CAP 5602
Summer, 2011

Lesson 8 Breadth Search

The topics

1. an algorithm for breadth search
2. a more efficient breadth search
3. the peasant, wolf, goat and cabbage problem

1. An algorithm for breadth search

Let us look at the first program for breadth search and see how it works.

```
% breadth search
% this program is taken from Bratko, third edition, pp 252
% solve(Start, Solution) returns a path (in reverse order) from Start
% to a goal
solve(Start, Solution):- breadthfirst([[Start]], Solution).
% breadthfirst([Path1,Path2, ..., ], Solution)
% is satisfied if Solution is an extension to a goal of one of the paths
breadthfirst([[Node | Path] | _ ], [Node | Path]):- goal_reached(Node).
breadthfirst([ Path | Paths], Solution):- extend(Path, NewPaths),
    % extend the first path
    append(Paths, NewPaths, Paths1),
    % append it at the end of the other paths
    breadthfirst(Paths1,Solution).    % search for a solution

extend([Node | Path], NewPaths) :- bagof([NewNode, Node | Path],
    % find a child that is not on the current path
    (successor(Node, NewNode), not(member(NewNode, [Node | Path]))),
    NewPaths), !.
extend(Path,[]).
```

The first argument of breadthfirst([Paths],Sol) is its input, and consists of a list of lists. Each list in the input is a path, listed in reverse order, from

the start node to a current node. Each path contains no duplications and the paths are listed in the increasing order of the length.

At the beginning, the input is [[start]]. If the input to breadthfirst is the empty list, breadthfirst fails. Otherwise, it removes the first path from the input [Path | Paths] and processes it. If the first node of Path is a goal node, then Path is the solution, as we can see from the first clause of breadthfirst.

Otherwise, the predicate extends(Path,NewPaths) finds all children of the current node of Path that are not already in Path. For each such child C, extends creates the path [C | Path] i.e. a path from start to C. The length of [C | Path] is one higher than the length of Path and the new path has no duplicate items. NewPaths is the list of all the paths [C | Path]. After extends returns, breadthfirst appends the NewPaths at the end of Paths, giving Path1 and recursively calls breadthfirst(Path1,Sol).

Let us trace the algorithm for the graph in Figure1 when we enter the query ?-solve(a,S).

We will show only the calls to breadthfirst.

breadthfirst([[a]],S)
breadthfirst([[b,a], [c,a], [d,a]],S)
breadthfirst([[c,a], [d,a], [c,b,a], [e,b,a], [f,b,a], [g,b,a]],S)
breadthfirst([[d,a], [c,b,a], [e,b,a], [f,b,a], [g,b,a],S)
breadthfirst([[c,b,a], [e,b,a], [f,b,a], [g,b,a], [h,d,a], [i,d,a]], S)
breadthfirst([[e,b,a], [f,b,a], [g,b,a], [h,d,a], [i,d,a],], S)
breadthfirst([[f,b,a], [g,b,a], [h,d,a], [i,d,a]], S)
breadthfirst([[g,b,a], [h,d,a], [i,d,a], [d,f,b,a]], S)
breadthfirst([[h,d,a], [i,d,a], [d,f,b,a]], S)
breadthfirst([[i,d,a], [d,f,b,a], [j,h,d,a], [k,h,d,a]], S)
breadthfirst([[d,f,b,a], [j,h,d,a], [k,h,d,a]], S)
breadthfirst([[j,h,d,a], [k,h,d,a], [h,d,f,b,a], [i,d,f,b,a]], S)
breadthfirst([[k,h,d,a], [h,d,f,b,a], [i,d,f,b,a]], S)
breadthfirst([[k,h,d,a], [h,d,f,b,a], [i,d,f,b,a]], [k,h,d,a])
% k is a goal node

After the last call we have a sequence of returns that assign the last output to the preceding one.

breadthfirst([[k,h,d,a], [h,d,f,b,a], [i,d,f,b,a]], [k,h,d,a])
breadthfirst([[j,h,d,a], [k,h,d,a], [h,d,f,b,a], [i,d,f,b,a]], [k,h,d,a])
breadthfirst([[d,f,b,a], [j,h,d,a], [k,h,d,a]], [k,h,d,a])
breadthfirst([[i,d,a], [d,f,b,a], [j,h,d,a], [k,h,d,a]], [k,h,d,a])

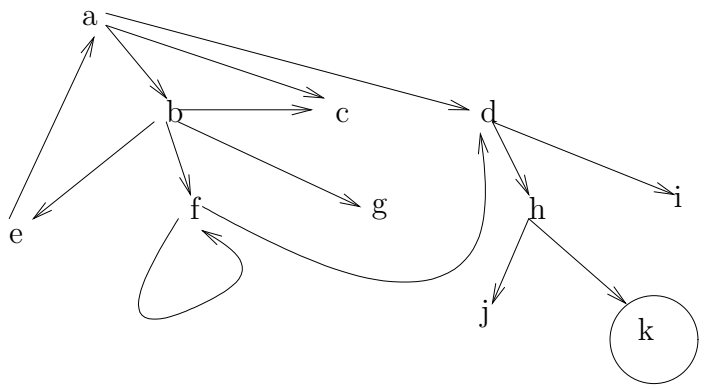

Figure 1: The graph for the breadthfirst

```
...
breadthfirst([[b,a], [c,a], [d,a]],[k,h,d,a])
breadthfirst([[a]],[k,h,d,a])
```

