University of Auckland

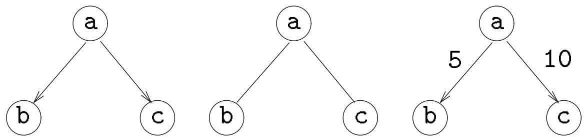
07.363

Functional and Logic Programming

Christian Collberg
October 7, 1994

Graphs

Graph Representation



Directed Unlabeled | Undirected Unlabeled | Directed Labeled

Unlabeled Graphs:

1. `arc(a,b). arc(a,c).`
2. `graph([a,b,c], [arc(a,b),arc(a,c)])`
3. `graph([a-->[b, c], b-->[], c-->[]])`

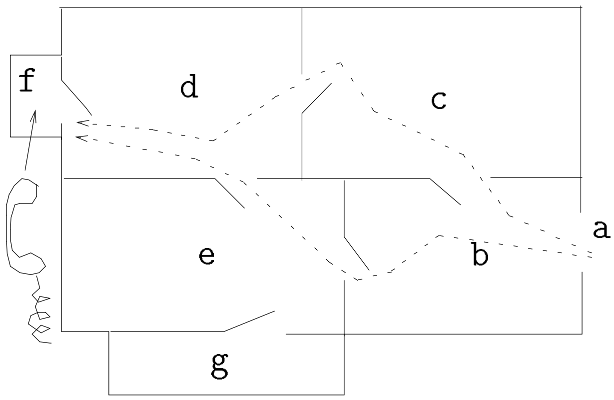
Labeled Graphs:

1. `arc(a,b,5). arc(a,c,10).`
2. `graph([a,b,c], [arc(a,b,10), arc(a,c,10)])`
3. `graph([a-->[b/5, c/10], b-->[], c-->[]])`

Slide 13–1

Depth-First Search I

```
?- find_phone(a, X, [], P).
   X = f; P = [a,b,e,d,f]
?- find_phone(a, _, [e], P).
   X = f; P = [a,b,c,d,f]
?- find_phone(a, _, [c,e], P).
   no
```



Slide 13–2

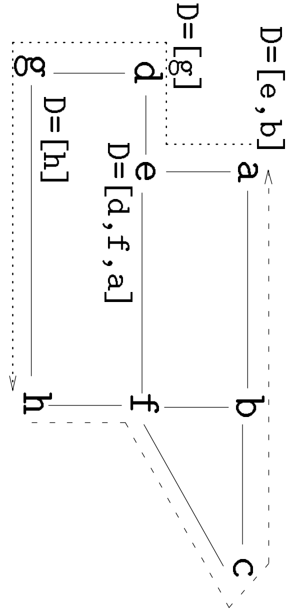
Depth-First Search II

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

% Find a path P from X to Y
% avoiding rooms in T.
paths(X, X, T, [X]).
paths(X, Y, T, [X|U]) :-
   (door(X,Z); door(Z,X)),
   not(member(Z,T)),
   paths(Z,Y,[Z|T], U).

find_phone(From,To,Avoid,Path) :-
   paths(From,To,Avoid,Path),
   has_phone(To).
```

Slide 13–3

Neighbors-First Search I

```
?- go(a,h,[],P).
   P = [a,e,d,g,h]

?- go(h,a,[],P).
   P = [h,f,c,b,a]
```



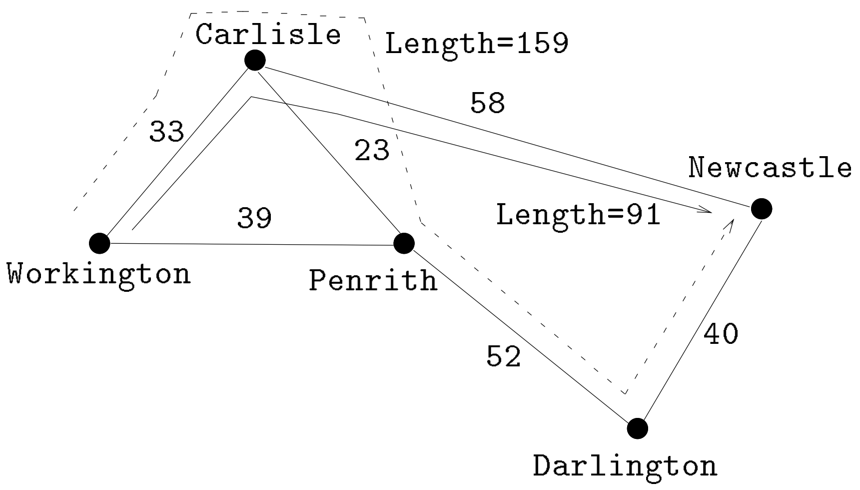
Slide 13–4

Neighbors-First Search II

```
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).

goodnode(X, Y, T) :-
   (a(X,Y); a(Y,X)),
   not(member(Y,T)).

go(X,X,_,[X]).
go(X,Y,T,[X|U]) :-
   bagof(Z, goodnode(X,Z,T), D),
   go_all(D,Y,T,U).

