifiers: *Var^Goal*

```
| ?- bagof(Child, parent(Parent, Child), Children).
Parent = homer, Children = [lisa, bart, maggie] ? ;
Parent = marge, Children = [lisa, bart, maggie] ? ;
no

| ?- bagof(Child, Parent^parent(Parent, Child), Children).
Children = [lisa, bart, maggie, lisa, bart, maggie] ? ;
no
```

  - If all variables appearing in *Goal* but not in the search *Term* are existentially quantified, then *bagof* behaves like *findall*

setof

- ***setof*** has similar behavior to *bagof*, but results are ordered and without repetitions

```
| ?- bagof(Child, parent(Parent, Child), Children).
Parent = homer, Children = [lisa, bart, maggie] ? ;
Parent = marge, Children = [lisa, bart, maggie] ? ;
no

| ?- setof(Child, parent(Parent, Child), Children).
Parent = homer, Children = [bart, lisa, maggie] ? ;
Parent = marge, Children = [bart, lisa, maggie] ? ;
no
```

  - The standard order of terms is used (see section 4.8.8.2)

setof

- Existential quantifiers can also be used with *setof*, with the same effect as with *bagof* (results will remain ordered and without repeats)

```
| ?- bagof(Child, Parent^parent(Parent, Child), Children).
Children = [lisa, bart, maggie, lisa, bart, maggie] ? ;
no

| ?- setof(Child, Parent^parent(Parent, Child), Children).
Children = [bart, lisa, maggie] ? ;
no
```

- If all variables in *Goal* but not in search *Term* are existentially quantified, then *setof* behaves like *findall* followed by *sort*

Collecting Solutions

- Note that the Goal must be able to generate answers

```
| ?- findall(X, (X > 0, X < 6), L).
! Instantiation error in argument 1 of (>)/2
! goal:  _321>0

| ?- findall(X, between(1, 5, X), L).
L = [1,2,3,4,5] ? ;
no
```

Agenda

- Collecting Solutions
- Graphs and Search
  - Puzzles and Games
- Binary Trees

Data Structures

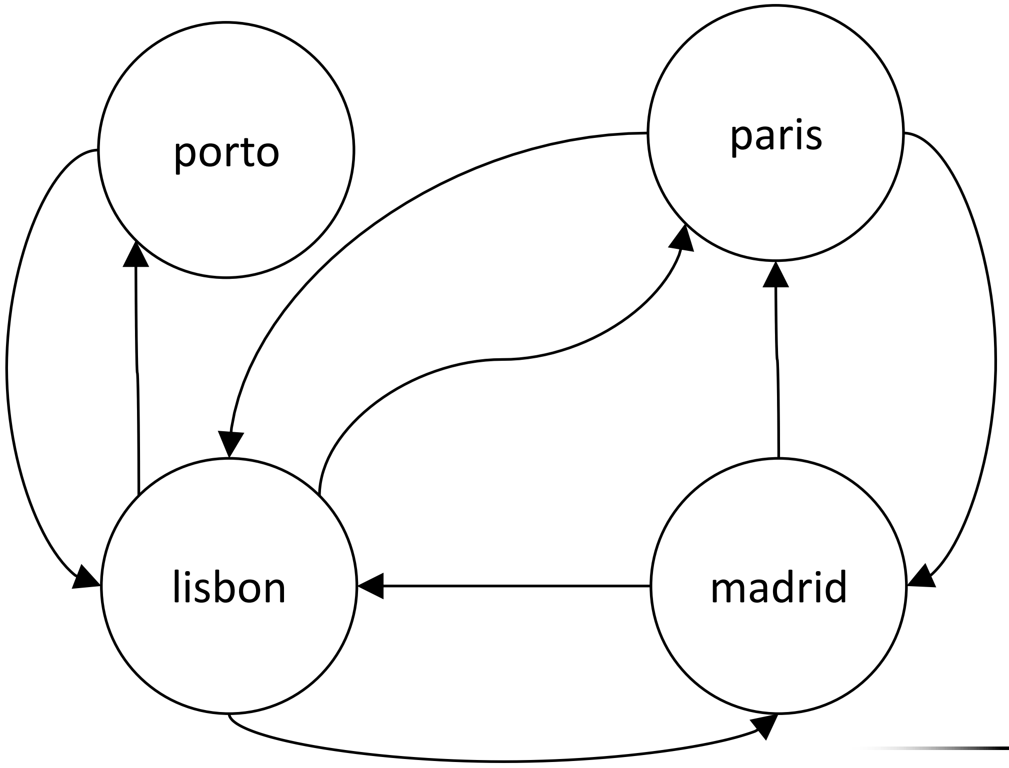
- Even though Prolog doesn't explicitly define types or data structures, terms can be used to do so
  - Unary predicates can be used to 'define a type'
    - The type *male* can be defined as the set of terms *X* such that *male(X)* is true
  - Simple types can be defined recursively
    - Lists, Trees, ...
  - *Pairs* are typically represented as *X-Y*
  - *Tuples* can be represented as *(X, Y, Z)*
    - However, a properly named functor should be used
  - More complex terms can be used to represent data structures

```
book('River God', author(smith, wilbur, 1933), 1993, Book).
```

