Fortgeschrittene Programmierung
PROLOG Teil 6

Andreas Karwath
(original slides by Peter Flach)

Logik für Informatiker: PROLOG
Part 7: Search

Andreas Karwath

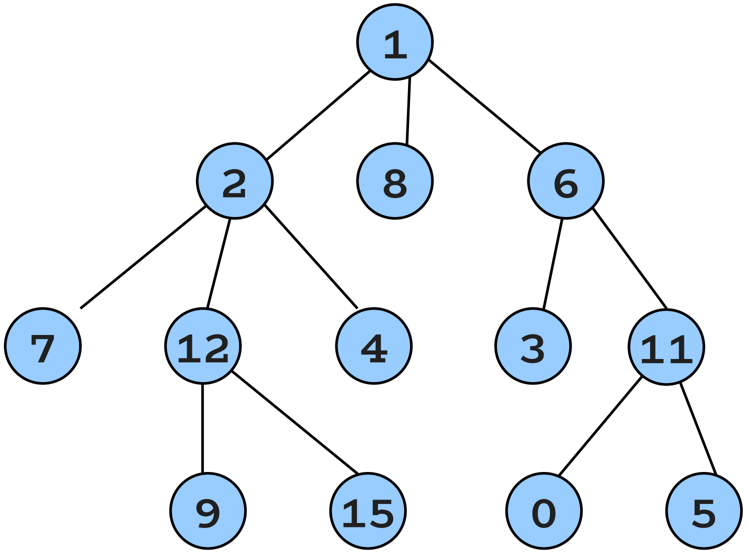
&

Wolfram Burgard

(original slides by Peter Flach)

Search

```
arc(1,2).
arc(1,8).
arc(1,6).
arc(2,7).
arc(2,12).
arc(2,4).
arc(12,9).
arc(12,15).
arc(6,3).
arc(6,11).
arc(11,0).
arc(11,5).
```



An example graph (e.g. a tree)

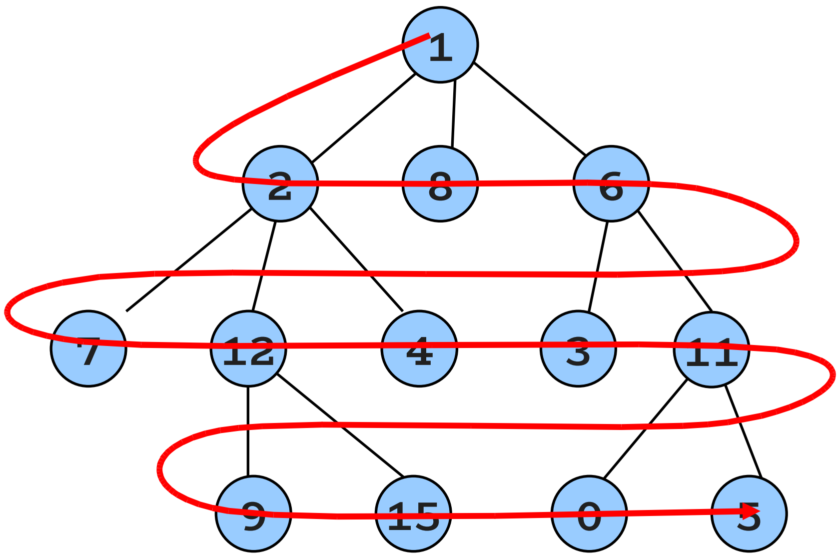
```
% search(Agenda,Goal) <- Goal is a goal node, and
  a
%                        descendant of one of the
%                        nodes on the Agenda
search(Agenda,Goal):-
  next(Agenda,Goal,Rest),
  goal(Goal).
search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

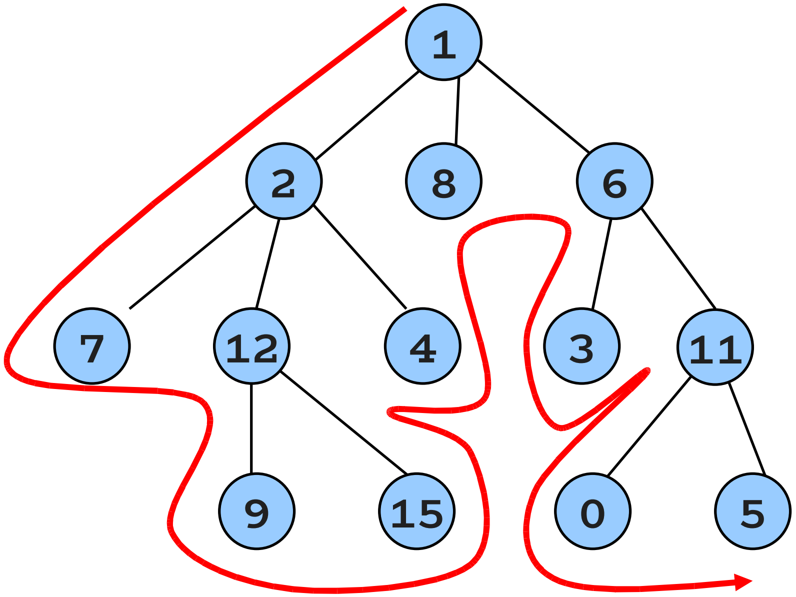
Agenda-based search

