Logic Programming

Examples

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
kutsia@risc.uni-linz.ac.at

Contents

Sorted Tree Dictionary

Need Associations between items of information.

Dictionary: Associates word with its definition or translation or with facts about it.

Purpose: Retrieval.

Challenge: Efficiency.

Sorted Tree Dictionary

Example

- Task: Make an index of the performance of horses in racing.
- Define: `winnings(X,Y)`, `X` – the name of the horse, `Y` – the number of guineas won.
- Facts:

```
winnings(abaris, 582).
winnings(careful,17).
winnings(jingling_silver,300).
winnings(maloja,356).
```

Data Search

Naive search:

- Linear search top-down.
- Facts at the beginning of the database are retrieved faster than those at the end.
- Might become an issue for big databases.

Data Search

Smarter way:

- Organize data in indices or dictionaries.
- Well-known techniques in computer science.
- Prolog itself uses some of these methods to store its facts and rules. (Will be discussed in next lectures.)
- Nevertheless, sometimes it is helpful to use these methods in our programs.
- In this lecture: A **sorted tree** method for representing a dictionary.

Sorted Trees

Sorted trees:

- Efficient way of using a dictionary.
- A demonstration how the lists of structures are helpful.
- Consist of structures called **nodes**.
- One node for each entry in the dictionary.

Sorted Trees

Nodes in sorted trees:

- Contain four associated items of infromation: key, extra info, two tails.
- Key: The name that determines its place in the dictionary, e.g., horse name.
- Extra info: contains any information about the object involved, e.g., the winnings.
- First tail: Points to a node whose key is alphabetically less than the key in the node itself.
- Second tail: Points to a node whose key is alphabetically greater than the key in the node itself.

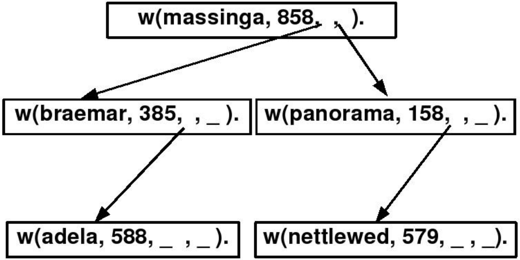
Data Structure

`w(H,W,L,G)` where

- `H`: The name of a horse (an atom), used as a key.
- `W`: the amount of guineas won (an integer).
- `L`: The structure with a horse whose name is less than `H`'s.
- `G`: The structure with a horse whose name is greater than `H`'s.

Data Structure

Structure for a small set of horses, represented as a tree:



Data Structure

Structure for a small set of horses, represented as a PROLOG structure:

```
w(massinga,858,
  w(braemar,385,
    w(adela,588,_,_),
  _),
  w(panorama,158,
    w(nettlewed,579,_,_).
  _)
).
```

Program

"Look up" names of horses in the structure to find out how many guineas they won.

- Structure: `w(H,W,L,G).`
- Boundary condition: The name of the horse we are looking for is `H`.
- Recursive case: Use aless to decide which branch of the tree, `L` or `G`, to look up recursively.
- Using these ideas, define the predicate `lookup(H,S,G)`: Horse `H`, when looked up in index `S` (a `w` structure), won `G` guineas.

Program

```
lookup(H, w(H,G,_,_),G) :- !.

lookup(H, w(H1,_,Before,_), G) :-
    aless(H,H1),
    lookup(H,Before,G).

lookup(H, w(H1,_,_,After), G) :-
     not(aless(H,H1)),
     lookup(H,After,G).
```

Asking Questions

Interesting property:

- If a name of a horse we are looking for is not in the structure, then the information we supply about the horse using `lookup` as a goal will be instantiated in the structure.

