

CSc 372

Comparative Programming Languages

32 : Prolog — Second-Order Predicates

Department of Computer Science  
University of Arizona

[collberg@gmail.com](mailto:collberg@gmail.com)

Copyright © 2013 Christian Collberg

# Second-Order Programming

## Second-Order Predicates

- When we ask a question in Prolog we will (if everything goes right) get an answer. **One** answer. We can if we want to ask Prolog to backtrack (using the semi-colon), but we will still only get one answer at a time.
- Furthermore, when we backtrack all the information gathered previously is lost.
- It isn't possible (in pure Prolog) to find the set of **all possible solutions** to a query.
- However, if we go outside pure Prolog (using the database manipulation features) we can construct procedures which collect all solutions to a query.
- They are called *second-order* because they deal with sets and the properties of sets, rather than about individual elements of sets.

## Second-Order Predicates

- `setof(X,Goal,List)`
  - `List` is a collection of `Xs` for which `Goal` is true.
  - `List` is sorted and contains no duplicates.
- `bagof(X,Goal,List)`
  - `List` is may contain duplicates.
- `setof` and `bagof` will fail if no `Goals` succeed.
- `findall(X,Goal,List)`
  - `findall` will return `[]` if no `Goals` succeed.

# Examples

```
remove_duplicates(X, Y) :-  
    setof(M, member(M,X), Y).
```

```
children(X,Kids) :-  
    setof(C, father(X,C), Kids).
```

# Uninstantiated Variables

- Consider `setof(X,Goal,List)` and `bagof(X,Goal,List)`.
- If there are uninstantiated variables in `Goal` which do not also appear in `X`, then a call to `setof` or `bagof` may backtrack, generating alternative values for `List`.
- If this is not the behavior you want, you can say  
$$Y \sim Goal$$
meaning there exists a `Y` such that `Goal` is true, where `Y` is some Prolog term (usually, a variable).
- `findall` does this automatically.

# Uninstantiated Variables...

- Consider this database:

```
foo(1,a).
```

```
foo(2,b).
```

```
foo(3,c).
```

- If we use both arguments of foo in our goal, we get what we expect:

```
| ?- findall(X/Y, foo(X,Y), L).
```

```
L = [1/a,2/b,3/c]
```

```
| ?- setof(X/Y, foo(X,Y), L).
```

```
L = [1/a,2/b,3/c]
```

```
| ?- bagof(X/Y, foo(X,Y), L).
```

```
L = [1/a,2/b,3/c]
```

## Uninstantiated Variables...

- If we only use one of foo's arguments in our goal, findall still gets us the expected result:

```
| ?- findall(X, foo(X,Y), L).  
L = [1,2,3]
```

- But, bagof doesn't:

```
| ?- bagof(X, foo(X,Y), L).  
L = [1]  
Y = a ? ;  
L = [2]  
Y = b ? ;  
L = [3]  
Y = c  
L = [1,2,3]
```



# Uninstantiated Variables...

- So, instead we have to do:

```
| ?- bagof(X, Y^foo(X,Y), L).  
L = [1,2,3]
```

## SetOf — Drinkers

```
:- op(500, yfx, 'drinks').
```

```
john drinks whiskey.  
martin drinks whiskey.  
david drinks milk.  
ben drinks milk.  
helder drinks beer.  
laurence drinks beer.  
chris drinks coke.  
louise drinks l_and_p.
```

```
?- setof(X, X drinks milk, S).  
   X = _9109,  
   S = [ben,david]
```

## Implementing bagof

```
bagof(Item, Goal, _) :-  
    assert(bag(marker)),  
    Goal,  
    assert(bag(Item)),  
    fail.
```

```
bagof(_, _, Bag) :-  
    retract(bag(Item)),  
    collect(Item, [], Bag).
```

```
collect(marker, L, L).  
collect(Item, ThisBag, FinalBag) :-  
    retract(bag(NextItem)),  
    collect(NextItem,  
        [Item|ThisBag], FinalBag).
```

# Implementing setof

- `setof` is implemented as a call to `bagof` followed by a call to `sort` which puts the elements in order and removes duplicates.

