Sim,plified Copy Trees

5.1 Introduction

In the previous chapter, we examined and presented various techniques for incremental graph propagation. It is argued that optimal overhead complexity is very difficult to achieve. An optimal graph propagator may not necessarily result in an optimal incremental evaluation algorithm[Ho087]. In this chapter, we argue that there are various issues that needs consideration before we develop the actual optimal incremental evaluator.

The first issue is to check if the client process calls for consistent values after every individual modification to the dependency graph. If this is not the case, then in a series of modifications, some modifications may be reversed by some others in the whole series. It is, therefore, possible to improve the performance of the evaluator by checking the modifications before invoking the graph propagator.

5.2 Identity Dependencies

It is not possible to change the vertex functions in a dependency graph, or optimise them in a manner so as to reduce the number of vertices, unless the underlying graph evaluable scheme is known. But it is possible to devise an efficient evaluation scheme for graphs when identity functions are present. AG based graph evaluable schemes generally utilise
many such identity functions. These are used for communicating values between widely separated derivation tree nodes. Hoover [Hoo87] has identified two types of implicit dependencies that can occur in dependency graphs. They are Copy Rules, which are chains of edges whose vertices have identity functions as dependencies; and Aggregates, which are structures used for communicating keyed values.

**Definition 5.1:** Let $u_i, u_k$ be two vertices in $D(T)$ such that $u_k$ is transitively dependent on $u_i$, i.e. $u_i \rightarrow u_k$, there is a path from $u_i$ to $u_k$. If all the intermediate vertices on the path $u_2, \ldots, u_{k-1}$ are identity functions then the value at $u_i$ is communicated through $u_j$ to $u_k$.

These nodes are redundant as they do not store the values of any intermediate computations. We define $O(T)$, an optimised dependency graph over $D(T)$, where all such identity chains are replaced by a single edge from $u_i$ to $u_k$.

Let $AFFECTED'$ be the set of vertices in $AFFECTED$ whose vertex functions are identity dependencies. The optimal reevaluation set becomes $(AFFECTED-AFFECTED')$ and this if achieved can result in considerably speeding up the graph propagation algorithm.

### 5.2.1 Copy Rules

In a dependency graph $D(T)$, if the vertex function associated with a vertex simply copies the value from another node, it is called a copy rule.

![A Copy Rule](image)
Let \( u_i \) and \( u_k \) be any two vertices in \( D(T) \) such that there is a directed edge from \( u_i \) to \( u_k \), as shown in figure 5.1. If the indegree of \( u_k \) is one and the vertex function of \( u_k \) is of the form \( u_k \cdot \text{val} := u_i \cdot \text{val} \). Then the vertex function of \( u_k \) is a copy rule and \( u_k \) is a copy vertex.

### 5.2.2 Copy Rule Chain

Let \( u \) be a path in \( D(T) \) such that all vertices \( u_i (0 \leq i \leq n) \) are copy vertices and \( u_0 \) is not a copy vertex. Any direct successor of \( u_n \), if any, is not a copy vertex.

Then the path \( u_n (0 \leq i \leq n) \) is a copy rule chain and \( u_0 \) is the copy head of this chain.

It is clear that \( \text{val}(u_n) = \text{val}(u_0) \), and all the intermediate vertices in this chain are just used to communicate the values.

### 5.2.3 Copy Trees

A vertex may be the copy head for more than one copy rule chain. Suppose, \( u_0 \) is the copy head for a number of copy rule chains \( \sigma_1, \ldots, \sigma_k \). As every vertex in this set of copy rule chains is a copy vertex, with an indegree of one, the paths \( \sigma_i, 1 \leq i \leq k \) form a tree with \( u_0 \) as the root of the tree. Except \( u_0 \), all the vertices are copy vertices. This tree is a copy tree in \( D(T) \) defined as \( \text{CT}(u_0) \). \( u_0 \) is the copy from the tree \( \text{CT}(u_0) \).

