MAU22C00 Lecture 10

John Stalker

Trinity College Dublin

Non-deterministic algorithms

Deterministic algorithms have

- a well defined start state
- a definite rule for what action to take in each state, possibly dependent on input, environment, etc.
- a termination condition
- (probably) a distinction between successful and unsuccessful termination

Non-deterministic algorithms have

- one or more initial states
- a *set* of possible actions to take in each state
- a termination condition, probably with a distinction between successful and unsuccessful-ish termination

Usually we're interested in whether a non-deterministic algorithm *can* terminate successfully, not whether it happens to for a particular choice of actions.

Linguistic examples

Generating elements of a language from a generative grammar is a non-deterministic algorithm.

- The start state is given by the start symbol.
- The actions available at any point are expanding a non-terminal symbol using one of its alternates or replacing a terminal symbol with one of its tokens.
- The termination condition is having no remaining symbols. In this case any termination is considered successful.

We can (but probably shouldn't) handle parsing by the same method. Successful termination now means that the final list of tokens matches the input and unsuccessful means it doesn't.

Note that the input belongs to the language if the algorithm *can* terminate successfully, not if it does for particular choices.

It may terminate unsuccessfully for all possible choices, telling you the input isn't in the language, but it may not terminate for some choices.

Zeroeth order logic example

The satisfiability version of the tableau method for zeroeth order logic can be viewed as a non-deterministic algorithm.

- The start state is one with the statement to the left of the line.
- At each step the actions are to take an unused statement, mark it used, and write down its consequences, in the non-branching case, or *one* of the two possibilities, in the branching case.
- The termination condition is running out of unused statements (successful) or finding a contradiction (unsuccessful).

If the algorithm *can* terminate successfully then the statement is satisfiable.

In this case, unlike the previous one, the algorithm will always terminate, successfully or not.

Similar remarks apply to proving tautologies, but "success" and "failure" mean the opposite of what you'd expect.

Another zeroeth order logic example

The Łukasiewicz formal system can be considered a non-deterministic algorithm.

- The initial state is an empty list of statements.
- In each state our options are to write down an axiom or apply a rule of inference to one or two previous statements.
- Successful termination is writing down the statement we were trying to prove.
- Unsuccessful termination is impossible, but non-termination is very common.

The given statement is a theorem if the algorithm *can* terminate successfully.

One complication is that substitution allows replacement of a variable by *any* expression and there are infinitely many expressions.

There are other systems, e.g. Nicod (see notes), which don't have this problem, but we can cope with this.

Making non-deterministic algorithms deterministic

How do we know whether the algorithm *can* terminate successfully?

Idea: simulate *all* possible choices, so if one of them works we won't miss it.

This is more or less what we did for zeroeth order logic. The tree structure of the tableau keeps track of the possible choices for us.

We can apply the same trick essentially any non-deterministic algorithm, using a tree to keep track of the choices.

There are some complications if there are sometimes infinitely many choices.

Trees of trees

When the non-deterministic algorithm constructs a tree, as in parsing, we get a tree of trees.

Each node in our tree of possible program states is labelled by a partially filled in parse tree.

The parse trees are all finite but the tree of possible states is usually infinite.

Sometimes different program paths lead to identical states. You can take advantage of this by replacing the state tree with a "directed acyclic graph". There are practical parsing algorithms which use this trick to avoid exponential growth.

Mixed metaphors

The terminology for trees is a mess.

In keeping with the tree metaphor we have the “root” and “leaves”.

For some reason trees are usually drawn growing downwards.

We also have “parents” and “children”.

Every node has exactly one parent, except the root, which has none.

Leaves are nodes with no children.

Nodes which share the same parent are called “siblings”.

Depth first versus breadth first traversal

To check whether the algorithm could successfully terminate we have to visit every node in the tree, but there might be infinitely many.

There are two standard ways to traverse a tree, depth first or breadth first.

Depth first means visiting all of a node's children before moving on to any of its siblings.

Breadth first means visiting all of a node's siblings before moving on to any of its children.

Depth first is usually easier to program, so if you've used a program which traverses a tree that's probably what it did.