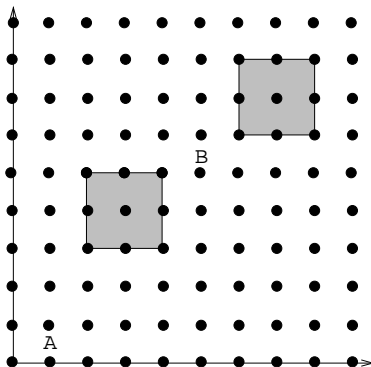
# Lee's Algorithm

# Lee's Algorithm

We are next going to look at a more involved example, an application from VLSI design. It uses the set of predicate to compute a shortest path between two points on a grid, subject to the conditions that

- 1 The path goes in the east-west-north-south direction only.
- 2 The path doesn't touch any obstacles.

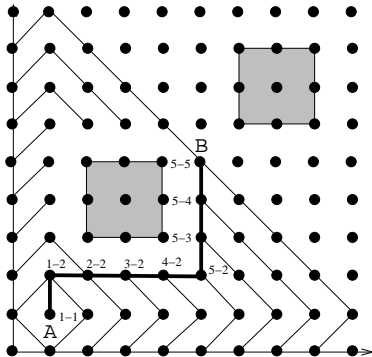
- VLSI routing on a grid.
- Find a shortest Manhattan route between A and B that doesn't pass through any obstacles.



```

lee_route(A,B,Obstacles,Path) :-
    waves(B, [[A], []], Obstacles, Waves),
    path(A,B,Waves,Path).
?- lee_route(1-1,5-5,[obst(2-3, 4-5),
    obst(6-6, 8-8)], P).

```





# Lee's Algorithm. . .

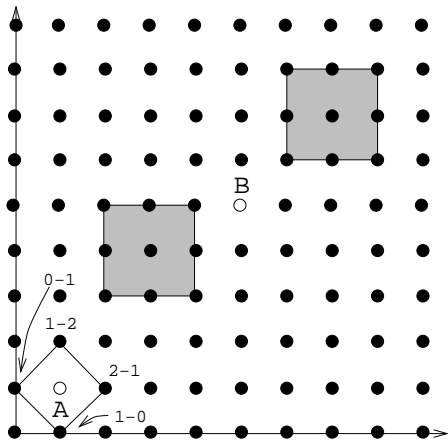
Lee's algorithm works in two stages:

- 1 First we generate a sequence of waves, where the first wave consists of the starting point itself.
- 2 Then we use the set of waves to find a shortest path.

# Lee's Algorithm. . .

- We start out with one wave which consists solely of the source point.
- From that point we generate all neighboring points. This forms the second wave.
- Each wave consists of points which are
  - 1 neighbors to points on the previous wave,
  - 2 not members of previous waves,
  - 3 not obstructed by any obstacles.
- We stop when the destination point is on the last generated wave.

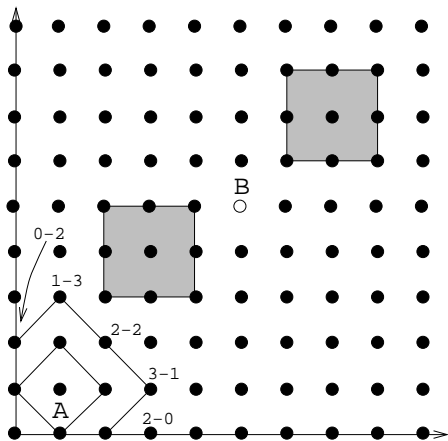
LastW = []  
Wave = [1-1]  
NextW = [0-1, 1-0, 1-2, 2-1]



LastW = [1-1]

Wave = [0-1, 1-0, 1-2, 2-1]

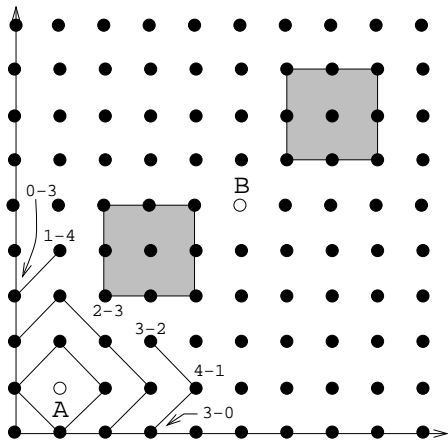
NextW = [0-0, 0-2, 1-3, 2-0, 2-2, 3-1]



LastW = [0-1, 1-0, 1-2, 2-1]

Wave = [0-0, 0-2, 1-3, 2-0, 2-2, 3-1]

NextW = [0-3, 1-4, 3-0, 3-2, 4-1]