2. Another breadth search program

Here is another version of breadth search.

```
% the second program is a more efficient breadth search
% this program is taken from Bratko, third edition, pp 253
% solve(Start, Solution) returns a path (in reverse order) from Start
% to a goal
solve(Start, Solution):- breadthfirst2([[Start] | Z] - Z, Solution).

% breadthfirst2([Path1,Path2, ..., ], Solution)
% is satisfied if Solution is an extension to a goal of aone of the paths
breadthfirst2([[Node |Path] | _ ] - _ , [Node | Path]):- goal_reached(Node).
breadthfirst2([ Path | Paths] - Z, Solution):- extend(Path, NewPaths),
    % extend the first path
    append(NewPaths, Z1, Z),
    % append it at the end of the other paths
    Paths = Z1,
    breadthfirst2(Paths - Z1,Solution).       % search for a solution

extend([Node | Path], NewPaths) :- bagof([NewNode, Node | Path],
```

```
        % find a child that is not on the current path
        (successor(Node, NewNode), not(member(NewNode, [Node | Path]))),
        NewPaths), !.
    extend(Path,[]).
```

This program represents the lists as a difference of 2 lists. For example, if L1 = [a,b,c,d,e] and L2 = [d,e], the list [a,b,c] is the difference L1 - L2.

The condition is that L2 must be a **suffix** of L1.

So, [a,b,c] can be written as the difference [a,b,c,d,e | Tail] - [d,e | Tail].

Now let us show that the 2 programs compute the same function.

The expand method is the same, so we have to show that whenver the inputs to breadthfirst and breadthfirst2 are the same, so are the inputs to their recursive calls.

The proof is illustrated in Figure 2. Each list is represented by a segment with arrows at both ends. If two segments abut then their concatenation is the segment that begins with the leftmost point of the two and end with their rightmost point.

So, let [Path | Paths] be the input of breadthfirst and [Path' | Paths'] -Z be the input of breadthfirst2 and assume that [Path | Paths] = [Path' | Paths'] -Z.

Then Path = Path' and Paths = Paths' - Z, as shown in the figure. The arguments to the 2 calls to extends are the same, so extends returns the same list NewPaths.

Then, the input to the recursive call in the body of breadthfirst is Paths1, where Paths1 is the result of appending NewPaths at the end of Paths as shown in the first 2 horizontal lines of Figure 2.

The input to breadthfirst2 is Paths'-Z1 and Z is the the result of appending Z1 to NewPaths, as displayed by the 3rd and the 4th horizontal lines of the figure.

This concludes the proof of the input/output equivalence of the 2 programs.

3. The comparison

The second program is more efficient than the first. We ran the two programs with the graph in Figure 1 as input. After each call ?-sove(a,Sol) we called ?-statistics.

Here are the stats for breadthfirst. The big difference was on the CPU time. For breadthfirst we got

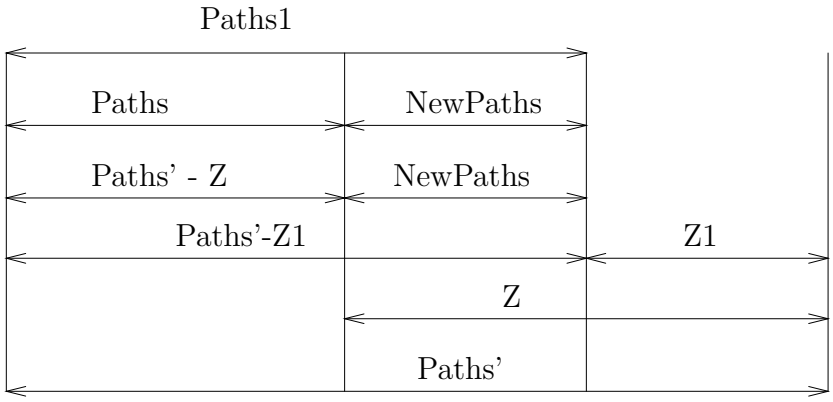

Figure 2: The correctness proof for breadthfirst2

0.11 seconds cpu time for 27,331 inferences

3,811 atoms, 2,532 functors, 2,160 predicates, 44 modules, 60,876 VM-codes.

For breadthfirst2 we had the stats below.

0.05 seconds cpu time for 27,222 inferences

3,809 atoms, 2,532 functors, 2,160 predicates, 44 modules, 60,876 VM-codes