Graphs and Search

- Graphs can be represented as the connections between nodes
  - set of facts representing [directed] edges

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```



Depth-First Search

- Searching for a possible connection between nodes is made easy by Prolog's standard depth-first search mechanism

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_dfs(S, F):-
      connected(S, F).
connects_dfs(S, F):-
      connected(S, N),
      connects_dfs(N, F).

| ?- connects_dfs(porto, madrid).
yes
| ?- connects_dfs(madrid, porto).
```

When does this approach fail?

Depth-First Search

- Adapted solution with an accumulator to avoid loops

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_dfs(S, F):-
      connects_dfs(S, F, [S]).

connects_dfs(F, F, _Path).
connects_dfs(S, F, T):-
      connected(S, N),
      not( member(N, T) ),
      connects_dfs(N, F, [N|T]).


| ?- connects_dfs(madrid, porto).
yes
```

What would we have to change to *return* the connecting path (route)?

Breadth-First Search

- We can also easily create a BFS solution using *findall*

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_bfs(S, F):-
      connects_bfs([S], F, [S]).

connects_bfs([F|_], F, _V).
connects_bfs([S|R], F, V):-
      findall(N,
      ( connected(S, N),
        not(member(N, V)),
        not(member(N, [S|R]))), L),
      append(R, L, NR),
      connects_bfs(NR, F, [S|V]).
```

What would we have to change to *return* the connecting path (route)?

Games and Puzzles

- Prolog (and search) can easily be used to search for a solution to one-person games or puzzles
- States are represented as the nodes of the graph
  - Initial state is the starting node
  - Winning conditions define the final nodes
- Movements are represented as the transitions between nodes
  - States don’t need to be represented in extension – transitions can specify new states based on the previous one and the move made

Generic Solver

- A generic [abstract] solver to one-person games/puzzles

```
initial(InitialState).

final(State):- winning_condition(State).

move(OldState, NewState):- valid_move(OldState, NewState).

play:- initial(Init),
       play(Init, [Init], States),
       reverse(States, Plays), write(Plays).

play(Curr, Path, Path):- final(Curr), !.
play(Curr, Path, States):- move(Curr, Next),
                           not( member(Next, Path) ),
                           play(Next, [Next|Path], States).
```

Games and Puzzles

- Example: fill a 5-gallon jug with 4 gallons of water, using the 5-gallon jug and a 3-gallon jug

```
initial(0-0).    % Jug5-Jug3

final(4-_).

move(_-S, 5-S). % fill jug 1
move(F-_, F-3). % fill jug 2
move(_-S, 0-S). % empty jug 1
move(F-_, F-0). % empty jug 2
move(F-S, NF-NS):- NF is max(0, F+S-3), NS is min(3, F+S). % 1->2
move(F-S, NF-NS):- NF is min(5, F+S), NS is max(0, F+S-5). % 2->1
```

Shortest Path

- To find the smallest set of plays we just need to find all paths and select the shortest one
  - Easily accomplished using *setof*

Is DFS the best way of doing this?

