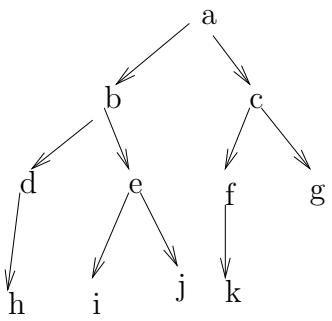

Figure 1: An ordered tree

CAP 5602
Summer, 2011

Lesson 7 Depth Search

The topics

1. depth search
2. the bounded depth search
3. the peasant, wolf, goat and cabbage problem

1. Depth Search

The depth search and the breadth search are used to traverse directed graphs. We assume that the children of all nodes are ordered, 0, 1, 2, 3, and so on. The algorithm for traversing a graph is as follows:

1. if the node is empty, return.
2. process the start node.
3. traverse subtree 0 using depth search
4. traverse subtrees 1, 2, ... using depth search.

Example Let's see how depth search lists the nodes of the tree in Figure 1 when the start node is a.

First, it processes a. Then it traverses the subtree with root b. It processes b, and then traverses the first subtree of b. So, it processes d and then traverses the leftmost (and the only) subtree of d. It processes h and since h has no subtrees, it returns to traverse the remaining subtrees of d. The node

d has no more subtrees, so depth search returns to traverse the next subtree of b. It processes e and then traverses the first subtree of e, and so on.

The final list is a, b, d, h, e, i, j, c, f, k, g.

This traversal is known in the computer literature as **preoder**.

Breadth search lists the nodes by how far they are from the start. It first processes the start node, then the nodes that are one arrow removed from start, beginning with the first child up to the last. Then, breadth search lists the nodes 2 arrows removed from the start. The children of the first child are listed first, then the children of the second, and so on. After that, breadth search lists the nodes 3 arrows removed and continues until all nodes are processed.

The final list is a, b, c, d, e, f, g, h, i, j, k.

This methods work fine for finite trees. But what do we do when we deal with graphs?

In this lecture we study the depth search. So, let us assume that we have an ordered graph, i.e. the children of each node are ordered. We have a start node and there some **solution** nodes identified by the predicate goal_reached(Node). We use the predicate successor(Node1,Node2) to indicate that Node2 is a child of Node1. The order on the set of children is accomplished by the order in which the successor facts are listed, or by the order in which they are generated by the start node, as will be seen in the peasant problem.

We need to find a path from the start node to a solution node. We saw from lesson 3, that the predicates path(Node1,Node2), that checks if there is path from Node1 to Node2, does not work for graphs. So, we need to keep track not only of the start node and the end node of the path, but also of what nodes are on the path. This way, when we want to add a new node to the path P, we first check if it is not already on the path. For this reason, depth_search uses the helper predicate depth_search2 that remembers the path from the start node to the current node.

```
% depth_search(Node,Sol) return a path,
% in the reverse order, from Node to a goal node
% Sol is a solution path
depth_search(Node, Sol) :- depth_search2(Node, [Node], Sol).

% depth_search2(Node, Path, Sol) is satisfied if Sol is
% a path from the start node to a goal, and Path is
```

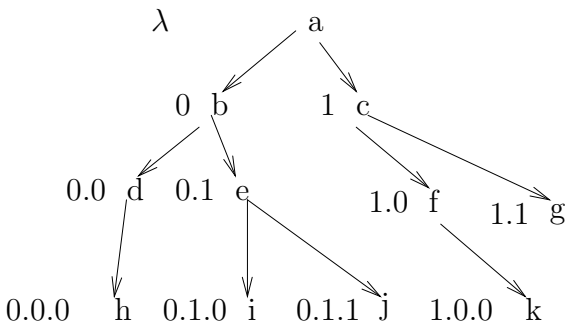

Figure 2: An ordered tree

```
% a path from start to Node.
depth_search2(Node, Sol, Sol):- goal_reached(Node).
depth_search2(Node, Path, Sol) :- successor(Node,Child),
    not(member(Child,Path)),     % Child is not on the path
    depth_search2(Child, [Child|Path],Sol).
    % search if there is a path from Child to a goal
```

If the graph is finite and there is a solution, depth_search will find it.

2. the bounded depth search

This method runs may not terminate when the graph is infinite.

Let us see why. First, we attach to each node of the graph its Dewey address. The Dewey address is a sequence of whole numbers defined as follows:

1. the address of the start node is the empty sequence, denoted by ϵ.

2. the addresses of the children 0, 1, ..., n of the node with address A are $A.0, A.1, \ldots A.n$

Figure 2 shows the Dewey addresses of the tree in Figure 1. The addresses are to the left of the node.