## Lee's Algorithm. . .

```
waves(Destination,Wavessofar,Obstacles,Waves) :-  
    Waves is a list of waves including  
    Wavessofar (except, perhaps, it's last wave)  
    that leads to Destination without crossing .  
    Obstacles.
```

```
next_waves(Wave,LastWave,Obstacles,NextWave) :-  
    Nextwave is the set of admissible points  
    from Wave, that is excluding points from  
    Lastwave, Wave, and points under Obstacles.
```

## Lee's Algorithm. . .

- The first *wave-rule* (the recursive base case for *wave*) states that once the last generated wave contains the destination point, we're done generating waves.
- The second *wave-rule* simply generates the next wave (using *next\_wave*), and then adds it to the beginning of the list of waves. Note that the list of waves is a *list-of-lists*.

# Lee's Algorithm. . .

- `next_wave` takes three input parameters:
  - 1 Wave is the last generated wave.
  - 2 `LastWave` is the wave generated before the last wave.
  - 3 `ObstacleIs` is the list of obstacles.
- `next_wave` uses `setof` to generate the set of all *admissible* points. A point is admissible if it belongs to the next wave.



## Lee's Algorithm. . .

```
waves(B, [Wave|Waves], Obstacles, Waves) :-  
    member(B, Wave), !.  
waves(B, [Wave, LastWave|LastWaves],  
    Obstacles, Waves) :-  
    next_wave(Wave, LastWave, Obstacles, NextWave),  
    waves(B, [NextWave, Wave, LastWave|LastWaves],  
    Obstacles, Waves).  
  
next_wave(Wave, LastWave, Obstacles, NextWave) :-  
    setof(X, admissible(X, Wave, LastWave, Obstacles),  
    NextWave).
```

X is **adjacent** to the points on Wave (i.e. X is a point on the next wave) if

- X is a neighbor to a point X1 on the previous wave (Wave, that is).
- X is not obstructed by an obstacle.

Notice that adjacent uses a **generate-and-test** scheme:

- 1 member & neighbor work together to generate new possible points:
  - 1 member generates points on the previous wave.
  - 2 neighbor uses the points generated by member to generate points which are neighbors to the points on the last wave.
- 2 obstructed weeds out generated point that are under an obstacle.

# Lee's Algorithm...

X is an admissible point if

- 1 it is a neighbor of a point on the previous wave
- 2 it is not on any previous wave
- 3 it is not obstructed by an obstacle

```
admissible(X,Wave,LastWave,Obst) :-  
    adjacent(X,Wave,Obst),  
    not member(X,LastWave),  
    not member(X,Wave).
```

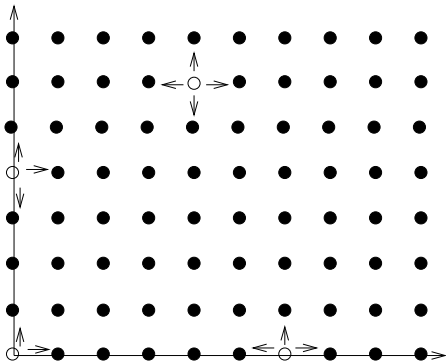
```
adjacent(X,Wave,Obstacles) :-  
    member(X1,Wave),  
    neighbor(X1,X),  
    not obstructed(X,Obstacles).
```

# Lee's Algorithm. . .

- `next_to` takes a number  $A$  and returns  $B=A+1$  and  $B=A-1$ .  
 $A-1$  is returned only if the result is  $>0$ .
- `neighbor` uses `next_to` to generate neighboring points. The rules of `neighbor` state:
  - 1 The point  $X_2-Y$  is a neighbor of point  $X_1-Y$  if  $X_2$  is  $X_1+1$ , or  $X_2=X_1-1$ . In other words, the first neighbor rule generates the points immediately above and below a given point.
  - 2 The point  $X-Y_2$  is a neighbor of point  $X-Y_1$  if  $Y_2$  is  $Y_1+1$ , or  $Y_2=Y_1-1$ . In other words, the second neighbor rule generates the points immediately to the left and right of a given point.

```
neighbor(X1-Y,X2-Y):- next_to(X1,X2).  
neighbor(X-Y1,X-Y2):- next_to(Y1,Y2).
```

```
next_to(A,B) :- B is A+1.  
next_to(A,B) :- A > 0, B is A-1.
```



## Lee's Algorithm. . .

- `obstructed(Point, Obstacles)` checks to see if the point is on the perimeter of any of the obstacles in the list of obstacles `Obstacles`.
- The rule `obstructs(Point, Obstacle)` checks to see if the point is on the perimeter of the obstacle.

