1993

Tree compression scheme for displaying large trees

Vasu Srinivasan
The University of Montana

Let us know how access to this document benefits you.
Follow this and additional works at: https://scholarworks.umt.edu/etd

Recommended Citation
https://scholarworks.umt.edu/etd/5107

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
Maureen and Mike
MANSFIELD LIBRARY

Copying allowed as provided under provisions of the Fair Use Section of the U.S. COPYRIGHT LAW, 1976.
Any copying for commercial purposes or financial gain may be undertaken only with the author’s written consent.

University of Montana
A Tree Compression Scheme for Displaying Large Trees

Vasu Srinivasan
Department of Computer Science
University of Montana

Presented in partial fulfillment of the requirements for the degree of
Master of Computer Science
University of Montana

May 11, 1993

Approved by
Chairman, Board of Examiners
Dean, Graduate School

Date
The technical focus of the thesis is tree compression as applied to the display of
tree data structures in Graphical User Interfaces. Tree compression uses heuristics to
enable an entire tree to be visually manipulated within a window of limited size by
"compressing" certain groups of nodes. This approach is in contrast to the tradition
of "clipping" an image to fit into the display window, i.e., where the complete tree
is logically displayed on a virtual canvas of infinite size, but clipped to fit into a
finite window that serves as viewport into the infinite canvas. An important aspect
of compression is that it is parameterized with compression factors that are user
controlled and dynamically alterable. This compression scheme can be laid atop
any general purpose tree drawing algorithm. One application of this approach is in
the Graphical User Interface (GUI) for an object oriented design methodology, in
which large tree structures are used to represent class-instance hierarchies. While
the implementation presented here is on the X-Window/Motif/Unix platform, it is
should be applicable to any other platform as well.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Illustrations</td>
<td>iv</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>24</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>34</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>60</td>
</tr>
<tr>
<td>References</td>
<td>62</td>
</tr>
<tr>
<td>Appendix A: Data Structure</td>
<td>63</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

Figure 2.1 A basic tree 6
Figure 2.2 A compressed tree 7
Figure 2.3 Compression regions 11
Figure 3.1 Layout algorithm - 1 16
Figure 3.2 Layout algorithm - 2 19
Figure 3.3 Links in uncompressed tree 20
Figure 3.3 Links in compressed tree 21
Figure 4.1 Compression tree loop 31
Figure 5.1 The Application framework 35
Figure 5.2 Basic Operations 38
Figure 5.3 Adding a node 39
Figure 5.4 Deleting a node and its subtree 40
Figure 5.5 Tree after deleting a node 41
Figure 5.6 Tree prior to compression 43
Figure 5.7 Tree after compression 44
Figure 5.8 Browsing a compressed tree 45
Figure 5.9 Explicit and implicit compression 47
Figure 5.10 Browsing 48
Figure 5.11 Deleting from compressed tree 49
Figure 5.12 Removing compression 50
Figure 5.13 Reverting to uncompresed state 51
Figure 5.14 Level compression 52
Figure 5.15 Changing level compression 53
Figure 5.16 Changing level compression factors 54
Figure 5.17 After changing level compression factors 55
Figure 5.18 Changing sibling compression factors 56
Figure 5.19 Changing sibling compression factors \hspace{1cm} 57
Figure 5.20 New sibling configuration \hspace{1cm} 58
Chapter 1
Overview

1. PREMISE

1.1. THE PROBLEM

In order to display large trees on a relatively small display screen, the traditional
display method provides a finite sized window, or viewport, that slides across an
infinite canvas on which the tree is laid out. Thus, the user views the tree in parts by
scrolling through the tree. The viewport is of finite size, so although the whole tree is
logically displayed, the application has to resort to clipping to show parts of the tree
image at a time. The problem with this approach is that one tends to lose perspective
of the entire tree. When viewing a broad tree, for example, it is impossible to view
a portion of the tree to one side of the tree image without the other side of the tree
going out of the viewport. Further, when viewing a tree on a system that is heavily
loaded, the scrolling performance can be quite poor, i.e, there tends to be a distinct
time lag between the user action and the application response in shifting the viewport
window. This can be disconcerting to the user.

This thesis proposes an alternate way to display large trees, based on a theory of
tree compression which can generally fit an entire tree into a static viewport.
The basic idea is that each portion of the tree will be shown either in full and uncompressed detail, or compressed to save display space. There is always an uncompressed focus node that represents the focus of the user’s attention. The compression scheme allows the user to view a set of nodes around the focus node while compressing the tree at certain distances away from the focus node. The degree of compression is formally defined in terms of key parameters that are controlled by the user.

To explain how the compression scheme is applied, it is necessary to first review how trees are generally displayed. Tree drawing is based on certain rules, or principles, that address the display of the tree nodes. The display space can be seen as a simple two dimensional cartesian system with X and Y coordinate axes, and conventions that the origin is at the upper left corner of the display space, the X coordinate increases horizontally from left to right, and the Y coordinate increases vertically from top to bottom.

Tree drawing algorithms attempt to layout the nodes in an aesthetic fashion that satisfies as many display rules as possible. The most commonly used display rules are the following.

1. Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.

2. A parent should be centered over its offspring.

3. A tree and its mirror image should produce displays that are reflections of one another. Moreover, a subtree should be drawn the same way, regardless of where
it occurs in the tree.