Goals

Example

```
?- lookup(ruby_vintage,X,582).
X = w(ruby_vintage,582,_B,_A);

?- lookup(ruby_vintage,X,582),lookup(maloja,X,356).
X = w(ruby_vintage,582, w(maloja,356,_C,_B),_A);

?- lookup(a,X,100),lookup(b,X,200),lookup(z,X,300),
     lookup(m,X,400).
X = w(a,100,_E, w(b,200,_D,
     w(z,300,w(m,400,_C,_B),_A)));
```

Searching Mazes

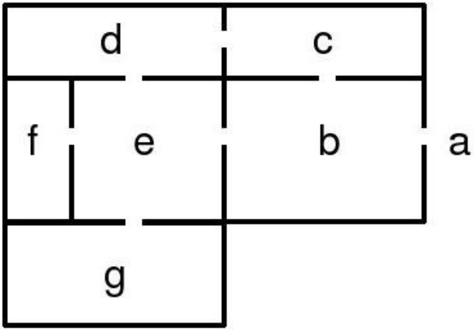
Searching for a telephone in a building:

- How do you search without getting lost?
- How do you know that you have searched the whole building?
- What is the shortest path to the telephone?

Steps

1. Go to the door of any room
2. If the room number in on the list (of already visited) ignore the room and go to step 1.
3. Add the room to the list.
4. Look in the room for a telephone.
5. If there is no telephone, go to step 1. Otherwise, we stop and our list has the path that we took to come to the correct room.

Maze



Idea

When in a room:

- We are in the room we want to be in, or
- We have to pass through a door, and continue (recursively).

We go into the other room if we have not been there yet (not on the list).

`go(X,Y,T)`: Succeeds if one can go from room X to room Y. T contains the list of roomes visited so far.

Program

```
go(X,X,_).

go(X,Y,T) :- door(X,Z),
   write('Go into room'),
   write(Z),nl,
   not(member(Z,T)),
   go(Z,Y,[Z|T]).

go(X,Y,T) :- door(Z,X),
   write('Go into room'),
   write(Z),nl,
   not(member(Z,T)),
   go(Z,Y,[Z|T]).
```

Run

`hasphone(g)`:

- Phone is in the room g.
- Add to the database.

Goals:

- `?- go(a,X,[]),hasphone(X).` Generate-and-test, inefficient.
- `?- hasphone(X),go(a,X,[]).` Better.

Findall

Determine all the terms that satisfy a certain predicate.

`findall(X,Goal,L)`: Succeeds if `L` is the list of all those `X`'s for which `Goal` holds.

Example

```
?- findall(X, member(X,[a,b,a,c]),L).

X = _G166
L = [a,b,a,c] ;
No

?- findall(X, member(X,[a,b,a,c]),[a,b,c]).

No
```

More Examples on Findall

Example

```
?- findall(X, member(5,[a,b,a,c]),L).

X = _G166
L = [] ;
No

?- findall(5, member(X,[a,b,a,c]),L).

X = _G166
L = [5,5,5,5] ;
No
```

More Examples on `Findall`

Example

```
?- findall(5, member(a,[a,b,a,c]),L).

L = [5,5] ;
No

?- findall(5, member(5,[a,b,a,c]),L).

L = [] ;
No
```

Implementation of `Findall`

`findall` is a built-in predicate.

However, one can implement it in PROLOG as well:

```
findall(X,G,_) :-
     asserta(found(mark)),
     call(G),
     asserta(found(X)),
     fail.

findall(_,_,L) :-
     collect_found([],M),
     !,
     L=M.
```

Implementation of `Findall`, Cont.

```
collect_found(S,L) :-
    getnext(X),
    !,
    collect_found([X|S],L).
collect_found(L,L).

getnext(X) :-
    retract(found(X)),
    !,
    X \== mark.
```

Sample Runs

```
?- findall(X,member(X,[a,b,c]), L).

L = [a,b,c] ;
No

?- findall(X, append(X,Y,[a,b,c]), L).

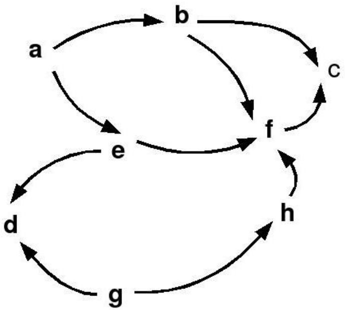
L = [[], [a], [a,b], [a,b,c]] ;
No

?- findall([X,Y], append(X,Y,[a,b,c]), L).

L = [[[],[a,b,c]], [[a],[b,c]], [[a,b],[c]],
[[a,b,c],[]]] ;
No
```