Note that `obstructed` is another generate-and-test procedure. `member` generates one obstacle at a time from this list, and `obstructs` checks to see if that obstacle obstructs the point.

## Lee's Algorithm. . .

- `obstructed(Point, Obstacles)` checks to see if the point is on the perimeter of any of the obstacles in the list of obstacles `Obstacles`.
- The rule `obstructs(Point, Obstacle)` checks to see if the point is on the perimeter of the obstacle.

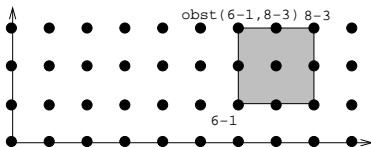
Note that `obstructed` is another generate-and-test procedure. `member` generates one obstacle at a time from this list, and `obstructs` checks to see if that obstacle obstructs the point.



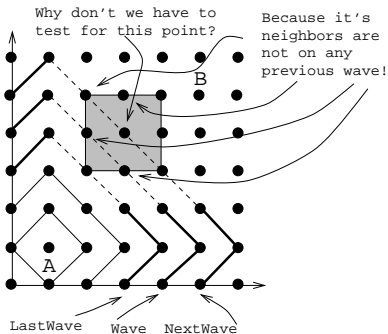
```

% Generate an obstacle, then test
% if it obstructs a point Pt.
obstructed(Pt,Obsts) :-
    member(Obst,Obsts), obstructs(Pt,Obst).
obstructs(X-Y,obst(X-Y1,X2-Y2)) :-
    Y1=<Y, Y=<Y2. % X-Y on bottom edge.
obstructs(X-Y,obst(X1-Y1,X-Y2)) :- Y1=<Y, Y=<Y2.
obstructs(X-Y,obst(X1-Y,X2-Y2)) :- X1=<X, X=<X2.
obstructs(X-Y,obst(X1-Y1,X2-Y)) :- X1=<X, X=<X2.

```



- Why do we only need to check the perimeter? Shouldn't we have to check if a point lies *inside* an object as well?
- No, such points will never be considered. Their neighbors (which are on a perimeter) cannot be on a previous wave:



## Lee's Algorithm. . .

The last part of the algorithm is to construct the actual path from the list of waves. The procedure `path` does this for us.

- 1 `path` starts by looking in the last wave for a neighbor of the destination node. In our example, the destination node is 5-5, and a neighbor of 5-5 in the last wave is the node 5-4.
- 2 `path` next looks for a neighbor for the new node in the next wave. Our example yields node 5-3 which is a neighbor of node 5-4.
- 3 Eventually we'll get to the last wave which only contains the source node, in our case node 1-1.

# Lee's Algorithm...

```
Waves = [[0-7,1-8,2-7,3-6,5-4],6-3,7-0,7-2,8-1],  
         [0-6,1-7,2-6,5-3],6-0,6-2,7-1],  
         [0-5,1-6,5-0,5-2],6-1],  
         [0-4,1-5,4-0,4-2],5-1],  
         [0-3,1-4,3-0,3-2],4-1],  
         [0-0,0-2,1-3,2-0,2-2],3-1],  
         [0-1,1-0,1-2],2-1],  
         [1-1]]
```

```
path(A,A,Waves,[A]) :- !.  
path(A,B,[Wave|Waves],[B|Path]) :-  
    member(B1,Wave),  
    neighbor(B,B1), !,  
    path(A,B1,Waves,Path).
```

# Readings and References

- Read `Clocksin & Mellish`, pp. 156--158.

homework

## Exercise

Write Prolog predicates that given a database of countries and cities

```
% country(name, population, capital).  
country(sweden, 8823, stockholm).  
country(usa, 221000, washington).  
country(france, 56000, paris).  
% city(name, in_country, population).  
city(lund, sweden, 88).  
city(paris, usa, 1). % Paris, Texas.
```

# Exercise. . .

answer the following queries:

- ① Which countries have cities with the same name as capitals of other countries?
- ② In how many countries do more than  $\frac{1}{3}$  of the population live in the capital?
- ③ Which capitals have a population more than 3 times larger than that of the secondmost populous city?