Let \( u_0 \) be the copy head for a number of copy rule chains \( \sigma_1, \ldots, \sigma_k \).

Then the \( \text{successor}_{\text{CT}}(u_0) \) is defined as the set of all vertices at the tail of paths \( \sigma_i \).

---

**Figure 5. II** A Copy Tree
A simple copy tree is shown in Figure 5.II. The copy tree $CT(a)$ has $a$ as the copy head. The set \{l,m,n,j,o\} denotes the successor$_{CT}(a)$.

An inherent shortcoming with $AG$ based systems is that nodes which are widely separated are unable to communicate directly. Let us suppose two derivation tree nodes $x$ and $y$ need to communicate some values. The only way to achieve this, if the $AG$ formalism is to be strictly followed, is to attach additional copy attributes to all the nodes on the path between $x$ and $y$. This not only increases the size of the $AG$, but also the time complexity of the $AG$ evaluator. We can achieve some degree of optimality in an evaluator if we could devise a mechanism to bypass all the copy rule chains.

5.3 Structure Trees

Hoover [Hoo87] has proposed a data structure called the Structure Tree. This minimizes the number of nodes in the copy tree, while preserving the tree structure.

Definition 5.2: Let $T=(V,E)$ be a tree and let $N$ be a set of vertices $N \subseteq V$. Let $SV$ be a subset of $V$ defined by the following three rules.

1. If $V \in N$, then $V \in SV$.
2. If $V_1, V_2 \in SV$ and $x$ is the least common ancestor of $V_1, V_2$ in $T$, then $x \in SV$.
3. If $N \neq \phi$ and $x$ is the root of $T$, then $x \in SV$.

Then a structure tree of $T$ wrt $N$, denoted by $ST(T,N)$ is a tree defined by $ST(T,N)=TISV$ where $ST$ is the projection of $T$ on to $SV$.

The structure tree corresponding to the tree of Figure 5.III is shown in the Figure 5.IV.
Let $T$ be a tree with a vertex subset $N$, then the number of vertices in $ST(T,N)$ is $O(|N|)$. The proof for this may be found in [Hoo87].

A structured tree is therefore, a structure preserving projection of a tree with respect to a given subset of vertices and the size of the tree is of the order of the given subset. The set $N$ is the set of marked vertices.

**Definition 5.3**: Let $CT(u_0)$ be a copy tree in $D(T)$ rooted at $u_0$. Let $N$ be a subset of marked vertices in the copy tree. We define a simplified copy tree $C_p(T(u_0),N)$ as:

$$C_p(T(u_0),N) = T(u_0) \cup ST(T(u_0),N)$$

An edge $e$ in $C_p(T(u_0),N)$ is called a reduced copy edge if $e$ is originally an edge of $ST(T(u_0),N)$ and it is called a simple copy edge otherwise.
5.4 Self Adjusting Binary Search Trees

Sleator and Tarjan [ST83] have proposed a very efficient solution to represent and maintain dynamic trees. The underlying tree is represented as a tree of dynamic paths, amenable to all operations of link-cut trees. Each path in the tree consists of one or more nodes, and each node in the tree is a member of exactly one path. A single node path is also allowed.

Assume that the edges in the tree are directed from vertices to their parents. The edges in the tree are first partitioned into two kinds, solid and dashed, so that any vertex is the tail of at the most one solid edge. \((a_1, ..., a_j)\) is a solid path if each edge \((a_i, a_{i+1})\) \((1 \leq i \leq j)\) is a solid edge. \(a_1\) is not the tail of any solid edge and \(a_j\) is not the head of any solid edge.

For a tree \(T\), the self adjusting search tree \(\tilde{T}\), is a tree with following properties:
1. For each solid path \((a_1, \ldots, a_j)\) in \(T\), there is a binary search tree \(B(a_1, \ldots, a_j)\) in \(\hat{T}\).

2. For each dashed edge \((a, b)\) in \(T\), there is a dashed edge \((a, b')\) in \(\hat{T}\), where \(b'\) is the root of the binary search tree containing \(b\).

