Chapter D2: Algorithms on Trees

There are some nice algorithms on trees: both greedy algorithms and recursion/dynamic programming can be successful.

D2.1 Traversal of Rooted Trees

For a rooted tree, a traversal visits the nodes of the tree in a specific order:

- In a **preorder** traversal, a node is visited before either of its children
- in a **inorder** traversal, a node is visited after visiting its left child and before its right child
- in an **postorder** traversal, a node is visited after both its children.

For example, in a binary search tree, an inorder traversal prints out the data in order.

Assume the tree is stored with pointers from parent to children. Then a preorder, inorder or postorder traversal can be performed in linear time with the obvious recursive algorithm. For example:

```
Inorder(root)
  if root=nil exit
  Inorder( LeftChild(root) )
  visit(root)
  Inorder( RightChild(root) )
```

There is a natural generalization to nonbinary rooted trees of preorder and postorder traversals, and the recursive algorithm again takes linear time.

D2.2 Domination

A set \( S \) of nodes is said to be **dominating** if every node in the graph is either in \( S \) or adjacent to a member of \( S \). (For example, in a distributed network copies of a database might be stored at a dominating set of nodes.) The **domination number** of a graph is the minimum size of a dominating set. For example, the domination number of a path on \( n \) nodes is \( n/3 \) rounded up.

As in the case of maximum matching, a greedy algorithms works for trees:
Again we need a proof that this works. Left as an exercise.

One can implement both the matching algorithm and the domination algorithm on trees using a postorder traversal. In the former, every time one visits a node, if there is a child available then one pairs the node with one of its children. In the latter, every time one visits a node, if some child needs to be dominated then the node is taken.

**D2.3 Independent Domination**

However, most graph problems cannot be solved on trees with a simple greedy algorithm. But many can be solved by a recursive postorder traversal algorithm, though it looks a bit dark and mysterious the first time you see it.

We consider an instance of this. In a graph, a set of nodes is said to be *independent* if no two of them are joined by an edge. We define an *ID-set* as one that is both independent and dominating. And then define the *ID-number* of a graph as the minimum size of an ID-set.

**Example.** The following graph has domination number 3 (take for example \{A, D, H\}). But it has ID-number 4 (take for example \{B, C, D, H\}).

![Diagram of a graph]

The paradigm entails working from the leaves up to the root, as in a postorder traversal. For each node \(v\), we define the subtree rooted at \(v\), denoted \(T_v\), as the tree consisting of \(v\) and all its descendants. The trick is, like in dynamic programming, to solve a slightly more general problem on the subtrees. Or rather, to solve several versions of the original problem on the subtrees.
Consider an ID-set $S$ for the whole tree. Now let $S_v$ be the subset of $S$ that is within the subtree $T_v$ and focus on it as a subset of $T_v$. First, $S_v$ is an independent set. Second, $S_v$ dominates all of $T_v$, except possibly the root $v$. For the root there are three possibilities:

1. $v$ is in $S_v$,
2. $v$ is not in $S_v$ but is dominated by one of its children,
3. $v$ is not dominated by $S_v$.

The key is to define a parameter for each of the three possibilities. We define $f_i(T_v)$ as the minimum size of a subset $S_v$ of nodes in $T_v$ such that:

- the set $S_v$ is independent;
- it dominates all of $T_v$ except possibly $v$; and
- the root $v$ has property $(i)$.

One can calculate parameters $f_1$, $f_2$, and $f_3$ recursively (see below).

So our algorithm is: visit the nodes with a postorder traversal. At each node, the values for children are already calculated, and so one can calculate the values for the parent by the recursive formulas. The result is a linear-time algorithm. (That this is true for non-binary trees requires a little bit of work to show.)

Furthermore, to determine the actual minimum ID-set $S$, one can work back down the tree: one looks at each stage at which of the $f_i$ was taken at the parent, and calculates whether the node is in or out of $S$.

\textbf{Gory Formulas}

Here are the recursive formulas for these three parameters. If $v$ is going to be in $S_v$, then each of its children must be in state 2 or 3. Thus,

$$f_1(T_v) = 1 + \sum_c \min(f_2(T_c), f_3(T_c))$$

where the sum is over all children $c$ of $v$.

If $v$ is not in $S_v$, then none of its children can be in state 3. Further, if $v$ is not dominated by $S_v$, then by definition none of its children can be in state 1. Thus

$$f_3(T_v) = \sum_c f_2(T_c)$$

Finally, if $v$ is dominated, then at least one of its children is in state 1. So $f_2(T_v)$ is the minimum sum of $f_1(T_c)$ or $f_2(T_c)$ over its children, subject to the constraint that at least one of these must be $f_1$. This evaluates to:

$$f_2(T_v) = \begin{cases} 
\sum_c \min(f_1(T_c), f_2(T_c)) & \text{if } f_1(T_c) \leq f_2(T_c) \text{ for some } c, \\
\sum_c f_2(T_c) + \min_c f_1(T_c) - f_2(T_c) & \text{otherwise}
\end{cases}$$
Now, having done the traversal, one can calculate the actual ID-number of the tree as

\[ \text{ID-number} = \min(f_1(T_r), f_2(T_r)) \]

where \( r \) is the overall root.

### D2.4 Weighted Matching in Trees

The weighted matching problem is another problem that can be solved in trees by postorder traversal. We assume the edges have weights and want the matching whose sum of weights is as large as possible.

The greedy algorithm does not work here. Consider for example the following tree. If we use the previous greedy algorithm we would start with the edge \( BE \), but this is wrong: the maximum weight matching consists of \( AB \) and \( CG \). Similarly, the greedy approach of taking the maximum weight edge also fails.

There is a recursive postorder traversal algorithm, like the one for ID-set above. At each level we calculate for the subtree \( T_v \) rooted at \( v \) two quantities:

- the maximum weight matching of \( T_v \), denoted \( m(T_v) \); and
- the maximum weight matching over all matchings in \( T_v \) such that the root is not in the matching, denoted \( mnr(T_v) \).

The maximum weight matching in the subtree \( T_v \) is easy to calculate if the root is specifically excluded. For, one takes the best matching in each of the subtrees rooted at each of the children and simply combines them:

\[ mnr(T_v) = \sum_{\text{children } c} m(T_c) \]

If the root is allowed to be in the matching, then apart from the above situation, one also has the possibility that there is an edge in the optimal matching connecting \( v \) to one of its children. Say that child is \( c \). Then in the subtree rooted at \( c \) one must take an
optimal matching that does not include $c$, while in the other subtrees we can take any matching. This gives the following (impressive) formula:

$$m(T_p) = \max \left\{ \frac{\text{mnr}(T_p)}{\max_{\text{children } c} w(p - c) + \text{mnr}(T_c) + \sum_{d \neq c} m(T_d)} \right\}$$

Again, we visit the nodes with a postorder traversal, calculating at each node the two values using the values for children are already calculated. The result is a linear-time algorithm.

**Exercises**

1. Show that the greedy algorithm for domination number works on trees.

2. An obvious strategy for a greedy algorithm for domination number is to always pick the remaining node with maximum degree, and remove it and its neighbors. Show that this algorithm fails, even on trees.

3. The independence number of a graph is the maximum size of an independent set. Show that the domination number, independence number, and ID-number can all be different.

4. Provide a greedy algorithm for independence number on trees.

5. Illustrate the workings of the maximum-weight matching algorithm on the following weighted tree.

6. Illustrate the workings of the ID-number algorithm on the above tree, ignoring the weights.