Rules 1 and 2 are due to Wetherell and Shannon [1] and Rule 3 was introduced by Tilford and Reingold [2]. As noted earlier, these heuristics are typically used to arrange tree nodes in an aesthetic fashion on an infinite canvas. The typical approach does not attempt to fit the tree into a window which is limited in its dimensions by the physical size of the workstation screen or by other constraints.

The compression approach proposed here, on the other hand, specifically addresses the issue of fitting large sized trees into finite sized windows. Like other display approaches, the goals of compression are expressed in terms of display principles.

1. The root and the current focus node (the node currently of interest to the user) shall always be shown to give the user a perspective of the entire tree.

2. The entire tree shall be fitted into a viewport. Each node shall be shown either in complete or compressed form.

3. Every node shall be accessible through a sequence of one or more focus selections from any other node. Selection of an uncompressed node causes it to become the new focus node. Selection of a compressed region causes the uncompression of one or more compressed nodes from that region, with one of the newly uncompressed nodes becoming the new focus node.

4. The application shall be modeless; the user should be able to browse through the entire tree and be able to modify the tree without explicitly switching modes.
5. The user shall have clearly defined compression parameters. Default values shall be used in the absence of user defined specifications.

Rule #3 implies that the user be able to select a new focus node from any of the currently displayed nodes or from the set of compressed regions. By a series of such selections, the user should be able to traverse the entire tree.

Rule #4 underscores the point that the compression scheme is fully dynamic - it should support functions such as add, delete or modify on any part of the tree.

The following chapters explain in detail the theory and the implementation of the compression scheme.
Chapter 2

A simple theory of tree compression

As mentioned in the previous chapter, this approach to tree compression lies at a higher level than the basic tree layout. That is, it is applied after the basic layout has been determined. Thus, the underlying algorithm used to represent the layout of the tree can be any one of the several tree layout algorithms that have been published. In both the discussion and implementation, the algorithm developed by Sven Moen [3] is used. This algorithm is used to set up the tree and determine the initial and subsequent coordinate positions of the nodes. Depending on the size of the tree and the size of the viewing window, the tree will either fit completely or partially into the window. If the tree fits entirely into the window, there is no need for compression. On the other hand, if the tree does not fit into the window, then compression must be applied. Thus, the tree display at any point exists in one of two modes, real or compressed. In the real mode, there is a 1-1 mapping between the displayed nodes and the nodes in the underlying tree data structure. In the compressed mode, two or more nodes have been compressed into a single compressed region for display. Thus, a real display node represents a node in the underlying tree in its entirety, whereas a compressed region represents a collection of nodes in the tree. A compressed region can be represented visually by any appropriate icon. In our implementation we use the icon of a cloud.
Figure 2.1 A basic tree
Figure 2.2 A compressed tree
To illustrate, Figure 2.1 shows an example of a tree, the window that encloses it, and a simple application user interface framework. Node #7 is the current focus node, shown with a pointer adjacent to it. Figure 2.2 shows the tree after a node has been added as a successor to node #7. At this stage the entire tree size exceeds the window size, so compression is applied. Figure 2.2 shows the tree after compression has ensued. The root node and nodes #1, #2, #3, #6, #7, #8, #21, #23, #15, #16, and #13 are represented by real display nodes, whereas nodes #11, #12 and #19, #20 and #22, and #17, #14, and #18 are represented as four separate compressed regions. Each compressed region represents an area corresponding to a bounding rectangle of the actual nodes in that region.

2.1. Formal Definitions

The compressed regions result from two forms of compression. Level compression is the compression of levels of the tree at certain distances above and below the the level of the focus node. Sibling compression is the compression of nodes at certain distances to the sides of the focus node.

In all definitions below, we assume a tree T with levels 0 (root) to m, and with the focus node $F_n$ at level $n$ ($0 \leq n \leq m$). The compression factors and regions are based on certain other functions that are defined below.

1. $Level(n, T) = \{N_\alpha \mid N_\alpha \text{ is in } T \text{ at level } n\}$.

2. $PreOrderSeqLevel(n, T) = <N_{\alpha_1}, N_{\alpha_2}, ..., N_{\alpha_k}, ..., N_{\alpha_r}>$ where $\forall i$
a. \( N_{\alpha_i} \in \text{Level}(n, T) \),

b. \( N_{\alpha_i} \) appears in preorder traversal of \( T \) before \( N_{\alpha_{i+1}} \), and

c. \( N_{\alpha_k} = F_n \).

3. \( \text{LeftNodes}(n, F_n) = \{ N_{\alpha_i} \mid N_{\alpha_1} \leq N_{\alpha_i} < N_{\alpha_k} \text{ in PreOrderSeqLevel}(n, T) \} \).

4. \( \text{RightNodes}(n, F_n) = \{ N_{\alpha_i} \mid N_{\alpha_k} < N_{\alpha_i} \leq N_{\alpha_r} \text{ in PreOrderSeqLevel}(n, T) \} \).

**Definition 1:** The upper compression factor is a nonnegative integer, UCOMP, that defines a closed interval of tree levels \([1..(n - UCOMP - 1)]\).

**Definition 2:** The lower compression factor is a nonnegative integer, LCOMP, that defines a closed interval of tree levels \([(n + LCOMP + 1)..m] \).

**Definition 3:** All tree levels in the closed interval defined by UCOMP fall into a region called the upper compression region (ucr).