The *path-successor* of the path whose last node is the root of the tree is *null*. The path-successor for every other node in the tree is the parent of the last node in the path containing that node. Sleator and Tarjan[ST85] use two basic operations to maintain link-cut trees. \textit{Splay}(a) moves the vertex \(a\) to the root of its binary search tree while rearranging the rest of the original path from \(a\) to the root, so that any vertex on the path is about half as far away from the root as it used to be. \textit{Expose}(a) creates a solid path starting from the node \(a\) and ending at the root of \(T\).

**Example**

![Path Tree T](image)

*Figure 5. V Path Tree T*
In the example given in Figure 5.V, a normal tree rooted at \( a \) is visualised as a path tree \( T \). There are three paths: \((t,...,a)\), \((u,...,f)\) and \((s,...,d)\), while the rest are single node paths.

In the self adjusting tree \( \overline{T} \), of Figure 5.VI, the above three paths are represented by the solid subtrees rooted at \( e, j \) and \( h \). Single node paths \( n, p \) etc. are connected to their parents through dashed edges.

### 5.4.1 Primitives

Since binary trees are used to represent paths in the original tree \( T \), it is obvious that the symmetric order of the nodes in the self-adjusting binary tree is the same as the order of the nodes in the path represented by the tree. So each solid tree in \( \overline{T} \) represents a path in \( T \). A binary tree representing a path is not unique, and therefore, neither is the self adjusting tree representing a path tree.

As has been mentioned earlier, most of the path operations restructure solid trees by splaying[ST85], ie, moving the given node to the root of the solid subtree while preserving
the symmetric order. A path is designated by the node at the current root of its solid subtree. We need some to define some primitives for path functions.

\textbf{makepath}(v): a new path is created containing a single node v.

\textbf{findpath}(v): returns the path containing node v (as a side effect v will become the root of its solid subtree).

\textbf{findlast}(p): returns v, the last node of path p (as a side effect v becomes the root of its solid subtree).

\textbf{join}(p, v, q): creates and returns a new path formed as a consequence of joining paths p, a one node path containing v and the path q, in that order. Either p or q can be null.

\textbf{split}(v): the path containing v is split into three parts, the path before v, a one node path v, and the path after v (v should be the root of its solid subtree). The \textit{below} and \textit{above} paths are returned as a pair.

We do not elaborate on these path functions as they can be found in [ST83].

Tree operations can restructure a solid tree, and at all times, a path is designated by the current root of its solid subtree. This means that a path can be treated as a tree vertex by the algorithm, but we need to be sure that the node we are referring to is the root of the solid binary subtree representing the path.

\section*{5.4.2 Representing Simplified Copy Trees as Path Trees}

We use path trees to represent simplified copy trees. Therefore, the path trees will represent the underlying copy trees and the structure tree will be superimposed on the copy trees. Each structure tree edge will represent a single path in the path tree. The paths must satisfy the following \textit{structure tree invariant}:

- All edges entering structure tree nodes are dotted. In other words, each structure tree edge is the beginning of a path.
- Each structure tree edge from a structure tree node \( x \) to its structure tree parent \( y \) is represented by a single solid path from \( x \) to a child \( v \) of \( y \), and a dotted edge from \( v \) to \( y \). If \( y \) has \( m \) children in the underlying copy tree, then we number the structure tree children from 1 to \( m \), even though some of them may be null. Therefore, if \( v \) is the ancestor of \( x \) and the \( i^{th} \) child of \( y \), then \( x \) is the \( i^{th} \) structure child of \( y \).

Figure 5. VII  Copy Tree as a Tree of Paths

Figure 5.VII shows the copy tree of Figure 5.II as a tree of paths. Figure 5.VIII shows a self adjusting binary search tree for the path tree of Figure 5.VII.
5.5 Data structures

All the vertices in the dependency graph $D(T)$ are partitioned into three dynamic categories, at any point of time during the lifetime of the graph.