Picture (depth first)

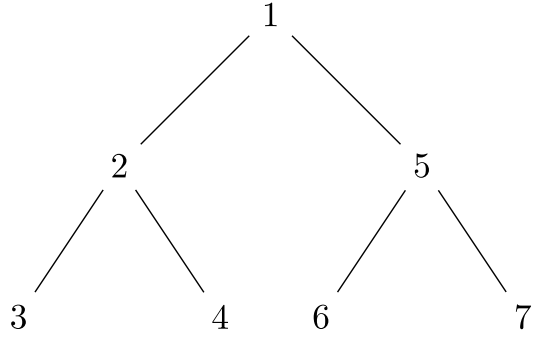

Figure 1: depth first tree traversal

Picture (breadth first)

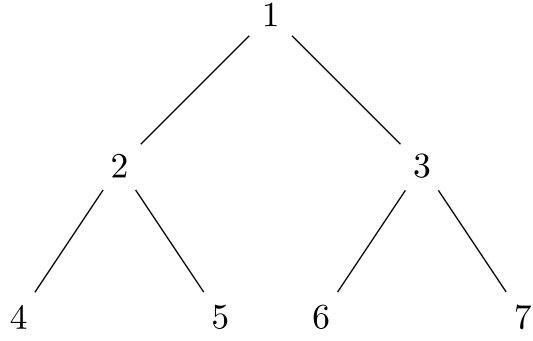

Figure 2: breadth first tree traversal

Infinite trees

Depth first traversal will work, i.e. visit every node, on finite trees and some, but not all, infinite trees.

Breadth first traversal will work, even on infinite trees, provided no node has infinitely many children.

Working means that if what we're looking for is there we will find it. If it's not there the traversal will continue forever, unless the tree was finite.

The trees in the tableau method for ZOL were finite, so either method works.

The trees for Łukasiewicz have branches of finite length, but infinitely many children. Neither method works for this.

The trees (of trees) for parsing usually have infinitely long branches but have only finitely many children, unless there are symbols with infinitely many tokens, so breadth first generally works and depth first generally doesn't.

Hybrid traversal methods

There are traversal methods which work even for trees with infinite branches and infinitely many children, provided the infinities aren't too bad.

When there are more than two children we can group them as the first one and the others, then create a new node for the others.

The result is a binary tree containing all the nodes of the original tree, on which we can do breadth first traversal.

Infinite children are hard to draw so we'll have to make do with three.

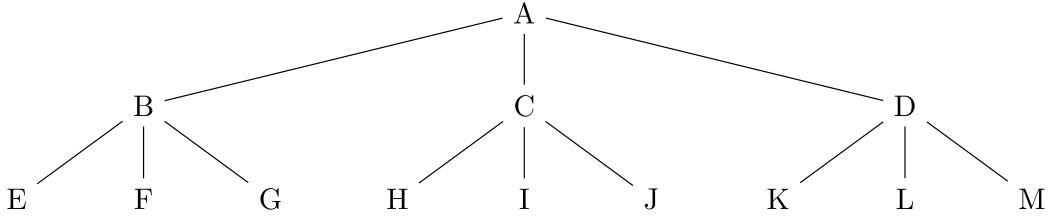

Figure 3: a non-binary tree

The corresponding binary tree

This is the corresponding binary tree.

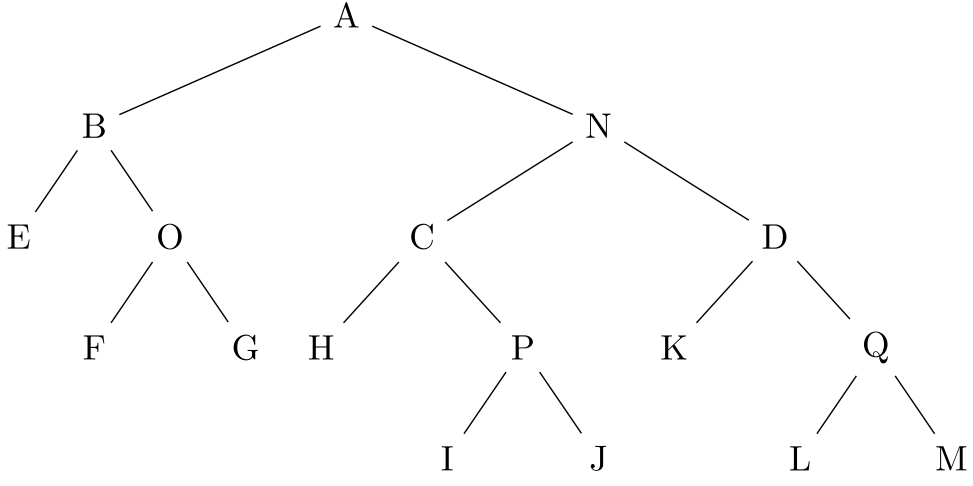

Figure 4: the corresponding binary tree