go_all([H|_],Y,T,U) :-
   go(H, Y, [H|T], U).
go_all([_|L],Y,T,U) :-
   go_all(L, Y, T, U).
```

Slide 13–5

Nearest-Neighbor Search I



Slide 13–6

Nearest-Neighbor Search II

```
?- path(workington,newcastle,[],P).
   P=[workington,carlisle,penrith,darlington,newcastle] ;
   P=[workington,carlisle,newcastle] ;
   P=[workington,penrith,carlisle,newcastle] ;
   P=[workington,penrith,darlington,newcastle]

?- path(carlisle,darlington,[],P).
   P=[carlisle,penrith,darlington] ;
   P=[carlisle,workington,penrith,darlington] ;
   P=[carlisle,newcastle,darlington]
```

Slide 13–7

Nearest-Neighbor Search III

- `setof` generates a list of `dist(distance,town)`.
- `setof` sorts the generated list *by distance.*

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

goodnode(X, Y, D, T) :-
   (a(X,Y,D); a(Y,X,D)),
   not(member(Y,T)).
```

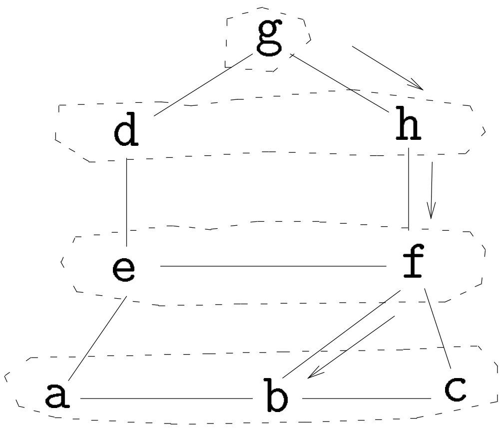
Slide 13–8

Nearest-Neighbor Search IV

```
go(X,X,_,[X]).
go(X,Y,T,[X|U]) :-
   setof(dist(Dist,Z),
      goodnode(X,Z,Dist,T), D),
   go_all(D,Y,T,U).

go_all([dist(_,H)|_],Y,T,U) :-
   go(H, Y, [H|T], U).
go_all([_|L],Y,T,U) :-
   go_all(L, Y, T, U).

path(X,Y,Avoid,Path,Dist) :-
   go(X, Y, [X|Avoid], Path).
```

Slide 13–9

Breadth-First Search I




```
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).

?- go(g,b,P).
   P = [g,h,f,b] ;
   no
```

Slide 13–10

BFS (First Try) II

- Just find out if there's a path.
- `addon` merges two lists, removing duplicates.
- `kids(L,[],M)` returns all children of nodes in `L`.

```
?- go(g,b).
   yes
?- addon([a,b,c],[x,b,y],P).
   P = [c,a,x,b,y]
?- kids([g],[],L).
   L = [d,h]
?- kids([d,h],[],L).
   L = [f,e,g]
```

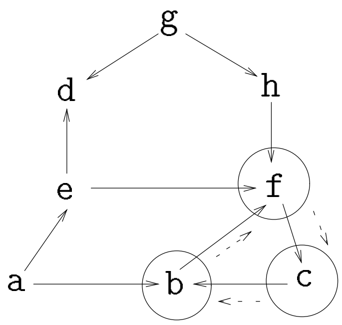
Slide 13–11

BFS (First Try) III

- Doesn't check for cycles.

```
?- go(g,k).
```
loops forever

- Doesn't handle directed cyclic graphs.




```
?- go(f,a).
```
loops forever

Slide 13–12

BFS (First Try) IV

```
go(X,Y) :- path([X],Y).

path(L,D) :- member(D,L), !.
path(L,D) :-
   kids(L,[],M), path(M,D).

kids([],L,L) :- !.
kids([N|Ns], Sf, L) :-
   bagof(D, (a(N,D); a(D,N)), Ds),
   addon(Ds,Sf,Sfl),
   kids(Ns, Sfl, L).

addon([],L,L).
addon([N|Ns],L,M) :-
   member(N,L), !, addon(Ns,L,M).
addon([N|Ns],L,M) :-
   addon(Ns,[N|L],M).
```

Slide 13–13

BFS (Second Try) V

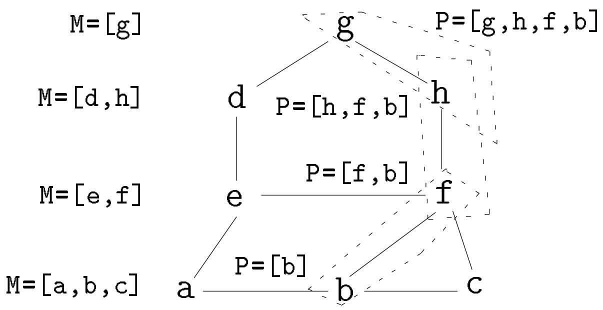
- Just find out if there's a path.
- Check for cycles.
- T stores nodes visited so far.

```
go(X,Y) :- path([X],Y,[X]).

path(L,D,T) :- member(D,L), !.
path(L,D,T) :-
   kids(L,[],M,T),
   append(T,M,T1),
   path(M,D,T1).

kids([],L,L,T) :- !.
kids([N|Ns], Sf, L, T) :-
   bagof(D, goodnode(N,D,T), Ds),
   addon(Ds,Sf,Sfl),
   kids(Ns, Sfl, L, T).
```