We ordeBr these addresses in the alphabetical order, i.e. $A < B$ if at the leftmost position where A and B differ the number of A is less than the number of B. So, $1.0.2.3 < 1.1$ because the first index where they differ is 2 and at that position the first address has 0 while the second one has 1.

It is easy to check that depth_search processes the nodes in the alphabetical order of the Dewey addresses of the graph.

Now let us assume that the tree has an infinite path λ, 0, 0.0, 0.0.0,

0.0.0.0, ..., 0.0.0...0 (n 0's), etc. Moreover, let 1 be the only goal node. Then depth_search will never get to address 1 because it will get stuck in the infinite path.

We can avoid this by puting a limit on the length of path.

% bounded_depth_search(Node,Sol,Depth) return a path,
% in the reverse order, from Node to a goal node
% Sol is the solution path, Depth is the maximum depth
bounded_depth_search(Node, Sol, Depth) :- bounded_depth_search2(Node, [Node], Sol,Depth).

% bounded_depth_search2(Node, Path, Sol, Depth) is satisfied if Sol is
% a path from the start node to a goal of length ¡= Depth, and Path is
% a path from start to Node.
bounded_depth_search2(Node, Sol, Sol, _):- goal_reached(Node).
bounded_depth_search2(Node, Path, Sol, Depth) :- Depth > 0, New_Depth is Depth - 1,
    successor(Node,Child), not(member(Child,Path)), % Child is not on the path
    bounded_depth_search2(Child, [Child|Path],Sol, New_Depth).
    % search if there is a path from Child to a goal

We see that every time a recursive call to bounded_depth_search2 is made, the depth decreases by 1.

As an example let us run this program on the graph from Figure 3 whose prolog program is shown below. In the figure we circled the goal state.

successor(a,b).
successor(a,c).
successor(a,d).

successor(b,c).
successor(b,e).
successor(b,f).
successor(b,g).

successor(e,a).
successor(f,f).

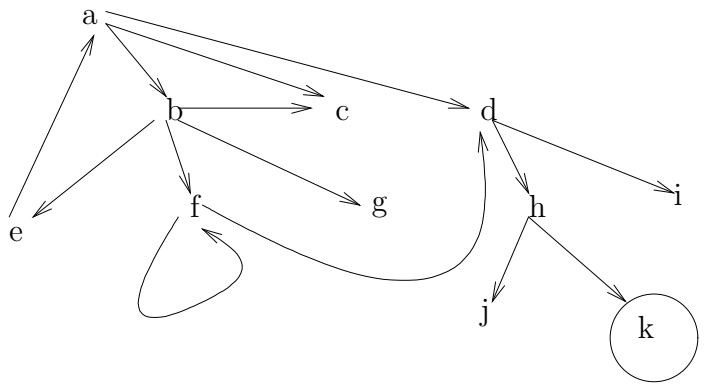

Figure 3: The graph for the bounded_depth_search

successor(f,d).

successor(d,a).
successor(d,h).
successor(d,i).

successor(h,j).
successor(h,k).

goal_reached(k).
The answer to the query ?-bounded_depth_search(a,Sol,3) is
Sol = [k, h, d, a] .
The answer to the query ?- bounded_depth_search(a,Sol,2) is
false
because the shortest path to the goal has length 3.

3. The peasant, wolf, goat and cabbage problem

This is a classic problem, attributed to Alcuin, a monk, poet, and scholar who lived from approximately 735 to 804 AD. Let us state the problem.

A peasant is on the left bank of a river with his possesions, a wolf, a goat, and a large cabbage. The peasant also has a boat that is large enough to hold him and one of his possessions. He wants to carry all his possessions safely across the river. If he is not present, the wolf will eat the goat and the goat will eat the cabbage. It is assumed that when anuy of the possessions is left alone on either bank, it will wait patiently for the return of the peasant.

A solution to this problem is a sequence of river crossing that will end up with all 4 of them safely on the right bank.

First, we must represent the states and the moves. The state is determined by the positions of the peasant, wolf, goat an cabbage. We need not worry about the boat, because it is always with the peasant. We represent these positions, by P,W,G,C. The actions are cross the river alone, cross the river with the wolf, cross the river with the goat, and cross the river with the cabbage. These actions are represented by the numbers 1, 2, 3, 4.

Thse actions are described by the successor predicate. Of course, before we perform an action, we have to check that the next state is safe i.e. the wolf is not alone with the goat or the goat with the cabbage.