1. Non-copy vertices (ncp)
2. Copy vertices (cpy)
3. Copy head vertices (cph)

5.5.1 Non-Copy Vertices

A vertex $v$, such that $type(v) = ncp$, is a normal graph vertex. The $indegree(v) \geq 0$ and $outdegree(v) \geq 0$. Links to the successors of vertex $v$ are maintained in the normal manner and the function $successor(v)$ returns all the normal graph successors of $v$. Each vertex will have the following fields:

```plaintext
vertex successor[1..m]
```
vertex predecessor[1..n]

where \( m \) and \( n \) is the number of successors and predecessors respectively of the vertex.

### 5.5.2 Copy Head Vertices

If \( \text{type}(v) = \text{cph} \), then \( v \) is the beginning of a number of copy rule chains merged into a copy tree. In other words, if there exists a copy tree \( CT(v) \) in \( D(T) \), then \( v \) is of the type \textit{copy head}. A \textit{cph} node will have the following fields:

- vertex successor[1..m]
- vertex predecessor[1..n]
- vertex parent - null, as it is the root of the copy tree.
- vertex child[1..k] - pointers to the k children of the root.
- boolean is_marked - always true.
- integer child_no - left undefined as it is a root vertex.

if is_st_vertex then

- vertex st_parent
- vertex st_child[1..k]
- integer st_child_no

### 5.5.3 Copy Vertices

A vertex \( v \) such that indegree(\( v \)) = 1 and semantic function of \( v \) is a copy rule of the form \( \text{semf}(a): \text{val}(a) := \text{val}(\text{predecessor}(a)) \), where \( \text{predecessor}(a) \) gives the predecessor of \( a \) in \( D(T) \), then \( v \) is a copy vertex and type(\( v \)) = \text{cpy}.

The structure of \textit{cpy} vertices is as follows:

- vertex parent
- vertex child[1..k] - pointers to the k children of the vertex.
- integer child_no
- boolean is_marked

if is_marked then

- vertex successor[1..m]

boolean is_st_vertex

if is_st_vertex then

- vertex st_parent
- vertex st_child[1..k]
- integer st_child_no
In the case of \textit{cph} vertices, we maintain some redundant fields like \texttt{child\_no}, \texttt{is\_st\_vertex}, \texttt{parent}, \texttt{st\_parent}, to maintain uniformity with the \textit{cpy} vertices. The field \texttt{predecessor} refers to the normal graph predecessors of the vertex. The field \texttt{successor} refers to the normal graph successors of the \textit{cph} vertices and \textit{cpy} vertices (if the flag \texttt{is\_marked} is true). The flag \texttt{is\_st\_vertex} is always true for a \textit{cph} vertex and is true for a \textit{cpy} vertex if its \texttt{is\_marked} flag is true or it is a structure tree vertex. The \texttt{child\_no} of a vertex is an integer which gives the child number of the vertex in relation to its parent. The \texttt{st\_child\_no} of a vertex gives the structure tree child number of the vertex with respect to its structure tree parent. If there is a path from a vertex \(x\) to a child \(y\) of its structure tree parent \(z\), and \(y\) is the \(i\)th child of \(z\), then the \texttt{st\_child\_no} of \(x\) is \(i\). The fields \texttt{child\_no} and \texttt{st\_child\_no} are left undefined for \textit{cph} vertices. The \texttt{parent} and \texttt{st\_parent} fields are pointers to the copy tree parent and the structure tree parent respectively of the vertex. In the case of \textit{cph} nodes, both of them are null.

### 5.6 Path Functions

Two additional path functions in \cite{ST83} need to be modified in order to apply them to structure trees: they are \textit{Splice} and \textit{Expose}. These functions modify the self-adjusting tree without affecting the underlying path tree.