```
search_df([Goal|Rest],Goal):-
  goal(Goal).
search_df([Current|Rest],Goal):-
  children(Current,Children),
  append(Children,Rest,NewAgenda),
  search_df(NewAgenda,Goal).

search_bf([Goal|Rest],Goal):-
  goal(Goal).
search_bf([Current|Rest],Goal):-
  children(Current,Children),
  append(Rest,Children,NewAgenda),
  search_bf(NewAgenda,Goal).

children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

Depth-first vs. breadth-first search

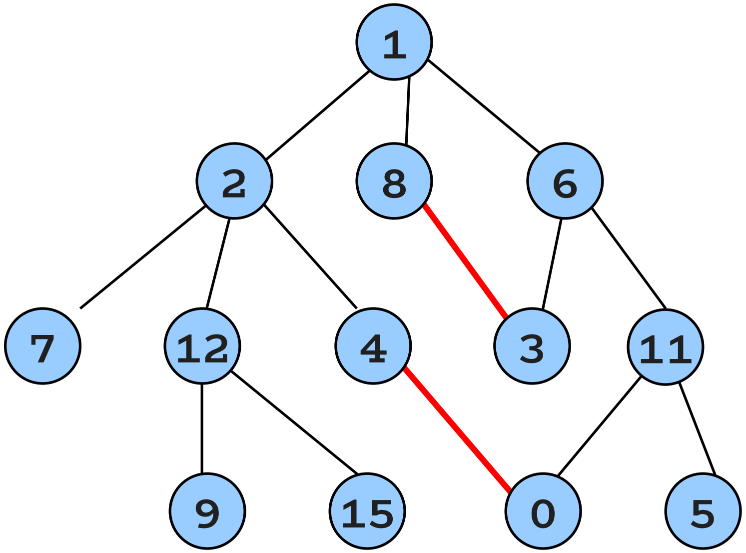
BFS vs DFS


BFS


DFS

Depth-first vs. breadth-first search

- Breadth-first search
    - agenda = queue (first-in first-out)
    - complete: guaranteed to find all solutions
    - first solution founds along shortest path
    - requires $O(B^n)$ memory
- Depth-first search
    - agenda = stack (last-in first-out)
    - incomplete: may get trapped in infinite branch
    - no shortest-path property
    - requires $O(B \times n)$ memory



Loop detection I

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal):-
   goal(Goal).
search_df_loop([Current|Rest],Visited,Goal):-
   children(Current,Children),
   add_df(Children,Rest,Visited,NewAgenda),
   search_df_loop(NewAgenda,[Current|Visited],Goal).

add_df([],Agenda,Visited,Agenda).
add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]):-
   not(member(Child,OldAgenda)),
   not(member(Child,Visited)),
   add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
   member(Child,OldAgenda),
   add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
   member(Child,Visited),
   add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Loop detection II

```
% depth-first search by means of backtracking
search_bt(Goal,Goal):-
  goal(Goal).
search_bt(Current,Goal):-
  arc(Current,Child),
  search_bt(Child,Goal).


% backtracking depth-first search with depth

% bound
search_d(D,Goal,Goal):-
  goal(Goal).
search_d(D,Current,Goal):-
  D>0, D1 is D-1,
  arc(Current,Child),
  search_d(D1,Child,Goal).
```

Backtracking search

```
search_id(First,Goal):-
   search_id(1,First,Goal).     % start with
                                % depth 1

search_id(D,Current,Goal):-
   search_d(D,Current,Goal).
search_id(D,Current,Goal):-
   D1 is D+1,                   % increase depth
   search_id(D1,Current,Goal).
```

- combines advantages of breadth-first search (complete, shortest path) with those of depth-first search (memory-efficient)

Iterative deepening

- *Problem representation*: the vertices of the graph represent the states of the problem, the edges represent the transitions, leading from one state to the next.