**Definition 4:** All tree levels in the closed interval defined by LCOMP fall into a region called the lower compression region (lcr).

**Definition 5:** The left sibling compression factor for a focus node \( F_n \) is a nonnegative integer LSCOMP that defines a closed subinterval of tree nodes \([N_{\alpha_1}..N_{\beta_1}]\) from \( \text{PreOrderSeqLevel}(n, T) \), where \( \beta_1 = k - LSCOMP \) and \( N_{\alpha_k} = F_n \).

**Definition 6:** The right sibling compression factor for a focus node \( F_n \) is a nonnegative integer RSCOMP that defines a closed subinterval of tree nodes \([N_{\beta_2}..N_{\alpha_r}]\) from \( \text{PreOrderSeqLevel}(n, T) \), where \( \beta_2 = k + RSCOMP \) and \( N_{\alpha_k} = F_n \).

**Definition 7:** The set of nodes in the closed subinterval defined by LSCOMP fall
into a region called the left sibling compression region (lscr).

**Definition 8:** The set of nodes in the closed subinterval defined by RSCOMP fall into a region called the right sibling compression region (rscr).

An alternative approach to sibling compression can be based on the functions \(NthPred\) and \(NthSucc\) which are defined as follows:

\[NthPred(i, F_n, T) = N_a\text{ where } N_a \text{ is the predecessor of } F_n \text{ at level } \alpha = n - i \text{ in } T.\]

\[NthSucc(i, F_n, T) = \{N_a | N_a \text{ is a successor of } F_n \text{ at level } \alpha = n + i\} \text{ in } T.\]

**Definition 9:** The left sibling compression region then comprises the set of nodes defined by

\[LeftNodes(n, F_n) - NthSucc(i, NthPred(i, F_n, T), T),\]

and the right sibling compression region then comprises the set of nodes defined by

\[RightNodes(n, F_n) - NthSucc(i, NthPred(i, F_n, T), T).\]

Level compression is the compression of the levels of the tree controlled by the UCOMP and LCOMP compression factors. Tree elements between the upper and lower compression regions are called the real nodes. Sibling compression is the compression of the real nodes to the left and right of the current focus node, controlled by the LSCOMP and RSCOMP compression factors. The compression regions are illustrated in Figure 2.3. Tree compression can be represented in terms of these regions and a four tuple of compression factors \(<UCOMP, LCOMP, LSCOMP, RSCOMP>\). Any compressed tree can be represented as a combination of these regions.
Figure 2.3 Compression regions
Using the trees in Figures 2.1 and 2.2 as an example, node #7 is the focus node and level 2 is the current focus level (the root is at level 0). The UCOMP and LCOMP values are each 1. This means that with a UCOMP value of 1, all levels that are more than 1 level above level 2 are compressed into one compressed region. But since Rule #1 states that the root is always shown, no upper compression takes place in this case and the upper compression region is empty. With a LCOMP value of 1, all levels below level 3 are compressed to form the lower compression region. Figure 2.2 is the compressed tree.

Definitions 5 through 8 represent a simple form of sibling compression. A LSCOMP of 2 will compress all nodes at a distance of 2 nodes away from the focus node where distance is measured in terms of nodes at the same tree level from the focus node. For example, in Figure 2.1, the focus node #7 has nodes #19, #20, #6, #8, #11, and #12 at the same level. With a LSCOMP of 2, all nodes at a distance of 2 or more nodes away from node #7 are compressed. Thus a left sibling compression is applied to all nodes to the left of node #6, i.e. #19 and #20. To the right of node #7 however, with an RSCOMP of 2, nodes #11 and #12 are compressed as shown in Figure 2.2.

Definition 9 provides a different view of “distance” for sibling compression, based upon predecessor/successor relationships. Note that with LSCOMP = 2 and RSCOMP = 2, applying Definition 9 to the tree in Figure 2.1 would yield no sibling compression, i.e. empty left and right sibling compression regions.
2.2. Maximum and Minimum compressions

For a given tree the maximum compression (i.e. minimum number of levels in the compressed tree) is dependent on the level of the focus node. Maximum level compression always occurs with $\text{UCOMP} = \text{LCOMP} = 0$. For a tree with $m + 1$ levels, and a focus node $F$ at level $n$, $0 \leq n \leq m$, there are 3 cases.

1. Case 1: $0 < n < m$.

All levels above the focus level and below the level 0 are compressed and all nodes below the focus level are compressed. There will be two real levels in the compressed tree, the root level and the focus level.

2. Case 2: $n = m$.

The focus node is at the lowest level in the tree. So, there will be two real levels in the compressed tree, root level and the focus level.

3. Case 3: $n = 0$.

The focus node is at the root. The number of real levels in the compressed tree will then be one, the root level.

These cases represent the maximum compression in terms of the levels in the tree. The number of actual nodes that are shown in the compressed tree is 1 if the focus node is the root, or $1 + K$ otherwise, where $K$ is the number of nodes falling within left and right sibling compression regions. Maximum sibling compression occurs when $\text{LSCOMP} = \text{RSCOMP} = 1$ (all nodes to the left and to the right of the focus node
are compressed). \( K \) then becomes 1 and the number of nodes actually shown is 2, the root node and the focus node.