#### 5.6.1 Splice

Splice converts a dotted edge containing a path \(p\) to its path\_successor \(v\) into a solid edge. Successor of \(p\) must be \(v\) and \(v\) must be the root of its structure tree root. If \(v\) has both left and right children in its solid subtree, then the path containing \(v\) is split into three parts: the path before \(v\), a single node path containing \(v\), and the path after \(v\). If path \(v\) is null, then the returned path will have \(v\) as its first node.
Algorithm 5.1 Splice

5.6.2 Expose

The function `expose` creates a solid path beginning at the given node and ending at the root of the path tree unless a structure tree node is encountered.

```
path function expose(vertex v)
vertex w;
path p:=null;
repeat
    w:= path_successor(findpath(v));
    p:= splice (v,p);
    v:=w;
until v:=null or v.is_st_vertex;
path_successor(p):=v;
return p;
end expose;
```

Algorithm 5.2 Expose

The function `expose` is used in most of the tree operations. It creates a solid path beginning at the node `v` to the root of the tree unless a structure tree vertex is detected by the condition `is_st_vertex` in the `until` statement. `Expose` does not maintain the structure tree invariant, so routines calling `expose` must ensure that the invariant is restored immediately after calling `expose`. The path returned by `expose` is the vertex where the last `splice` was performed.
5.6.3 Mark

The *mark* operation adds a vertex \( v \) to the set of marked vertices of the copy tree, ensuring that the structure tree definition is still satisfied in the resulting tree. There are a number of ways in which a marked vertex can be placed in a structure tree. There are two primary cases:

1. If the vertex \( v \) is already a structure tree vertex, though it was not previously marked.
2. If the vertex \( v \) is not a structure tree vertex, then some ancestor of \( v \) must be a structure tree vertex.

In the first case, we need not do anything as the vertex is already in the structure tree. For the second, there are three possibilities as shown in Figure 5.9:

2.1 Vertex \( v \) lies on a path connecting two structure tree nodes.
2.2 Vertex \( v \) has an ancestor which lies on a path connecting two structure tree vertices.
2.3 Vertex \( v \) has an ancestor which is a structure tree node and it does not lie on any path connecting any two structure tree nodes.

After performing an expose on \( v \), \( \text{path}_\text{successor}(p) \) will return its structure tree ancestor into \( x \), while \( \text{findlast}(p) \) returns \( c \), the copy tree child of \( x \). If \( c \) is not on the path connecting \( x \) to any of its structure tree children, then we have the subcase 2.3. We make a new structure tree edge by calling \( \text{make}_\text{st}_\text{edge}_\text{m}(p,q,r) \), which makes \( q \) the \( r^{th} \) structure tree child of \( p \). If \( c \) is on a path connecting \( x \) to one of its structure tree children, then \( w \) contains the representative vertex of the path returned by the expose operation. Further, if \( v \) and \( w \) are not the same, we have subcase 2.2 implying that an ancestor \( w \) of \( v \) is on a path connecting \( x \) to its structure tree child \( y \). A new structure tree edge is connected between \( v \) and \( w \), and the existing structure tree edge between \( x \) and \( y \) is split into two structure tree edges connecting \( x \) to \( w \) and \( w \) to \( y \). If \( w \) and \( v \) are the same, we have the subcase 2.1, and we simply split the existing structure tree edge into two. The function \( \text{make}_\text{st}_\text{edge}(p,c) \) makes \( c \) the structure tree child of \( p \), but calls \( \text{findlast}(\text{findpath}(p)) \) to ascertain the proper child number.
Figure 5. IX Marking a vertex