```
play:-   initial(Init),
         setof( Length-Path, (
                    play(Init, [Init], Path),
                    length(Path, Length) ),
               [_ShortestLength-States|_] ),
         reverse(States, Path), write(Path).
```

What if we wanted the path with the lowest cost?

How could we obtain all paths with shortest length?

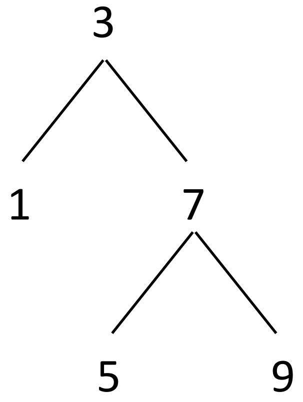
Agenda

- Collecting Solutions
- Graphs and Search
  - Puzzles and Games
- Binary Trees

Binary Trees

- A binary tree can be recursively defined using node elements
    - Empty node represented as `null`
    - Other nodes as `node(Value, Left, Right)`
- Definition of a binary tree

```
binary_tree(null).
binary_tree( node(Value, Left, Right) ):-
      binary_tree(Left),
      binary_tree(Right).
```




```
node(3,  node(1, null, null),
         node(7, node(5,null,null),
                 node(9,null,null)  )  ).
```

Binary Trees

- Tree operations are easily implemented from this definition
  - Check if value is a member of the tree

```
tree_member(Val, node(Val, _L, _R) ).
tree_member(Val, node(V, L, _R) ):-
      [Val < V,] tree_member(Val, L).
tree_member(Val, node(V, _L, R) ):-
      [Val > V,] tree_member(Val, R).
```

[code] if we consider the tree to be a binary search tree

  - List all tree elements (in-order traversal)

```
tree_list( null, [] ).
tree_list( node(Val, L, R), List ):-
      tree_list(L, Left),
      tree_list(R, Right),
      append(Left, [Val|Right], List).
```

Binary Trees

- Verify if tree is ordered

```
tree_is_ordered(Tree):-
      tree_list(Tree, List),
      sort(List, List).
```

- Insert an element into the tree