Slide 13–14

BFS (Third Try) VI

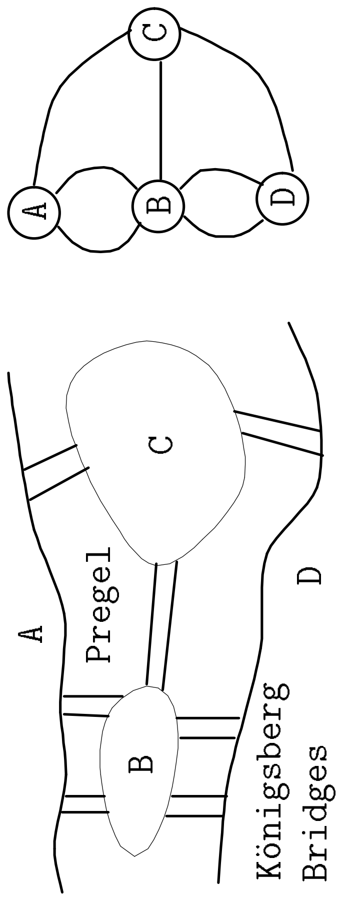
- Retrieve the path.
- path finds two nodes to connect consecutive levels.




```
go(X,Y,P) :- path([X],Y,[X],P).
path(L,D,T,[D]) :- member(D,L), !.
path(L,D,T,[A,B|P]) :-
   kids(L,[],M,T), append(T,M,T1),
   path(M,D,T1,[B|P]),
   member(A,L), (a(A,B); a(B,A)).
```

Slide 13–15

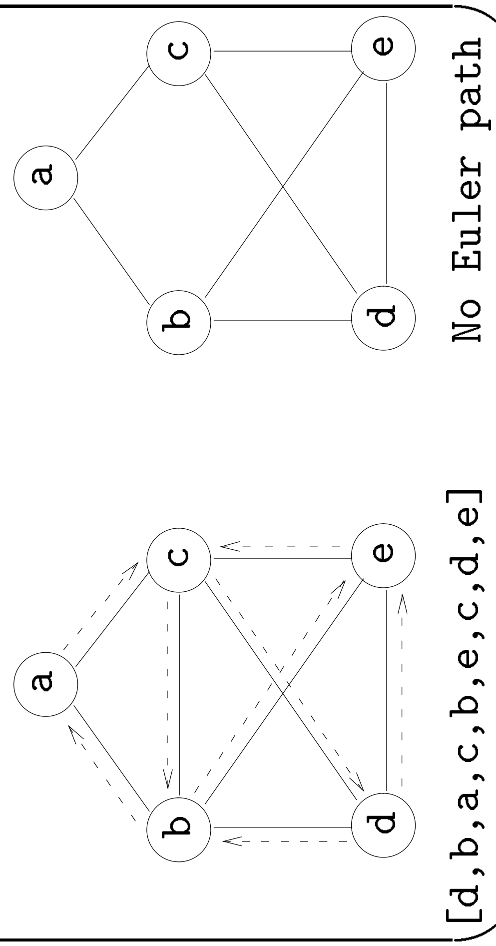
Euler Path I



- An **Euler Path** is a path passing through all vertices, in which every edge occurs exactly once.

Slide 13–16

Euler Path II



Slide 13–17

Euler Path III

- An **Euler Graph** is a graph that contains an Euler path.

G is an Euler graph

1. if G has only one edge.
2. if we remove an edge $E = (a, b)$ from G and are left with an Euler graph with a tour starting in a or b.

Alternatively:

1. G is an Euler graph iff it is connected and has none or two vertices of odd degree.

Slide 13–18

Euler Path IV

```
path([X-Y],[X,Y]).
path([Y-X],[X,Y]).
path(G, [X,Y|R]) :-
   takeout(G, X-Y, G1), path(G1, [Y|R]).
path(G, [X,Y|R]) :-
   takeout(G, Y-X, G1), path(G1, [Y|R]).

takeout([X-Y|R], X-Y, R).
takeout([E|Es], M, [E|R]) :-
   takeout(Es, M, R).

graph(1,[a-b,a-c,b-c,b-d,b-e,c-d,c-e,d-e]).
graph(2,[a-b,a-c,b-d,b-e,c-d,c-e,d-e]).

?- graph(1,G), path(G,P).
   P=[d,e,b,c,d,b,a,c,e]
?- graph(2,G), path(G,P).
   no
```

Slide 13–19