Algorithm 5.3 Mark
5.6.4 Path_compress
The unmark and cut operations should remove the unmarked structure tree nodes which may no longer be needed according to the structure tree definition. An auxiliary procedure is needed to check whether a node which must not be marked should be removed from the structure tree or not, satisfying the structure tree definition at the same time. If the node has more than one structure tree children, then it must not be removed, in order to maintain the structure tree invariant. Otherwise, \( v \) is removed. The function \( \text{only}_\text{st}_\text{child}(v) \) returns the structure child of \( v \) if it has only one structure child, otherwise it returns a \( \text{null} \). \( v \) is removed from the structure tree and a single structure tree edge replaces two former structure tree edges incident on \( v \).

```
procedure path_compress(vertex v);
    vertex child,parent ;
    if v.is_marked then return; /* cannot remove marked vertex*/ .
    child:= only_st_child(v);
    if child := null then return ; /*v has other st_child and is still needed*/
    parent:=v.st_parent ;
    v.st-parent:= null;
    v.st_child[child.st_child_no]:= null;
    v.is_st_vertex:= false;
    if parent ~ null then
        make_st_edge_m(parent,child,v.st_child)
    else
        child.st_parent:= null;
        path_successor(splice(v,findpath(child))):= parent;
end path_compress;
```

Algorithm 5.4 path_compress

5.6.5 Unmark
Unmark uses \( \text{path}_\text{compress} \) to simplify its work. This routine unmarks a vertex \( v \) in the structure tree possibly removing the vertex and some other vertices from the structure tree. There are two cases here. First, if \( v \) is not a leaf node, then we simply call \( \text{path}_\text{compress} \) to remove \( v \) from the structure tree. If \( v \) is a leaf node, the structure tree
edge between \( v \) and its structure tree parent \( w \) is removed, and \textit{path\_compress} called with \( w \) as its argument.

\begin{verbatim}
procedure unmark(vertex v);
    vertex w = v.st_parent;
    v.is_marked = false;
    v = findpath(v);
    if v has no structure tree child then /*it is a leaf of the ST*/
        begin
            if w = null then
                begin
                    w.st_child[v.st_child_no] = null;
                    v.is_st_vertex = false;
                    v.st_parent = null;
                    path_compress(findpath(w));
                end;
            else
                path_compress(v);
        end;
    end unmark;
\end{verbatim}

Algorithm 5.5 Unmark

**5.6.6 Link**

The \textit{link} operation links two nodes \( v_1 \) and \( v_2 \), with \( v_1 \) becoming the parent of \( v_2 \). Their associated copy trees and structure trees are also linked together. In order to simplify \textit{link}, \( v_1 \) is marked if it is not already a node in the structure tree. The trees containing the two vertices are then linked together. Afterwards, \( v_1 \) is unmarked if necessary.

\begin{verbatim}
procedure link(vertex v_1, v_2);
    boolean remove = not v_1.is_st_vertex;
    if remove then mark(v_1);
    path_successor(findpath(v_2)) = v_1;
    make_st_edge_m(v_1, v_2, v_2.child_no);
    if remove then unmark(v_1);
end link.
\end{verbatim}

Algorithm 5.6 Link
5.6.7 Cut

The cut operation removes a subtree rooted at $v_2$ from the copy tree, at the same time performing the necessary structure tree manipulations. The edge connecting $v_2$ to its copy tree parent $v_1$ is cut, and $v_2$ becomes the head of a new copy tree. By marking $v_i$, we can easily find if the subtree containing $v_2$ contains any structure tree nodes. Vertex $v_2$ is also marked as it becomes the head of a separate copy tree now. Afterwards $v_i$ can be unmarked if necessary.

```plaintext
procedure cut(vertex $v_1,v_2$);
  boolean remove := not $v_1$.is_st_vertex;
  if remove then mark($v_1$);
  mark($v_2$);
  path_successor(findpath($v_2$)) := null;
  $v_2$.st_parent:=null;
  if $v_1$.st_child[$v_2$.child_no]~null then
    $v_1$.st_child[$v_2$.child_no]:=null;
  if remove then unmark($v_1$) else path_compress(findpath($v_i$));
end cut;
```

Algorithm 5.7 Cut

5.6.8 Maketree

The function maketree creates a new tree node, and its is marked flag is set to true. If $v$ is already a vertex of the type ncp, then we copy all the normal graph predecessors and successors of the old vertex into this new vertex. We may assume that whenever a new vertex is created through maketree, it will have the default type cph.

```plaintext
vertex maketree($v$)
  path_successor(makepath($v$)):=null;
  $v$.parent:=null;
  $v$.child_no:=null;
  $v$.is_marked:=true;
end maketree;
```

Algorithm 5.8 Maketree
5.7 Analysis of Running Time

In analyzing the running of these routines, much of the analysis in [ST83] can be used. The main difference has to do with the expose operation. Because expose as described here does not always create a solid path all the way to the root, the analysis must be slightly modified. First, we need a few definitions regarding nodes in the underlying tree.