The final state is described by the goal_reached predicate. The predicate depth_search looks for a path from the start state to a final state. The predicate print_solution prints the solution.

```
% The program below will find an algorithm for a safe crossing.
% form successor(Current_State, Next_State, Action)
% Of course, Next state must be a safe state.
successor(state(P,W,G,C), state(New_P,W,G,C), 1):-
    opposite(P,New_P),          % the peasant is on the opposite bank
    safe(state(New_P,W,G,C)).
successor(state(Bank,Bank,G,C), state(Other_Bank,Other_Bank,G,C), 2):-

    opposite(Bank, Other_Bank),
    safe(state(Other_Bank,Other_Bank,G,C)).
successor(state(Bank,W,Bank,C), state(Other_Bank,W,Other_Bank,C), 3):-

    opposite(Bank,Other_Bank),
    safe(state(Other_Bank, W, Other_Bank, C)).
successor(state(Bank, W, G, Bank), state(Other_Bank, W, G, Other_Bank),
4):-
    opposite(Bank, Other_Bank),
    safe(state(Other_Bank, W, G, Other_Bank)).

% safe(State) checks that the wolf is not alone with the goat,
% and the goat is not alone with the cabbage
safe(state(P,W,G,C)):- (W=G; P = W), (G=C; P = G).
% opposite relates the left and the right bank.
```

```
opposite(left, right).
opposite(right,left).
% in the final state, everybody is on the right bank
goal_reached(state(right,right,right,right)).

% problem independent depth-search
depth_search(Start, Solution_Path):-
    depth_search2(Start, [Start/none], Solution_Path).
% Solution_Path is a path from Start to a goal state
% together with the actions that brought Start onto this state
% depth_search2 has 3 arguments. The first is the current state,
% the second a path from the start state to the current state,
% and the third is a path from the start state to a solution state.
depth_search2(State, Solution, Solution):-
    goal_reached(State).        % State is a goal state
depth_search2(State, Path, Solution):-
    % get the next state
   successor(State, Next_State, Action),
    % check that the state is not already in Path
    not(member(Next_State/_,Path)),
    depth_search2(Next_State,[Next_State/Action|Path],Solution).
    % find a solution from Next_State

% Now let us find a solution.
solve:- print_title,        % print the title
    % print the initial state
    print_state(state(left,left,left,left)),
   (depth_search(state(left,left,left,left), Sol),
    print_solution(Sol); print_error_message).
% print_title prints the title
print_title :- nl, tab(20), write('SOLUTION TO THE PEASANT"S PROB-
LEM'),
    nl, tab(20), write('================================='),
    nl,nl.
% print_state prints the state
print_state(state(P,W,G,C)):-
    nl, write('The peasant is on the '), write(P), write(' bank, '),
    write('the wolf is on the '), write(W), write(' bank, '),
```

```
    nl, write('the goat is on the '), write(G), write(' bank, '),
    write(' and the cabbage is on the '), write(C), write(' bank.'),nl.

% print_solution prints a solution to the problem
print_solution(Sol):- reverse(Sol,Rev_Sol),      % reverse the list
    delete_first(Rev_Sol,Real_Sol),      % delete the first item
    print_list(Real_Sol).

% print_list prints the actions and the states
print_list([]).
print_list([State/Action|Tail]):- print_action(Action),
    print_state(State), print_list(Tail).

% print_action(Action) prints the action
print_action(1):-nl, write('The peasant crosses the river alone.').
print_action(2):-nl, write('The peasant crosses the river with the wolf.').
print_action(3):-nl, write('The peasant crosses the river with the goat.').
print_action(4):-nl, write('The peasant crosses the river with the cabbage.').
% delete_first deletes the first element of a list
delete_first([Head—Tail],Tail).
%print_error_message sates that there is no solution.
print_error_message:- nl, write('There is no solution from this state.').
```

Below is the output of the query ?-solve.

```
    SOLUTION TO THE PEASANT"S PROBLEM
    ================================

The peasant is on the left bank, the wolf is on the left bank,
the goat is on the left bank, and the cabbage is on the left bank.

The peasant crosses the river with the goat.
The peasant is on the right bank, the wolf is on the left bank,
the goat is on the right bank, and the cabbage is on the left bank.

The peasant crosses the river alone.
The peasant is on the left bank, the wolf is on the left bank,
the goat is on the right bank, and the cabbage is on the left bank.
```

The peasant crosses the river with the wolf.
The peasant is on the right bank, the wolf is on the right bank,
the goat is on the right bank, and the cabbage is on the left bank.

The peasant crosses the river with the goat.
The peasant is on the left bank, the wolf is on the right bank,
the goat is on the left bank, and the cabbage is on the left bank.

The peasant crosses the river with the cabbage.
The peasant is on the right bank, the wolf is on the right bank,
the goat is on the left bank, and the cabbage is on the right bank.

The peasant crosses the river alone.
The peasant is on the left bank, the wolf is on the right bank,
the goat is on the left bank, and the cabbage is on the right bank.

The peasant crosses the river with the goat.
The peasant is on the right bank, the wolf is on the right bank,
the goat is on the right bank, and the cabbage is on the right bank.
true