The minimum level compression is when there is no compression of the levels in the tree at all. If \(( \text{UCOMP} > n)\), then \( \text{UCOMP} \) defaults to \( n \) and the upper compression region is empty, i.e. all levels above the focus level are shown. Similarly with \(( \text{LCOMP} > (m - n))\), \( \text{LCOMP} \) defaults to \( (m - n) \), the lower compression region is empty and all levels below the focus level are shown. The values of \( \text{UCOMP} \) and \( \text{LCOMP} \) can never be negative of course. Given the definition of the function \( \text{PreOrderTraversal} \), minimum sibling compression occurs for each of the definitions of sibling compression when \( \text{LSCOMP} \geq k - 1 \) and \( \text{RSCOMP} \geq r - k \).

These factors control the tree compression. Note that we distinguish between compression and “clipping” of nodes at any level by the left and right borders of the “viewport” on the screen, i.e. nodes that are clipped by the window edges and nodes that are compressed by values of \( \text{UCOMP} \), \( \text{LCOMP} \), \( \text{LSCOMP} \) and \( \text{RSCOMP} \).
Chapter 3

TREE LAYOUT

The compression scheme sits on top of a basic tree layout algorithm that lays out the initial tree on the infinite canvas. The layout algorithm used here was developed by Sven Moen[3]. The layout algorithm works by setting up contours around nodes and subtrees, and merging the individual contours to form a complete contour around the entire tree. A contour is formed as polygon using line segments that are hooked to form a chain which represents the entire contour. The algorithm uses two main functions, Layout and Branch, which accomplish the task of putting together and disassembling the contours.

3.1. Layout Function

The layout function is responsible for setting up the basic contours described above. It uses the three steps outlined above to form individual and complete contours. There are two basic types of contours: Leaf contours and Branch contours.

Leaf Contour: If a node is a leaf, the contour for the node is a simple rectangle that encloses it as shown in Figure 3.1a. Space is allocated for the border around each node. The left edge is ignored to allow for hooking to other contours. The other three edges are divided between two polylines where each polyline is a list of connected line segments.
Figure 3.1 Layout algorithm - 1
**Branch Contour:** If a node is a branch node, its contour is formed as a union of the contours in its subtree according to the following steps.

1. A contour is first formed for each subtree in the tree.

2. The contours of individual children are placed close to one another, and the relative positions of the children are noted.

3. Next, the union of the individual contours is formed by merging the contours of the children.

4. Finally, the offset between the parent and the children is computed and the parent's contour is then completed.

There are two basic sub functions that accomplish the task of setting up the contours and merging the individual contours as a union, Join and Merge. The Join function first sets up the parent contour equal to the leftmost child's contour. It then calls the Merge function which performs the actual union of the children contours. Finally, the union of the children contours is attached to the parent contour by attaching the line segments to the parent contour line segments (see Figure 3.1b).

### 3.2. Branch Function

The Branch function handles the case of disassembling contours whenever new nodes are added or deleted from the existing tree structure. Since the addition or deletion of a node will affect the contours above the node, the affected contours
are first disassembled, recomputed, and then reassembled. For example, to add a subtree, the contour of the subtree is calculated, and the parent node to which it is to be attached is specified. A function Unzip disassembles the affected contours. The contour for the new subtree is added, then another function does the Zip operation to reassemble the contours. Figure 3.2, a and b, illustrates the major parts of the layout algorithm. Since the application seeks to make the compression algorithm independent of the underlying layout algorithm, the layout algorithm is not presented in any greater detail here.

3.3. Data Structure

Once the initial tree has been laid out by the layout algorithm, a compressed tree structure is represented by the data structure explained here. The compression scheme uses the tree laid out by the layout algorithm and held by the data structure to perform compression on the tree structure. The details of the data structure that holds the tree in memory are presented in Appendix A. Each node has a link to a child node and a sibling node. Thus the elements in a row of siblings are connected to their parent via the parent’s child node. The elements in a row of siblings are also connected to one another as a doubly linked list via the sibling and prev links illustrated in Figures 3.3 and 3.4. In each row of nodes at a particular level, a set of sibling nodes is connected to an adjacent set of sibling nodes by the nbor and prev_nbor links.

Each node can be one of two types: real or compressed (virtual). Each real node
Figure 3.2 Layout algorithm-2
Figure 3.3 Links in uncompressed tree
Figure 3.4 Links in compressed tree
represents a single node in the data structure and on the display. However, because a collection of real nodes can be collapsed or compressed by the compression scheme into a single compressed node, the compressed node in the data structure carries some additional information that allows it to represent the collection of nodes.

Whenever a collection of nodes is compressed into a node, the resultant node must still retain its links in a level to the adjacent nodes that have not been compressed. Two additional links are provided, \( vnbor \) and \( prev\_vnbor \), to link to the nearest real nodes. The \( nbor \) and \( prev \) links thus serve to interconnect the nodes in a tree to form a lattice. The \textit{region} structure allows a compressed node to represent the bounding region for collection of nodes that it represents. Rather than holding links to all the levels and nodes that the region represents, the \textit{region} structure merely holds the coordinates of the bounding rectangle of the compressed nodes. This is particularly useful in the selection of next focus node using a simple geometrical midpoint algorithm.