- *Problem solving*: finding a path from the initial state to the final state by application of a sequence of transition rules

- *Goal:*
  general framework for the solution of such problems using depth-first search. Formulated generally, such that arbitrary problems within the framework can be solved

Application: search in a state space

```
% solve_dfs(State,Visited,Transitions) :-
% Transitions is the sequence of transitions
% to reach a desired final state from the
% current State. Visited contains the states
% visited previously.

solve_dfs(State,Visited,[]) :-
  final_state(State).
 solve_dfs(State,Visited,[Transition|Transitions]) :-

  transition(State,Transition),
  update(State,Transition,State1),
  legal(State1),
  not(member(State1,Visited)),
  solve_dfs(State1,[State1|Visited],Transitions).

 % Testing the framework:
 test_dfs(Problem,Transitions) :-
  initial_state(Problem,State),
  solve_dfs(State,[State],Transitions).
```

Application: search in a state space

`transition(State,Transition)`: binary predicate specifying the states; `Transition` is applicable to `State`

`update(State,Transition,State1)`: applying `Transition` to `State` leads to `State1`

`legal(State)`: `State` is a legal state

`not member(State,Visited)`: cycles can be detected; all visited states are stored in `Visited`

`solve_dfs(State,Visited,Transitions`): incrementally the sequence of transitions, from the initial to the final state, is built

Application: search in a state space

Representing a problem:

- The states have to represented
- For the predicates `transition`, `update`, `legal`, etc. axioms have to be found.

Application: search in a state space

Example: the water jugs problem

There are two water jugs of capacity 8 and 5 liters with no markings, and the problem is to measure out exactly 4 liters from a vat containing 20 liters (or some other larger number). The possible operations are filling up a jug from the vat, emptying a jug into the vat, and transferring the contents of one jug to another until either the pouring jug is emptied completely, or the other jug is filled to capacity.

Required facts:

`capacity(C,JC)`, for `C` equal j1 or j2;
`jugs(C1,C2)`, where `C1` and `C2` give the current contents of the jugs

the initial state: `jugs(0,0)`

the final states: `jugs(4,0)` or `jugs(0,4)`

Six kinds of `transition`s:
Filling up and emptying a jug, and transferring the contents of one jug to the other. E.g.,

```
transition(jugs(C1,C2),fill(j1)))
```

Transitions for emptying can be optimized (emptying already empty jugs is useless...)

`update` predicate:
- for emptying and filling: easy
- for transferring:enough capacity?

Checking for `legal` states: trivial

Application: search in a state space

```
initial_state(jugs,jugs(0,0)).

final_state(jugs(4,C2)).
final_state(jugs(C1,4)).

transition(jugs(C1,C2),fill(j1)).
transition(jugs(C1,C2),fill(j2)).
transition(jugs(C1,C2),empty(j1)) :-
        C1 > 0.
transition(jugs(C1,C2),empty(j2)) :-
        C2 > 0.
transition(jugs(C1,C2),transfer(j2,j1)).
transition(jugs(C1,C2),transfer(j1,j2)).
```

Application: search in a state space

```
update(jugs(C1,C2),empty(j1),jugs(0,C2)).
update(jugs(C1,C2),empty(j2),jugs(C1,0)).
update(jugs(C1,C2),fill(j1),jugs(Capacity,C2)) :-
        capacity(j1,Capacity).
update(jugs(C1,C2),fill(j2),jugs(C1,Capacity)) :-
        capacity(j2,Capacity).
update(jugs(C1,C2),transfer(j2,j1),jugs(W1,W2)) :-
        capacity(j1,Capacity),
        Water is C1 + C2,
        Overhang is Water - Capacity ,
        adapt(Water,Overhang,W1,W2).
update(jugs(C1,C2),transfer(j1,j2),jugs(W1,W2)) :-
        capacity(j2,Capacity),
        Water is C1 + C2,
        Overhang is Water - Capacity ,
        adapt(Water,Overhang,W2,W1).
```

Application: search in a state space

```
adapt(Water,Overhang,Water,0) :- Overhang =< 0.
adapt(Water,Overhang,C,Overhang) :-
        Overhang > 0,
        C is Water - Overhang.

legal(jugs(C1,C2)).

capacity(j1,8).
capacity(j2,5).
```

Application: search in a state space

```
 ?- test_dfs(jugs,T).

T =
   [fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),trans
   fer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2)] ? ;

T =
   [fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),trans
   fer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2)] ? ;

T =
   [fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),trans
   fer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2)] ? ;

T =
   [fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),trans
   fer(j1,j2),empty(j2),transfer(j1,j2),fill(j1),transfer(j1,j2),empty(j2),transfer(j1,j2),fill(
   j1)] ? ;

no
```