**Definition 5.4:** The size of a node $v$ is the number of descendants of $v$ in the underlying tree, including $v$ itself.

**Definition 5.5:** An edge from $v$ to its parent $w$ is heavy if $2 \cdot \text{size}(v) > \text{size}(w)$ and light otherwise.

**Lemma 5.1.** If $v$ is any node, there is at most one heavy edge entering $v$.

**Lemma 5.2** There are at most $\lfloor \log n \rfloor$ light edges on the tree path from $v$ to the tree root.

**Proof:** The proof of Lemma 5.1 is obvious. For Lemma 5.2, we call a node light if the edge to its parent is light. If the size of a light node is $x$, then the size of its parent must be at least $2x$. If there are more than $\lfloor \log n \rfloor$ light edges on the tree path from $v$ to the tree root, then the weight of the root must be at least $2^{\lfloor \log n \rfloor + 1} > n$, a contradiction.

These lemmas allow us to prove the following theorem.

**Theorem 5.1:** A sequence of $m$ structure tree operations including $n$ maketree operations requires $O(m)$ path operations and in addition, at most $m$ calls to expose. The expose calls require $O(m \log n)$ splices, each of which requires $O(1)$ time.

**Proof:** The **split** and **join** calls in **splice** are constant time operations, since no splaying is needed. Therefore, each **splice** takes a constant number of path operations, so $m$ structure tree operations require $O(m)$ path operations. **Mark** can call **expose**, as can **link** and **cut** since they can call **mark**. **Unmark** and **maketree** do not call **expose** at all. So overall, $m$ structure tree operations require at most $m-n < m$ expose calls.
Next, we prove that the \( m \) expose calls require \( \Theta(m \log n) \) splice calls. By Lemma 5.2, for any given expose, each of at most \( \lceil \log n \rceil \) splices will turn a light, dotted edge into a light, solid edge, increasing the number of heavy, solid edges by one, since any edge converted to dotted must be light.

We bound the number of heavy, solid edges created by noting that after the \( m \) operations, the tree has at most \( n-1 \) edges. Therefore, the number of heavy, solid edges created by the \( m \) expose calls is at most \( n-1 \) plus the number of heavy, solid edges destroyed by the operations.

Each call to expose destroys at most \( \lceil \log n \rceil +1 \) heavy, solid edges, possibly one for the first splice call, and at most one for each light, solid edge created. In addition to the heavy, solid edges destroyed by expose, the other operations can destroy additional heavy, solid edges. Maketree and unmark do not destroy any solid edge. link can destroy at most \( \lceil \log n \rceil \) additional heavy, solid edges. cut can destroy as many as \( \lceil \log n \rceil \) +1 heavy, solid edges. mark can destroy at the most one solid heavy edge.

When we combine these bounds, we see that at most \( O(m \lceil \log n \rceil) \) solid, heavy edges are destroyed by the expose calls, and \( O(m \lceil \log n \rceil) \) additional solid, heavy edges are destroyed by the other operations. So \( O(n-1+m \lceil \log n \rceil) \) heavy edges are created by the exposes. Adding the \( O(m \lceil \log n \rceil) \) splices that do create solid, light edges, the theorem is proved.

Taking this theorem into account and the analysis given in [ST83], we state the following theorem without proof:

**Theorem 5.2:** If we use self-adjusting binary trees to represent solid paths, a sequence of \( m \) structure tree operations including \( n \) maketree operations requires \( O(m \log n) \) time.

The cost of making any structure tree operation, amortised over a number of operations is \( \log n \).