As soon as it is formed, the tree is converted into a lattice, and every node added is inserted in place in the lattice. The \textit{level\_list} and \textit{vlevel\_list} structures serve to hold the lattice. Each is a list of pointers to a row (level) of the tree. The slots in the \textit{level\_list} structure each point to a row in the tree. Thus by traversing down and across we can travel across the entire lattice, both up and down. When the tree has been compressed, some of the levels will be compressed to a single level (and some of the nodes in any level could be compressed to a single node). The \textit{vlevel\_list} holds the
compressed lattice. Thus conceptually, the compressed tree can be seen as an overlay of the compressed lattice over the uncompressed lattice.

The other parts of the \textit{TREENODE} structure shown in Appendix A are used in the tree layout algorithm to hold the contours of the subtrees used by the tree layout algorithm.
Chapter 4

Implementation of compression

The entire compression application can be thought of as a pipeline that is divided into three major stages: Tree Manager stage, Tree Compression stage and Tree Display stage.

4.1. Tree Manager stage

In the tree management stage falls all actions that change the tree data structure. These include:

1. reading in the initial tree structure,

2. choosing the next focus node, and

3. executing Add/Delete operations.

Of these three actions the first and the third affect the data structures that hold the uncompressed tree and the compressed tree. The second action, choosing the focus node, does not change the data structure that holds the compressed tree but is central to the actions of adding/deleting subtrees; hence it is included in Tree Management.
When the application\textsuperscript{1} tool is started, it reads the initial structure of the tree from an external file. The tree is represented as a preorder list of nodes. A rudimentary parser processes the list description and builds the tree in memory using the data structure outlined previously. That is, as the tree is read in, the rows in every level are interconnected, to form a lattice. Thus the tree is represented conceptually at two levels: the regular tree structure with the familiar parent-child connections, and as a lattice of nodes. When the application is terminated, the tree is written back onto the external file in the same external format.

The operation of choosing a focus node is central to all user actions. All Add/Delete operations and the tree display are governed by the node that is the \textit{Current Focus Node}. The lattice structure is suited for quick traversal in either coordinate direction in response to changes in the focus node. That is, a common user action is to change the focus node, in order to move around (i.e browse) the tree.

There are two cases that arise when the user selects a new focus node.

1. The newly selected node is a real node. In this case, because there is a one to one correspondence between the node on display and the node in the data structure, the selected node becomes the next focus node.

2. The user selects a compressed region. In this case, a single node must be selected from amongst the collection of nodes that this compressed region represents.

Any compressed region can be viewed as a bounding region of a collection of

\textsuperscript{1} The implementation of the compression algorithm is referred to here as the “application".
nodes in a two dimensional coordinate space. Intuitively, any suitable heuristic/algorithm could be used to choose the next focus node. A simple geometric mid-point approach is used here to simplify the processing. Since each compressed region is a set of nodes in 2-D space, hooked together as a lattice of nodes, the central node in the middle row of that lattice is chosen. For example, if the user selects a compressed region that represents an upper compression region which has 3 levels, the middle node of level 2 is chosen as the next focus node. All subsequent tree operations are with reference to this node.

The *Add Node* operation adds a node as a child of the current focus node. It affects the data structure holding the tree and may trigger a compression if the resultant size of the tree exceeds the viewport size.

The *Delete Node* operation deletes the subtree headed by the current focus node (including the focus node). This operation reduces the size of the tree; the Tree Display Manager will automatically redisplay the resultant tree. If, after the subtree has been deleted, the tree fits into the viewport, then there is no compression and every node is shown. On the other hand, if the tree is still too large for the viewport, then the compression of the tree is recomputed, and the resultant compressed tree is displayed.

Since the focus node is deleted as part of the deleted subtree, the next focus node is undefined at this point. However, a particular choice for the next focus node in this case can easily be incorporated as either a heuristic or a configurable parameter.
The tree management system includes a rudimentary form of two-phase commit for Add/Delete operations. When the user performs either of these operations, the affected nodes are displayed in a suitable color to indicate their different status. The actual operation is only carried out when the user chooses to *Commit the operation*. All nodes that are added, and all parents of subtrees that are marked to be deleted are kept in a separate buffer. When the action is committed, the application performs the Add/Delete operation on the entire buffer. Additional system level primitives that ensure that the sequence of tree manipulations is “atomic” could be easily added.

4.2. Tree Compression Stage

The compression stage is at the heart of the application pipeline and is triggered as a response to any user action that changes the tree’s structure or changes the visual aspect of the tree (the choice of the next focus node). The compression stage concerns itself primarily with fitting the tree structure into the viewport. The *level.list* data structure holds the lattice that represents the tree, but in an uncompressed form. As the list is traversed, the compression algorithm first determines which of the *compression regions* each row of tree nodes falls into. If the row falls into the upper or lower compression region, the region enclosing the current row is appended to the corresponding compression region. By just holding the set of compressed nodes as a region we avoid having to hold lists of nodes that span multiple levels in any complex data structure. If the row falls into the *real region*, the compression
algorithm determines which kind of compression to perform. If the row is the *focus level*, it performs *explicit compression*. This is the compression that is determined by the user as per the sibling compression parameters.

Starting with the current focus node, all nodes that are to be compressed to the right and to the left of the focus node as determined by the parameters, RSCOMP and LSCOMP, are encapsulated into a newly created compressed region. The function `compress_focus_level` implements this process and returns the node to the extreme left of the tree row, which might be in a compressed region or a real node. Thus effectively, a row of tree nodes is *overlayed* over the existing tree row. The real nodes are used as such and the newly created compressed regions are hooked to the adjacent nodes in that row via the links described above. The pointer to the first node in the list is returned and hooked into a new slot in the `vlevel_list` that holds the compressed tree lattice. Rows of tree nodes that are not affected by the compression may still be clipped by the viewport on either side, and added to another slot in the `vlevel_list` structure. When the compression is complete, all the levels in the tree have been compressed into a compressed lattice with each row in the lattice frame further compressed as needed.