Representing Graphs

```
a(g,h).
a(g,d).
a(e,d).
a(h,f).
a(e,f).
a(a,e).
a(a,b).
a(b,f).
a(b,c).
a(f,c).
```



Moving Through Graph

Simple program for searching the graph:

- ```
  go(X,X).
  go(X,Y) :- a(X,Z),go(Z,Y).
  ```
- Drawback: For cyclic graphs it will loop.
- Solution: Keep trial of nodes visited.
  ```

Improved Program for Graph Searching

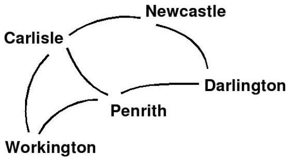
`go(X,Y,T)`: Succeeds if one can go from node `X` to node `Y`. `T` contains the list of nodes visited so far.

```
go(X,X,T).
go(X,Y,T) :- a(X,Z),
     legal(Z,T),
     go(Z,Y,[Z|T]).

legal(X,[]).
legal(X,[H|T]) :- X \= H,
     legal(X,T).
```

Car Routes

```
a(newcastle,carlisle,58).
a(carlisle,penrith,23).
a(darlington,newcastle,40).
a(penrith,darlington,52).
a(workington,carlisle,33).
a(workington,penrith,39).
```



Car Routes Program

```
a(X,Y) :- a(X,Y,_).

go(Start,Dest,Route) :-
     go0(Start,Dest,[],R),
     rev(R,Route).

go0(X,X,T,[X|T]).
go0(Place,Dest,T,Route) :-
     legalnode(Place,T,Next),
     go0(Next,Dest,[Place|T],Route).
```

Car Routes Program, Cont.

```
legalnode(X,Trail,Y) :-
     (a(X,Y) ; a(Y,X)),
     legal(Y,Trail).

legal(X,[]).
legal(X,[H|T]) :- X \= H,
      legal(X,T).

rev(L1,L2) :- revzap(L1,[],L2).

revzap([X|L],L2,L3) :-
     revzap(L,[X|L2],L3)
revzap([],L,L).
```

Runs

```
?- go(darlington,workington,X).

X = [darlington,newcastle,carlisle,
     penrith,workington];

X = [darlington,newcastle,carlisle,
     workington];

X = [darlington,penrith,carlisle,workington];

X = [darlington,penrith,workington];

no
```

Findall Paths

```
go(Start,Dest,Route) :-
    go1([[Start]],Dest,[],R),
    rev(R,Route).

go1([First|Rest],Dest,First) :-
    First = [Dest|_].
go1([[Last|Trail]|Others],Dest,Route] :-
    findall([Z,Last|Trail],
         legalnode(Last,Trail,Z),
         List),
    append(List,Others,NewRoutes),
    go1(NewRoutes,Dest,Route).
```

Depth First

```
?- go(darlington,workington,X).
X = [darlington,newcastle,
     carlisle,penrith,workington];
X = [darlington,newcastle,
     carlisle,workington];
X = [darlington,penrith,
     carlisle,workington];
X = [darlington,penrith,workington];
no
```

Depth, Breadth First

```
go1([[Last|Trail]|Others],Dest,Route]:-
     findall([Z,Last|Trail],
       legalnode(Last,Trail,Z),
       List),
     append(List,Others,NewRoutes),
     go1(NewRoutes,Dest,Route).

go1([[Last|Trail]|Others],Dest,Route]:-
     findall([Z,Last|Trail],
       legalnode(Last,Trail,Z),
       List),
     append(Others,List,NewRoutes),
     go1(NewRoutes,Dest,Route).
```

Breadth First

```
?- go(darlington,workington,X).
X = [darlington,penrith,workington];
X = [darlington,newcastle,
     carlisle,workington];
X = [darlington,penrith,
    carlisle,workington];
X = [darlington,newcastle,
    carlisle,penrith,workington];
no
```