```
tree_insert( null, V, node(V, null, null) ).
tree_insert( node(V, L, R), V, node(V, L, R) ).
tree_insert( node(V, L, R), Val, node(V, NL, R) ):-
      Val < V, tree_insert( L, Val, NL).
tree_insert( node(V, L, R), Val, node(V, L, NR) ):-
      Val > V, tree_insert( R, Val, NR).
```

Binary Trees

- Determine the height of the tree

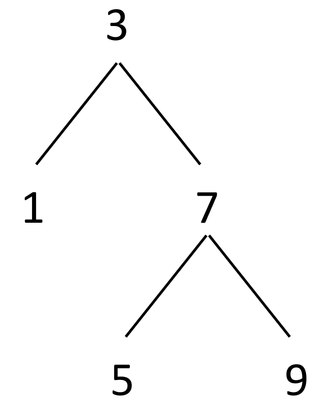
```
tree_height( null, 0).
tree_height( node(Val, L, R), H):-
      tree_height(L, HL),
      tree_height(R, HR),
      H is 1 + max(HL, HR).
```

- Check whether the tree is balanced

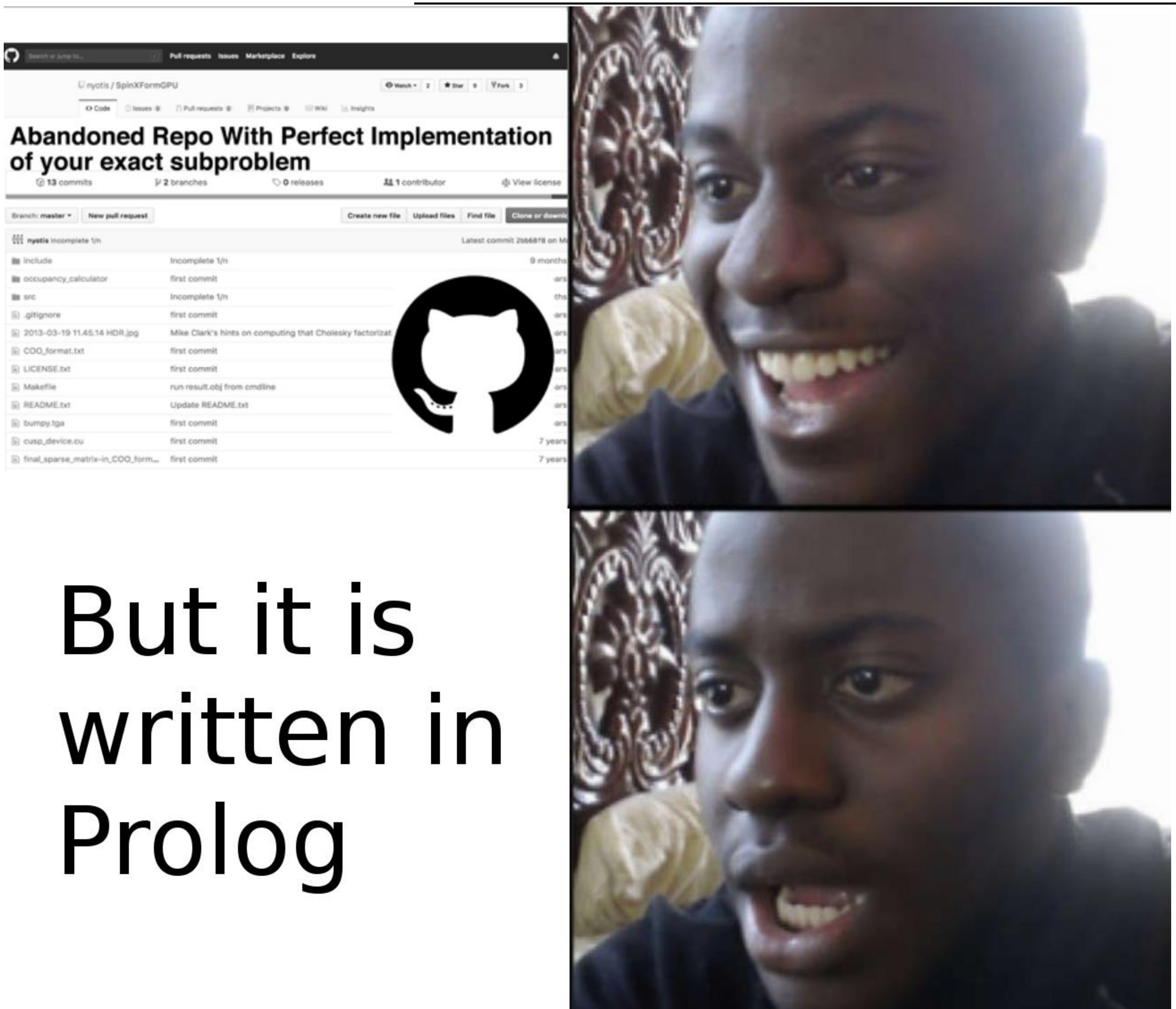
```
tree_is_balanced( null ).
tree_is_balanced( node(Val, L, R) ):-
      tree_is_balanced(L),
      tree_is_balanced(R),
      tree_height(L, HL),
      tree_height(R, HR),
      abs(HL-HR) =< 1.
```

Binary Trees

```
| ?- test_tree(_T), tree_member(5, _T).
yes
| ?- test_tree(_T), tree_member(4, _T).
no
| ?- test_tree(_T), tree_list(_T, L).
L = [1,3,5,7,9] ?
yes
| ?- test_tree(_T), tree_is_ordered(_T).
yes
| ?- test_tree(_T), tree_height(_T, H).
H = 3 ?
yes
| ?- test_tree(_T), tree_is_balanced(_T).
yes
| ?- test_tree(_T), tree_insert(_T, 2, NT).
NT = node(3,node(1,null,node(2,null,null)),node(7,node(5,null,null),node
(9,null,null))) ?
yes
| ?- test_tree(_T), tree_insert(_T, 6, NT), tree_is_balanced(NT).
no
```



Q & A


Abandoned Repo With Perfect Implementation of your exact subproblem
But it is written in Prolog