4.3. Tree Display Stage

The tree display stage is the final stage in the application pipeline. After the compression algorithm has done its work, the tree is displayed in its compressed form.
by the display manager routines of the application. The main work of this stage is to
construct the final display holding the real nodes, icons for the compressed regions in
the viewport, and draw the connecting lines.

Each real node's original position is maintained for as long as possible. The upper
and lower compressed regions are each displayed in the center of their respective
regions. The other compressed regions are displayed in a position that fits into the
viewport.

4.4. The Compression Algorithm

The compression algorithm is made up of three simple functions: get_focus_node,
compute_compression_parms and compress_tree.

**get_focus_node**: The function is responsible for selecting the next focus node from
the tree data structure in response to the user's selection. There are two cases here:

1. User selects a real node in the display. This is the simplest case. Since there
is a one to one correspondence between the node on the display and the node
in the underlying data structure, the node that was selected becomes the next
focus node.

2. User selects an compressed region. Here again there are two sub cases. If the
selection were made from either the upper or lower compression regions,
then a simple geometric midpoint selection is made. Since the compressed
regions may span one or more levels, first the middle level is chosen. Next the
middle node in that middle level (row) is selected as the next focus node.

If the selection is made from any one of the other compressed regions (set of nodes compressed either due to sibling compression or implicit clipping by the window border) a simple mid-node selection from that set of nodes is made.

**compute_compression_parms:** The function simply takes the current compression parameters set by the user and calculates the size (in terms of the levels) of the upper compression region, the lower compression region, the upper real region, the lower real region. These are of course dependent on the focus level, which is known once the next focus node is set by the `get_focus_node` function. It is also responsible for setting sensible default values for the compression region sizes.

**compress_tree:** The compress tree function is a simple loop that loops through each row in the `level_list` lattice and fills the slots in the `vlevel_list` lattice structure.

There are several simple functions that are referred to in the compress tree loop presented in Figure 4.1.

1. **compress_upper_region** is a function that abstracts all levels in upper compression region to a single compressed region and returns this compressed region as its result.

2. **compress_lower_region** is a function that does the same for the lower compression region.

3. **compress_real_level** is a function that abstracts all nodes clipped by the left
Data Structures:

vp - tree node
tp - temporary list pointer.
level_list - list that holds uncompressed tree mesh
vlevel_list - list that holds compressed tree mesh

vp = compress_upper_region();
append_hdr(vp);
Set tp to point to the first level in the upper real region.
while (tp ≤ last level in lower real region)
begin
  if (level = uar)
    vp = compress_real_level();
  if (level = focus level)
    vp = compress_focus_level();
  append_hdr(vp);
end
vp = compress_lower_region();
append_hdr(vp);

Figure 4.1 Compress tree loop
and right window borders into compressed regions. It attaches these compressed regions to the existing nodes by the vnbor links and returns the left most node in that row.

4. compress_focus_level is a function that starts at the focus node in that level, moves LCOMP nodes to the right, and compresses all nodes to a single compressed region. It then attaches that node to the rightmost uncompressed region by the vnbor link, does the same to the left of the focus node, and returns the leftmost node.

5. append_hdr is a function that opens a new slot in the vlevel_list list. It then attaches the pointer to a new row in the compressed mesh.

The overall complexity of the Compression algorithm is derived from the major algorithm components.

1. get_focus_node: There are three cases here.

(a) Selection of real node: Since there is one to one correspondence between the node on display and the node in the underlying data structure, the selection time is a constant.

(b) Selection of upper or lower or compressed regions: The region may span one or more levels. Assuming \( r \) rows in the compressed region, \( r/2 \) rows need to be traversed. If the \( r/2 \)th row has a \( n \) nodes, \( n/2 \) nodes are traversed. So the selection is of the order \( O(r/2 + n/2) \) or \( O(n) \).
(c) Selection in real region: With \( n \) nodes in each set of compressed nodes, the order of selection is \( O(n/2) \) or \( O(n) \).

Thus, the complexity of the selection function is \( O(n) \).

2. Compression loop

(a) Compression of upper and lower compression regions: Since the size of the regions have already been computed, computing the size of this region is a constant.

(b) Compression of real and focus levels: Since the compression is done while traversing the row linearly, the worst case is of order \( O(n) \) where \( n \) is the length of the row.

(c) \texttt{append_hdr}: This function merely appends the pointer to the compressed row to the end of \texttt{vlevel_list}. Hence, it requires constant time.

In the worst case, the upper and lower compression regions and the left and right sibling regions are all empty and the entire tree falls into the real region of nodes, i.e all the tree nodes are real and are displayed. In this case, all nodes in the tree are traversed in both the compression loop and while displaying the nodes on the viewport. Thus, in the worst case, the order of complexity is \( O(n) \).

Since the compression algorithm calculates the compression in a separate pass after the layout algorithm has done its work, the compression algorithm does not add to the magnitude of the complexity.
Chapter 5

Example Application

This chapter brings together all the concepts described in the preceding chapters by means of a tutorial that walks the reader through the main features of the application. The application is programmed using the X-Window/Motif toolkit. The user interface provides the common look and feel of a Motif application.

5.1. The Application GUI framework

When the application is started from the system command line, the user is presented with a typical GUI framework as shown in the Figure 5.1. The top portion of the application screen is the main menu bar with the choices labeled. The right half is devoted to the application canvas where the tree is displayed. The bottom right window has a Message window that displays the application messages in a scrolled text window region.

To the left of the application canvas is the Command space of the toolkit. Here are three command buttons labeled Add, Delete and Commit that represent the major user actions that manipulate the tree.

Below the command area is an open space provides an area for visual feedback. A compression bar shows the current value of the tree compression on a scale of 1 to 100. This is given by \((\text{number of compressed nodes} / \text{total number of nodes in tree}) \times 100\).
Figure 5.1 The Application framework
Other information like the tree size, and values of the compression factors could also be shown here.

To start the application the user types "twd" at the command line. When the application has started, it presents the familiar Motif Graphical User Interface.

Externally, a tree is treated as a simple file. The tree data structure is read in when the application's File | Open command is executed, which allows the user to select a tree structure stored in an external file. As the tree is read in, it is converted into the tree data structure in memory.

When a tree is opened, it is displayed on the display canvas. Figure 5.2 shows a tree that has been opened. Each displayed node in the tree is actually implemented as a Motif PushButton allowing extensive forms of node/user interaction. The root of the tree is displayed with the label "Root". All other nodes are labeled with an unique integer node id. When the user selects a particular node with the pointer device, the node button assumes a depressed look to show the result of the selection, i.e that the selected node is the focus node for any subsequent action.

The commands Save and SaveAs save the tree structure to the same or to a different file. The command Close saves and closes the current tree, reinitialises the displayed canvas, and allows the user to open another tree.

The Quit button is used to exit the application. The application will ask the user to confirm the quit command by a confirmation dialog. Before quitting, the application will save all changes to the tree being manipulated.
5.2. Basic Operations

Figure 5.2 shows a tree that has been opened. If now node# 1 is selected as the focus node and the Add button selected in the command area to the left of the canvas, the resultant tree is shown in Figure 5.3. Node#19 is shown in a different color to emphasise its status as a newly added node. The commit action serves to finally commit the user actions to the application. For example, if after adding a new node, the commit button is selected, the newly added node reverts to the color of a normal node.

The Delete operation deletes the entire subtree of which the current focus node is the parent. An example clarifies. If node# 7 is chosen from the tree in Figure 5.3 and the Delete button is clicked upon, a warning dialog pops up on the application screen which cautions the user that the entire subtree will be deleted and allows the user provision to cancel this operation. Upon confirmation, the subtree appears grayed out. The nodes are desensitized - the user cannot interact with them. The nodes however remain on the screen. When the commit button is pressed, the nodes get deleted both from the display and from the application, Figures 5.4 and 5.5.

5.3. Browsing and Compression

This section walks the user through a series of steps that illustrate the tree compression and the browsing mode of the application. It shows how the user can control the compression by means of editing the compression parameters and also demon-
Figure 5.2 Basic Operations
Figure 5.3 Adding a node
Figure 5.4 Deleting a node and its subtree
Figure 5.5 Tree after deleting a node
strates how the application automatically switches between the compressed and uncompressed states, depending on the size of the tree.

Figure 5.6 shows the application with a tree in the application canvas at some state. Node #7 is the current focus node. The tree is shown just before a node is added to the current focus node. When a node is added to node #7, the resultant tree is shown in Figure 5.7. The addition of node #23 to node #7 expands the tree beyond the confines of the tree viewport and compression ensues. The tree in Figure 5.6 has been compressed into the tree shown in Figure 5.7.

The current compression factors are UCOMP = 1, LCOMP = 1, LSCOMP = RSCOMP = 2. Thus with focus node #7, all nodes at level #4 and below (all levels LCOMP levels away from level #2, the current focus node level), have been compressed and are represented by the cloud icon. Since the level two levels above that of node #7 is the root level, no compression takes place in the levels above level 2.

Figure 5.8 shows another tree configuration. Here the focus node has been selected from the lower abstract section represented by the cloud icon in Figure 5.7. The application uses a particular selection algorithm to make one of the nodes in the lower abstract section the next focus node, node #14. The application uses the same compression factors to compress a different section of the tree.

Figures 5.6 through 5.8 have shown an example of browsing through the tree by selecting different nodes as the next focus node and allowing the application to
Figure 5.6 Tree prior to compression
Figure 5.7 Tree after compression
Figure 5.8 Browsing a compressed tree
automatically display a window of real nodes around the current focus node while compressing away sections of the tree away from the focus node.

If now node #21 is selected as the next focus node, the tree of Figure 5.9 results. This tree shows nodes #15, #16, and #13 all compressed by the right sibling compression factor of 2. This is an example of explicit compression. Above node #22 at level 2, all nodes to the left of node #19 have been compressed too. This results from the clipping by the left border of the canvas (viewport) window and is an example of implicit compression. If the next focus node is selected from the upper abstract window of tree in Figure 5.9, the application selects the node #2 as the next focus node as indicated in Figure 5.10.

When a subtree is removed from the tree and the resultant tree fits into the viewport, all compression is automatically removed.

The tree in Figure 5.11 is the result of choosing node #7 as the next focus node from the tree in Figure 5.10, preparatory to deleting the subtree beneath node #7. If now the Delete operation is performed, Figure 5.12 shows the subtree beneath node #7 grayed out. If the delete operation is committed now, the tree in Figure 5.13 is displayed. The removal of the subtree, brings the tree down to a size that allows it to be displayed in its entirety within the viewport.

5.4. Changing Compression Parameters

The next two sections show examples of browsing through a tree while changing the compression factors and seeing how they affect the displayed tree. This section
Figure 5.9 Explicit and implicit compression
Figure 5.10 Browsing
Figure 5.11 Deleting from compressed tree
Figure 5.12 Removing compression
Figure 5.13 Reverting to uncompressed state
Figure 5.14 Level compression
Figure 5.15 Changing level compression
Figure 5.16 Changing level compression factors
Figure 5.17 After changing level compression factors
Figure 5.18 Changing sibling compression factors
Figure 5.19 Changing sibling compression factors
Figure 5.20 New sibling configuration
illustrates with examples the concept of changing the Upper and Lower compression factors. Figure 5.14 shows a tree prior to compression. If node #32 is chosen as the focus node and a node(#33) added to it, the resulting compressed tree is Figure 5.15. The compression factors at this stage are $UCOMP = 1$, $LCOMP = 1$, $LSCOMP = 2$ and $RSCOMP = 2$. If the next focus node is chosen from the upper cloud icon in Figure 5.15, the tree in Figure 5.16 results. If now the Edit choice in the main menu is chosen, and theParms subchoice from the drop-down menu is selected, a Parameter dialog box appears. The compression factors are changed so that $UCOMP = 3$, $LCOMP = 2$, $LSCOMP = 2$ and $RSCOMP = 2$. If now node #26 chosen from the tree depicted in Figure 5.16, the tree with the new compression is displayed as in Figure 5.17. All nodes at level 6 and below are compressed because of the lower compression of 2. The parameters get adjusted according to the level of the current focus node #26 so that $UCOMP = 2$.

Again, we pick up the tree at some compressed stage, as in Figure 5.18, after node #22 has been added to the current focus node #8. The compression factors here are $UCOMP = 1$, $LCOMP = 1$, $LSCOMP = 2$, $RSCOMP = 2$. If now the parameters are edited to make $LSCOMP = 1$ and $RSCOMP = 3$, and node #21 chosen as the next focus node, the tree in Figure 5.20 results. With this change, all nodes to the left of node #21 have been compressed, whereas there is no compression to the right of node #21. This is because there are no nodes that are at a node distance of 3 or more (the value of $RSCOMP$) to the right of the current focus node (#21).
Chapter 6

Conclusion and Directions for further Research

The preceding chapters have presented the idea of a tree compression scheme, the theory behind it and an implementation to support the theory. The theory provides a basis for future work on a more detailed and specific formulation of the compression parameters. The implementation is an extension of prior work on Dynamic Tree Display techniques. It extends the concepts of the heuristics to display trees to include rules for compressing portions of the tree on the display. The implementation provides a test bench for the theory and would help in getting user feedback on our approach of compressing portions of large trees in the display versus scrolling through them. Methods to measure the compression and its performance on large trees could be introduced. Further additions could be made to the parametrization of the compression factors. In particular, the concept of sibling compression could be extended to include levels above the focus level. Other algorithms could be easily introduced for selecting the focus node. Also, the compression of the nodes could be calculated in the first pass along with the layout algorithm.

Although the application is designed to deal with trees, it could be extended to include general purpose graphs. Such an extension would be extremely useful as a visualisation tool for large networks which can be treated as graphs. The tool could also be extended to work across a network of workstations. This would enable users
sitting at different workstations to manipulate the same data structure. Primitives would have to be built into the system to ensure the integrity of the data structure objects across the network according to some suitable protocol.

Certain visual clues could be added to aid the user in understanding and manipulating the tree compression. It would be useful to give the user visual feedback on the number of nodes in any compressed region and the parental relationships of the nodes within that region. The user, by selecting a compressed region, could open a new window that showed the nodes in that compressed region in greater detail.
REFERENCES:


APPENDIX A

---------

Tree Node Data Structure

***************

/* the rectangular region in the real tree
 * that each abstract node represents */
typedef struct region_struct {
    int tlx, tly, /* top left and bottom right coordinates */
        brx, bry;
    struct region_struct *next; /* not used now; might have list of abstract
        regions later */
} REGION, *REGIONPTR;

/* data structure for the tree contours */
typedef struct line {
    short dx, dy;
    struct line *link;
} *POLYLINE;

struct polygon {
    struct {
        POLYLINE head, tail;
    } lower, upper;
};

typedef struct tnode {
    char label[5];
    struct tnode *parent, *child,
        *sibling, *prev; /* for first child, prev points
        to parent; for subsequent
        children prev points to
        previous child in sibling
        rail */
    struct tnode *prev_nbor, *nbor; /* for virtual connections */
    int width, height, border;
    int level;
    struct {
        short px,
        x, py, y;
    } pos, offset;
    struct polygon contour;
    Widget NodeButton;
    int DeleteFlag; /* delete node marker */
    NODETYPE node_type;
    REGIONPTR regionp; /* node region */
} TREENODE, *TREE;